Information Flows to Support Software Developers in Using Security APIs

vorgelegt von
M.Sc.
Peter Leo Gorski
ORCID: 0000-0003-0391-4054

an der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender: Prof. Dr. Sergio Lucia
Gutachter: Prof. Dr.-Ing. Sebastian Möller
Gutachter: Prof. Dr.-Ing. Luigi Lo Iacono
Gutachterin: Prof. Dr. Simone Fischer-Hübner

Tag der wissenschaftlichen Aussprache: 3. Juli 2020

Berlin 2021

To my twin daughters Marlene and Johanna
– for a better future.

Abstract

At the end of 2019, about 4.1 billion people on earth were using the internet. Because people entrust their most intimate and private data to their devices, the European legislation has declared the protection of natural persons in relation to the processing of personal data as a fundamental right. In 2018 23 million people worldwide, having the responsibility of implementing data security and privacy, were developing software.

However, the implementation of data and application security is a challenge, as evidenced by over 41 thousand documented security incidents in 2019. Probably the most basic, powerful, and frequently used tools software developers work with are Application Programming Interfaces (APIs). Security APIs are essential tools to bring data and application security into software products. However, research results have revealed that usability problems of security APIs lead to insecure API use during development. Basic security requirements such as securely stored passwords, encrypted files or secure network connections can become an error-prone challenge and in consequence lead to unreliable or missing security and privacy. Because software developers hold a key position in the development processes of software, not properly operating security tools pose a risk to all people using software. However, little is known about the requirements of developers to address the problem and improve the usability of security APIs.

This thesis is one of the first to examine the usability of security APIs. To this end, the author examines to what extent information flows can support software developers in using security APIs to implement secure software by conducting empirical studies with software developers.

This thesis has contributed fundamental results that can be used in future work to identify and improve important information flows in software development. The studies have clearly shown that developer-tailored information flows with adapted security-relevant content have a positive influence on the correct implementation of security. However, the results have also led to the conclusion that API producers need to pay special attention to the channels through which they direct information flows to API users and how the information is designed to be useful for them. In many cases, it is not enough to provide security-relevant information via the documentation only. Here, proactive methods like the API security advice proposed by this thesis achieve

significantly better results in terms of findability and actionable support. To further increase the effectiveness of the API security advice, this thesis developed a cryptographic API warning design for the terminal by adopting a participatory design approach with experienced software developers. However, it also became clear that a single information flow can only support up to a certain extent. As observed from two studies conducted in complex API environments in web development, multiple complementary information flows have to meet the extensive information needs of developers to be able to develop secure software. Some evaluated new approaches provided promising insights towards more API consumer-focused documentation designs as a complement to API warnings.

Zusammenfassung

Ende 2019 nutzten rund 4,1 Milliarden Menschen auf der Erde das Internet. Da die Menschen ihre intimsten und privatesten Daten ihren Geräten anvertrauen, hat die europäische Gesetzgebung den Schutz natürlicher Personen bei der Verarbeitung personenbezogener Daten zu einem Grundrecht erklärt. Im Jahr 2018 haben weltweit 23 Millionen Menschen Software entwickelt. Diese tragen die Verantwortung, die Datensicherheit und den Datenschutz zu gewährleisten.

Die Umsetzung von Daten- und Anwendungssicherheit ist allerdings eine Herausforderung, wie über 41 Tausend dokumentierte Sicherheitsvorfälle im Jahr 2019 belegen. Die wohl grundlegendsten, leistungsfähigsten und am häufigsten verwendeten Werkzeuge, mit denen Software-Entwickler arbeiten, sind Application Programming Interfaces (APIs). Security-APIs sind elementare Werkzeuge, um Daten- und Anwendungssicherheit in Softwareprodukte zu integrieren. Forschungsergebnisse haben jedoch gezeigt, dass Usability Probleme von Security-APIs zu einer unsicheren Nutzung bei der Entwicklung führen. Grundlegende Sicherheitsanforderungen, wie sicher gespeicherte Passwörter, verschlüsselte Dateien oder sichere Netzwerkverbindungen, können dadurch zu einer fehleranfälligen Herausforderung werden und in der Konsequenz zu unzuverlässiger oder fehlender Sicherheit und Privatsphäre führen. Da Softwareentwickler eine Schlüsselrolle in den Entwicklungsprozessen von Software spielen, stellen nicht ordnungsgemäß funktionierende Sicherheitswerkzeuge ein Risiko für alle Personen dar, die Software verwenden. Es ist jedoch nur wenig über die existierenden Anforderungen von Entwicklern bekannt, um dieses Problem anzugehen und die Nutzbarkeit von Security-APIs zu verbessern.

Diese Arbeit ist eine der ersten, die zu der Gebrauchstauglichkeit von Security-APIs forscht. Zu diesem Zweck untersucht der Autor, inwieweit Informationsflüsse Softwareentwickler bei der Nutzung von Security-APIs zur Implementierung sicherer Software unterstützen können, indem er empirische Studien mit Softwareentwicklern durchführt.

Diese Thesis hat grundlegende Ergebnisse erbracht, die in zukünftigen Arbeiten zur Identifizierung und Verbesserung wichtiger Informationsflüsse in der Softwareentwicklung genutzt werden können. Die Studien haben deutlich gezeigt, dass auf den Entwickler zugeschnittene Informationsflüsse mit abgestimmten sicherheitsrelevanten Inhalten einen

positiven Einfluss auf die korrekte Implementierung von Sicherheit haben. Die Ergebnisse haben jedoch auch zu der Schlussfolgerung geführt, dass API-Produzenten besonders darauf achten müssen, über welche Kanäle sie Informationsflüsse zu API-Benutzern leiten und wie die Informationen gestaltet sein müssen, damit sie für die Zielgruppe gebrauchstauglich sind. In vielen Fällen reicht es nicht aus, sicherheitsrelevante Informationen nur über die Dokumentation zur Verfügung zu stellen. Hier erzielen pro-aktive Methoden, wie die in dieser Arbeit vorgeschlagenen API-Sicherheitsempfehlungen, deutlich bessere Ergebnisse in Bezug auf die Auffindbarkeit und eine direkt anwendbare Unterstützung. Um die Wirksamkeit von API-Sicherheitsempfehlungen weiter zu verbessern, wurde in dieser Arbeit ein Design für kryptographische API-Warnungen in der Konsole entwickelt. Dazu wurde ein partizipativer Designansatz mit erfahrenen Softwareentwicklern gewählt. Es wurde jedoch auch deutlich, dass ein einzelner Informationsfluss nur bis zu einem gewissen Grad Unterstützung leisten kann. In zwei Studien, die in komplexen API-Umgebungen für die Webentwicklung durchgeführt wurden, konnte beobachtet werden, dass mehrere sich ergänzende Informationsflüsse den umfangreichen Informationsbedarf der Entwickler decken müssen, damit diese in die Lage versetzt werden, sichere Software zu entwickeln. Ergänzend zu API-Warnungen, lieferten einige der neuen evaluierten Ansätze vielversprechende Erkenntnisse über mehr benutzerorientierte API Dokumentationsdesigns.

Publications

Parts of this thesis have been previously published in the following peer-reviewed workshop and conference publications:

I. **"Towards the Usability Evaluation of Security APIs."** Peter Leo Gorski and Luigi Lo Iacono. *10th International Symposium on Human Aspects of Information Security and Assurance (HAISA)*. University of Plymouth. 2016, pp. 252 – 265. (https://www.cscan.org/?page=openaccess&eid=17&id=287)

I was the main responsible for the paper's work. Luigi Lo Iacono provided feedback during the literature research and model development, and reviewed the manuscript.

II. **"I Do and I Understand. Not Yet True for Security APIs. So Sad."** Luigi Lo Iacono and Peter Leo Gorski. *2nd European Workshop on Usable Security (EuroUSEC)*. Internet Society. 2017, pp. 1 – 11. (http://dx.doi.org/10.14722/eurousec.2017.23015)

I was involved in all the work for the paper. I have designed and conducted the online-survey and analyzed the results. Luigi Lo Iacono and I have worked on the security API classification, framework analysis, and writing the paper.

III. **"Developers Deserve Security Warnings, Too: On the Effect of Integrated Security Advice on Cryptographic API Misuse."** Peter Leo Gorski, Luigi Lo Iacono, Dominik Wermke, Christian Stransky, Sebastian Möller, Yasemin Acar, and Sascha Fahl. *14th Symposium on Usable Privacy and Security (SOUPS)*. USENIX Association. 2018, pp. 265–281. (https://www.usenix.org/conference/soups2018/presentation/gorski)

I was the main responsible for the paper. I have designed, implemented, and conducted the online study. Co-authors provided feedback during the study design phase. Co-authors and I have recruited participants, contributed to the data analysis, writing, and reviewing the paper.

IV. **"Warn if Secure or How to Deal with Security by Default in Software Development?"** Peter Leo Gorski, Luigi Lo Iacono, Stephan Wiefling and Sebastian Möller. *12th International Symposium on Human Aspects of Information Security and Assurance (HAISA)*. University of Plymouth. 2018, pp. 170–190. (`https://www.cscan.org/?page=openaccess&eid=20&id=388`)

I was the main responsible for the paper. I have designed and implemented the laboratory study. I conducted most of the experiments and analyzed the results. Co-authors provided feedback on the study design, helped to conduct experiments, implementing an annotation software, writing, and reviewing the paper.

V. **"Listen to Developers! A Participatory Design Study on Security Warnings for Cryptographic APIs."** Peter Leo Gorski, Yasemin Acar, Luigi Lo Iacono, and Sascha Fahl. *2020 CHI Conference on Human Factors in Computing Systems (CHI)*. Association for Computing Machinery. 2020, pp. 1–13 (`https://doi.org/10.1145/3313831.3376142`)

I was the main responsible for the paper. I have designed, implemented, and conducted the focus groups, and analyzed the results. Co-authors provided feedback on the study design, discussed coding results, and helped to review the manuscript.

The following paper was under submission to a peer-reviewed conference when this thesis was submitted for review:

VI. **"'I just looked for the solution!' - On integrating Security-Relevant Information in Non-Security API Documentation to Support Secure Coding Practices."** Peter Leo Gorski, Sebastian Möller, Stephan Wiefling, and Luigi Lo Iacono. 2020.

I was the main responsible for the paper. I have designed, implemented, and conducted the laboratory study, and analyzed the results. Co-authors provided feedback on the study design, helped writing, and to review the manuscript.

Published work in the field of Usable Security that is not part of this thesis:

"Signalling over privileged mobile applications using passive security indicators." Luigi Lo Iacono, Peter Leo Gorski, Josephine Grosse, and Nils Gruschka. *Journal of Information Security and Applications*, Volume 34, Part 1. 2017, pp. 27–33. (https://doi.org/10.1016/j.jisa.2016.11.006)

"Consolidating Principles and Patterns for Human-centred Usable Security Research and Development." Luigi Lo Iacono, Matthew Smith, Emanuel von Zezschwitz, Peter Leo Gorski, and Peter Nehren. *The 3rd European Workshop on Usable Security (EuroUSEC)*. Internet Society. 2018. (http://dx.doi.org/10.14722/eurousec.2018.23010)

"On providing systematized access to consolidated principles, guidelines and patterns for usable security research and development." Peter Leo Gorski, Emanuel von Zezschwitz, Luigi Lo Iacono, and Matthew Smith. *Oxford Journal of Cybersecurity*, Volume 5, Issue 1, tyz014, 2019. (http://dx.doi.org/10.1093/cybsec/tyz014)

Published work in the field of Security that is not part of this thesis:

"Service Security Revisited." Peter Leo Gorski, Luigi Lo Iacono, Hoai Viet Nguyen, and Daniel Behnam Torkian. *International IEEE Conference on Services Computing (SCC)*. IEEE Computer Society. 2014, pp. 464–471. (https://doi.org/10.1109/SCC.2014.68)

"SOA Readiness of REST." Peter Leo Gorski, Luigi Lo Iacono, Hoai Viet Nguyen, and Daniel Behnam Torkian. *The Third European Conference on Service-Oriented and Cloud Computing (ESOCC)*. Springer Berlin Heidelberg. 2014, pp. 81–92. (https://doi.org/10.1007/978-3-662-44879-3_6)

"On the Need for a General REST-Security Framework." Luigi Lo Iacono, Hoai Viet Nguyen, and Peter Leo Gorski. *Future Internet Journal*, 11(3), 2019. (`https://doi.org/10.3390/fi11030056`)

"*Privacy matters – Privacy is what allows us to determine who we are and who we want to be.*"
– Edward Snowden, 2013

Acknowledgments

I am very grateful that the opportunity to do a doctorate opened up for me. I still consider it a privilege. Without a doubt, the most valuable thing was the time I shared with many wonderful and inspiring people. Many of them have consciously or unconsciously propelled me over time to continue with my research. Some of them have also supported my work actively. This thesis would not have been possible without this motivation, support, and preliminary work by other scientists and software developers. I'm standing on the shoulders of giants. My thanks go to all of you.

I thank those people who dedicate their lives to the precious goal of research and teaching, inspiring others to follow in their footsteps. My mentor, Professor Luigi Lo Iacono, showed me this path and always accompanied me on the long journey. I am grateful for the numerous lessons, his advice, encouragement, and confidence. I will miss this time. Special thanks also go to my supervisor Professor Sebastian Möller, for his extraordinary support and commitment. I appreciate his expert advice and his quick answers to all questions. I'm thankful for a great time in Berlin. I would also especially like to thank Professor Simone Fischer-Hübner and Professor Sergio Lucia for serving on my doctoral committee.

The love and support of my family have made many things more comfortable. I thank my wonderful and beloved wife, Tatjana, who gave birth to our family. I'm most proud to be Johanna and Marlene's dad. I thank my twins for being fighters and for challenging and surprising me every day. The doctorate was not the only difficult path we have walked together as a family. I thank my father, Peter, for many passionate and encouraging late-night conversations and my mother, Ingeborg, for her listening ear, warmth, and dedication. I thank my brother, Marco, for his inspiring courage to go his own way. I am very proud of you. I thank my grandmother, Jenny, for becoming old enough to see me get a doctorate. And I thank my father-in-law, Robert, for showing me relaxed and practical ways of doing things. I love you all.

I am grateful to Viet Nguyen for the time we spent together on our doctorates. A sorrow shared is a sorrow halved. I am obliged to thank all the other members of the DAS Group at TH Köln for their support and cooperation. It was a pleasure to work with Stephan Wiefling, Jan Tolsdorf, Markus Marcinek, Paul Höller, Rafael Mäuer, Tobias Mengel, Florian Dehling, Peter Nehren und Daniel Torkian. It was also an honor to collaborate with Prof. Sascha Fahl, Yasemin Acar, Dominik Wermke, and Christian Stransky. I'm thankful for a great and inspiring time in Hannover. Furthermore, I like to thank the team members of the Quality and Usability Lab at TU Berlin: Lydia Kraus, Maija Poikela, Tobias Jettkowski, and Vera Schmitt.

My sincere thanks go to all participants of my studies, without whom my research would not have been possible. I am also deeply indebted to the citizens of the Federal Republic of Germany. The funding instrument "Forschung an Fachhochschulen," of the German Federal Ministry of Education and Research, made it possible for me to do my doctorate in the first place. Therefore I always felt a motivating responsibility to finish the doctorate successfully.

Finally, I would also like to take the opportunity to thank the reader. I hope this thesis will help subsequent research to extend knowledge and improve the usability of security APIs ~ panta rhei (everything flows).

This work has been partially funded by the German Federal Ministry for Economic Affairs and Energy (Grant no. 01MU14002) and by the German Federal Ministry of Education and Research within the funding program "Forschung an Fachhochschulen" (contract no. 13FH016IX6).

GEFÖRDERT VOM

Contents

1. Motivation and Introduction **1**

1.1. The Human Aspect of Data Security and Privacy 1
1.2. Usable Security 2
1.3. Usable Security for Software Developers 4
1.4. Research Questions and Contributions 5
1.5. Outline 7

2. State of the Art **9**

2.1. Software Developers 9
2.2. Studying Developers 10
2.3. Secure Sofware Development Tools 11
2.3.1. Development Environments 12
2.3.2. Code Creation Tools 12
2.3.3. Static Code Analysis 13
2.3.4. Manual Code Review 13
2.4. Sources of Information 13
2.4.1. Individual Community Support 14
2.4.2. Documentation 14
2.5. Software Development Management 15
2.5.1. Third Party Components Risk Management 15
2.5.2. Social and Organizational Security Culture 15
2.6. Productive and Test Runtime 16
2.6.1. Software Environment 16
2.6.2. Testing 18
2.7. Application Programming Interface (API) 18
2.8. API Design Space 20

2.9. Security APIs 21
2.9.1. Classification of Security APIs 21
2.9.2. Usability Problems of Security APIs 23
2.10. Conclusions from the State of the Art 24

3. The IFSIF Model **26**
3.1. Model Structure 27
3.2. Model Validity 29
3.3. Application of the Model to the State of the Art 31

4. Security API Design **35**
4.1. Study 1: Usability Characteristics of Security APIs 35
4.1.1. Motivation 35
4.1.2. Methodology 36
4.1.3. Results 37
4.1.4. Discussion 46
4.2. Study 2: Abstraction Levels of Security APIs 48
4.2.1. Motivation 48
4.2.2. Methodology 50
4.2.3. Results 52
4.2.4. Discussion 62

5. Security API Warnings **64**
5.1. Study 3: Security API Warning Prototype 65
5.1.1. Motivation 65
5.1.2. Security Advice Prototype Design 67
5.1.3. Methodology 71
5.1.4. Results 82
5.1.5. Discussion 87
5.2. Study 4: Participatory Design 90
5.2.1. Motivation 90
5.2.2. Methodology 93
5.2.3. Results 99
5.2.4. Discussion 113

6. Additional Information Flows for Improving Security APIs 116
6.1. Study 5: Security by Default in Web Development Frameworks . . . 117
6.1.1. Motivation . . . 117
6.1.2. Content Security Policies . . . 119
6.1.3. Methodology . . . 124
6.1.4. Results . . . 131
6.1.5. Discussion . . . 136
6.2. Study 6: Secure Code Examples in API Documentation . . . 142
6.2.1. Motivation . . . 142
6.2.2. Security in API Documentation . . . 145
6.2.3. Methodology . . . 146
6.2.4. Results . . . 156
6.2.5. Discussion . . . 170

7. Conclusions and Future Work 172
7.1. Conclusions . . . 172
7.1.1. Security API Design . . . 173
7.1.2. Security API Warnings . . . 175
7.1.3. Additional Information Flows for Improving Security APIs . 176
7.2. Future Work . . . 177

Bibliography 180

A. Detailed API Usability Model Attributes (Study 1) 210

B. Online Questionnaire (Study 2) 213

C. Online Study Questionnaire (Study 3) 218

D. Laboratory Study Interview Questions (Study 5) 223

E. Laboratory Study Interview Questions (Study 6) 226

Acronyms

ACM:	Association for Computing Machinery
AES:	Advanced Encryption Standard
API:	Application Programming Interface
ASIDE:	Application Security in the IDE
AWS:	Amazon Web Services
CA:	Certificate Authority
CBC:	Cipher Block Chaining Mode
CCPA:	California Consumer Privacy Act
CFB:	Cipher Feedback Mode
CHI:	Computer Human Interaction
CI:	Continuous Integration
CSP:	Content Security Policy
CSRF:	Cross-Site Request Forgery
CSS:	Cascading Style Sheets
CTR:	Counter Mode
CWE:	Common Weakness Enumeration
DES:	Data Encryption Standard
DevOps:	Software Development and information-technology Operations
ECB:	Elliptic Curve Builder
fMRI:	Functional Magnetic Resonance Imaging
GDPR:	General Data Protection Regulation
GUI:	Graphical User Interface
HCI:	Human-Computer Interaction
HMAC:	Keyed-Hash Message Authentication Code
HSTS:	HTTP Strict Transport Security
HTTP:	Hypertext Transfer Protocol
HTTPS:	Hypertext Transfer Protocol Secure
HTML:	HyperText Markup Language
ID:	Identification number
IDC:	International Data Corporation
IDE:	Integrated Development Environment

IEC:	International Electrotechnical Commission
IEEE:	Institute of Electrical and Electronics Engineers
IFSIF:	Influencing Factors on Security-Relevant Information Flows
ISO:	International Organization for Standardization
ITU:	The International Telecommunication Union
IDC:	International Data Corporation
IRB:	Institutional Review Board
IT:	Information Technology
IV:	Initialization Vector
JEE:	Java Platform, Enterprise Edition
KDF:	Key Derivation Function
MWU:	Mann-Whitney U test
NIST:	National Institute of Standards and Technology
OWASP:	Open Web Application Security Project
PDF:	Portable Document Format
PGP:	Pretty Good Privacy
Q&A:	Question and Answer
RC4:	Rivest Cipher 4
REST:	Representational State Transfer
RPC:	Remote Procedure Call
RQ:	Research Question
RSA:	Rivest–Shamir–Adleman
SANS:	SysAdmin, Audit, Network, Security Institute
SD:	Standard Deviation
SDLC:	Software Development Life Cycle
SDP:	Software Development Process
SHA:	Secure Hash Algorithms
SQL:	Structured Query Language
SSDLC:	Secure Software Development Life Cycle
SSL:	Secure Sockets Layer
STL:	Standard Template Library
SVN:	Apache Subversion

TLS:	Transport Layer Security
URI:	Uniform Resource Identifier
URL:	Uniform Resource Locator
W3C:	World Wide Web Consortium
XHR:	XMLHttpRequest
XML:	Extensible Markup Language
XOR:	Exclusive OR
XSS:	Cross-Site Scripting

1. Motivation and Introduction

Cambridge Dictionary explains the internet as a *"large system of connected computers around the world that allows people to share information and communicate with each other"* [41]. These two primary purposes – communication and information exchange – have a pervasive impact on all personal areas of people.

We can record our intimate thoughts and feelings in an online diary. We can organize and simplify our private and professional life with online services. We can send messages to well-known people or make them public to the whole world. We can run businesses and scientific studies. We can manage global financial systems, automate industries, and wage wars. From a user's point of view, privacy and public space have visually just the distance of a graphical button. Limits of what is possible are only technically set by the capabilities of hardware and software, creating, storing, processing, and presenting data.

1.1. The Human Aspect of Data Security and Privacy

At the end of 2019, approximately 7.7 billion people lived on earth. The ITU estimates that at that time, about 53.6%, or 4.1 billion people, were using the internet [93]. A third of them were under eighteen years old [121]. This high number of internet users has a great need for software across a variety of application platforms, like mobile, wearables, desktop, or devices using embedded systems.

The ratio of software developers to internet users is approximately 1:178. In 2018 23 million people worldwide were developing software with a varying degree of professionalism [52]. The IDC's Worldwide Developer Census estimates that 52% are full-time, 28% are part-time, and 19% are nonprofessional developers [52]. This heterogeneous group of people satisfies our current need for software. Furthermore,

just as most end-users are not familiar with data and application security, not every software developer is a security expert. Nevertheless, they have the responsibility of implementing data security and privacy for over four billion people.

As we know, people entrust their most intimate and private data to their devices, such as photos, voice, and video recordings, health records, or other sensitive documents. Disclose of personal data happens consciously by users as well as unconsciously if software collects data without indication in a user interface. Therefore, Europeans have developed a deliberate sense of privacy when these data are stored, processed, and used by third parties. With the GDPR (General Data Protection Regulation) [193], European legislation has declared the protection of natural persons in relation to the processing of personal data as a fundamental right. With a similar intention, on January 1, 2020, the California Consumer Privacy Act of 2018 (CCPA) [38] went into effect.

However, the enforcement of this right is a challenge. In a parliamentary democracy, governments and administrations have to implement the laws that have been passed. Nevertheless, the implementation already causes problems on a much more fundamental technical level. In 2019, the Verizon Data Breach Investigations Report [203], published annually since 2008, contains 41,686 security incidents. Of these, a total of 2.013 cases are confirmed data breaches.

Not properly operating security tools pose a particularly high risk for some groups of people, such as human rights defenders, political activists, and journalists [9]. For example, they are dependent on the confidentiality of data. Otherwise, their integrity may be at risk. However, every other person who wants to protect their privacy against third parties or conduct digital financial transactions also needs reliable security tools.

1.2. Usable Security

If one wants to make security systems more reliable, the question naturally arises why a large number of security incidents occur. Part of the answer lies in the usability of the software components, which should provide the security of the data. The ISO standard 9241-11 for ergonomics of human-system interaction defines

usability as the *"extent to which a system, product or service can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use"* [97]. The standard uses this formulation since 1998 [95] and explicitly names systems and services in addition to products since its revision in 2018. Consequently, this definition is also applicable in the context of this thesis for software and especially for Application Programming Interfaces (APIs). The context of use is a combination of users, their goals and tasks, available resources like tools, and environmental influences. The international standard for systems and software engineering ISO/IEC 25010 considers both usability and security as central quality features [98]. Software engineers can meet security requirements by utilizing security concepts like accountability, authenticity, confidentiality, integrity, and non-repudiation [98]. Concerning a software system, security is also a continuous process [173].

When humans meet technologies, mechanisms, and products, an important question is whether people can use these functions effectively and efficiently and how satisfied they are with that product and the achieved results. Usability problems in security can lead to functional failure because users may start bypassing security features, switching them off, or they cannot fulfill their actual primary task. That, in consequence, exposes users to risk [110, 55, 168]. The opinion that users are the weakest link in the chain when it comes to data and application security is still widespread among the public [73]. People typically blame other people when a company or person suffers damage due to missing or non-functioning security [33]. In this respect, users only operate within the limits of a provided software framework. A critical consideration of missing support by hardware and software and a lack of usability is not common when it gets to security at the time of this thesis, but is urgently needed as research proofs.

Security and usability are not software characteristics that are strictly mutually exclusive [169]. The publications *"User-Centered Security"* [236], *"Users Are Not the Enemy"* [6], and *"Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0"* [216], which manifest the beginning of the Usable Security research field around 1996, already convey another approach through their titles. They address the problem of how software products can support users in reliably implementing security mechanisms. Scientists put users' capabilities, perceptions, and expectations

into the center of the considerations in the development process to empower people to become a strong link in a security system or even *"The Strongest Link"* [205], which stresses the paradigm shift.

Usable security research is an interdisciplinary field. For measuring usability, scientists can apply empirical methods of human-computer interaction. Effectiveness and efficiency can be measured, e.g., by time measurements, success, or failure rates. Questionnaires or interviews can measure, e.g., perception of success and satisfaction with a software product.

The context of use is a challenge when designing usable security studies. People have a subjective perception of risk and have different security knowledge leading to individual behavior. Researchers cannot expose their participants to real risk. In real life, other people may attack the resilience of a system in a targeted and active way. Also, we have limited cognitive capacity, habituation can strongly influence our behavior, or we are subject to natural principles such as Least Effort. The application of security functions in real situations is often not a user's primary goal, but a secondary task on which, for example, the current ability to concentrate, temporal stress or other environmental influences can have a decisive influence.

1.3. Usable Security for Software Developers

Of course, usability problems of end-users are not the only reason for security incidents. Twenty years after the beginning of usable security research, researchers are also increasingly focusing on usability problems of other essential target groups, such as system administrators [56], system integrators, and especially software developers [223, 89, 4]. Research on these target groups is currently precious, as scientific knowledge is scarce to date. However, it also poses a challenge because potential study participants are not so numerous and, therefore, difficult to recruit.

In terms of numbers, these groups are not initially as attractive as the large group of end-users, but they hold key positions in the development and operation processes of software. It is particularly necessary to understand the demands of software developers, as insecure code in productive software triggers an epidemic that affects all users installing and using the vulnerable application.

Probably the most basic, powerful, and frequently used tools software developers work with are, besides programming languages, Application Programming Interfaces (APIs). By interacting with APIs, developers integrate required functions into their software without necessarily having to program them from scratch. Security APIs are essential tools for all software developers to bring data and application security into their products to take care of any user data. Specialized programmers implementing security protocols and cryptographic algorithms provide this complex program code for non-security experts and also design the interfaces with which other developers interact [30].

However, security experts are not necessarily usability experts, either. Thus, research results have also revealed in the case of security APIs that usability problems lead to insecure API use during development [76, 61, 159]. Especially cryptographic APIs are difficult to use [133, 2] even for experienced and professional programmers [141]. Basic security requirements such as securely stored passwords, encrypted files or secure network connections can become an error-prone challenge [135] and in consequence lead to unreliable or missing security and privacy. However, little is known about the requirements of developers to address the problem and improve the usability of security APIs.

An important development phase for the integration of security functions is the implementation stage [19], in which the software developer writes the program code and API calls. The development environment of a software in which it is written and processed is complex. Thus, it can comprise several phases and actors. In addition to the APIs as basic building blocks, developers need information about usage, as well as tools for programming, managing development projects, and testing and publishing software versions. There is a particular need to understand whether information flows in complex development environments support software developers in using security APIs and implementing secure software.

1.4. Research Questions and Contributions

The motivation of this work is to extend the current state of knowledge about the usability of security APIs. To this end, the author examines to what extent infor-

mation flows can support software developers in using security APIs to implement secure software. Hence the central research question of this thesis is:

> **To what extent do information flows support software developers in using security APIs and implementing secure software?**

Since this thesis is one of the first to examine the usability of security APIs, the author follows a specific methodology that addresses the general research question by defining a sequence of more specific research questions in a bottom-up approach. As an initial starting point, the author reviewed and systematized related work on the usability of APIs intending to answer

> **RQ1**: *Are approaches from the API usability research sufficient for the specific context of security APIs?*

Besides knowing about the usability specifics of security APIs, it is also important to understand what types of security APIs exist and what developers expect when interacting with a security API, which led to

> **RQ2**: *What level of abstraction do software developers prefer when working with security APIs?*

Early results of this work pointed out that the design of security APIs cannot remove all complexity and that developers need access to functionality that can be error-prone. This insight led to the idea of implementing functionalities that could promote the flow of security-relevant information into an API at the code level. As related work had not produced any findings in this regard, this thesis explored a novel approach with

> **RQ3**: *Does API-integrated security advice have a significant effect on code security and perceived API usability?*

The first positive results motivated the author to further question and improve this own approach by

> **RQ4**: *What kind of design do software developers find helpful for cryptography warning messages in the console?*

To explore the extent to which information flows can be supportive, the complex API environment of a web development framework had been examined in

> **RQ5**: *How does the enforcement of security by default affect the usability of a web development framework?*

Based on these results, the author had developed a new constructive approach which led to

> **RQ6**: *How can documentation writers usefully integrate security-relevant information into a documentation website of a web API?*

This thesis contributes to the field and literature (1) a provisioning model of security-relevant information in software development; (2) An API usability model that also applies to security APIs and a list of usability characteristics that are specific to the usability of security APIs [84]; (3) a classification of security APIs and the result of an online survey, that software developers want an API to offer features in different levels of abstraction [122]; (4) a prototype for security advice directly integrated into an API and the proof of concept by an online development study with software developers [85]; (5) a participatory design for cryptographic API warnings based on focus groups with software developers [83]; (6) usability requirements for the implementation of security defaults in web development frameworks through a laboratory study with inexperienced web developers [86]; (7) a prototype for the integration of Content Security Policies (CSP) [222] in documentation as well as the evaluation through a laboratory eye-tracking study with inexperienced web developers [87].

1.5. Outline

The thesis is structured as follows. Chapter 2 explains the theoretical background and terminology that are necessary for the understanding of this thesis and summarizes the current state of scientific knowledge. Chapter 3 introduces a model to structure the related work to software development areas and to explain the scientific relevance and contribution of this work. Each section presenting research results instantiates

this model at its beginning in order to explain the motivation of the study and the respective object of research.

The first two studies described in Chapter 4 examine basic questions regarding the design of security APIs. Based on a literature review, Section 4.1 discusses similarities and differences between usability characteristics that are common to programming interfaces and those that are specific to security APIs. The following study in Section 4.2 uses an online survey to investigate which type of design approach to the abstraction level of an API is preferred by software developers when using security APIs.

The findings from the security API design studies have motivated research that, for the first time, examines the approach of a security warning in the developer console that comes directly from the API (cf. Chapter 5). Section 5.1 presents an online experiment that examines a prototypical warning design for an existing cryptographic API applying accepted heuristics from usable security research for end-users. Pursuing and questioning initial results, Section 5.2 presents a participatory design for cryptographic API warnings in the developer console that software developers have developed in focus groups.

Chapter 6 presents two laboratory studies focusing on additional information flows to support software developers in using security APIs and implementing secure software. Section 6.1 examines usability deficiencies of a "security by default" design in development environments for web applications. Section 6.2 presents the results of an eye-tracking study that examined places in API documentation where application security guidance can help developers configure CSPs in a framework.

The thesis ends with conclusions arising from the results of a total of six studies and the elaborated model, and a discussion of potential future work that could tie in with this research (cf. Chapter 7).

2. State of the Art

This chapter reflects the state of the art on developer-centered security research. Related work tries to support software developers in implementing security in different areas. In 2019 the first survey on developer-centered security had been published at the European Workshop for Usable Security [192], which indicates that this has become an active field in recent years.

2.1. Software Developers

In 2018, there were, according to projections, more than 23 million software developers worldwide [52]. These are different in the way they program and how they interact with APIs. Steven Clarke has observed the behavior of developers in several usability studies and characterized the programming styles by three personas [47, 48, 184, 49]:

The *opportunistic developer* focuses on the quick solution of his task and proceeds exploratively according to the bottom-up principle. This type of programming is the most common among developers. He starts directly with the solution of his concrete problem by using API components with a high level of abstraction, which allows him to hide detailed correlations. He wants his program code to work as quickly as possible and is only interested in understanding the relationships if they are relevant to the direct context of the task. The opportunistic developer prefers easy-to-use and straightforward programming languages like Visual Basic, or Python, which are designed for high productivity at the expense of control. They see themselves as problem-solvers for business problems.

The *systematic developer* proceeds according to the top-down defensive principle and thus follows the opposite pattern of action to the opportunistic developer. This type of programming is scarce. Before using the API, he first tries to gain

a holistic understanding of the technology. He avoids assumptions by analyzing the program code. The systematic developer distrusts warranties of the API and performs additional tests in his environment for validation. The functionality plays only a minor role because this type of developer wants to know why the API works, what assumptions the API developers have made, and where errors can occur. He prefers programming languages with extensive control options, such as C++, C, or hardware-related assembly languages. Systematic developers believe that they achieve elegant solutions.

The *pragmatic developer* is more common than the systematic developer and is less common than the opportunistic developer. In their behavior, this type of programmer also falls between the other two personas. At first, the pattern of action resembles the opportunistic developer. He begins to look for a quick solution to the problem using the bottom-up principle. However, if he encounters limits or problems, he switches to a top-down approach, similar to the approach of the systematic developer. Pragmatic developers say of themselves that they develop robust programs and prefer programming languages such as Java and C#.

Accordingly, individual behavior is characterized by motivations, resulting in different user expectations of an API. Expectations also arise from mental models [106], a personal mental representation that leads to a developer's belief in how an API works, that are shaped by knowledge, experiences, and ideas which can change dynamically during use. From a usability perspective the actual model of the API should be close to the mental version of a developer [186]. Users can also expect compliance with common conventions [124]. Whether an API is usable or not is thus also directly related to whether a design considers individual user preferences.

2.2. Studying Developers

[182] based on Jupyter Notebook [108] and several server components, and published it as open-source software [183].

In the academic environment of computer science, in particular, the question arises as to what validity study results with student samples have. Previous studies could not find significant differences compared to professional developers [5, 135,

134]. Naiakshina et al. conducted a qualitative developer study and investigated how computer science students implemented secure password storage [135]. Later they repeated this study with professional developers [134] and compared the results. They found that neither professional developers nor students are sufficiently familiar with security concepts to be able to apply them securely. Acar et al. also compared developer populations with similar conclusions [5]. However, they found evidence that developers having more experience with a programming language will more likely produce secure code [5]. Therefore, the population of prospective and junior developers is a sensitive target group that particularly needs support with implementing security into their products.

A problematic situation arises from the fact that programming tasks that affect software security are often only secondary tasks missing attention [223, 140, 89, 135]. Risks and vulnerabilities are also a problem, which arises because the security context is not apparent to a developer, or security during programming is not part of the mindset. Oliviera et al. call this context "API blindspots" [140, 141]. A typical example is an API that receives and processes user input from a web page. The written source code may look unremarkable, but there is a risk that attackers could enter their code into the form, which executes because the developer has not validated the input [144]. At this point, a secure implementation has to filter the input using a Security API. Oliveira et al. found that specific code related hints about vulnerabilities can improve security awareness [140]. Naiakshina et al. could prove that fewer study participants ignored security when it was an explicit part of the task description [135, 136]. The question remain, how software development tools can rise awareness for security tasks in practice.

2.3. Secure Sofware Development Tools

Programmers can draw from a wide range of tools that support them in the development of software. This section provides an overview of tools and developer-centered design approaches focusing on the implementation of security.

2.3.1. Development Environments

In contrast to user-defined development environments consisting of individual independent programs, an Integrated Development Environment (IDE) combines software development tools, like a text editor, file explorer, a debugger, a developer console, or version control systems into one single software. This integration allows developers to avoid software and environment changes to work more efficiently. IDEs can usually be extended with plugins or addons to enhance their functionality. Several scientists have presented approaches for IDE extensions to improve code security:

IDEs offer graphical capabilities that can draw the developer's attention to security vulnerabilities or secure programming practices. The tool ASIDE (Application Security in the IDE) [226, 234] enhances the text editor with interactive code annotation [118], including highlighted source code, icons as security indicators, real-time warnings, and code generation. Usability evaluations with students [233] and professional developers [227] had positive results. It also effectively gives support to mitigate access control vulnerabilities [195, 232] and to educate secure programming [215, 214, 191, 190]. Nguyen et al. have successfully adopted a similar approach to AndroidStudio. They developed a plugin called FixDroid [137], which helps developers write more secure code by insecure code highlighting and automated code fixes, which give the users implicit actionable support at a click. In a user study, they found that FixDroid users write significantly more secure code than participants without the plugin.

2.3.2. Code Creation Tools

Krüger et al. presented CogniCrypt [115], an Eclipse IDE plugin for the Java programming language to take the burden of writing code off the developer. Developers first define their security needs in a wizard and get corresponding generated source code that securely uses cryptographic APIs from the tool.

2.3.3. Static Code Analysis

Static code analysis is another type of tool to find bugs and mitigate insecure code use, which is also usually supplied as a plug-in for IDEs. The challenge lies especially in the development of algorithms [57] that analyze written source code and in interaction design, which should communicate analysis results and advise the user. Too many false positives and complicated error messages are significant hurdles for developers and thus reasons why they do not want to use these tools [105]. Also, the warning message design is a pervasive challenge [21, 22, 23, 45]. An exact configuration of program analyzers can improve the number of relevant warnings for a specific user context [13]. However, this requires the user to have profound know-how. While processing the analysis results, developers need visual [17] and actionable [196, 175] assistance with code review to find answers to questions about vulnerabilities, attack-vectors, fixes, and more [176].

2.3.4. Manual Code Review

Developers can review code manually to find security vulnerabilities. Edmundson et al. examined this approach [60]. They hired thirty professional developers to find vulnerabilities in a web application. Comparable to static code analysis, false positives prove to be a problem also with this manual type of code analysis. The results also showed that manual code evaluation is a difficult task because none of the test persons could find all configured vulnerabilities. The results also indicate that manual code reviews cannot generally be performed reliably by experienced software developers, but probably only by specialized developers.

2.4. Sources of Information

Software developers can draw from a wide range of available online sources to meet their information needs and to find solutions to problems they encounter during development. However, reliable sources of information on security issues are less numerous [1].

2.4.1. Individual Community Support

Online question and answer platforms (Q&A) focusing on software development topics are popular. There is even the "Security Stack Exchange" for security professionals [178]. They usually follow a very solution-oriented approach, which firstly allows only clearly stated and problem-related questions and secondly allows only answers that serve to solve that specific problem. This concept makes it helpful for everyone dealing with a similar problem, and such services fit very well with developers preferring the most common opportunistic programming style. About 50 million people visit the popular Stack Overflow each month [179], thus developers frequently consult actionable advice from such community platforms. However, as research has shown, the risk is high that they copy and paste insecure code examples [3, 24].

2.4.2. Documentation

When having a free choice, developers use not only secondary web sources such as blogs and developer-centered Q&A platforms, but also the official sources of software vendors [3]. Many studies in recent years draw the same conclusion that the content and form of official API documentation, in particular, should be more oriented towards the needs of users through easily accessible, practice-oriented recommendations for action and code examples [1, 2, 217]. These usability factors can have a significant impact on developers' security decisions [3]. Mindermann and Wagner [129] have presented an open-source web platform that collects such examples for various cryptographic APIs and makes them centralized available for software developers.

Different forms are suitable for documenting acquired knowledge about secure software development. Principles, patterns, and guidelines are typical formats. Principles are general rules on a high level of abstraction, guidelines provide instructions on how to adopt principles for a particular context, and patterns document actionable advice to concrete implementation problems [88]. However, researchers were not yet able to prove that the use of security patterns improves the security of software designs or the productivity of the software designers [229, 230]. Nevertheless, Lo Iacono et al. have published an online platform with a collection of

principles, guidelines, and patterns as guidance for software developers to implement usable security [123, 88]

2.5. Software Development Management

Professional software development is a complex process. This chapter presents research on management factors influencing the security of software. In addition to tools of the development environment (cf. Section 2.3.1), tools for managing development processes and teamwork, such as issue tracking, continuous integration, or source code collaboration, have become firmly established in software development [103]. However, aspects influencing the quality of software security are mostly unexplored.

2.5.1. Third Party Components Risk Management

Any use of third party components also carries the risk of adopting security vulnerabilities. For this reason, APIs are usually continuously improved, and producers repair identified vulnerabilities. Continuous update processes are supposed to make these changes effective in the user's implementations. Derr et al. could prove that most Android developers do not update their third party components, although this would be possible without code adjustments [53].

2.5.2. Social and Organizational Security Culture

Reasons for lack of software security are not only technical, but related work has identified several social and organizational reasons. Xie et al. found that a developer's attitude of not being responsible for security is a significant obstacle [225]. Weir et al. have discovered that there is even no consensus in the industry on how to motivate developers individually to improve their security skills [211]. Bartsch found indications in their results that agile development with its positive effects on internal communication could spread awareness and expertise among developers [27].

Social and organizational factors influence developers' decisions of security tool adoption, as well. When choosing tools, they orientate themselves towards their peers

and prefer these over others [224, 221]. Nevertheless, developers with excellent online reputations can also influence software development communities [224]. Besides, experience, inquisitiveness, and security awareness are driving intrinsic factors in whether a developer adapts a security tool or not [220, 221]. Poller et al. could observe a positive effect of security developer training as part of a penetration test, even over time. However, the developers could not effectively apply what they had learned due to firmly established routines within the organizational structures [199, 155]. Assal and Chiasson also found evidence that developers being security motivated struggle with missing organizational or process support [19]. Furthermore, best practices for secure software development in the literature do ignore organizational needs and in consequence, do not fit real-life security practices [18].

2.6. Productive and Test Runtime

Developers use programming languages to write program code that can be executed by a computer in test environments and productive runtime environments. In this productive and test environment, researchers identified some factors that can influence the development of secure software.

2.6.1. Software Environment

The software environment includes programming languages or extensive software platforms, compilers, and interpreters.

Programming Languages

The landscape of programming languages available to developers is extensive. The IEEE Spectrum annually presents the top ten programming languages weighed for job seekers and open-source enthusiasts [43]. Besides the ranking of 52 languages in total for the year 2019, the list also indicates whether the programming languages are suitable to implement mobile, web, embedded, or enterprise applications. Amongst the top ten, Java, C, C++, C#, and Swift are currently most in demand for the development of mobile applications. Developers of websites and applications do

have a focus on languages, including Python, Java, JavaScript, C#, and Go. A total of nine of the ten programming languages with Python, Java, C, C++, R, C#, Matlab, Swift, and Go are also suitable for developing enterprise, desktop, and scientific applications. Languages used to program embedded device controllers are C, C++, Python, and C#.

Compilers

In compiled programming languages, a compiler translates human-readable program code into machine code. Thus it is the technical link between a developer and his computer. During compile-time, a compiler communicates general information about the process, explicit warnings, and errors, for example, in case of syntactical errors in the written source code, to the developer via a console. A console is typically a window that offers a command-line interface. It receives commands in text form and returns feedback also in text form to the user. In software development, the console is a central and vital point for information. Thus, it is also relevant for security-related warning or error messages.

Barik et al. conducted an eye-tracking study to investigate the use of Java compiler error messages and found evidence that participants do read compiler error messages [26]. They also found that reading this type of message is comparably complex to reading source code. The problems developers have with the complexity of compiler errors have also motivated previous research to find design improvements.

Barik et al. also have formulated three principles compiler developers should apply for the design [25]: (1) “Allow developers the autonomy to elaborate arguments”, (2)“Distinguish fixes from explanations”, and (3) “Apply argument structure and content to the design and evaluation of error messages”. Becker found evidence that an extended editor can help developers make fewer errors by complementing compiler errors with additional explanations [29]. Traver proposes and theoretically discusses eleven abstract principles for compiler error message design: (1) “Clarity and brevity”, (2) “Specificity”, (3) “Context-insensitivity”, (4) “Locality”, (5) “Positive tone”, (6) “Constructive guidance”, (7) “Programmer language”, (8) “Nonanthropomorphism”, (9) “Consistency”, (10) “Visual design”, and (11) “Ex-

tensible help" [197]. These show broad similarities with established and approved usability heuristics for user interface design [138].

2.6.2. Testing

Continuous Integration (CI) is a widely used approach [103] to automate repetitive processes in development, such as compilation, building, and testing, to make them more efficient and traceable. A challenge is organizing and regulating access to CI tools to guarantee code integrity [90]. Rahman and Williams found that DevOps (software development and information-technology operations) automations for deployment, testing, and monitoring can support developers and system operators in developing secure software [158].

2.7. Application Programming Interface (API)

Application Programming Interfaces (APIs) are a central object of research in this thesis because they are a core tool for developing software[1]. This chapter first defines the term and explains their concept. A mature proposal for the definition of application programming interfaces is:

> "*An application programming interface (API) specifies a component in terms of its operations, their inputs and outputs. Its main purpose is to define a set of functionalities that are independent of their implementation, allowing the implementation to vary without compromising the users of the component. An API augments a programming language (or a set of languages with an interoperable calling convention). Alternatively, an API may be described in an interface definition language.*" Bloch [32]

The reason for the success of APIs and their widespread use is that they represent the building blocks of software development with which every developer builds his applications. APIs abstract the complexity of implementations on code level and make concrete functions usable via an interface. Software developers integrate

[1] The present section is based on "I Do and I Understand. Not Yet True for Security APIs. So Sad." by Luigi Lo Iacono and Peter Leo Gorski [122].

already existing problem solutions in order to solve other problems without having to start from scratch. From a technical perspective, there are several possibilities to link software with each other. Figure 2.1 shows the prevailing styles.

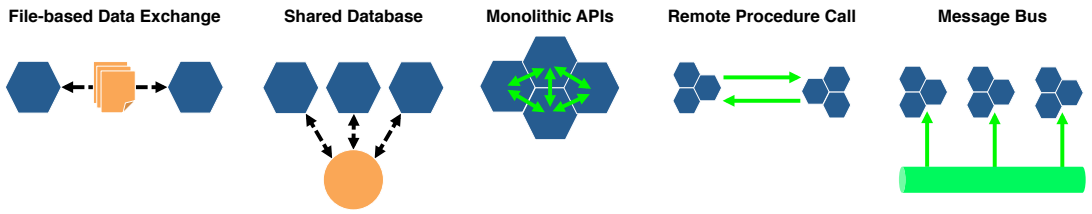


Figure 2.1.: Overview of application integration styles (adapted and extended from [149]). The three rightmost styles encompass the usage of APIs. © Internet Society

The "file-based data exchange" style is straightforward, but this software integration approach does not use interfaces satisfying the above definition of an API. A component specifying the functionalities by its operations, inputs, and outputs is merely missing in this integration style. Only a general data interchange mechanism via file input and output is available. The same is true for the "shared database" integration strategy. No specific functionality are available besides generic data retrieval and storage operations. APIs, according to the above definition, are deployed in the integration style denoted as "monolithic APIs". This term states the use of specific APIs that are limited to a determined execution environment running on a single host. This category most commonly includes stand-alone applications that are tightly composed by compile-time API calls. These can be part of libraries, toolkits, frameworks, or development kits, which are all synonyms for APIs [186]. The "remote procedure call" and "message bus" strategies also make extensive use of APIs. In addition to monolithic APIs, both enclose remote APIs that enable the integration of functionalities provided by external services running on distributed and potentially heterogeneous hosts. This technology is, e.g., required for implementing Web Services. Modern internet-based applications commonly base on the architectural style REST [72] and adopt the RPC integration strategy. These remote APIs are commonly specified using interface definition languages.

2.8. API Design Space

Stylos and Myers have outlined a design framework (cf. Figure 2.2) for API design decisions in object-oriented programming languages. Within this scope, API producers can design APIs and influence its usability [186].

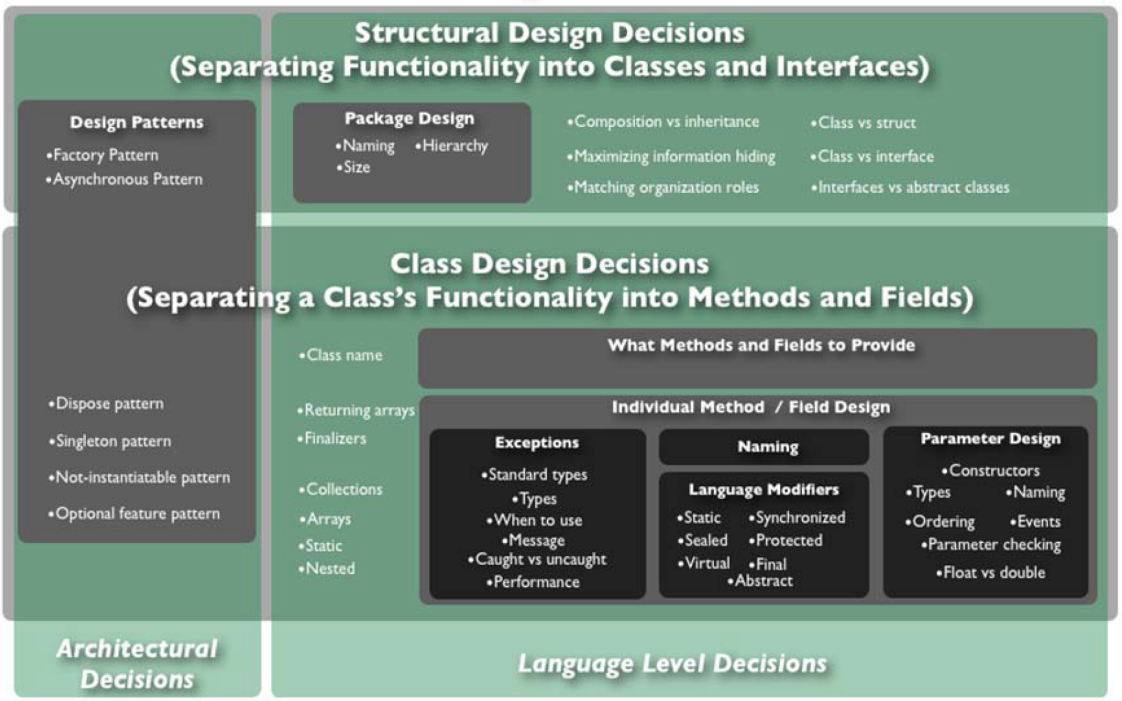


Figure 2.2.: An overview of the scope of API design decisions for object-oriented programming languages by Stylos and Myers [186]. © IEEE

Stylos and Myers divided the freedom of choice shown in Figure 2.2 horizontally and vertically. The vertical division distinguishes between abstract design decisions that affect the API architecture (left) and decisions that a developer has to make at the programming language level (right). The horizontal division separates the design of API structure (above), like the namespace, division of functionalities between packages and classes, and the design within such a class (below), concerning, e.g., methods and fields. Recommendations based on expert opinions exist for the topics that are preceded by a quotation point.

The usability is crucial to support API consumers with correctly integrating requested features and thus to mitigate risks of program errors and security is-

sues [89, 132]. Therefore, designing security APIs requires special attention to factors influencing the communication of security-relevant information between API designers and consumers.

2.9. Security APIs

As security APIs are a particular subgroup of APIs, the general API definition is also valid here[2]. Following this definition, a differentiation to other APIs is possible by defining the particular offered set of functionalities. Considering the current field of application comprehensively, the author has provided this definition for security APIs:

> "*A security API is an application programming interface that provides developers with security functionalities that enforces one or more security policies on the interactions between at least two entities.*" [84].

2.9.1. Classification of Security APIs

The Security API domain consists of two main subgroups (cf. Figure 2.3). "Security primitives APIs" have a rather low level of abstraction, and "Security controls APIs" have a higher level of abstraction.

Security primitives APIs include most prominently cryptographic APIs, but also basic steganographic and watermarking schemes for information hiding. Software developers can use these foundational means to realize essential security services such as confidentiality, integrity, authenticity, and non-repudiation. Security primitives APIs allow selecting, initializing, and combining the associated cryptographic primitives to specific security controls as needed. However, this flexibility demands developers to have a high degree of knowledge and expertise. Without these skills, they will likely fail to develop robust and effective security protection means (cf. Section 2.9.2).

Security controls APIs offer more concrete security functions, like secure communication or data storage. These types of APIs are less flexible than APIs belonging

[2]The present section is based on "I Do and I Understand. Not Yet True for Security APIs. So Sad." by Luigi Lo Iacono and Peter Leo Gorski [122].

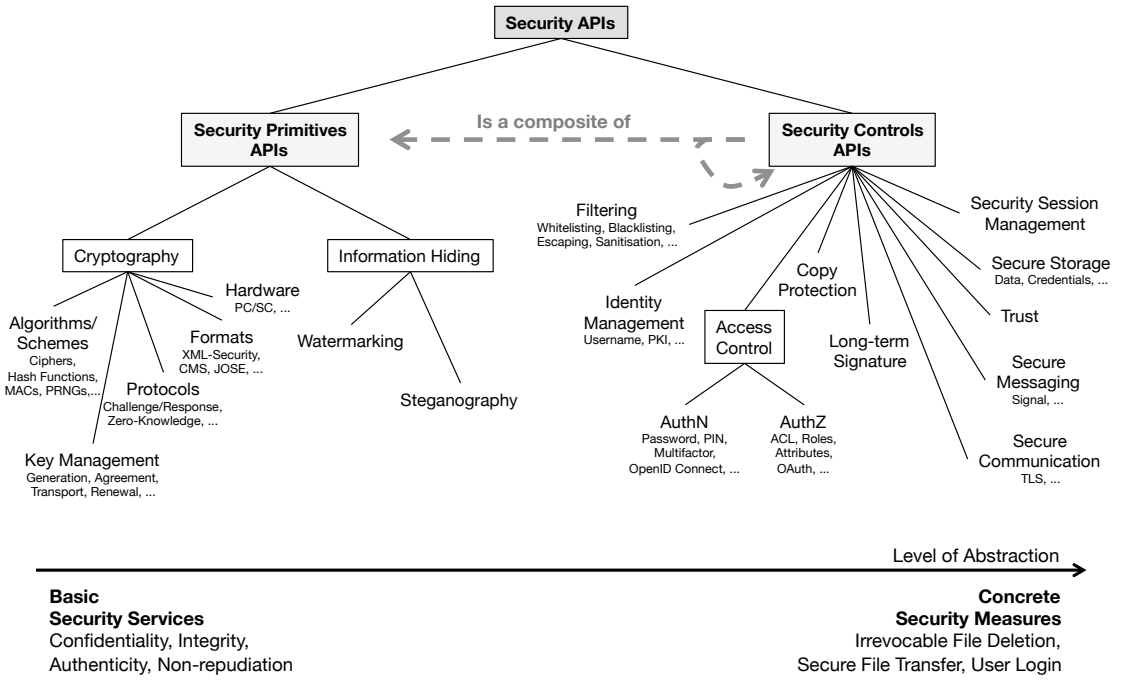


Figure 2.3.: This classification of security APIs shows the main structure of the domain. "Security primitives APIs" provide common mechanisms for implementing basic security services such as confidentiality and integrity. "Security controls APIs" instead provide more concrete security measures including user login and secure file transfer. © Internet Society

to the security primitives. This way, they are ready to use for developers who want to meet specific security requirements. When implemented correctly, they put a lot of security expertise and know-how into practice by default and thus relieve the developers.

Between security primitives and controls APIs, several relationships exist. Proper composition of security primitives most commonly construct security controls. A security controls API can implement a pseudo-random generation API to overwrite files multiple times with junk data and thus offer the functionality of irretrievable deleting a file. This relation is unilateral and does not exist the other way round. Moreover, it does not apply to all security controls. Some security controls do not utilize security primitives at all for performing their internal tasks. For instance, an API filtering data input does not need cryptography for whitelisting input to

a web application form or a network packet filter. Another relationship exists between security controls APIs themselves. Like building blocks, one control can build its functionality on top of another one in a reflexive relation. A single-sign-on API, e.g., requires authentication and probably an identity management API to implement its functionality.

From the usability point of view, it is necessary to consider which target groups security primitives and controls APIs address. Knowledgeable developers with experience in deploying basic security measures should be the primary user group accessing security primitives APIs. The API design of security controls should expect minor security knowledge and respect mental models of common developers by providing required security functionalities with adequate security defaults. Likewise, expert users having regular security requirements will likely use already available implementations of security controls APIs. The more critical scenario seems to be missing security controls or missing flexibility for individual software security requirements. In consequence, developers would have to use security primitives APIs to meet their programming tasks.

2.9.2. Usability Problems of Security APIs

The fact that software developers have problems when using security APIs leading to insecure software, is sufficiently proven by research revealing the symptoms. Particularly critical examples with far-reaching consequences are TLS/SSL connection establishment in Android [65] and iOS [66], TLS/SSL certificate validation in non-browser software [76] and also for OAuth single-sign-on implementations [188]. Current vulnerabilities in software are often blamed on software developers who write insecure code for various reasons. Causes can be, missing implementation, misunderstanding, or simply human mistake [204]. However, the developers had the intention to implement security but failed when using security APIs in insecure ways.

Research has tried to better understand the needs of developers in terms of security APIs to improve their usability. Unfortunately, the common opportunistic approach carried out by developers for exploring and then adopting an API seems not to be compatible with current security APIs. Cryptographic APIs often demand

a high level of cryptography expertise [89, 217]. The right choice of algorithms, the correct sequence of method calls, and the confident handling of parameters overburden non-security expert developers [133, 217, 134]. However, Nadi et al. found that a majority of problems were related to API complexity rather than a lack of domain knowledge when analyzing the top 100 Java cryptography posts on Stack Overflow [133]. Besides complicated designs, missing secure defaults, missing API features supporting needed use cases, and missing information or guidance in documentations favor error-prone use of APIs [3, 2, 136].

Commonly, cryptographic APIs offer insecure features and function calls like outdated cryptographic algorithms or parameters to support compatibility requirements. However, programmers are not informed or warned when implementing such API functionalities into their products. Warnings pointing to an insecure cryptographic API use or potential risks in program code are essential for raising the developers' attention to security vulnerabilities [89, 4]. However, it is not yet well understood how to design proper security warning messages for developers [89].

Previous work proposed ideas to improve code security by enhancing cryptographic API design on a semantic [92] and abstraction level [30]. Jain and Lindqvist found indications that an API design can influence developers' programming choices, which API producers could use to design privacy promoting APIs [101]. Following the principle of the data economy, the researchers have tested an API that does not offer a request for the exact geographic coordinates - as is often the case in practice - but only for less precise but sufficient information, such as country, state, or city. However, to date, we do not know if or how we can design security APIs that developers intuitively can understand and learn how to use them securely [101].

2.10. Conclusions from the State of the Art

Since developer-centered security is still a young field of research, there is little knowledge about the psychology of developers, social and organizational factors, and the methodology of usability studies with developers. When reflecting existing research results, however, it becomes clear that software developers are not a homogeneous group. They differ in many aspects, including their style of programming, their

motivation to consider security aspects during development, and their specialization within software development.

From previous studies with developers, we can learn that computer science students provide comparable results to professional software developers. Therefore usability studies, in software development also consider students as test persons, especially if young or inexperienced developers are needed to answer a research question. When designing tasks for studies, scientists should take into account that security is not necessarily part of a developer's mindset, and it is a secondary task. Researchers investigating the usability of a security mechanism for software developers should carefully distinguish between research questions addressing security as a secondary task and questions dealing with primary tasks, e.g., if requirements engineering processes explicitly determine security programming tasks.

Researchers are trying to find ways to help developers write secure code and thus to improve the documentation, development tools, and security APIs. Since the implementation of security mechanisms is a widespread problem for many developers, this thesis focuses on the essential building blocks that all developers have to work with - security APIs. API producers, therefore, play an essential role and have the trusted task of making the APIs usable and thus supporting their users in the secure application of their products. Most approaches of previous work in the area of software development tools and management, as well as test and productive runtimes, are not located in the API design space and thus elude the direct influence of an API designer. The usability research of security APIs is, therefore, equally important, as there is currently a lack of recommendations and guidelines to help API providers make their products usable. This problem raises the question of which design aspects of a security API exist, that an API producer could use to provide API consumers with security-relevant support and information. In order to find answers to this question in a structured way, the following chapter presents a model of influencing factors on security-relevant information flows in software development.

3. Model of Influencing Factors on Security-Relevant Information Flows in Software Development (IFSIF)

The topics addressed by the state of the art (cf. Chapter 2) illustrate that software development is a complex task and that developers work with diverse software environments. Data and application security aspects add additional complexity. Despite the many optional variables that form a specific usage context for software development, like development tools, management, and runtime environments, every programmer aiming to implement secure software gets into contact with security APIs. Usability problems of security APIs which have been identified by previous work (cf. Section 2.9.2) indicate that a significant reason for insecure API usage lies in the situation that API consumers do not receive the information they need to use it securely. Complicated designs hinder an intuitive understanding and thus impede access to information on how to use APIs. Developers have to find warnings about explicitly used insecure parameters or implicitly used insecure defaults in the documentation where they might overlook this critical information. Also, user-oriented guidance showing how to realize typical use cases are missing, and developers have to search for third party information. Thus it is essential to analyze which ways are particularly suitable to bring information reliably from API providers to API consumers. However, it is not straightforward to get an overview of the many influencing aspects to classify available research results and to identify research gaps.

Based on these considerations, the author of this thesis developed a model of influencing factors on security-relevant information flows (IFSIF) in software development (cf. Figure 3.1). The model structures the field of software development, setting the focus on API producers and API consumers. This perspective allows

identifying aspects of a security API that an API producer can use to provide API consumers with security-relevant information and support.

This chapter presents the IFSIF model structure, discusses its validity, and applies it to systematize the body of knowledge about security-relevant information flows during software development. This approach identifies specific research gaps, to address the central research question to what extent information flows support software developers in using security APIs and implementing secure software.

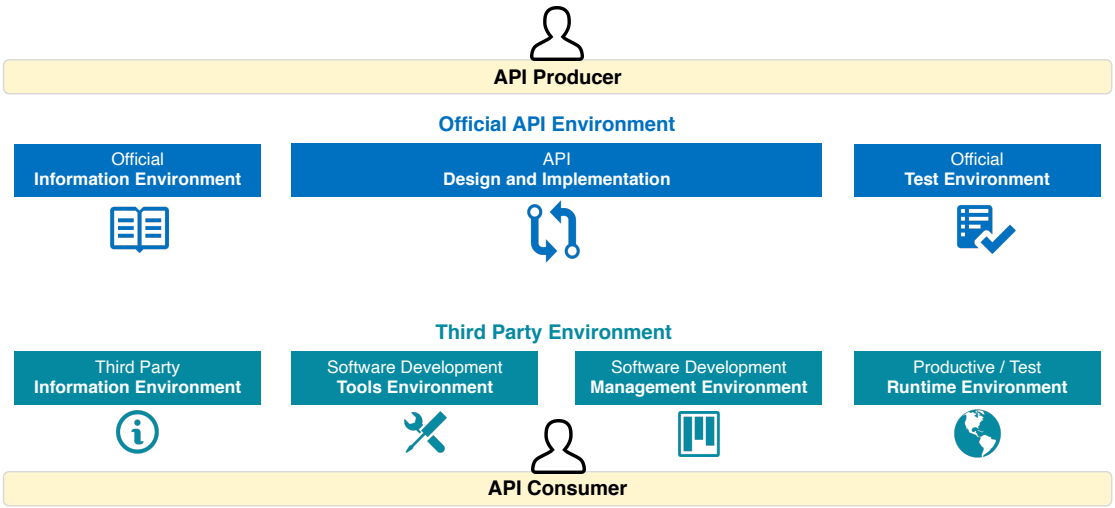


Figure 3.1.: The IFSIF Model - Model of Influencing Factors on Security-Relevant Information Flows in Software Development. The flow direction of the information is free within the model, and therefore not restricted to any direction.

3.1. Model Structure

The center of the IFSIF model illustrates communication channels locating between the API producer and the API consumer (cf. Figure 3.1), which build the official API environment, and the third-party environment.

The API producer directly influences the official API environment, which consists of three subenvironments: (1) the official information environment, (2) the API consisting of the design and the interface implementation, and (3) the official test

environment. The term “official” describes that the API producer is the source of the respective artifacts.

The model groups influencing factors on information flows by third parties into four categories. Besides the official sources, programmers also obtain information from a third party information environment. For example, in software development, developers also use search engines to obtain needed information. The Internet offers a lot of different media formats, such as websites with tutorials, videos, books, or social community platforms. Information on the use of an API can reach the consumer via such third party sources. In the implementation stage, an API consumer can use various tools to work with APIs, which are grouped by the software development tools environment. When writing code, tools such as IDEs, e.g., draw attention to information by highlighting API calls in the text editor. Static code analysis tools can be a source of such information. The software development management environment comprises all aspects of managing a software development process. Essential factors in the development management are, e.g., how teams deal with defined security requirements or potential security risks, how these are documented, evaluated, and communicated in further development processes. Furthermore, the fourth group covers aspects of the productive and test runtime environment hosting the software, which integrates an API. Security relevant information flows do not stop after deployment. The reliability of the security functions should be monitored and tested with every release update. APIs also have to communicate security-relevant status information and evaluation results during software runtime, such as a positive or negative result of certificate validation.

The flow direction of the information is free within the model, and therefore not restricted to any direction. For example, an API consumer might read the API documentation containing security-relevant information on an official website. Thus the information flows from the API provider over the official information environment to the API consumer. However, the documentation page might be auto-generated using documentation comments which are part of the API implementation. These comments can also be processed and displayed by third party tools like IDEs. An information backflow is also possible, for example, if a consumer asks the provider for help or reports a bug on a public support platform. However, information flows can become more complex when also considering detailed technical processing. Because

the model has procedural parts and can get instantiated, it goes beyond a simple taxonomy.

3.2. Model Validity

A real-world case study, e.g., the online documentation of Cryptography.io [157], is suitable for a discussion concerning the validity of the API environment. Figure 3.2 shows the detailed API Environmet of the IFSIF model.

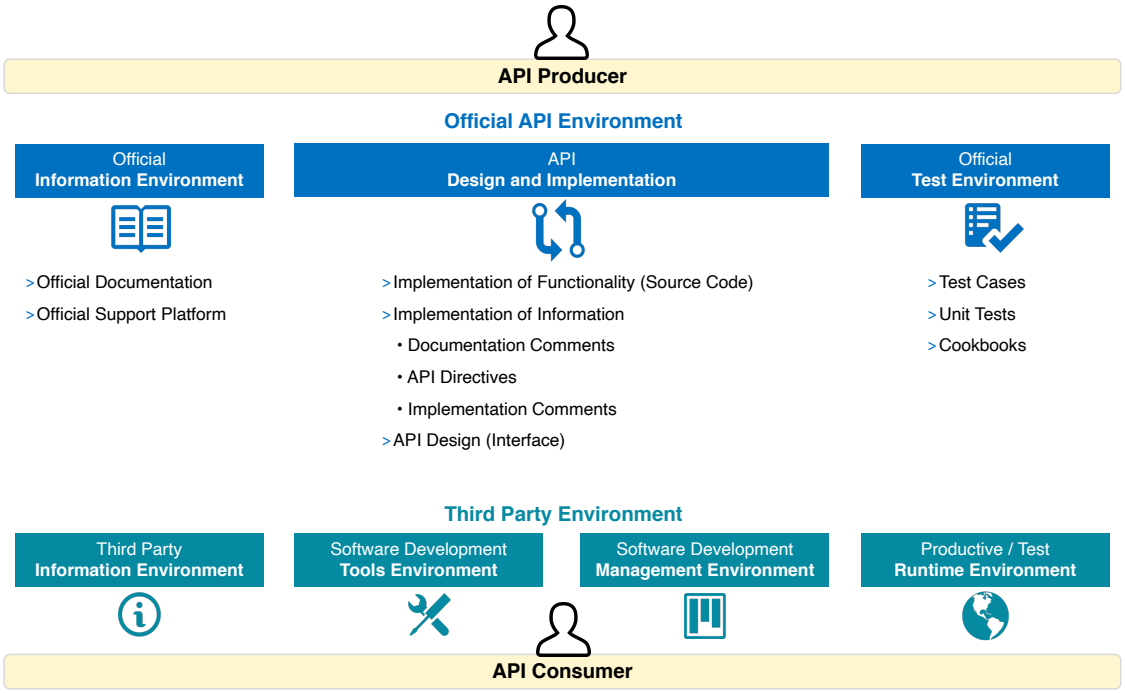


Figure 3.2.: Detailed API environmet of the IFSIF model

Cryptography.io is a cryptographic Python API, which belongs to the group of security primitives APIs (cf. Section 2.9, p. 21). Its official information environment offers online documentation as well as a support platform via Github, where also the source code of the implemented functions is accessible. The developers also write comments about implementation details into the source code files. Code comments can also contain specific usage advice, called API directives [130]. The interface design of the Cryptography.io API offers a suitable level of abstraction for

both specialized developers and non-specialized developers. It offers the so-called "hazardous materials layer" and warns in the documentation about the risk of insecure implementations and the less flexible "recipes layer" with secure defaults. The official test environment consists of unit tests, which check whether the offered functionality works as expected in the API consumer environment, and trusted test cases, also called test vectors, which prove the correctness of the implemented security mechanisms. The Cryptography.io website does not provide a separate task or problem-oriented collection of code examples, often called "cookbook." The model covers all these case study elements by the API environment.

The validity of the model is further assumed based on three perspectives. The first perspective considers the currently most used programming tools [181, 103]:

1. Source code collaboration tool
2. Standalone IDE
3. Lightweight desktop editor
4. Continuous integration or continuous delivery tool
5. Issue tracker
6. Static analysis tool
7. Code review tool
8. In-cloud editor or IDE

The third party environment covers all these tools from the top list (cf. Figure 3.3).

The second perspective considers standardized software development life cycles (SDLC) from the field of systems and software engineering. The ISO12207 standard [96] specifies with the agreement, organizational project-enabling, technical management, and technical processes four main groups. Technical processes and technical management are relevant for API usage, which are covered by the four third party environments of the IFSIF model.

The third - security-related - perspective considers approved secure software development life cycles (SSDLC) [18]. The purpose of SSDLCs is to establish security

engineering as an integral part of software development to ensure secure products. Assal and Chiasson brought together six SSDLCs from respected companies and institutions such as the Open Web Application Security Project (OWASP) and derived a list of twelve security best practices [18]:

1. Identify security requirements.
2. Design for security.
3. Perform threat modeling.
4. Perform secure implementation
5. Use approved tools and analyze third-party tools' security.
6. Include security in testing.
7. Perform code analysis.
8. Perform code review for security.
9. Perform post-development testing.
10. Apply defense in depth.
11. Recognize that defense is a shared responsibility.
12. Apply security to all applications.

These twelve security best practices can also be assigned to or integrated as IFSIF model elements (cf. Figure 3.3).

3.3. Application of the Model to the State of the Art

Figure 3.3 shows the application of the IFSIF model to the state of the art as presented and discussed in Chapter 2. Table 3.1 shows the detailed assignment of related work. The main aspects investigated in a study were taken into account in

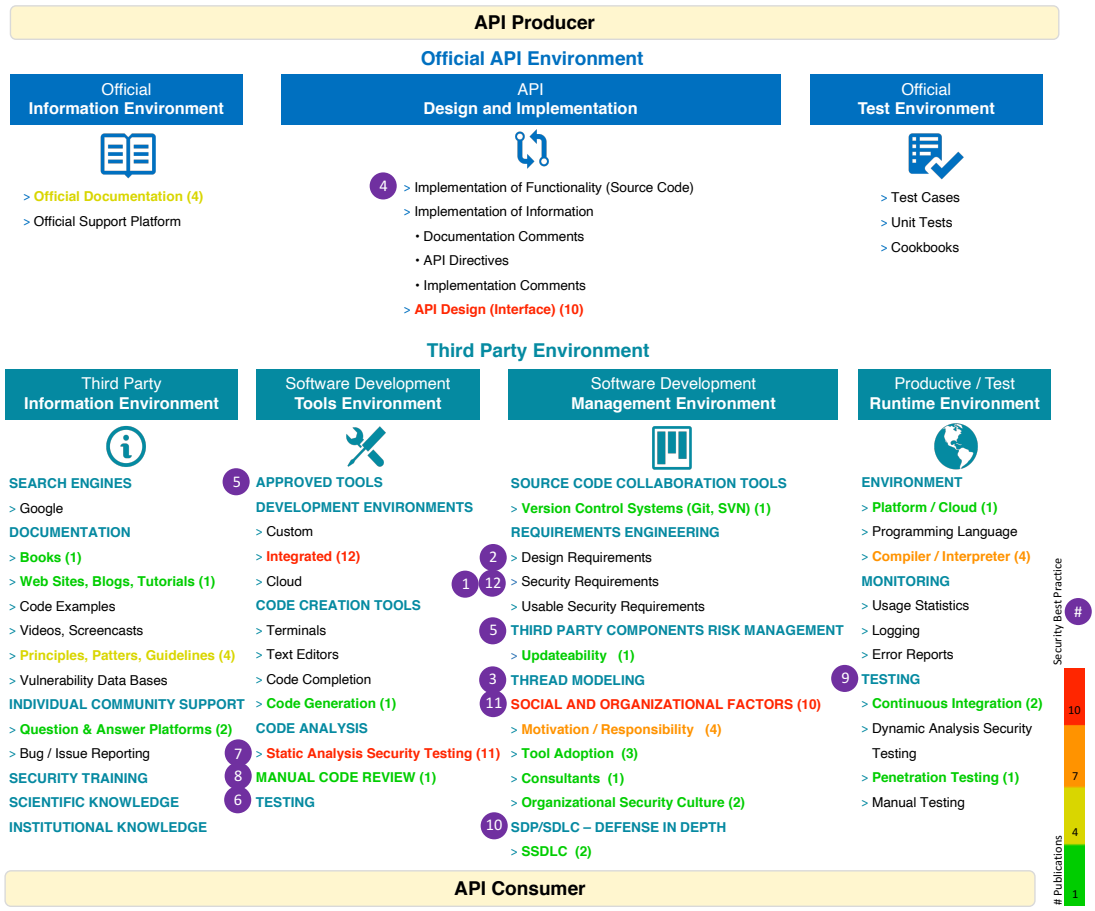


Figure 3.3.: Application of the IFSIF model to the state of the art

the assignment. One work could also be thematically assigned to more than one elements of the IFSIF model.

Quantitatively it becomes visible that previous research focused on IDEs, static code analysis, social and organizational factors, and the API design. However, results have particularly pointed out problems within the environments (cf. Chapter 2), indicating a lack of information flows. Several factors have not yet been examined concerning their security relevance. Among them, especially aspects of the official API environment where an API producer has direct influence.

Fundamentally, research has not yet investigated whether the usability of security APIs compared to the general usability of APIs places particular demands on the API design (**RQ1**). Furthermore, studies in the field of API design have concluded

IFSIF Model Element	Related Work	#
Official Information Environment		
Official Documentation	[1, 2, 3, 217]	4
API Implementation and Design		
API Design (Interface)	[2, 3, 30, 89, 92, 101, 133, 134, 136, 217]	10
Third Party Information Environment		
Books	[3]	1
Web Sites, Blogs, Tutorials	[1]	1
Principles, Patters, Guidelines	[88, 123, 229, 230]	4
Question & Answer Platforms	[3, 24]	2
Software Development Tools Environment		
Integrated	[118, 137, 190, 191, 195, 215, 214, 226, 227, 232, 233, 234]	12
Code Generation	[115]	1
Static Analysis Security Testing	[13, 17, 21, 22, 23, 45, 57, 105, 175, 176, 196]	11
Manual Code Review	[60]	1
Software Development Management Environment		
Version Control Systems	[5]	1
Updateability	[53]	1
Motivation / Responsibility	[19, 27, 211, 225]	4
Tool Adoption	[220, 221, 224]	3
Consultants	[155]	1
Organizational Security Culture	[155, 199]	2
SSDLC	[18, 19]	2
Productive/Test Runtime Environment		
Platform/ Cloud	[137]	1
Compiler / Interpreter	[25, 26, 29, 197]	4
Continuous Integration	[90, 158]	2
Penetration Testing	[199]	1

Table 3.1.: Detailed assignment of the "state of the art" to the IFSIF model.

that security APIs should primarily provide secure defaults. High API abstraction levels should avoid implementing insecure features. So far, a classification of security APIs concerning levels of abstraction is missing. Neither research of which levels of abstraction are required from the software developer's point of view has been conducted (**RQ2**). In particular, there are no research results on whether the implementation of information can promote security-relevant information flows (**RQ3** and **RQ4**). Also, there is a lack of knowledge about what information flows must be available to implement security defaults of APIs in more complex API environments, such as web development frameworks (**RQ5** and **RQ6**), in an appropriate way to support API consumers in implementing security. The studies

presented in the following chapters aim to make contributions to these identified research gaps in order to answer to what extent information flows support software developers in using security APIs and implementing secure software.

4. Security API Design

This chapter examines **RQ1** and **RQ2** regarding the design of security APIs. A fundamental prerequisite towards the general goal of developing security API usability enhancements is to understand the specific characteristics of security APIs in contrast to other APIs as well as classes of security APIs. The related work has yet provided little knowledge in this regard (cf. Section 2.7, p. 18). For this purpose, the author has designed, conducted, and evaluated two distinct studies. Study 1 develops an API usability model that also applies to security APIs and elaborates on an expanding list of high-level usability characteristics that are specific to the usability of security APIs (cf. Section 4.1). Study 2 investigates aspects of interface abstraction levels (cf. Section 4.2, p. 48). Therefore, it presents a classification of security APIs and results of an online survey, examining at what levels of abstraction software developers expect a security API to offer its security functionalities.

4.1. Study 1: Usability Characteristics of Security APIs

4.1.1. Motivation

Previous work investigating the usability of APIs has identified several aspects that API producers should consider if they want to design a usable API for their users. It is currently assumed that these aspects generally apply to many APIs. However, it is not yet clear whether the approaches from the API usability research are sufficient for the specific context of security APIs (**RQ1**). This study investigates the differences and similarities between the usability of APIs and the usability of security APIs. Figure 4.1 shows an IFSIF model (cf. Chapter 3) classifying the contributions of this study to the "official API environment." The first contribution

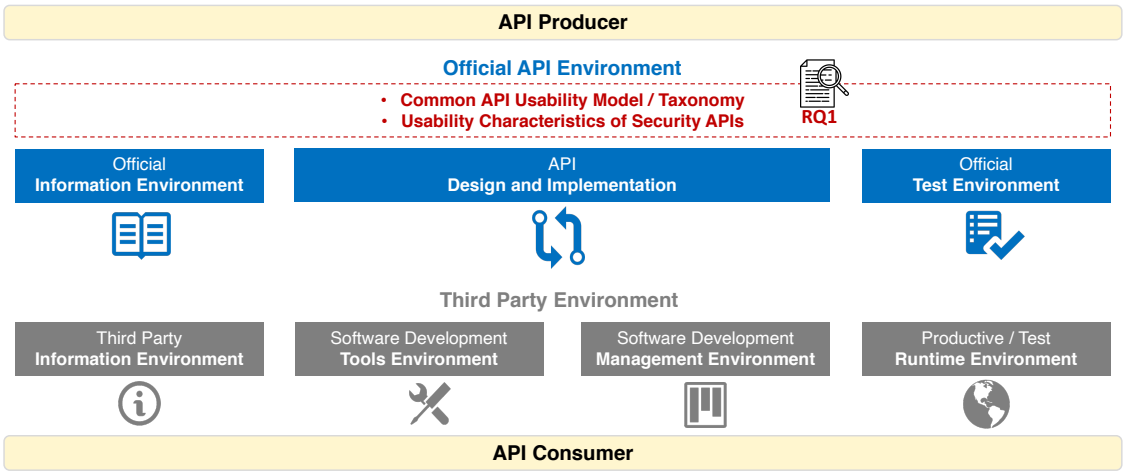


Figure 4.1.: IFSIF model classifying contributions of Study 1.

is a general API usability model (cf. Table 4.1, p. 39), which lists aspects of all three subareas: "official information environment," "API design and implementation," and "official test environment." All these environments are within an API producer's direct sphere of influence to communicate security-relevant information directly to API users. As security APIs are a particular subgroup of APIs, this model also applies to security APIs. The second contribution is a list and discussion of high-level characteristics that are specific to the usability of security APIs.[1]

4.1.2. Methodology

Two conditions need to be met to answer the research question. First, an overview of available approaches from the API usability research is required. Second, the current state of knowledge about specific requirements of security APIs has to be elaborated. To meet the first condition, the author classifies research on API usability according to (1) the particular examined API usability aspects and (2) to the examined action targets of API users. For the API context, an adaptation of the comprehensive usability model by Winter et al. [219] is used as the initial

[1]The contents of the present chapter were previously published in the paper **"Towards the Usability Evaluation of Security APIs."** by Peter Leo Gorski and Luigi Lo Iacono, which appeared in the Proceedings of the *10th International Symposium on Human Aspects of Information Security and Assurance (HAISA)*. Frankfurt, Germany: Plymouth University, 2016, pp. 252 – 265 [84].

basic structure. The classification iteratively creates an increasingly detailed picture of the current research coverage as well as on untreated topics, as a solid basis for answering the research question. The literature base consists of an extensive collection of literature [51] from the special interest group for API usability — the group formed in 2009 at the ACM CHI Conference on Human Factors in Computing Systems. In order to find further relevant publications in the area, an online search was carried out. The considered work reflects empirical research only and excludes guidelines based on expert knowledge or opinion.

To analyze and assess whether API usability approaches are sufficient for the security context, also the second condition needs to be fulfilled. Usability problems of security APIs have been identified by previous work (cf. Section 2.9.2, p. 23). In addition, research results on applied software security give valuable insights into the practical requirements of security APIs. Thus, the author applied a bottom-up approach to retrieve specific characteristics of security API usability. Focusing on widely deployed security mechanisms, which are relevant for many developers, the author selected studies of the SSL/TLS protocol [160], the OAuth 2.0 Framework [161], and OpenID Connect [166]. Thus, the review consisted of ten security analyses published between 2012 and 2015, which did not study malicious attacks but logic errors of applied software security, relating to usability shortcomings in API design.

4.1.3. Results

The results of this literature study are a taxonomy of API usability in the form of a model, and a list of eleven usability characteristics of security APIs.

API Usability Model

The API usability model (cf. Table 4.1) has a two-dimensional vertical structure. Regarding the usability framework defined by the ISO standard 9241-11 [97], a user's interaction with an API is influenced by the product and the context of use. The focus of this study lies on the product, which is the API environment that API producers design. However, an extension of the model for the context of use was also developed (cf. Table 4.2). Following the usability model approach by Winter et al.

[219], the product is differentiated between the physical interface (1.1) and the logical architecture (1.2). The documentation (1.3) as an essential component of an API completes this second structure level. Aspects examined in the individual studies determine the entries of the vertical axis from the third level onwards. The author has also added aspects for which no study results are available so far. In the third level, e.g., the space of API design decisions (1.2.3) has been integrated, which was introduced by Stylos and Myers [186]. As specified by the ISO standard 9241-11 [97], the context of use covers the users (2.1), the goals and tasks (2.2), the resources (2.3), and the environment (2.4). The model's horizontal structure consists of action targets while a user interacts with an API (A-K). Also, for this horizontal axis, action targets examined in the individual studies determine the entries. The author supplemented the three hitherto unconsidered action targets "form an intention," and "find an API," as these indicate potential research aspects. Since software development often includes secondary tasks like security, the goal "form an intention" describes how, e.g., frameworks can make a software developer aware of relevant security aspects to trigger an initial intention to use an API. One dimension turned out to be appropriate for classifying action targets. Empty table cells are aspects where research results are missing. Available API usability recommendations are represented by positive (+), negative (−), positive and negative (±), or neutral (○) impact indicators. These are strongly related to the usability context of an empirical study [x]. These are documented as follows [219] (cf. Appendix A):

[aspect|property] → effect [interaction|metrics] [empirical study]
Example:
[Code examples|existence] → + [Learn an API|duration] [124]

The elaborated model visualizes the contemporary space of API usability. It has not an immutable structure. Instead, this is the current state of the research field, which can be enhanced by future work or results that have been missed by this study. Only a few early studies have minor points of contact with security APIs. Ellis et al. [63] evaluated the usability of the Factory pattern in API design. In one of the experiment tasks, the participants were instructed to instantiate an SSLSocket using the Java Standard Edition (SE) API version 1.5. Important subsequent tasks,

Product/Aspects \ Interactions/Action targets	A) Form an intention	B) Find an API	C) Explore the API	D) Find class, method, etc.	E) Select class, method etc.	F) Creating code	G) Find example	H) Understand API	I) Debug code	J) Learn an API	K) Maintain an API
1. Product											
1.1 Physical interface											
1.2 Logical architecture											
1.2.1 Form of appearance											
1.2.1.1 Libraries											
1.2.1.2 Toolkit											
1.2.1.3 Framework											
1.2.1.4 Web-APIs											
1.2.2 Programming languages											
1.2.2.1 Idioms											
1.2.3 API-design decisions											
1.2.3.1 Structural design											
1.2.3.1.1 Design patterns											
1.2.3.1.1.1 Factory patterns						−[63]					
1.2.3.1.2 Package design											
1.2.3.1.2.1 Number of classes				−[170]							
1.2.3.1.2.2 Sub packages				+[170]							
1.2.3.1.3 Configuration-based design											
1.2.3.1.3.1 Annotations						+[172]					
1.2.3.1.3.2 File-based						+[172]					
1.2.3.1.3.3 Fluent interfaces						+[172]					
1.2.3.1.3.4 Combinations											
1.2.3.2 Class design											
1.2.3.2.1 Class names				+[187]				+[151]			
1.2.3.2.2 Design patterns											
1.2.3.2.2.1 Create-set-call			+[184]								
1.2.3.2.3 Method placement				−[187]							
1.2.3.2.4 Number of methods				−/○[170]							
1.2.3.2.5 Method names				+[170, 171]				+[151]			
1.2.3.2.6 Method overloads				−[171]							
1.2.3.2.7 Parameter design				+[187, 151, 59]		−[170]					
1.2.3.2.8 Exceptions								−[59]			
1.2.3.2.9 Object creation											
1.2.3.2.9.1 Default constructors						+[184]					
1.2.3.2.9.2 Optional constructors						○[184]					
1.2.3.2.9.3 Required parameters						−[184]○[151]			○[184]		
1.2.3.2.9.4 Static methods						−[170]					
1.2.3.2.10 Access rules						+[235]					
1.2.4 4 Implementation of the functionality											
1.2.4.1 Performance											
1.2.4.2 Reliability											
1.2.5 Runtime behavior											
1.3 Documentation											
1.3.1 Form											
1.3.1.1 Written documentation										+[235, 164]	
1.3.1.2 Examples						±[34]				+[124]±[164]	
1.3.1.3 Runnable tests											
1.3.1.4 Comments in source code							+[124]		+[151]		
1.3.1.5 Web resources						±[59]					
1.3.2 Content											
1.3.2.1 Design concept					+[164]						

Legend: + positive effect − negative effect ± both pos. and neg. effect ○ neutral [Reference] in the usability context of the empirical study

Table 4.1.: API usability model - part 1: The API product

Product/Aspects \ Interactions/Action Targets	A) Form intention	B) Find an API	C) Explore the API	D) Find class, method, etc.	E) Select class, method etc.	F) Creating code	G) Find example	H) Understand API	I) Debug code	J) Learn an API	K) Maintain an API
2. Context of use											
2.1 Users											
2.1.1 User types											
2.1.2 Skills and knowledge											
2.1.3 Personal attributes											
2.1.3.1 Programming style		±[49]								±[49]	
2.1.4 Expectations											
2.1.4.1 Mental models								+[186]			
2.1.4.2 Conventions										+[124]	
2. Goals and tasks											
2.2.1 Security-critical requirements											
2.3 Resources											
2.3.1 Development environment											
2.3.1.1 Operating systems								+[171]			
2.3.1.2 IDEs										±[34]	
2.3.1.3 Web resources											
2.3.2 Development tools											
2.3.2.1 Debugger				+[171, 37]		+[131, 142]					
2.3.2.2 Auto completion											
2.3.2.3 Text editor											
2.3.2.4 Web-search							+[34]±[185]				
2.3.2.5 Recommendations		+[58]		+[16]	+[231]						
2.4 Environment											
2.4.1 1 Organizational environment											
2.4.1.1 Development processes											
2.4.2 Technical third party environment											
2.4.2.1 Online availability											
2.4.2.2 API evolution guidelines											+[64]
2.4.3 Physical environment											
2.4.4 Social environment											

Legend: + positive effect − negative effect ± both pos. and neg. effect ○ neutral [Reference] in the usability context of the empirical study

Table 4.2.: API usability model - part 2: The API context of use

such as certificate validation, have been out of focus. Five of twelve participants failed to complete the task in the given time. Thus, Ellis et al. [63] concluded that the Factory pattern hinders the usability of an API. This result provides evidence that general API usability research also applies to security API usability. Still, the Factory pattern is the design of the choice to construct SSLSockets in the latest Java SE version 13. A web authentication task has been part of a user study conducted by Stylos and Myers [187]. They used a self-modified version of the Apache Axis2 API [11] in order to focus on specific user behavior with optional API classes. However, they do not particularly mention the security context in the study results. This study emphasizes that general API usability findings are also adaptable to security-specific contexts. These studies allow the conclusion that available research results, because of their essential nature and relevance for various APIs, also build a fundamental basis for the usability of security APIs.

Since detailed knowledge about the usability of APIs in the security context is currently lacking, the API usability model cannot yet be supplemented or extended more specifically for security APIs. However, the results of security studies allow concluding general usability characteristics of security APIs. These should be investigated more closely in usability studies. Future results can then be incorporated into the model.

Usability Characteristics of Security APIs

When analyzing the outcomes of recent security studies in the light of API usability, it becomes clear that current API usability research already provides some baseline approaches and tools to gain usability for security APIs. However, this baseline is not sufficient for the security context. Thus, the subsequent paragraphs introduce eleven security API specific usability characteristics.

1. End-user Protection The insecure usage of security APIs in software implementations can compromise users' data security, often without the users even noticing. Thus, especially security APIs must be designed while keeping end-user security in mind because they are the final users of the functionality of an API. The "End-user Protection" characteristic describes an API's ability to reduce or eliminate

this dependency on programmers. Fahl et al. [66] proposed the "User Protection" characteristic and have defined it as a limitation of a developer's capabilities to prevent an invisible risk for end-user data. This definition bases on the observation that developers of mobile applications take full responsibility for integrating security functionalities as well as for communicating any security-relevant information to end-users [65, 66].

Georgiev et al. [76] have identified similar issues for various SSL/TLS libraries, software development kits, and middleware. Wang et al. [207] refer to the due diligence of application developers who implement relying-party components in single sign-on systems. According to Wang et al. [208], application developers are finally responsible for organizing user applications, relying-parties, and identity providers in a secure manner. Nevertheless, this is also true for programmers who implement libraries, software development kits, or frameworks. Incorrect handling of tokens caused by unusable security APIs, in any of those software products, could lead to unauthorized access to user accounts even without possessing any credentials. Thus, due diligence exists for all persons involved in software development processes to ensure end-user protection.

2. Case Distinction Management Error prevention and the handling of exceptions and errors are crucial aspects of APIs in general but are vital for security APIs. Thus, API producers should thoroughly design a "Case Distinction Management" to support API users considering API events adequately. In the context of security APIs, exceptional events like, e.g., a negative certificate validation, require attention. These are no failures preventing security measures, but obligatory validation results in preserving security. Because such validations happen frequently, API producers and API users should not treat them as rare exceptions or software errors. These cases have to be well managed by an API design that enables developers to handle case distinctions accurately.

For instance, the verification process of certificates is a crucial part of the SSL/TLS protocol for establishing a trust relationship between a client and a server. Critical processes are chain-of-trust verification, hostname verification, and reviewing the certificate status. Georgiev et al. [76] found that security-critical events are communicated inconsistently by runtime errors, return values, or internal

flags, which API users have to validate by additional code. Such designs already resulted in ineffective security measures in deployed software.

3. Adherence to Security Principles More than forty years ago, Saltzer and Schroeder [167] described fundamental principles of information security, which are still approved and prevailing. Since then, further area-specific principles evolved, such as the "OWASP Secure Coding Practices" [143] or documented risks like the "CWE/SANS Top 25 Most Dangerous Software Errors" [194]. Adhering to these security principles when designing security APIs will help software developers to use security measures effectively.

Security studies found several examples of API design decisions violating well-documented security principles. One of them was the Android SSL/TLS library [81] that partially contradicted the "economy of mechanism" principle. Android applications are usually exchanging data with just a few well-known hosts, but the Android system trusted over 100 Certificate Authorities (CA) by default. Mechanisms like a certificate or public key pinning, which allow API users to precisely select needed CAs, had to be self-implemented. Thus, the API forced users to take a higher security risk by default. Security studies proved that certificate or public key pinning was not in widespread use for Android [65] or Web applications [111].

4. Testability The security studies that this analysis is built upon are prime examples for how difficult it is for common software developers to test security mechanisms in their applications. Much effort and expertise is needed to develop test beds for static code analysis and conduct manual code audits. Still, software developers need to see clearly if security mechanisms have been adopted, integrated and deployed correctly and this needs to be examined not only by self-written unit test code. Due to a lack of time and expertise or sometimes also the blind faith, some developers do not test integrated libraries or used frameworks at all [66]. Not less badly, even modified code for testing purposes finds its way in deployed software products, causing security flaws [76]. Supporting and reliable test routines, written by security experts, e.g., for certificate validation and adversarial testing in TLS implementations [36], should be available and easy to apply for programmers in typical use-cases.

5. Constrainability It is the nature of programming to customize code in order to meet various requirements. However, customization in the context of security causes serious risks. Functionalities, like data validation, represent essential constraints establishing security [109] against, e.g., cross-site scripting. Georgiev et al. [76] stated: *"In general, disabling proper certificate validation appears to be the developers' preferred solution to any problem with SSL libraries."* These findings seem to legitimate constraining users in using security APIs, indicating a tradeoff between flexibility and error susceptibility. If customization tends to be the rule rather than the exception, though, the design decisions of API providers are presumably not appropriate for their target users and should be reviewed. Further usability evaluations have to examine situations where usage constraints support or hinder the usability of security APIs.

Instead of writing source code, the configuration of security mechanisms might be a useful instrument to force constraints. Fahl et al. [66] have proposed such an approach for SSL/TLS development on Android. However, no empirical evidence is available, proving this to be an appropriate approach for usable security APIs. One example showing the opposite is HTTP Strict Transport Security (HSTS) [162]. A crucial task for applying HSTS is configuring the HTTP header. Kranich et al. found that deployed headers were not standard conform, were malformed, and misused values, mostly undermining data security of end-users [111]. This finding highlights that there are usability aspects to be considered in the configuration that has not yet been understood and investigated.

6. Information Obligation API providers have to communicate security-relevant specifics to API users accurately. A significant challenge is to provide relevant information at the right place, in the right moment, and a usable manner [74]. If an application does not provide any protection means for confidentiality, e.g., ignoring SSL/TLS connections, an end-user will not notice this situation, due to a lack of information. The same is true for security API users if API producers do not communicate security implications via components of their API environment.

7. Degree of Reliability Interviews with developers [66] who implemented security mechanisms incorrectly found that application developers coping with security-

related programming tasks need support from reliable information resources and APIs. When running into problems or knowledge barriers, programmers will consult Web resources. However, developers should not generally trust proposed problem solutions without review, as they might come from, e.g., equally inexperienced developers. Therefore reliable testing tools, as well as visible trust indicators, preferably issued by reputable institutions, are needed. Future work should examine what kind of resources application developers trust. One approach could be to measure a developers's self-assessed level of confidence in their security-relevant implementations. The results should indicate which instance should primarily deliver approved information or well-tested code examples for various use cases to meet the developers' expectations.

8. Security Prerequisites To provide security functionality effectively, API users have to meet mandatory prerequisites of security APIs. These are typically unknown, unclear, or misused. Sun et al. found that relying-parties implementing OAuth 2.0 missed implementing confidentiality for the protocol via SSL/TLS [188]. Wang et al. [208] unveiled shortcomings in correctly protecting and verifying tokens in single sign-on systems. They suspected missing comprehension of security implications to be the reason. Li and Mitchell [116] were able to identify deficiencies against Cross-Site Request Forgery (CSRF) attacks in deployed services caused by misused parameters. Developers used static and guessable values instead of random and unique character sequences. API providers have to meet their obligation to inform and support their users to prevent security risks caused by missed security prerequisites.

9. Execution Platform Several different execution platforms require security functionalities from security APIs. However, they have to be precisely adapted to the requirements of the respective software ecosystem, including existing platform-specific development opportunities and risks in particular. Wang et al. [208] traced back software vulnerabilities to API designs, not considering execution platform specifics. Using the OAuth 2.0 client-flow in web browsers, e.g., exposes tokens to various browser-specific attack vectors. Thus, Sun and Beznosov [188] "believe that OAuth 2.0 at the hand of most developers – without a deep understanding of web security – is likely to produce insecure implementations." Chen et al. [44]

called attention to sensitive differences between mobile and web platforms because of uncovered problems in adapting OAuth for mobile applications. The engineers of SSL/TLS intentionally designed the protocol for browser environments. Its widespread deployment for transport security in non-browser applications such as Android [65, 207], iOS [66], and other platforms [76] leads to widespread man-in-the-middle vulnerabilities, potentially affecting millions of end-users. Concluding on these results, security API design processes have to consider target execution platforms, including their user needs. The central question is how security-relevant information can be communicated during development processes.

10. Delegation API producers should not delegate tasks of implementing security functionalities to non-security experts. Already mentioned findings show that shifting responsibility to implement security lead to incorrect implementations. Georgiev et al. [76] found several SSL/TLS libraries delegating hostname verification or certificate validation to higher-level software. Brubaker et al. [36] even encountered missing code in an essential "if" condition, which just provided a comment to API users. Such design decisions are especially critical if API users assume a complete security solution and instead get an incomplete product. In such cases, API users have to get well informed about being responsible for completing open tasks to fulfill security prerequisites. Even better would be if API producers would suggest concrete solutions or responsible best practices.

11. Implementation Error Susceptibility Future research on security API usability should pursue the goal of minimizing the error susceptibility of security APIs. They can reduce security risks by addressing each characteristic above.

4.1.4. Discussion

The study results answer **RQ1** of this thesis. Approaches from the API usability research are not sufficient for the specific context of security APIs. Nevertheless, available research results from the general field of API usability build a fundamental basis for the usability of security APIs. Thus, one contribution of this study is an initial consolidated proposal of common usability aspects that API producers need

to consider when designing APIs for security mechanisms. The API usability model enables easy access to the field for novices, it is easily extendable, and it allows the uncovering of open research questions. In particular, it draws a fragmented picture of API usability, demanding further research.

The introduced eleven specific usability characteristics of security APIs might still be an incomplete set of relevant topics. Future research should aim to confirm or revise the set in order to obtain a validated baseline for the usability evaluation of security APIs. However, security API producers can consider the proposed eleven conceptual usability characteristics for their API design. To improve information flows between an API producer and an API consumer, most likely, more detailed and actionable advice is needed. Research is required here to elaborate concrete usability aspects, with a lower level of abstraction, like those listed in the space of API usability.

The next study in this thesis, therefore, examines whether a security API design with different levels of abstraction can achieve some of the developed usability characteristics.

4.2. Study 2: Abstraction Levels of Security APIs

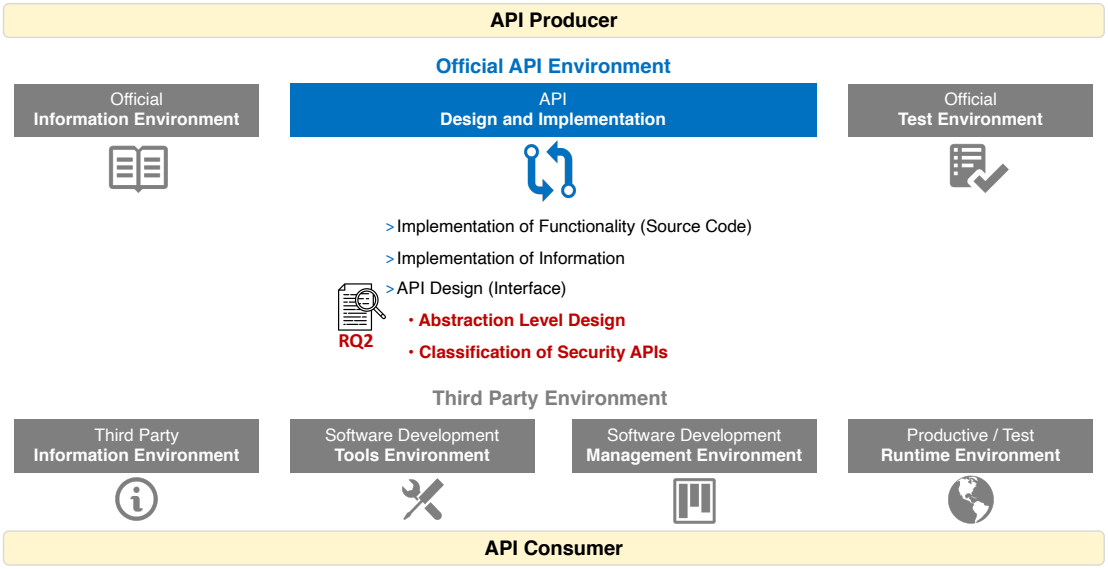


Figure 4.2.: IFSIF model classifying contributions of Study 2.

4.2.1. Motivation

A promising approach for usable security APIs, addressing at least three usability characteristics (cf. Section 4.1.3, p. 41), is to implement several API versions with different levels of abstraction. The Python library Cryptography.io (cf. Section 3.2, p. 29), e.g., consists of two versions, the so-called "hazardous materials layer," and the "recipes layer." This cryptographic API does not delegate tasks that require detailed security knowledge to users (cf. Section "10. Delegation", p. 46) like selecting secure algorithms and parameters or taking care of prerequisites (cf. Section "8. Security Prerequisites", p. 45) like random number generation. Instead, the API producers implemented besides a hard-to-use API version an easy-to-use "recipes layer" that implements security best practices by default. This API version constrains an API user (cf. Section "5. Constrainability", p. 44) to avoid insecure API usage. The critical question is, at this point, what requirements professional developers have for using security APIs and whether it would be appropriate to have an API that only offers one level of abstraction.

Most of the research available so far is mainly targeting APIs for cryptographic schemes or protocols (cf. Chapter 2).[2] However, the field of security APIs covers much more than cryptographic APIs [188, 198]. Consequently, a security API classification is required to structure this field and guide further research. It can help to systematically analyze security APIs, especially in the light of their usability, as it would allow identifying target user groups for different types of APIs. Software developers with a specialization in cryptography or security might be able to use any security API but may also prefer a specific class of APIs. On the other hand, regular developers might require a certain minimum level of abstraction in order to fit their language and mental models.

So far, professional developers have not been asked for their opinion or requirements on abstraction levels of security APIs. Therefore, **RQ2** of this thesis is: *What level of abstraction do software developers prefer when working with security APIs?* An IFSIF model (cf. Chapter 3) shows that this study makes contributions to the interface level of an API design (cf. Figure 4.2). The study's first contribution is a classification of the various security APIs, which was already introduced and explained in Section 2.9.1 (p. 21). It considers the level of abstraction as the primary classification criteria. In order to answer the research question, this study will assess (1) whether developers desire the classification to be fully available, and (2) if there exists a sufficiently diverse coverage of security APIs in the wild to satisfy the security needs of regular software development projects. Thus, the first step is a quantitative analysis of provided security APIs in the most popular programming environments. The results should provide qualitative insights on which security APIs are ready to use for developers out of the box. The second contribution is a questionnaire-based online study gathering an opinionated view of developers on the abstraction level design of security APIs.

[2]The contents of the present chapter were previously published in the paper **"I Do and I Understand. Not Yet True for Security APIs. So Sad."** by Luigi Lo Iacono and Peter Leo Gorski, which appeared in the Proceedings of the *2nd European Workshop on Usable Security (EuroUSEC)*, co-located with the 2nd IEEE European Symposium on Security and Privacy. Paris, France: Internet Society, 2017. [122].

4.2.2. Methodology

Security API Classification Development

The security API classification in Section 2.9.1 (p. 21) is based on an analysis of standards related to security mechanisms as well as various catalogs of security patterns [15, 71, 31, 228] and security controls [14]. An elaborated set of unique security mechanisms, each mentioned at least once in the analyzed publications, has determined the structure of the classification. However, it does not show a comprehensive picture of the domain, but the main structure that classifies the field according to the abstraction level that each type of security API addresses.

Popular Programming Environments Analysis

Not only the landscape of programming languages is vast (cf. Section 2.6.1, p. 16), but also software development kits available to developers are numerous as there have been many developed for each programming language. The analysis of popular programming environments took place in 2017. Using the IEEE ranking of most popular programming languages in 2017 [42], a top ten list was determined and assigned. Depending on which applications are typically developed with each language, they were assigned to the categories mobile, web, embedded, and enterprise. From a conceptual and technology perspective, enterprise and web software development have a lot in common. Also, embedded and mobile development share programming languages. Thus, the market research analysis considered native programming language support for security only once. Also, for each language and software category, one comprehensive and representative development framework was selected. The analysis first identified what security mechanisms were "built-in" and could potentially be used by API users without the need to search for additional security APIs components. The availability of such available extensions provided by third parties was evaluated in a second step.

Survey Design

The author conducted the online survey in March 2017 using a self-hosted Lime Survey server [117]. The main goals were to (1) validate the usefulness of the

introduced classification, (2) gather insights about what level of abstraction software developers wish to have when using security APIs, and (3) to identify possible discrepancies between developers' needs and circumstances in practice.

The survey consisted of 19 main questions plus three conditional questions, depending on the answer. Appendix B contains the full questionnaire. As it focused on qualitative results, the survey contained eight open questions that did not provide the respondents with answers. The question design considered related work. Robillard [164] surveyed to identify existing learnability issues of APIs in general, and Nadi et al. [133] conducted two surveys with a focus on usability issues of Java cryptography APIs.

The first five questions (Q1-Q5) collected demographical data, including the country of residence, occupation, development experience, what types of software the respondents developed, and what programming languages they used. Their familiarity with the security domain was evaluated on the one hand by asking for an educational background in security (Q6) and, if this was the case, for specific aspects of this education (Q6a). It should be analyzed whether there is a transitive connection between education, struggling with security APIs, and the demanded level of abstraction. On the other hand, it was accessed how often security mechanisms are used (Q7).

In the questionnaire's core, Q8 first asked who is, in the respondent's opinion, responsible for integrating security mechanisms in software systems. This question should assess whether there was a general awareness for the responsibility of API producers and developers of tools and documentation. The questionnaire asked in Q9 for currently applied security mechanisms and in Q10 for a usage ranking of prominent features offered by security primitives APIs as well as security controls APIs, to verify and give further substance to the classification proposed in Section 2.9.1 (p. 21). In order to identify specific aspects of security APIs and their usability, the survey asked in Q11 whether there are typical work steps to implement security mechanisms and in Q12 if there is any difference between security-related programming tasks and non-security related tasks. Additionally, to implicitly find hints to the needed level of abstraction of security APIs, Q13 asked whether a developer ever had problems implementing security mechanisms and, if this was the case, for specific information and reasons (Q13a and Q13b). Q14 asked which

level of abstraction a security API should offer to meet development needs. The applied differentiation between the levels of abstraction is based on three distinct software developer personas proposed by Clarke [184, 49]: Opportunistic (high), Pragmatic (medium), Systematic (low). Q16 asked what kind of security API is more appropriate. Besides, the author was interested in whether the participants would recommend any API, tool, or information resource to a peer or friend who struggles with implementing a security mechanism (Q15).

The survey ended with the possibility to give further comments, thoughts, or suggestions (Q17) and to provide an email address (Q18) for receiving results or invitations to further studies (Q19). A separate database stored answers to these last three questions to keep survey answers anonymous.

Recruiting

Executives or heads of department of various software development companies personally known to the author and peers were asked to forward a survey invitation to in-house software developers. Additionally, personally known programmers have been invited to fill the questionnaire. The author received 59 full responses, 4 of them invalid. Thus, the following analysis bases on $n = 55$ answers in total.

Limitations

The sample of the conducted online survey was mostly limited to software developers from Germany. However, the gained data was appropriate to achieve the above-defined objectives of this survey. Besides, the survey, as well as the market analysis, took both place in 2017. The results described may be subject to change over time.

4.2.3. Results

Market Analyis Results

The outcomes from the analysis of available security APIs in popular programming environments (cf. Table 4.3) emphasize that security APIs are directly available to developers as security primitives APIs dominate “built-in” functions. The identified security mechanisms that are built-in and which developers can potentially use

without requiring to search for appropriate components are denoted by a filled circle (●). A half-filled circles (◐) indicat externally available functions provided by third parties. An empty-circle symbol (○) shows cases where neither a built-in nor an external component was found.

	Cryptography	*Steganography*	*Watermarking*	*Authentication*	*Authorization*	*Secure Communication*	*Filtering*	...
	Security primitives APIs			Security controls APIs				
Embedded								
C/C++	●	◐	○	◐	◐	◐	○	
Mobile								
Java/Android	●	◐	○	●	●	●	●	
C#/Windows Phone	○	◐	○	○	◐	○	○	
JavaScript/Hybrid	◐	◐	○	◐	◐	◐	●	
Objective-C/iOS	●	◐	○	◐	◐	●	●	
Swift/iOS	●	◐	○	◐	◐	●	●	
Enterprise								
C/C++/STL	◐	◐	○	●	●	◐	◐	
Java/JEE	●	◐	○	●	●	●	●	
Python	●	◐	○	●	◐	●	●	
R	◐	◐	○	◐	◐	○	●	
C#	◐	◐	○	○	◐	○	○	
Ruby	●	○	○	◐	◐	◐	●	
Go	●	◐	○	◐	◐	◐	◐	
Web								
Java/Spring	●	◐	○	●	●	●	●	
Python/Django	●	◐	○	●	◐	●	●	
C#/ASP.NET	●	◐	○	●	◐	●	●	
PHP/Symfony	○	◐	○	●	◐	●	●	
JavaScript/Meteor	◐	◐	○	●	◐	●	●	
Ruby/Ruby on Rails	●	○	○	●	◐	●	●	
Go/Revel	●	◐	○	●	◐	●	●	

Table 4.3.: Available security APIs in popular programming environments in 2017. The built-in availability is denoted as ●. Third party security functionalities are denoted as ◐. If no according security APIs are available, ○.

As can be seen, developers - whether security-experienced or not - most commonly have to deal with basic cryptographic functionalities. Typically web engineering environments include some security controls by default, based on this application domain's years of risk experience associated with distributed systems on the web. There is an apparent demand to close this gap in other domains and to expand the set of security controls APIs.

Online Survey Respondents

Forty-six of the participants (83%) came from Germany. The other nine answers came sparsely from eight different countries, including Austria, France, Mexico, Netherlands, Norway, Republic of Korea, Switzerland, and the USA.

The recruiting gained answers from 38 participants (69%) who described themselves as industrial and eight (15%) as freelance developers. One industrial researcher, one undergraduate student, one graduate student, and five people with other occupations responded. One participant preferred not to answer. Furthermore, the sample represents mostly experienced developers. Almost half of the participants (45%) have more than ten years of experience in software development followed by 25% between five and ten years, 18% between two and five years, and the remaining 11% less than two years (cf. Figure 4.3).

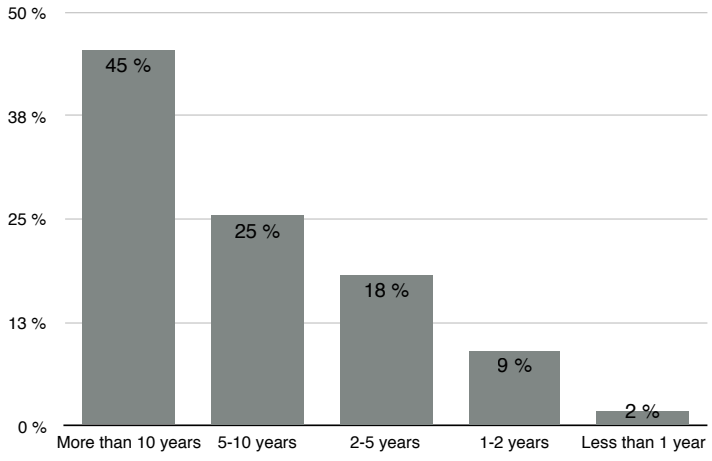


Figure 4.3.: Programming experience of the study participants. © Internet Society

The sample represents developers of all five main software domains. However, the participants' focus lies with 85% on web applications as well as enterprise applications with 58% (cf. Figure 4.4).

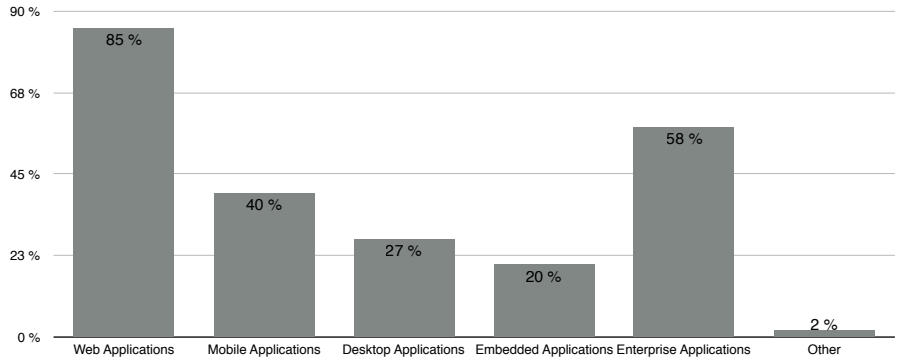


Figure 4.4.: Types of software which the participants develop. © Internet Society

As previously discussed in Section 4.2.2 the set of programming languages, a developer is mainly using, depends on the types of software she is developing. Consistently to the distribution of software types, participants name a wide range of languages with a strong emphasis on Java and JavaScript (cf. Figure 4.5).

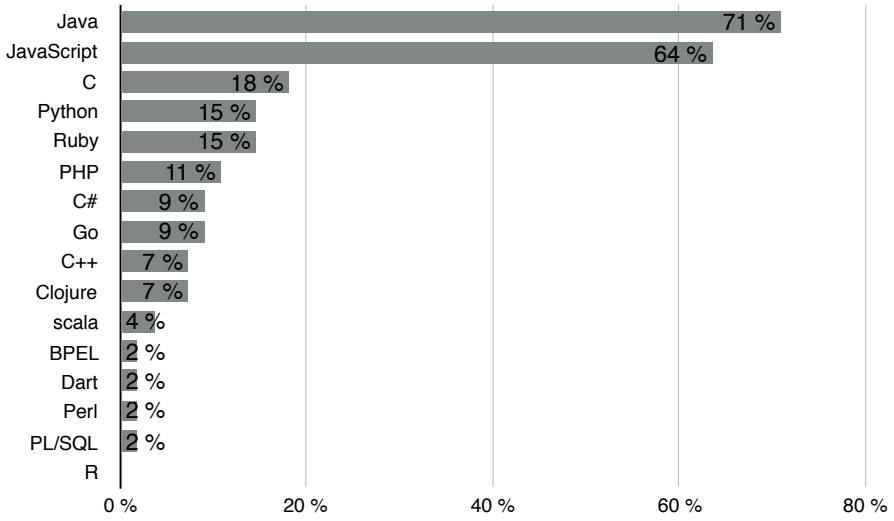


Figure 4.5.: Primarily used programming languages by the participants of Study 2. © Internet Society

In summary, the survey participators have much experience programming various types of software with a broad set of programming languages. This result gives evidence that the survey reached a relevant target group.

Online Survey

The structure of this section follows the defined objectives in Section 4.2.1 (p. 48). First, the introduced classification from Section 2.9.1 (p. 21) is assessed. The results give evidence that it can be applied to structure the field of security APIs. The given free-text responses to Q9, which security mechanisms the respondents generally implement, can almost entirely be assigned to either security primitives APIs or security controls APIs by using the proposed classification (cf. Figure 2.3, p. 22) for a content-related coding. Therefore, mechanisms like protection against SQL injection or cross-site scripting vulnerabilities were assigned to "Escaping" or, for instance, cross-origin resource sharing to "Whitelisting." Named end-user software products have been ignored in this process. Furthermore, the answers reflect mainly the determined rating options in Q10 and thus support their validity.

Derived from these ratings, developers have to use security mechanisms most often for (1) filtering functionalities like input validation, named first by 42%, and second by 22% (1^{st}: 42%, 2^{nd}: 22%), for (2) authorization and authentication for access control (1^{st}: 25%, 2^{nd}: 33%), and (3) mechanisms for secure connections and communication (1^{st}: 22%, 2^{nd}: 16%). These three groups of security control functionalities take in total 89% of the first rank (1^{st}) and still 71% of the second rank (2^{nd}). The comparison of total ratings (cf. Figure 4.6) considering selections for all fifteen ranks further strengthens this result. Most of the participants also generally use cryptography like encryption and decryption (67%) and have to store credentials like user names and passwords securely (73%).

However, over 80% of the respondents are using these rated security mechanisms just occasionally. Twenty-seven participants (49%) use security mechanisms rarely in less than 33% of their development tasks. Nineteen persons (35%) use security mechanisms occasionally in more than 33% but less than 66% of their tasks. Only nine participators (16%) have to deal frequently with security in more than 66% of their tasks. With no exception, all participants have to integrate security

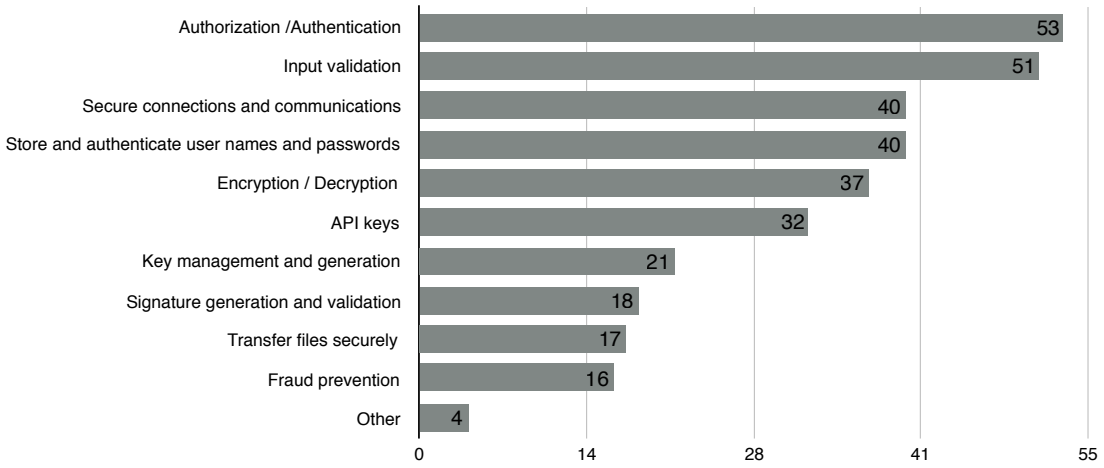


Figure 4.6.: Ranking of security-related functions most commonly used by study participants in their software projects. © Internet Society

mechanisms in their code. This result indicates that most participants have other primary programming tasks but also have to implement security functionalities from time to time.

The results from Q9 and Q10 have a clear tendency to security control functionalities. Also, if a developer just rarely or occasionally uses this kind of APIs, she will likely make more errors with low-level designs that require detailed background knowledge. API producers have to consider this target group in the abstraction level design of security APIs. Only 5% of software developers who took part in the survey want to determine implementation details by working with a low-level API (cf. Figure 4.7). 31% agree with not being interested in implementation details and only want the security to work by using a high-level API. 53% prefer a medium level API, which should offer low as well as high levels of implementation details and opportunities, depending on the actual programming task. 11% did not agree with the offered statements and chose other wordings.

Q16 directly asked which kind of security API the developers do find more appropriate for their needs. The questionnaire explained the used terms as follows: Security APIs can be grouped into cryptographic APIs (e.g., encryption algorithms, hash functions, and digital signatures) and security controls APIs (e.g., security protocols and mechanisms for authentication or authorization). 53% answered both,

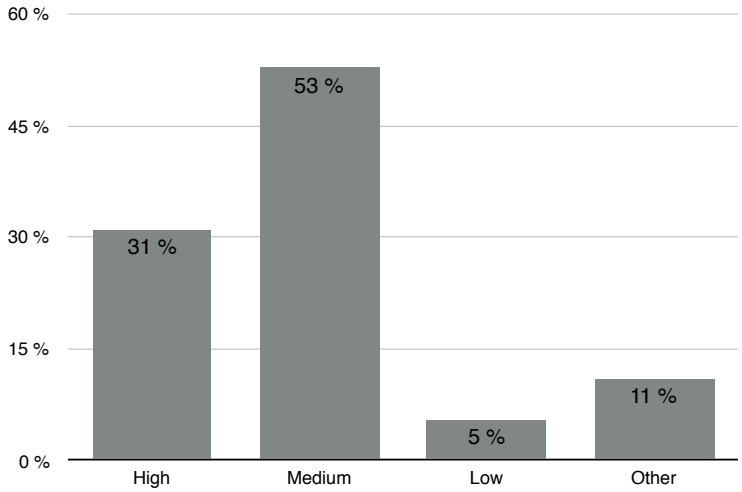


Figure 4.7.: Demanded security API abstraction level. © Internet Society

36% security controls APIs, 7% cryptographic APIs and 4% none (cf. Figure 4.8). There is an obvious tendency towards security controls APIs. Also, expert users demand more abstract controls. A reason might be that they appreciate the shortcuts when developing frequent standard protections. Still, in the answers to Q15, the given recommendation on particular security APIs, do not show such a tendency. Only nine recommendations included security controls APIs, ten pointed to cryptographic APIs and 11 participants would recommend general information resources. The other respondents did not give any recommendation.

Thirty-four respondents (61%) encountered problems while implementing security mechanisms. Seven respondents generally stated they had problems just "doing it right". Eleven others explained in more detail that they have struggled to understand or use an API: *"Some security libraries are not user-friendly. Hence, people tend to just get it running due to time restrictions. By this, they may break the actual security mechanism, making the whole usage of the library senseless."*

Other respondents reported issues in their answer to Q13 concerning the understanding of underlying security concepts (four times), the complexity of concepts (four times), and finding best practices as well as secure implementations (three times). The participants saw different root causes for these problems: (1) Themselves, as the human factor, having little experience (five times), or making insufficient

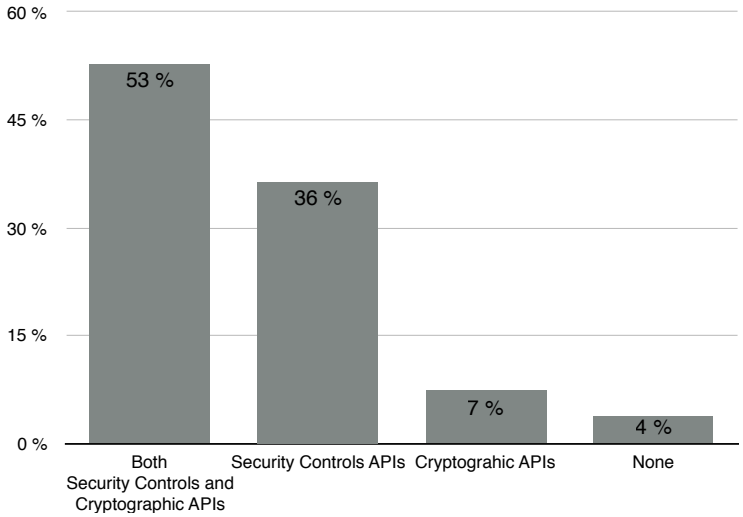


Figure 4.8.: The participants' needs for security APIs. © Internet Society

efforts (three times). (2) Other developers, as nine respondents determine the design of an API, library, or documentation as the reason for their problems: *"Libraries that are not well documented (missing documentation or very detailed documentation that is for the praxis irrelevant, we need HowTos that for the most common scenarios allow a fast implementation)."* (3) The organizational environment, as it is responsible for a tight timescale (named four times) and not counting security to business priorities (named three times). (4) The complexity of security concepts, again, was the last issue mentioned by five developers as the root cause.

In this context, also a connection between the educational security background and having problems with the adoption of security APIs was examined. The relation between participants having some educational security background (58%) and those without background (42%) is almost balanced in the sample. Those 32 persons who answered yes were additionally asked for particular aspects. As expected, the answers do not contain complete handbooks of modules but indicate a broadly based educational landscape. Most named topics range from cryptography and encryption (20 times), network security and security protocols (8 times), vulnerabilities and attacks (6 times), security services (6), public key infrastructures (4 times), practical countermeasures (4 times) and best practices and guidelines (3 times).

Although most software developers have to integrate security mechanisms in their software, a large number do not have an educational background in security. The design of high or medium-level security APIs should especially consider this finding to lower the initial hurdle for programmers with minor experience in security. This demand is especially urgent for frequently used functionalities like input validation and sanitation, authorization and authentication, as well as for secure connections and communications. Nevertheless, there was no statistically significant correlation between previous knowledge from education and having problems when implementing security mechanisms (Chi-squared (χ^2), p=0.493).

With (1) the developer himself, (2) other developers, and (3) the organizational environment, different reasons for problems with security APIs have been reported. The question arises who is responsible for integrating security mechanisms in software systems in the first place (Q8). The participants' opinions are widely dispersed, ranging from just "the developer" (11 times) or just "the software architect" (5 times) to "everybody" (8 times) or "the entire team" (3 times). Others named specialists in the domain of security (3 times) or concerning project knowledge ("*The developer or software architect who knows the software in depth.*"). Twenty-three answers describe multiple groups of persons. In summary, this diverse groups include - in addition to software developer, software architect and specialist - software designer, tester, operator, customer, business analyst, quality analyst, requirements engineer, group leader, software integrator as well as language designer (named once) and framework provider (named twice).

As the development of complex software systems may follow various software engineering processes involving many stakeholders with different competencies, it can not be generally stated who takes responsibility for bringing security measures into software finally. The results show that software developers mostly see themselves, at least partially, responsible for the integration. However, they do also rely on other entities, such as security APIs offering security functionalities, as this telling example in the web framework AngularJS illustrates: *"Each version of AngularJS 1 up to, but not including 1.6, contained an expression sandbox, which reduced the surface area of the vulnerability but never removed it.* ***In AngularJS 1.6, we removed this sandbox as developers kept relying upon it as a security feature even though it was always possible to access***

arbitrary JavaScript code if one could control the AngularJS templates or expressions of applications. *[10]"*

The following question (Q11) asked in which work steps developers should be supported to solve a security-related task. The obtained responses to Q11 give valuable insights to this. Some very technical responses being not suitable for this evaluation were excluded. Other answers were clustered into several groups. Thirteen different answers describe the need to specify requirements in the first step. As software developers have an information need when beginning a programming task, six participants would start with research to understand a security mechanism and to get information about their current development status.

Additionally, seven respondents would work out a risk or threat model. In all steps, 20 participants (36%) mentioned the searching for best practices and available tools like libraries, frameworks, or security APIs. After creating a concept (named seven times) and consulting other persons (named ten times), developers would implement the measure (named 14 times). In all steps, 32 participants (58%) would write and perform tests like functional unit testing or fuzzing. Testing is also relevant for already existing systems which have to be analyzed (named four times) or tested for bug fixing or patching after release. As a second line of validation, ten developers would request a code audit or review after the implementation.

The responses to Q12 show that developers have controversial opinions concerning the question if there is a difference between security-related programming tasks and non-security related tasks. The difference is argued mainly by the security context. Summarizing, developers describe security as a non-functional requirement that brings additional complexity, requires different thinking, specialized domain knowledge, more effort, attention as well as care and which results in more severe consequences. These might be reasons for a typical negative perception of security-related tasks: *"Security-related tasks are not welcomed by sponsors, tend to have poor management attention and often feel like a burden. They tend to require more attention to the review and quality assurance because it is often more complex to prove that they fulfill their purpose."*

A different perspective, which is shared by 14 participants, is relativizing the opinion above. From their point of view, all tasks are security-related or require the same conceptual approach. One respondent chose the following wording: *"I think*

most tasks that are involved in creating a software system do involve security to a certain extend. Having a strict separation between security and non-security tasks is a reason that a lot of systems do have security holes after their release." Whether there is a difference between security-related programming tasks and non-security related tasks or not, both described paradigms demand due diligence when dealing with security-related implementation tasks, because of the critical implications.

4.2.4. Discussion

Discussion of an ideal state

All kinds of security APIs deserve comprehensive research and development in order to improve their usability for software developers. Still, according to the study results, further research needs a sophisticated look at the field of usable security APIs. As the introduced classification advocates for, there should be at least a distinction of security APIs in security primitives and security controls. This diverse view supports also distinguishing various distinct groups of software developers. For the ones being inexperienced and unknowing in respect to security, the focus should be to develop appropriate and sufficient security controls APIs. This class offers first of all the demanded level of abstraction that is also closest to the users' language and mental models. Most security controls are made up of a composition of various security primitives and may hide the involved complexity behind a more graspable API. By this, security controls APIs do not interfere as much with the primary goal of developers - i.e., finalizing the software - as security primitives APIs do since the latter requires an in-depth understanding of the involved concepts, algorithms, and relationships amongst the security primitives. Such expertise and knowledge that requires a whole lot of experience will remain the competence of a specialized and henceforth rather small - in comparison with the general case - group of developers. A specific focus on security controls APIs seems to open the path for promising approaches towards usable security APIs.

Discussion of the actual situation

The answer to **RQ2** is, although most of the participants were not security experts, that developers expressed a definite requirement for a security API to provide both high-level and low-level interfaces. Developers desire to flexibly use an API according to requirements of their development projects. This demand generally sounds like a desirable characteristic for a software development tool. In consequence, neither security controls APIs nor primitive APIs should provide their users only with high-level interface designs that are fail-safe to use, but constrain application and, thus, likely also usability. Much more balanced designs of different levels of abstraction that support the API user in implementing various use cases are necessary. Furthermore, the actual situation is critical, with missing security controls. Consequently, regular developers have to continue coping with security primitives APIs to meet their programming tasks.

Thus, two findings suggest that further research needs to address the usability of error-prone security primitives APIs. First, the current limited availability of controls APIs forces developers who are no security experts to cope with error-prone security primitives APIs. Second, security primitives APIs are also the building blocks for future security controls APIs. Usability issues of security primitives APIs must not compromise the reliability of security controls APIs. For this reason, the following chapter contributes to the improvement of information flows between API producers and API users.

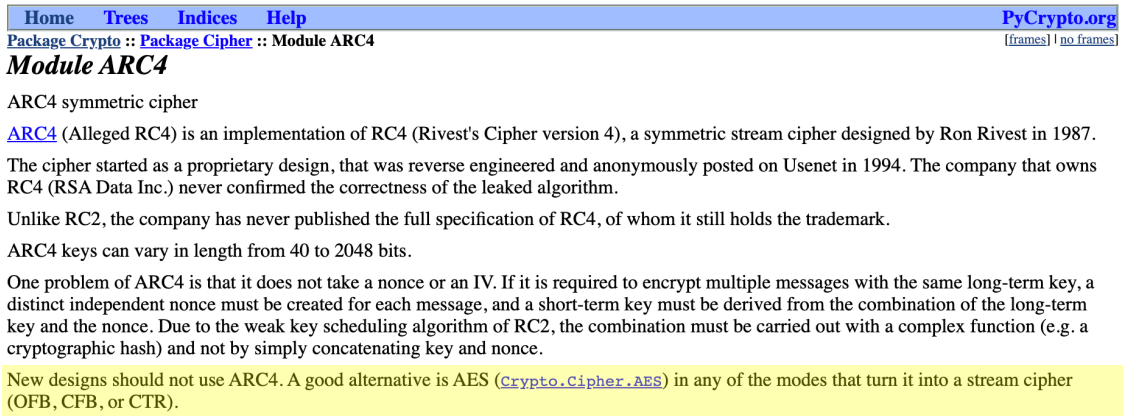
5. Security API Warnings

The previous two studies have shown that the security context places particular demands on an API. For example, the API producers have to provide API users with security-relevant information to support them in developing secure software for end-users. Furthermore, security primitives APIs, which tend to be utilized insecurely by average users due to a lack of usability, are far from being used only by specialized developers.

Many cryptographic APIs offer insecure features and function calls for compatibility reasons, e.g., to support outdated cryptographic algorithms or parameters. So far, programmers are not informed or warned when using such API functionalities. Warnings that point the developer to an insecure API use or potential risks in program code play an important role in secure software development research (cf. Chapter 2). However, API producers currently communicate warnings against the use of certain API functions mainly through documentation. These are not always in the focus of API users or security-relevant information locates in places that can easily be overlooked as an example of the PyCrypto documentation [119] in Figure 5.1 illustrates. The recommendation at the end of several paragraphs to avoid using the RC4 algorithm is not highlighted in the original online documentation.

Clearly identified as a research gap by applying the IFSIF model to the state of the art (cf. Section 3.3, p. 31) and further motivated by the results of the previous Chapter 4, this chapter examines **RQ3** and **RQ4** regarding the effectiveness and design of security APIs warnings. Thus, it explores a specific aspect of security-relevant information flows in software development to improve security API usability.

In Study 3, the author initially designs, implements, and examines a security API warning prototype conducting a between-subjects online controlled experiment with 53 experienced Python developers (cf. Section 5.1). Study 4 aims to revise and improve this first design approach developed in Study 3 (cf. Section 5.2, p. 90). It



Home Trees Indices Help PyCrypto.org

Package Crypto :: Package Cipher :: Module ARC4 [frames] | no frames]

Module ARC4

ARC4 symmetric cipher

ARC4 (Alleged RC4) is an implementation of RC4 (Rivest's Cipher version 4), a symmetric stream cipher designed by Ron Rivest in 1987.

The cipher started as a proprietary design, that was reverse engineered and anonymously posted on Usenet in 1994. The company that owns RC4 (RSA Data Inc.) never confirmed the correctness of the leaked algorithm.

Unlike RC2, the company has never published the full specification of RC4, of whom it still holds the trademark.

ARC4 keys can vary in length from 40 to 2048 bits.

One problem of ARC4 is that it does not take a nonce or an IV. If it is required to encrypt multiple messages with the same long-term key, a distinct independent nonce must be created for each message, and a short-term key must be derived from the combination of the long-term key and the nonce. Due to the weak key scheduling algorithm of RC2, the combination must be carried out with a complex function (e.g. a cryptographic hash) and not by simply concatenating key and nonce.

New designs should not use ARC4. A good alternative is AES (`Crypto.Cipher.AES`) in any of the modes that turn it into a stream cipher (OFB, CFB, or CTR).

Figure 5.1.: Screenshot of the PyCrypto API documentation describing an implementation of the encrytion algorithm RC4 [119]. The recommendation at the end of several paragraphs to avoid using the RC4 algorithm is not highlighted in the original online documentation.

gathers further knowledge about the dimensions, aspects, advantages, and disadvantages of API warnings from a software developer's perspective by conducting four focus groups with a total of 25 professional software developers.

5.1. Study 3: Security API Warning Prototype

5.1.1. Motivation

This study develops and evaluates a novel approach that supports API users with security warnings presenting context-sensitive documentation and code examples as part of an API.[1] This contribution is called security advice and is integrated into an APIs implementation, as the IFSIF model in Figure 5.2 illustrates, e.g., by using logging mechanisms that developers utilize for testing. The information flow is directed to a console or terminal displaying API warnings during software runtime. A terminal or console is commonly a window that offers a command-line interface.

[1] The contents of the present chapter were previously published in the paper **"Developers Deserve Security Warnings, Too: On the Effect of Integrated Security Advice on Cryptographic API Misuse."** by Peter Leo Gorski, Luigi Lo Iacono, Dominik Wermke, Christian Stransky, Sebastian Möller, Yasemin Acar, and Sascha Fahl, which appeared in the Proceedings of the *14th Symposium on Usable Privacy and Security (SOUPS)*, co-located with USENIX Security '18. Baltimore, MD, USA: USENIX Association, 2018, pp. 265 – 281 [85].

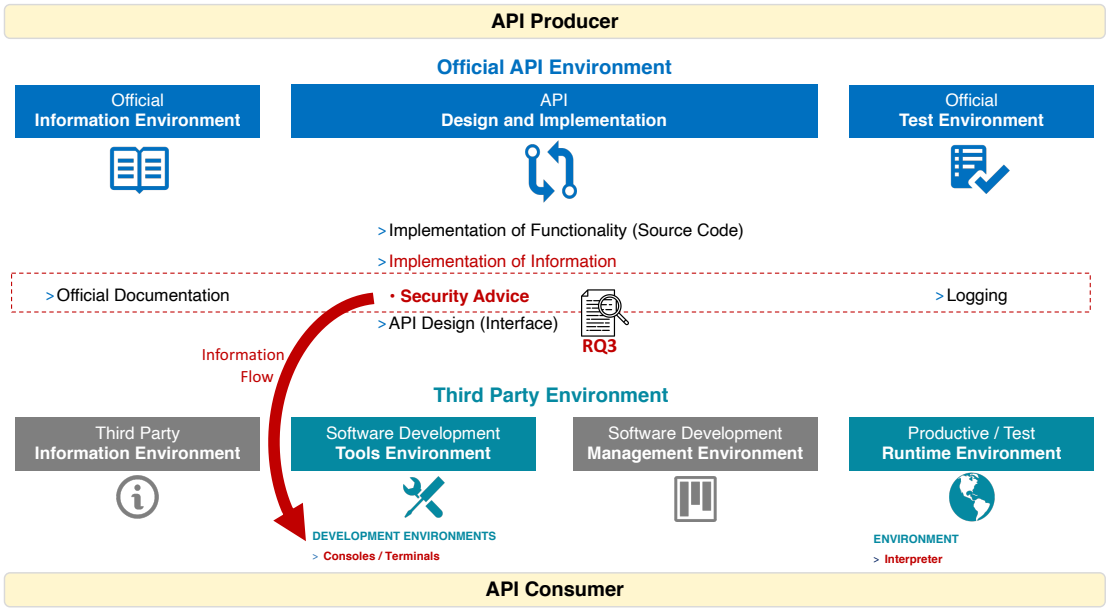


Figure 5.2.: IFSIF model classifying contributions of Study 3.

It receives user commands in text form and returns feedback in text form as well. Terminals and consoles are standard parts of a development environment. To the best of the author's knowledge, in contrast to related work, this study introduces and studies security advice as part of an API for the first time.

The approach of implementing security advice directly into an API has several advantages. From a technical point of view, API warnings do not impose conditions on their environments because they dynamically integrate into the API user's programming environment. They can provide helpful information in IDEs as well as in command line terminals, unaffected by the way compilers or interpreters translated a programming language into machine code. Extensions and plugins for IDEs or editors (e.g., [115, 137]) do not have this property. Third party tool developers have to adapt their software to the development environments of their users instead. An API producer can use API warnings to give immediate feedback to an API user pro-actively, including contextual security-relevant information and secure code examples for code improvement. Such an approach has the potential to support developers if they overlook security-relevant information in the API documentation and before they decide to consult insecure online resources [3]. API

warnings also have advantages in the question of how to get them to the users. They are directly available through the regular distribution channels of an API. Thus, users immediately benefit from feedback integration after using an updated API version.

Thus, this study focuses on the challenges of using cryptographic APIs securely. The author designs and implements an API-integrated security advice concept to support API users in writing secure cryptographic code. This novel approach for security APIs allows API providers of existing and future cryptographic APIs to improve software security. Therefore they do not have to change their interface design, rely on the development and application of plugins for IDEs, or hope that the third party information environment will improve in giving reliable advice. The author implemented his approach for PyCrypto, a cryptographic API for the programming language Python, and conducted a between-subjects online controlled experiment with 53 experienced Python developers.

The study answers **RQ3** *if API-integrated security advice has a significant effect on code security and perceived API usability.* Analyzing all changes made to the code after security advice had been shown, results show that the approach had a significant positive impact on 73% of participants who had written insecure code in their first attempt to solve programming tasks. API warnings were able to support them in improving their code. While security significantly improved, no statistically significant impact on the perceived usability could be found.

5.1.2. Security Advice Prototype Design

In contrast to end-user research, this work is the first to investigate a security warning concept targeted at software developers. Thus, when designing security warnings with a specific focus on supporting developers to make secure programming choices, there is no previous work to build on. Therefore, the security advice prototype design is based on lessons learned from research on end-user security warnings.

Sunshine et al. [189] investigated and improved the effectiveness of SSL warnings but found that many participants clicked-through a warning. They also recommended reducing their occurrence. Felt et al. found that opinionated design significantly improved user adherence rates to SSL warnings in Google Chrome [68,

70, 7]; however, they could not significantly improve users' comprehension of warnings. Weinberger and Felt [210] ran a field study to investigate how long the Chrome browser should store users' decisions for SSL warnings to minimize the effect of habituation. Similarly, Vance et al. [202] conducted a functional magnetic resonance imaging (fMRI) experiment to study warning message habituation. Both studies conclude that the risk of habituation decreases after one week. Almuhimedi et al. [8] investigated factors that contribute to why Chrome users click-through their malware warnings and find that familiarity with a website had a significant impact on users' click-through behavior. Egelman et al. [62] investigated the difference between passive and active warnings against phishing attacks and found that active were more successful than passive warnings. Bravo-Lillo et al. [35] designed and tested multiple attractors for security warnings.

Primarily, Bauer et al. [28] provided and discussed a comprehensive set of design guidelines for security warnings with a focus on end-users. The general and abstract principles of the guideline were also assumed to be appropriate for initially designing developer-centered security warnings. Thus, their design goals were evaluated to be suitable to the context of developer-centered security warnings and then applied to the API-integrated security advice concept of this study. The prototype also considers lessons learned from previous developer-centered secure programming studies [2, 3, 5, 26]. Figure 5.3 shows a wireframe of the resulting concept aiming for five design goals. Although it is a contribution of this study, the online experiment does not compare different approaches. Study 4 (cf. Section 5.2, p. 90) develops a participatory design for cryptographic API warnings.

Goal 1 - Follow a Consistent Layout In contrast to security warnings for graphical user interfaces, an API-integrated security feedback mechanism that relies on the output capabilities of command-line interfaces such as terminals or consoles only allows limited user interaction. Interface elements like buttons are hardly available in these environments. For this reason, the design can not consider layout and guideline aspects, given by Bauer et al. [28], addressing this kind of control elements. However, following the example of Bauer, a layout concept aims to provide API users with a consistent look and feel when dealing with API security warnings in independent situations. Figure 5.3 illustrates all seven sections of an API security

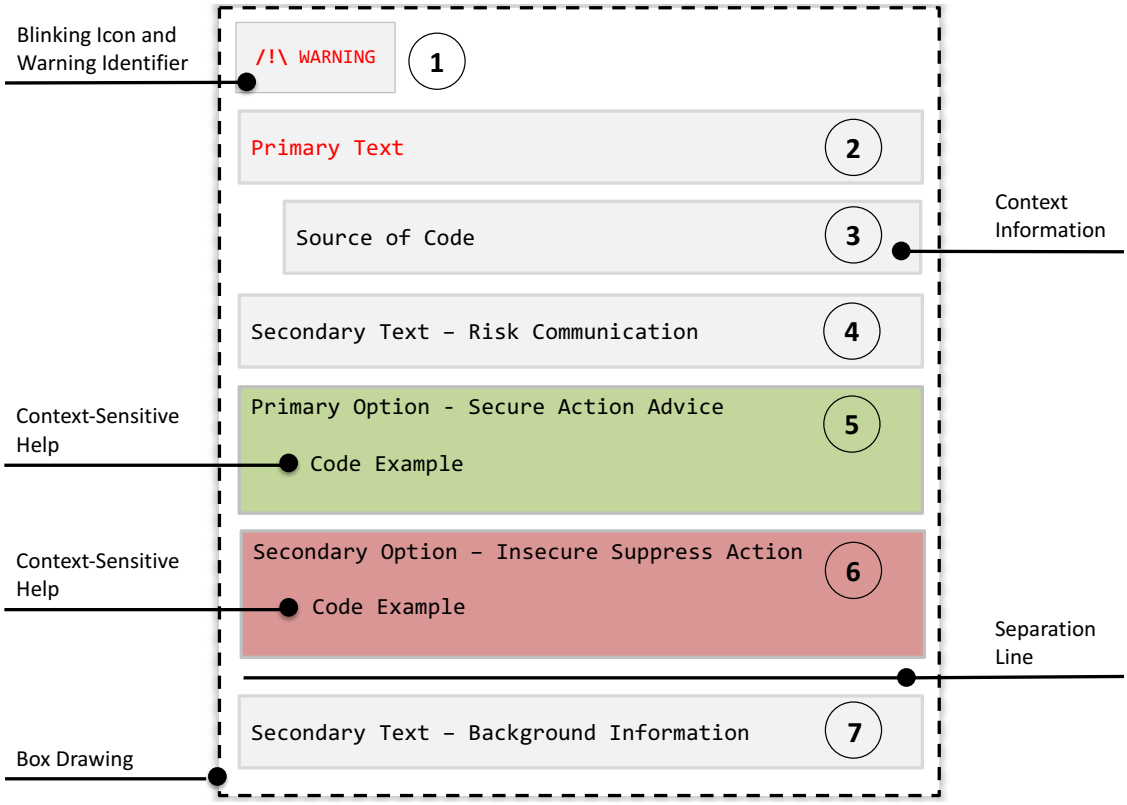


Figure 5.3.: Wireframe of the design concept for API-integrated security feedback. © The USENIX Association

warning. The elements in the upper left corner (1) indicate a dangerous situation. Section (2) gives a brief description of the security warning's root cause and section (3) provides the developer with the file and line number of the warning's origin. Section (4) explains the risk arising from the insecure API call. API users get context-sensitive and actionable advice in sections (5) and (6). Section (5) provides advice to improve code security, while section (6) shows an opportunity to turn off the warning feature. Section (7) gives links to sources giving further background information.

Goal 2 - Describe the Risk Comprehensively The design aims to clearly and comprehensively communicate the underlying risk to the developer. In contrast to TLS warnings [68] or Android permission dialogs [69], an API security warning does not have to deal with false positives. Even in cases when developers made

insecure choices intentionally, e.g., for backward compatibility requirements of legacy systems, a security warning is still a true positive.

Sections (1), (2), and (4) communicate the respective risk to the developer. Section (1) uses a red flashing text icon "/!\", indicating a warning sign. Additionally, the word "WARNING" in capital letters and red color is part of section (1). Section (2) uses red-colored text explaining the root causes of the warning, e.g., "You are using the weak encryption algorithm RC4 (aka ARC4 or ARCFOUR)". Additional details to communicate the existing risk and its potential consequences are provided in section (4). In case of an RC4 warning, e.g., "The use of ARC4 puts the processed data's confidentiality at risk and may lead to data disclosure."

Goal 3 - Present Relevant Contextual Information The design presents relevant contextual information, including the specific location in the source code that triggered the security advice. This information helps API users to identify the insecure API call that should be improved. In addition to the filename and line number, section (3) includes a snippet of the source code that triggered the warning.

Goal 4 - Offer Meaningful Options The most crucial aspect of a security warning is to offer meaningful options to solve the problem that triggered the warning. The API user is expected to either modify code and fix a security issue or suppress the warning message for future runs (i.e., click through the warning). Section (5) provides a secure code snippet to turn the insecure code into a secure code, and section (6) offers an insecure option, which disables this specific security warning. Additionally, the warning provides links to more background information in section (7) like OWASP or NIST guidelines for secure programming.

Goal 5 - Be Concise and Accurate The guideline of Bauer et al. [28] focuses on the design of end-user warnings and recommends to avoid technical jargon. However, since the warning addresses software developers, this recommendation was partially ignored. Technical jargon from the software development domain, such as specific names, locations, and values of source codes, are common elements for developers. Thus, such terms are part of the warning message. Terms, concepts, technologies, and standards from the cryptography domain, however, can not be expected to be

general knowledge of a developer [2]. Hence, cryptographic jargon is omitted as much as possible.

5.1.3. Methodology

The methodology of the study is presented below:

Prototype Implementation

The author implemented the design concept of the API security advice (cf. Section 5.1.2) to the open-source API PyCrypto for the programming language Python. Acar et al. [2] found that developers using this API are likely to produce functionally correct but insecure code. Furthermore, in another developers study [5], more than 30% of 307 participants preferred PyCrypto over other cryptographic APIs for Python. Thus, the author supposed a high potential for improving this API with the integrated security advice approach. Figure 5.4 shows a security warning example triggered by using the insecure RC4 algorithm for symmetric encryption.

Since Python has a large user base [77, 139] and supports different fields of application (cf. Section 2.6.1, p. 16), it was considered to be suitable for recruiting many developers. PyCrypto belongs to the group of security primitives APIs (cf. Section 2.9.1, p. 21). Thus, it provides low-level interfaces for symmetric as well as asymmetric encryption, supports multiple hashing algorithms, and utility features. The developers of PyCrypto used code comments for auto-generating API documentation [119], which provides detailed information about each API class and module, including minimal code examples, and they give a general overview of the API [120]. For encryption, the documentation recommends the Advanced Encryption Standard (AES) and provides an example. However, it also describes the Data Encryption Standard (DES), which is now considered insecure, as cryptographically secure. However, this is likely because the API had last been updated in 2014. The documentation formulates warnings against using the exclusive-or (XOR) function: *"Do not use it for real applications!"*, and recommends not to use the RC4: *"New designs should not use ARC4."* (cf. Figure 5.1, p. 65). However, the documentation does not comment on implementing the Electronic Code Book (ECB) mode, or an empty initialization vector (IV), as insecure defaults [126, 50].

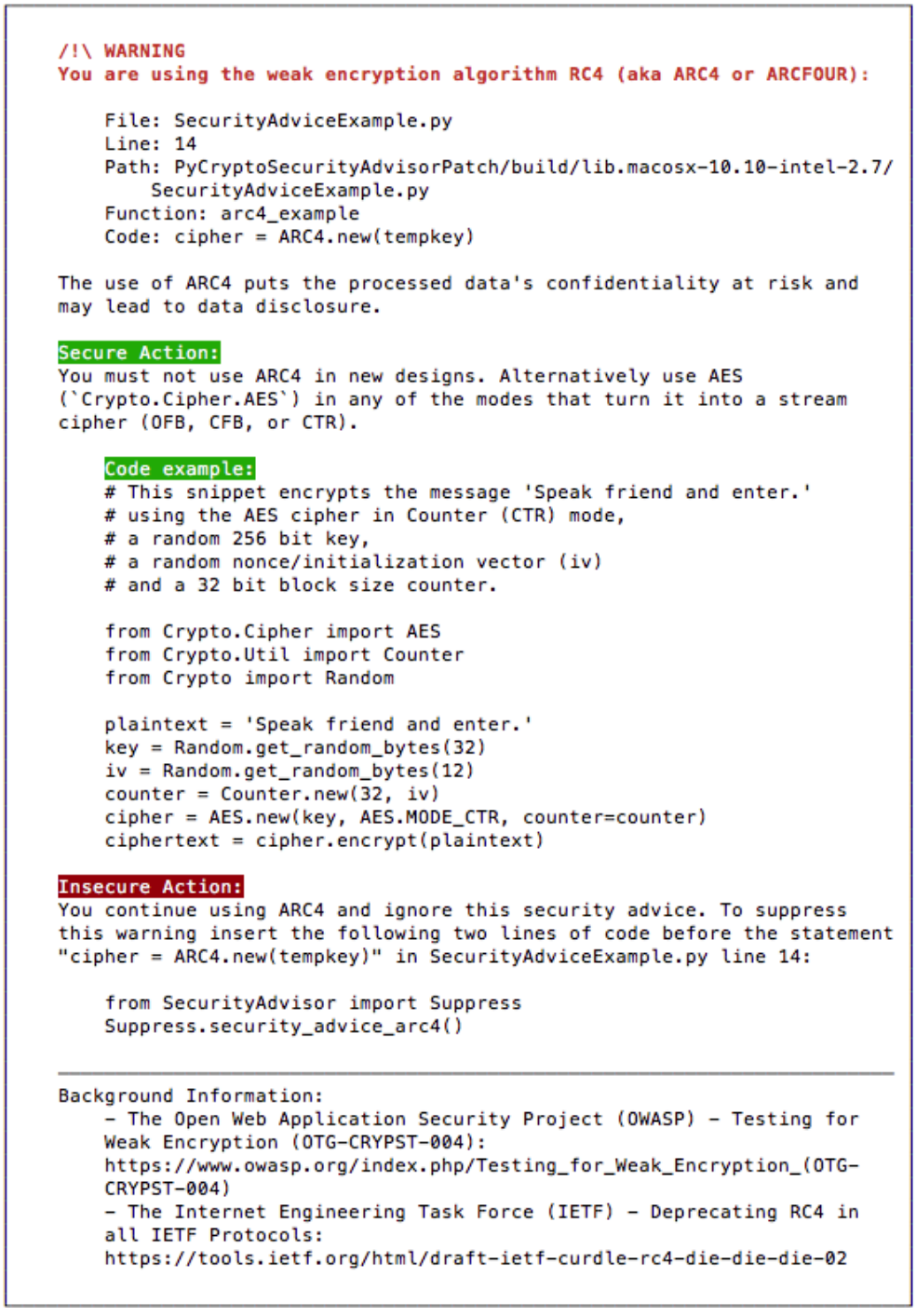


Figure 5.4.: Security advice design of the patched version of PyCrypto triggered by an RC4 usage and displayed in a terminal running Python code. © The USENIX Association

The PyCrypto patch reacts to selected API calls that create instances of weak cryptographic objects such as `Crypto.Cipher.ARC4.new()`, which creates a new cipher object that uses the insecure RC4 [163] algorithm. Whenever an API user creates an insecure cryptographic object, the patch calls an advice method retrieving contextual information for integration into a warning. In more detail, the patch method uses Python's `inspect` module, to obtain detailed information from the cryptographic object's stack frame. The stack frame provides the name of the API call using insecure cryptography, its line number, and the corresponding file name. From this contextual information, the PyCrypto patch compiles the context-specific security warning (cf. Section 5.1.2, p. 67).

Table 5.1 shows for which PyCrypto API calls the patch implements security warnings. It focuses on functions that participants could use to solve the programming tasks. Thus, the patch recommends the use of the Advanced Encryption Standard (AES) to API users as a secure alternative to weak symmetric encryption algorithms. This advice is consistent with the official PyCrypto documentation. The security warning also recommends an upgrade for the insecure Electronic Code Book (ECB) mode. As the official PyCrypto documentation recommends the counter mode (CTR) streaming cipher as a secure mode of operation, the patch recommends it in all security warnings concerning symmetric encryption. In addition to warnings for insecure symmetric encryption algorithms, the patch also triggers warnings for weak hash algorithms (cf. Table 5.1) and recommends the use of the SHA-512 hash function as a secure alternative. To avoid confusing participants if they would use the documentation for solving the study tasks, all provided security warnings adhere to the documentation. However, while the patch implements all features related to the programming tasks in the developer study, it does not cover any of the public key and digital signature schemes provided by PyCrypto. Though extending the patch to cover a more comprehensive list of features is possible.

Developer Study

In an online experiment, Acar et al. [2] compared the usability of five cryptographic APIs for Python (PyCrypto, cryptography.io, M2Crypto, Keyczar, and PyNaCl). They used a between-subjects study design, including symmetric and asymmetric

PyCrypto modules triggering a security warning	Security Advice
Crypto.Cipher	
AES.new(k, AES.MODE_ECB, iv)	AES.new(k, AES.MODE_CTR, iv)
CAST.new(k, CAST.MODE_ECB, iv)	CAST.new(k, CAST.MODE_CTR, iv)
ARC2.new(k, mode, iv)	AES.new(k, AES.MODE_CTR, iv)
ARC4.new(k, mode, iv)	
Blowfish.new(k, mode, iv)	
CAST.new(length(k) < 128 bit, mode, iv)	
DES.new(k, mode, iv)	
DES3.new(k, mode, iv)	
XOR.new(k, mode, iv)	
Crypto.Hash	
MD2.new()	SHA512.new()
MD4.new()	
MD5.new()	
RIPEMD.new()	
SHA.new()	

Table 5.1.: Patched PyCrypto API calls triggering security warnings (`k`: key parameter, `mode`: mode of operation, `iv`: initialization vector).

encryption tasks. They found significant usability differences between libraries and tasks, with poor usability causing insecure code. As the study design of Acar et al. proofed to be successful, the author of this thesis decided to use it as a model for the presented study. Thus, the study consisted of an online, between-subjects experiment to compare how effectively developers could write correct and secure code using either the PyCrypto API as a control condition or the patched version of PyCrypto with the integrated security advice.

Recruitment and Framing

Developers who were familiar with the Python programming language were recruited to gain meaningful and ecologically valid results. 38,533 randomly sampled contributors from 100,000 publicly available Python repositories on the code collaboration platform GitHub were invited to participate via email. Invitations were additionally posted in Python forums, sent to a Python mailing list and the personal network of the author and his peers. The invitation asked Python developers to participate in a Python study using an online code editor. A security or cryptography context was not mentioned to avoid biasing potential participants. Links in the email offered the opportunity to learn more about the study and to blacklist the recipient's email from any further communication. The participation link contained a unique

pseudonymous identifier (ID), which allowed an assignment of study results and GitHub statistics to the invited email addresses. Recipients who clicked the link to participate in the study were sent to a landing page containing a consent form. Once they confirmed their legal age, consented to the study, and were comfortable with participating in the study in English, they were introduced to the study framing also previously used by Acar et al. [2]. Participants were asked to imagine they were developing code for an app called "CitizenMeasure", which was introduced as "a new global monitoring system that will allow citizen-scientists to travel to remote locations and make measurements about such issues as water pollution, deforestation, child labor, and human trafficking. Please keep in mind that our citizen-scientists may be operating in locations that are potentially dangerous, collecting information that powerful interests want kept secret. Our citizen scientists may have their devices confiscated and hacked" [2]. This framing was meant to raise the interest of participants and encourage them to implement a secure task solution. Finally, before the experiment began, the online development environment was explained.

Task Design

These tasks had been chosen by Acar et al. [2] to be "short enough so that the uncompensated participants would be likely to complete them before losing interest, but still, complex enough to be interesting and allow for some mistakes" and designed to "model real-world problems that Python developers could reasonably be expected to encounter in their professional career." Two symmetric encryption tasks were selected: (1) generating an encryption key and (2) storing it securely in a password-protected file and using the key to encrypt some plain text. In both tasks, participants found stub code and some commented instructions. These stubs were designed to make the task clear and ensure the participants could quickly test their results. Also, the main method was pre-filled with code to test the provided stubs. These predefined code structures served participants for orientation and to save time. They were asked only to use the PyCrypto documentation, if possible, and to report any additional documentation resources they consulted by writing inline code comments.

Participants were asked to solve the two programming tasks in an online Python coding environment [182]. They were randomly assigned to either the PyCrypto control condition or the PyCrypto patch condition. In the PyCrypto patch condition, participants solved both programming tasks with the pre-installed patched PyCrypto API, containing the security warnings in question. If they successfully executed functional but insecure code and the API call was patched (cf. Table 5.1) the respective warning was shown. Participants could then consider the advice for their task solution. However, ignoring or bypassing the security advice was also possible. They were not informed that they were using a patched version of PyCrypto. In the PyCrypto condition, participants solved the same two tasks with the original PyCrypto API and, in consequence, without the support of the security advice. The task order was randomized in both conditions. The participants were asked to solve one task with symmetric encryption and one with key generation and storage.

Experiment Infrastructure

A customized version of the open-source framework "Developer Observatory" [182, 183] was applied as infrastructure for the experiment. It allows participants to write and execute code in their browser. The framework is based on Jupyter Notebook [108] and was self-hosted on multiple Amazon Web Service (AWS) servers. Thus, the environment allowed the author to customize and control the study-critical components of the participants' development environment. Foremost, determining available libraries like PyCrypto or the PyCrypto patch, and retrieving code solutions and corresponding metadata like copy and paste events.

As the security advice implementation uses ANSI escape sequences [99] to colorize text in various terminals on various platforms, the Jupyter Notebook version used in the "Developer Observatory" framework needed an update to version 4.4.0 in order to be able to display the warning appropriately. ANSI colors were not processed correctly by Jupyter until version 4.1.0. Due to API changes, further adjustments to the Developer Observatory implementation were necessary. Depending on conditions, participants used the original PyCrypto or the patch. Because both versions share the same namespace and have an identical API, each library had a separate

virtual Python 2.7.12 environment. During an experiment, the Jupyter kernel used only one of these Python environments. Figure 5.5 illustrates how the information of the security advice technically flows from the patched PyCrypto API via the Python interpreter and Python logging facility to a participant's homogeneous test environment of Developer Observatory.

Each participant worked on a separate server to prevent interference between participants. A pool of ready-to-go instances was prepared to avoid any waiting time for participants. When a participant finished the experiment, the instance was automatically terminated to avoid between-subjects contamination.

The two tasks were displayed individually, and an indicator was showing the state, e.g., 1 of 2 tasks. Participants controlled the experiment's progress with buttons. They could "Run and test" their code, and move on when a task was solved ("Solved, next task") or not solved ("Not solved, but next task"). After each button press, the participant's current code, along with metadata like timing, was stored by a remote database.

Allowing participants to write and execute Python code presents serious security concerns. Therefore, the AWS images used only the necessary software packages. The AWS firewall restricted incoming traffic to port 80 and limited outgoing traffic only to the study database, which was password protected and restricted to sanitized insert commands. All instances terminated after a participant had not been active for four hours.

Exit Survey

At the end of the online experiment, participants responded to a brief exit survey. It collected their opinions about the tasks they had completed and the PyCrypto API, including a usability questionnaire for security APIs [2]. The participants evaluated their submitted code for functional correctness and security. As a reference for each task-related question, the participant's code was imported from the database and displayed. Furthermore, participants were asked for their programming experience in general and Python in particular, and demographic data. The survey also contained questions to find out whether participants perceived a security warning at all, if

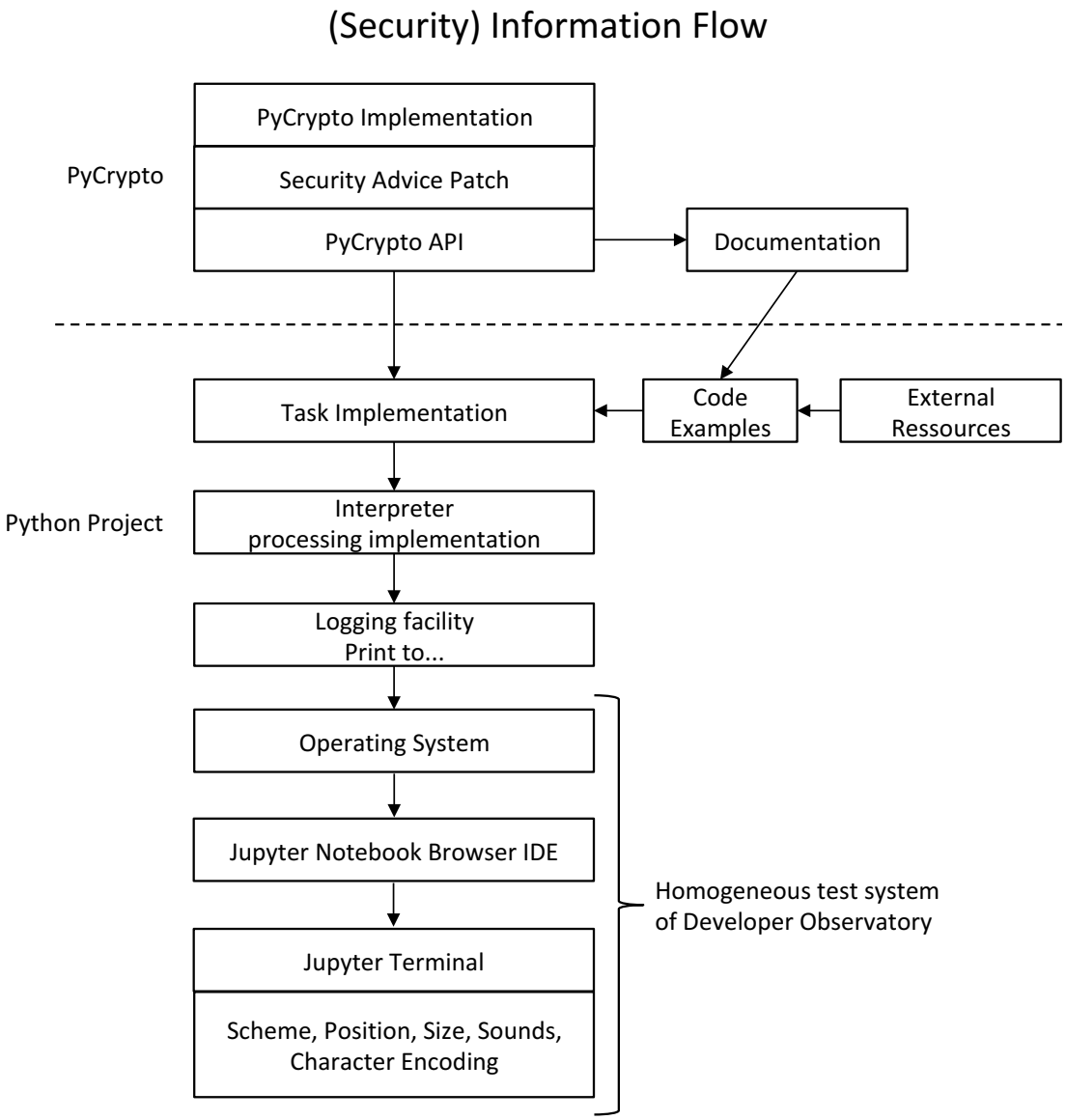


Figure 5.5.: Security information flow in the development environment of the online developer study with the cryptographic Python API PyCrypto and the Developer Observatory [182]. © The USENIX Association

it was helpful, and if participants could recall the security warning's content. The complete exit survey can be found in Appendix C.

Ethics and Pre-testing

Due to the location of the author's university, there was no formal IRB process. However, the study material and procedures were modeled after an IRB-approved study and adhered to the strict German data and privacy protection laws. The design and the implementation of the security advice had been evaluated through expert reviews. Experienced human-computer interaction researchers walked through the warnings and provided feedback. Additionally, the functionality of the PyCrypto patch was pre-tested with participants who were excluded from the later study.

Evaluating Solutions

The analysis is based on the participants' code submissions for each task. Submitted solutions were evaluated both for functional correctness and security. Each task was evaluated independently by two coders based on a subset of the codebook provided by [2]. A third coder adjudicated disagreements between the two coders what allowed to solve all conflicts.

A functionality score of 1 was assigned to a participant's code solution if it was executable without errors, passed the tests, and completed the assigned task. If not, a score of 0 was assigned. Security scores were only assigned to those solutions which were graded as functional. Several different security parameters were considered to determine a security score. The scoring followed the relevant parts of the security scoring in [2]. For key generation, key size and randomness were checked. For key storage, it was checked if encryption keys were encrypted and, if a proper encryption key was derived from the provided password. For key derivation, the use of a static or empty salt, HMAC-SHA1, or below as the pseudorandom function, and less than 10,000 iterations were scored as insecure. For the symmetric encryption task, participants had to select encryption parameters. Therefore, the security of the chosen encryption algorithm, mode of operation, and initialization vector were scored. ARC2, ARC4, Blowfish, (3)DES, and XOR were scored as insecure, and AES as secure. The ECB was scored as an insecure mode of operation and CBC,

CTR, and CFB as secure. Static, zero, or empty initialization vectors were scored insecure. The Krippendorff's alpha [113] was calculated for the initial coding of the two coders across all security codes and was with $\alpha = 0.764$ within reasonable bounds for agreement [67]. Conflicts were resolved afterward.

Participant Stories In addition to the assessment of code functionality and security, participants' code in detail was qualitatively analyzed, based on the recorded code and console output that was automatically stored for each test run of code. The sequence of task solutions that each participant executed was recreated. From these "participant stories," it gets visible whether they were shown a security warning, which version was shown, whether they subsequently adapted their code to incorporate the suggestions, and whether this reaction led to a secure version of their solution. The analysis of participant stories additionally compares the responses to the exit survey, whether they have seen a warning in the exit survey, and whether they perceived it as useful. These participant stories give insights into four questions: (1) did the developers see the warning?, (2) did they react by modifying their code?, (3) did they use the code examples in their code?, and (4) did this consideration lead to improved code security?

Limitations

Four main limitations are addressed below:

Security Advice Design The security advice design applied heuristics defined by previous research on end-user warning message design. Additionally, the design considered lessons learned from previous work on secure programming studies. After manual pre-testing and expert reviews, it was decided to use the approach shown in Figure 5.3 (p. 69). However, there might be more effective designs that were not considered (e.g., following an opinionated design approach might provide better results). Although this is a limitation of the current approach, results for the presented approach show a significant positive impact on code security. Hence, future work should examine changes to the design and compare different versions.

Security Advice Implementation The security advice implementation does not cover all possible insecure choices PyCrypto users could make, e.g., the patch did not implement security warnings for PyCrypto's asymmetric API, however, these were not included in the study design. The data analysis (cf. Section 5.1.4) addressed participants who used APIs not covered by the security advice, as well as cases where the patch failed to show security advice (e.g., non-random IVs for symmetric tasks).

User Study The author decided to conduct an online study over a laboratory study because it is challenging to recruit software developers (rather than students) at a reasonable cost. This design decision allows for less control over the study environment. On the other side, it is possible to recruit a diverse geographic set of participants. It was not possible to recruit participants from an online service like Amazon Mechanical Turk for end-user focused studies. Since it is challenging to manage participant compensations outside such infrastructures, no compensation was offered. Due to the combination of unsolicited email invites and no compensation, a strong self-selection bias was expected. Thus, the results might not necessarily be representative for all developers but, in particular, for those who are interested and motivated enough to participate. In any online study, some participants may not provide full effort or may answer haphazardly. Low-quality data was removed before the analysis, but it is hardly possible to discriminate perfectly. Additionally, a limited and straightforward scenario was tested, which may have limited applicability to complex real-world code.

Real-world applicability

Critically, a real-world application of the advice depends on the commitment of API producers. While this requirement severely limits employment across all libraries, the increasing research on developer-centered security attracts attention, and demands for more usable security APIs can be perceived more often. The author, therefore, hopes that this study is not only of academic relevance but can and will be applied to libraries with a large userbase.

Data Analysis

The data analysis, uses the non-parametric Mann-Whitney-U test (MWU) to compare two groups with continuous responses, compares categorical responses with Person's chi-squared test (χ^2) or instead with Fisher's exact test where applicable.

5.1.4. Results

The results refer to 53 valid participants. Participants were generally successful in functionally solving the tasks, while security results varied across conditions. In the experiment, the patched API was an improvement for the original PyCrypto API. Participants who wrote code that triggered a warning message changed it to a secure solution 15× as likely as participants who wrote similar insecure code in the PyCrypto condition. However, the limited applicability of the warnings negatively impacted the effectiveness of the PyCrypto patch.

Participants

Participants were mainly recruited by sending email invitations to GitHub developers. Of 38,533 sent invitation emails, 3,422 (8.9%) were not deliverable, and 65 (0.2%) recipients requested to be removed from the mailing list.

The author did not receive reports of technical problems with the experiment infrastructure. One participant refused to participate in the study because the Amazon AWS instances were not accessible via HTTPS. Another person refused to participate because he perceived the invitation email as dubious.

272 people agreed to the consent form, and 177 started working on the tasks. Of those, 70 finished the tasks, and 68 completed the exit survey.

15 participants were excluded since their results indicated a lack of serious participation or were the result of curious clicking-through. Unless stated otherwise, the results refer to the remaining 53 valid participants who finished the tasks and completed the exit survey.

49 participants reported being male (92.5%) while the remaining participants reported being female (1), other (1), or preferred not to answer (2) (cf. Table 5.2). The reported age was between 20 and 60 years (mean: 34.9, SD: 8.1). 44 participants

Age	Youngest, Oldest	20, 60
	Prefer not to answer	3
	Mean years (SD)	34.9 (8.1)
Sex	Male	49
	Female	1
	Other	1
	Prefer not to answer	2
Recruitment	GitHub	44
	Other	9
Experience	Mean development years (SD)	15.8 (8.2)
	Mean Python years (SD)	8.44 (4.7)
	Prefer not to answer	3
Occupation	Professional	48
	Student	3
	Both	1
	Prefer not to answer	1

Table 5.2.: Demographics of 53 participants that completed the exit survey.

received invitation emails as GitHub developers. At the same time, the remaining 9 were recruited on developer forums, a developer mailing list, or the personal network of the author and his peers. The participants had a mean developer experience of 15.8 years (SD: 8.2, prefer not to answer: 3) and a mean Python experience of 8.44 years (SD: 4.7, prefer not to answer: 3). 48 reported being a professional software developer, and 3 reported being students (both: 1, prefer not to answer: 1).

Dropouts

95 did not continue to the study while 177 started the first programming tasks by clicking the begin button. 57 participants stopped in the key-storage task, further 44 in the content-encryption task, and 4 in the final test routine before finishing the online programming part of the study. 29 had written code to solve a task in contrast to 76, who did not modify any text in the Jupyter notebook. 5 dropped out of the PyCrypto patch condition after having triggered security advice. 70 proceeded to the exit survey. 68 participants finished the exit survey, of which 15

persons had to be excluded due to non-serious participation and technical issues in the infrastructure.

Out of 115 participants in the PyCrypto patch condition, 90 participants dropped out of whom 5 were shown a warning. However, the 25 who finished the study were shown 11 warnings, so it is assumed that seeing a warning was not a strong reason to drop out of the study. 34 dropouts out of 62 were counted for developers who started the PyCrypto condition. The increased count of starting participants was due to an effort to counterbalance the limited applicability of the warnings.

Results for Functionality

Generally, participants were well able to solve the study tasks. 87.8% of attempted tasks were functional (89.7% functional in the PyCrypto condition, 85.9% in the PyCrypto patch condition). The warning messages did not significantly impact the results, positive or negative. However, this was expected because only functionally correct PyCrypto API calls triggered the warning. Nevertheless, interruptions caused by API warnings did not lead participants to break their code.

Results for Security

The analysis showed 26.9% secure solutions in the PyCrypto condition compared with 50.7% in the PyCrypto patch condition. A small number of 11 tasks triggered the warning, and three ended up with insecure code in the PyCrypto patch condition, as well as only one of 22 tasks that would have triggered a warning but was modified to be secure in the PyCrypto condition. Fisher's exact test (cf. Table 5.3) was significant ($p<0.01$), with an odds ratio of 56.

		Secure	
		F	T
Warning	F	21	1
	T	3	8

Table 5.3.: Contingency table for secure task solutions and triggered warnings used in the Fisher's exact test.

The warning messages were noticed by participants, which was clear both from self-reported memories and their code changes. In 8 out of 11 cases, a warning message leads to a change from initial insecure code to a secure solution. Generally, the applicability of the warning message was limited. It applied to 24 of 44 insecure solutions across conditions and was shown in 11 of 22 insecure cases in the PyCrypto patch condition.

Impact of Intervention on Perceived Usability API usability as interpreted by the questionnaire proposed by Acar et al. [2], which bases on the Cognitive Dimensions framework [49], did not change for better or worse with the warning. This was expected, as only one of 10 questions that are calculated into the usability score focus on meaningful warning/error messages.

For further analysis of responses, agreement answers were transformed to a numerical 5-point likert-scale (strong disagreement:−2, disagreement:−1, neutral:0, agreement:+1, strong agreement:+2). The participants in the PyCrypto condition agreed (mean:1, median:1) that security warnings displayed in the console helped to solve this task, compared with an agreement of 0.76 (median = 1) in the PyCrypto patch condition (MWU-test; U=32.5; p=0.4205). Participants in the PyCrypto condition agreed (mean:0.593, median:1) to the statement "When I made a mistake, I got a meaningful error message/exception", compared with an agreement of 0.833 (median = 1) in the PyCrypto patch condition (MWU-test; U=384; p=0.2167). Participants in both conditions also accepted the statement "Using the information from the error message/exception, it was easy to fix my mistake.", with a mean agreement of 0.846 (median = 1) in the PyCrypto condition, and 0.917 (median = 1) in the PyCrypto patch condition (MWU-test; U=296; p=0.7484). These results can be interpreted as a generally positive perception of the warning, despite the preliminary concerns of annoying or overwhelming developers. However, even in these specific cases, perceptions were not significantly more positive or negative than in the control condition.

Detailed Task Analysis

Participants were asked to rate their functional and security success after completing the tasks (cf. Figure 5.6).

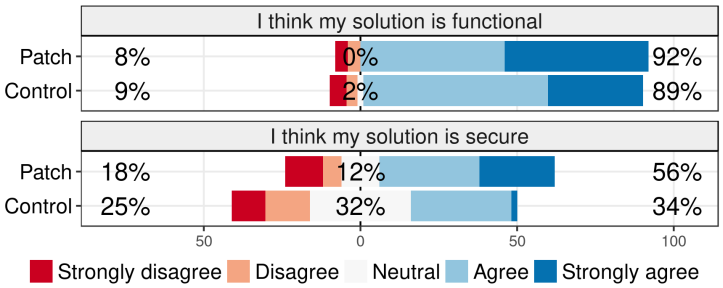


Figure 5.6.: Likert-plot showing the participants' perceptions regarding functionality and security of their solutions. "I don't know" answers were omitted. © The USENIX Association

For the encryption tasks, all participants who saw the warning message were correct in assessing their solution's security. In the control condition only 66% rated the security of their encryption task solution correctly, in cases where the patch would have applied. Vice versa, 73% of the developers in the patched condition assessed the security of the key storage task correctly, while all assessments were correct in the control group.

Participant Stories Further qualitative results were collected from the participants' stories. When focusing on the content-encryption task, 7 participants were shown a security warning. All of them saw and remembered it, as they reported in the exit survey. 2 of the 7 participants did not choose to use the guidance to improve their code. One tried to suppress the advice, and another one ignored it. The remaining 5 participants accepted the advice and modified their code. 2 of them adopted the example code provided by the advice; they later expressed their satisfaction: *"The warning helpfully directed me towards an improved solution, and provided example code"* and *"The warning explained clearly that DES was considered as insecure, and provided an example to use AES instead. This helped me solving this task in a more secure manner."* The remaining 3 participants partially followed the advice: they did adapt their code in response to the warning but chose a different mode of

operation than was suggested in the warning. The proposed solution recommended the use of standard encryption algorithm AES in counter (CTR) mode. The 3 participants instantiated AES in cipher block chaining (CBC) mode instead. A closer look at their code revealed that 2 of them appeared to have problems in transferring the suggested code snippet into running code. While this indicates a usability problem with the advice, the analysis showed that 4 of 5 participants who modified their code in reaction to the warning at least attempted to adhere to the suggestion. Altogether, 5 out of 7 participants who saw the warning for the encryption task modified their code into a secure solution.

Similar behavior was observed for the key generation and storage task. Here, 4 participants were shown security advice; all of them noticed the warning. One ignored the warning; the remaining 3 modified their code. One adopted the suggested code snippet as-is; the other 2 chose CBC mode instead of CTR mode.

Limited applicability of warning message The programming tasks in this study had been designed in a way participants had only to use symmetric cryptography. Thus the prototype did not cover insecure asymmetric API features (cf. Table 5.1, p. 74). However, 4 participants in the patch condition solved tasks by using asymmetric methods. They implemented key derivation for asymmetric RSA keys or applied RSA to encrypt keys and messages. The security advice implementation was not able to help these participants using symmetric cryptography since it is not possible to give task sensitive advice at this position. For this reason, these task solutions were excluded from the analysis. Furthermore, the patch was not able to support all appeared insecure API practices (cf. Table 5.4), like static initialization vectors (IV) or custom key derivation functions (KDF).

5.1.5. Discussion

In this study, the author examined the first API-integrated security advice for cryptographic APIs. An initial design propopsal followed end-user guidelines by Bauer et al. [28] and considerd lessons learned from previous work on developer-centered security. The author developed and implemented a prototype for Python's

Insecure API use	Count	%
Key In Plain	1	3.57%
Weak Cipher	9	32.14%
Weak Mode	7	25.00%
Static IV	9	32.14%
No KDF	16	57.14%
Custom KDF	16	57.14%
KDF Salt	1	3.57%
KDF Algorithm	3	10.71%
KDF Iterations	1	3.57%
Participants	28	100%

(a) Key file task.

Insecure API use	Count	%
No Encryption	0	0.00%
Weak Algorithm	10	35.71%
Weak Mode	9	32.14%
Static IV	11	39.29%
Participants	28	100%

(b) Encryption task.

Table 5.4.: General reasons for insecure API use in the PyCrypto condition

PyCrypto API and customized the Developer Observatory framework [182] to conduct a between-subjects online controlled experiment.

The results of this study answer **RQ3** of this thesis. The API-integrated security advice had a significantly positive effect on code security. The majority of the participants who received security advice turned insecure code into secure code. Also, the adherence rate to the security advice (73%) was promising. Figure 5.7 places the achieved improvement in the context of the study results by Acar et al. [2], who compared the usability of several cryptographic APIs.

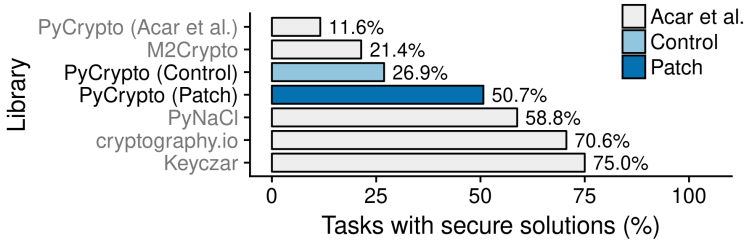


Figure 5.7.: Comparison between secure solution percentages of libraries from Acar et al. [2] and the PyCrypto Control/Patch results. To match the tasks, only symmetric encryption tasks are considered for libraries from Acar et al. © The USENIX Association

However, the perceived usability of PyCrypto as a cryptographic API was not affected by the security warning. Only one participant who received the advice, suppressed security warnings for future runs and two participants copied secure code snippets from a warning into their code. Thus, the presented approach allows API providers to improve code security for existing cryptographic APIs on the implementation level, without having to change existing API interfaces, or the official API documentation, or relying on resources outside their sphere of influence, such as third party information resources, IDE plugins or static code analysis tools.

However, the study results emphasize that designing and implementing effective security advice is challenging and has its limitations. Thus, the prototype was not able to support insecure API practices like static initialization vectors, custom key derivation functions, or an unsuitable application of asymmetric cryptography to a given task. This demonstrates the importance of informing developers about security prerequisites. Here warning messages show further potential to support users of security APIs. However, providing context sensitive information and secure and ready-to-use code snippets is complex and requires further research.

The presented security advice prototype design followed security warning design guidelines by Bauer et al. [28] and considered lessons learned from related work on developer-centered usable security research [2, 3, 137]. However, this was a proof of concept for one initial implementation of the approach. Changing parameters such as text or advice design might result in even more secure code. Furthermore, despite the positive results, it is unclear whether this approach is acceptable for software developers and their development environments, or to which extent the concept can be applied to other security APIs. For this reason, the following study conducts a focus group study with software developers to develop a participatory design for security API warnings.

5.2. Study 4: Participatory Design

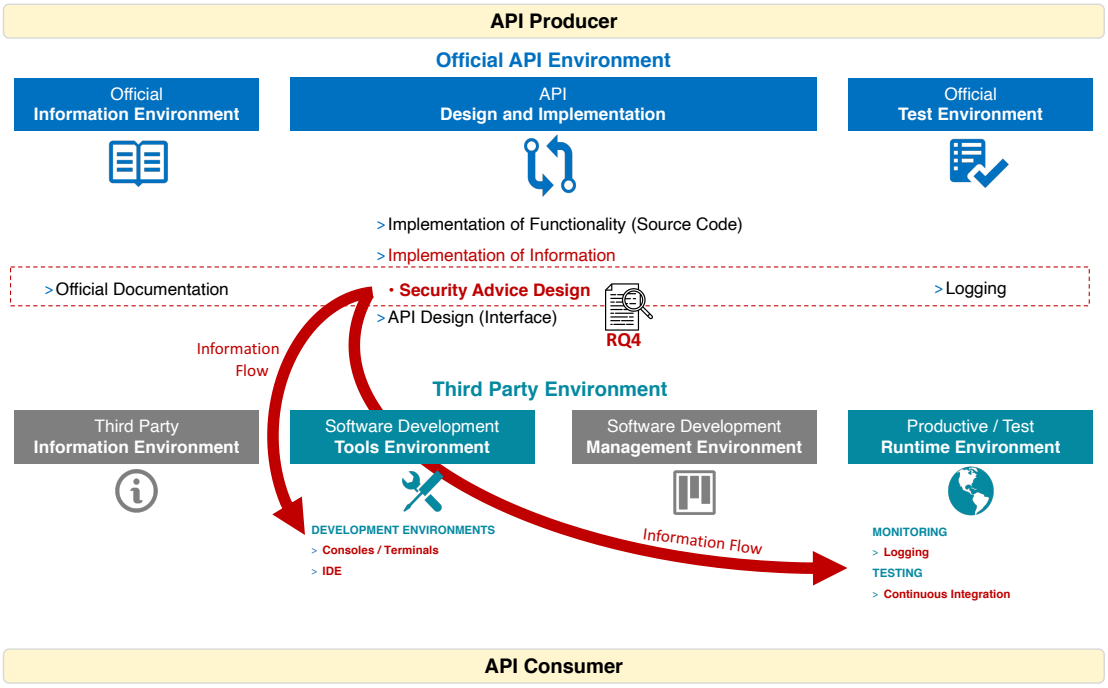


Figure 5.8.: IFSIF model classifying contributions of Study 4.

5.2.1. Motivation

Study 3 of this thesis has shown that security API warnings displayed in the environment of a terminal or console can support developers in using an API securely (cf. Section 5.1, p. 65). While the experiment demonstrated improved code security, the design approach applied heuristics from end-user research [28] that are not yet proven to be suitable in a real-world context of software development and console environments.[2]

[2]The contents of the present chapter were previously published in the paper **"Listen to Developers! A Participatory Design Study on Security Warnings for Cryptographic APIs."** by Peter Leo Gorski, Yasemin Acar, Luigi Lo Iacono, and Sascha Fahl which appeared in the Proceedings of the *2020 CHI Conference on Human Factors in Computing Systems (CHI)*. Honolulu, US-HI, USA: Association for Computing Machinery (ACM), 2020, pp. 1 – 13 [83].

The author is not aware of research on the usability of other types of API warnings like an update, deprecation (API feature removal or end of support), or linting (code analysis, e.g., for programming errors or inconsistent style) warning. Unlike other API warnings, cryptographic API warnings indicate that cryptography is not being handled securely to protect data. A warning could indicate that, e.g., the confidentiality or integrity of data is not present, even though API calls are present that are intended to integrate such functionality. This situation is difficult to detect because the program superficially seems to work as intended and behaves just as it would if the data were handled securely. The consequence is an intransparent lack of data protection at the code level, which affects the data of all users of the software. This research, therefore, aims to contribute to the avoidance of unintentional or accidental insecure security API use by examining how to give appropriately designed API integrated security advice to developers (cf. Figure 5.8). Results emphasize that terminals, IDEs, logging files, and testing tools for Continuous Integration processes are essential environments for security-relevant information, but they all require specifically designed information flows.

Because the developer console is a central point of information in software development (cf. Section "RQ4a - Warning Context", p. 106), the purpose of this study is to revise and improve the previous security API warning design for cryptographic APIs (cf. Section 5.1.2, p. 67). It aims to gather more knowledge about the dimensions, aspects, advantages, and disadvantages of the API warning approach from a software developer's perspective. The research question is (**RQ4**): *What kind of design do software developers find helpful for cryptography warning messages in the console?* The study examined designs for cryptography warnings in the console based on the experiences and preferences of the target group. Therefore, the author followed a participatory design approach and conducted four focus groups with a total of 25 developers. A three-step process was applied. First, focus groups compiled information they perceived as helpful when being warned of insecure cryptographic API use. Second, participants visualized and drew warnings they perceived as helpful. In a third step, the group members compared their approach with the design approach of the previous Study 3 (cf. Section 5.1.2, p. 67).

Focus group results were also analyzed concerning three expanding questions (RQ4a - RQ4c) to answer the main question adequately: (**RQ4a**) Do software devel-

opers find console security warnings helpful in development processes and software environments? The design approach was put into the real-world context to learn from the developers how they work with the developer console and how they assess the relevance of security warnings in this particular environment. (**RQ4b**) From the developer's point of view, is there a difference between security warnings and other types of console warnings? The author wondered how security warnings differ from other warnings in the console to gain further insight into design requirements. (**RQ4c**) Would developers implement their warning design into a cryptographic API? Common hurdles were identified for implementing cryptography warnings in an API.

The participants were asked to design and comment on existing "security warnings" for cryptographic APIs displayed in developer consoles. However, at the end of the focus groups, the views were spread across a wider development environment context. In a broader tool-wide sense, the general term "security feedback" is used.

This study makes the following contributions: (1) The author conducted four focus groups to improve cryptographic API feedback for developers. (2) It was found that developers generally find security feedback important and useful. For console messages, the participants suggested five core elements. These are message classification, title message, code location, link to detailed external resources, and coloring. (3) Participants emphasized the importance of tailoring the detail and content of security information to the context. Console warnings call for concise communication; further information needs to be linked externally. Therefore, security feedback should transcend tools (cf. Figure 5.8, p. 90) and should be adjustable by software developers across development tools, considering the application context and developer needs. (4) This work shows for the first time from a developer's point of view that existing end-user centered security warning guidelines cannot be applied directly. Design proposals, coming from the target group, are presented for tailored amounts of information in the development process (cf. Figure 5.14, p. 107). API providers and researchers can use and build upon these results to further develop warning implementations and support API users with security-relevant information.

After presenting the methodology, results are reported by answering each defined research question in a separate section.

5.2.2. Methodology

Four focus groups are conducted with experienced software developers to gather a wide range of opinions, perceptions, and ideas emerging from group discussion [114]. Focus groups can also help to identify and address aspects that may not be mentioned in individual interviews. There was no formal IRB process at the author's university. However, the study design complies with the requirements of the European General Data Protection Regulation [193]; participants agreed to the anonymized use of their study data by reading and signing a consent form before the study. Focus groups were audio-recorded and transcribed. Anonymized transcripts, as well as pictures taken of materials developed in the study, were kept in secure storage; audio files and personally identifiable information were deleted.

Study Protocol

The study protocol was developed according to focus group guidelines [114]. The protocol was reviewed by four subject matter experts (including the author) and revised according to their feedback. The focus group protocol was structured as follows and presented via a slide deck:

Warm-up questions:

1. Why are you enthusiastic about programming?
2. What do you associate with data security in the context of software?
3. What do you associate with working with a console or a terminal?

Console warning experiences:

4. Recall which consoles you have worked with before and which warnings were displayed there? What experiences have you had? (Showed six example screenshots of consoles) Write on cards and discuss them.

Participatory warning message design:

5. What are the reasons why displaying console warnings can be (a) helpful or important (b) unhelpful or unimportant in the development process? Write on cards, then work and discuss on a pin-board.

6. (Showed an RC4 Python code example.) Imagine a cryptographic API issuing a security-related warning message in the console to give the developer a recommendation for action. Please list the information and instructions that you would expect to be helpful in such a case. What aspects or content should a security warning include in the console? Write on cards, then work and discuss on a pin-board.

7. Please rate your agreement to the following statements: I find aspect X (a single card or grouped cards at the pin-board) helpful as a part of a security warning message in the console. *(strongly disagree, disagree, neutral, agree, strongly agree)*

8. What do you think a helpful security warning message looks like in the console? Please sketch an example message on a piece of paper. You are allowed to draw multiple messages. Present and discuss your drawings.

9. Please look at the following security warning proposal (cf. Figure 5.13, p. 105) for a few minutes. What do you (a) like and (b) dislike about it compared to your design? Discuss.

Warning context:

10. What could be reasons that would make you: (a) Implement the recommendation of the security warning? (b) Ignore the recommendation of the security warning? Discuss.

Warning relevance:

11. Imagine you are developing a cryptographic API in the future or you already developed one, would you implement console security warnings to help users of your API to prevent insecure cryptography use? Please elaborate on the reasons that informed your decision and discuss.

The main results were summarized by the moderator:

12. Do you feel my summary includes all the important points?

13. Are there any other aspects we should have talked about?

The study protocol was tested with the first focus group. However, only slight changes were made to the protocol based on the participants' feedback: One slide was revised to make one question more easily accessible. The study was conducted in German; however, participants organically produced their written and drawn materials mostly in English. The author led all focus groups and moderated group discussions.

Recruitment

The recruitment addressed developers with professional programming experience in the Cologne area. Professional programming experience was required but did not restrict the recruitment to a specific developer population, to include a wide variety of experiences and opinions. Potential participants were invited via mailing lists, e.g., of PyCologne, a local Python developer group, and via requests to personal contacts at software development companies. Also, participants were recruited based on recommendations by focus group participants after their focus group. The invitation emails included a link to the research project website and a brief qualification questionnaire. Thereby, information was collected about prospective candidates' work and programming experience. Overall, the recruitment ended up with 25 participants for four focus groups. Four focus groups were conducted since saturation was reached after the third focus group. The fourth group did not develop or discuss new aspects.

Participants

In total, 25 participants were recruited from the Cologne metro area. These were organized into four focus groups that took place in February, May, and July of 2019. Participants were between 20 to 43 years old (mean age: 31 years, SD: 6). All participants were male (cf. Section "Limitations", p. 99). They reported to have been programming for 10 years on average (SD: 6) and have been working professionally as software developers for 6 years on average (SD: 6) (cf. Table 5.5).

	G1P1	G1P2	G1P3	G1P4	G1P5	G1P6	G2P1	G2P2	G2P3	G2P4	G2P5	G2P6	G2P7	G2P8	G2P9	G3P1	G3P2	G3P3	G3P4	G3P5	G3P6	G4P1	G4P2	G4P3	G4P4
Profession																									
softw. dev. in industry	●	●	●	●	●[1]	-	●	●	●	●[3]	●	●	●	●	●	●	-	●	●[5]	-	●	●	●	●	●
system administrator	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●	-	-	●	-	-	-	-
industrial researcher	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●	-	-	-	-	-	-
academic researcher	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●[4]	-	-	●	-	-	-	-	-
bachelor student	-	-	-	-	-	-	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
other	-	-	-	-	-	●[2]	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Software																									
web applications	-	-	●	●	●	●	-	●	●	●	●	●	●	●	●	●	●	-	-	●	●	●	-	-	●
mobile applications	-	●	-	-	●	-	-	-	-	-	-	-	-	-	-	●	-	-	●	-	●	-	-	-	-
desktop applications	-	-	-	-	-	-	-	-	-	-	-	●	-	-	-	-	-	-	-	-	●	●	-	●	-
embedded systems	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●	-	-	-	-	●	-	-
enterprise software	-	-	-	-	●	-	-	-	-	-	-	-	-	-	●	-	-	-	-	-	-	●	-	-	-
front-end	-	-	●	-	●	●	-	●	-	-	-	-	●	-	●	●	●	-	-	-	●	●	-	-	●
back end	-	●	●	●	-	●	●	●	●	●	●	●	●	-	●	●	●	●	●	-	●	●	-	-	●
software for developers	-	●	●	●	-	-	-	●	●	-	●	-	●	-	●	-	●	-	●	●	●	●	-	-	-
other	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●[6]	-	-	●[7]	-	-	-	-	-
Language																									
Python	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●	-	●	●	-	-	-	-	-
C++	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●	-	-	-	●	●	-
Java	-	-	-	-	-	●	-	-	●	-	-	-	●	-	●	-	-	●	●	-	●	●	-	-	-
C	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●	●	-	-	-	●	-
C#	-	-	-	-	-	-	-	-	●	-	-	-	-	-	-	-	-	-	-	-	●	-	-	-	-
PHP	-	-	●	-	-	-	●	●	-	●	●	●	●	●	●	●	-	-	-	-	●	-	-	-	●
JavaScript	-	-	●	●	●	●	-	-	-	●	●	-	●	-	●	●	-	●	-	-	●	●	-	-	●
Go	-	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
Ruby	-	-	-	●	-	-	-	-	-	-	-	-	-	-	●	-	-	-	-	-	●	-	-	-	-
other	-	●[8]	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●[9]	●[10]	-	-	-	-	-
Education/Experience																									
PhD	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●	-	-	-	-	-	-
master degree	-	-	-	●	●	●	-	-	-	-	-	-	●	-	-	-	●	●	-	-	-	●	●	●	-
bachelor degree	●	●	●	-	-	-	●	●	-	-	●	-	●	-	-	●	-	-	-	●	-	-	-	-	●
IT Specialist (Certified)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●	-	-	-	-
years developing	8	9	6	15	10	10	6	6	5	2	11	9	4	4	18	5	20	11	23	19	3	20	1	3	12
working years	3	7	6	6	8	2	6	6	1	2	6	8	3	4	18	3	5	11	20	15	3	11	1	2	5
Demographics																									
gender	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m
age	32	32	30	30	29	32	40	28	22	21	31	26	27	30	43	24	37	37	42	34	20	37	28	28	31

[1]consultant, [2]cross-process coordination with focus on software architecture, [3]apprenticeship as application developer, [4]research software engineer, [5]managing director, [6]workflow systems for scientific operation, [7]high performance computing, [8]Kotlin, [9]Objective-C, [10]Fortran

Table 5.5.: Demographic data of the focus group study participants.

Data Analysis

Focus groups lasted about two and a half hours. After each focus group, photos of pinboards were taken, and all drawings or notes were digitized. Full transcripts of audio recordings, participants' notes, and drawings were grouped by question and task and imported to ATLAS.ti [20].

The codebook was created by four researchers (including the author), reflecting different experiences and backgrounds, which include formal education in media technology, mathematics, and computer science; all have research experience in security, privacy, and human factors research. The coding was applied for grouping the focus groups' contents. Beginning with open coding, the author analyzed and coded each contribution to the discussions. In this inductive process, (1) exhaustiveness was considered to ensure codings will reflect all main topics when assigned to higher-level semantic groups. It was also adhered to (2) mutual exclusiveness of semantic groups to keep codes unambiguous. Because it was paid attention to completeness, a total number of 210 sub-codes of 21 higher-level code groups were operationalized following a group discussion between all four researchers.

Two additional coders, with a computer science background and minor HCI knowledge, were introduced to the codebook in a one-hour briefing. Following, they independently coded focus group number one (about 25% of all transcripts, exceeding recommendations for double-coding in qualitative research [91]) applying the codebook. The intention was to use inter-rater agreement scores to identify ambiguous codes or codings that were forgotten by the primary coder. While the granular codebook was difficult for the additional coders to apply, mainly because they were inexperienced in the study's topic and qualitative coding and due to the high number of codes, the following comparison and discussion resulted in a more precise operationalization of the codebook and confirmed the main coder's coding decisions. After completing the coding process, research questions were matched to relevant code groups (cf. Table 5.6), and the focus groups' results were elaborated.

To be able to give weight to the assessment of helpful content of cryptography warnings, quantitative data was extracted from the focus groups in the following ways: Participants rated their suggestions on a 5-point Likert-scale. The author counted the individual contents of one drawing per participant. The moderator

Code group	RQ	Description
Developer Console Activities	4	Things you do with a console or activities you use the console for.
Developer Console Properties	4	Properties of a development console.
Developer Console Types	4	Different types or consoles, even in terms of differences in operating systems.
Documentation	4	Statements about the documentation of APIs and program code.
Security Feedback	4c	Particularities of warnings in a security context and differences to other warnings.
Software Development Tools	4b	Tools for software development that are addressed.
Software Environment	4b	Statements about an environment in which software is developed or executed.
Warning Content Negative	4	Warning content that is undesired, unimportant or unhelpful - contra arguments.
Warning Content Positive	4	Warning content that is desired, important or helpful - pro arguments.
Warning Design in General	4	General statements about console warning design not relating to content or properties.
Warning Display Location	4a	Where, in addition to the console, a warning is also displayed.
Warning Implementation	4c	Aspects of implementing console warnings in APIs.
Warning People Involved	4a	People involved when a warning is displayed.
Warning Processing Practices	4a	About working with a warning or processing warnings.
Warning Properties Negative	4	Negative properties of warnings in the console that are not represented by warning content.
Warning Properties Positive	4	Positive properties of warnings in the console that are not represented by warning content.
Warning Purpose	4b	The purpose of a warning message. What is to be achieved by the warning?
Warning Reason	4b	The reason of a warning message. Why is a warning given?
Warning Target Group	4	Persons addressed by a warning.
Warnings Reactions	4	Developers' reactions to warnings or how they deal with the situation.
Weak Crypto Use Case	4	Reasons why an insecure API call is used.

Table 5.6.: Code group descriptions.

asked the participants for a definite yes or no statement as to whether they would implement cryptography warnings in their APIs.

Limitations

This sample is not representative for all developers: The participants came from Germany and notably, only men participated in the study (cf. Table 5.5). It was attempted to recruit women by directly emailing female developers, which, with more than 92% developer roles in Germany being filled by men [179], proved to be hard. Three women signed up for the focus groups. However, two did not respond to an appointment request, and one did not fulfill the qualifying requirement of having any development experience. Therefore, this study represents the views of local male developers.

All results are an artifact of the participants' opinions and experiences. The group compositions may also have influenced participants' statements. All participants of the second group work in different areas and project teams in the same software company. This diversity is also reflected in their self reported programming experiences (cf. Table 5.5). The recruitment was able to represent a broad spectrum of experiences with programming languages and software types in the sample, which is a prerequisite for answering the research questions.

Symmetric encryption is one of the most common tasks with a cryptographic API [133]. Therefore the RC4 was used as an example in the focus groups. RC4 is typical for insecure encryption and transferable to cases in which other insecure features are used, like hashing algorithms or API parameters. It is not representative of all types of security issues with cryptographic APIs. Further studies are required to compare and evaluate different designs and identified aspects of this work to develop a mature guideline gradually.

5.2.3. Results

The following section, reports the results of the focus groups. Each subsection refers to one of the four research questions (cf. Section "Motivation", p. 90).

RQ4 - Participatory Warning Message Design

The participatory design approach bases on three different study artifacts. In a first step, participants were asked to collect helpful information on a pinboard (cf. Figure 5.9) and rate each suggestion individually on a five-point Likert scale (cf. Figure 5.10). Then they were asked to outline an ideal message independently (cf. Figure 5.11, p. 103 and Figure 5.12, p. 104) and to evaluate the proposed warning design of Study 3 (cf. Figure 5.13, p. 105).

Rating of Helpful Information All participants had concrete ideas about helpful cryptography warning message content. However, most of the proposals were discussed controversially mainly because developer consoles are used for various use cases (cf. Section "RQ4a - Warning Context", p. 106) and should, therefore, not be overloaded with unimportant or unnecessary information.

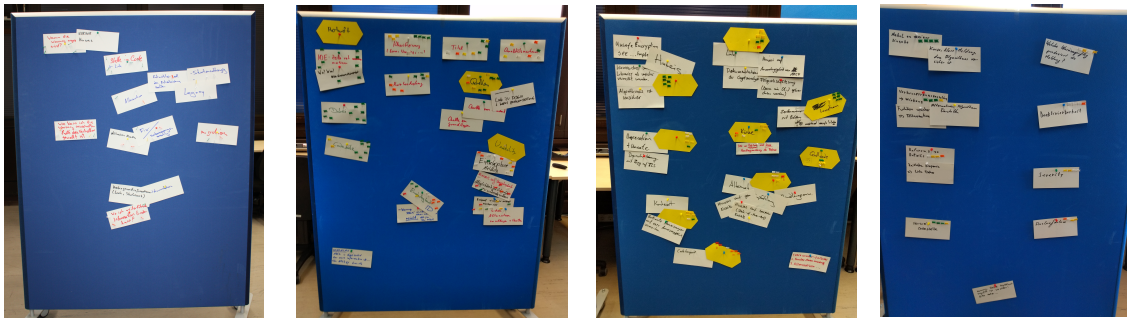


Figure 5.9.: Pinboard photos of all four focus groups' collection of helpful information.

In all groups, at least one participant mentioned the aspects of source code location, title message, link to external resources, and alternatives for discussion (cf. Figure 5.10). Three groups discussed an option to deactivate the warning. Since these aspects were mentioned in several groups, they were considered being particularly relevant. 12 additional aspects were named in single groups. The rating also shows content that participants consider helpful but exclude as part of a console warning.

A unanimous agreement can only be found for two aspects. First, the exact **source code location** to which the warning refers: *"With file, line, and code*

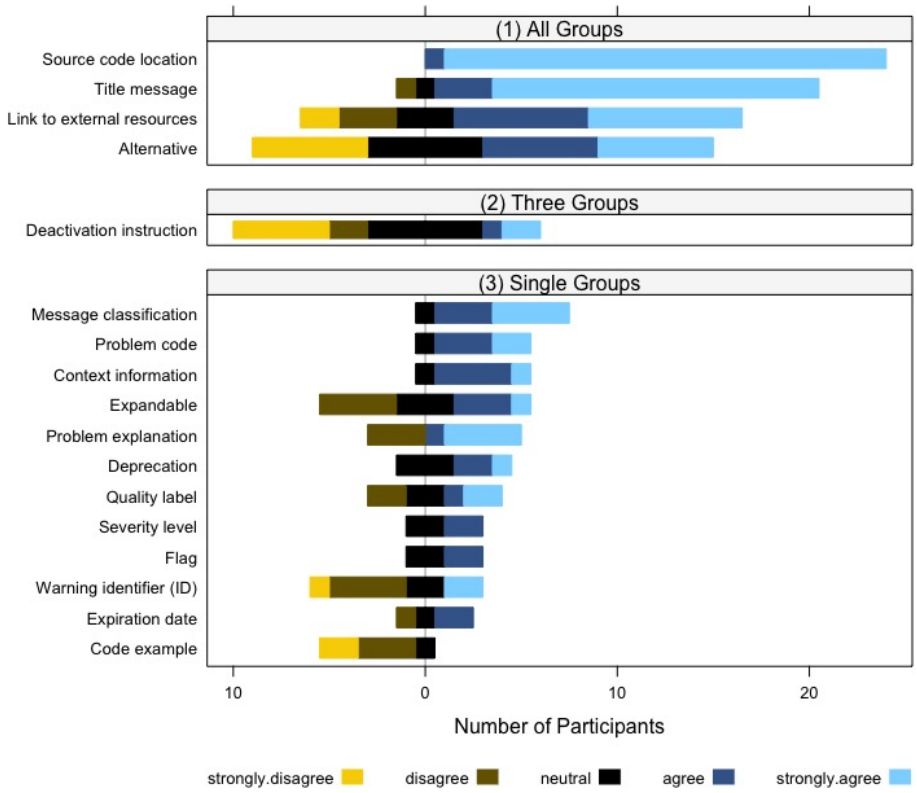


Figure 5.10.: Participants' rating results of helpful aspects as part of a security warning for a cryptographic API (step six and seven of the study protocol). Grey labels indicate how many groups have listed the items. © ACM

you find the place." (G4P1)[3], which is preferably clickable to quickly reach the problem code: *"I would like [...] to have a direct link to it"* (G1P2). Second, a clear **title message**: *"A short title that precisely describes the actual problem, unsafe cryptographic algorithm RC4"* (G2P7). In the following, reasons are reported why participants strongly disagreed with some content.

A direct **link to external resources** e.g., documentation that belongs to the API or the problem, is generally desirable to get information fast: *"Somehow you have to be able to find something on the internet in case of doubt. The best is if the link is directly included"* (G3P1). However, some developers find a link unnecessary because *"I can copy error messages and throw them into Google myself."* (G2P1)

[3]All quotes have been translated from German into English.

or *"[...] you take the name and find out what this algorithm actually does because the algorithm is described"* (G1P1). Another prerequisite is the availability of online resources: *"So a link, yes, but please not only because websites disappear. [...] [T]here should please be more [information] instead of just a link"* (G3P2). For similar reasons, the programmers disagreed with **warning identifiers** like numerical IDs. *"Without documentation, the ID is good for nothing"* (G2P8).

However, the rating shows that 12 developers take a critical view (at least neutral) of suggesting an **alternative**. The context of cryptographic APIs is seen as complex. They doubt that a warning can make a suitable suggestion: *"I would be at least skeptical with encryption whether one can make a generally valid statement"* (G3P3). *"Because no one will be able to offer you a fix for a broken cipher, because no one actually knows [...] what the background was, why you took it in the first place. If someone now tells you, hey, take this cipher with a key three times as long, then, unfortunately, I have to say, well, it's just bad, but it doesn't fit [laughs]"* (G1P1).

If the use of weak cryptography is a requirement or legacy systems need to be supported, a developer must ignore a warning message. Because the warning can be annoying in this case, the developers want to be able to disable it. Nevertheless, participants argue that warnings are not the right place for **deactivation instruction**. They suspect that it would be used carelessly: *"I could copy and paste [the code for deactivation], and then I would have achieved exactly the opposite of what I actually wanted [(security)]. So it's counterproductive at that point"* (G3P5). Documentation is a more suitable place for such information: *"I'd put that on an external source. In my opinion, you want it to be implemented correctly"* (G2P5). Online sources like documentations can easily be found by search engines when needed: *"If someone really wants to do this, they'll find ways to google it"* (G2P7).

Deactivation should only happen temporarily in a developer's local environment: *"[...] I turned this one warning off very locally and still [inadvertently] put two of my warnings off as well, which I didn't see any more"* (G3P5). Developers can also forget to undo the setting and miss important information at a later date. *"You still have to think twice if you want it gone, maybe you need it again"* (G3P2). It is also not desirable to mute a warning for an entire team when working in collaborative development: *"I wouldn't suggest [...] to put it in the code so that other users would check out the project and also wouldn't see the message anymore"* (G2P3).

Surprisingly, up to this point of the study, only focus group three discussed whether **code examples** should be available in the console: *"I mean a proposed solution is always bound to the context. You can't do that, it's always different. Copy and paste would rather not work"* (G3P6). Interestingly, participant G3P2 brought the aspect to the pinboard and changed his opinion later: *"No, I think I would run the risk myself to say relatively quickly, aha, here is the solution, I don't need to think about it. This is also true for others, I suppose (laughs)"*. Other focus groups considered code examples in console messages inappropriate: *"I definitely don't need a code example there. This can be hidden behind a link"* (G4P4). Code examples should also not be hardcoded into warnings because they can age or expire, which means they have to be maintained: *"Yeah, but it can change over time"* (G2P2).

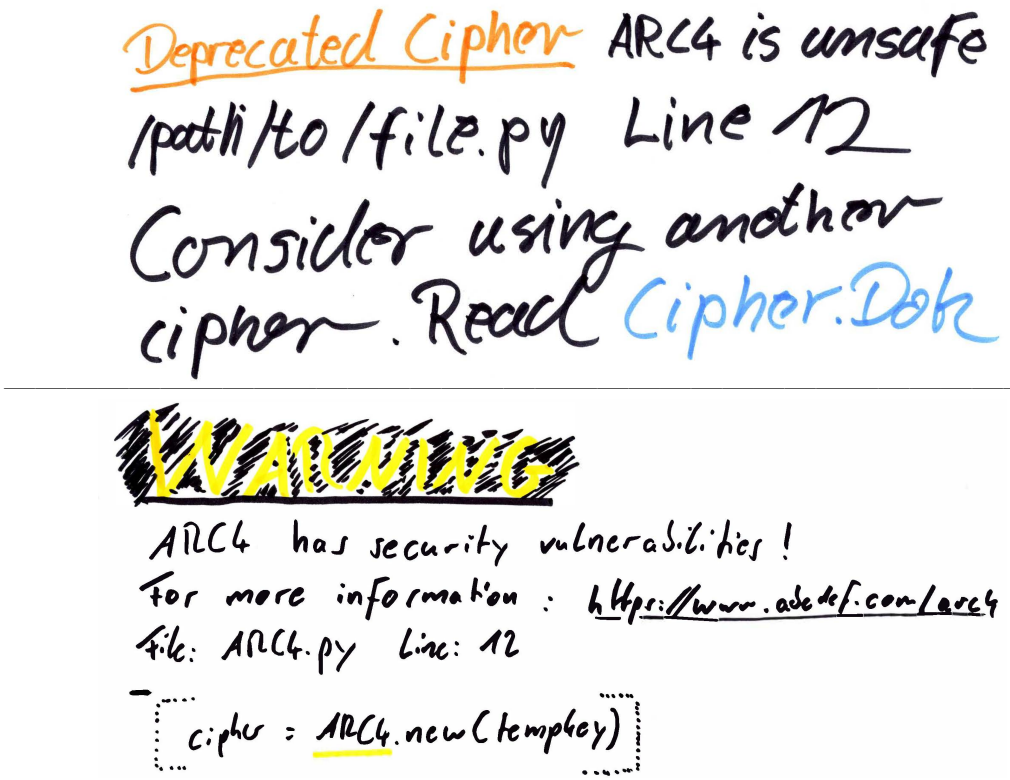


Figure 5.11.: Participants G3P3 (top) and G2P2 (bottom) drew these mockup warnings when asked for their favorite cryptography warning design (step eight of the study protocol). They are similar in content. © ACM

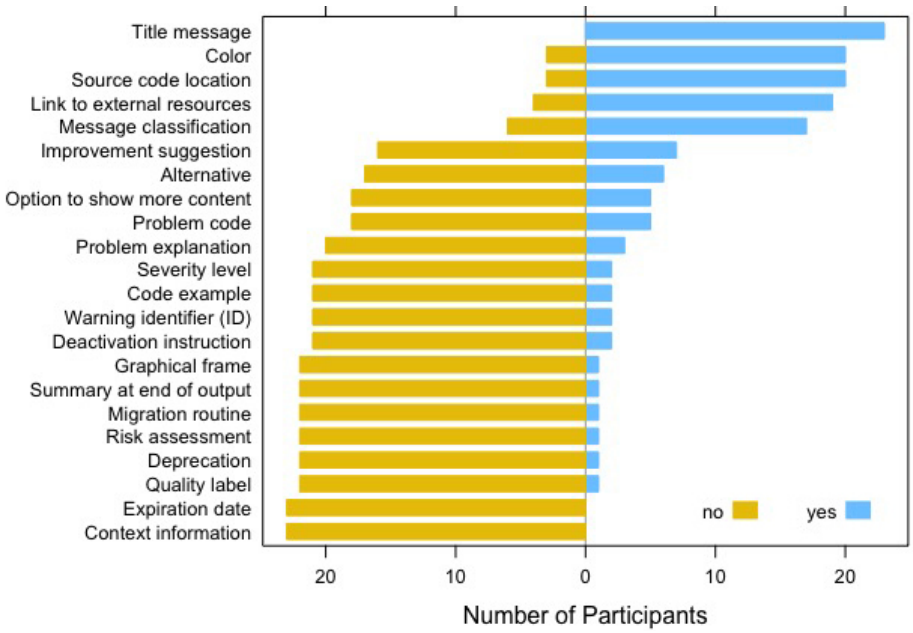


Figure 5.12.: Number of focus group participants that used or did not use a specific item for cryptography warning content in their drawing (step eight of the study protocol). © ACM

Drawings In all four groups, a consistent result for a helpful cryptography warning message design in the developer console was achieved. The individual design drafts are characterized by their short and concise form (cf. Figure 5.11). When evaluating the content of drawings (cf. Figure 5.12), a clear break stands out with a difference of 10 number of uses after the first five design aspects. 23 developers used a **title message** like *"ARC4 has security vulnerabilities!"* (cf. Figure 5.11). 2 participants did not make a drawing. The paper prototypes also revealed the use of **color** (20) as an essential design aspect: *"It is also important to me that the warning is highlighted in color, differently from an error"* (G3P5). Participants described them as a code for message classification and used them accordingly in the drawings. The red color is associated with errors, while yellow or orange is mainly expected in case of a warning. Following the rating result (cf. Figure 5.10), a **source code location** was present in most prototypes. 19 developers decided to include a **link to external resources**. Also, a **message classification** such as "WARNING" clearly identifying the message was considered as a valid part of the warning: *"so*

it's categorized, that you know it's a warning" (G4P3). Although three groups did not mention this aspect in the previous task (7), 17 participants used them in their drawing. Slightly contrary to the rating results, only 6 drawings contained an **alternative**.

In addition to the five most common design elements, participant G3P3 decided to draw a **deprecation** warning and to give an **improvement suggestion**. At the same time, G2P2 integrated an **option to show more content** (cf. Figure 5.11). It offers to show the **problem code** after clicking on the code location.

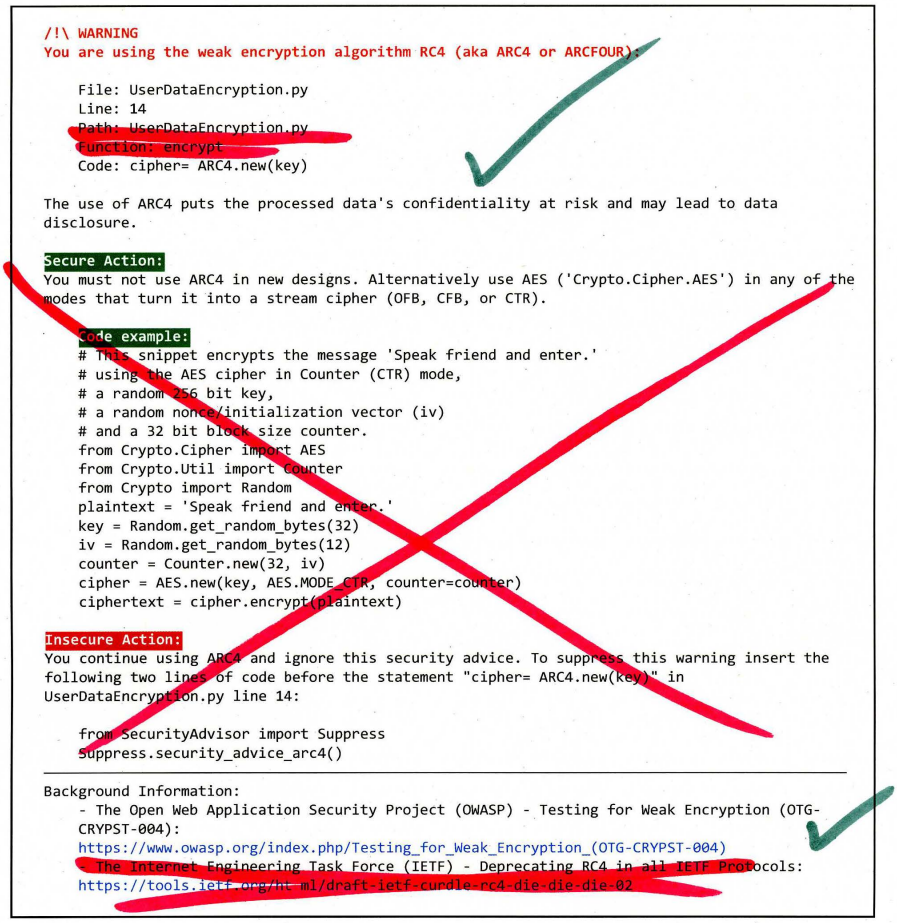


Figure 5.13.: Participant G1P6 revised the warning tested by the previous Study 3 (cf. Section 5.1.2, p. 67) based on the focus group results and his own preferences (step nine of the study protocol). © ACM

Evaluation of a Design Proposal Reactions to the printout (cf. Figure 5.13) emphasize the warning size being out of the question for most participants: *"First, I thought if I could print it out as a PDF (laughs). So this form would be far too much for me"* (G4P2). Because four of the five core design elements (message classification, title message, coloring, and location) are in the upper part, this section was assessed positively: *"[...], but I find the basic warning, here above [...] this should always be displayed no matter what the user configured. I think this is the key information"* (G2P8). Similar to participant G1P6, who made the green checkmarks in Figure 5.13, developer G2P8 wants a link to detailed external resources directly at hand: *"I would prefer to directly have the link up here in 'what's the secure way.' I have to come to the bottom to be able to get more information"*. However, the red color was found to be inappropriate as it usually indicates error messages: *"It's very present up here for a warning. It sounds more like an error, something you can't compile with. If it's so important, then maybe warning is the wrong level. [...] It looks like a warning that wants to be an error"* (G4P1).

RQ4a - Warning Context

The developers reported that a console is an integral part of their toolset. It is needed during development for building and compiling, *"It's a nice summary of all things to see during a build process"* (G3P4). These actions are frequently repeated, especially when testing or debugging code. It is also a useful tool for getting feedback about events at runtime because information about software internal events are typically not shown in the GUI for end-users. *"I don't find any reason for a program crash inside the application. Every relevant information is given to the console [...]"* (G3P6). This statement means the console is a central point of information in software development: *"The console is a central place, i.e., you always look at it during the development process, i.e., you don't have to search long for the messages coming from all possible parts of the system. Rather, they end up there, at a defined step during the development process"* (G4P1).

The focus groups addressed different types of warnings like updates, deprecation, linting, and security warnings. This result means security feedback competes with any other information within a console. Due to the amount of information gathered

in it, the console can appear overloaded, and data processing becomes necessary to find relevant information. *"Yeah, obfuscating output. If you just have something that produces so many warnings that the one important point is not noticeable anymore"* (G1P1). This context explains why a short and concise warning is mostly preferred. Thus, in general, warnings draw attention to problems or negative consequences that could otherwise be overlooked. Participants reported appreciating warnings: *"You can change implementation by a warning message before a real error occurs"* (G2P7).

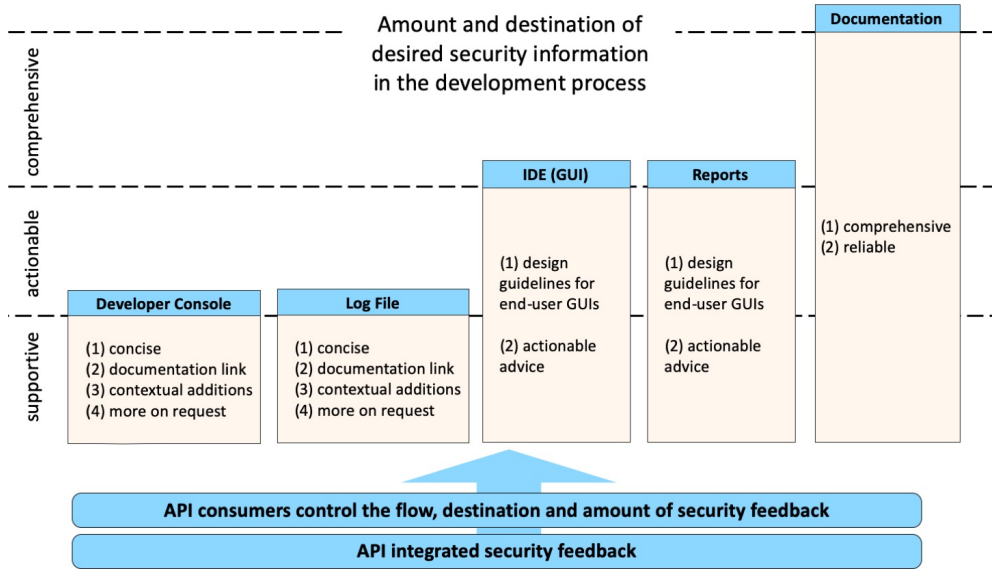


Figure 5.14.: Amount and destination of desired security feedback in the development process. © ACM

While this study set out to study cryptography console warnings, participants were interested in designing not only for the console but felt strongly about different needs for different contexts. Focus groups generally agreed that different use cases call for different types of information. When rating potential helpful warning content (cf. Section "Rating of Helpful Information", p. 100), G1P1 stated: *"A lot of things are so situation-dependent for me. There are situations where it makes total sense and there are situations where it just bothers me"*. To gather more insights into this particular aspect, suggestions were extracted for what developers want in the contexts that were discussed (cf. Figure 5.14).

Developer Console Console cryptography warnings should initially provide only the most important information. Nevertheless, developers value an option to reach information quickly even in the console to bypass searching in the documentation or via search engines: *"I could also just copy the warning and type it into Google and see what the community has done with it. But if I can get the solution by using an optional parameter, then it's really valuable."* (G1P2), *"[...] if you have the possibility to directly output something like that, you'll get much faster with the development"* (G1P5). Also, a specific context like a server environment may require adding more content to a console warning as described by G3P4: *"I would like to have the source code in any case, yes, because it shows me the source of the error. Then I don't have to open a file when I'm on a Continuous Integration server or somewhere else. We are talking about the console here, and I often have log messages in the console but not the file where it comes from at hand. Then I would like to see it at this place in the console."*

Log File Log output shown in a console is not stored permanently by default. One option is to store the information in a log file for later use. However, also in this environment, it is not generally appreciated by developers to put all helpful information in these files: *"I just have a project with about 130,000 lines of MATLAB code that raises a couple of 1000 warnings. I don't want to see a letter page for each one in the log file. Because then I can't do anything with it anymore"* (G3P4). The high number of messages, warnings, or errors is particularly important in this context. Participants speak of storing feedback information in parallel at different locations in different versions. *"I want to be able to pipe it. So in the case of 5000 lines, I can pipe my output into a log file, but still, see my warnings on the console."* (G3P5), *"[...] I can just look in the log afterward and there it is written in detail"* (G3P2). To support this control of information flow, at least two warning versions differing in the amount of information are necessary. It is concluded, for cryptographic warnings, both in consoles and in log files, additional security advice should only be given on request. However, the demand for conciseness does not continue for IDEs and post-processing reports.

IDE An IDE offers significantly more graphical interaction possibilities with the developer than command-line tools. For this type of development environment, participants expressed a clear need for rich, helpful information. Consequently, the additional content of Figure 5.13 (p. 105), including a problem explanation, improvement suggestion, code example, and deactivation instruction were perceived positively: *"I want the IDE to highlight and underline this use of ARC4.new as described here [refers to Figure 5.13 (p. 105)], and if I'd move the mouse over it and see the text, I would say it would be cool"* (G3P3). An IDE also offers an extensive feature set, which can be used to process further the given information of a warning message: *"In the IDE you have a display for problem cases. This is a list with only one headline. You can unfold it, there are additional details. By double-clicking, it jumps to the code position. This is the environment where I want to have this level of detail [refers to Figure 5.13 (p. 105)]. A small side window opens with the detail view, also directly loading this website. Where code snippet exchange mechanisms from the IDE can be used directly"* (G4P1). Participant G3P5 summarized: *"So there are good warnings, there are bad warnings. In many cases, they are still bad and hidden. But especially with IDE tools, it is much more comfortable to work with them if you get your warnings while writing code, not only in the console, afterward"*.

Reports The draft in Figure 5.13 (p. 105) was also perceived positively as a report generated from downstream processing of console output: *"So in Jenkins [automation server software [102]] I'd find that ok in terms of size, but if I saw that locally while developing, I'd find it too much."* (G2P8), *"But definitely, that's how expressive I would like to see it in my Jenkins"* (G2P9).

Documentation Documentation is an integral part of an API [132]. The documentation was often mentioned by the developers when they talked about a link in the warning message. According to the preferences in the focus groups, documentation should provide the most comprehensive information, so that developers can find reliable information directly from the API producers when they need it: *"A reliable documentation where I can find out if I want to do anything about the warning. So I would like to have a decision guide if I want to get rid of this warning now and maybe how I can fix it"* (G3P1). Again, the detail level of the example in Figure 5.13

(p. 105) was accepted in this context: *"If it were inline documentation, I would find it well structured. For a warning message, it is clearly much too long"* (G3P4).

RQ4b - Cryptography Warning Specifics

In this section, four aspects are reported that are specific to console cryptography warnings; three of them deal with decisions on the code implementation level when designing a cryptographic API. By considering the specific context of an encryption algorithm, some participants, surprisingly, questioned their design approach towards the end of the focus group. They raised controversial design questions which are presented by discussing participants' different opinions.

Quantity means importance? While generally perceived as annoying or overloaded, an extensive warning, as shown in Figure 5.13 (p. 105), which tries not only to draw attention to the problem but also gives instructions for action and points out risks, may indicate importance. The spontaneous reaction of developer G2P8 was: *"Wow, that must be a serious error [group laughs]."* Later he explained: *"I had two feelings about it. The first was: too much text. The second feeling was: but okay, if it's that much, it must be something bad."* Participant G3P3 stated: *"In general, I find such a thing too extensive. In the special case of cryptography, however, I think it might be useful. Because this background knowledge cannot be taken for granted in the developer community in general."* G1P1 contradicts at this point, *"But the importance of a problem doesn't depend on how many lines of text my program uses to tell me. It depends on the problem and the program can't decide that. Only I, as a developer, can do that."* G3P4 also took a clear position: *"I would switch it off immediately or change the programming language [group laughs]. It doesn't fit my way of working at all."*

From these statements, it becomes clear that some developers will not accept an overly extensive warning. This characteristic is a quality feature of an API and can be decisive for its success. For this reason, it is recommended considering conciseness when implementing API warnings. Additional content should be tailored as carefully as possible to the needs of API users. If possible, developers should be able to get more information by using a feature of the programming language or environment.

Warning or error? Participants clearly distinguished between warnings and errors: *"At some point, someone has defined a log level. An error is a defect terminating the flow of the program, which causes my application to stop working correctly. That is not the case. The cipher algorithm works, it's just not secure, and therefore it's definitely nothing higher than a warning"* (G1P1). They generally work with errors, and do not necessarily engage with warnings that can be ignored: *"Warnings can be ignored at first [laughs]."* (G1P4), *"I would first ignore the warning and focus on the overall goal that I want to achieve and perhaps later dedicate myself to the problem"* (G1P2). These statements underline the need for further research in the field of API warnings, as this reaction may question the effectiveness of security warnings in consoles. Studies should investigate whether this is a typical behavioral pattern in software development. This behavior could mean that the handling of insecurely used cryptographic APIs becomes an unimportant task once executable API calls have been found, even in the presence of warnings.

An important question is whether and how this risk assessment can be influenced in the development process. However, the participants generally consider security feedback and security console warnings to be important and raised the question of attitude: *"This is a question of conscience, yes, how do I deal with my software, which I deliver at the end"* (G2P9).

Implementing cryptography warnings as errors to increase awareness was not proposed by any group. While the participants did not explicitly mention this possibility, some development tools can be configured to treat warnings as errors [25]. Thus, developers can increase their awareness for warnings if a program crashes, which is also addressed in the proposal of severity levels for cryptography warnings.

Severity Level Concerning ignoring warnings, G2P7 took the issue further: *"Exactly, that's why you have to reconsider if that's another special kind of warning. One that almost goes in the direction of an error. Now you might have to look again if there are still differences in the warning classifications between serious and not so serious"*. The focus groups raised the question, whether a severity level should express the importance of a cryptography warning to indicate what will happen if developers choose not to adhere. Also, estimates of independent specialists could be helpful: *"Perhaps especially with algorithms it is still interesting to know what the*

expiry date is. So, if an authority publishes a list, the algorithm is still recommended until day x, and after that, you might want to think about an alternative" (G4P2). At least two participants pointed out that such an assessment depends on the situational context: *"I think this has more dimensions than just the log level. Because what does that mean in this security area? Is the algorithm quickly crackable, or can it be bypassed? It's a subject-specific decision on how important it is, not a technical one."* (G4P1), *"I think that every developer has to consider what kind of application it is, what kind of impact does it have? Is it an internal tool? Are we developing it for a customer? What would happen if there was a vulnerability? Because that's the risk, and every developer has to keep that in mind"* (G2P5). A severity level could supplement a message classification. Whether this would help software developers assess risk or improve awareness for cryptography warnings and security warnings need further research.

Deprecation For security reasons, some of the participants had the thought to technically deprecate insecure features rather than continue support: *"I'd appreciate it if the API would deprecate the function immediately and in the next version only offer the decryption"* (G3P1). However, this approach would have a different consequence. Users would not be able to update the API if they had to continue using the feature and ignored warnings for this reason: *"But there are also warnings sometimes that say something is deprecated. Well, my goodness, then it's just deprecated, but I just currently need this version. That's the way it is. Then I ignore the thing."* (G2P8), *"For example, because I somehow have to create compatibility to something existing that unfortunately uses this cipher, I don't want it, but I have to, then I'd not want it deprecated [laughs]"* (G3P4). A deprecation process cuts the feature set of an API. Users of the API are forced to react, which can be annoying. However, there must still be APIs available so that developers can create compatibility with legacy systems. Thus, deprecating weak cryptographic features in APIs to avoid insecure use should be considered carefully.

RQ4c - Attitude towards Implementation

Across focus groups, 19 of 25 participants clearly stated, as an API designer, they would implement a security warning into a cryptographic API (one participant had to leave before this question was asked). 5 developers said they wouldn't integrate a warning; 3 would opt for deprecation, and 2 would like to have the security feedback in the documentation, but not in the console: *"I'd rather put it in the documentation, too because I see the time effort that will be immense. And also, the costs will definitely go beyond the scope. You can rather keep documentation clean and reasonable"* (G4P4).

5.2.4. Discussion

The software developers in this study (**RQ4**) wish for console warnings from cryptographic APIs as a tool to prevent insecure API use, and most of them would implement this in their APIs as well. This result supports the finding in Study 3 (cf. Section 5.1, p. 65) that cryptographic API warnings are a helpful and effective tool for software developers.

Until now, only guidelines for end-user warnings could be applied for cryptography warnings in the developer console. Following a participatory design approach, it is concluded that only two of the six design goals of the guideline by Bauer et al. [28] can directly be adopted in the concept of cryptographic API warnings: (1) "Be concise and accurate" and (2) "Follow a consistent layout." The findings emphasize that developers are not end-users, and API producers should not apply guidelines for end-user warnings as a starting point for cryptographic API warning design. Future research in API warnings now has design suggestions at hand that were developed by software developers based on their experiences and opinions, to develop appropriate recommendations.

As the results show, there is a requirement for conciseness for cryptographic APIs in the developer console and log file environment. Study participants suggested five core elements for cryptographic API warnings. At first glance, these can also be applied generically to the context of other API warnings. It is suspected that these are common design characteristics for API warnings in general. However, the participants have expressed higher information requirements for an API warning

in environments such as the IDE or CI reports as compared to the console. Developers partially accept existing end-user guidelines if more information is desired, which indicates an overlap between populations. Some context-specific aspects for cryptography were considered helpful by some of the developers like a severity level, a risk assessment, a recommendation of alternative cryptographic algorithms, or a label to assess a cryptography warning's quality. At this point, uniform design for all types of API warnings is not sufficient. Also, aspects specific to cryptography warnings at the code implementation level were discussed. However, whether these presented design aspects and approaches can be validated as guidelines for specific environments or also apply to other types of API warnings needs to be evaluated through further studies.

In the related field of compiler error messages (cf. Section 2.6.1, p. 17), Barik et al. [25] have formulated three principles to support compiler developers. The study results of this study can confirm that the principle "Allow developers the autonomy to elaborate arguments" is also relevant in the context of cryptographic API warnings (cf. Section "RQ4a - Warning Context", p. 106). Traver [197] theoretically discusses eleven abstract principles for compiler error message design, like "clarity and brevity," "context-insensitivity," and "locality," which can also be confirmed in the context of API warnings.

The participants also expressed some reservations about the approach of API-integrated cryptography warnings because they saw the implementation as an increased workload. However, the results show: An appropriate initial design for cryptography warnings in the console should be concise. The effort to implement this minimum requirement can be regarded as comparatively low in comparison with writing and maintaining documentation. Therefore, the findings may also serve as a starting point for API producers and tool developers.

Reflecting influencing factors on security-relevant information flows in software development (cf. Chapter 3), security API producers' focus should be on providing security-relevant information by designing their entire API environment. That includes the official test environment, the interface design (cf. Chapter 4), the implementation (cf. Chapter 5, p. 64), and the official information environment (cf. Section 2.4, p. 13). However, the results clearly show that, providing all the helpful information that the participants of this study desire (cf. Figure 5.10,

page 101 and Figure 5.12, page 104), is out of scope for API producers. Thus, third party information relating to cryptography, such as specifications, risk assessments, or expiration times, should be published centrally by experts. API producers should make these sources quickly reachable for API users by links, so they do not have to look for it themselves and are less likely to find insecure online information instead.

Due to their concise form, cryptography warnings in the console have the primary purpose of alerting developers to a problem. This constraint means further security-relevant information has to be given to the developers at other locations to support them in handling a problem, e.g., in the runtime environment by log files and post-processing reports, or in the tool environment by IDEs. Therefore, the next chapter focuses on additional information flows for improving security APIs. It examines complex security-relevant information flows of security controls APIs to implement Content Security Policies taking part in web development frameworks.

6. Additional Information Flows for Improving Security APIs

The results of Chapter 5 have led to the conclusion that security API warnings targeted at consoles and terminals can support software developers in implementing secure software (cf. Section 5.1, p. 65). However, the developer console is a central point of information in software development, handling a lot of incoming messages during the development process - not only API warnings. For this reason, software developers want warnings mainly to be concise in this environment (cf. Section 5.2, p. 90). Additional means at other locations have to support them with further security-relevant information to handle security APIs right. Therefore, this chapter focuses on additional information flows for improving security APIs and the extent to which these can be supportive in complex API frameworks for web development.

Web development is an attractive field for many developers because the Web, with its countless application areas, is very closely linked to our daily lives. A rich set of frameworks for many languages is available, providing a ready to use development environment that allows programmers efficiently to develop, test, and publish web applications. Students of computer science with basic knowledge of programming are usually able to develop complex and extensive web applications after a six-month course. In principle, this is an indication of usable software development products. This means, many deployed software products are not produced with the know-how to consider data and application security during the development lifecycle [18]. Due to the high degree of networking, the sensitivity of user data, and legal requirements [193], many secure coding practices for APIs [145] and web applications [144] are mandatory in web engineering. Especially developers unfamiliar with security need support to protect the data of their users and to make their applications secure.

The first study (cf. Section 6.1) examines complex security-relevant information flows of a security control API to implement Content Security Policies taking part in a web development framework by default (**RQ5**). The results reveal several influencing factors leading to usability problems. Section 6.2 (p. 142) is devoted to one of these identified aspects examining, for the first time, how third-party producers of web APIs can usefully integrate security-relevant information concerning security APIs into their API documentation (**RQ6**).

6.1. Study 5: Security by Default in Web Development Frameworks

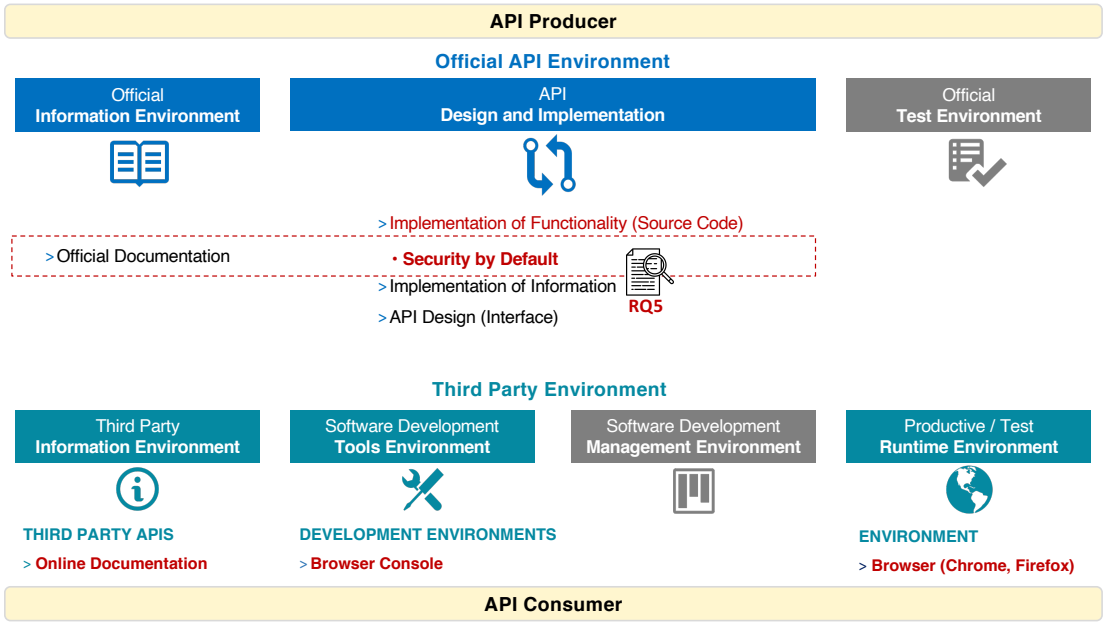


Figure 6.1.: IFSIF model classifying contributions of Study 5.

6.1.1. Motivation

Since security is often a secondary task in software development (cf. Section 4.2.3, p. 52), there are also efforts to equip development environments with security

defaults [66, 115] (cf. Section "3. Adherence to Security Principles", p. 43). This support can relieve developers to a certain degree from designing and implementing every protection feature by themselves (cf. Section "10. Delegation", p. 46), which are necessary to build a development project on a secure basis. Even without a deep understanding of risks and attack models, domain-specific security functionalities can be delivered by non-security specialists out of the box. Many web frameworks take, e.g., the responsibility for implementing defenses against Cross-Site Request Forgery (CSRF) [144] attacks to avoid delegating this task to their users. One indication for this approach to be successful is the vanishing of CSRF attacks from the "OWASP Top 10" list in 2017 after being a permanent member for many years. However, not every security requirement in web development (cf. Section "9. Execution Platform", p. 45) can be covered by default without the developer actively configuring or implementing measures by using security controls APIs (cf. Section 2.9.1, p. 21) which are an integral part of web development frameworks (cf. Section 4.2.3, p. 52).[1]

A proven example are Content Security Policies (CSP) [222], which can be an effective means to protect the integrity of website and web application elements against Cross-Site Scripting (XSS) [146] attacks. However, studies have shown that the policies are rarely used in practice and the configuration is error-prone [212, 147, 209, 40]. Thus, to be able to improve the usability of security defaults in software development frameworks, this study examines security-relevant information flows related to security controls APIs like CSP. This intention led to the following research question **RQ5**: *How does the enforcement of security by default affect the usability of a web development framework?*

This study focused on the Play Framework for Java and its design decision to enforce Content Security Policies (CSPs) [222] by default, to answer the research question. Figure 6.1 shows an IFSIF model (cf. Chapter 3) classifying the contributions of this study to the implementation of functionality which takes place

[1]The contents of the present chapter were previously published in the paper **"Warn if Secure or How to Deal with Security by Default in Software Development?"** by Peter Leo Gorski, Luigi Lo Iacono, Stephan Wiefling and Sebastian Möller, which appeared in the Proceedings of the *12th International Symposium on Human Aspects of Information Security and Assurance (HAISA)*. Dundee, Scotland, UK: University of Plymouth, 2018, pp. 170–190 [86].

in the "official API environment." The author analyzed effects by conducting a laboratory experiment with 30 computer science students. Furthermore, the study evaluated two state of the art design approaches of CSP violations messages in the browser consoles of Chrome and Firefox. Additional relevant aspects were the official documentation of the Play Framework, and the third party online documentation of the Google Maps API.

6.1.2. Content Security Policies

The Content Security Policy standard was not presented as part of the state of the art chapter (cf. Chapter 2) because it is a specific security measure. Some background information about Content Security Policies is required to understand the context of the present study. Therefore, this section first explains what Content Security Policies are and how they work. Furthermore, it gives an overview of previous research results on their practical application and usability[2].

Introduction to Content Security Policies (CSPs)

Content Security Policy (CSP) was initially proposed by Stamm et al. [180] in 2010 and became a W3C security standard [222] since 2012, which is supported by all modern web browsers [54]. A primary goal of CSP is to prevent, mitigate, and report code injection attacks such as cross-site scripting and clickjacking [144] resulting from the execution of malicious content on web pages [46]. CSP restricts the inclusion of content for web pages through a whitelisting mechanism that declares approved origins for the content that browsers are allowed to load. This defense is an additional measure behind input validation and sanitization. Even if an attacker can find a hole through which to inject content, if it does not match the whitelist, it would not be executed or rendered, respectively. Directives bind content types (e.g., JavaScript, CSS, HTML frames, web workers, fonts, images, and other features) to lists of sources from which a CSP-protected web page is allowed to fetch and include resources of that specific type.

[2]The present section is based on "'I just looked for the solution!' - On integrating Security-Relevant Information in Non-Security API Documentation to Support Secure Coding Practices." by Peter Leo Gorski, Sebastian Möller, Stephan Wiefling, and Luigi Lo Iacono. Unpublished.

CSP specifies directives like `img-src` for images, `script-src` for scripts, and `style-src` for styles. Table 6.1 shows a selected list of specified directives. If a policy directive specifies, e.g., the source `https://some.cdn/` for the `script-src` directive, the protected web page fetches scripts only from this specified source. If a malicious injection tries to fetch scripts from another site such as, e.g., `https://-evil.site/`, the browser will refuse to load them. By default, directives are wide open, i.e., if no specific policy is set, that directive behaves accordingly, and no restrictions are applied when fetching resources for the corresponding content type. The `default-src` directive can override this default behavior. If it is present and if no explicit CSP directive for a given content type is present, the `default-src` is applied to it. Generally, this applies to any directive that ends with `-src`.

CSP Directive	Description
`connect-src`	Defines allowed sources for connecting via XML HTTP Request (XHR), WebSockets, and EventSource.
`default-src`	Is applied as a fallback to content types for which the according directive is missing.
`font-src`	Defines allowed sources that can serve web fonts.
`frame-src`	Defines allowed sources for nested browsing contexts using elements such as <frame> and <iframe>.
`img-src`	Defines allowed sources of images and favicons.
`media-src`	Defines allowed sources for loading media using the <audio>, <video> and <track> elements.
`object-src`	Defines allowed sources for the <object>, <embed>, and <applet> elements.
`script-src`	Defines allowed sources for scripts (JavaScript).
`style-src`	Defines allowed sources for style sheets (CSS).
`worker-src`	Defines allowed sources for Worker, SharedWorker, or ServiceWorker scripts.

Table 6.1.: List of selected CSP directives web developers can use to declaratively restrict content types to be loaded from allowed sources only.

The source list in each policy directive can specify sources by the scheme (`blob:`, `data:`, `https:`), or ranging from hostname-only to a fully qualified URI. Wildcards are accepted, but only as a scheme, a port, or in the leftmost position of the hostname. As web pages require to load a bunch of external resources, the list of sources can enumerate distinct sources separated with spaces. The source list also accepts four keywords that require to be specified single-quoted. The `'none'` keyword effectively disables security for assigned content types. The current origin, but not its

subdomains, is matched by `'self.'` With `'unsafe-inline'` inline JavaScript and CSS is allowed. Accordingly, `'unsafe-eval'` allows text-to-JavaScript mechanisms like the `eval()` function in JavaScript.

A developer has three options for explicitly allowing the execution of inline scripts, as well as the application of inline styles inside an HTML file. Either the `'unsafe-inline'` keyword, a nonce-source, or a hash-source has to be specified that matches the respective inline-block. While browsers calculate hash values for inline blocks and compare them with those specified in a policy (e.g., `'sha256-fortytwo42'`), random numbers have to be included both in the policy and in the respective HTML attributes (e.g., `<script nonce=42 src=/trusted.js> </script>`) for comparison. The random numbers must change with each response from a server.

The CSP is a string containing the policy directives that should be applied to a given web page. The policy directives are separated with semicolons. One can use as many or as few of these policy directives as makes sense for the given application and its context. One pitfall is to make sure that all required sources of a given content type are listed in a single corresponding policy directive.

CSPs are specified by web developers using HTTP headers or meta elements in HTML pages. However, the use of HTTP headers dominates [40]. The HTTP response message transports the policies from the webserver to the browser, and the browser side enforces them on a page-by-page basis. Thus, the server needs to send a CSP in every response that requires protection. The CSP standard specifies two distinct HTTP response headers for this purpose. The `Content-Security-Policy` header is used to transfer the policy to the browser so that it can be enforced by it. Browsers communicate detected violations of CSP rules via red-colored messages with small warning icons in their JavaScript console. Figure 6.2 shows messages printed by Chrome and Firefox as an example.

In contrast, the `Content-Security-Policy-Report-Only` header instructs the browser to report any violation of the CSP directive to a specified server endpoint. The letter allows web developers to test policy effects - but not enforcing them - and to monitor any violation events, which may indicate resource changes on the service provider side or attack attempts. Listing 6.1 (p. 127) contains a complete

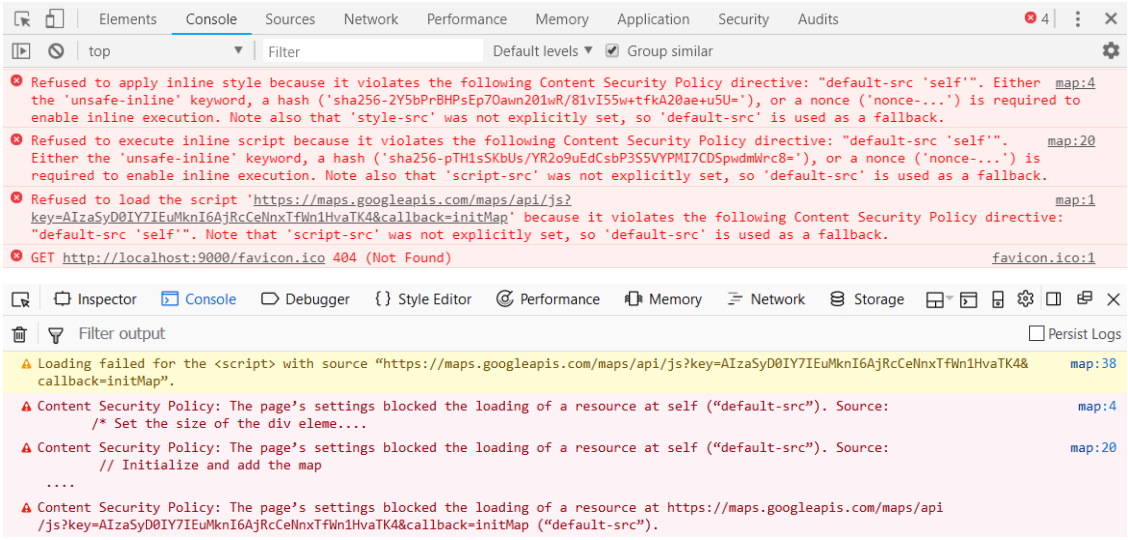


Figure 6.2.: Example CSP violation messages displayed in Chrome (top) and Firefox (bottom). The messages were triggered by integrating Google Maps code into an HTML template file using the Play Framework for web application development. © University of Plymouth

example CSP, which is part of an HTTP response message carrying a web page to the browser.

CSP messages in browser consoles are no warnings in a narrow sense. Typically, warning messages are trying to communicate the risk of an insecure situation to the user by giving crucial information that should help to make an informed decision. Those are following and implementing the “Warn when unsafe” pattern proposed by Garfinkel [75]. In contrast, CSP warnings display information about a CSP rule violation. Thus, this is a message of enforced protection, giving the information that potential risk has been prevented, similar to messages of virus scanners or firewalls. Nevertheless, since CSP messages in a console have the shape of warnings or errors, it should be discussed if this instead follows a “Warn when safe” pattern, which has not yet been proposed in a software development context.

To integrate CSPs, software developers are required to configure their server-side web application to send a corresponding `Content-Security-Policy` HTTP header with each HTTP response. Several web frameworks ease this process and offer possibilities to activate CSPs in their configurations with little effort

Web Framework	**Programming Language**	**Content Security Policy Support: Avalability / Implementation**
Play Framework	Java / Scala	● Available ● included and enforced by default
Spring / Spring Boot	Java	● Available (Spring Security Module) ○ neither included nor enforced by default
Ruby on Rails	Ruby	● Available (since Rails 5.2) ◐ included but not enforced by default
Express.js	JavaScript	● Available (helmet-csp middleware) ○ neither included nor enforced by default
Meteor	JavaScript	● Available (browser-policy package) ◐ not included but enforced by default
Django	Python	● Available (Django-CSP middleware) ◐ not included but enforced by default
Flask	Python	● Available (flask-csp extension) ◐ not included but enforced by default
Laravel	PHP	● Available (laravel-csp middleware) ◐ not included but enforced by default
Symfony	PHP	○ Not available ○ CSP header must be set manually
Gin	Go	● Available (e.g., Secure middleware) ◐ not included but enforced by default
BeeGo	Go	○ Not available ○ CSP header must be set manually
ASP.NET	C#	● Available (e.g., NWebsec library) ◐ not included but enforced by default

Table 6.2.: List of popular web frameworks for the 2017 top programming languages Python, Java, C#, JavaScript, PHP and Go [42] and their available support for CSP. A CSP implementation can be directly available by a framework or by additional software components developers have to add. Additionally, the table shows if the respective software implements the CSP header for HTTP responses by default and when used or manually activated by developers, whether it enforces a secure default CSP.

(cf. Table 6.2). However, in contrast to the seemingly smooth integration of CSPs in web development frameworks, previous work showed that a secure implementation is difficult in practice.

Related Work on CSPs

In practice, the implementation of CSP is a problem. Large scale studies on deployed policies revealed adoption problems, resulting in low usage rates and comprehensive

misconfiguration [212, 147, 209, 40, 165]. In 2016 over 90% of deployed CSPs were not effective, mainly because users compiled insecure whitelists [209]. Furthermore, typos and negligence, ill-formed policies, a lack of reporting, harsh policies, vulnerable policies, and frequent maintenance are identified problems [39, 40, 165]. In this study, similar categories of vulnerable policies were observed (cf. Section 6.1.4, p. 131). Related work analyzed already deployed policies, although researchers have called for research on CSP design and usability [39]. Thus, this study is the first, to the best of the author's knowledge, evaluating usability issues developers have when working with CSP enabled by default in a web development framework.

To support developers, Weichselbaum et al. [209] developed a CSP Evaluator [213], that developers can use to check their policies for security. However, developers still have to create CSPs by a reverse engineering approach, even with semi-automated policy generation [148, 212, 209], and problems in CSP adoption remain [40].

6.1.3. Methodology

The following sections present the methodology of this study to measure the effect of a whitelisting origin-only CSP enabled by default in a web development environment, only allowing sub-resources deriving from the same server as the embedding web page. In this sophisticated setting, security-relevant information flows between the framework (API) producer and API users are analyzed. A specific focus lies on two state oft the art console message designs in the Chrome and Firefox browser, to communicate CSP violations.

Study Design

The author conducted a between-groups laboratory experiment with three groups. A control group worked on a task with disabled CSPs while using the Chrome browser. Another two groups used different browsers: Chrome or Firefox, both with enabled CSP by default. These three conditions were selected to (1) compare the times needed to fulfill a task with the Chrome browser when CSPs are enabled or disabled, and (2) to evaluate whether the different CSP violation message designs of Chrome and Firefox differ in supporting the participants with implementing a

secure CSP. The presence or non-presence of CSPs was not communicated to the participants.

Due to the predominant market share of Chrome [206], this browser was chosen for the control condition. It was assumed that a second control group using Firefox would not be necessary since the solution procedures for the given task were almost identical when using Chrome.

Test persons were advised to use the official Google Maps JavaScript API documentation [80] as a starting point, but they were free to use all available Internet resources. The participants were asked to think aloud during the experiment. The software OBS Studio [177] was used for audio, video, and screen recording and streaming. While observing the experiment, a self-developed audio and video stream annotation software was used to document actions and key events associated with automatically generated real-time stamps. After the experiment, it was possible to add the corresponding video file into the software and jump on click to the according time in the recording for review, analysis, and coding.

In a semi-structured exit interview (cf. Appendix D), additional data was obtained. The questions comprised self-assessments of functionality and security regarding the developed web application as well as basic questions about previous experiences with operating systems, the Play Framework, the IntelliJ IDEA IDE, software development, security, and CSPs.

Task Design

Participants were asked to implement a map into a single page web application using the Google Maps JavaScript API. The task description defined the required size and geolocation coordinates. Additionally, participants were asked to place a visible marker at the location of the Cologne Cathedral. A Google Maps API key was provided inside a text file for use.

The study task was set up with the "Play Java Starter Example" [153] for the Play version 2.6. This official template project was created, especially for Play beginners. Thus, developers find well-commented code and links to further documentation. This template was slightly extended by adding a `/map` resource path to the application by editing routes, adding a controller, and adding a view with a prepared HTML

scaffold for orientation at what code location the map should be integrated. For the control group, CSP was deactivated by editing the application's configuration file as CSP was enabled by default in the original Play project.

The participants developed the application inside the IntelliJ IDEA Ultimate IDE [104]. This environment offered features like syntax highlighting, auto-completion, on-the-fly compilation, assisted refactoring, and debugging. All files and folders within the Model-View-Controller [112] project structure were easily accessible. Participants achieved the task if the Cologne Cathedral was shown by the corresponding browser (Chrome or Firefox), as shown in Figure 6.3. An experiment had been stopped after one hour.

Figure 6.3.: Browser screenshot of a solved task showing the successful integration of a Google Maps via the Google Maps JavaScript API showing the Cologne Cathedral. © University of Plymouth

The task description did not mention the secondary task of handling CSP implementation. The Play Framework enforced the CSP measure by default. Listing 6.1 illustrates how a secure CSP for the study task looked like. In this example, all scripts and styles of the Google Maps documentation example [80] are placed into external server files if possible. This practice helps to prevent XSS attacks and enables flexibility in editing script code. The value `'self'` refers to these files that are placed on the server-side instead of writing script code and styles inline in an HTML file. Script, style, image, and font sources are defined separately. Inline styles which are allowed to be loaded from Google Maps are defined as Base64 encoded

SHA256 hashes. The CSP string has to be placed inside the `application.conf` file of the Play project.

```
contentSecurityPolicy = "
        default-src 'self';
        script-src 'self'
                https://maps.googleapis.com/;
        style-src 'self'
                https://fonts.googleapis.com/
                'sha256-UvsJ5gtL0c/whxmyVt4YoNv7YnPUd0tANZOlq3NshXE='
                'sha256-5KvpSTE2xdGq8rDdvgAPP3mCfTXBc1ohjt2UiAQt9k4='
                'sha256-/VVOq+Ws/EiUxf2CU6tsqsHdOWqBgHSgwBPqCTjYD3U='
                'sha256-a2VR/Wq1VPr0+3GRY+lEmAQm7wjwwnDtPpcCPs2zTrw='
                'sha256-/4YlUlisxufO6wbKvYp4xV8Hkzxm85+1Tb31SjmAox0='
                'sha256-47DEQpj8HBSa+/TImW+5JCeuQeRkm5NMpJWZG3hSuFU=';
        img-src https://maps.gstatic.com/mapfiles/
                https://maps.googleapis.com/maps/;
        font-src https://fonts.gstatic.com/s/
"
```

Listing 6.1: Sample solution of a Content Security Policy for the study task. Line breaks and indentations were inserted to improve readability.

The messages logged into the console of Chrome and Firefox provide different information (cf. Figure 6.2, p. 122). Thus, assuming that participants follow the hints given by the CSP violation messages inside Chrome's JavaScript console, an expected solution will be similar to Listing 6.2. For each CSP violation, Chrome provides a single message giving hints on how to solve them, e.g., *"Either the* `'unsafe-inline'` *keyword, a hash (*`'sha256-[hash value]'`*), or a nonce (*`'nonce-...'`*) is required to enable inline execution."* However, after whitelisting the first given CSP violations for the Google Maps example code, additional CSP violation messages will appear inside the console since the allowed scripts and styles subsequently load additional external resources (fonts, images, inline styles). Five policy adjustment iterations are needed to solve all CSP violation messages. Listing 6.2 shows the compiled CSP result. According to security checks proposed by Weichselbaum et al. [209] this CSP is assumed to be a secure solution. However, due to the presence of `localhost:9000` inside the CSP, the result might not work on a production server.

Chrome does not give a hint to use the `'self'` value, which refers to the same origin (same scheme, host, and port) as the web page.

```
contentSecurityPolicy = "
        default-src 'self';
        script-src http://localhost:9000/assets/javascripts/map.js
                https://maps.googleapis.com/;
        style-src https://fonts.googleapis.com/
                http://localhost:9000/assets/stylesheets/styles.css
                'sha256-UvsJ5gtL0c/whxmyVt4YoNv7YnPUd0tANZOlq3NshXE='
                'sha256-5KvpSTE2xdGq8rDdvgAPP3mCfTXBc1ohjt2UiAQt9k4='
                'sha256-/VVOq+Ws/EiUxf2CU6tsqsHdOWqBgHSgwBPqCTjYD3U='
                'sha256-a2VR/Wq1VPr0+3GRY+lEmAQm7wjwwnDtPpcCPs2zTrw='
                'sha256-/4YlUlisxufO6wbKvYp4xV8Hkzxm85+1Tb31SjmAox0='
                'sha256-47DEQpj8HBSa+/TImW+5JCeuQeRkm5NMpJWZG3hSuFU=';
        img-src https://maps.gstatic.com/mapfiles/
                https://maps.googleapis.com/maps/;
        font-src https://fonts.gstatic.com/s/
"
```

Listing 6.2: Content Security Policy generated by following the advice provided by the CSP violation messages listed in the developer console of Google Chrome. Line breaks and indentations were inserted to improve readability.

Since Firefox provides messages only relating to the violated directive and the refused URL, it was assumed that participants would copy this exact URL to a CSP directive. After doing so, many CSP messages follow, since Google Maps reloads a massive number of external resources (e.g., small images building the map). Since no additional hints are given to add a relative path instead of the exact filename, it was not assumed to be possible to implement a CSP only with the information given by Firefox CSP messages without consulting external resources.

The study design, experimental infrastructure, and procedure had been sequentially pretested in a pilot study with three test persons. These helped to test and revise the study infrastructure, the exit interview, and the task design. Their results were excluded for analysis.

Experiment Procedure

The experiment took place in a usability laboratory. It was designed to create a pleasant atmosphere for the participants. It was furnished comparable to a living and working room in order to avoid appearing as a "sterile experiment room."

Participants were assigned by round-robin to the three experimental conditions. After welcoming the participant and offering water and candy bars, a consent form was explained and signed by all participants of the study. All participants were of legal age. The author's institution does not have a formal IRB process for computer science studies, but the study complied with the European General Data Protection Regulation [193]. Following a determined protocol, all participants got a short briefing about the development environment comprising of the Play framework, the IntelliJ IDEA IDE, and the respective browser development tools. After the briefing, participants got a task sheet explaining and illustrating the task. After clearing general questions of understanding, the study moderator started the recording and streaming, asked the participant to start, and left the laboratory. The moderator followed the experiment by observing all participant's actions via a screen, video, and audio transmission to a room next door, logging relevant actions with an annotation software.

After finishing the task, participants responded to the questions of a semi-structured exit interview. In a debriefing, the study moderator explained the purposes of the study. Participants were asked not to talk about these contents to other students until the study was over.

Recruitment

In total, 43 bachelor and master students of the author's faculty were recruited via email, knowing that they had previous experience with web development, Java, JavaScript, and the Play Framework. The invitation email asked them to participate in a study about web programming. It explained that this study consisted of a small programming task on the premises of the university, which will take about an hour. The email did not mention a security context or the Play Framework. Five candidates rejected, and eight did not answer the invitation, even after sending a reminder. The remaining 30 candidates voluntarily participated in the study

within three weeks in May 2018. The author could not compensate them for their participation but offered water and candy bars for their wellbeing.

Limitation

The presented results are bound to several study-specific factors, which are discussed in the following section.

First, the recruited group is not representative of the heterogeneous group of software developers as the sample of participants came only from Germany. A student sample was selected, as prospective developers or junior software developers are an essential target group, with minor experience, who need support from their development environments to use security APIs effectively. Thus, only candidates with experience in web development were invited (cf. Section "Participants", p. 131). Previous research could not find significant differences between professional developers and computer science students (cf. Section 2.2, p. 10).

Second, the study was designed to evaluate the effect of enforcing CSP by default in a framework for web application development in the first place. The time needed by participants to open the JavaScript console in a browser and to notice a CSP violation message varied considerably between both groups (cf. Section "Task Solutions", p. 132). Thereby, the time participants were dealing with these warnings also varied. Thus, the results only allow drawing limited conclusions on the effect of different warning message design.

The study design mainly limited the time frame to one hour, as the participants offered their time voluntarily. It is not clear if more time to work on the task would have changed the results on functionality or security. The study moderator terminated ten experiments in total (seven in the Chrome condition and three in the Firefox condition). However, in these cases, participants seemed to be slightly annoyed and unmotivated to continue. That was clearly shown by comments like *"That is terrible"*[3] or *"I have no clue what causes this error."* Thus, it is assumed that demanding participants for more time would not have been appropriate for this specific study design.

[3]All quotes have been translated into English.

The task did not include handling security-relevant data. Thus, in the laboratory experiment, a real risk was missing. This context is likely different in real-world settings, potentially resulting in developers having a different motivation to configure CSP. Seven participants decided to disable the CSP mechanism. However, unawareness and lack of appropriate documentation are problems, likewise occurring in realistic settings - large scale analyses proofing CSP deployment to be challenging support this argument (cf. Section “Related Work on CSPs”, p. 123).

6.1.4. Results

The following sections present the data analysis results. First, participants are characterized. The analysis of task solutions compares the control condition with the Chrome condition to evaluate the effect of enabled or disabled CSPs on time needed to fulfill the task. Further, it evaluates whether an effect caused by different CSP violation message design approaches implemented in Chrome and Firefox could be observed. Finally, the functionality and security of task solutions are evaluated.

Participants

The study took place in 2018 and was completed with 30 participants, ten in each condition. Their age was between 19 and 31 years (mean 25.2, SD: 3.1). Only one of the participants was female; all others were male. Nineteen were bachelor students (mean semester 8.0, SD: 2.2), and 11 were master students (mean semester 2.2, SD: 1.2).

Except for one person in the control group, all had previously developed at least one web application in a university course. Their mean experience with software development in total was 4.1 years (SD: 2.3), 4.8 years (SD: 1.9) with the programming language Java and 2.4 years (SD: 1.4) with JavaScript. The lower value for the total experience in comparison to Java experience can be explained by the students not counting their first semesters of learning programming (typically with Java) to their experience with developing software applications. This behavior was revealed during the interviews. 73% (22/30) had already been paid for programming, which proves a certain level of professionalism in software development.

One third (10/30) never considered any security measure during software development. The others mentioned the application of scattered security functionalities in the interview, including e.g., secure connections via TLS, input validation against cross-site scripting, authentication, authorization, session cookies, and security tokens. Thus, the overall experience with security in web development in this sample can be described as limited.

The participants were familiar with the offered development environment. Thus, during the experiments, no problems were encountered caused by the experiment setup. Only two participants in the Chrome condition were not familiar with the Windows operating system, and four had not used the Play Framework before (Control: 3; Chrome: 1). However, the latter four were able to finish the task comparably fast. Sixteen were familiar with the IntelliJ IDEA IDE, evenly distributed on the conditions (Control: 5; Chrome: 5; Firefox: 6). All participants reported having used browser developer tools before.

Task Solutions

Figure 6.4 illustrates the distributions of the total time needed by participants to solve the task in all three conditions. First, the Control condition is compared to the Chrome condition. In both groups using the Chrome browser, the independent variable was the default CSP behavior of Play. The mean time in the control group was 16 minutes (median: 9.9) in comparison to a mean time of 56.6 minutes (median: 60) in the Chrome group with CSPs enabled by default. The samples in the Chrome group are not normally distributed as seven of ten experiments in this group were aborted after a maximum task duration of one hour. Thus, the upper quartile is also the median (60 minutes). There is no time overlap between both conditions (Mann-Whitney U test; U: 0; $p<0.001$). Thus participants trying to solve the task in the Chrome group being confronted with CSPs needed significantly more time than participants in the Control condition, with seven of ten participants exhausting the maximum study time of 60 minutes. Also participants in the Firefox condition with enabled CSP by default needed significantly more time (Mann-Whitney U test; U: 8; $p<0.001$) to solve the task than participants in the Control condition with Chrome and disabled CSP (cf. Figure 6.4).

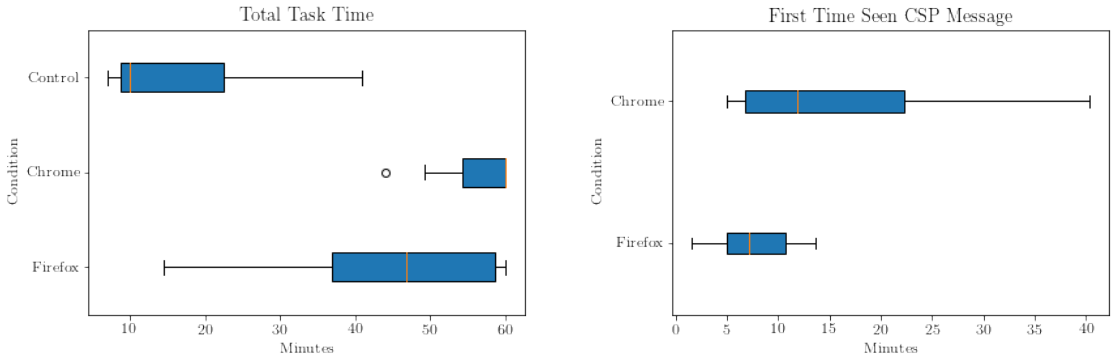


Figure 6.4.: Comparison of total times to solve the task in all three study conditions (left) and of first time seen a CSP Message (right). © University of Plymouth

Second, the processing time between the Chrome and the Firefox group is compared, both conditions with enabled CSP by default (Mann-Whitney U test; U: 24.5; p: 0.022). The participants using the Firefox browser finished the experiment in a mean time of 43.8 minutes (median: 46.9). Thus, participants finished the task significantly faster in the Firefox condition ($p < 0.05$). In this group, three experiments needed a complete time frame of 60 minutes. Thus, in total, the study moderator terminated 50% of all 20 experiments, which included CSP by default, after one hour. In each condition with CSP enabled by default, three participants had used CSPs before. However, none of them configured the policy securely. Three of these six participants did not solve the task in the given time frame of one hour.

Functionality and Security

To be able to assess whether a developed task solution was functional and secure, it was coded. The solution was rated as functional if the browser showed the map as demanded by the task description (cf. Figure 6.3, p. 126). For security, the CSPs were evaluated following the checks proposed by Weichselbaum et al. [209]. Thus, a solution was rated as insecure if the keyword `'unsafe-inline'` was used without setting a nonce-source, an object-source was not explicitly nor implicitly set by the fallback `default-source`, or a wildcard was used in a whitelist. Also, implementations that had switched off the CSP were coded as insecure. The coding

results were compiled independently by the author and one member of the author's working group. A discussion on mismatch solved discrepancies. The security of non-functional solutions was not rated.

All solutions in the Control condition were functional in comparison to nine in the Firefox group and only four in the Chrome group. The seven participants with non-functional solutions did not see any map in the browser during the experiment. Only one did not open the browser console and, consequently, did not see a CSP message. Three test persons saw and read the messages but were not able to find out how to proceed with the given information. The last three of the non-functional group tried to configure a CSP but were not able to find a policy that would have allowed to show the Google map.

In the semi-structured exit interview, participants were asked to rate their agreement to the statements: "I solved the task correctly" and "I solved the task securely" on a five-point Likert scale (strongly disagree (1); disagree (2); neutral (3); agree (4); strongly agree (5)). The objective results for functionality are also represented in the subjective ratings with mean values of 4.8 (SD: 0.4) in the Control, 4.3 (SD: 0.9) in the Firefox, and 2.8 (SD: 1.3) in the Chrome condition.

None of the 10 participants in the Chrome and Firefox group developed a secure solution. Table 6.3 lists the reasons for insecure task coding, which appeared in the experiments.

	Chrome Condition	**Firefox Condition**
Not functional	6	1
`'unsafe-inline'`	1 `script-src`, 2 `script-src` and `style-src`	2 `script-src`, 1 `script-src` and `style-src`
Wildcard	-	1 `default-src *`
CSP = null	1	3
All security header filters disabled	-	1
All default filters disabled	-	1

Table 6.3.: Number of non-functional results and reasons for insecure task coding

It is striking that six test persons in the Chrome condition produced non-functional results. In the Firefox group, it is remarkable that six participants just allowed all external sources to be included on the web site by disabling CSP completely or setting a default wildcard. The two test persons who disabled Play filters got the information from the official Play documentation web site. Two others copied

and pasted insecure code snippets from Stack Overflow posts. The remaining two unsuccessfully tried to configure a CSP but ended up disabling the mechanism. The seven test persons in total who switched off CSP rated their solution's security as low with a mean rating of 2.6 of 5 (SD: 0.9; median: 3.0). In the interview, they explained; *"I set CSP to null (laughing) to bypass the error. Afterward, I didn't configure it in more detail," "I had no choice,"* or *"I disabled CSP to make it work."* The six participants who found a working CSP configuration using the directive `'unsafe-inline'` rated their application to be secure with a mean rating of 3.6 of 5 (SD: 0.7, median: 4.0). In fact, the application of `'unsafe-inline'` makes a CSP insecure.

In the following, it is analyzed if observations in both groups are related to the different CSP violation messages of Chrome and Firefox (cf. Figure 6.2, p. 122). Participants in the Chrome condition saw the CSP message in the browser console in mean after 17 minutes (median: 11.9) in comparison to a mean time of 7.5 minutes (median: 7.2) in the Firefox group (cf. Figure 6.4). The participants in the Firefox condition found the messages significantly ($p < 0.05$) earlier in the experiment than in the Chrome condition (Mann-Whitney U test; U: 23.0; p: 0.0396). However, the developer tools in Chrome and Firefox are almost identical to use, and every participant got a briefing on these tools (cf. Section 6.1.3, p. 129). Thus, it is assumed that the significant difference is a result of software development being a highly creative task, typically not following one specific behavior pattern. Another reason for the significant difference between the Chrome and Firefox groups could perhaps be found in some individual characteristics of participants, which have not been measured. No significant difference in the overall development experience between the two groups could be found.

The first three reactions of each participant to the CSP violation messages in the browser's console were diverse. They range from (a) initially ignoring it over (b) editing code on spec, (c) searching and checking project files to (d) reading in Play and Google Maps documentation, and using a search engine. However, a statistically significant difference for subsequent actions relating to further information needs could not be found. The time difference measured from the moment participants saw a CSP message until using Google Search was not significant between the Chrome and the Firefox group (Firefox mean: 3.1 minutes, SD: 2.1; Chrome mean 3.6 minutes,

SD: 2.9). Neither for entering CSP related keywords like, e.g., `default-src`, `'unsafe-inline'` or "content security policy" into Google Search or the search function of Play's official documentation the first time after having seen the CSP message (Firefox mean: 7.6 minutes, SD: 9.8; Chrome mean 12.1 minutes, SD: 16.0).

6.1.5. Discussion

The enforcement of the fail-safe defaults principle [167] has a significant adverse effect on the usability of a web development framework (**RQ5**) if security-relevant information flows are not considered adequately in the design process. The study results reveal that this design decision on the API implementation level poses particular challenges for other IFSIF factors (cf. Figure 6.1, p. 117). These are the official framework (API) documentation, CSP messages in browser consoles, IDE support as well as security-relevant information given by third-party service providers. Most of them are outside the sphere of influence of framework producers. Thus, improving these influencing factors of security-relevant information flows is a distinct approach aiming to empower software developers to write more secure web applications. The following sections discuss the findings in detail.

CSP by Default

It came as a surprise, and it was an unexpected situation for all participants in the Chrome and Firefox condition when the map did not show up after integrating the code example provided by the Google Maps API documentation: *"I'm not allowed to call the Google Maps API because of any security settings. This usually works."* There is a design shift from "developers have to switch on security" to "developers have to manage security directly." The results show a negative effect on functionality and security: 35% (7/20) of the participants in the Chrome and Firefox group with CSP by default were not able to achieve a functional solution, and none of the 20 participants developed a secure solution.

The current design of the default origin-only enforcing CSP in Play resulted in developers being uncertain about the functionality of their code. This uncertainty had been further amplified by a lack of information about what was happening. Consequently, errors were assumed to be in the own source codes and caused the

revision of various project files for functional or logical mistakes. The common trial-and-error strategy failed as the security by default mechanism blocked the trials and left the developer with no additional hints on the cause of the non-functioning application. This lack of information leads to an in-place security mechanism being confused with flawed code. The observed program behavior is perceived as a mistake rooted in their implementations. To ensure that this is not the case or to finally get some insights into the problem cause, participants even used third party runtime environments like JSFiddle [107] or created local HTML files outside the Play project to test their code. Two participants disabled all security headers (cf. Table 6.3, p. 134) to identify the cause of the problem. Thus, the CSP integration design can lead to entirely disabled security headers, also affecting other security defaults or settings.

This result emphasizes that informative feedback is crucial for developers to support them in narrowing down the real cause of a problem. Warning messages in the browser console are one pillar in this respect. This study showed, however, that an adequate information flow requires additional feedback at more locations and layers within the development environment and during the development process, respectively. As the developers do make extensive use of documentation, the enhancements of documentation with security-related aspects are one such missing piece. Thus, also service providers have an information obligation (cf. Section "6. Information Obligation", p. 44) to add missing CSP-related information to their documentation (cf. Section "Information Resources", p. 140). Web frameworks do have a particular mode of operation while developing a web application. When this mode is enabled, the framework provides detailed error messages in respect to compile or runtime errors. This feature could be enhanced with feedback on the current state and context of the application. Along these lines, such a framework provided feedback could link to additional resources such as CSP generators (cf. Section "Tool Support", p. 137).

Tool Support

During the experiments, none of the participants searched for a CSP generator or related tools. After every change to a code or style block, a new hash value has

to be generated, e.g., a Base64 encoded SHA-265, in order to update the CSP. An IDE integration could help developers with this process by offering the possibility to add or replace a hash to the CSP configuration file automatically or calling this support for specific inline style or script blocks. Also, the `'nonce'` directive could be directly supported by the framework. Promising future work is to design and evaluate tool support for CSP deployment. These could be tools inside the IDE or also tools offering external support.

What is triggering the message and what is the reason? CSP messages in Chrome and Firefox are designed in red color to gather attention. Indeed, they were perceived by 19/20 of the participants when they opened the JavaScript console of the browser. They were mainly interpreted as errors: *"Always the same error message, I've no idea what I can do with it," "An error is displayed."* In fact, these messages give information about an enforced CSP which had been violated. Thus, the message is part of a working security mechanism in a secure situation, which is perceived as an error. This secure situation had been changed to a more insecure situation by 13/20 participants by disabling CSP and even further-reaching other security means.

Adding a correct policy entry could result in many more messages showing up at the next test run. The number of warnings displayed in this study was critical. Participants were confronted with many messages at once, with a maximum number of 105 triggered by loading many small images building a map. Participants had to iteratively edit the CSP to configure scripts, styles, fonts, and images and verify the effect on messages displayed in the console. This required iterative process was a hurdle. When participants just deactivated CSP, all messages disappeared.

Participants who switched off CSP exactly knew that it was lowering security, but disabled the security mechanism to be able to solve the given task (cf. Section "Functionality and Security", p. 133). Thus, knowing that there is a more secure implementation could lead to a responsible reaction in production, maybe delegating the task to other developers or investing more time in learning how to manage CSPs. This finding might also indicate that the relevance of CSP to the application's security was assessed high. Thus, the effect of the default enforcement

of CSP on the relevance perception of the developers is an interesting topic for future work.

How to configure CSPs and where to set them? In comparison to Firefox, Chrome includes the information that it is required to add `'unsafe-inline,'` a hash, or a nonce to the policy to enable inline execution. However, no recommendation for the offered three directive options is given. Only one participant in the Chrome condition tried to add the given hash value to the policy. No one tried to add a nonce. Except by the name, there is no indication that the use of the `'unsafe-inline'` directive will result in an insecure CSP configuration. The rather high-security ratings of participants that used the `'unsafe-inline'` directive seem to reflect the demand for more and better information. Further studies could evaluate whether showing a specific console warning could help to advise on how to improve a CSP when the `'unsafe-inline'` directive is applied.

It was also a problem for test persons to find out where the CSP should be configured, manifested in numerous comments on the warnings like *"What is* `default-src` *and where do I have to set this? or I have to google* `default-src` *because I have no idea what this is."* Participants tried to edit or change CSP at various places of the framework, not only in the recommended application configuration file but also e.g., in controller files, filter files, or in meta tags inside of view files. Settings in the `application.conf` file will override default settings loaded from different files [152]. Thus, commented configuration lines in the `application.conf` file do not mean to disable a default configuration. They instead match the defaults and are just commented for documentation purposes in starter Play projects. This design did not match the participants' expectations of how a configuration file works. Instead, a list of enabled features was expected, which can be disabled by commenting them out.

CSP Violation Message Design

The design of CSP violation messages does influence the information flow, too, as the situation hardly changed after participants had seen the CSP violation message. They had difficulties in understanding the given indications in the browser

console: *"Honestly, I have no idea why the map isn't displayed. The messages do not really explain that. It can't be so difficult."* As found in Study 3 (cf. Section 5.1, p. 65), in the context of primitive security APIs, warning messages displayed in development consoles can be an effective measure to improve software security. Browser CSP warnings are similar to security API warnings, as they both use a console environment for the related purpose of supporting developers with security. However, CSP warnings presented by Chrome and Firefox could not help any of the participants to configure a functional and secure policy as information to four main questions was missing or unclear: What is triggering the message? What is the reason? How to configure CSPs? Where to set them?

Applying the five core design elements for security-related console messages proposed in Study 4 (cf. Section 5.2.3, p. 100) to these state of the art message designs identifies potential points for improvement. The messages should be classified and colored as warnings instead of errors. The Chrome browser does not use a title message, avoiding developers to quickly understanding the message reference. Both browsers provide code locations, but both are missing links to supportive external resources. Due to their concise form, the primary purpose of these messages is alerting developers to CSP violations. This constraint means, additional information resources at other locations have to support developers with further security-relevant information to answer their questions.

Information Resources

The information given by the browser CSP messages was not enough. During the study, 18/20 tried to get further information about CSP by using Google Search or the search function in the Play documentation. Participants searched for CSP information also directly on the Google Maps documentation website [80]. As the CSP for the task should whitelist this Google service only, the idea of getting a ready to use CSP directly from the service provider is plausible. However, the official documentation [80, 213] does not provide a ready to use policy as explicitly demanded by one test person: *"Oh man, just give an example!"*. In the case of the deployed Google Maps service, this would perfectly fit in the already given "Tips

and troubleshooting" section, which was also read by participants. At least there should be explained how to gain a secure configuration for typical use cases.

It is promising future work to evaluate if specific web-service-related information about CSP configuration and ready to use examples can support web application developers in using security APIs. Precisely this aspect is the research question of the next and last study of this thesis to investigate whether the security by default approach can be improved.

6.2. Study 6: Secure Code Examples in API Documentation

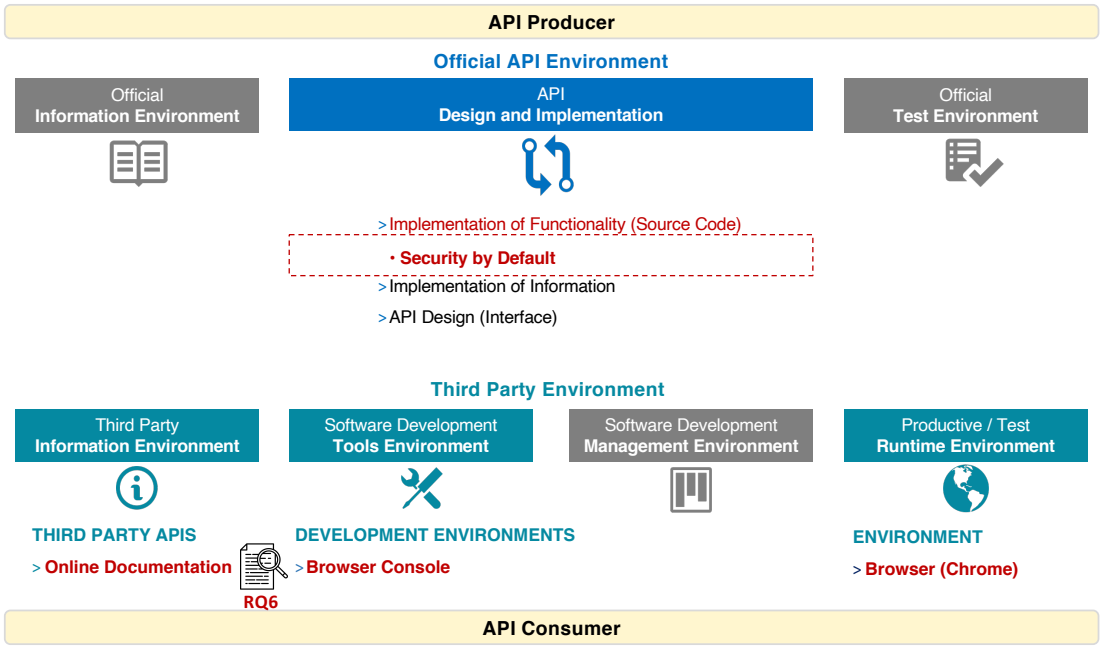


Figure 6.5.: IFSIF model classifying contributions of Study 6.

6.2.1. Motivation

The qualitative laboratory Study 5 (cf. Section 6.1, p. 117) identified usability problems developers have when working with Content Security Policies (CSPs). It was found that the given information and support by warnings of browser developer tools, the web development framework, and its documentation, the API documentation, and third party resources from the Internet were not sufficient. The solution to the problem should not be to abandon or remove security by default because of the problems [154] but to investigate what additional information web developers need to implement a secure application and how tools and documentation can reliably provide this information.

It is wrong to say that developers do not read the documentation of an API.[4] Instead, they choose the information sources that best meet their information needs from the full range of available sources. When having a free choice, also the official materials of software vendors are appreciated (cf. Section 2.4.2, p. 14). Software developers today build complex systems based on plenty of third-party libraries. Library documentation is a key to understanding and using the functionality provided via the libraries' APIs. Furthermore, API documentation increasingly requires the consideration of cross-cutting concerns such as security to meet today's requirements. However, documentations of JavaScript libraries for use in web browsers, e.g., do not specify how to add Content Security Policies (CSPs) to protect against cross-site scripting attacks.

Due to the high degree of networking, many aspects of web development implicate security requirements. Developers can take advantage of over 22,000 web APIs [156] to add functionality to their applications. Even if these are not directly security-related, such as a cryptographic API, the integration of a web API such as Google Maps can result in security requirements. Security-related information should also be part of the API documentation, to assist developers in dealing with these requirements reliably (cf. Section "7. Degree of Reliability", p. 44). API vendors provide essential and trustworthy first-hand information. In this way, each API provider could supply its consumers with a tailored CSP for their service. After all, who should know better what resources a web application requires to integrate to use an API as intended and which entities deliver them? Before such a demand can be addressed to the numerous API providers, it should first be understood whether this approach can help developers to implement a secure CSP in a web development framework. Research has not yet answered the question of how to usefully integrate security-relevant information into the documentation of a web API, which leads to the following research question (**RQ6**): *How can documentation writers usefully integrate security-relevant information into a documentation website of a web API?* Thus, the contributions of this study address the "third party

[4]The content of the present chapter is based on **"I just looked for the solution!" - On integrating Security-Relevant Information in Non-Security API Documentation to Support Secure Coding Practices.** by Peter Leo Gorski, Sebastian Möller, Stephan Wiefling, and Luigi Lo Iacono, 2020. Unpublished.

information environment", as illustrated by the IFSIF model (cf. Chapter 3) shown in Figure 6.5.

In this study, four more specific questions will be used to answer the central question: (**RQ6a**) How do developers deal with documentation, and to what elements do they pay attention? Eye-tracking was used to analyze the behavior of developers and to identify areas of attention by evaluating heatmaps for API documentation. (**RQ6b**) Does it make any difference at what places documentation presents security-relevant information? In a between-groups lab study, the original Google Maps JavaScript API documentation was compared with three different prototypes. Placement effects of security-relevant information on the developers' performance were studied, analyzing gaze data distributions. (**RQ6c**) Does CSP documentation affect the functionality and security of participants' programming task solutions? Programming task results were evaluated whether the documentation approach supported developers with CSP configuration and thus with integrating a functional and secure Google Maps API into a web page. (**RQ6d**) Does CSP documentation support the developers' comprehension of CSPs and their confidence in the security of their web page? After the experiment, the developers were interviewed to gather supplementing qualitative data.

This study made the following contributions to answer these questions: (1) An eye-tracking lab study with 49 prospective software developers was conducted. (2) Three different documentation prototypes to integrate security-relevant CSP code examples and information into API documentation had been designed, implemented, and tested. (3) It was evaluated in detail how prospective developers work with the API documentation and how effective the approach was. (4) Limitations were identified. This study contributes a first in-depth understanding of security-relevant code examples in API documentation examining a novel approach to help developers adopting API functionality securely. It concludes with a recommendation for API providers to include CSP examples into their API documentation.

Since prospective and junior developers were an essential target group of this study, a student sample was selected (cf. Section 6.2.3). As most of the participants were near the end of their studies, and some already got paid for developing software (cf. Section 6.2.4, p. 156), the following sections refer to them mainly as developers.

6.2.2. Security in API Documentation

None of the previous usable security studies related to API documentation (cf. Section 2.4.2, p. 14) examined the integration of security-relevant information into online documentation of web APIs, which do not have a security reference. Thus, in addition to the discussed state of the art in security-related documentation, this section summarizes related work dealing with API documentation in general. Differences and similarities to the approach investigated in the presented Study 6 are pointed out. An introduction to Content Security Policies can be found in Section 6.1.2 (p. 119).

Based on survey results, Uddin and Robillard [201] presented ten types of documentation problems concerning the content and presentation. In the present Study 6, these two aspects were precisely examined, namely how the presentation of security-related content in API documentation affects the security of a web page using the API. Inzunza et al. [94] have compiled a list of ten elements, such as "overview," "sample code," and a "get started guide," representing minimum requirements for complete API documentation. The list does not take security aspects into account. This present study enhances one single Google Maps API documentation page, which does not have to fulfill all of these elements.

Meng et al. [127] found, conducting interviews and a questionnaire, that developers follow mainly two different concepts when working with API documentation, which should both be supported. They either adopt a concepts-oriented or a code-oriented learning strategy. In the context of the present study, the test persons clearly showed code-oriented behavior. In an observation study with 11 developers, Meng et al. [128] also examined how developers use API documentation. Based on their results, they proposed high-level guidelines for designing API documentation in three main aspects: (1) Enable efficient access to relevant content, (2) Facilitate initial entry into the API, and (3) Support different development strategies. All guidelines are focused on the main functionality of an API and do not consider cross-cutting requirements like security. One important guideline when supporting different developer strategies is "Signal text-to-code connections." To the best of the author's knowledge, this is the first study investigating the opposite approach, "Signal code-to-text connections." Meng et al. [128] also clearly identify the need for

"research providing a more fine-grained analysis of the information units, such as text versus code examples, which developers attend to when using a certain section of the documentation." This study presents to the best of the author's knowledge the first results of an eye-tracking study offering fine-grained results in how developers use web API documentation.

6.2.3. Methodology

In a between-group laboratory experiment, software developers were asked to solve four web development tasks. For this purpose, they were provided with a web development framework, where several security mechanisms took effect by default – including a strict CSP – to support them in implementing a secure application. The core of the experiment was the integration of a Google Map on a web page. In this step, the default security setting from the server-side intervened and prevented browser clients from loading external Google resources like scripts, styles, fonts, or images. This behavior means the security default prevented the participants from fulfilling the given primary programming task. For the web page to display a map, participants had to configure a CSP whitelist for third party resources accordingly (cf. Section 6.1.2, p. 119). The previous Study 5 (cf. Section 6.1, p. 117) has shown that inexperienced developers have severe difficulties in coping with this security default because they lack the needed information. An original Google Maps API documentation page was enhanced with information, including an API specific policy example, to support the participants also with the secondary security task of implementing a secure CSP for their web page requirements. Security-related information was integrated in three different ways into the original Google Maps API documentation, and the effects of these different versions were examined.

One study run lasted about two hours. It consisted of a preliminary discussion, a briefing, the experiment, a structured interview, and a debriefing.

Preliminary Discussion & Ethics In the preliminary discussion, the test persons first read and signed a consent form confirming that they were of legal age. There was no formal IRB (institutional review board) process at the author's university.

Nevertheless, the study kept the data handling requirements of the European GDPR [193].

Previous work found that fewer study participants ignored security when it was an explicit part of the task description [136]. In this study, the participants could not ignore security because of defaults, but they could decide to deactivate the security mechanism. As the handling of the documentation should not be influenced by a specific security task, the tasks had been framed [200] in order to counteract deactivation decisions. Therefore, at the end of the preliminary discussion, the study moderator pointed out that "it should be a secure solution." Thus, it should be ensured that the participants saw the application's security as a positive and desired aspect. During the experiments, the framing proved to be effective (cf. Section "Programming Task Results", p. 164).

Development Environment Briefing In a fifteen-minute briefing, the study moderator gave all test persons an overview of the experiment's development environment to compensate for potential differences in their experience with the tools of the test environment. This included the Visual Studio Code IDE, the folder and file structure of the Go [78] framework following the Model View Controller [112] design pattern, used Go packages (app dependencies) and the Developer Tools of the Google Chrome browser. Participants were allowed to ask questions about the development environment. After the briefing, the study moderator asked them to read through the four tasks of the experiment. The participants also had the opportunity to ask questions about understanding the tasks. The developers were asked to think aloud and the meaning of the term was explained. Then the study moderator calibrated the eye-tracking, started the experiment, and left the room. The experiment was followed via video, audio, and screencast.

Programming Tasks The study comprised four programming tasks. The first two served as warm-up tasks so that the test persons could first get used to the study situation. Short tasks were selected that were assumed the participants to complete quickly, and that would not violate the framework's CSP default setting. In task one, the developers were asked to integrate a given icon as web page favicon. In

task two, the participants should use Cascading Style Sheets (CSS) to integrate an image as a background for the page.

In task three – the main task – the developers were asked to integrate a map into a web page using the Google Maps JavaScript API. They should implement the tutorial example of the official Google Maps API documentation. The task description provided an API key for the service and a link to the web page. However, the browser enforced the default CSP, did not execute the code example from the documentation and issued a warning in its JavaScript console instead. For the browser to display the map, the compilation of a CSP whitelist with the required Google Maps resources was a required secondary task – not mentioned in the task description – confronting the developer. Except in the "Control" group, the appropriate CSP was provided for the tutorial example in the documentation.

The last task was to set other coordinates for the map and a marker. To complete the task, the developer had to replace the CSP hash value for the changed lines of inline JavaScript code. This task should answer the question of whether the developers could build up enough understanding through the example to be able to adapt it to their requirements, including a securely customized CSP.

Documentation The original documentation page [80][5] (cf. Figure 6.6) starts with a headline, a short introduction, and prominently shows a Google map as a practical application example of the API. The developers could implement this example directly by copying and pasting the corresponding code from the "Try it yourself" section. For quick access, the short section "Getting started" with hyperlinks leads directly to the three following sections, which explain step-by-step details of the code example. "Step1" describes the structure of the HTML page, "Step2" the JavaScript code with API calls, and "Step3" the API key handling, which regulates access to the service and serves the service provider for billing. At the very bottom of the page, readers will find the section "Tips and troubleshooting" with short notes on known usage problems. This page of the Google Maps documentation does

[5] At the end of 2019, Google changed its website design. However, the content of the page has not changed at the time of writing this thesis. The original website used in the study is accessible via Internet Archive [79].

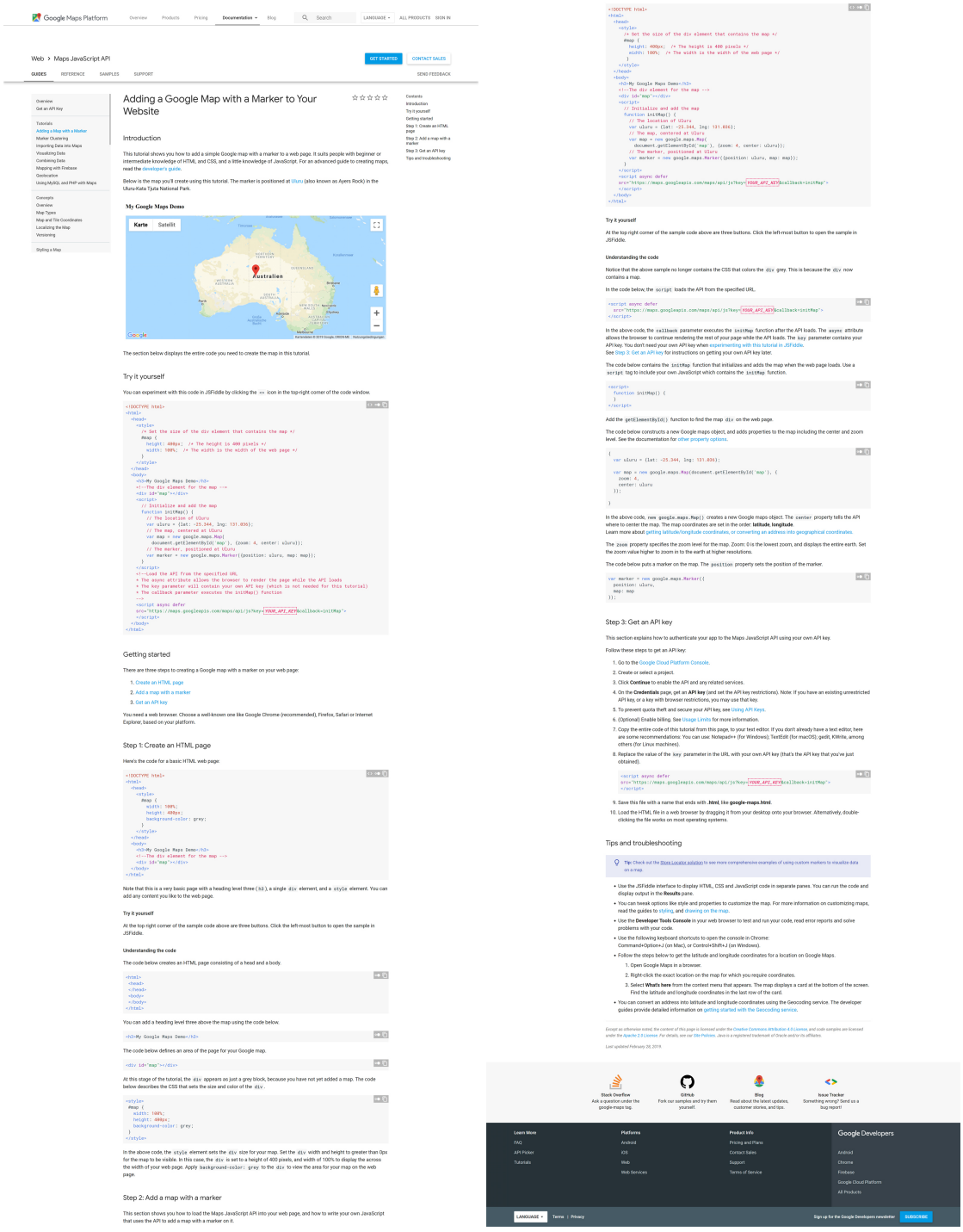


Figure 6.6.: The original Google Maps documentation page (first part left, second part right)

not contain any security advice other than instructions on how a developer can get and use the mandatory Google API key.

Figure 6.7.: Overview of the documentation and locations where CSP information is added in each study condition.

To support software developers in implementing secure programming practices of web engineering, in this laboratory study, the content of the original Google Maps documentation was explicitly changed, and the effects were investigated. The primary purpose of the documentation is to support users in implementing a

Google Maps API. Therefore, being realistic, it was excluded for the study design, that the page can present security-relevant aspects prominently in a section at the topmost positions. This decision raised the practical problem of how to integrate security-related information in a way that users would find it when they need it.

Three different approaches were designed (cf. Figure 6.7). The first approach, "Tips", integrates a subsection in the section "Tips and troubleshooting", at the very bottom of the web page, assuming that it could be difficult to find there. The CSP section (cf. Figure 6.8) explains how a developer can identify problems and then gives a recommendation for action. The formulations also use the keywords "directive," "hash," and "inline," which the Chrome browser developer tools use in their CSP warnings. Detailed step-by-step instructions could have been written in the documentation on how to define a CSP in the study web framework. However, since it is not practical for an API provider to do this for every available web framework, a general formulation was chosen. The section also provides a code example that the developer can copy and paste directly into his environment and an illustrative example that explains the individual components. The end of the section gives a link to a comprehensive CSP guide from Google [213] and to a tool that checks if a CSP is secure [209].

The second approach, "Step 4", takes up the given structure of the web page and adds a separate fourth section entitled "Set a Content Security Policy (CSP)", which precedes the chapter "Tips and troubleshooting". The content is identical to the previous approach. Also, a CSP link was added to a navigation sidebar and the "Getting started" section (cf. Figure 6.9).

Based on observations in the pilot studies, it was assumed that each respondent would be most likely to work with the Google Maps code example in the documentation. Hence, in the third approach of this study, "Code Comment", the effect of a source code comment was tested (cf. Figure 6.10) in the code examples of the API documentation. It was assumed that this approach could help developers who would potentially copy sample code without reading the documentation. This comment briefly points out a potential problem and provides a link to the CSP section. Identical to the first approach, the CSP documentation is placed at the end of the page (cf. Figure 6.7).

a) CSP section with guidance and CSP example

- If the map does not show up, one reason might be a CSP is set by your web server and enforced by your web browser. Check the **Developer Tools Console** in your web browser for error messages to see if elements are blocked.
- You **need all** of the following CSP directives to configure your web server for the Google Maps Demo code on this page and allow Google maps resources on your web site.
- You can open the *Developer Tools* and then reload your page. The *Console tab* will contain error messages with the **correct sha256 hash** for each of your refused inline elements. Alternatively you can use a hashing tool.

CSP Header String (raw):

```
default-src 'self'; script-src 'self' https://maps.googleapis.com/ 'sha256-purvb6auV8U0MiZWQ6SKq7Tjza
```

CSP Header String (pretty print including comments):

```
default-src     // Secure default
                'self';

script-src      // JavaScript from your host
                'self'
                // Maps JavaScript API
                https://maps.googleapis.com/
                // Inline script hash for the Demo code on this page
                // If you change the script in the Demo code
                // this hash needs to be updated
                'sha256-purvb6auV8U0MiZWQ6SKq7TjzaLqXH7qNlKpVUeXr6E=';

style-src       // Styles from your host
                'self'
                // Fonts specification
                https://fonts.googleapis.com/css
                // Inline style hash for the Demo code on this page
                // If you change the style in the Demo code
                // this hash needs to be updated
                'sha256-rXxm3Q5rCvjaI3w7x0f/JJDqECh8mxg68PKLP8Jo7pQ='
                // Styles
                'sha256-UvsJ5gtL0c/whxmyVt4YoNv7YnPUd0tANZO1q3NshXE='
                // Control active
                'sha256-VjKqXV9i0mo5RzxvaQpz7qQA91PkjLVqLQGYNI4Cc/I='
                // UI hover effects
                'sha256-g9aHNH7iF2hhGZYtVVd5mKQSnyLPmXWw5gwiuxBVonI='
                // Styles
                'sha256-2WQZQFa8KGAig8CPptpS8JDqetQ2jb5arMlI6fTGWiU='
                // Media print
                'sha256-/VVOq+Ws/EiUxf2CU6tsqsHdOWqBgHSgwBPqCTjYD3U='
                // Styles
                'sha256-a2VR/Wq1VPr0+3GRY+lEmAQm7wjwwnDtPpcCPs2zTrw='
                // Styles
                'sha256-mmA4m52ZWPKWAzDvKQbF7Qhx9VHCZ2pcEdC0f9Xn/Po='
                // Google Maps JavaScript API error
                'sha256-+1AnUMTxYqnxCnuWd5ie3PIFYaEJTJj6eAy7VkZ7jyc=';

img-src         // Images from your host
                'self'
                // UI images using data scheme
                data:
                // Satellit images
                https://maps.gstatic.com/mapfiles/
                https://maps.googleapis.com/maps/
                https://khms0.googleapis.com/kh
                https://khms1.googleapis.com/kh;

font-src        // Fonts
                https://fonts.gstatic.com/;

object-src      // Secure default
                'none';
```

- For an advanced guide to creating a CSP, read the Content Security Policy guide.
- Use the CSP Evaluator to check if a CSP serves as a strong mitigation against cross-site scripting attacks.

Figure 6.8.: Distinct CSP documentation extension a) CSP section with guidance and CSP example. Integrated into the original Google Maps JavaScript API documentation.

b) "Getting started" section with link to CSP section

Figure 6.9.: Distinct CSP documentation extension b) "Getting started" section with link to CSP section. Integrated into the original Google Maps JavaScript API documentation.

c) Source code comment in code examples

```
/**
 * Notes on Content Security Policies
 * If the map does not show up, check the Developer Tools Console in your web browser
 * for error messages to see if elements are blocked by a Content Security Policy
 * (e.g. default-src 'self';). Find tips and troubleshooting at:
 * https://developers.google.com/maps/documentation/javascript/adding-a-google-map#CSP
 */
```

Figure 6.10.: Distinct CSP documentation extension c) Source code comment in code examples. Integrated into the original Google Maps JavaScript API documentation.

Implementation and Study Environment The study took place in a usability laboratory. The developers worked with a Windows computer and two monitors. On the left monitor, they used the Visual Studio Code IDE for programming within a Go framework. Code completion and syntax highlighting and parsing were enabled for the code editor. Scaffolding was added for the tasks to keep complexity low. Like most web development frameworks, it implemented the model view controller pattern [112]. Security functionality was integrated with the middleware "Secure" [100] and the secure CSP default configuration `default-src 'self'` was kept. On the right monitor, the developers used the Chrome browser with Internet access and Chrome's integrated developer tools. The changes of the official Google Maps developers documentation web page were implemented with three custom

Chrome extensions, one for each condition with CSP documentation. The extensions monitored the requested URLs and replaced the original documentation with one of the CSP-extended variants. The participants accessed the documentation as they usually would. The software iMotions7 recorded eye-tracking data for the browser monitor with a Gazepoint GP3 HD sensor. The workplace was equipped with a microphone and two cameras to record the experiments and stream them live to the study moderator through a local network. The moderator annotated activities, statements, and key events during the sessions.

Structured Interview Following the experiment, the study moderator conducted a structured interview with each of the participants (cf. Appendix E). It was divided into three parts. Part one contained questions about the experiment. In the second part, previous experience in the field of software development was collected. And the third part served to gather demographic data.

Debriefing Following the exit interview, the study moderator held a twenty minutes debriefing session with each participant in order to explain the experienced programming problems. The dialogue clarified the purpose of the study and raised awareness for the topic of Usable Security. The moderator asked every participant not to share the contents of the experiment with others until the end of the study.

Pilot Studies The study design had been developed iteratively and the experiment was pilot tested twice. For both rounds, two members of the author's working group were asked, who were not involved in the study design. On the technical side, the pilot studies helped to adjust the hardware setup and to fix minor bugs in the web development framework. On the usability side, descriptions of the briefing and the task descriptions could be clarified, and explanations and elements of the CSP section were improved. Most significantly, the observations from the pilot studies led to the idea of integrating security-relevant information into the API documentation in different ways, which resulted in the three versions for the study.

Recruiting and Compensation The study was conducted at the end of a web development course. Data and application security was not part of the curriculum.

Participation in the study was an offer for a total of 103 participants in the course to reduce the workload of a semester project. In return, several required features were waived. It was not a prerequisite for passing the course. The study was announced in the lecture as a study on web development with the Go programming language. During lectures and practical exercises, the students were regularly reminded about the study. Interested students could register online for an appointment. They chose one of the offered appointments themselves and were assigned round-robin to the four groups. During the study, water and candy bars were offered.

Performance Measures Several metrics were used for the evaluation of the experiment. With eye-tracking, the total time a developer spent on the API documentation page was measured. The gaze data also allowed assessing parts of a documentation web page to which participants were paying attention. The video recordings were used to measure the time that a test person worked on each programming task. The test moderator documented key moments directly while watching a live stream of the experiments. After the study, the session recordings were used to check and precise each time stamp before using them for statistical analysis. The structured interview answers were evaluated to gain qualitative insights. The developers assessed the security of their program code, and the participants were asked to describe the purpose of a CSP for assessing the conveyed understanding.

Limitations The results of the study are subject to several limitations. First, students were recruited who had taken a course in web development. This population is not representative of all web developers. However, previous studies could not find significant differences between student samples and professional developers (cf. Section 2.2, p. 10).

It was also observed that the behavior of the participants solving programming problems varied greatly (cf. Section 6.2.4). Some of the test persons had already gained experience in web development before the course. The sample nevertheless had a narrow range of programming experience. A significant correlation between secure task solutions and programming experience was not found.

The laboratory study design used a concrete software development scenario (cf. Section “Programming Tasks”, p. 147). The results are, therefore, only condition-

ally transferable to other software development contexts. API documentation, in general, also has many different forms. Therefore, this study uses the real and well-maintained documentation web page of the popular Google Maps JavaScript API as a baseline. Nevertheless, the structure and content are not representative of all API documentation.

The experiments were conducted with a Go software development environment, which has some technical limitations. This study, refers to CSP level 2 as browsers offer only incomplete support for level 3 features [125, 82, 54, 12]. Therefore, CSP level 3 security features [222, 165] were not considered for the CSP example in the prototypes. However, the primary approach of providing a developer with API tailored CSPs does not change by considering additional CSP features. Additionally, a local webserver was used as an embedded part of the framework. Thus another potential source of error was excluded. Under real circumstances, stand-alone web servers may also set HTTP headers. How to support developers in even more complex scenarios is left for future work.

Finally, participants could not find the added contents of the prototypes by a search engine. However, it is assumed that depending on search queries, a search engine could have improved the discoverability of the CSP documentation.

6.2.4. Results

The analysis answers the research questions formulated in Section 6.2.1 (p. 142). First, the study sample is characterized based on the demographics and software development experience. Then the results of the experiments are presented.

Participants

Forty-nine participants took part in the study over four weeks in summer 2019, 41 males, and 8 females. They were, on mean average, 25 years old (SD: 4). Nineteen studied the program in Media Technology, and 28 studied the program in Computer Science and Engineering. The students mainly reported studying in the 6th semester (SD: 2). The standard period of study is seven semesters. Thirty-eight recruited participants, with more than five study semesters, were nearing the end of their Bachelor's degree. Thus, the participants were typical candidates who

start their professional life as junior software developers after their Bachelor's thesis. The fact that 17 of the students (35%) had already been paid for programming before the study supports this assumption.

The developers reported having 4 years of programming experience on average (SD: 2). All of them started learning the Go programming language in the web development course about half a year before the study and implemented at least one Go web application. However, 23 have also developed other web applications before. Thus the average experience with the JavaScript programming language was 2 years (SD: 2). The groups were mainly balanced in terms of demographic data and the previous development experience (cf. Table 6.4).

The qualitative answers show that the developers of the study sample had little experience in applying data and application security measures. None of the participants named CSPs when asked what security measures they had ever considered in the development of software. Directly asked only 3 stated that they had previous experience with CSPs. All three could remember that they had problems with the implementation like developer G4P5 explained: *"I briefly touched on it, but I mostly let it go because it was too complicated, and it didn't want to work."*[6] Eleven participants stated that they never considered any security measure.

Areas of Attention

First, it is analyzed where the participants were looking. The eye-tracking sensors measure 150 individual gaze points per second (150Hz). If gaze points are close together in time and space, this is an indication for visual attention. The eye-tracking software calculates these gaze clusters and counts them as "fixations" for a defined area of interest. The heatmaps in Figure 6.11 visualize the general distribution of attention on the documentation based on fixations. The green, yellow, and red colors transparently covering the four test items represent the attention intensity from low to high in ascending order. The circled numbers on the right side indicate the most noticed areas, in chronological order of first attention. Colored bars on the left side mark added CSP documentation elements. The heatmaps are used to answer **RQ6a** by comparing the areas of attention of the experimental groups.

[6]All quotes have been translated into English.

Demographics	Control	Tips	Step 4	Code Comment
Age (mean/SD)	24.0/4.6	26.2/4.2	24.8/3.2	25.2/3.8
Male	8	11	12	10
Female	2	2	1	3
Study Semester (mean/SD)	6.2/1.5	6.3/2.1	5.7/1.8	5.9/1.3
BaMT[1]	5	3	6	5
BaCSE[2]	5	9	7	7
MaCSE[3]	0	1	0	1
Previous Experience	**Control**	**Tips**	**Step 4**	**Code Comment**
Years of Dev. Exp. (mean/SD)	4.8/1.9	3.1/1.2	4.9/3.2	3.4/2.0
Years of Go Exp. (mean/SD)	0.6/0.2	0.5/0.1	0.6/0.2	0.6/0.2
Years of JavaScript Exp. (mean/SD)	2.0/2.1	1.2/1.0	1.5/1.7	1.4/1.8
Paid for programming	4/10	5/13	4/13	4/13
Go Web Applications	10/10	13/13	13/13	13/13
Other Web Applications	5/10	7/13	6/13	5/13
Content Security Policy	0/10	0/13	2/13	1/13
Mobile Development	4/10	9/13	9/13	6/13
Desktop Development	5/10	7/13	8/13	8/13
Embedded Development	4/10	5/13	5/13	4/13
Enterprise Development	2/10	1/13	1/13	1/13
Other Development	2/10	1/13	2/13	1/13
Windows	10/10	13/13	13/13	13/13
Visual Studio Code	7/10	12/13	10/13	11/13
Browser Dev. Tools	8/10	13/13	13/13	13/13

[1]Bachelor Media Technology, [2]Bachelor Computer Science and Engineering, [3]Master Computer Science and Engineering

Table 6.4.: Distributions of reported demographic data and development experience in the four study groups.

Figure 6.12 gives a detailed quantitative overview of fixation distribution for each section and condition.

The task asked the developers to call up the Google Maps tutorial. Thus, they started reading and exploring the page at the top. However, they did not strictly approach the web page in chronological order. All groups skipped areas due to their selective search behavior for the task solution by scrolling and repeatedly visiting the page.

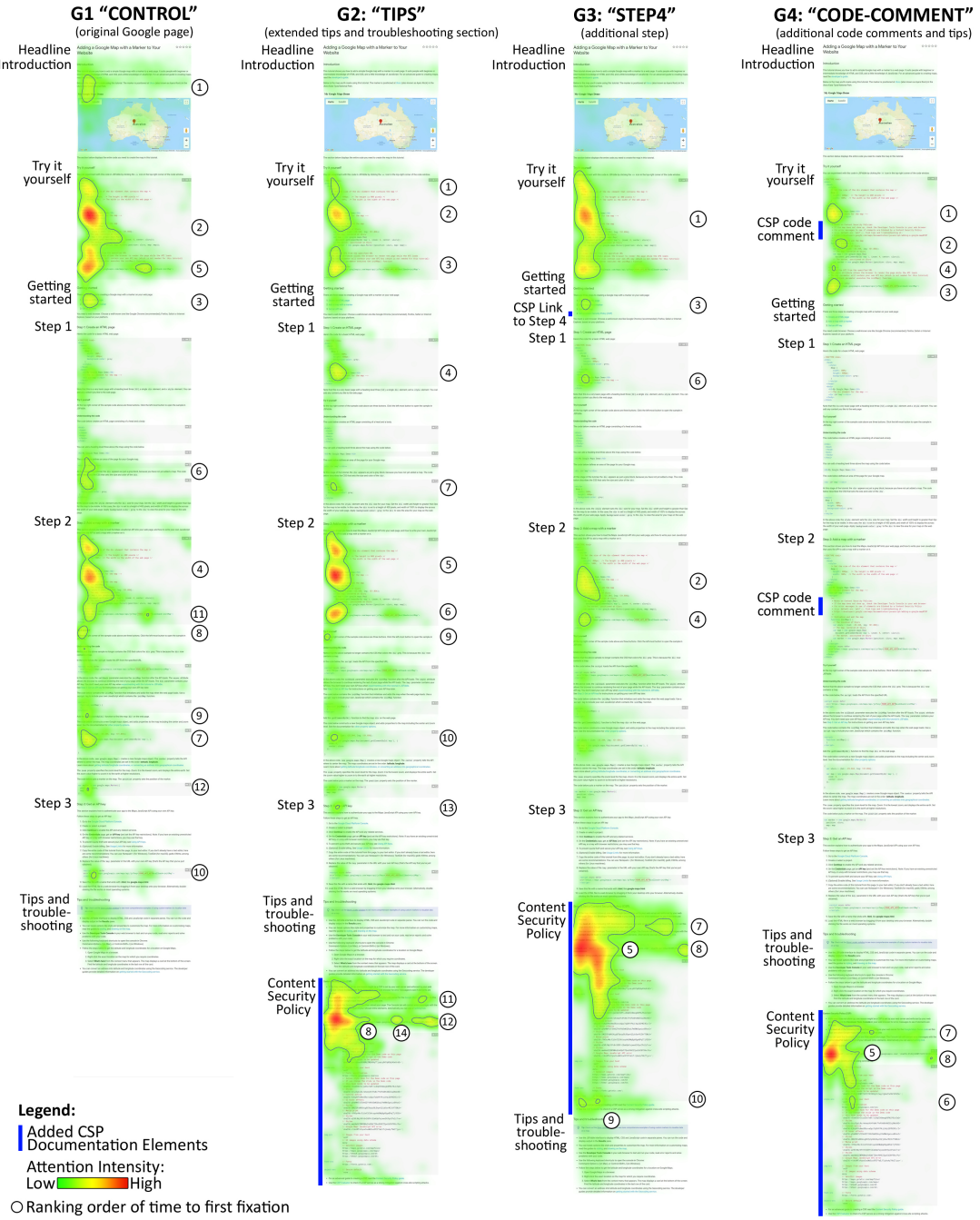


Figure 6.11.: Heatmap showing the attention intensity on the Google Maps API documentation aggregated for all participants in each of the four study conditions.

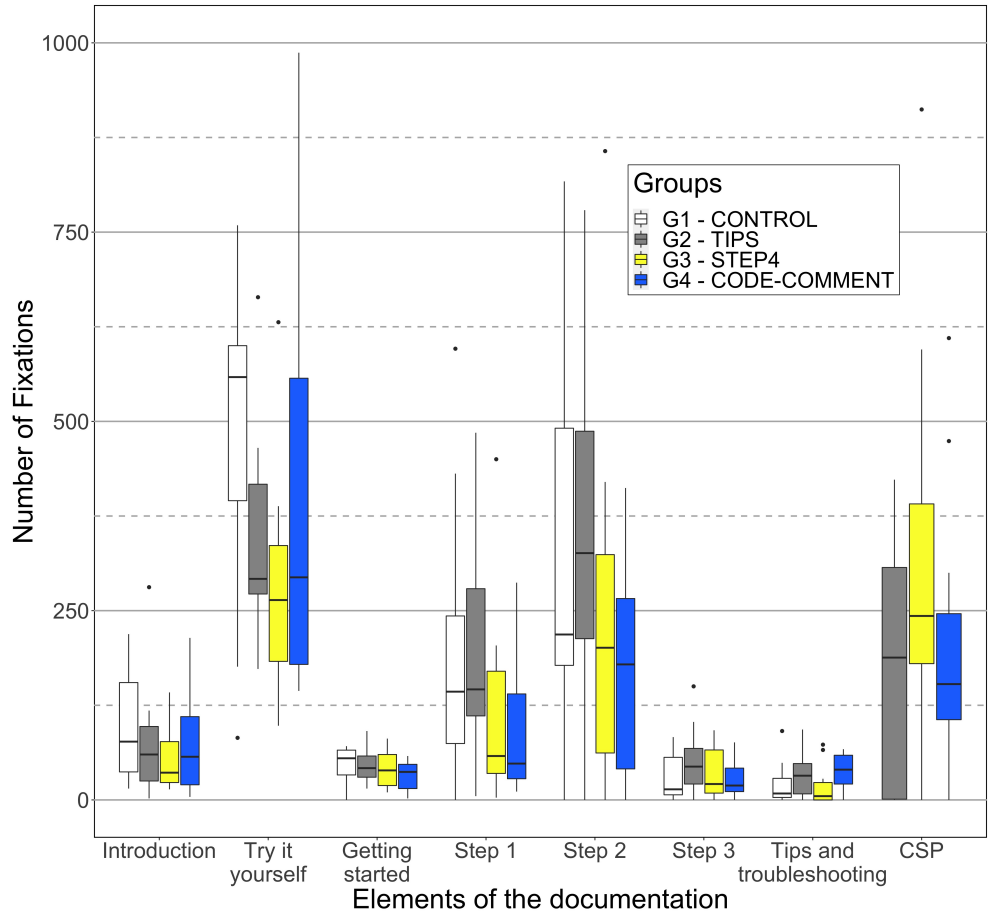


Figure 6.12.: Box plot diagram of fixation distribution by condition for each documentation part and study group.

The participants paid the most attention to the code in section "Try it yourself," "Step 2," and "CSP." All red areas in the heatmap, indicating the most attention, are related to code examples. Participants directly found the Google Maps example and started adapting it to their development environment. This behavior corresponds to their primary task of implementing the map and the more complex secondary task of implementing a CSP.

Thus, in the group "Step 4" and "Code Comment," attention has shifted to the CSP areas. The middle page areas show clearly fewer points of primary attention. The "Code Comment" group even concentrates attention on only eight main areas. The code comment itself is not part of these. It needs only short attention, as it

merely refers to the CSP section. Group "Control" and "Tips" searched more for information solving their problem in original page elements. Thus they repeatedly focused on the Google Maps code examples in the sections "Try it yourself" and "Step 2." Group "Tips" is in chapter "Step 2" more similar to the "Control" group.

The green areas show that developers have worked with all elements of the site but with limited attention. This impression is correct for the upper elements up to and including "Step 1" but limited for the areas below. A total of nine participants in the three groups with CSP elements have not seen or found the CSP section (cf. Table 6.5).

Documentation Part	Control	Tips	Step 4	Code Comment
Intruduction	10/10	13/13	13/13	13/13
Try it yourself	10/10	13/13	13/13	13/13
Getting started	9/10	13/13	13/13	13/13
Step 1	9/10	13/13	13/13	13/13
Step 2	9/10	12/13	12/13	12/13
Step 3	9/10	12/13	11/13	12/13
Tips & Troubleshooting	8/10	12/13	8/13	12/13
Content Security Policy	-	9/13	10/13	11/13

Table 6.5.: Participant ratio per documentation part and study group

Based on these results, **RQ6a** is answered. It is concluded that the developers of the study were focused on code examples, mostly skimming the documentation while searching for a quick solution to their programming task. As one developer aptly described his behavior in the interview: *"There's a page where they explain all this [CSP]. But honestly, I didn't read it carefully. I just looked for the solution!"* (G4P6)

Information Placement Effects

Information placement effects are further analyzed to answer **RQ6b**. During the programming tasks, participants spent an average of 7 minutes on the site (SD: 3). There was no significant difference (Kruskal-Wallis $H(3) = .727$, $p = .867$) between the groups (cf. Figure 6.13).

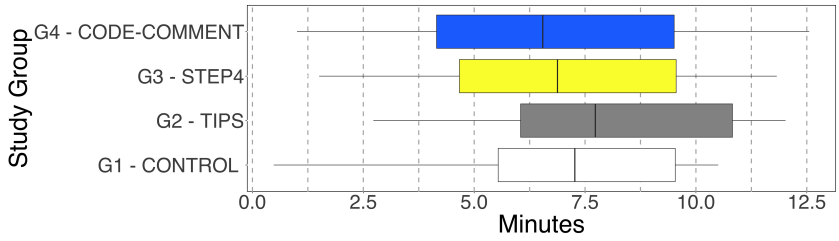


Figure 6.13.: Total time each group spent on the documentation site, based on gaze data

The box plot diagram in Figure 6.14 shows the participants' fixation distribution on the documentation page by condition (1) for all elements, (2) only for original elements, and (3) only for the CSP elements. Consistent to the time spent, there are no significant differences in the attention the groups paid to the entire documentation page. (Kruskal-Wallis H(3) = 2.538, p = .468). Thus, on average, the groups paid equal attention to the Google Maps documentation. We neither find a significant difference between the three groups in the distribution of fixations across the CSP documentation elements (Kruskal-Wallis H(2) = 2.040, p = .361). Thus, the groups paid similar attention to the CSP documentation elements. However, a significant difference could be found (Kruskal-Wallis H(3) = 9.243, p = .026) in the distributions of fixations on the elements of the original page. From this result, it is deduced that the characteristics of the condition led to an attention shift to CSP documentation elements. Dunn Bonferroni tests show that group "Control" and "Step 4" (z = 2.553, p = .011, r=.532), as well as "Control" and "Code Comment" (z = 2.451, p = .014, r=.511), differ significantly, both with an effect size over 0.5. It is concluded that the designs of the group "Step 4" and "Code Comment" both are suitable to transport the security-relevant information to the developer.

While the statistical results of attention and usage time do not show significant differences between the three CSP groups, there is a significant difference in the time it took the developers, after starting task three, to find the CSP documentation (Kruskal-Wallis H(2) = 9.048, p = .011) (cf. Figure 6.15). Dunn Bonferroni posthoc tests show that group "Tips" and "Code Comment" significantly differ (z = 3.0, p = .003, r=.671), with an effect size over 0.6. On average, participants in the "Tips" group needed 13 minutes longer to find the security-relevant CSP information

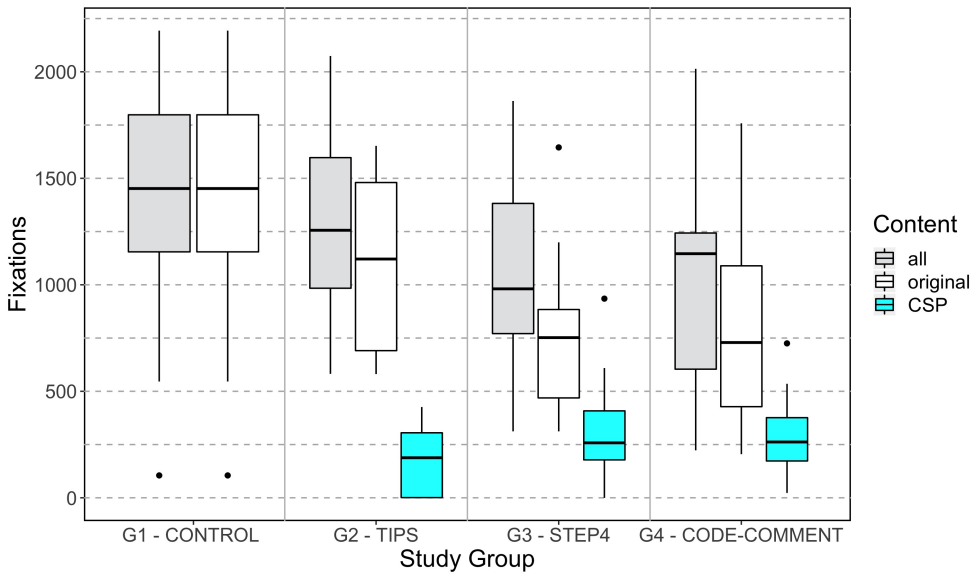


Figure 6.14.: Box plot diagram of fixation distribution by condition for the entire page, original elements, and CSP element.

in the documentation than in the "Code Comment" group. However, the time frame of the study was sufficient to compensate for this difference. Therefore a significant difference between groups was not found in the total usage time of the documentation.

Ten out of 13 participants of the group "Code Comment" copied the CSP code comment from the documentation and inserted it into their IDE. Three deleted it, two immediately after pasting, another one after a few minutes. *"I'll delete all the unnecessary comments; it's just annoying"* (G4P1). Altogether the code comment was found in seven final results. However, 9 of 13 participants became aware of the documentation by the reference in the code comment: *"This is beautiful here, just as you wish. You couldn't have a simpler bug-fix"* (G4P7). Of the four others, two had not found the documentation at all, compared to four in the "Tips" group (cf. Table 6.5), and two had found it by scrolling down the site. Only three developers in the "Step 4" group found the CSP elements by the "Getting started" link. In total, 10 participants directly took up the note on the browser developer tools from the CSP documentation. Thus, they directly found the CSP warning messages in the browser console and could identify their problem.

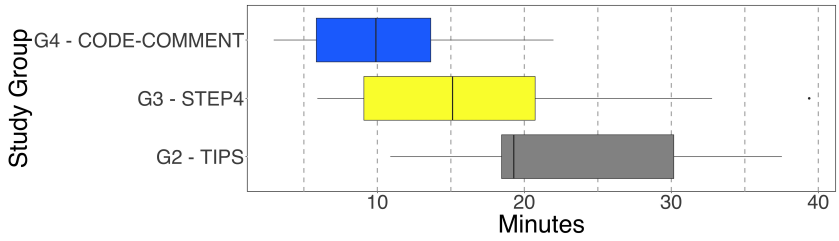


Figure 6.15.: Time to CSP Documentation after starting task 3

Thus, it is concluded for **RQ6b**, that the placement of security-related code examples does matter and that it has an impact on the time it takes to find the required information. Remarkably, the discoverability of security-related information and code examples in API documentation proves to be a problem if it locates at a distance to the API code examples.

Programming Task Results

The following section answers **RQ6c** by analyzing how much time the developers spent on each task within the overall programming time frame of one hour and by evaluating to what extent the tested approach could support the developers to achieve functional and secure results. Task one was rated as functional if the favicon was displayed, task two when the background image was visible, and in tasks three and four if the Google Map appeared on the web page. The rating of secure task solutions was limited to the CSP configurations. The security of final CSP configurations applied for task three and four were evaluated by criteria proposed and applied by Weichselbaum et al. [209]. The sample solution from the CSP documentation (cf. Figure 6.8, p. 152) was also based on these characteristics and was therefore considered secure.

The programming tasks evaluation (cf. Table 6.6) shows that most of the test persons handled the warm-up tasks well. All developers solved task one in an average of 6 minutes (SD: 4), and only one developer each in the “Tips” and “Code Comment” groups were unable to implement a functional solution for task two. Some participants in these two groups spend a comparatively long time on the second task because they had difficulties with CSS debugging. On average, developers worked for 11 minutes (SD: 9) on task two.

Condition	Control	Tips	Step 4	Code Comment	Total
number of participants	10	13	13	13	49
task 1 functional	10	13	13	13	49
task 2 functional	10	12	13	12	47
task 3 functional	6	2	8	4	20
task 3 functional and secure	**0**	**2**	**5**	**3**	**10**
task 4 functional	6	2	3	3	14
task 4 functional and secure	**0**	**2**	**3**	**2**	**7**
Programming Event	Control	Tips	Step 4	Code Comment	Total
found CSP documentation	-	9	10	11	30
copy & paste header string	-	3	5	7	15
copy & paste pretty print	-	1	1	0	2
copy & paste header & pretty print	-	3	3	2	8
configuration file entry	6	6	6	5	23
HTML meta tag	2	3	4	4	13
meta tag & config file entry	0	2	3	1	6
secure CSP meta tag	0	2	3	2	7
secure CSP configuration file	0	3	3	3	9
secure CSP meta tag & secure CSP configuration file	0	2	0	0	2
deactivated CSP	4	1	3	1	9

Table 6.6.: Programming results for each condition and in total.

Participants spent an average of 38 minutes (SD: 12) on the main task three (cf. Figure 6.16). A total of 17 people started task four and worked on it for 8 minutes (SD: 5) until the total study time of one hour was reached, or they pressed a bell to signal they finished the tasks.

In the first attempt, the participants copied the Google Maps sample code (all or only parts) from the documentation and pasted it into their development environment. However, the numbers of working and secure solutions of tasks three and four reflect that the CSP default mechanism has been a problem for the developers: *"I am confused, very confused, I just have to copy, and it has to appear there now. Why am I blocked"* (G2P4)? In all groups, 20 and thus less than half of the test persons were able to implement a functional solution for the third task.

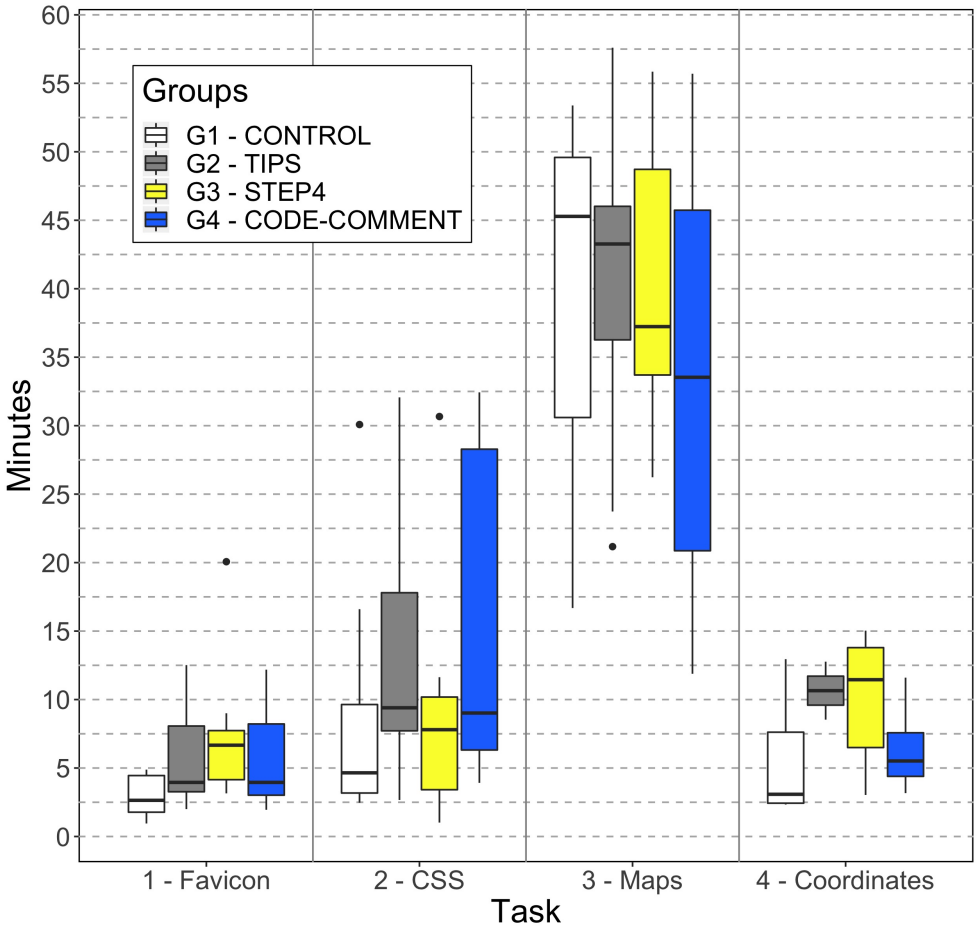


Figure 6.16.: Box plot diagram of time spend per task and group.

Nine of them have disabled the CSP mechanism in their framework. Participant G1P9 commented: *"I can't get it right! Why? Okay, let's do it another way. Let's just say screw security for now. I'm gonna put a* `'none'` *here, and then we'll move on – just for the record."* Changing the coordinates in the final task without a CSP was then easy for these developers. Nevertheless, the number of deactivations were lowered by the framing in the preliminary discussion: *"Since it was just said that the site should be secure, I'll do that"* (G3P5)! Overall, the "Control" group achieved more functional but insecure solutions in task 4 than the other groups.

Fifteen developers copied and pasted only the CSP header string from the documentation, two only used parts of the pretty print, and eight copy and pasted

		Secure Solution			
		Task 3		Task 4	
		true	false	true	false
Used CSP Documentation	true	10	20	7	23
	false	0	19	0	19

Table 6.7.: Contingency table for secure solutions and used CSP documentation in task three and four.

both during their attempts to integrate the CSP into the framework. A total of ten developers who did not disable the CSP mechanism were able to integrate a secure and working solution into the web page. However, they subsequently faced further problems of understanding when dealing with the customization task: *"But why is it a security problem if all I do is change the coordinates"* (G3P7)? Seven of them had understood the CSP concept to such an extent that they were able to achieve a functional and secure solution.

All four groups have worked with the documentation for about the same amount of time (cf. Section "Information Placement Effects", p. 161), and no major differences were seen in the numbers of functional and secure solutions between the groups with CSP documentation. Therefore the effect of CSP support on the security of task solutions was compared across all groups (cf. Table 6.7). Fisher's exact test is significant ($p<0.01$) for task three and also significant ($p<0.05$) for task four. The developers did not need more time to work with the documentation, which means that additional security-related information was not an extra burden. The opposite is the case; it helped developers to achieve better results. Thus, the CSP documentation has significantly improved the security of the solutions. 33% of the participants who worked with the CSP documentation were able to implement a CSP securely in task three, and 30% in task four. None of the developers who did not have the support of the CSP documentation approach achieved a secure solution. However, these results also clearly show that the majority of the test persons would have needed more support. The approach could only help to bring security-relevant information to the developer and thus provided a sample CSP. From the moment of integration into the framework, the experiment has shown that there are still other factors where support is lacking. These are summarized in the following.

The integration into the framework, especially where exactly the CSP code example had to be inserted, turned out to be a problem: *"I understand that I have to copy this string somewhere, I just don't know where"* (G2P12). Searching for a solution on the Internet, participants found mainly meta-tag CSP examples, although the use of HTTP headers dominates in practice [40]. However, meta-tag CSP examples are suitable as self-contained code examples that can be easily integrated into an HTML file and tested in a browser. Thus, a Google search has led participants to the website "SELFHTML" [174], where they have found a meta tag code example. If a respondent had defined a secure CSP using a meta tag, this was the case for 9 developers, both policies, the CSP of the configuration file and the CSP of the meta tag, apply. The test persons did not understand this standardized [222] behavior of the development environment: *"And how do I get it [the browser] to just take that? [...] It ignores the line completely, is that possible"* (G3P8)? It was difficult for the developers to trace the origin of the policy (e.g., server-side or client-side). A hint that a CSP was set twice by server and client-side was missing to resolve this double policy confusion. Answers from the Stack Overflow community were also not considered helpful: *"Ok, that's how it works, but I don't think that's how the security policy was meant to be. But I found it very hard to figure out how to configure something via Stack Overflow"* (G1P5). The results also confirm usability problems with CSP warning messages in the Chrome browser console, which have already been documented and described in the previous Study 5.

Concluding for **RQ6c**, the CSP documentation has significantly improved the security of the solutions and helped participants with CSP configuration until the moment of integration into the framework. Thus, the approach could not compensate for all problems that prevented participants from implementing a working and secure solution. Service providers can help software developers defining CSPs by providing additional information in their documentation. Other aspects lie outside their direct sphere of influence.

CSP Comprehension

This last analysis section serves to answer **RQ6d**. Figure 6.17 shows the participants' perceptions regarding the security of their task solutions. Noticeable is the

predominant “disagreement” and “strong disagreement,” except in the “Step 4” group, which had the highest number of secure solutions for task three and four. In comparison, there is less “agreement” on the other side, and only a few “strong agreements.”

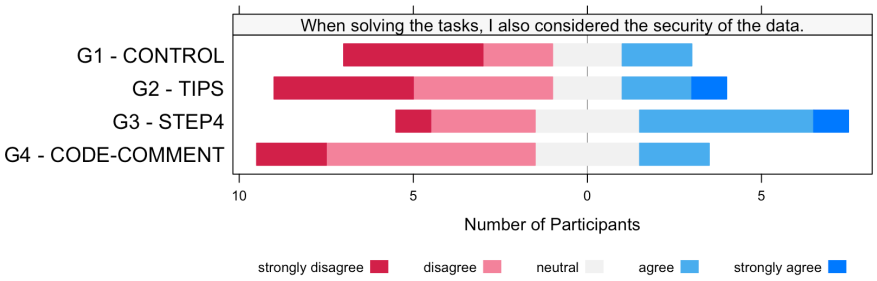


Figure 6.17.: The participants’ subjective assessment of whether they have considered the security of the application data.

The participants who worked with the documentation provided different reasons for their assessments. Seven reported to agree or strongly agree because they used the CSP. Three named other technical reasons. Nine participants explained that they disagree or strongly disagree because they had only focused on solving the task, not thinking about security. Two participants had not agreed because they bypassed the CSP. One stated he did not exactly know what he was doing when configuring the CSP. Another two gave other technical reasons for their negative assessment. Thus, ten developers made a conscious assessment of security based on their CSP handling, which was guided by the documentation. However, most developers were able to develop a basic understanding of the CSP mechanism’s sense and purpose during the experiment. In the interview, 21 gave an explanation that was correct in terms of content. Five stated that they could not develop any understanding, and three answers were incorrect.

Summarizing the answer to **RQ6d**, the tested CSP documentation focusing on examples has a limited ability to provide understanding and confidence in the security mechanism.

6.2.5. Discussion

This study contributes a first in-depth understanding of security-relevant code examples in the API documentation, examining a novel approach to help developers adopting API functionality securely. The results answer the research question **RQ6**. (1) Developers are focused on elements with code examples, mostly skimming the documentation while searching for a quick solution to their programming task. (2) The placement of security-related code examples does matter and significantly impacts the time it takes to find security-relevant information. For this reason, security-relevant code examples and information should not only be integrated as a peripheral feature but should be placed as close as possible to the code examples of the APIs. Therefore, code comments have proven to be very helpful in pointing developers to security-relevant information that locates elsewhere in the documentation. (3) The results also show that this approach significantly supports developers to produce secure CSP implementations with security APIs. Web API producers should consider these three findings to usefully integrate security-relevant information into their web API documentation. However, (4) developers have additional information needs that the tested approach can not meet. (5) Furthermore, it was found that CSP examples have a limited ability to provide understanding and confidence in the security mechanism.

The results of the study highlight that API providers have an exclusive responsibility because they can provide high quality and trustworthy first-hand information. Thus, they can give software developers secure CSPs for their service at hand. Instead of imposing a reverse engineering approach on the API users, who have to build a CSP for web resources following the try and error principle, API providers could provide pro-actively and transparently service tailored CSPs.

It is recommended to take up the approach of writing CSP documentation and CSP examples into API documentation. The security-relevant work step should be clearly stated in a documentation's content structure (group "Step 4"), or the developer should be most effectively made aware of necessary secondary tasks via a code comment (group "Code Comment"). If API providers specify a CSP for their services, also a positive training effect is assumed. Developers, whether young or experienced, would repeatedly deal with CSPs, which on the one hand,

can increase the attention of developers for CSPs over a short time, and at the same time, convey very vividly how secure CSP policies need to be structured and implemented. Search engines would also be able to display more suitable solutions to implementation problems. Secure examples could reduce the numerous existing tutorials on circumventing the security mechanism. More documentation would allow best-practices to mature and could ultimately gain secure CSP usage.

Future work should then support software developers in emerging issues like frequent merging CSPs of different providers. As web services underly frequent changes, API users need to adjust their policies several times a month [165]. Thus, also during this study, the example CSP had to be adjusted once. API vendors could provide API users with up-to-date CSP examples as a valid reference for action. Future work should also examine suitable semi-automated routines to merge multiple CSP examples. Here the potential to extend the CSP standard is assumed. In the long run, prominent placement in the documentation may no longer be necessary, as software development then usually would expect service providers to indicate the policies in a suitable place.

Future work should further investigate how documentation can better support developers in achieving a more profound understanding (cf. Section “CSP Comprehension”, p. 168) during the implementation of CSPs and confidence in their result, after implementation. In addition to documentation, it is no less important to examine and improve other tools involved (cf. Section “Programming Task Results”, p. 164), like CSP warning messages in browser consoles and web development frameworks. Also, security defaults are difficult for developers to understand. How the security concepts of a web development framework look like and how they work is not easy to overview. It is challenging to inform developers quickly and transparently about reliable security support in the background and about mechanisms that developers need to adapt. These should still work securely for their requirements on the one hand, and should not get in the way on the other hand. Moreover, it must be apparent which security services developers have to take care of themselves.

7. Conclusions and Future Work

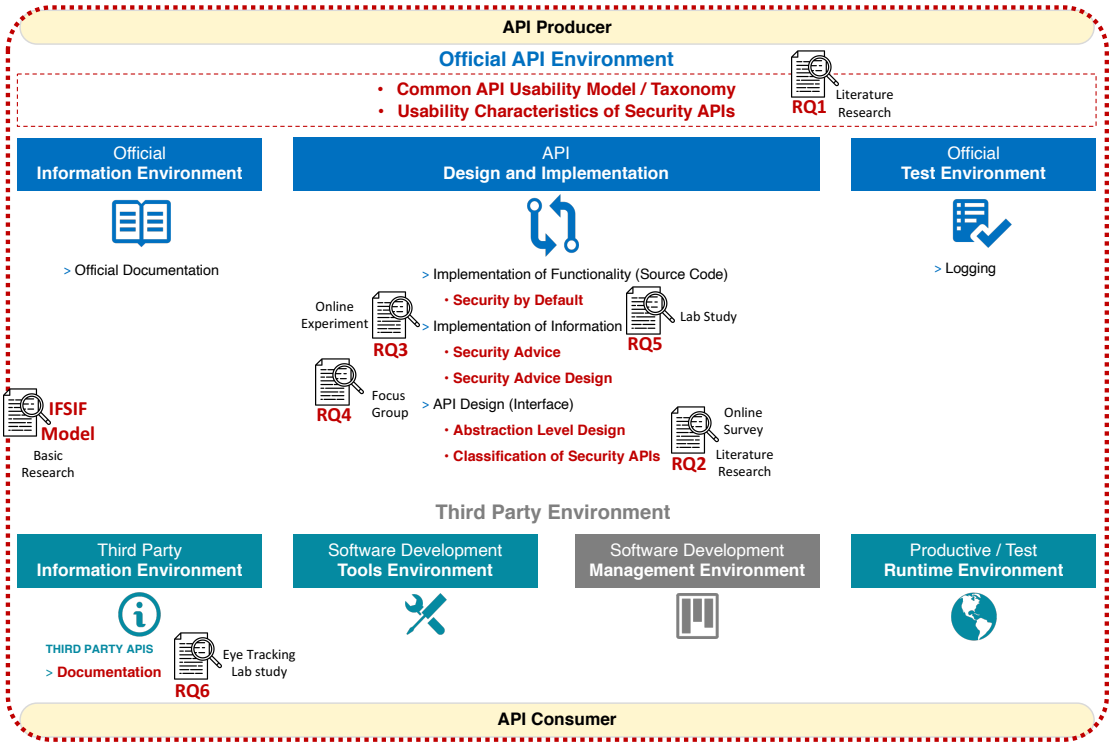


Figure 7.1.: IFSIF model classifying the contributions of this thesis.

7.1. Conclusions

This thesis has made several contributions to extend the current state of knowledge about the usability of security APIs. A total of six studies, using various methods, have contributed to the results of this thesis (cf. Figure 7.1) to answer the key question formulated in the introduction (cf. Section 1.4, p. 5):

To what extent do information flows support software developers in using security APIs and implementing secure software?

This thesis has contributed fundamental results that can be used in future work to identify and improve important information flows in software development. The studies have clearly shown that developer-tailored information flows with adapted security-relevant content have a positive influence on the correct implementation of security. However, the results have also led to the conclusion that API producers need to pay special attention to the channels through which they direct information flows to API users and how the information is designed to be useful for them. In many cases, it is not enough to provide security-relevant information via the documentation only. Here, proactive methods like the API security advice proposed by this thesis achieve significantly better results in terms of findability and actionable support. To further increase the effectiveness of the API security advice, this thesis developed a cryptographic API warning design for the terminal by adopting a participatory design approach with experienced software developers. However, it also became clear that a single information flow can only support up to a certain extent. As observed from two studies conducted in complex API environments in web development, multiple complementary information flows have to meet the extensive information needs of developers to be able to develop secure software. Some evaluated new approaches provided promising insights towards more API consumer-focused documentation designs as a complement to API warnings.

In the course of this thesis, the IFSIF model was developed, a model of influencing factors on security-relevant information flows in software development. This model can be utilized to illustrate, evaluate, and design information flows between an API producer and an API user. It has proved very useful in the context of this work and illustrates the contributions made to answer the key question at the beginning of each study section of this thesis.

7.1.1. Security API Design

Since the present thesis is one of the first to make a scientific contribution to the field of Security API Usability, Chapter 4 examined basic questions regarding the design

of security APIs. A literature review answered the question **RQ1:** *Are approaches from the API usability research sufficient for the specific context of security APIs?*

A developed API usability model has shown that available research results from the general field of API usability build a fundamental basis for the usability of security APIs. Thus, API producers need to consider common usability aspects when designing APIs for security mechanisms. However, the presented API usability model draws a fragmented picture of API usability, which cannot yet provide a comprehensive and reliable baseline. Furthermore, a review of research conducting security analyses unveiled the security API usability has specific characteristics not covered by the API usability research. In total, eleven individual characteristics were identified and described: (1) end-user protection, (2) case distinction management, (3) adherence to information security principles, (4) testability, (5) constrainability, (6) information obligation, (7) degree of reliability, (8) security prerequisites, (9) execution platforms, (10) delegation, and (11) implementation error susceptibility. At several points in this thesis, these have proven to be suitable for specifying particular characteristics of security APIs. Security API producers can consider these proposed conceptual usability characteristics for their security API design or evaluation.

The second study combined a second literature review with an online survey and examined the **RQ2:** *What level of abstraction do software developers prefer when working with security APIs?*

Although most of the participants were not security experts, the developers expressed a definite requirement for a security API to provide both high-level and low-level interfaces. Developers desire to flexibly use an API according to requirements of their development projects. This demand generally sounds like a desirable characteristic for a software development tool. In consequence, neither security controls APIs nor primitive APIs should provide their users only with high-level interface designs that are fail-safe to use, but constrain application and, thus, likely also usability. Much more balanced designs of different levels of abstraction that support the API user in implementing various use cases are necessary. Furthermore, the actual situation is critical, with missing security controls. Consequently, regular developers have to continue coping with security primitives APIs to meet their programming tasks.

Two findings suggest that further research needs to address the usability of error-prone security primitives APIs. First, the current limited availability of controls APIs forces developers who are no security experts to cope with error-prone security primitives APIs. Second, security primitives APIs are also the building blocks for future security controls APIs. Usability issues of security primitives APIs must not compromise the reliability of security controls APIs. For this reason, the concept of security API warnings has been examined.

7.1.2. Security API Warnings

The findings from the security API design studies have motivated research that, for the first time, examined the approach of a security warning in the developer console that comes directly from the API. Results could answer (**RQ3**) *whether API-integrated security advice has a significant effect on code security and perceived API usability.*

The online controlled experiment evaluated a prototype implementing an initial design proposal for API-integrated security advice. The results gave evidence for a positive effect on code security. The majority of the participants who received security advice turned insecure code into a secure code. Also, the adherence rate for security advice (73%) was promising. However, API security warnings did not affect perceived usability. Thus, the presented design concept for API-integrated security feedback allows API providers to improve code security for existing cryptographic APIs on the implementation level, without having to change existing API interfaces.

Pursuing and questioning these initial conclusions, a focus group study developed the first participatory design approach for cryptographic API warnings in the developer console in order to find out (**RQ4**) *what kind of design do software developers find helpful for cryptography warning messages in the console.*

It was found that developers generally find security feedback essential and useful. For console messages, the participants suggested five core elements. These are (1) message classification, (2) title message, (3) code location, (4) link to detailed external resources, and (5) coloring. The results show a requirement for conciseness for cryptographic APIs in the developer console and log file environment. At first glance, the five core elements can also be applied generically to the context of

other API warnings. It is suspected that these are common design characteristics for API warnings in general. However, the participants have expressed higher information requirements for an API warning in environments such as the IDE or CI reports as compared to the console. Developers partially accept design proposals of existing end-user guidelines if more information is desired, which indicates an overlap between populations. Some context-specific aspects for cryptography were considered helpful by some of the developers like a severity level, a risk assessment, a recommendation of alternative cryptographic algorithms, or a label to assess a cryptography warning's quality. At this point, uniform design for all types of API warnings is not sufficient.

7.1.3. Additional Information Flows for Improving Security APIs

Exploring the extent to which information flows can be supportive, two laboratory studies examined additional information flows to support software developers in using security APIs and implementing secure software. The first examined (**RQ5**) *how the enforcement of security by default affects the usability of a web development framework.*

In the concrete application case of Content Security Policies, it is concluded that only following the "security by default" principle is not enough in the context of software development. If it is not integrated with usability in mind, security measures are likely to get in the developers' way. The current security-relevant information flows influenced by designs of CSP violation messages, documentation, and IDE support are individually and in combination, neither effective nor efficient, nor do they lead to a satisfying result for users. The results of this study showed that developers who are inexperienced with CSP would currently not be able to intuitively or quickly implement a secure CSP. The CSP measure was misconfigured, was bypassed, or prevented participants from implementing the given study task successfully. Thus, it is concluded that a correct configuration of CSPs is a time-consuming and complex task, and the decision to enable this security measure by default in the Play Framework should be carefully improved. However, the study unveiled a multi-layered issue likewise related to producers of frameworks, IDEs, browsers, and web service providers. Software developers do not only have

to be well informed or warned on insecure cases but also of secure cases in which security measures intervene by default unrecognizable in the background. Thus, in addition to the "Warn when unsafe" [75] pattern, a "Warn if secure" or "Warn at intervention" approach for CSP application by default is proposed to transparently inform developers about this security measure and prevent the adverse effects observed in this study.

The second laboratory study using eye-tracking contributed a first in-depth understanding of security-relevant code examples in the API documentation, examining a novel approach to help developers adopting API functionality securely. It was examined (**RQ6**) *how documentation writers can usefully integrate security-relevant information into a documentation website of a web API.*

Developers are focused on elements with code examples, mostly skimming the documentation while searching for a quick solution to their programming task. The placement of security-related code examples does matter and significantly impacts the time it takes to find security-relevant information. For this reason, security-relevant code examples and information should not only be integrated as a peripheral feature but should be placed as close as possible to the code examples of the APIs. Therefore, code comments have proven to be very helpful in pointing developers to security-relevant information that locates elsewhere in the documentation. The results also show that this approach significantly supports developers to produce secure CSP implementations with security APIs. Web API producers should consider these findings to integrate security-relevant information into their web API documentation usefully. However, developers have additional information needs that the tested approach can not meet. Furthermore, it was found that CSP examples have a limited ability to provide understanding and confidence in the security mechanism.

7.2. Future Work

The knowledge gained from this work identifies research needs and opportunities that can build on the results of this thesis.

The API usability model draws a fragmented picture of API usability, demanding further research filling gaps. Therefore it enables easy access to the field, also for novices, and it is easily extendable for new contributions to the field. An online version would be desirable so that the community can use it as an active research and development tool.

Although the studies of this thesis could support several of the introduced eleven specific usability characteristics of security APIs, this might still be an incomplete set of relevant topics. Future research should aim to confirm or revise the set in order to obtain a validated baseline for the usability evaluation of security APIs. In research they have already been applied as part of evaluation methods [218, 217].

Research on security API warnings should be extended to other types of security APIs, like controls APIs and other console environments, like terminals in IDEs, to gain profound design guidelines. Recently published work shows that the proposed approach to place security-relevant information in a console can help software developers to implement secure software also outside the context of cryptographic APIs [150]. The design recommendations for console warnings developed by this thesis should be further evaluated and developed. For this purpose, field studies would be particularly suitable to explore further insights into their weaknesses and strengths. Appropriate information flows for different amounts of security-relevant information, as described by the focus groups, should be further explored in detail. The aim should be to provide evaluated and approved guidelines that researchers and API producers can use to improve security API designs. This thesis has made the initial contribution to this goal.

In respect to security defaults, future work should focus on the complex interplay of warning message design, the quality and availability of information resources, and tools to support developers with the deployment of CSPs or other security controls. However, the results of this thesis have shown that the possibilities of individual stakeholders to influence information flows are limited. These results demand all actors of software development to promote and improve security-relevant information flows. The goal should be to empower software developers to write more secure web applications, resulting in less vulnerable software, putting end-users at risk.

This thesis did not address every influencing factor on security-relevant information flows in software development (cf. Figure 7.1, p. 172). Especially aspects of the software development management environment and the official API test environment were out of focus. Nevertheless, also open aspects in the addressed areas offer the potential for gaining knowledge and improving information flows as the application of the IFSIF model to state of the art showed (cf. Section 3.3, p. 31). It is also interesting that secure software development lifecycles currently address only minor parts of the IFSIF model, and aspects such as third party information, the API Design, and the implementation of information are not considered so far. Here it should be examined whether existing guidelines for secure software development should be extended.

Bibliography

[1] Y. Acar, C. Stransky, D. Wermke, C. Weir, M. L. Mazurek, and S. Fahl. "Developers Need Support, Too: A Survey of Security Advice for Software Developers". In: *IEEE Cybersecurity Development*. SecDev. Sept. 2017, pp. 22–26. DOI: 10.1109/SecDev.2017.17.

[2] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky. "Comparing the Usability of Cryptographic APIs". In: *IEEE Symposium on Security and Privacy*. S&P. IEEE, 2017, pp. 154–171. DOI: 10.1109/SP.2017.52.

[3] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky. "You Get Where You're Looking for: The Impact of Information Sources on Code Security". In: *IEEE Symposium on Security and Privacy*. S&P. IEEE, May 2016, pp. 289–305. DOI: 10.1109/SP.2016.25.

[4] Y. Acar, S. Fahl, and M. L. Mazurek. "You are Not Your Developer, Either: A Research Agenda for Usable Security and Privacy Research Beyond End Users". In: *IEEE Cybersecurity Development*. SecDev. IEEE, Nov. 2016, pp. 3–8. DOI: 10.1109/SecDev.2016.013.

[5] Y. Acar, C. Stransky, D. Wermke, M. L. Mazurek, and S. Fahl. "Security Developer Studies with GitHub Users: Exploring a Convenience Sample". In: *Symposium on Usable Privacy and Security*. SOUPS. USENIX Association, 2017, pp. 81–95. URL: https://www.usenix.org/conference/soups2017/technical-sessions/presentation/acar.

[6] A. Adams and M. A. Sasse. "Users Are Not the Enemy". In: *Communications of the ACM* 42.12 (Dec. 1999), pp. 40–46. ISSN: 0001-0782. DOI: 10.1145/322796.322806.

[7] D. Akhawe and A. P. Felt. "Alice in Warningland: A Large-scale Field Study of Browser Security Warning Effectiveness". In: *USENIX Security Symposium*. SSYM. USENIX Association, 2013, pp. 257–272. URL: http://dl.acm.org/citation.cfm?id=2534766.2534789.

[8] H. Almuhimedi, A. P. Felt, R. W. Reeder, and S. Consolvo. "Your Reputation Precedes You: History, Reputation, and the Chrome Malware Warning". In: *Symposium On Usable Privacy and Security*. SOUPS. USENIX Association, 2014, pp. 113–128. URL: https://www.usenix.org/conference/soups2014/proceedings/presentation/almuhimedi.

[9] Amnesty International. *Encryption - A matter of human rights*. 2016. URL: https://www.amnestyusa.org/wp-content/uploads/2017/04/encryption_-_a_matter_of_human_rights_-_pol_40-3682-2016.pdf (visited on 03/16/2021).

[10] AngularJS. *AngularJS: Developer Guide: Security*. 2020. URL: https://docs.angularjs.org/guide/security (visited on 03/16/2021).

[11] Apache. *Welcome to Apache Axis2/Java*. 2018. URL: https://axis.apache.org/axis2/java/core/ (visited on 03/16/2021).

[12] Apple. *Documentation Archive - What's New in Safari - Safari 10.0*. 2018. URL: https://developer.apple.com/library/archive/releasenotes/General/WhatsNewInSafari/Articles/Safari_10_0.html#//apple_ref/doc/uid/TP40014305-CH11-SW1 (visited on 03/16/2021).

[13] J. E. M. Araujo, S. Souza, and M. T. Valente. "Study on the relevance of the warnings reported by Java bug-finding tools". In: *IET Software* 5.4 (Aug. 2011), pp. 366–374. ISSN: 1751-8806. DOI: 10.1049/iet-sen.2009.0083.

[14] O. S. Architecture. *Control Catalog*. 2021. URL: http://www.opensecurityarchitecture.org/cms/library/0802control-catalogue (visited on 03/16/2021).

[15] O. S. Architecture. *Security Architecture Patterns*. 2021. URL: http://www.opensecurityarchitecture.org/cms/library/patternlandscape (visited on 03/16/2021).

[16] M. Asaduzzaman, C. K. Roy, S. Monir, and K. A. Schneider. "Exploring API Method Parameter Recommendations". In: *International Conference on Software Maintenance and Evolution*. ICSME. IEEE, 2015. DOI: 10.1109/ICSM.2015.7332473.

[17] H. Assal, S. Chiasson, and R. Biddle. "Cesar: Visual representation of source code vulnerabilities". In: *IEEE Symposium on Visualization for Cyber Security*. VizSec. Oct. 2016, pp. 1–8. DOI: 10.1109/VIZSEC.2016.7739576.

[18] H. Assal and S. Chiasson. "Security in the Software Development Lifecycle". In: *Symposium on Usable Privacy and Security*. SOUPS. USENIX Association, Aug. 2018, pp. 281–296. URL: https://www.usenix.org/conference/soups2018/presentation/assal.

[19] H. Assal and S. Chiasson. "'Think Secure from the Beginning': A Survey with Software Developers". In: *SIGCHI Conference on Human Factors in Computing Systems*. CHI. ACM, 2019. DOI: 10.1145/3290605.3300519.

[20] ATLAS.ti. *ATLAS.ti 8 Mac User Manual, updated for program version 8.4*. 2019. URL: https://downloads.atlasti.com/docs/manual/manual_a8_mac_en.pdf (visited on 03/16/2021).

[21] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix. "Using Static Analysis to Find Bugs". In: *IEEE Software* 25.5 (Sept. 2008), pp. 22–29. DOI: 10.1109/MS.2008.130.

[22] N. Ayewah and W. Pugh. "A report on a survey and study of static analysis users". In: *Workshop on Defects in Large Software Systems* (International Symposium on Software Testing and Analysis). DEFECTS. July 20, 2008, pp. 1–5. DOI: 10.1145/1390817.1390819.

[23] D. Baca. "Identifying Security Relevant Warnings from Static Code Analysis Tools through Code Tainting". In: *International Conference on Availability, Reliability and Security*. ARES. IEEE, Feb. 2010, pp. 386–390. DOI: 10.1109/ARES.2010.108.

[24] W. Bai, O. Akgul, and M. L. Mazurek. “A Qualitative Investigation of Insecure Code Propagation from Online Forums”. In: *IEEE Cybersecurity Development*. SecDev. IEEE, 2019, pp. 34–48. DOI: 10.1109/SecDev.2019.00016.

[25] T. Barik, D. Ford, E. Murphy-Hill, and C. Parnin. “How Should Compilers Explain Problems to Developers?” In: *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE. ACM, 2018, pp. 633–643. DOI: 10.1145/3236024.3236040.

[26] T. Barik, J. Smith, K. Lubick, E. Holmes, J. Feng, E. Murphy-Hill, and C. Parnin. “Do Developers Read Compiler Error Messages?” In: *International Conference on Software Engineering*. ICSE. IEEE, 2017, pp. 575–585. DOI: 10.1109/ICSE.2017.59.

[27] S. Bartsch. “Practitioners’ Perspectives on Security in Agile Development”. In: *International Conference on Availability, Reliability and Security*. ARES. IEEE Computer Society, 2011, pp. 479–484. DOI: 10.1109/ARES.2011.82.

[28] L. Bauer, C. Bravo-Lillo, L. Cranor, and E. Fragkaki. *Warning Design Guidelines*. Tech. rep. CMU-CyLab-13-002. CyLab, Carnegie Mellon University, 2013. URL: http://www.cylab.cmu.edu/research/techreports/2013/tr_cylab13002.html.

[29] B. A. Becker. “An Effective Approach to Enhancing Compiler Error Messages”. In: *ACM Technical Symposium on Computing Science Education*. SIGCSE. ACM, 2016, pp. 126–131. DOI: 10.1145/2839509.2844584.

[30] D. J. Bernstein, T. Lange, and P. Schwabe. “The Security Impact of a New Cryptographic Library”. In: *International Conference on Cryptology and Information Security in Latin America*. LATINCRYPT. Springer Berlin Heidelberg, 2012, pp. 159–176. DOI: 10.1007/978-3-642-33481-8_9.

[31] B. Blakley and C. Heath. *Security Design Patterns*. Tech. rep. The Open Group, 2004. URL: http://pubs.opengroup.org/onlinepubs/9299969899/toc.pdf.

[32] J. Bloch. *A Brief, Opinionated History of the API*. Invited Talk at SPLASH 2014. Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU). Portland, 2014. URL: http://2014.splashcon.org/event/plateau2014-invited-speaker-josh-bloch.

[33] R. Brandom. *Former Equifax CEO blames breach on a single person who failed to deploy patch*. The Verge. Oct. 3, 2017. URL: https://www.theverge.com/2017/10/3/16410806/equifax-ceo-blame-breach-patch-congress-testimony (visited on 03/16/2021).

[34] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer. "Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code". In: *SIGCHI Conference on Human Factors in Computing Systems*. CHI. ACM Press, 2009. DOI: 10.1145/1518701.1518944.

[35] C. Bravo-Lillo, S. Komanduri, L. F. Cranor, R. W. Reeder, M. Sleeper, J. Downs, and S. Schechter. "Your Attention Please: Designing Security-decision UIs to Make Genuine Risks Harder to Ignore". In: *Symposium on Usable Privacy and Security*. SOUPS. ACM, 2013, 6:1–6:12. DOI: 10.1145/2501604.2501610.

[36] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov. "Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations". In: *IEEE Symposium on Security and Privacy*. S&P. 2014. DOI: 10.1109/SP.2014.15.

[37] M. Bruch, M. Monperrus, and M. Mezini. "Learning from Examples to Improve Code Completion Systems". In: *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ESEC/FSE. ACM Press, 2009. DOI: 10.1145/1595696.1595728.

[38] California State Legislature. *The California Consumer Privacy Act of 2018*. California Legislative Assembly Bill No. 375 CHAPTER 55, L119/1. 2018. URL: https://leginfo.legislature.ca.gov/faces/billTextClient.xhtml?bill_id=201720180AB375.

[39] S. Calzavara, A. Rabitti, and M. Bugliesi. "Content Security Problems?: Evaluating the Effectiveness of Content Security Policy in the Wild". In: *ACM SIGSAC Conference on Computer and Communications Security*. CCS. ACM, 2016, pp. 1365–1375. DOI: 10.1145/2976749.2978338.

[40] S. Calzavara, A. Rabitti, and M. Bugliesi. "Semantics-Based Analysis of Content Security Policy Deployment". In: *ACM Transactions on the Web* 12.2 (Jan. 2018), 10:1–10:36. ISSN: 1559-1131. DOI: 10.1145/3149408.

[41] Cambridge Dictionary. *Meaning of the internet in English*. 2021. URL: https://dictionary.cambridge.org/dictionary/english/internet (visited on 03/16/2021).

[42] S. Cass. *The 2017 Top Programming Languages - Python jumps to No. 1, and Swift enters the Top Ten*. IEEE Spectrum. 2017. URL: http://web.archive.org/web/20170718211203/https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages (visited on 03/16/2021).

[43] S. Cass. *The Top Programming Languages 2019 - Python remains the big kahuna, but specialist languages hold their own*. IEEE Spectrum. 2019. URL: https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019 (visited on 03/16/2021).

[44] E. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague. "OAuth Demystified for Mobile Application Developers". In: *ACM SIGSAC Conference on Computer and Communications Security*. CCS. ACM Press, 2014. DOI: 10.1145/2660267.2660323.

[45] M. Christakis and C. Bird. "What Developers Want and Need from Program Analysis: An Empirical Study". In: *IEEE/ACM International Conference on Automated Software Engineering*. ASE. ACM, 2016, pp. 332–343. DOI: 10.1145/2970276.2970347.

[46] C. Cimpanu. *Hackers are collecting payment details, user passwords from thousands of sites - Servers of at least seven companies compromised to deliver malicious code to thousands of sites*. ZDNet. 2019. URL: https://

www.zdnet.com/article/hackers-are-collecting-payment-details-user-passwords-from-4600-sites/ (visited on 03/16/2021).

[47] S. Clarke. "Measuring API Usability". In: *Dr. Dobb's Journal* 29 (2004), pp. 6–9. URL: http://www.drdobbs.com/windows/measuring-api-usability/184405654#.

[48] S. Clarke. "What is an End User Software Engineer?" In: *End-User Software Engineering*. Dagstuhl Seminar. 2007. URL: http://drops.dagstuhl.de/opus/volltexte/2007/1080/pdf/07081.ClarkeSteven.Paper.1080.pdf.

[49] S. Clarke. "How Usable Are Your APIs?" In: *Making software: what really works, and why we believe it*. Ed. by A. Oram, G. Wilson, A. Oram, and G. Wilson. Theory in practice. O'Reilly, 2010, pp. 545–565. ISBN: 978-0-596-80832-7.

[50] The MITRE Corporation. *CWE-329: Not Using a Random IV with CBC Mode*. 2006. URL: http://cwe.mitre.org/data/definitions/329.html.

[51] J. Daughtry, J. Lawrance, B. Myers, and A. Santos. *API Usability - Publications*. Google Sites. 2018. URL: https://sites.google.com/site/apiusability/resources/publications (visited on 03/16/2021).

[52] A. Dayaratna. *IDC's Worldwide Developer Census, 2018: Part-Time Developers Lead the Expansion of the Global Developer Population*. Oct. 2018. URL: https://web.archive.org/web/20200117065523/https://www.idc.com/getdoc.jsp?containerId=US44363318 (visited on 03/16/2021).

[53] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes. "Keep Me Updated: An Empirical Study of Third-Party Library Updatability on Android". In: *ACM SIGSAC Conference on Computer and Communications Security*. CCS. ACM, 2017, pp. 2187–2200. DOI: 10.1145/3133956.3134059.

[54] A. Deveria. *Can I use content security policy?* Can I use. 2021. URL: https://caniuse.com/?search=content%20security%20policy (visited on 03/16/2021).

[55] A. J. DeWitt and J. Kuljis. “Aligning Usability and Security: A Usability Study of Polaris”. In: *Symposium on Usable Privacy and Security*. SOUPS. ACM, 2006, pp. 1–7. DOI: 10.1145/1143120.1143122.

[56] C. Dietrich, K. Krombholz, K. Borgolte, and T. Fiebig. “Investigating System Operators’ Perspective on Security Misconfigurations”. In: *ACM SIGSAC Conference on Computer and Communications Security*. CCS. ACM, 2018, pp. 1272–1289. DOI: 10.1145/3243734.3243794.

[57] L. N. Q. Do, K. Ali, B. Livshits, E. Bodden, J. Smith, and E. Murphy-Hill. “Just-in-Time Static Analysis”. In: *ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA. ACM, 2017, pp. 307–317. DOI: 10.1145/3092703.3092705.

[58] E. Duala-Ekoko and M. P. Robillard. “Using Structure-Based Recommendations to Facilitate Discoverability in APIs”. In: *European Conference on Object-Oriented Programming*. ECOOP. 2011. DOI: 10.1007/978-3-642-22655-7_5.

[59] E. Duala-Ekoko and M. P. Robillard. “Asking and Answering Questions about Unfamiliar APIs: An Exploratory Study”. In: *International Conference on Software Engineering*. ICSE. IEEE, 2012. DOI: 10.1109/ICSE.2012.6227187.

[60] A. Edmundson, B. Holtkamp, E. Rivera, M. Finifter, A. Mettler, and D. Wagner. “An Empirical Study on the Effectiveness of Security Code Review”. In: *International Conference on Engineering Secure Software and Systems*. ESSoS. Springer-Verlag, 2013, pp. 197–212. DOI: 10.1007/978-3-642-36563-8_14.

[61] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. “An Empirical Study of Cryptographic Misuse in Android Applications”. In: *ACM SIGSAC Conference on Computer & Communications Security*. CCS. ACM, 2013, pp. 73–84. DOI: 10.1145/2508859.2516693.

[62] S. Egelman, L. F. Cranor, and J. Hong. “You’Ve Been Warned: An Empirical Study of the Effectiveness of Web Browser Phishing Warnings”. In: *SIGCHI*

Conference on Human Factors in Computing Systems. CHI. ACM, 2008, pp. 1065–1074. DOI: 10.1145/1357054.1357219.

[63] B. Ellis, J. Stylos, and B. Myers. "The Factory Pattern in API Design: A Usability Evaluation". In: *International Conference on Software Engineering*. ICSE. IEEE, 2007. DOI: 10.1109/ICSE.2007.85.

[64] T. Espinha, A. Zaidman, and H.-G. Gross. "Web API Growing Pains: Stories from Client Developers and Their Code". In: *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering, Software Evolution Week*. CSMR-WCRE. 2014. DOI: 10.1109/CSMR-WCRE.2014.6747228.

[65] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. "Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security". In: *ACM Conference on Computer and Communications Security*. CCS. ACM, 2012, pp. 50–61. DOI: 10.1145/2382196.2382205.

[66] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith. "Rethinking SSL Development in an Appified World". In: *ACM SIGSAC Conference on Computer and Communications Security*. CCS. ACM, 2013, pp. 49–60. DOI: 10.1145/2508859.2516655.

[67] P. J. Fahy. "Addressing some common problems in transcript analysis". In: *The International Review of Research in Open and Distributed Learning* 1.2 (2001), pp. 1–6. URL: https://auspace.athabascau.ca/handle/2149/1206.

[68] A. P. Felt, A. Ainslie, R. W. Reeder, S. Consolvo, S. Thyagaraja, A. Bettes, H. Harris, and J. Grimes. "Improving SSL Warnings: Comprehension and Adherence". In: *SIGCHI Conference on Human Factors in Computing Systems*. CHI. ACM, 2015, pp. 2893–2902. DOI: 10.1145/2702123.2702442.

[69] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. "Android Permissions: User Attention, Comprehension, and Behavior". In: *Symposium on Usable Privacy and Security*. SOUPS. ACM, 2012, 3:1–3:14. DOI: 10.1145/2335356.2335360.

[70] A. P. Felt, R. W. Reeder, H. Almuhimedi, and S. Consolvo. “Experimenting at Scale with Google Chrome’s SSL Warning”. In: *SIGCHI Conference on Human Factors in Computing Systems*. CHI. ACM, 2014, pp. 2667–2670. DOI: 10.1145/2556288.2557292.

[71] E. Fernandez-Buglioni. *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*. John Wiley & Sons, Apr. 2013. ISBN: 978-1-119-97048-4.

[72] R. T. Fielding. “REST: Architectural Styles and the Design of Network-based Software Architectures”. PhD thesis. University of California, Irvine, 2000. URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[73] T. Foremski. *Phishing: You’re not as good at spotting scams as you think you are*. ZDNet. 2020. URL: https://www.zdnet.com/article/phishing-is-becoming-more-sophisticated-only-5-can-spot-all-scams/ (visited on 03/16/2021).

[74] S. Garfinkel and H. R. Lipford. *Usable Security: History, Themes, and Challenges*. Morgan & Claypool, 2014. 150 pp. ISBN: 978-1-62705-529-1.

[75] S. L. Garfinkel. “Design Principles and Patterns for Computer Systems That Are Simultaneously Secure and Usable”. PhD thesis. Massachusetts Institute of Technology, 2005. URL: http://simson.net/thesis/.

[76] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. “The Most Dangerous Code in the World: Validating SSL Certificates in Non-browser Software”. In: *ACM Conference on Computer and Communications Security*. CCS. ACM, 2012, pp. 38–49. DOI: 10.1145/2382196.2382204.

[77] GitHub. *The State of the Octoverse*. 2019. URL: https://octoverse.github.com/#top-languages (visited on 03/16/2021).

[78] Go. *The Go Programming Language*. 2021. URL: https://golang.org/ (visited on 03/16/2021).

[79] Google. *Adding a Google Map with a Marker to Your Website*. Archived. 2019. URL: https://web.archive.org/web/20190715140525/https://developers.google.com/maps/documentation/javascript/adding-a-google-map (visited on 03/16/2021).

[80] Google. *Adding a Google Map with a Marker to Your Website*. 2021. URL: https://developers.google.com/maps/documentation/javascript/adding-a-google-map (visited on 03/16/2021).

[81] Google. *Best Practices for Security & Privacy*. Android Developers. 2021. URL: https://developer.android.com/training/best-security.html (visited on 03/16/2021).

[82] Google. *Chrome Platform Status*. 2021. URL: https://www.chromestatus.com/features#csp (visited on 03/16/2021).

[83] P. L. Gorski, Y. Acar, L. Iacono Lo, and S. Fahl. "Listen to Developers! A Participatory Design Study on Security Warnings for Cryptographic APIs". In: *SIGCHI Conference on Human Factors in Computing Systems*. CHI. ACM, 2020, pp. 1–13. DOI: 10.1145/3313831.3376142.

[84] P. L. Gorski and L. Lo Iacono. "Towards the Usability Evaluation of Security APIs". In: *International Symposium on Human Aspects of Information Security and Assurance*. University of Plymouth, 2016, pp. 252–265. URL: https://www.cscan.org/?page=openaccess&eid=17&id=287.

[85] P. L. Gorski, L. Lo Iacono, D. Wermke, C. Stransky, S. Möller, Y. Acar, and S. Fahl. "Developers Deserve Security Warnings, Too: On the Effect of Integrated Security Advice on Cryptographic API Misuse". In: *Symposium on Usable Privacy and Security*. SOUPS. USENIX Association, Aug. 2018, pp. 265–281. URL: https://www.usenix.org/conference/soups2018/presentation/gorski.

[86] P. L. Gorski, L. Lo Iacono, S. Wiefling, and S. Möller. "Warn if Secure or How to Deal with Security by Default in Software Development?" In: *International Symposium on Human Aspects of Information Security and Assurance*. University of Plymouth, 2018, pp. 170–190. URL: https://www.cscan.org/?page=openaccess&eid=20&id=388.

[87] P. L. Gorski, S. Möller, S. Wiefling, and L. Lo Iacono. "'I just looked for the solution!' On integrating Security-Relevant Information in Non-Security API Documentation to Support Secure Coding Practices". 2020. Unpublished.

[88] P. L. Gorski, E. von Zezschwitz, L. Lo Iacono, and M. Smith. "On providing systematized access to consolidated principles, guidelines and patterns for usable security research and development". In: *Oxford Journal of Cybersecurity* 5.tyz014 (1 2019), pp. 1–19. DOI: 10.1093/cybsec/tyz014.

[89] M. Green and M. Smith. "Developers are Not the Enemy!: The Need for Usable Security APIs". In: *IEEE Security & Privacy* 14.5 (Sept. 2016), pp. 40–46. ISSN: 1540-7993. DOI: 10.1109/MSP.2016.111.

[90] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig. "Trade-Offs in Continuous Integration: Assurance, Security, and Flexibility". In: *Joint Meeting on Foundations of Software Engineering*. ESEC/FSE. ACM, 2017, pp. 197–207. DOI: 10.1145/3106237.3106270.

[91] R. Hodson. *Analyzing documentary accounts*. Quantitative Applications in the Social Sciences 128. SAGE Publications, July 1999. ISBN: 978-0761917434.

[92] S. Indela, M. Kulkarni, K. Nayak, and T. Dumitraş. "Helping Johnny Encrypt: Toward Semantic Interfaces for Cryptographic Frameworks". In: *ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! ACM, 2016, pp. 180–196. DOI: 10.1145/2986012.2986024.

[93] International Telecommunication Union (ITU). *Measuring digital development - Facts and figures 2019*. 2019. URL: https://www.itu.int/en/ITU-D/Statistics/Documents/facts/FactsFigures2019.pdf (visited on 03/16/2021).

[94] S. Inzunza, R. Juárez-Ramírez, and S. Jiménez. "API Documentation". In: *Trends and Advances in Information Systems and Technologies*. Ed. by Á. Rocha, H. Adeli, L. P. Reis, and S. Costanzo. Springer International Publishing, 2018, pp. 229–239. DOI: 10.1007/978-3-319-77712-2_22.

[95] ISO. *ISO 9241-11:1998 - Ergonomic requirements for office workwith visual display terminals (VDTs) -Part 11 :Guidance on usability*. 1998. URL: https://www.iso.org/standard/16883.html.

[96] ISO. *ISO/IEC/IEEE 12207:2017 - Systems and software engineering — Software life cycle processes*. 2017. URL: https://www.iso.org/standard/63712.html. Published.

[97] ISO. *ISO 9241-11:2018 - Ergonomics of human-system interaction — Part 11: Usability: Definitions and concept*. 2018. URL: https://www.iso.org/standard/63500.html.

[98] ISO/IEC 25010. *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. First edition. 2011. URL: http://www.iso.org/iso/catalogue_detail.htm?csnumber=35733.

[99] ISO/IEC 6429. *Information technology – Control functions for coded character sets*. Third edition. 1992. URL: https://www.iso.org/standard/12782.html.

[100] C. Jacobsen. *Secure - HTTP middleware for Go that facilitates some quick security wins*. 2021. URL: https://github.com/unrolled/secure (visited on 03/16/2021).

[101] S. Jain and J. Lindqvist. "Should I Protect You? Understanding Developers' Behavior to Privacy-Preserving APIs". In: *Workshop on Usable Security*. USEC. Internet Society, Apr. 27, 2014. DOI: 10.14722/usec.2014.23045.

[102] Jenkins. *Jenkins User Documentation*. 2021. URL: https://jenkins.io/doc (visited on 03/16/2021).

[103] JetBrains. *The State of Developer Ecosystem 2019*. 2019. URL: https://www.jetbrains.com/lp/devecosystem-2019 (visited on 03/16/2021).

[104] JetBrains. *IntelliJ IDEA: The Java IDE for Professional Developers by JetBrains*. 2021. URL: https://www.jetbrains.com/idea/ (visited on 03/16/2021).

[105] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. "Why Don't Software Developers Use Static Analysis Tools to Find Bugs?" In: *International Conference on Software Engineering*. ICSE. IEEE, 2013, pp. 672–681. DOI: 10.1109/ICSE.2013.6606613.

[106] P. N. Johnson-Laird. *Mental Models: Towards a Cognitive Science of Language, Inference, and Consciousness*. Harvard University Press, 1986. ISBN: 978-0-674-56882-2.

[107] JSFiddle. *Create a new fiddle - JSFiddle*. 2021. URL: https://jsfiddle.net/ (visited on 03/16/2021).

[108] Jupyter. *The Jupyter Notebook*. 2021. URL: https://jupyter-notebook.readthedocs.io/en/stable/index.html (visited on 03/16/2021).

[109] C. Kern. *Preventing Security Bugs through Software Design*. Invited Talk at the 24th USENIX Security Symposium. Washington, D.C., USA, 2015. URL: https://www.usenix.org/conference/usenixsecurity15/symposium-program/presentation/kern.

[110] I. Kirlappos and M. A. Sasse. "What Usable Security Really Means: Trusting and Engaging Users". In: *International Conference on Human Aspects of Information Security, Privacy, and Trust*. Ed. by T. Tryfonas and I. Askoxylakis. HAS. Springer International Publishing, 2014, pp. 69–78. DOI: 10.1007/978-3-319-07620-1_7.

[111] M. Kranch and J. Bonneau. "Upgrading HTTPS in Mid-Air: An Empirical Study of Strict Transport Security and Key Pinning". In: *The Network and Distributed System Security Symposium*. NDSS. 2015. URL: http://www.internetsociety.org/doc/upgrading-https-mid-air-empirical-study-strict-transport-security-and-key-pinning.

[112] G. E. Krasner and S. T. Pope. "A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80". In: *Journal of Object-Oriented Programming* 1.3 (Aug. 1988), pp. 26–49. ISSN: 0896-8438.

[113] K. Krippendorff. *Content Analysis: An Introduction to Its Methodology*. 2nd ed. SAGE Publications, 2004. ISBN: 978-0761915454.

[114] R. A. Krueger and M. A. Casey. *Focus Groups: A Practical Guide for Applied Research*. 5th ed. SAGE Publications, 2015. ISBN: 978-1483365244.

[115] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler, and R. Kamath. "CogniCrypt: Supporting Developers in Using Cryptography". In: *IEEE/ACM International Conference on Automated Software Engineering*. ASE. IEEE, 2017, pp. 931–936. DOI: 10.1109/ASE.2017.8115707.

[116] W. Li and C. J. Mitchell. "Security Issues in OAuth 2.0 SSO Implementations". In: *International Information Security Conference*. ISC. 2014. DOI: 10.1007/978-3-319-13257-0_34.

[117] LimeSurvey. *LimeSurvey: the online survey tool - open source surveys*. 2021. URL: https://www.limesurvey.org/ (visited on 03/16/2021).

[118] H. Lipford, T. Thomas, B. Chu, and E. Murphy-Hill. "Interactive Code Annotation for Security Vulnerability Detection". In: *ACM Workshop on Security Information Workers*. SIW. ACM, 2014, pp. 17–22. DOI: 10.1145/2663887.2663901.

[119] D. Litzenberger. *PyCrypto API Documentation*. 2012. URL: https://www.dlitz.net/software/pycrypto/api/current/ (visited on 03/16/2021).

[120] D. Litzenberger. *Python Cryptography Toolkit*. 2013. URL: https://github.com/dlitz/pycrypto/blob/master/Doc/pycrypt.rst (visited on 03/16/2021).

[121] S. Livingstone, D. Kardefelt-Winther, and M. Saeed. *GLOBAL KIDS ONLINE - Comparative report*. United Nations Children's Fund (UNICEF). 2019. URL: https://www.unicef-irc.org/publications/pdf/GKO%20LAYOUT%20MAIN%20REPORT.pdf (visited on 03/16/2021).

[122] L. Lo Iacono and P. L. Gorski. "I Do and I Understand. Not Yet True for Security APIs. So Sad". In: *European Workshop on Usable Security* (2nd IEEE European Symposium on Security and Privacy). EuroUSEC. Internet Society, 2017, pp. 1–11. DOI: 10.14722/eurousec.2017.23015.

[123] L. Lo Iacono, M. Smith, E. von Zezschwitz, P. L. Gorski, and P. Nehren. "Consolidating Principles and Patterns for Human-centred Usable Security Research and Development". In: *European Workshop on Usable Security*. EuroUSEC. Internet Society, Apr. 23, 2018. DOI: 10.14722/eurousec.2018.23010.

[124] S. G. McLellan, A. W. Roesler, J. T. Tempest, and C. I. Spinuzzi. "Building More Usable APIs". In: *IEEE Software* 15.3 (May 1998), pp. 78–86. DOI: 10.1109/52.676963.

[125] MDN Web Docs. *Content-Security-Policy - Browser compatibility*. Mozilla. 2021. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy#Browser_compatibility (visited on 03/16/2021).

[126] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*. CRC press, 1996. ISBN: 978-0849385230.

[127] M. Meng, S. Steinhardt, and A. Schubert. "Application Programming Interface Documentation: What Do Software Developers Want?" In: *Journal of Technical Writing and Communication* 48.3 (2018), pp. 295–330. DOI: 10.1177/0047281617721853.

[128] M. Meng, S. Steinhardt, and A. Schubert. "How Developers Use API Documentation: An Observation Study". In: *Communication Design Quarterly* 7.2 (Aug. 2019), pp. 40–49. DOI: 10.1145/3358931.3358937.

[129] K. Mindermann and S. Wagner. "Usability and Security Effects of Code Examples on Crypto APIs". In: *Annual Conference on Privacy, Security and Trust*. PST. Aug. 2018. DOI: 10.1109/PST.2018.8514203.

[130] M. Monperrus, M. Eichberg, E. Tekes, and M. Mezini. "What should developers be aware of? An empirical study on the directives of API documentation". In: *Empirical Software Engineering* 17.6 (Dec. 2012), pp. 703–737. ISSN: 1573-7616. URL: https://doi.org/10.1007/s10664-011-9186-4.

[131] M. Mooty, A. Faulring, J. Stylos, and B. A. Myers. "Calcite: Completing Code Completion for Constructors Using Crowds". In: *IEEE Symposium on Visual Languages and Human-Centric Computing*. VL/HCC. 2010. DOI: 10.1109/VLHCC.2010.12.

[132] B. A. Myers and J. Stylos. "Improving API Usability". In: *Communications of the ACM* 59.6 (May 2016), pp. 62–69. DOI: 10.1145/2896587.

[133] S. Nadi, S. Krüger, M. Mezini, and E. Bodden. "'Jumping Through Hoops': Why do Java Developers Struggle With Cryptography APIs?" In: *International Conference on Software Engineering*. ICSE. ACM, 2016, pp. 935–946. DOI: 10.1145/2884781.2884790.

[134] A. Naiakshina, A. Danilova, E. Gerlitz, E. von Zezschwitz, and M. Smith. "'If You Want, I Can Store the Encrypted Password': A Password-Storage Field Study with Freelance Developers". In: *SIGCHI Conference on Human Factors in Computing Systems*. CHI. ACM, 2019. DOI: 10.1145/3290605.3300370.

[135] A. Naiakshina, A. Danilova, C. Tiefenau, M. Herzog, S. Dechand, and M. Smith. "Why Do Developers Get Password Storage Wrong?: A Qualitative Usability Study". In: *ACM SIGSAC Conference on Computer and Communications Security*. CCS. ACM, 2017, pp. 311–328. DOI: 10.1145/3133956.3134082.

[136] A. Naiakshina, A. Danilova, C. Tiefenau, and M. Smith. "Deception Task Design in Developer Password Studies: Exploring a Student Sample". In: *Symposium on Usable Privacy and Security*. SOUPS. USENIX Association, Aug. 2018, pp. 297–313. URL: https://www.usenix.org/conference/soups2018/presentation/naiakshina.

[137] D. C. Nguyen, D. Wermke, Y. Acar, M. Backes, C. Weir, and S. Fahl. "A Stitch in Time: Supporting Android Developers in Writing Secure Code". In: *ACM SIGSAC Conference on Computer and Communications Security*. CCS. ACM, 2017, pp. 1065–1077. DOI: 10.1145/3133956.3133977.

[138] J. Nielsen. "Enhancing the Explanatory Power of Usability Heuristics". In: *SIGCHI Conference on Human Factors in Computing Systems*. CHI. ACM, 1994, pp. 152–158. DOI: 10.1145/191666.191729.

[139] S. O'Grady. *The RedMonk Programming Language Rankings: January 2020*. Feb. 28, 2020. URL: https://redmonk.com/sogrady/2020/02/28/language-rankings-1-20/ (visited on 03/16/2021).

[140] D. Oliveira, M. Rosenthal, N. Morin, K.-C. Yeh, J. Cappos, and Y. Zhuang. "It's the Psychology Stupid: How Heuristics Explain Software Vulnerabilities and How Priming Can Illuminate Developer's Blind Spots". In: *Annual Computer Security Applications Conference*. ACSAC. ACM, 2014, pp. 296–305. DOI: 10.1145/2664243.2664254.

[141] D. S. Oliveira, T. Lin, M. S. Rahman, R. Akefirad, D. Ellis, E. Perez, R. Bobhate, L. A. DeLong, J. Cappos, and Y. Brun. "API Blindspots: Why Experienced Developers Write Vulnerable Code". In: *Symposium on Usable Privacy and Security*. SOUPS. USENIX Association, 2018, pp. 315–328. URL: https://www.usenix.org/conference/soups2018/presentation/oliveira.

[142] C. Omar, Y. S. Yoon, T. D. LaToza, and B. A. Myers. "Active Code Completion". In: *International Conference on Software Engineering*. ICSE. IEEE, 2012. DOI: 10.1109/ICSE.2012.6227133.

[143] OWASP. *OWASP Secure Coding Practices - Quick Reference Guide Version 2.0*. The Open Web Application Security Project. 2010. URL: https://owasp.org/www-project-secure-coding-practices-quick-reference-guide/migrated_content (visited on 03/16/2021).

[144] OWASP. *OWASP Top 10 - 2017 - The Ten Most Critical Web Application Security Risks*. The Open Web Application Security Project. 2017. URL: https://owasp.org/www-project-top-ten/ (visited on 03/16/2021).

[145] OWASP. *API Security Top 10 2019*. The Open Web Application Security Project. 2019. URL: https://owasp.org/www-project-api-security/ (visited on 03/16/2021).

[146] OWASP. *Cross Site Scripting (XSS)*. 2021. URL: https://owasp.org/www-community/attacks/xss/ (visited on 03/16/2021).

[147] K. Patil and F. Braun. “A Measurement Study of the Content Security Policy on Real-World Applications”. In: *International Journal of Network Security* 18.2 (Mar. 2016), pp. 383–392. ISSN: 1816-3548. URL: http://ijns.jalaxy.com.tw/download_paper.jsp?PaperID=IJNS-2014-08-03-1&PaperName=ijns-v18-n2/ijns-2016-v18-n2-p383-392.pdf.

[148] K. Patil, T. Vyas, F. Braun, M. Goodwin, and Z. Liang. “Poster: UserCSP-User Specified Content Security Policies”. In: *Symposium On Usable Privacy and Security*. SOUPS. 2013. URL: http://cups.cs.cmu.edu/soups/2013/posters/soups13_posters-final1.pdf.

[149] C. Pautasso, O. Zimmermann, and F. Leymann. “Restful Web Services vs. "Big" Web Services: Making the Right Architectural Decision”. In: *International Conference on World Wide Web*. WWW. ACM, 2008, pp. 805–814. DOI: 10.1145/1367497.1367606.

[150] S. T. Peddinti, I. Bilogrevic, N. Taft, M. Pelikan, Ú. Erlingsson, P. Anthonysamy, and G. Hogben. “Reducing Permission Requests in Mobile Apps”. In: *Internet Measurement Conference*. IMC. Association for Computing Machinery, 2019, pp. 259–266. DOI: 10.1145/3355369.3355584.

[151] M. Piccioni, C. A. Furia, and B. Meyer. “An Empirical Study of API Usability”. In: *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM. 2013, pp. 5–14. DOI: 10.1109/ESEM.2013.14.

[152] Play Framework. *Documentation – Configuration file syntax and features*. 2018. URL: https://www.playframework.com/documentation/2.6.x/ConfigFile (visited on 03/16/2021).

[153] Play Framework. *Play samples - play-java-starter-example*. 2019. URL: https://github.com/playframework/play-samples/tree/2.6.x/play-java-starter-example (visited on 03/16/2021).

[154] Playframework. *Playframework - Content Security Policy Filter*. 2018. URL: https://github.com/playframework/playframework/pull/8242 (visited on 03/16/2021).

[155] A. Poller, L. Kocksch, S. Türpe, F. A. Epp, and K. Kinder-Kurlanda. "Can Security Become a Routine? A Study of Organizational Change in an Agile Software Development Group". In: *ACM Conference on Computer Supported Cooperative Work and Social Computing*. CSCW. ACM, 2017, pp. 2489–2503. DOI: 10.1145/2998181.2998191.

[156] ProgrammableWeb. *APIs show Faster Growth Rate in 2019 than Previous Years*. 2019. URL: https://www.programmableweb.com/news/apis-show-faster-growth-rate-2019-previous-years/research/2019/07/17 (visited on 03/16/2021).

[157] Python Cryptographic Authority. *Welcome to pyca/cryptography*. 2021. URL: https://cryptography.io/ (visited on 03/16/2021).

[158] A. A. U. Rahman and L. Williams. "Software Security in DevOps: Synthesizing Practitioners' Perceptions and Practices". In: *IEEE/ACM International Workshop on Continuous Software Evolution and Delivery*. May 2016, pp. 70–76. DOI: 10.1145/2896941.2896946.

[159] B. Reaves, N. Scaife, A. Bates, P. Traynor, and K. R. B. Butler. "Mo(bile) Money, Mo(bile) Problems: Analysis of Branchless Banking Applications in the Developing World". In: *USENIX Security Symposium*. SSYM. USENIX Association, 2015, pp. 17–32. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/reaves.

[160] RFC5246. *The Transport Layer Security (TLS) Protocol Version 1.3*. Ed. by E. Rescorla. Proposed Standard. Internet Engineering Task Force (IETF), 2018. URL: https://tools.ietf.org/html/rfc8446.

[161] RFC6749. *The OAuth 2.0 Authorization Framework*. Ed. by D. Hardt. Proposed Standard. Internet Engineering Task Force (IETF), 2012. URL: https://tools.ietf.org/html/rfc6749.

[162] RFC6797. *HTTP Strict Transport Security (HSTS)*. Ed. by J. Hodges, C. Jackson, and A. Barth. Proposed Standard. Internet Engineering Task Force (IETF), 2012. URL: https://tools.ietf.org/html/rfc6797.

[163] RFC7465. *Prohibiting RC4 Cipher Suites*. Ed. by A. Popov. Proposed Standard. Internet Engineering Task Force (IETF), 2015. URL: https://tools.ietf.org/html/rfc7465.

[164] M. P. Robillard. "What Makes APIs Hard to Learn? Answers from Developers". In: *IEEE Software* 26.6 (Nov. 2009), pp. 27–34. ISSN: 0740-7459. DOI: 10.1109/MS.2009.193.

[165] S. Roth, T. Barron, S. Calzavara, N. Nikiforakis, and B. Stock. "Complex Security Policy? A Longitudinal Analysis of Deployed Content Security Policies". In: *The Network and Distributed System Security Symposium*. NDSS. 2020. URL: https://www.ndss-symposium.org/ndss-paper/complex-security-policy-a-longitudinal-analysis-of-deployed-content-security-policies/.

[166] N. Sakimura, J. Bradley, M. Jones, B. d. Medeiros, and C. Mortimore. *OpenID Connect Core 1.0 incorporating errata set 1*. Final. The OpenID Foundation, 2014. URL: http://openid.net/specs/openid-connect-core-1_0.html.

[167] J. H. Saltzer and M. D. Schroeder. "The Protection of Information in Computer Systems". In: *Proceedings of the IEEE* 63.9 (1975), pp. 1278–1308. DOI: 10.1109/PROC.1975.9939.

[168] M. A. Sasse, S. Brostoff, and D. Weirich. "Transforming the "Weakest Link" - a Human Computer Interaction Approach to Usable and Effective Security". English. In: *BT Technology Journal* 19.3 (2001), pp. 122–131. ISSN: 1358-3948. DOI: 10.1023/A:1011902718709.

[169] M. A. Sasse and M. Smith. "The Security-Usability Tradeoff Myth". In: *IEEE Security & Privacy* 14.5 (Sept. 2016), pp. 11–13. ISSN: 1540-7993. DOI: 10.1109/MSP.2016.102.

[170] T. Scheller and E. Kuhn. "Influencing Factors on the Usability of API Classes and Methods". In: *International Conference and Workshops on Engineering of Computer-Based Systems*. ECBS. IEEE, 2012. DOI: 10.1109/ECBS.2012.27.

[171] T. Scheller and E. Kühn. “Influence of Code Completion Methods on the Usability of APIs”. In: *IASTED International Conference on Software Engineering*. SE. 2013. DOI: 10.2316/P.2013.796-027.

[172] T. Scheller and E. Kühn. “Usability Evaluation of Configuration-Based API Design Concepts”. In: *International Conference on Human Factors in Computing & Informatics*. SouthCHI. 2013. DOI: 10.1007/978-3-642-39062-3_4.

[173] B. Schneier. *The Process of Security*. 2000. URL: https://www.schneier.com/essays/archives/2000/04/the_process_of_secur.html (visited on 03/16/2021).

[174] SELFHTML-Wiki. *Sicherheit/Content Security Policy*. 2021. URL: https://wiki.selfhtml.org/wiki/Sicherheit/Content_Security_Policy (visited on 03/16/2021).

[175] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford. “How Developers Diagnose Potential Security Vulnerabilities with a Static Analysis Tool”. In: *IEEE Transactions on Software Engineering* 45.9 (Sept. 2019), pp. 877–897. ISSN: 2326-3881. DOI: 10.1109/TSE.2018.2810116.

[176] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford. “Questions Developers Ask While Diagnosing Potential Security Vulnerabilities with Static Analysis”. In: *Joint Meeting on Foundations of Software Engineering*. ESEC/FSE. ACM, 2015, pp. 248–259. DOI: 10.1145/2786805.2786812.

[177] O. B. Software. *OBS Studio*. 2021. URL: https://obsproject.com (visited on 03/16/2021).

[178] Stack Exchange. *Welcome to Information Security Stack Exchange*. 2021. URL: https://security.stackexchange.com/tour (visited on 03/16/2021).

[179] Stack Overflow. *Developer Survey Results 2019*. 2019. URL: https://insights.stackoverflow.com/survey/2019 (visited on 03/16/2021).

[180] S. Stamm, B. Sterne, and G. Markham. “Reining in the Web with Content Security Policy”. In: *International Conference on World Wide Web*. WWW. ACM, 2010, pp. 921–930. DOI: 10.1145/1772690.1772784.

[181] Statista. *Programming/development tools used by software developers worldwide in 2018 and 2018*. 2019. URL: https://www.statista.com/statistics/869106/worldwide-software-developer-survey-tools-in-use/ (visited on 03/16/2021).

[182] C. Stransky, Y. Acar, D. C. Nguyen, D. Wermke, D. Kim, E. M. Redmiles, M. Backes, S. Garfinkel, M. L. Mazurek, and S. Fahl. "Lessons Learned from Using an Online Platform to Conduct Large-Scale, Online Controlled Security Experiments with Software Developers". In: *USENIX Workshop on Cyber Security Experimentation and Test*. CSET. USENIX Association, Aug. 2017. URL: https://www.usenix.org/conference/cset17/workshop-program/presentation/stransky.

[183] C. Stransky and L. Löhle. *Developer Observatory*. GitHub. 2020. URL: https://github.com/developer-observatory/developer-observatory (visited on 03/16/2021).

[184] J. Stylos and S. Clarke. "Usability Implications of Requiring Parameters in Objects' Constructors". In: *International Conference on Software Engineering*. ICSE. IEEE, May 2007, pp. 529–539. DOI: 10.1109/ICSE.2007.92.

[185] J. Stylos and B. A. Myers. "Mica: A Web-Search Tool for Finding API Components and Examples". In: *IEEE Symposium on Visual Languages and Human-Centric Computing*. VL/HCC. 2006. DOI: 10.1109/VLHCC.2006.32.

[186] J. Stylos and B. A. Myers. "Mapping the Space of API Design Decisions". In: *IEEE Symposium on Visual Languages and Human-Centric Computing*. VL/HCC. IEEE, Sept. 2007, pp. 50–60. DOI: 10.1109/VLHCC.2007.44.

[187] J. Stylos and B. A. Myers. "The Implications of Method Placement on API Learnability". In: *ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE. 2008. DOI: 10.1145/1453101.1453117.

[188] S.-T. Sun and K. Beznosov. "The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems". In: *ACM Conference on Computer and Communications Security*. CCS. ACM, 2012, pp. 378–390. DOI: 10.1145/2382196.2382238.

[189] J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor. "Crying Wolf: An Empirical Study of SSL Warning Effectiveness". In: *USENIX Security Symposium*. SSYM. USENIX Association, 2009. URL: http://dl.acm.org/citation.cfm?id=1855768.1855793.

[190] M. Tabassum, S. Watson, B. Chu, and H. R. Lipford. "Evaluating Two Methods for Integrating Secure Programming Education". In: *ACM Technical Symposium on Computer Science Education*. SIGCSE. ACM, 2018, pp. 390–395. DOI: 10.1145/3159450.3159511.

[191] M. Tabassum, S. Watson, and H. Richter Lipford. "Comparing Educational Approaches to Secure programming: Tool vs. TA". In: *Workshop on Security Information Workers* (Thirteenth Symposium on Usable Privacy and Security). WSIW. USENIX Association, July 2017. URL: https://www.usenix.org/conference/soups2017/workshop-program/wsiw2017/tabassum.

[192] M. Tahaei and K. Vaniea. "A Survey on Developer-Centred Security". In: *European Workshop on Usable Security* (4th IEEE European Symposium on Security and Privacy). EuroUSEC. June 2019, pp. 129–138. DOI: 10.1109/EuroSPW.2019.00021.

[193] The European Parliament and the Council of the European Union. *Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)*. Official Journal of the European Union, L119/1. 2016. URL: http://data.europa.eu/eli/reg/2016/679/oj.

[194] The MITRE Corporation. *CWE/SANS Top 25 Most Dangerous Software Errors Version: 1.0.3*. Ed. by S. Christey. 2011. URL: http://cwe.mitre.org/top25/archive/2011/2011_cwe_sans_top25.html (visited on 03/16/2021).

[195] T. Thomas, B. Chu, H. Lipford, J. Smith, and E. Murphy-Hill. "A study of interactive code annotation for access control vulnerabilities". In: *IEEE*

Symposium on Visual Languages and Human-Centric Computing. VL/HCC. IEEE, Oct. 2015, pp. 73–77. DOI: 10.1109/VLHCC.2015.7357200.

[196] T. W. Thomas, H. Lipford, B. Chu, J. Smith, and E. Murphy-Hill. "What Questions Remain? An Examination of How Developers Understand an Interactive Static Analysis Tool". In: *Workshop on Security Information Workers* (12th Symposium on Usable Privacy and Security). WSIW. USENIX Association, June 22, 2016. URL: https://www.usenix.org/conference/soups2016/workshop-program/wsiw16/presentation/thomas.

[197] V. J. Traver. "On Compiler Error Messages: What They Say and What They Mean". In: *Advances in Human-Computer Interaction* 2010.602570 (Jan. 2010), pp. 1–26. DOI: 10.1155/2010/602570.

[198] S. Türpe. "Idea: Usable Platforms for Secure Programming – Mining Unix for Insight and Guidelines". In: *International Symposium on Engineering Secure Software and Systems*. Vol. 9639. LNCS. Springer International Publishing, Apr. 2016, pp. 207–215. DOI: 10.1007/978-3-319-30806-7_13.

[199] S. Türpe, L. Kocksch, and A. Poller. "Penetration Tests a Turning Point in Security Practices? Organizational Challenges and Implications in a Software Development Team". In: *Workshop on Security Information Workers*. WSIW. USENIX Association, June 2016. URL: https://www.usenix.org/conference/soups2016/workshop-program/wsiw16/presentation/turpe.

[200] A. Tversky and D. Kahneman. "The framing of decisions and the psychology of choice". In: *Science* 211.4481 (1981), pp. 453–458. ISSN: 0036-8075. DOI: 10.1126/science.7455683.

[201] G. Uddin and M. P. Robillard. "How API Documentation Fails". In: *IEEE Software* 32.4 (July 2015), pp. 68–75. ISSN: 1937-4194. DOI: 10.1109/MS.2014.80.

[202] A. Vance, B. Kirwan, D. Bjornn, J. Jenkins, and B. B. Anderson. "What Do We Really Know About How Habituation to Warnings Occurs Over Time?: A Longitudinal fMRI Study of Habituation and Polymorphic Warnings". In:

SIGCHI Conference on Human Factors in Computing Systems. CHI. ACM, 2017, pp. 2215–2227. DOI: 10.1145/3025453.3025896.

[203] Verizon. *2019 Data Breach Investigations Report*. 2019. URL: https://enterprise.verizon.com/resources/reports/2019-data-breach-investigations-report.pdf (visited on 03/16/2021).

[204] D. Votipka, K. R. Fulton, J. Parker, M. Hou, M. L. Mazurek, and M. Hicks. "Understanding security mistakes developers make: Qualitative analysis from Build It, Break It, Fix It". In: *USENIX Security Symposium*. SSYM. USENIX Association, Aug. 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/votipka-understanding.

[205] E. W. *People: The Strongest Link*. Keynote presentation at CyberUK In Practice 2017. Liverpool, UK: NCSC, 2017. URL: https://www.ncsc.gov.uk/speech/people--the-strongest-link.

[206] W3Schools. *Browser Statistics - The Most Popular Browsers*. 2021. URL: https://www.w3schools.com/browsers/default.asp (visited on 03/16/2021).

[207] H. Wang, Y. Zhang, J. Li, H. Liu, W. Yang, B. Li, and D. Gu. "Vulnerability Assessment of OAuth Implementations in Android Applications". In: *Annual Computer Security Applications Conference*. ACSAC. ACM Press, 2015. DOI: 10.1145/2818000.2818024.

[208] R. Wang, S. Chen, and X. Wang. "Signing Me onto Your Accounts through Facebook and Google: A Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services". In: *IEEE Symposium on Security and Privacy*. S&P. 2012. DOI: 10.1109/SP.2012.30.

[209] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc. "CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy". In: *ACM SIGSAC Conference on Computer and Communications Security*. CCS. ACM, 2016, pp. 1376–1387. DOI: 10.1145/2976749.2978363.

[210] J. Weinberger and A. P. Felt. “A Week to Remember: The Impact of Browser Warning Storage Policies”. In: *Symposium on Usable Privacy and Security*. SOUPS. USENIX Association, 2016, pp. 15–25. URL: https://www.usenix.org/conference/soups2016/technical-sessions/presentation/weinberger.

[211] C. Weir, A. Rashid, and J. Noble. “How to Improve the Security Skills of Mobile App Developers? Comparing and Contrasting Expert Views”. In: *Workshop on Security Information Workers*. WSIW. USENIX Association, June 2016. URL: https://www.usenix.org/conference/soups2016/workshop-program/wsiw16/presentation/weir.

[212] M. Weissbacher, T. Lauinger, and W. Robertson. “Why Is CSP Failing? Trends and Challenges in CSP Adoption”. In: *Research in Attacks, Intrusions and Defenses*. Ed. by A. Stavrou, H. Bos, and G. Portokalidis. Springer International Publishing, 2014, pp. 212–233. DOI: 10.1007/978-3-319-11379-1_11.

[213] M. West and J. Medley. *Content Security Policy*. 2020. URL: https://developers.google.com/web/fundamentals/security/csp/ (visited on 03/16/2021).

[214] M. Whitney, H. R. Lipford, B. Chu, and T. Thomas. “Embedding Secure Coding Instruction Into the IDE: Complementing Early and Intermediate CS Courses With ESIDE”. In: *Journal of Educational Computing Research* 56.3 (2018), pp. 415–438. DOI: 10.1177/0735633117708816.

[215] M. Whitney, H. Lipford-Richter, B. Chu, and J. Zhu. “Embedding Secure Coding Instruction into the IDE: A Field Study in an Advanced CS Course”. In: *ACM Technical Symposium on Computer Science Education*. SIGCSE. ACM, 2015, pp. 60–65. DOI: 10.1145/2676723.2677280.

[216] A. Whitten and J. D. Tygar. “Why Johnny Can’t Encrypt: A Usability Evaluation of PGP 5.0”. In: *USENIX Security Symposium*. SSYM. USENIX Association, 1999, pp. 14–14. URL: http://dl.acm.org/citation.cfm?id=1251421.1251435.

[217] C. Wijayarathna and N. A. G. Arachchilage. "Why Johnny Can't Store Passwords Securely? A Usability Evaluation of Bouncycastle Password Hashing". In: *International Conference on Evaluation and Assessment in Software Engineering*. EASE. ACM, 2018, pp. 205–210. DOI: 10.1145/3210459.3210483.

[218] C. Wijayarathna, N. A. G. Arachchilage, and J. Slay. "A Generic Cognitive Dimensions Questionnaire to Evaluate the Usability of Security APIs". In: *Human Aspects of Information Security, Privacy and Trust*. HAS. Springer International Publishing, 2017, pp. 160–173. DOI: 10.1007/978-3-319-58460-7_11.

[219] S. Winter, S. Wagner, and F. Deissenboeck. "A Comprehensive Model of Usability". In: *IFIP International Conference on Engineering for Human-Computer Interaction*. EIS. 2007. DOI: 10.1007/978-3-540-92698-6_7.

[220] J. Witschey, S. Xiao, and E. Murphy-Hill. "Technical and Personal Factors Influencing Developers' Adoption of Security Tools". In: *ACM Workshop on Security Information Workers*. SIW. ACM, 2014, pp. 23–26. DOI: 10.1145/2663887.2663898.

[221] J. Witschey, O. Zielinska, A. Welk, E. Murphy-Hill, C. Mayhorn, and T. Zimmermann. "Quantifying Developers' Adoption of Security Tools". In: *Joint Meeting on Foundations of Software Engineering*. ESEC/FSE. ACM, 2015, pp. 260–271. DOI: 10.1145/2786805.2786816.

[222] World Wide Web Consortium (W3C). *Content Security Policy Level 3*. Working Draft. Mar. 2021. URL: https://www.w3.org/TR/CSP3/.

[223] G. Wurster and P. C. van Oorschot. "The Developer is the Enemy". In: *New Security Paradigms Workshop*. NSPW. ACM, 2008, pp. 89–97. DOI: 10.1145/1595676.1595691.

[224] S. Xiao, J. Witschey, and E. Murphy-Hill. "Social Influences on Secure Development Tool Adoption: Why Security Tools Spread". In: *ACM Conference on Computer Supported Cooperative Work & Social Computing*. CSCW. ACM, 2014, pp. 1095–1106. DOI: 10.1145/2531602.2531722.

[225] J. Xie, H. R. Lipford, and B. Chu. "Why do programmers make security errors?" In: *IEEE Symposium on Visual Languages and Human-Centric Computing*. VL/HCC. Sept. 2011, pp. 161–164. DOI: 10.1109/VLHCC.2011.6070393.

[226] J. Xie, B. Chu, H. R. Lipford, and J. T. Melton. "ASIDE: IDE support for web application security". In: *Annual Computer Security Applications Conference*. ACSAC. Dec. 5, 2011, pp. 267–276. DOI: 10.1145/2076732.2076770.

[227] J. Xie, H. Lipford, and B.-T. Chu. "Evaluating Interactive Support for Secure Programming". In: *SIGCHI Conference on Human Factors in Computing Systems*. CHI. ACM, 2012, pp. 2707–2716. DOI: 10.1145/2207676.2208665.

[228] J. Yoder and J. Barcalow. "Architectural Patterns for Enabling Application Security". In: *Conference on Patterns Language of Programming*. 1997. URL: https://joeyoder.com/PDFs/appsec.pdf.

[229] K. Yskout, R. Scandariato, and W. Joosen. "Does Organizing Security Patterns Focus Architectural Choices?" In: *International Conference on Software Engineering*. ICSE. IEEE Press, 2012, pp. 617–627. DOI: 10.1109/ICSE.2012.6227155.

[230] K. Yskout, R. Scandariato, and W. Joosen. "Do Security Patterns Really Help Designers?" In: *International Conference on Software Engineering*. ICSE. IEEE Press, 2015, pp. 292–302. DOI: 10.1109/ICSE.2015.49.

[231] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. "MAPO: Mining and Recommending API Usage Patterns". In: *European Conference on Object-Oriented Programming*. ECOOP. 2009. DOI: 10.1007/978-3-642-03013-0_15.

[232] J. Zhu, B. Chu, H. Lipford, and T. Thomas. "Mitigating Access Control Vulnerabilities through Interactive Static Analysis". In: *ACM Symposium on Access Control Models and Technologies*. SACMAT. ACM, 2015, pp. 199–209. DOI: 10.1145/2752952.2752976.

[233] J. Zhu, H. R. Lipford, and B. Chu. "Interactive Support for Secure Programming Education". In: *ACM Technical Symposium on Computer Science Education*. SIGCSE. ACM, 2013, pp. 687–692. DOI: 10.1145/2445196.2445396.

[234] J. Zhu, J. Xie, H. R. Lipford, and B. Chu. “Supporting secure programming in web applications through interactive static analysis”. In: *Journal of Advanced Research* 5.4 (2014), pp. 449–462. ISSN: 2090-1232. DOI: 10.1016/j.jare.2013.11.006.

[235] M. F. Zibran, F. Z. Eishita, and C. K. Roy. “Useful, But Usable? Factors Affecting the Usability of APIs”. In: *Working Conference on Reverse Engineering*. WCRE. IEEE, 2011. DOI: 10.1109/WCRE.2011.26.

[236] M. E. Zurko and R. T. Simon. “User-Centered Security”. In: *New Security Paradigms Workshop*. NSPW. ACM, Sept. 1996. DOI: 10.1145/304851.304859.

A. Detailed API Usability Model Attributes (Study 1)

(1) [Factory pattern|Use] → − [Create object|Duration] [63]

(2) [Package design|High number of classes (>15)] → − [Find class|Duration] [170]

(3) [Package design|Subpackages in case of many classes] → + [Find class|Duration] [170]

(4) [Annotations|Configuration of classes] → + [Create code|Duration] [172]

(5) [File-based|Configuration without recompilation] → + [Create code|Duration] [172]

(6) [Fluent interfaces|Simple configurations (no long concatenations to configure constructors, methods, parameters)] → + [Create code|Duration] [172]

(7) [Class names|Common, recognizable, early in the alphabet] → + [Select implementable start class|Duration] [187]

(8) [Class names|Consistent, descriptive, clear, generic] → + [Understanding an API| Duration] [151]

(9) [Create-Set-Call Pattern|Applicable] → + [Explore an API|Duration] [184]

(10) [Method placement in helper objects|Existence] → − [Find method|Duration] [187]

(11) [Number of methods|High number of identical prefixes] → − [Find method|Duration] [170]

(12) [Number of methods|High number of identical prefixes] → − [Find method|Duration] [171]

(13) [Number of methods|In one class] → ∘ [Find method|Duration] [170]

(14) [Method names|Prefix corresponds to the expectations of the developer] → + [Find method|Duration] [171]

(15) [Method names|Carefully chosen and distinctive] → + [Find method|Duration] [170]

(16) [Method names|Consistent, descriptive, clear, generic] → + [Understand an API| Duration] [151]

(17) [Overloads|Number per method > 3] → − [Find (overloaded) method|Optimal selection] [171]

(18) [Parameter design|References to required classes] → + [Find classes|Duration] [187]

(19) [Parameter design|References to required classes] → + [Find classes|Duration] [151]

(20) [Parameter design|References to required classes] → + [Find classes|Duration] [59]

(21) [Parameter design|High number of method parameters] → − [Integrate method|Duration] [170]

(22) [Exception|For communicating the result of an executed method] → − [Understand an API|Duration] [59]

(23) [Default constructor|Available] → + [Create code|Duration] [184]

(24) [Optional constructors|Available] → ∘ [Create object|Duration] [184]

(25) [Constructors|Required parameters] → − [Create code|Duration] [184]

(26) [Constructors|Required parameters] → ∘ [Create code|Duration] [151]

(27) [Constructors|Required parameters] → ∘ [Code debuggen|Duration] [184]

(28) [Object creation|Static methods] → − [Object creation|Duration] [170]

(29) [Access rules|Needed elements like classes, methods, variables etc. are public and can be modified] → + [Create code|Duration] [235]

(30) [Documentation|Correct, complete and consistent with the implementation] → + [Learn an API|Duration] [235]

(31) [Documentation|Suitably structured and complete] → + [Learn an API|Duration] [164]

(32) [Code examples|Integration] → ± [Create code|Duration] [34]

(33) [Examples|Existence] → + [Learn an API|Duration] [124]

(34) [Examples|Presentation of complex use cases] → ± [Learn an API|Duration] [164]

(35) [Comments|Dynamic processing] → + [Find examples|Duration] [124]

(36) [Documentation|Comments in source code correct, unambiguous, complete] → + [Learn an API|Duration] [151]

(37) [Examples from online sources|Find, select and adjust] → ± [Create code|Duration] [59]

(38) [Documentation|Explain basic considerations] → + [Choose between alternatives|Program efficiency] [164]

(39) [Programming style|Type] → ± [Explore an API|Manner] [49]

(40) [Programming style|Type] → ± [Learn an API|Manner] [49]

(41) [Mental models|Compliance with actual API model] → + [Learn an API|Duration] [186]

(42) [Conventions|Compliance] → + [Learn an API|Duration] [124]

(43) [IDE|Design of the API considering characteristics (auto-completion) of the target IDE] → + [Understand an API|Duration] [171]

(44) [Internet resources|Use] → ± [Learn an API|Duration] [34]

(45) [Auto-complete|Display of methods and overloads as a list in two separate windows] → + [Find (overloaded) method|optimal selection] [171]

(46) [Auto-complete|Smart algorithms] → + [Find class, method etc.|Duration] [37]

(47) [Auto-complete|Instantiation of objects] → + [Create code|Duration] [131]

(48) [Auto-complete|Active GUI-based code completion] → + [Create code|Duration] [142]

(49) [Online search engines|Translation of API terminology] → + [Find examples|Duration] [185]

(50) [Online search engines|Translation of API terminology] → + [Find documentation and examples|Duration] [34]

(51) [Online search engines|Use] → ± [Find examples|Duration] [185]

(52) [Recommendations|Structure- and synonym-based analysis] → + [Explore an API|Duration] [58]

(53) [Recommendations|Consideration of the local context of method parameters] → + [Select parameters|Duration] [16]

(54) [Recommendations|Usage pattern of API Components] → + [Create code|fewer bugs] [231]

(55) [Development guidelines|Existence] → + [Maintain API|Frequency] [64]

B. Online Questionnaire (Study 2)

Welcome to our survey! We are researchers from the Cologne University of Applied Sciences (Germany) and our goal is to improve the usability of security APIs. The purpose of this survey is to understand what developers do require to integrate security functionalities into their software. The survey consists of 19 (+3 conditional) questions which will be answered in approximately 15 minutes.

A note on privacy: This survey is anonymous. The record of your survey responses does not contain any identifying information about you, unless a specific survey question explicitly asked for it. If you have any question, remark or comment about this survey, please contact Peter Gorski (email address).

1. What country do you live in? (dropdown list)

2. What is your current occupation? (multiple choice):
   ☐ Freelance developer ☐ Industrial developer
   ☐ Industrial researcher ☐ Academic researcher
   ☐ Graduate student ☐ Undergraduate student
   ☐ Prefer not to answer ☐ Other (*free response*)

3. How many years of development experience do you have? (radio list):
   - More than 10 years
   - 5-10 years
   - 2-5 years
   - 1-2 years
   - Less than 1 year
   - Prefer not to answer

4. What type(s) of software do you develop? (multiple choice):
   ☐ Web Applications ☐ Mobile Applications
   ☐ Desktop Applications ☐ Embedded Applications
   ☐ Enterprise Applications ☐ Other (*free response*)

5. What programming language(s) do you use primarily? (multiple choice):
   ☐ C ☐ Java
   ☐ Python ☐ C++
   ☐ R ☐ C#
   ☐ PHP ☐ JavaScript
   ☐ Ruby ☐ Go
   ☐ Other (*free response*)

6. Have IT security topics been part of your educational background?
   ☐ Yes ☐ No

7. [If Yes for question 6] What aspects of IT security have been part of your education?
   free response

8. How often do you need to integrate security mechanisms in your code? (radio list):
   - Frequently - I use security mechanisms in more than 66% of my development tasks
   - Occasionally - I use security mechanisms in more than 33% but less than 66% of my development tasks
   - Rarely - I use security mechanisms in less than 33% of my development tasks
   - Never - I never use security mechanisms in any of my development tasks

9. Who is, in your opinion, responsible for integrating security mechanisms in software systems?
   free response

10. Which security mechanisms did you implement in your code?
    free response

11. What are the most common security-related functions you use in your code? Don't rank functions you never used. Use "Other (1)" to "Other (5)" for missing functions as needed. (ranking):
    - API keys
    - Authorization / Authentication
    - Encryption / Decryption
    - Fraud prevention
    - Input validation
    - Key management and generation
    - Secure connections and communications
    - Signature generation and validation
    - Store and authenticate user names and passwords
    - Transfer files securely
    - Other (1) (free text)
    - Other (2) (free text)
    - Other (3) (free text)
    - Other (4) (free text)
    - Other (5) (free text)

12. Assume your current task is to integrate a security mechanism in your code (e.g., secure communication, user login or input validation). Please list the first work steps you would do. (multiple free text):
    (1) 1. step (*free response*)
    (2) 2. step (*free response*)
    (3) 3. step (*free response*)

(4) 4. step (*free response*)

(5) 5. step (*free response*)

13. What is, in your opinion, the difference between security-related programming tasks and non-security related tasks?
free response

14. Did you ever had problems implementing security mechanisms or integrating them into your code?
☐ Yes ☐ No

15. [If Yes for question 14] What were those problems?
free response

16. [If Yes for question 14] What do you consider as the root cause for these problems?
free response

17. Which level of abstraction should a security API offer to meet your development needs? (radio list):
    - High - It should be a high level API. I am not interested in implementation details. I just want the security to work.
    - Medium - An API should offer me low as well as high levels of implementation details and opportunities, depending on my actual programming task.
    - Low - It should be a low level API. I want to determine implementation details myself.
    - Other (*free response*)

18. Is there any API, tool or information resource you would recommend a colleague or friend who struggles with implementing a security mechanism?
free response

19. What do you think is more appropriate for your needs? Security APIs can be grouped into cryptographic APIs (e.g., encryption algorithms, hash functions

and digital signatures) and security controls APIs (e.g., security protocols and mechanisms for authorization or authentication). (radio list):

- Cryptographic APIs
- Security controls APIs
- Both
- None

20. Do you have further comments, thoughts or suggestions?
    free response

Thank you for taking part in the survey! If you like to receive the study results and/or to participate in future studies then please provide your email address. This data will be stored seperately from the survey and will not be published.

21. Check any that apply (multiple choice):
    ☐ I would like to receive the study results
    ☐ You may consider me for future studies

22. Please enter your email address:
    free response

C. Online Study Questionnaire (Study 3)

Task-specific questions: Asked about each task

1. Please rate your agreement to the following statements:

   a) I think I solved this task correctly.
   ☐ strongly disagree ☐ disagree ☐ neutral
   ☐ agree ☐ strongly agree ☐ I don't know

   b) I think I solved this task securely.
   ☐ strongly disagree ☐ disagree ☐ neutral
   ☐ agree ☐ strongly agree ☐ I don't know

2. Did you use the PyCrypto API documentation to solve this task?
   ☐ Yes ☐ No

3. [If Yes for question 2] Please rate your agreement to the following statement.

   The documentation was helpful in solving this task.
   ☐ strongly disagree ☐ disagree ☐ neutral
   ☐ agree ☐ strongly agree ☐ I don't know

4. Which parts of the documentation did you use?
   document structure multiple choice

5. Did you see any security warnings while working on this task?
   ☐ Yes ☐ No

6. [If Yes for question 5] Please rate your agreement to the following statement.

   The security warnings displayed in the console helped to solve this task.
   ☐ strongly disagree ☐ disagree ☐ neutral
   ☐ agree ☐ strongly agree ☐ I don't know

7. Please explain why the security warnings were helpful or rather unhelpful.
   free response

General questions about previous experience

8. Have you used the PyCrypto library before? For example, maybe you worked on a project that used PyCrypto, but someone else wrote that portion of the code.
   ☐ I have used PyCrypto before
   ☐ I have seen PyCrypto used but have not used it myself
   ☐ No, neither ☐ I don't know

9. Have you used or seen code for tasks similar to the tasks given in the study before? For example, maybe you worked on a project that included a similar task, but someone else wrote that portion of the code.
   ☐ I have written similar code
   ☐ I have seen similar code but have not written it myself
   ☐ No, neither ☐ I don't know

Usability perception

10. Please rate your agreement to the following questions on a scale from "strongly agre" to "strongly disagree."

    ☐ strongly disagree ☐ disagree ☐ neutral
    ☐ agree ☐ strongly agree ☐ does not apply

    - a) I had to understand how most of the assigned library works in order to complete the tasks.
    - b) It would be easy and require only small changes to change parameters or configuration later without breaking my code.
    - c) After doing these tasks, I think I have a good understanding of the assigned library overall.
    - d) I only had to read a little of the documentation for the assigned library to understand the concepts that I needed for these tasks.

- e) The names of classes and methods in the assigned library corresponded well to the functions they provided.
- f) It was straightforward and easy to implement the given tasks using the assigned library.
- g) When I accessed the assigned library documentation, it was easy to find useful help.
- h) In the documentation, I found helpful explanations
- i) In the documentation, I found helpful code examples.
- j) When I made a mistake, I got a meaningful error message/exception.
- k) Using the information from the error message/exception, it was easy to fix my mistake.

Message design assessment

11. Please rate your agreement to the following statements concerning this console warning:
    [Example security advice figure (cf. Figure 5.4, p. 72)]
    How helpful would you rate...

    ☐ not helpful at all ☐ somewhat unhelpful ☐ neutral
    ☐ somewhat helpful ☐ very helpful ☐ I don't know

    - ...the risk explanation?
    - ...the recommendation for secure action?
    - ...the given code example?
    - ...the described option for insecure action?
    - ...the given background information?
    - ...the structure of this security advice?
    - ...the amount of information in the message?
    - ...the appearance of this kind of messages when using the PyCrypto Library?

12. What aspects of the warning could be improved, in your opinion?
free response

Development Environment

13. Please tell us some details about your usual Python software development tool chain.

a) Which console do you use?
free response

b) Which text editor do you use?
free response

c) What IDE do you use?
free response

d) Do you use other tools for software development?
free response

Demographic Questions

14. What type(s) of software do you develop?
☐ Web Applications ☐ Desktop Applications
☐ Mobile Applications ☐ Enterprise Applications
☐ Embedded Applications ☐ Other: *free response*

15. How many years of development experience do you have?
numeric response 0-100 ☐ Prefer not to answer

16. How many years have you been programming in Python?
numeric response 0-100 ☐ Prefer not to answer

17. What is your current occupation?
☐ Freelance developer ☐ Industrial developer
☐ Industrial researcher ☐ Academic researcher
☐ Graduate student ☐ Undergraduate student
☐ Prefer not to answer ☐ Other: *free response*

18. What is your gender?
    ☐ Female ☐ Male
    ☐ Prefer not to answer ☐ Other: *free response*

19. What country do you live in?
    Please choose... (Dropdown)

20. How old are you?
    numeric response 0-100 ☐ Prefer not to answer

D. Laboratory Study Interview Questions[1] (Study 5)

1. Please rate your agreement with the following statements:

   a) I solved the task correctly.
   ☐ strongly disagree ☐ disagree ☐ neutral
   ☐ agree ☐ strongly agree ☐ I don't know

   b) I solved the task securely.
   ☐ strongly disagree ☐ disagree ☐ neutral
   ☐ agree ☐ strongly agree ☐ I don't know

2. Please explain your assessment of the statement "I have implemented a secure solution."
   free response

3. Did you use Content Security Policies to solve the task?
   ☐ Yes ☐ No ☐ I don't know

4. [If Yes for question 3] Why or for what purpose did you use Content Security Policies to solve the task?
   free response

5. Please describe, according to your understanding, the sense and purpose of a Content Security Policy.
   free response

6. Which operating systems are you familiar with?
   ☐ Mac ☐ Windows ☐ Linux ☐ Other

[1] The questionnaire was translated from German into English.

7. Which operating systems do you prefer to work with?
   ☐ Mac ☐ Windows ☐ Linux ☐ Other

8. Have you ever used the Play Framework before the study?
   ☐ Yes ☐ No ☐ No answer

9. [If Yes for question 8] What have you ever used the Play Framework for?
   free response

10. Have you ever used IntelliJ IDEA before the study?
    ☐ Yes ☐ No ☐ No answer

11. [If Yes for question 10] What have you used IntelliJ IDEA for before?
    free response

12. Have you ever used the developer tools of a browser before the study?
    ☐ Yes ☐ No ☐ No answer

13. [If Yes for question 12] What have you done with the Developer Tools?
    free response

14. Have you ever developed a web application before the study?
    ☐ Yes ☐ No ☐ No answer

15. [If Yes for question 14] What kind of web applications have you ever developed?
    free response

16. What kind of security measures have you ever considered when developing software?
    free response

17. Have you ever used a Content Security Policy before the study?
    ☐ Yes ☐ No

18. [If Yes for question 17] What have you used a Content Security Policy for before this study?
    free response

19. Have you ever developed other types of software?
    ☐ Web Applications ☐ Mobile Applications
    ☐ Desktop Applications ☐ Embedded Applications
    ☐ Enterprise Applications ☐ Other

20. How many years have you been developing software?
    numeric free response

21. How many years have you been programming in Java?
    numeric free response

22. How many years have you been programming in JavaScript?
    numeric free response

23. Have you ever been paid for programming software?
    ☐ Yes ☐ No

24. How old are you?
    numeric free response

25. In which study program are you matriculated?
    ☐ Bachelor Media Technology
    ☐ Master Media Technology
    ☐ Other

26. In which semester are you studying?
    numeric free response

27. To which gender do you feel you belong?
    ☐ Female ☐ Male ☐ Other

E. Laboratory Study Interview Questions[1] (Study 6)

1. Please rate your agreement with the following statements:

   a) I solved the tasks correctly.
   ☐ strongly disagree ☐ disagree ☐ neutral
   ☐ agree ☐ strongly agree ☐ I don't know

   b) When solving the tasks, I also considered the security of the data.
   ☐ strongly disagree ☐ disagree ☐ neutral
   ☐ agree ☐ strongly agree ☐ I don't know

2. Please explain your assessment of the statement "When solving the tasks, I also considered the security of the data."
   free response

3. Did you use Content Security Policies to solve the task?
   ☐ Yes ☐ No ☐ No answer

4. [If Yes for question 3] How did you know that you need to configure a Content Security Policy?
   free response

5. [If No for question 3] Please explain why you have not used Content Security Policies.
   free response

6. Please describe, according to your understanding, the sense and purpose of a Content Security Policy.
   free response

[1] The questionnaire was translated from German into English.

7. What information does the Google Maps API documentation provide about Content Security Policies?
   free response

8. What did you find most difficult about integrating the Content Security Policy?
   free response

9. Which operating systems are you familiar with?
   ☐ Mac ☐ Windows ☐ Linux ☐ Other

10. Which operating systems do you prefer to work with?
    ☐ Mac ☐ Windows ☐ Linux ☐ Other

11. Have you ever implemented a Go web application before the study?
    ☐ Yes ☐ No ☐ No answer

12. [If Yes for question 11] What kind of web application have you implemented in Go?
    free response

13. Have you ever developed a web application before the study - except with Go?
    ☐ Yes ☐ No ☐ No answer

14. [If Yes for question 13] What kind of web applications - except with Go - have you ever developed?
    free response

15. Have you ever used Visual Studio code before the study?
    ☐ Yes ☐ No ☐ No answer

16. [If Yes for question 15] What have you used Visual Studio code for before?
    free response

17. Have you ever used the developer tools of a browser before the study?
    ☐ Yes ☐ No ☐ No answer

18. [If Yes for question 17] What have you done with the Developer Tools?
    free response

19. What kind of security measures have you ever considered when developing software?
    free response

20. Have you ever used a Content Security Policy before the study?
    ☐ Yes ☐ No ☐ No answer

21. [If Yes for question 20] In what context have you ever used a Content Security Policy before the study?
    free response

22. Have you ever developed other types of software?
    ☐ Mobile Applications ☐ Desktop Applications
    ☐ Embedded Applications ☐ Enterprise Applications
    ☐ Other

23. How many years have you been developing software?
    numeric free response

24. How many years have you been programming in Go?
    numeric free response

25. How many years have you been programming in JavaScript?
    numeric free response

26. Have you ever been paid for programming software?
    ☐ Yes ☐ No ☐ No answer

27. How old are you?
    numeric free response

28. In which study program are you matriculated?
    ☐ Bachelor Media Technology
    ☐ Bachelor Computer Science and Engineering
    ☐ Other

29. In which semester are you studying?
    numeric free response

30. To which gender do you feel you belong?
    ☐ Female ☐ Male ☐ Other