Listen to Developers! A Participatory Design Study on Security Warnings for Cryptographic APIs

Peter Leo Gorski, Yasemin Acar*, Luigi Lo Iacono, Sascha Fahl*
TH Köln/University of Applied Sciences, *Leibniz University Hannover

ABSTRACT
The positive effect of security information communicated to developers through API warnings has been established. However, current prototypical designs are based on security warnings for end-users. To improve security feedback for developers, we conducted a participatory design study with 25 professional software developers in focus groups.

We identify which security information is considered helpful in avoiding insecure cryptographic API use during development. Concerning console messages, participants suggested five core elements, namely message classification, title message, code location, link to detailed external resources, and color. Design guidelines for end-user warnings are only partially suitable in this context.

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 and should be adjustable by software developers across development tools, considering the work context and developer needs.

Author Keywords
security warning design, focus groups, participatory design, cryptographic APIs, developer console, software development

CCS Concepts
•**Security and privacy** → **Software security engineering; Usability in security and privacy;**

INTRODUCTION
Over 26 million people worldwide are developing software, including full-time, part-time, and nonprofessional developers [15]. These developers satisfy our current need for software across a variety of application platforms and have the responsibility of implementing security and privacy for billions of users. Legal requirements like the General Data Protection Regulation [38] require personal data to be handled carefully, which often involves the use of cryptography. Instead of writing cryptographic code from scratch, APIs (Application Programming Interfaces) are therefore an essential tool for software developers to bring cryptography into their products.

CHI'20, April 25–30, 2020, Honolulu, HI, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-6708-0/20/04...$15.00

DOI: https://doi.org/10.1145/3313831.3376142

Because available cryptographic APIs (CAPIs) and tools are difficult to use [31, 1] even for experienced developers [34], common requirements such as securely storing passwords, encrypting files or establishing encrypted network connections can be challenging and error-prone [32].

Many CAPIs offer insecure features and function calls for compatibility reasons, e.g. to support outdated cryptographic algorithms or parameters. As per the status quo, programmers are not informed or warned when using such API functionalities. Warnings that point the developer to an insecure CAPI use or potential risks in program code play an important role in secure software development research [18, 3]. However, how to design good security warning messages for developers is yet not well understood [18].

Previous work discussed different approaches to communicate security feedback to developers, including security warnings in the IDE (Integrated Development Environment) triggered by plugins [33], static analysis tools [28], or security feedback implemented as API warnings in the developer console [17]. However, the designs of previous approaches are based on research for end-user warnings and security experts' opinions without considering the perspective of actual software developers.

Because the developer console is a central point for information in software development (cf. Section RQ2 - Warning Context), the purpose of this work is to revise and improve previous security API warning designs [17] for CAPIs. We aim to improve cryptographic advice in terms of usability, and to learn more about the dimensions, aspects, advantages, and disadvantages of the API warning approach from a software developer's perspective. Therefore, we followed a participatory design approach and conducted four focus groups (FG) with a total of 25 developers. The goal of our FGs was to answer the following research questions:

RQ1 What kind of design do software developers find helpful for cryptography warning (CW) messages in the console? Based on a participatory design approach, we examine designs for CWs in the console based on the experiences and preferences of the target group. We follow a three-step process. First, FGs compile information they perceive as helpful when being warned of insecure CAPI use. Second, participants visualize and draw warnings they perceive as helpful. In a third step, the respondents compare their approach with an existing design approach [17].

RQ2 Do software developers find console security warnings helpful in development processes and software environments? We put the design approach 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.

RQ3 From the developer's point of view, is there a difference between security warnings and other types of console warnings? We would also like to know how security warnings differ from other warnings in the console to gain further insight into design requirements.

RQ4 Would developers implement their own warning design into a CAPI? We identify common hurdles when implementing CWs in an API.

With respect to console messages, our participants suggested five core elements. These are message classification, title message, code location, link to detailed external resources and coloring. Further research should confirm these results to formulate validated design guidelines for API warnings. 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 and should be adjustable by software developers across development tools, considering the application context and developer needs.

In the FGs, participants were asked to design and comment on existing "security warnings" for CAPIs displayed in developer consoles. A console provides a command-line interface receiving commands in text form and giving feedback in text form as well. However, at the end of the FGs, the views were spread across a wider development environment context. In a broader tool-wide sense, we use the general term "security feedback" throughout the paper. APIs are also called libraries because they provide a collection of programming blocks. We consistently use the term API.

We make the following contributions: (1) We conduct four FGs with the goal to improve CAPI feedback for developers, (2) We find that developers generally find security feedback important and useful, (3) We find that security feedback needs to be adapted to its context, (4) Our 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.

RELATED WORK

The API design space provides an overview of the capabilities available to an API producer to design an API and make it usable [37, 16]. Usability is an important design characteristic for an API to support consumers in using features correctly and to mitigate insecure use, which otherwise likely results in program errors and security issues [18, 30]. Therefore, dealing with APIs within the security API domain [20] requires special attention to the communication between API designers and consumers.

Research has identified specific challenges developers face when working with CAPIs. Users of CAPIs are often required to have a high level of cryptography expertise [18]. Achieving secure software requires the right choice of algorithms, the correct sequence of method calls, and confident handling of parameters [31]. Overly complicated designs for the average developer, missing API features addressing typical use cases, and immature documentations lead to error-prone APIs [1]. These manifest in programming errors caused by cognitive breakdowns in implementation activities [23, 25]. Also, developers who try to solve this situation by consulting actionable advice from community platforms will likely copy insecure code examples [2, 8]. To help users of security APIs, multiple research approaches are focused inside and outside of the API design space. We briefly discuss related work in IDEs, static code analysis tools, message design for compiler errors, and API warnings. Developers also need additional support in dealing with identified security issues [35]. Therefore, research is being carried out into how developer centered approaches can improve debugging interfaces [29, 24, 26]. This topic, however, lies outside the focus of this work.

The graphical capabilities of an IDE can be used to draw the developer's attention to security vulnerabilities. ASIDE offers interactive code annotation, including highlighted source code and icons as security indicators in a text editor [39, 41]. This approach could also be improved by FixDroid, where automated code fixes give the users implicit actionable support at a click [33]. The IDE extension CogniCrypt takes the burden of writing code off the developer. Needs can be defined in a wizard, and the corresponding source code is generated for the user [28]. Such IDE tools are not located in the API design space and thus elude the direct influence of an API designer.

Another active research field to mitigate insecure cryptography use are static code analysis tools. Algorithms analyze written source code and trigger warnings. These tools are often supplied as plug-ins for IDEs. High false positives rates [22] and bad warning message design [7, 14] are significant problems when working with warnings from program analyzers. They also need to be thoroughly configured to improve the number of relevant warnings for a given user context [5]. Static code analysis is not located in the API design space either, as an API design has no direct effect on analysis results.

Previous research has also focused on the improvement of compiler errors that are shown in the developer console during compile time. A compiler is used in compiled programming languages to translate human-readable source code into machine code. CAPIs are not limited to compiled programming languages and are also not a part of compiler software. In this context, the feedback of an API is displayed during runtime only. Compiler and API messages share a target medium with the developer console. However, differences or similarities in design have not yet been examined. Compiler messages refer strongly to the syntax of a programming language, whereas API messages refer to specific problems or features that an API designer can communicate to the API user at runtime.

To support compiler developers in designing new error messages or evaluate existing, Barik et. al [9] have formulated three principles. We can confirm that the principle "Allow developers the autonomy to elaborate arguments" is also relevant in the context of CAPI warnings (cf. Section RQ2 - Warning Context). Traver [40] theoretically discusses eleven abstract principles for compiler error message design, like "clarity and brevity", "context-insensitivity" and "locality", which are also confirmed in the context of API warnings by the results of this study (cf. Section Results). Barik et. al also found evidence that participants do read compiler error messages by conducting an eye-tracking study [10]. Reading such messages is comparably complex to reading source code. Becker [12] found evidence that providing explanations in complement to compiler errors by an extended editor can help developers make fewer errors. Compiler messages are not located in the API design space, either.

For direct communication between API designers and API users at program runtime via the developer console, Stylos & Myers [37] only mention exceptions in their model. Exceptions handle unusual conditions that can lead to program errors. However, they should not be used to handle ordinary control flow [13]. Most recently Gorski et. al proposed an initial developer console warning for CAPIs and confirmed its effectiveness in an online experiment [17]. While they were able to demonstrate improved code security with their CAPI console warning, the researchers applied heuristics that are based on end-user research and have not yet been proven to be optimal for the specific context of software development and console environments [11]. We examine the results of this study and try to improve the design of CAPI warnings in the console. To the best of our knowledge, there are no guidelines or recommendations for developer security warnings in their working environment, including the developer console. There is neither an understanding of overlaps nor differences between developer and end-user environments.

We are not aware of research on the usability of other types of API warnings like updates, deprecation, or linting warnings. Unlike other API warnings, CAPI 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. Further research should, therefore, be done on CAPI warnings to avoid unintentional or accidental insecure use by giving appropriate information to API users.

METHODOLOGY

We conducted four FGs with experienced software developers to gather a wide range of opinions, perceptions and ideas emerging from group discussion [27]. FGs can also help to identify and address aspects that may not be mentioned in individual interviews. There is no formal IRB process at our university. Our study design complies with the requirements of the European GDPR [38]; participants agreed to anonymzed use of their study data by reading and signing a consent form prior to the study. FGs were audio-recorded and transcribed by the main author. Anonymized transcripts as well as pictures taken of materials developed in the study are kept in secure storage; audio files and PII were deleted.

Study Protocol

We developed the study protocol according to FG guidelines [27]. The protocol was reviewed by subject matter experts (two authors, two non-authors) and revised according to their feedback. Our FG 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've 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.

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 CAPI issuing a security-related warning message in the console to give the developer a recommendation for action. Please list information and/or 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 in the console looks like? 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 4) 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 CAPI 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?

We tested the study protocol with the first FG and made slight changes to the protocol based on the participants' feedback: We revised one slide 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 main author led all FGs and moderated group discussions.

Recruitment
We recruited developers with professional programming experience in the Cologne area. To include a wide variety of experiences and opinions, we required professional programming experience but did not restrict our recruitment to a specific developer population. We invited potential participants via mailing lists, e.g., of PyCologne, a local Python developer group, and we emailed personal contacts at software development companies. We also recruited participants based on recommendations by FG participants after their FG. The invitation email included a link to the research project website and a brief qualification questionnaire. In the qualification questionnaire, we collected information about prospective candidates' work and programming experience. Overall, we ended up recruiting 25 participants for four FGs. We conducted four FGs since we reached saturation after the third FG. The fourth group did not develop or discuss new aspects.

Participants
We recruited 25 participants from the Cologne metro area. We organized these into four FGs 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). They reported to have been programming for 10 years on average (sd:6) and to have been working professionally as software developers for 6 years on average (sd:6) (cf. Table 1).

Data Analysis
FGs lasted about two and a half hours. After each FG, 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 [6].

The codebook was created by multiple researchers, reflecting different experiences and backgrounds which include formal education in media technology, mathematics and computer science; all authors have research experience in security, privacy and human factors research. We used the coding for grouping the FGs' contents. Beginning with open coding, the first author analyzed and coded each contribution to the discussions. In this inductive process, he considered (1) exhaustiveness to ensure codings will reflect all main topics when assigned to higher-level semantic groups. He also adhered to (2) mutual exclusiveness of semantic groups to keep codes unambiguous. Paying attention to completeness, a total number of 210 subcodes of 21 higher level code groups were operationalized following a group discussion between the authors. Two additional coders with computer science background and minor HCI knowledge were introduced to the codebook in a one hour briefing, following which they independently coded FG number one (about 25% of all transcripts, exceeding recommendations for double-coding in qualitative research [19]) applying the codebook. The intention was to use inter-rater agreement scores to identify ambiguous codes or codings that were forgotten by the main coder. While the granular codebook was difficult for the additional coders to apply, mainly because they were inexperienced in the topic and in 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, we matched research questions to relevant code groups (cf. Table 2) and identified the FGs' major results.

To be able to give weight to our assessment of helpful content of CWs, we extracted quantitative data from the mostly qualitative FGs in the following ways: We asked the participants to rate their suggestions on a 5-point Likert-scale. We counted the individual contents of one drawing per participant. We also asked the participants for a clear yes or no statement as to whether they would implement CWs in their APIs.

Limitations
This sample is not representative for all developers: Our participants came from Germany and notably and to our frustration, only men participated in our study (cf. Table 1). We attempted to recruit women by directly emailing female developers, which, with more than 92% developer roles in Germany being filled by men [36], proved to be hard. Three women signed up for the FGs, 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 our participants' opinions and experiences. The group compositions may also have influenced participants' statements. All participants of our second group work in different areas and project teams in the same software company. This is also reflected in their self reported programming experiences (cf. Table 1). We were able to represent a broad spectrum of experiences with programming languages and software types in our sample, which is a prerequisite for answering our research questions.

Symmetric encryption is one of the most common tasks with a CAPI [31], therefore we used RC4 as an example in the FGs. 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 CAPIs. Further studies are required to compare and evaluate different designs and identified aspects of this work to gradually develop a mature guideline.

RESULTS
In the following, we report the results of our FGs. Each section refers to one of the four research questions (cf. Introduction).

RQ1 - Participatory Warning Message Design
The participatory design approach is based on three different study artifacts. In a first step, we asked our participants to collect helpful information on a pinboard, rating each suggestion individually on a five point Likert scale (cf. Figure 1). We then asked them to independently outline an ideal message (cf. Figure 2 and Figure 3) and to evaluate a proposed warning design by Gorski et. al [17] (cf. Figure 4).

Rating of Helpful Information
All participants had concrete ideas about helpful CW message content. However, most of the proposals were discussed

Table 1. Demographic data of the focus group participants.

	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 researchers	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●	-	-	-	-	-	-
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 2. Code group descriptions.

Code group	RQ	Description
Developer Console Activities	1	Things you do with a console or activities you use the console for.
Developer Console Properties	1	Properties of a development console.
Developer Console Types	1	Different types or consoles, even in terms of differences in operating systems.
Documentation	1	Statements about the documentation of APIs and program code.
Security Feedback	3	Particularities of warnings in a security context and differences to other warnings.
Software Development Tools	2	Tools for software development that are addressed.
Software Environment	2	Statements about an environment in which software is developed or executed.
Warning Content Negative	1	Warning content that is undesired, unimportant or unhelpful - contra arguments.
Warning Content Positive	1	Warning content that is desired, important or helpful - pro arguments.
Warning Design in General	1	General statements about console warning design not relating to content or properties.
Warning Display Location	2	Where, in addition to the console, a warning is also displayed.
Warning Implementation	4	Aspects of implementing console warnings in APIs.
Warning People Involved	2	People involved when a warning is displayed.
Warning Processing Practices	2	About working with a warning or processing warnings.
Warning Properties Negative	1	Negative properties of warnings in the console that are not represented by warning content.
Warning Properties Positive	1	Positive properties of warnings in the console that are not represented by warning content.
Warning Purpose	3	The purpose of a warning message. What is to be achieved by the warning?
Warning Reason	3	The reason of a warning message. Why is a warning given?
Warning Target Group	1	Persons addressed by a warning.
Warnings Reactions	1	Developers' reactions to warnings or how they deal with the situation.
Weak Crypto Use Case	1	Reasons why an insecure API call is used.

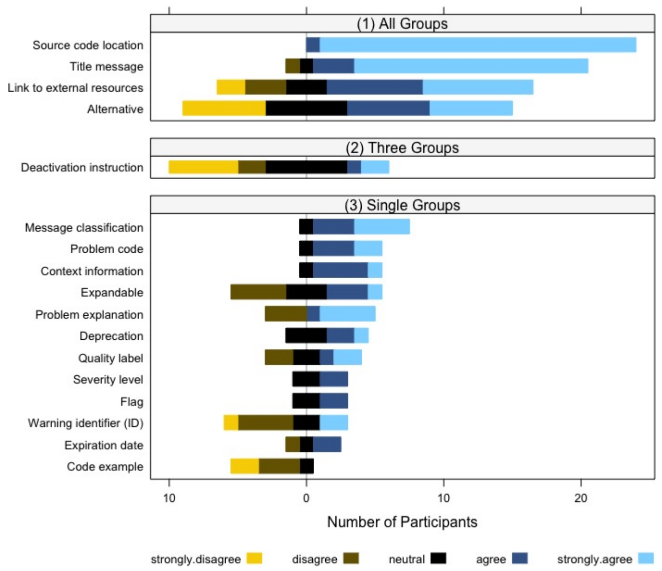


Figure 1. Participants' rating results of helpful aspects as part of a security warning for a CAPI (step six and seven of the study protocol). Grey labels indicate how many groups have listed the items.

controversially mainly because developer consoles are used for diverse use cases (cf. Section RQ2 - Warning Context) and should therefore not be overloaded with unimportant or unnecessary 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 1). Three groups discussed an option to deactivate the warning. Since these aspects were mentioned in several groups we consider them 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.

An 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 you find the place."* (G4P1)[1], 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, we report reasons 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) 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).

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

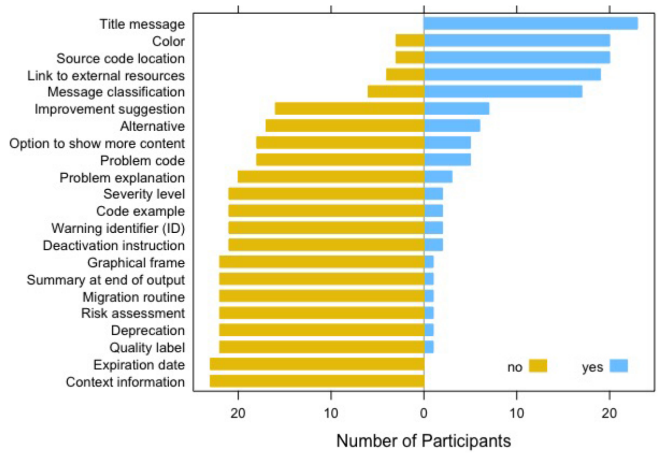


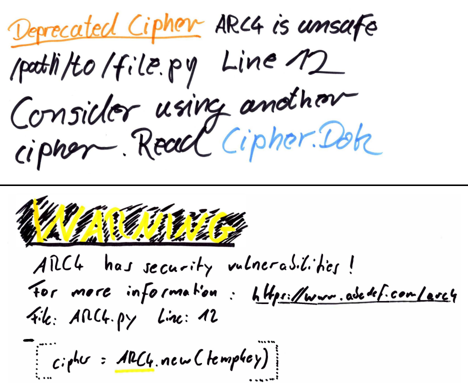
Figure 2. Number of focus group participants that used or did not use a specific item for cryptography warning content in their drawing (step eight of the study protocol).

For similar reasons, our 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 CAPIs 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, our 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 anymore"* (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*



Deprecated Cipher ARC4 is unsafe
/path/to/file.py Line 12
Consider using another cipher. Read Cipher.Dok

WARNING
ARC4 has security vulnerabilities!
For more information: https://www.abcdef.com/arc4
File: ARC4.py Line: 12
cipher = ARC4.new(tempkey)

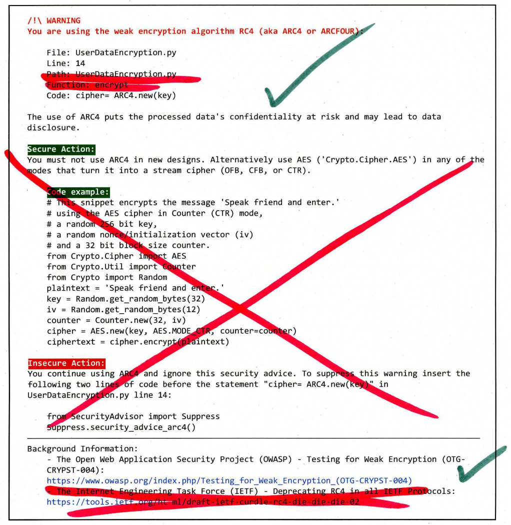
Figure 3. Participants G3P3 (top) and G2P2 (bottom) drew these mockup warnings when asked for their favorite cryptography warning design (step eight of the study protocol). They are similar in content.

it in the code so that other users would check out the project and also wouldn't see the message anymore" (G2P3).

To our surprise, up to this point of the study, only FG 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 FGs 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).

Drawings

A uniform result for a helpful CW message design in the developer console was achieved in all four groups. The individual design drafts are characterized by their short and concise form (cf. Figure 3). When evaluating the content of drawings (cf. Figure 2), 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 3). 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. In accordance with the rating result, 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 missed it in the previous task (7) 17 participants used them in their drawing. Slightly contrary to the rating results only 6 drawings contained an **alternative**.



```
/!\ WARNING
You are using the weak encryption algorithm RC4 (aka ARC4 or ARCFOUR):

    File: UserDataEncryption.py
    Line: 14
    ~~Path: UserDataEncryption.py~~
    ~~Function: encrypt~~
    Code: cipher= ARC4.new(key)

The use of ARC4 puts the processed data's confidentiality at risk and may lead to data
disclosure.

Secure Action:
You must not use ARC4 in new designs. Alternatively use AES ('Crypto.Cipher.AES') in any of the
modes that turn it into a stream cipher (OFB, CFB, or CTR).

    Code example:
    # This snippet encrypts the message 'Speak friend and enter.'
    # using the AES cipher in Counter (CTR) mode,
    # a random 256 bit key,
    # a random nonce/initialization vector (iv)
    # and a 32 bit block size counter.
    from Crypto.Cipher import AES
    from Crypto.Util import Counter
    from Crypto import Random
    plaintext = 'Speak friend and enter.'
    key = Random.get_random_bytes(32)
    iv = Random.get_random_bytes(12)
    counter = Counter.new(32, iv)
    cipher = AES.new(key, AES.MODE_CTR, counter=counter)
    ciphertext = cipher.encrypt(plaintext)

Insecure Action:
You continue using ARC4 and ignore this security advice. To suppress this warning insert the
following two lines of code before the statement "cipher= ARC4.new(key)" in
UserDataEncryption.py line 14:

    from SecurityAdvisor import Suppress
    Suppress.security_advice_arc4()
______________________________________________________________________________________________
Background Information:
    - The Open Web Application Security Project (OWASP) - Testing for Weak Encryption (OTG-
    CRYPST-004):
    https://www.owasp.org/index.php/Testing_for_Weak_Encryption_(OTG-CRYPST-004)
    ~~- The Internet Engineering Task Force (IETF) - Deprecating RC4 in all IETF Protocols:~~
    ~~https://tools.ietf.org/html/draft-ietf-curdle-rc4-die-die-die-02~~
```

Figure 4. Participant G1P6 revised the warning tested by a previous study [17] based on the focus group results and his own preferences (step nine of the study protocol).

In addition to the five most frequent design elements, participant G3P3 decided to draw a **deprecation** warning and to give an **improvement suggestion** while G2P2 integrated an **option to show more content** (cf. Figure 3). It offers to show the **problem code** after clicking on the code location.

Evaluation of a Design Proposal

Reactions to the printout (cf. Figure 4) emphasize the warning size being out of 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 check marks in Figure 4, 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).

RQ2 - Warning Context

Our developers reported that a console is an important part of their tool set. 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 means

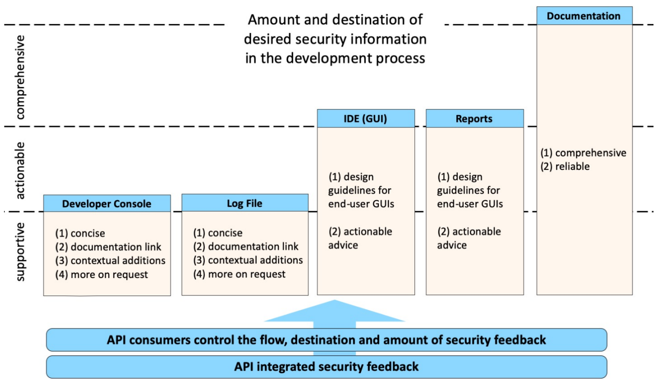


Figure 5. Amount and destination of desired security feedback in the development process.

the console is a central point for 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 FGs addressed different types of warnings like updates, deprecation, linting, and security warnings. This 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 important 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 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 to appreciate warnings: *"You can change an implementation by a warning message before a real error occurs."* (G2P7)

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. FGs generally agreed that different use cases call for different types of information. When rating potential helpful warning content (cf. Section Rating of Helpful Information) 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 to this particular aspect, we extract suggestions for what developers want in the contexts that were discussed (cf. Figure 5).

Console CWs 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 a 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 to add 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 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. But 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 afterwards 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. We conclude, 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.

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 4, 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 4], 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 further process 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 4]. 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's good warnings, there's 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, afterwards"*.

The draft in Figure 4 was also perceived positively as a report generated from downstream processing of console output: *"So in Jenkins [automation server software [21]] 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 is an integral part of an API [30]. The documentation was often mentioned by the developers when they talked about a link in the warning message. According to the preferences in our FGs, documentation should provide the most comprehensive information, so that you can find reliable information directly from the API developers when you 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). And again, the detail level of the example in Figure 4 was accepted in this context: *"If it were an inline documentation, I would find it well structured. For a warning message, it is clearly much too long."* (G3P4).

RQ3 - Cryptography Warning Specifics

In this section, we report on four aspects that are specific to console CWs; three of them deal with decisions on the code implementation level when designing a CAPI. By considering the specific context of an encryption algorithm, some participants to our surprise questioned their design approach towards the end of the FG. They raised controversial design questions which we present by discussing participants' different opinions.

Quantity means importance? While generally perceived as annoying or overloaded, an extensive warning as shown in Figure 4, 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."*. But 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 took also a clear position: *"I would switch it off immediately or change the programming language [group laughs]. It doesn't fit to my way of working at all."*. From these statements, it becomes clear that an overly extensive warning will not be accepted by some developers. This is a quality feature of an API and can be decisive for its success. For this reason, we recommend considering conciseness when implementing API warnings. Additional content should be tailored as closely 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 normal behavioral pattern in software development. This could mean that the handling of insecurely used CAPIs 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 risky assessment can be influenced in the development process. However, our 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 CWs as errors to increase awareness was not proposed by any group. While our participants did not explicitly mention this possibility, some development tools can be configured to treat warnings as errors [9]. Thus, developers can increase their awareness for warnings if a program crashes, which is also addressed in the proposal of severity levels for CWs.

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 en error. Now you might have to look again, if there are still differences in the warning classifications between serious and not so serious"*. Our FGs raised the question whether a severity level should express the importance of a CW to indicate what will happen if developers choose not to adhere. Also estimates of independent specialists could be helpful: *"Perhaps especially with algorithms 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 CWs and security warnings needs 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.

RQ4 - Attitude towards Implementation

Across FGs, 19 of 25 participants clearly stated as an API designer they would implement a security warning into a CAPI (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 a documentation clean and reasonable"* (G4P4).

DISCUSSION

The software developers in our study wish for console warnings from CAPIs as a tool to prevent insecure API use and most of them would implement this in their APIs as well. This confirms the finding of Gorski et. al that CAPI warnings are a helpful and effective tool for software developers [17].

Until now, only guidelines for end-user warnings could be applied for CWs in the developer console. Following a participatory design approach, we conclude that only two of the six design goals of the guideline by Bauer et. al [11] can directly be adopted in the concept of CAPI warnings: (1) "Be concise and accurate" and (2) "Follow a consistent layout". Our findings emphasize that developers are not end-users, and API producers should not apply guidelines for end-user warnings as a starting point for CAPI warning design. To develop appropriate recommendations, future research in API warnings now has design suggestions at hand that were developed by software developers based on their experiences and opinions.

As our results show, there is a requirement for conciseness for CAPI in the developer console and log file environment. Study participants suggested five core elements for CAPI warnings. At first glance, these can also be applied generically to the context of other API warnings. We suspect that these are common design characteristics for API warnings in general. However, our 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 our developers like a severity level, a risk assessment, a recommendation of alternative cryptographic algorithms, or a label to assess a CW's quality. At this point, a uniform design for all types of API warnings is not sufficient. Also, aspects specific to CWs 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.

Our participants expressed some reservations about the approach of API integrated crypto warnings because they saw the implementation as an increased workload. However, our results show: A good initial design for CWs 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 our findings may also serve as a starting point for API producers or tool developers.

Due to their concise form, CWs in the console have the primary purpose of alerting developers to a problem. This means further relevant information has to be given to the developers at a different locations, like log files, IDEs, post-processing reports, and API documentation, in order to support them in handling the problem. However, our results clearly show that providing all the helpful information that our participants desire (cf. Figure 1 and Figure 2) is out of scope for API producers. External 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. CAPI producers' focus should be on providing information that relates to the use of an API, like code examples and test cases for many use cases [4].

CONCLUSION

Insecure CAPI use is a critical issue for software security with far-reaching consequences for users [31, 1]. Our work contributes to a potential mitigation of the problem. While previous work showed that API redesign [1], improved documentation [2] and better tool support [33] are promising directions, we follow another line of research and improve a proposal for CAPI based console security feedback [17]. Our participatory design approach helped to identify areas of improvement for existing approaches. The FG sessions uncovered specific design aspects for CAPI warnings compared to other warnings in the developer console. Most of our participants would implement such a security warning for CAPIs. They designed these in the FGs, but went one step further: They discussed how much information they need and prefer in which phase of their development process. Our results can support API developers and researchers in further developing usable CWs.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers and shepherd for providing valuable feedback; and all participants of this study for their generous voluntary participation. This work was partially funded by the German Federal Ministry of Education and Research within the funding program "Forschung an Fachhochschulen" (contract no. 13FH016IX6).

REFERENCES

[1] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. 2017. Comparing the Usability of Cryptographic APIs. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Jose, CA, USA, 154–171. DOI:http://dx.doi.org/10.1109/SP.2017.52

[2] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. 2016a. You Get Where You're Looking for: The Impact of Information Sources on Code Security. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Jose, CA, USA, 289–305. DOI: http://dx.doi.org/10.1109/SP.2016.25

[3] Yasemin Acar, Sascha Fahl, and Michelle L. Mazurek. 2016b. You are Not Your Developer, Either: A Research Agenda for Usable Security and Privacy Research Beyond End Users. In *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, Boston, MA, USA, 3–8. DOI:http://dx.doi.org/10.1109/SecDev.2016.013

[4] Yasemin Acar, Christian Stransky, Dominik Wermke, Michelle L. Mazurek, and Sascha Fahl. 2017. Security Developer Studies with GitHub Users: Exploring a Convenience Sample. In *Thirteenth Symposium on Usable Privacy and Security (SOUPS)*. USENIX Association, Santa Clara, CA, USA, 81–95. https://www.usenix.org/conference/soups2017/technical-sessions/presentation/acar

[5] Joao Eduardo M. Araujo, Silvio Souza, and Marco Tulio Valente. 2011. Study on the relevance of the warnings reported by Java bug-finding tools. *IET Software* 5, 4 (August 2011), 366–374. DOI: http://dx.doi.org/10.1049/iet-sen.2009.0083

[6] ATLAS.ti. 2019. ATLAS.ti 8 Mac User Manual, updated for program version 8.4. [Online]. Available: https://downloads.atlasti.com/docs/manual/manual_a8_mac_en.pdf. (2019). Last accessed 8 January 2020.

[7] Dejan Baca. 2010. Identifying Security Relevant Warnings from Static Code Analysis Tools through Code Tainting. In *2010 International Conference on Availability, Reliability and Security (ARES)*. IEEE, Krakow, Poland, 386–390. DOI: http://dx.doi.org/10.1109/ARES.2010.108

[8] Wei Bai, Omer Akgul, and Michelle L. Mazurek. 2019. A Qualitative Investigation of Insecure Code Propagation from Online Forums. In *2019 IEEE Cybersecurity Development (SecDev)*. IEEE, Tysons Corner, VA, USA, 34–48. DOI: http://dx.doi.org/10.1109/SecDev.2019.00016

[9] Titus Barik, Denae Ford, Emerson Murphy-Hill, and Chris Parnin. 2018. How Should Compilers Explain Problems to Developers?. In *26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, New York, NY, USA, 633–643. DOI:http://dx.doi.org/10.1145/3236024.3236040

[10] Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin. 2017. Do Developers Read Compiler Error Messages?. In *39th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, Buenos Aires, Argentina, 575–585. DOI: http://dx.doi.org/10.1109/ICSE.2017.59

[11] Lujo Bauer, Cristian Bravo-Lillo, Lorrie Cranor, and Elli Fragkaki. 2013. *Warning Design Guidelines*. Technical Report CMU-CyLab-13-002. CyLab, Carnegie Mellon University. http://www.cylab.cmu.edu/research/techreports/2013/tr_cylab13002.html

[12] Brett A. Becker. 2016. An Effective Approach to Enhancing Compiler Error Messages. In *47th ACM Technical Symposium on Computing Science Education (SIGCSE)*. ACM, Memphis, Tennessee, USA, 126–131. DOI:http://dx.doi.org/10.1145/2839509.2844584

[13] Joshua Bloch. 2008. *Effective Java* (second ed.). Addison-Wesley, Upper Saddle River, NJ.

[14] Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis: An Empirical Study. In *31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, New York, NY, USA, 332–343. DOI: http://dx.doi.org/10.1145/2970276.2970347

[15] Arnal Dayaratna. 2018. IDC's Worldwide Developer Census, 2018: Part-Time Developers Lead the Expansion of the Global Developer Population. [Online]. Available: https://www.idc.com/getdoc.jsp?containerId=US44363318. (October 2018). Last accessed 8 January 2020.

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

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

[18] Matthew Green and Matthew Smith. 2016. Developers are Not the Enemy!: The Need for Usable Security APIs. *IEEE Security & Privacy* 14, 5 (Sept 2016), 40–46. DOI: http://dx.doi.org/10.1109/MSP.2016.111

[19] Randy Hodson. 1999. *Analyzing documentary accounts*. Number 128 in Quantitative Applications in the Social Sciences. SAGE Publications, Inc,, Thousand Oaks, California.

[20] Luigi Lo Iacono and Peter Leo Gorski. 2017. I Do and I Understand. Not Yet True for Security APIs. So Sad. In *Second European Workshop on Usable Security (EuroUSEC)*. Internet Society, Paris, France, 1–11. https://www.ndss-symposium.org/wp-content/uploads/2018/03/eurousec2017_15_LoIacono_paper.pdf

[21] Jenkins. 2020. Jenkins User Documentation. [Online]. Available: https://jenkins.io/doc/. (2020). Last accessed 8 January 2020.

[22] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *35th International Conference on Software Engineering (ICSE)*. IEEE, San Francisco, CA, USA, 672–681. DOI: http://dx.doi.org/10.1109/ICSE.2013.6606613

[23] Amy J. Ko and Brad A. Myers. 2003. Development and evaluation of a model of programming errors. In *IEEE Symposium on Human Centric Computing Languages and Environments (HCC)*. IEEE, Auckland, New Zealand, 7–14. DOI: http://dx.doi.org/10.1109/HCC.2003.1260196

[24] Amy J. Ko and Brad A. Myers. 2004. Designing the Whyline: A Debugging Interface for Asking Questions About Program Behavior. In *SIGCHI Conference on Human Factors in Computing Systems (CHI)*. ACM, Vienna, Austria, 151–158. DOI: http://dx.doi.org/10.1145/985692.985712

[25] Amy J. Ko and Brad A. Myers. 2005. A Framework and Methodology for Studying the Causes of Software Errors in Programming Systems. *Journal of Visual Languages & Computing* 16, 1-2 (Feb. 2005), 41–84. DOI:http://dx.doi.org/10.1016/j.jvlc.2004.08.003

[26] Amy J. Ko, Brad A. Myers, and Duen Horng Chau. 2006. A Linguistic Analysis of How People Describe Software Problems. In *Visual Languages and Human-Centric Computing (VLHCC)*. IEEE, Brighton, UK, 127–134. DOI:http://dx.doi.org/10.1109/VLHCC.2006.3

[27] Richard. A. Krueger and Mary Anne Casey. 2015. *Focus Groups: A Practical Guide for Applied Research, 5th Edition*. SAGE Publications, Inc,, Thousand Oaks, California.

[28] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. 2017. CogniCrypt: Supporting Developers in Using Cryptography. In *32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Urbana-Champaign, IL, USA, 931–936. DOI: http://dx.doi.org/10.1109/ASE.2017.8115707

[29] Brad A. Myers and Amy J. Ko. 2003. Studying Development and Debugging to Help Create a Better Programming Environment. In *Workshop on Perspectives in End User Development, ACM Conference on Human Factors in Computing Systems*. ACM, Fort Lauderdale, FL, USA, 65–68.

[30] Brad A. Myers and Jeffrey Stylos. 2016. Improving API Usability. *Commun. ACM* 59, 6 (May 2016), 62–69. DOI:http://dx.doi.org/10.1145/2896587

[31] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2016. "Jumping Through Hoops": Why do Java Developers Struggle With Cryptography APIs?. In *38th International Conference on Software Engineering (ICSE)*. ACM, Austin, Texas, 935–946. DOI: http://dx.doi.org/10.1145/2884781.2884790

[32] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, Marco Herzog, Sergej Dechand, and Matthew Smith. 2017. Why Do Developers Get Password Storage Wrong?: A Qualitative Usability Study. In *2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, Dallas, Texas, USA, 311–328. DOI:http://dx.doi.org/10.1145/3133956.3134082

[33] Duc Cuong Nguyen, Dominik Wermke, Yasemin Acar, Michael Backes, Charles Weir, and Sascha Fahl. 2017. A Stitch in Time: Supporting Android Developers in Writing Secure Code. In *2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, Dallas, TX, USA, 1065–1077. DOI: http://dx.doi.org/10.1145/3133956.3133977

[34] Daniela Seabra Oliveira, Tian Lin, Muhammad Sajidur Rahman, Rad Akefirad, Donovan Ellis, Eliany Perez, Rahul Bobhate, Lois A. DeLong, Justin Cappos, and Yuriy Brun. 2018. API Blindspots: Why Experienced Developers Write Vulnerable Code. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS)*. USENIX Association, Baltimore, MD, USA, 315–328. https://www.usenix.org/conference/soups2018/presentation/oliveira

[35] Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bill Chu, and Heather Richter Lipford. 2015. Questions Developers Ask While Diagnosing Potential Security Vulnerabilities with Static Analysis. In *10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, Bergamo, Italy, 248–259. DOI: http://dx.doi.org/10.1145/2786805.2786812

[36] Stack Overflow. 2019. Developer Survey Results 2019. [Online]. Available: https://insights.stackoverflow.com/survey/2019#developer-profile-demographics-gender-minorities-by-country. (2019). Last accessed 8 January 2020.

[37] Jeffrey Stylos and Brad Myers. 2007. Mapping the Space of API Design Decisions. In *IEEE Symposium on Visual Languages and Human-Centric Computing, 2007 (VL/HCC)*. IEEE, Coeur d'Alene, ID, USA, 50–60. DOI: http://dx.doi.org/10.1109/VLHCC.2007.44

[38] The European Parliament and the Council of the European Union. 2016. 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. [Online]. Available: http://data.europa.eu/eli/reg/2016/679/oj. (2016). Last accessed 8 January 2020.

[39] Tyler Thomas, Bill Chu, Heather Lipford, Justin Smith, and Emerson Murphy-Hill. 2015. A study of interactive code annotation for access control vulnerabilities. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Atlanta, GA, USA, 73–77. DOI: http://dx.doi.org/10.1109/VLHCC.2015.7357200

[40] V. Javier Traver. 2010. On Compiler Error Messages: What They Say and What They Mean. *Advances in Human-Computer Interaction* 2010, Article 3 (Jan. 2010), 26 pages. DOI: http://dx.doi.org/10.1155/2010/602570

[41] Michael Whitney, Heather Lipford-Richter, Bill Chu, and Jun Zhu. 2015. Embedding Secure Coding Instruction into the IDE: A Field Study in an Advanced CS Course. In *46th ACM Technical Symposium on Computer Science Education (SIGCSE)*. ACM, Kansas City, Missouri, USA, 60–65. DOI: http://dx.doi.org/10.1145/2676723.2677280