Developer-centered security and the symmetry of ignorance

Olgierd Pieczul
IBM
Dublin, Ireland

Simon Foley
IMT Atlantique, Lab-STICC,
Université Bretagne Loire
Rennes, France

Mary Ellen Zurko
MIT Lincoln Laboratory
Massachusetts Institute of Technology
Lexington, USA

ABSTRACT

In contemporary software development anybody can become a developer, sharing, building and interacting with software components and services in a virtual free for all. In this environment, it is not feasible to expect these developers to be expert in every security detail of the software they use, and we discuss how difficult it can be to build secure software. In this respect, the practical challenges of the emerging paradigm of developer-centered security are explored, where developers would be required to consider security from the perspective of those other developers who use their software. We question whether current user-centered security techniques are adequate for this task and suggest that new thinking will be required. Two directions—symmetry of ignorance and security archaeology—are offered as a new way to consider this challenge.

CCS CONCEPTS

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

ACM Reference format:
Olgierd Pieczul, Simon Foley, and Mary Ellen Zurko. 2017. Developer-centered security and the symmetry of ignorance. In *Proceedings of New Security Paradigms Workshop, Santa Cruz, California, USA, October 1–4, 2017 (NSPW'17),* 11 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Recent shifts in software engineering technology and culture make the traditional view of the software developer obsolete. Software development is no longer the exclusive domain of the highly-skilled technology professional, or the apprentice coder working in a single language with a fully fleshed out design and lead architect for guidance. Contemporary systems enable a continuum of developers, ranging from the end-user using domain-specific languages to tailor their application use, application developers constructing software from a palette of existing components, administrators scripting complex network deployments, and programmers focused on highly specialized code. Within this continuum, each developer has their own domain expertise and uses this expertise, along with the available systems and tools, to solve their domain problems. However, individuals in this continuum are not omniscient; expertise is within some domain and there will always be parts of the system/domains about which they are ignorant: the end-user ignorant of the implementation of the system he uses, the application developer ignorant of esoteric details of the APIs with which they program, while also not necessarily understanding every application of the software they build nor the network infrastructure that hosts their application. We have *symmetry of ignorance* between individuals: expertise and ignorance distributed across the continuum, which in the context of user-centered design, Rittel [42] describes as a wicked problem.

An individual in this continuum, while expert in their own domain, may introduce security vulnerabilities through being ignorant of some detail of some artifact that they use in solving their problem. In addition, they have no notion of ensuring that the callers of their code will not do the same. The challenge is, therefore, to ensure that the developer of any artifact provides it with all appropriate security, from the perspective of those who use the artifact while being ignorant of its underlying detail.

This challenge is well understood from the perspective of end-users and the systems that they operate. With over 20 years of research on *user-centered* security, it is almost universally acknowledged that users should not be blamed for bad security design of computer systems' security [44]. While there are still problems to be solved, the security for users has improved significantly.

Usable security for developers encompasses a broader range of factors, such as the design of the APIs that they use, availability, quality and accessibility of documentation and code examples, development tools and developer-oriented user interfaces. Contemporary developers have to rely on security qualities and mechanisms provided by the numerous components they integrate into their software. However, as a result of bad (or little) design, difficult access to important information and a general tendency to transfer security burdens, they often fail to understand and use them correctly (e.g. [13] and [14] for just two published examples). The traditional view that developers are fully responsible for the security of their software and that awareness, education and testing are sufficient is beginning to change. Over the last few years we observe an emerging paradigm of developer-centered security and attempts to adopt well established usable security measures to software development. This also maps within our broader continuum: developers of a software artifact should consider its security from the perspective of other developers who may use it while being ignorant of its underlying detail.

However, engaging in usably secure software artifact design in a manner that considers its developer-user's needs, while cognizant of its user's ignorance, is not a straightforward task. The fact that programming became more accessible does not mean that the development of secure software is less complex. The scale and

NSPW'17, Santa Cruz, California, USA

DOI: 10.1145/nnnnnnn.nnnnnnn

layered nature of the development ecosystem, the diversity of programming tools, processes, platforms, and developers themselves, is unprecedented. The well-established solutions are too simple for that problem and, we posit, the familiar research methodologies are too limited.

In this paper we discuss practical challenges of the emerging developer-centered security paradigm, focusing on scale and complexity of all sorts. The key focus and contribution of this paper is deepening our understanding and exploration of this paradigm. We suggest a view through the lens of *the symmetry of ignorance* (Section 5). Symmetry of ignorance provides us with a new way to consider the challenge of developer-centred security. While not a paradigm per se, it has been used in Software Engineering to provide insights on software design. In this paper we use it to provide a new way to synthesize and understand the challenges of developer-centred security; we are not aware of previous research using symmetry of ignorance in this way. Recognising that symmetry of ignorance underlies developer-centred security forces us to rethink whether a user-centred paradigm is adequate. In this paper we explore these inadequacies and put forward a challenge for a new security paradigm.

The paper is organized as follows. Section 2 discusses developers today, providing the foundation for pertinent complexity and scale challenges. Section 3 outlines the challenges we see from that foundation. Section 4 lays out special considerations that will inhibit the impact of developer centered security findings and results. Section 5 introduces the Symmetry of Ignorance approach. Section 6 discusses the current state of developer-centered security research. The concluding section summarizes the paper thus far.

2 TODAY'S DEVELOPERS

In the past, most developers used to work in relatively specific environments involving a single language and platform, such as C and Unix. Programming used to be a specialized, engineering discipline. While practicing this craft, the programmers became more experienced, gained understanding and expertise in their field, language and platform. Switching between them was considered a challenge that required training (formal or experiential).

2.1 Accidental developers

Today, programming is not a specialized task, but rather one of many aspects of *using* computer systems. Initiatives such as CoderDojo promote teaching programming for kids as young as seven. In many countries, programming is part of the national curriculum in primary schools. In a few years, most school graduates will be familiar with programming, much like their older friends or parents were familiar with spreadsheet software, word processors, or email.

Basic programming is just one of many useful skills that potential job candidates are expected to have. As computer software becomes part of everyday life, programming becomes easier and has more applications. Commonly used software products, such as office applications, provide embedded programming environments to extend their functionality. Social media services and mobile platforms provide ways to extend or integrate their services with minimal effort and only very basic coding skills. Simple apps, plugins, integrations and extensions are developed by thousands every week. Also, many businesses today integrate various service providers such as online stores, map services, delivery providers (see Section 2.2). This, in turn, encourages more people to learn some programming, as yet another IT skill that may be expected from their future employers [45].

2.2 Everyday developers

The way professional developers interact with coding has changed in recent years. Software business pushes towards increasingly faster software development paradigms and higher levels of abstraction. Applications are developed as mash-ups of software components, cloud services and platforms. Developers are expected to code rapidly using off-the-shelf solutions. The competition, and a continuous push for providing more functionality and integration, results in developers being expected to gain a basic understanding of a vast number of new technologies, APIs, platforms and frameworks.

Rather than gaining a more in-depth understanding of a specific platform, developers increase the breadth of the languages and tools they are able to use. It is typical that a modern developer would use a number of different programming languages routinely in their daily job. In addition, the configuration of software frameworks and platforms is often so advanced and customizable that the boundary between *configuration* and *coding* becomes blurred.

Similarly, the traditional separation between development and operations or administration is vanishing. With new software delivery approaches such as *DevOps*, developers are expected to automate packaging, deployment, monitoring and operation of their *service*. This further increases the scope of their interests as well as requires an even broader understanding of service delivery elements that programmers of the past did not have to consider. The software has to be automatically deployed in some, often virtual, environments which are also typically exposed to developers as a programmable service. The infrastructure becomes yet another software component that has to be integrated with their code, and potentially extended with more code using integration points provided by the platform.

The time spent on what was traditionally considered "programming" becomes smaller, compared with the time required to find and learn new tools and technologies, analyze sample code and decide on the integration approach. This makes developers more *users* of the software than experienced *programmers* of the software they use.

Anecdotally, we believe that some application developers know less about the components they integrate into their product than many of the end-users know about their everyday application. Much greater diversity and variability of components means less time and attention using and developing skills. Often, professional *usage* of an application, such as CAD or finance analytics, requires more tool-specific skill than *development* of software.

2.3 Proliferation of software components

While early software development was a language based discipline, modern software development is a component integration job. It is more like engineering, without the physical constraints on the form factor of the components. Today's software is developed using

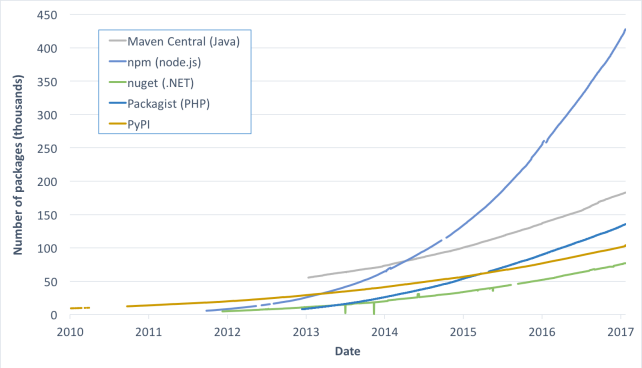

Figure 1: Number of components in the most popular public repositories, data from http://www.modulecounts.com

many separate, interoperating components. In recent years, and with growing popularity of platforms such as Node.JS, the number of components has grown significantly [55]. Figure 1 depicts the growth of a number of components in popular repositories in recent years. For example, by July 2017 the NPM repository for JavaScript components has reached 470,000 distinct packages/modules, with over 500 new modules on average being published daily. In each case, these software components can be effortlessly downloaded and integrated into an application using package managers that automatically resolve all dependencies, adding components when necessary. It is not uncommon that a very simple application, can depend upon many hundreds of software modules and be a dependent of many hundreds of others. The level to which developers rely on re-usable code is also unprecedented [31]. The time for LangSec has come and gone. It is time to work on ComposeSec [25, 26].

3 CHALLENGES FOR DEVELOPER CENTERED SECURITY

3.1 Layered software abstractions

As we've discussed, today's software is a collection of interoperating components working in parallel and/or encapsulating one another. Misunderstanding of an API or unexpected component interoperation may lead to unpredictable security exposures. They emerge from component misuse in areas that might not have been considered as relevant to security. For example, developers typically do not expect that the URL class from the Java standard library allows URLs to point to local files [38] and corresponding documentation can be misleading by focusing primarily on HTTP URLs. Such counter-intuitive behavior, the "dark side of the code" [38], can be encapsulated in a component and effectively hide all information about potential risk from its consumer.

A developer, casually integrating software components may not fully comprehend their end-to-end operation. This, in turn, may mean that they unwittingly enable execution paths that were not expected, and those paths may lead to security problems. It is likely that neither the developer integrating the component, nor the developer of the component, anticipated the usage of the component in that particular way may have security impacts. API usability, even when not thought to be related to security, has an impact on their usable security.

3.2 Definition of interface

We find usable security work clustering around user-facing security related features or tasks. Examples include authentication, access control and social media sharing, app permissions, personal information exposure, crypto use for protections, crypto related errors, and software updates. It is not clear, however, what should be considered to be the equivalent of the interface in the context of developer's security usability. Developer centered security research recognizes documentation and information sources as part of the developer's interface. We believe there is more to the "interface" to be considered.

In the most narrow sense, the interface is just a set of method names or their REST calls, their parameters, return values and so forth. Developers experienced with an API, or working with code that already uses an API, may be comfortable interacting with only that. For example, a developer aware of the `"GET /user/:id"` REST API that retrieves the user record may expect that corresponding `"DELETE /user/:id"` API should be used to delete a user record, based solely on the fact that such an API exists.

The wider definition of an interface could include immediately available documentation that is presented to a developer when writing code, or just listing available APIs. Such brief documentation would typically contain a short description of an operation of an API, descriptions of parameters, and so forth. For example, it is common that Java developers will be presented with Javadoc documentation tooltips while writing the code. Developers browsing REST APIs will typically see similar information generated using a tool such as Swagger.

An even wider definition of an interface may include exogenous documentation with a discussion on the APIs usage, security considerations, and code samples. In modern platforms access to that documentation may be just one click away and developers will consider referring to it as an essential or natural part of their programming routine. Even further from the code context, "interfaces" include official support forums, blog posts and articles. We see developer security studies beginning to use all these aspects in various limited combinations as the community builds up the base model of what it means to study developers doing their tasks securely.

If we take a view that an interface is what a developer interacts with when working with an API, the interface would also include third-party content such as community forums and third party code samples. These sources of information can be equally or, for some developers, more popular than the official ones [2, 41]. Note that the vendor or API producer does not have control over that content. It is suggested [2] that improved documentation writing, or making it more question and answer focused, may make more developers use official sources of information. In the same way that web search has come to dominate how end-users look up a definition or reference explanation, developers will continue to use unofficial information, especially if their question is simple or their engagement with the component/service in question is incidental. There is progress towards acknowledging community

sources such as Stack Overflow as part of the interface, albeit not fully controlled. Major software vendors have begun to provide their official documentation using the Stack Overflow platform and experience [33]. A more thorough approach would promote safe usage of APIs and also correct wrong recommendations in community sources. Would it take a Heartbleed type exposure traced to those sources to incentivize such a behavior change?

The gradually increasing scope of the developer's interface when interacting with an API does not need to mean that programmers would typically work in one scope or the other; or that broader scopes include all of the narrower elements. Each developer may uniquely combine them. In addition to a developer's preference, the effective interface will also depend on the type of API, the platform and development tools, and what they provide. It may also depend on the programmer's familiarity with the API or API type, or APIs from the same vendor, its current usage in the code, or whether the vendor has provided useful material in any of the sources.

For end-user interfaces, the most narrow scope is the GUI, then integrated help, online documentation, and so forth. Research on usable security recognizes that users may prefer to use different parts of the interface [15], types of help and documentation [24] or errors and warnings [11]. We argue that in the case of developers, the scale and possible variety in different combinations of elements that define their interface are much larger, and consequently raise new challenges to systematic analysis.

The developer's lifecycle does not stop at programming and direct usage of APIs. With the cultural shift towards the DevOps model, developers are often responsible for configuration and operation of their software. Depending on the nature and scale of their software, developers may perform these tasks manually using user/administrator interfaces, or automatically using additional APIs.

Effectively, the concept of interface, in the context of developers, is the API itself and the entire ecosystem surrounding it. As the traditional boundary between the user, programmer and administrator becomes blurred, the area of study of developer-centered security expands to encompass all of these.

3.3 Demographics and work environment

Ecological validity of study demographics is a key issue in usable security research. For end-user studies, appropriate population sampling techniques are used. In addition, exposure to relevant security concepts is often measured and reported to check for bias or impact. While the population of developers may be seen as more homogeneous than general software users, in the context of the efficacy of security usability measures it may be much more diverse. Some research points out different groups of programmers [34], such as novice or professional. [3] suggests considering company size as well. However, we argue that the demographic factors to consider go beyond what is currently in the literature. Below, we describe some additional key factors. We also discuss environmental complexities that we believe will call for significantly new approaches to developer assistance and research.

Developer skill and experience. Developers have varying levels of programming skill, with no universally accepted measure [8]. They may be professionals who program for a living or just accidental. This can be captured with a survey question. They may also be professionals in another computing field for which programming is a side task, which is more difficult to capture. Programmers varying understanding of and experience with security is also currently captured with survey questions. Developers will have different abilities in comprehending various parts of the extended interface, such as documentation. As we discussed, where they go for interface information will vary, and also vary within a single developer by target field, platform, tool or API.

Team, structure, culture and policies. Another set of demographic factors is the environment in which a developer works, the development process, technical leadership and oversight. A programmer working on their own will be dependent on their own understanding of the API. In professional environments developers work in teams, sometimes code in pairs and continuously exchange information. They may commonly use different, or multiple, sources of security best practices advice that may vary in content [4]. The written code is typically reviewed, sometimes in the context of security. This may strengthen (or weaken) developer motivation for due diligence, or make it simpler to find key information and interpret it. Development processes may have an impact (e.g. waterfall, agile, hybrid). The existence of specific security policies and processes will have an impact. These include having a secure development lifecycle, who is explicitly responsible for security, whether there is security testing, what security test tools are used, and are there compliance mandates (e.g. government certification, PCI).

Existing code. Another factor impacting a developer's approach to security is whether the developer is coding in a language, integrating the component or service with their software, or working with existing component integration calls. The existing contextual code is likely to drive assumptions and exposure to extended interface information. Working with existing integration calls is likely to limit their exposure to a very narrow part of the component interface, such as API name and parameters. It may also bias their understanding, making them follow existing, good, or wrong interpretation and practices.

In addition, every time a developer works with a new API (directly or indirectly), they will begin with assumptions based on all the APIs they've worked with, and their work with the new API will shift their model of what to expect with the next API.

Automated assistance. User security can be improved by automatic assistance, reminding or warning them about potential threats. Similarly, automatic assistance can be provided to developers, warning them about potentially dangerous coding patterns or mistakes. Such tools range from simple static analysis with prescribed patterns, through data-flow models, to psychology-driven solutions recognizing developers' shortcomings [9]. However, the breadth of possible activities and complex nature of security errors in contemporary software [38] limits the efficacy of these tools. This raises a research question, how far can automated developer assistance scale to cover modern software. Is it just a matter of larger databases of problems, more sophistication in static analysis, or taking advantage of community and crowdsourcing, or do we need to look for new mechanisms? Perhaps the assistance tools can involve humans and help to automate and simplify human-based review or community oversight [32].

3.4 Impact on experiments

[3] provides an excellent overview and research agenda for taking the usable security research techniques and approaches used for end-users, and applying them to developer usable security. Their emphasis is on what might reasonably be accomplished in this more complex domain. In this context, we are looking towards what new approaches might be applied given the complexity we have outlined above. We are particularly interested in new approaches to field studies or analyzing data from the field. [3] rightly calls out that in-situ observational, field, and/or diary studies are likely to be needed. In particular, field-based work will be most obviously needed when research is meant to cover code development at the scale beyond a single developer working on a single app. Organizational demographics have the potential for a great deal of impact. Also, recruiting developers, especially professional, for experiments may be harder and more costly than software users.

We previously reported on the limitations of expert review of a security UX in the wild [59]. In the context of displaying public key certificates, we found that expert review was only helpful for the most approachable parts of that task, and could not provide help on areas that required deep domain expertise to understand and discuss. While that is a diminishing problem for end-user security experience, it is likely to loom large for developer tools and APIs for security. One author forced herself to use a very painful text editor during the early days of usability evaluations of text editors, to remind herself of the limitations of the technology and approaches. At this stage we can only recommend rigorous evaluation of usability impact of any expert reviews for developers' tools with instruments that work on tasks in the wild or in context.

We have begun (albeit naively and without the help of a professional in the field) to explore how **Archaeology** can be applied to developer artifacts to produce insights. The term "software archaeology" is currently in use as a pure software engineering notion applied to support considerations of poorly documented legacy code, aimed at understanding what the code does. Instead, we propose integration with true archaeological activities as a new paradigm for developing insights into human activities in developer-centered security, in the production or use of security relevant software. What differentiates archaeology from other approaches using sociology and ethnographic anthropology [49] is that archaeology starts its analysis with artifacts and other forms of material culture. Instead of working first to achieve access to the humans involved, an archaeological approach can start with artifacts such as programs, binaries, and source code. Similar archaeological approach has been applied in other disciplines, such as media [27].

We find the possibilities particularly interesting for studying human activity through the analysis of open source code and projects, since those are highly available artifacts. We also see the potential for an artifact-first approach to provide insights into contexts that are difficult for current usable security research methods to handle, including closed cultures and historical organizational activities that are no longer available to be directly inspected or analyzed. One archaeological approach is to look at artifacts in layers over time, reconstructing the progression and mapping to actual time. This approach might yield new insights into how the developers approached or impacted the security of a code base over time, as each layer builds upon or responds to the previous ones. This can yield insights into tools and techniques that build or maintain security in a code base over time.

Another archaeological approach that might yield insights into developer centered security is considering "how was this thing used" (or "for what purpose was it used"). This is done by doing detailed analysis of the object. Archaeologists examine bits of a pot to see what did it contain at some point (cabbage shows up a lot), was it ever in a cooking fire, does it have a handle that has wear marks from being handled by a hand or by a stick, etc. Archaeological analysis of the use of security code (such as APIs) might log and analyze the various values passed in as parameters, the types of programs it is found in, the set of languages in the system it is in, and so on.

The archaeological approach can be seen as a form of expert review, but instead of UX experts, with experts versed in software developer history and culture, and in software architecture, tools, and artifacts. At this stage of exploration, it is not clear which fields and techniques used in archaeology will yield the best results when analyzing software. Organizational culture and history are likely to have a deeper impact on developer centered security than they do on end-user security. We see this as a new opportunity for collaboration with emerging archaeology grad students.

Our initial foray [39] considers the evolution of security defenses in a contemporary open-source software package over a twelve year period. The qualitative analysis style study systematically analyzed security advisories, codebase revisions and related discussions. A number of phenomena emerged from this analysis that provide insights into the process of managing code-level security defenses. This study confirmed in the wild a phenomena previously observed only in lab experiments, such as developer "blind spots" [37], or only demonstrated at small scale [38]. It also showed that metrics, such as CVSS, often used in quantitative studies are arbitrary and inconsistent. The study also showed that performing qualitative style investigations requires significant effort even for the relatively limited scope.

The recommendation in [3] to model lab studies of bug finding and fuzzing tools after successful investigations into end-user security tools consciously assumes that the lab tasks will be simplified in terms of both code base and organizational interactions. In our experience in the wild, the context of use of existing tools is nothing near that simple. In general, the current state of security bug finding tools presumes familiarity with both the code base and what the security error case means. Familiarity with the code base can be made tractable in a lab study, but, as we have discussed, is not common in the wild for systems of more than app size. Understanding of security error cases is more likely to be a place where lab studies of existing tools can produce insights. For example, when fuzz testing finds a crash, the developer has a crash to analyze, and a protocol run captured by the tool with one or more errors, potentially over an extended period of time. When fuzz testing finds a system that is unresponsive (or inappropriately responsive) in the same context (either to a single request or for some short period of time), the developer has the captured protocol run and a dynamic, running system. The debugging challenge is nontrivial, in terms of understanding the system/code, error cases executed up to

that point, and protocol specifications/requirements. We leave aside the cases when the fuzz testing itself is in error (which we have encountered). In short, we recommend field studies as a precursor to any lab studies of existing vulnerability-finding tools, to provide qualitative insights into their use.

IDEs are considered effective in practice. Developers are notoriously fussy (or fanatical) about their IDE of choice. In some cases, there are very compelling reasons to choose the very simplest text editors (e.g. vi, emacs). Developing code for contexts such as network devices forces a lifecycle over a very restricted UI (e.g. ssh'ing into the device to debug and change the code). This will only become more common with the growth of IoT. Research on whether techniques pioneered in a full GUI IDE can survive the transition to vi will be sorely needed.

4 ADOPTION INHIBITORS

The adoption of security usability measures for developers may face challenges that are different from those of security for software users.

4.1 Incorrect assumptions

While software vendors may accept that users require support, they have not yet recognized that developers, users of their APIs, are experiencing similar problems and require assistance [17]. Developers of the libraries or APIs continue to assume (or are allowed to assume) that the consumers will read the documentation. Security-related information relegated to the documentation can include guidance on secure usage of the component, or due diligence tasks. Given the number of components the developer has to deal with means that it is often impossible to carefully examine the documentation. We see a place in the research agenda for using the same economic techniques that have been used to determine the amount of time and user value reading all privacy policies would cost. Even if attempted, developers may not reach a part that covers a specific security-related information consideration [38].

In the case of small open source software components, the documentation often does not exist and developers are directed to the source code. Examining the discussions of the popular code platforms confirms that many developers assume that the consumers will examine the source code before using the code [6]. Developers often do not attempt to read official documentation and instead look for easy examples in web search or on sites such as Stack Overflow [41]. This is not a surprise, as in many cases developers work with examples, often have very simple problems to solve and seek a quick answer. This, unfortunately, can lead to replicating insecure code or patterns [2]. As with other aspects of human nature, it will take the boldest possible statements about the results of usable security studies to provide direction that alters assumptions and misconceptions.

4.2 Misaligned incentives

Usable security for developers is not currently considered to be a feature that gives a market advantage to the vendor. When making their software or services extensible with APIs, many vendors still provide it only at a basic level, considering it secondary to the end-user functionality. Their main objective in the area of APIs is likely to be allowing partners to cover gaps in functionality or explore niche markets or use cases. In the case of open source, or community-driven products, volunteer contributors are often more interested in solving interesting technical problems, rather than focusing on good developer experience or documentation. SimplySecure is one of the few initiatives targeted at bringing usable security and privacy to open source components.

Vendors' interest in rapid adoption of their APIs emphasizes allowing developers to integrate quickly and easily. Some measures that enhance secure usage of the service by developers may be perceived as slowing down the adoption [35]. For example, many cloud services offer consumers the ability to receive runtime updates from the service delivered to endpoints on their server, known commonly as webhooks. Since the communication between the service and consumer is done over the Internet, the communication should be encrypted and authenticated. Service vendors could provide only HTTPS endpoints to eliminate the possibility of a consumer implementing insecure communication. This would require the consumer to perform extra steps to enable HTTPS at initial integration and testing, including obtaining a certificate and securely storing the private key. Vendors have not yet generally decided to implement such requirements. We have yet to see a developer security study that measures task time to securely use a service or API in the same way user security studies have.

The incentives for developers to use their APIs securely is another aspect meriting attention with economic analysis techniques. Software end-users share the incentive for secure application usage with usable application designers, though they are not always aware of the risks. Developers consuming APIs are less incentivized to cooperate on security responsibility. Security can be a concern secondary to functionality and time to market [3]. In addition, developers are not by and large directly impacted by not taking advantage of the (usable) security measures provided to them. It is application users that will eventually be exposed.

4.3 Backward compatibility

Changes impacting users, such as changes to the user interface, can be immediately consumed, especially in the context of cloud-based services. However, consuming changes to API security usability has much more friction. Developers use a particular version of the API, and do not look for updates, especially if the library in that particular version is already used. They have dependencies on particular versions which will result in costs in development time and testing, for themselves and their team, if they update. Finally, when they write the code they will typically not actively look for any updates in API or documentation.

Improving the security usability of their programming interfaces is also challenging to vendors. They can not easily deprecate APIs, because of the cost of upgrading to their consumers. This means that any change results in another version of the API that has to be supported in parallel with old versions, often for several years. The same applies to documentation, code samples and so forth. The more popular and widespread the API is, the harder it will be to deprecate the old, unusable one. This is especially expensive in the context of cloud APIs where the support means maintaining the software that handles every version of the API. The technical

difficulty and additional cost will discourage software vendors from investing in improving security usability of their programming interfaces.

With the proliferation of software components and cloud APIs, vendors have to frequently update their software interfaces. The software community has yet to develop mechanisms for making API changes easy to handle and adopt by the consumers. Future research in this area should consider the problem of addressing remediation of usability problems. We see an analogy with the recent emergence of end-user software patching as a research topic in usable security.

4.4 Security usability vulnerabilities

In our previous research [39] we observed examples where problems that were not technically security vulnerabilities, but were usability problems, were reported to the development team as vulnerabilities and fixed as such. In some cases, the security mechanism was so counter-intuitive and difficult to work with that it resulted in an implementation bug by the component's own development team. In that same exploration, we found instances of similar problems that were dismissed by component developers as not technically a vulnerability. Even within a single community, the approach to usable security problems can be inconsistent.

Security usability problems may lead to security exposures in the component or service consumers. Some may be considered to be bugs, or unacceptable negligence by the vendor. A usability bug is more likely to require API changes than other vulnerabilities in a component, which can often be fixed by internal component changes. When internal vulnerabilities are fixed, they only require the consumer to update the component to a new version and, in the case of cloud services, the consumer also owns deploying that update. A security usability bug fix can have more expensive impact.

This raises a question of what processes and practices the software community should adopt to deal with usability problems that have an adverse security effect. For example, should a misleading API name, or insecure usage of an API included in documentation, be considered just a flaw in usability or a security vulnerability? Recognizing a problem as a vulnerability has consequences, especially in the context of professional/enterprise software, where secure development lifecycles typically levy requirements on the process of disclosure to the customer, and official vulnerability advisories. The effort required to process a usability problem this way will discourage vendors from including them in the vulnerability classifications, especially if, strictly speaking, the component itself is not vulnerable [39]. This is in addition to the backward compatibility problems explained in the previous section.

On the other side, with the number of software components used and easy to use software repositories [55], consumers may not be reading security advisories, but rather rely on tools that automatically update vulnerable software components to safer versions. This will prevent them from learning that the "vulnerability" might have an impact on the way their software was implemented and may require changes in their code to remediate the problem.

We believe that effective notification about security usability problems is, on its own, a security usability problem. It involves extending research to areas such as CVSS, CVEs, security incident response teams, and the secure development lifecycle requirements on all of them. The software community should establish a process for handling this kind of problem, considering that it is unlikely that consumers will actively look for information about such problems or analyze the changes in APIs or documentation.

5 SYMMETRY OF IGNORANCE

User-centered design strives to avoid the bias inherent in technology-dominated development whereby technology experts are believed to better understand the end-user's needs. On the other hand, as an application domain expert, the end-user best understands their own needs, while their understanding of the system they use is limited to how they consume its interface (as characterized in Section 3.2) and are typically ignorant of its underlying implementation/technology. Equally, the professional developer, while considered an expert on the implementation underlying the interface they produce, is often ignorant of the domains to which their systems will be applied by end-users. Thus there is a *symmetry of ignorance* [42] between the end-user and developer: the end-user domain expert is ignorant of the implementation while the developer implementation expert is ignorant of the user domain. The *symmetry* in this context means that some degree of ignorance, thought not necessarily equal, exists at either side.

This symmetry of ignorance plays out across the many stakeholders in a contemporary system. As illustrated in the previous sections, we have many kinds of developers who could be considered domain experts for the interfaces that they produce while potentially ignorant of the implementation of the interfaces produced by others and which they consume. However, symmetry of ignorance is not limited to end-users and developers, it applies to all stakeholders in the system, including system administrators and architects.

The case-study in [38] can be seen as an example of symmetry of ignorance. In this case-study, a developer of an online bookmarking service uses an existing library component to take a web-page snapshot of the URL to be bookmarked. However, they are ignorant that integrating this component may expose systems in their local network, and consider that any such problem, if it exists, should be handled by an administrator of the system that uses an application. The administrator, however, is ignorant of the fact that a bookmark application can expose their network, and ignorant of the expectation that they should provide security controls. In addition, the snapshot library developer is unaware that the network access platform API they use allows for access of local files through `"file:"` URLs, while a platform API developer does not expect its users to have not first fully read and understood the documentation and the cited RFC. Lastly, the application developer, consuming the library, is ignorant of this issue as it is hidden by encapsulation.

Thus, user-centered security should not be limited to just end-users and developers, it should concern all the producers and consumers of interfaces and the recognition that there is both expertise and ignorance distributed across these stakeholders, which in the context of user-centered design, Rittel [42] describes as a wicked problem.

Fischer [18, 19] argues that user-centered design techniques are generally limited to closed systems; the techniques focus on understanding the needs of the user and "designing the system for use before use". There is a similar emphasis in user-centered security: security is designed by considering user needs before use. Participatory-design approaches, where users are actively involved in the design, have a similar focus; for example, in the use of dialectics as a means to understand security needs in mobile app development [54]. While security-usability studies may help to improve understanding of how to support user needs securely, it is at heart, design for use before use. Studies that test security usability are iterations, and in the sense of this model, the design for next use. Usable security is lacking in techniques that can predictively design for the next range of uses which will dynamically evolve as part of use.

5.1 Whither user-centered security?

This separation between design and then use does not fit easily with our contemporary system view, where user needs evolve, requirements change and new technologies are incorporated, all as a part of the normal 'use' of the system. Fischer considers that it is not possible to design such open systems that anticipate all uses in advance and that one must "design to support design after design" [19]. With respect to security, we see this need for a new paradigm that supports design for design after design across the examples in the previous sections. However, it is not clear whether user-centered security can enable this. Much of the discussion and examples in the paper illustrate this; we focus on a number of examples in the following.

Dark side of the code. There is a *dark-side* to an API [38], whereby part of its underlying behavior, while understood by its developer, is not properly understood by the developer who uses the API, and through mis-use introduces security vulnerabilities. Equally, the component developer may not appreciate the various ways that his API may be used. This symmetry of ignorance was illustrated in Section 3.1 and the case study [38] on the online bookmarking service outlined above. At its heart is the problem that design for use before use promotes a dark-side of the code. Decisions are made about how an API is used when least is known about how the API will be used. In some respects, through design for use before use, we are revisiting the age-old debate about structured development [29] which pointed to the dangers of making implementation-design decisions when least is known about the implementation.

Today's secure default is tomorrow's vulnerability. Commonly recommended for user-centered security [10] and equally relevant to developer-centered security, secure default is intended to allow secure use of the system without having to configure security beforehand. For example, the producer of an SSL package provides a default keystore configured with trusted CAs and secure cipher specifications, enabling secure communication by default. As a simple dictum, secure default is an example of design for use before use. Once the design is deployed, CA keys may be compromised, ciphers deprecated and vulnerabilities discovered, whereupon today's secure default becomes tomorrow's vulnerability. Of course, for every threat-scenario against a safe default a counter-argument can be made: package consumers have a software update process in place, the producer maintains the package and consumes security advice from elsewhere, and so forth. However, we question whether it is possible to design, before use, a secure default that can anticipate every possible use. Symmetry of ignorance between package consumer, producer and the sources of security advice means we cannot a priori provide a secure default for every possible use scenario and, therefore, there is a need to consider the design of a security default that supports design after design.

An interface by any other name. Another recommendation for developer-centered security could be an unambiguous naming convention for artifacts in producer interfaces, APIs, and so forth. A reliable naming scheme should ensure that interface artifacts can be referred to using globally distinct names, while avoiding attacks such as [51] which uses typo squatting and namespace manipulation to trick developers into using fake replacements for popular packages. Relying on a global body, such as ICANN, to decide names will not scale to the proliferation of software components and developers, and furthermore, a malicious producer can still mis-represent/ignore any naming conventions. One could consider using X.509 certificates to securely tie the names to real world entities (producer domains with public keys), however, based on past experiences [23], the required PKI would likely be unmanageable and not support usable-security. Moreover, given the transient nature of Internet domains and the declining numbers of system administrators relative to the number of Internet domains [30], we envisage that it would become impossible for consumers to maintain a consistent view of the relationships between producers and their interface names. Using a decentralized naming scheme such as SDSI [12] could provide a 'web of trust' for interface names, however, while attractive in theory, in practice SDSI naming has had little adoption over its twenty year history. Furthermore, all of these naming schemes are vulnerable to subterfuge attacks [21] whereby an attacker can confuse a consumer's understanding of a name by injecting malicious naming certificates.

These conventional naming schemes are designs for naming before use. Our intuition here is that it is akin to building naming mechanisms that uphold some safety property in an Alpern-Schneider [5] sense: it is known before use, that the mechanism will ensure that some predicate holds in every possible valid naming state. However, it is argued in [21] that naming is not a safety property, but a security property. Schneider [43] demonstrates that run-time mechanisms can only enforce safety properties with the observation that security mechanisms are (safe) approximations of security properties. Interpreting this result for a secure naming mechanism means that any naming mechanism is an approximation for secure naming and that potentially valid naming states that must be excluded from use by the mechanism. However, in designing the mechanism we cannot anticipate every possible use, before use, and therefore at some point it may be desired to revise the excluded states. This necessitates building a different mechanism (while approximating the same secure naming property). Thus, it does not support a design for design after design.

Our position is that the current user-centred security paradigms that support user-centered security through design for use before use are inadequate and that a new security paradigm is needed

that enables design for for design after design. Meta-design [18, 19], with its view of the system as a continuous collaborative development between stakeholders and technologies may provide a means to support design for design after design. Whether this view has application for security in contemporary systems is a topic for future research.

6 DEVELOPER CENTERED SECURITY RESEARCH SO FAR

NSPW has been the venue of choice for the earliest efforts to appropriately focus on Developer Centered Security. As [22] points out, the earliest work in usable security, [60], included the need to focus on developers. [20] was the first full paper at NSPW to focus on developers, exploring a way to increase developer awareness of security and risk tradeoffs at requirements design time. It the potential impact of a method and tool for security requirements identification. [56] calls for concentrating attention on the tools that provide security to developers. It is the first to call out the importance of usability of (security) APIs. It also calls for security mechanisms that do not require security expertise from developers who use them. In addition, it calls for integrated security solutions that do not require an opt-in; what we might call "secure by default" today. [52] brought the analogy of "point and shoot" to the security design space, mapping out three levels of abstraction in precision and control, and showing where there are gaps in the current tooling. [16] considered applying ethnographic tools such as document analysis, interviews, and participatory observation to technological infrastructures. Ethnoarchaeology may yield additional insights, marrying the ethnographic approach with archaeology, and emphasizing inspection of the artifacts alone (without access to the people and cultures producing them). We list these references here because considered developers activities and tools in the context of NSPW. None of these references delves into current practice as a foundation for consideration of the security of the creation and use of developer created artifacts, as this paper does.

There is a growing body of work on usable security and app developers. [7] surveyed over 200 app developers on resources they go to for security. The most common response was they searched online. Many consulted their friends and social network. [13] analyze apps for HTTPS vulnerabilities, setting up the notion that systemic code vulnerabilities can point to the need for better usable security for developers. [14] conduct interviews with developers of apps found to have HTTPS vulnerabilities, to evaluate their acceptance of proposed countermeasures. [2] used lab study, survey, and code analysis to show that API document produces more secure results but is harder to use than Stack Overflow. [1] conducted a controlled experiment of python developers across 5 cryptographic APIs. They find that simplified APIs can produce security results, but also need documentation, code examples, and extended functionality support for related features to ensure both functional correctness and security of the tasks. More strongly, easy to use documentation and available code examples seemed to compensate for more complex APIs in terms of functionality, though not security. Across the board, about 20% of the participants thought their code was secure when it was not. These results focus on individual app developers and their activities. We call out the complexities of large team, long term software development, which raises issues that we believe will need to be addressed in the developer-centered security research agenda.

[36] triangulates Stack Overflow posts, a survey, and GitHub code analysis to analyze developer use of crypto APIs. The APIs are seen as too low level. Yet they find that developers are (surprisingly) confident of their understanding of crypto concepts. [28] surveyed 59 software developers on their security API use and their preferences on security API abstractions. They find a tendency towards use of higher level abstraction APIs, and a good number of difficulties using the APIs available. These papers also focus on individual developers, not the broader organizational and team culture that they work in.

Microsoft's pioneering efforts in Secure Development Lifecycles made them also pioneers in recognizing the gaps in developer centered security and calling for and experimenting with approaches to close the gaps. [47]'s SOUPS keynote was a follow-on to Microsoft's experience with making threat modeling accessible to developers through the STRIDE structure [46]. [48] made the threat modeling interface both richer and gamified. [40] of Microsoft discussed the NEAT approach to producing security warnings (the presentation came complete with glasses for having your whiskey neat). This work focuses on the approaches they developed, while in this paper we concentrate on the challenges in the current software development environment and their implications for effective secure development techniques and tools (some of which are likely to have also been motivators of the previous work).

A number of works research whether a specific technique or tool can impact the secure development practices of developers. In the search for influencing developers to code securely, [58] studies 32 teams of masters students working on a realistically sized banking system. They found no detectable difference in security between teams with and without using security patterns. [53] Was a one-year study on the impact pen testing had on a software development team. Pen-tests improved developer awareness of security, but there were no reports of tangible changes in secure development practices. A stream of work out of UNCC looks at an IDE with interactive static analysis of security problems [50, 57]. Semi-structured interviews with students and professionals show that they can have impact, and inform the developer about security vulnerabilities. [37] finds that simply priming developers in context can make them more aware of potential vulnerabilities in code. There is not yet a study on how much priming would result in saturation/overload. These studies focus on a specific tool or technique and its impact, while we focus on existing practices likely to explain some of the lack of impact of some techniques, and limit the impact of others.

[54] looks at what impacts developers to code (more) securely. A Grounded Theory qualitative analysis of interviews with 12 developers discovered that developers relied on dialectics during the development lifecycle to help consider security. Participants were chosen for their experience with app security. Part of our thesis is that software security is impacted by many more developer activities than those undertaken by app developers knowingly working on app security.

7 CONCLUSION

In this paper we lay out the major changes in development culture, processes, and technology that we believe create a discontinuity in the scale and complexity of evaluating developer-centered security. This includes the continued developer expertise shift from depth to breadth and an unprecedented scale of code reuse. The pace of this change seems to be increasing over time. Engagement with these changes and better understanding of today's (and tomorrow's) software developer are essential pre-requisites to the much desired impact of studies on developer's security usability.

The assumption that developers are like end-users [47] has been an excellent first step, but like all analogies, it has its limitations. We agree that, just like end-users, they should not have to be omniscient in security, however, we argue that usability measures developed for users will not provide the same foundation and coverage for developers. There are many more individuals involved in the development process with different incentives and constraints. Consequently, studying this problem requires new methods. We argue that current approaches for studying and analyzing humans that use software will not scale for many aspects of developers. In some cases, we recommend adding to the current research agenda with familiar techniques from economics. We also see potential in fields of study of human activity that address complexity at scale and with an artifacts-first approach, such as archeology.

In addition to the challenge of how to conduct developer security usability studies, we need new approaches to support developer-centered security. User-centered security emphasizes design for use before use. Our position is that this does not directly translate to developer-centered security. The task of the development of secure code is a significantly more complex and recursive problem than secure usage of already developed software, and symmetry of ignorance means that the problem is not a simple partitioning of user and developer. The challenge is one of designing security for design after design.

ACKNOWLEDGEMENTS

The authors would like to thank Bill Andreas, Senior UX Designer at CA and amateur archaeologist, for his insights into the field of archaeology. We also thank our anonymous reviewers, workshop attendees, and our shepherds Heather Lipford and Kent Seamons, for their help in making this a better paper. Simon Foley's research is supported in the Cyber CNI Chair of Institute Mines-Télécom which is held by IMT Atlantique and supported by Airbus Defence and Space, Amossys, EDF, Orange, La Poste, Nokia, Société Générale and the Regional Council of Brittany; the Chair has been acknowledged by the French Centre of Excellence in Cybersecurity. Mary Ellen Zurko's work on this paper was done while she was an independent consultant.

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 *IEEE Symposium on Security and Privacy*.

[2] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky. 2016. You Get Where You're Looking for: The Impact of Information Sources on Code Security. In *2016 IEEE Symposium on Security and Privacy (SP)*. 289–305.

[3] Yasemin Acar, Sascha Fahl, and Michelle L. Mazurek. 2016. You are Not Your Developer, Either: A Research Agenda for Usable Security and Privacy Research Beyond End Users. In *Cybersecurity Development (SecDev)*.

[4] Yasemin Acar, Christian Stransky, Dominik Wermke, Charles Alexander Forbes Weir, Michelle Mazurek, and Sascha Fahl. 2017. *Developers Need Support, Too: A Survey of Security Advice for Software Developers.* IEEE, 22–26.

[5] B. Alpern and F.B. Schneider. 1987. Recognizing Safety and Liveness. *Distributed Computing* 2 (1987), 117–126.

[6] Adam Baldwin. 2015. A Malicious Module on npm. blog, https://blog.liftsecurity.io/2015/01/27/a-malicious-module-on-npm. (2015).

[7] Rebecca Balebako and Lorrie Cranor. 2014. Improving app privacy: Nudging app developers to protect user privacy. *IEEE Security & Privacy* 12, 4 (2014), 55–58.

[8] Frederick P. Jr. Brooks. 1975. *The Mythical Man-Month.* Addison-Wesley, Reading, Mass.

[9] Justin Cappos, Yanyan Zhuang, Daniela Oliveira, Marissa Rosenthal, and Kuo-Chuan Yeh. 2014. Vulnerabilities As Blind Spots in Developer's Heuristic-Based Decision-Making Processes. In *Proceedings of the 2014 New Security Paradigms Workshop (NSPW '14).* ACM, New York, NY, USA, 53–62.

[10] Lorrie Faith Cranor. 2008. A Framework for Reasoning About the Human in the Loop. , Article 1 (2008), 15 pages.

[11] Serge Egelman and Eyal Peer. 2015. The Myth of the Average User: Improving Privacy and Security Systems Through Individualization. In *Proceedings of the 2015 New Security Paradigms Workshop (NSPW '15).* ACM, New York, NY, USA, 16–28.

[12] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. 1999. SPKI Certificate Theory. RFC 2693 (Experimental). (Sept. 1999).

[13] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12).* ACM, New York, NY, USA, 50–61.

[14] Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. 2013. Rethinking SSL Development in an Appified World. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13).* ACM, New York, NY, USA, 49–60.

[15] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. 2012. Android Permissions: User Attention, Comprehension, and Behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security (SOUPS '12).* ACM, New York, NY, USA, Article 3, 14 pages.

[16] Laura Fichtner, Wolter Pieters, and André Teixeira. 2016. Cybersecurity As a Politikum: Implications of Security Discourses for Infrastructures. In *Proceedings of the 2016 New Security Paradigms Workshop (NSPW '16).* ACM, New York, NY, USA, 36–48.

[17] Barbara Filkins. 2016. *IT Security Spending Trends.* Technical Report. SANS Institute.

[18] G. Fischer, D. Fogli, and A. Piccinno. 2017. Revisiting and Broadening the Meta-Design Framework for End-User Development. In *New Perspectives in End User Development.* Kluwer Publishers, Dordrecht, The Netherlands.

[19] Gerhard Fischer and Thomas Herrmann. 2011. Socio-Technical Systems: A Meta-Design Perspective. *International Journal of Sociotechnology and Knowledge Development (IJSKD)* 3, 1 (2011), 1–33.

[20] Ivan Flechais, M. Angela Sasse, and Stephen M. V. Hailes. 2003. Bringing Security Home: A Process for Developing Secure and Usable Systems. In *Proceedings of the 2003 Workshop on New Security Paradigms (NSPW '03).* ACM, New York, NY, USA, 49–57.

[21] Simon N. Foley. 2013. Noninterference Analysis of Delegation Subterfuge in Distributed Authorization Systems. In *Trust Management VII - 7th IFIP WG 11.11 International Conference, IFIPTM 2013, Malaga, Spain, June 3-7, 2013. Proceedings.* 193–207.

[22] M. Green and M. Smith. 2016. Developers are Not the Enemy!: The Need for Usable Security APIs. *IEEE Security Privacy* 14, 5 (Sept 2016), 40–46.

[23] Peter Gutmann. 2014. *Engineering Security.*

[24] Almut Herzog and Nahid Shahmehri. 2007. User Help Techniques for Usable Security. In *Proceedings of the 2007 Symposium on Computer Human Interaction for the Management of Information Technology (CHIMIT '07).* ACM, New York, NY, USA, Article 11.

[25] H. M. Hinton. 1997. Under-specification, Composition and Emergent Properties. In *Proceedings of the 1997 Workshop on New Security Paradigms (NSPW '97).* ACM, New York, NY, USA, 83–93.

[26] H. M. Hinton. 1998. Composing partially-specified systems. In *Proceedings. 1998 IEEE Symposium on Security and Privacy (Cat. No.98CB36186).* 27–37.

[27] Erkki Huhtamo and Jussi Parikka. 2011. *Media archaeology: Approaches, applications, and implications.* Univ of California Press.

[28] Luigi Lo Iacono and Peter Leo Gorski. 2017. I Do and I Understand. Not Yet True for Security APIs. So Sad.. In *2nd European Workshop on Usable Security, EuroUSEC 2017.*

[29] M.A. Jackson. 1989. Getting It Wrong: A Cautionary Tale. In *JSP & JSD: The Jackson Approach to Software Development*, John Cameron (Ed.). IEEE CS Press.

[30] Daniel E. Geer Jr. 2012. Power. Law. *IEEE Security & Privacy* 10 (2012), 94–95.

[31] Tobias Lauinger, Abdelberi Chaabane, William Robertson, Christo Wilson, and Engin Kirda. 2017. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA*.

[32] Heather Richter Lipford and Mary Ellen Zurko. 2012. Someone to Watch over Me. In *Proceedings of the 2012 New Security Paradigms Workshop (NSPW '12)*. ACM, New York, NY, USA, 67–76.

[33] Kevin Montrose. 2016. Introducing Stack Overflow Documentation Beta. StackOverflow blog, https://stackoverflow.blog/2016/07/21/introducing-stack-overflow-documentation-beta/. (2016).

[34] B. A. Myers, A. J. Ko, T. D. LaToza, and Y. Yoon. 2016. Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools. *Computer* 49, 7 (July 2016), 44–52.

[35] Brad A. Myers and Jeffrey Stylos. 2016. Improving API Usability. *Commun. ACM* 59, 6 (May 2016), 62–69.

[36] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2016. Jumping Through Hoops: Why Do Java Developers Struggle with Cryptography APIs?. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 935–946.

[37] D. Oliveira, M. Rosenthal, N. Morin, K-C Yeh, J. Cappos, and Y. Zhuang. 2014. It's the Psychology Stupid: How Heuristics Explain Software Vulnerabilities and How Priming Can Illuminate Developer's Blind Spots. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC '14)*. ACM, New York, NY, USA, 296–305.

[38] Olgierd Pieczul and Simon N. Foley. 2015. The Dark Side of the Code. In *Security Protocols XXIII: 23rd International Workshop, Cambridge, UK, March 31 - April 2, 2015, Revised Selected Papers*, Bruce Christianson, Petr Švenda, Vashek Matyáš, James Malcolm, Frank Stajano, and Jonathan Anderson (Eds.). Springer International Publishing, Cham, 1–11.

[39] Olgierd Pieczul and Simon N. Foley. 2017. The Evolution of a Security Control. In *Security Protocols XXIV: 24th International Workshop, Brno, Czech Republic, April 7-8, 2016, Revised Selected Papers*, Jonathan Anderson, Vashek Matyáš, Bruce Christianson, and Frank Stajano (Eds.). Springer International Publishing, Cham, 67–84.

[40] Rob Reeder, E. Cram Kowalczyk, and Adam Shostack. 2011. Helping engineers design NEAT security warnings. In *Proceedings of the Symposium On Usable Privacy and Security (SOUPS), Pittsburgh, PA*.

[41] Ninlabs research. 2013. API Documentation. online. (2013). http://blog.ninlabs.com/2013/03/api-documentation/.

[42] H Rittel. 1984. *Developments in Design Methodology*. John Wiley & Sons, New York, Chapter Second Generation Design Methods, 317–327.

[43] Fred B. Schneider. 2000. Enforceable Security Policies. *ACM Trans. Inf. Syst. Secur.* 3, 1 (Feb. 2000), 30–50.

[44] Bruce Schneier. 2016. Stop Trying to Fix the User. *IEEE Security and Privacy* 14, 5 (Sept. 2016), 96–96.

[45] Charlotte Seager. 2015. Will learning to code help you get a job? Guardian Careers. (2015).

[46] A. Shostack. 2008. Experiences threat modeling at Microsoft. In *Workshop on Modeling Security (ModSec)*.

[47] Adam Shostack. 2010. Engineers are People Too. Proceedings of the Symposium On Usable Privacy and Security (SOUPS), keynote. (2010).

[48] Adam Shostack. 2014. Elevation of Privilege: Drawing Developers into Threat Modeling. In *2014 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 14)*. USENIX Association, San Diego, CA.

[49] Sathya Chandran Sundaramurthy, John McHugh, Xinming Ou, Michael Wesch, Alexandru G. Bardas, and S. Raj Rajagopalan. 2016. Turning Contradictions into Innovations or: How We Learned to Stop Whining and Improve Security Operations. In *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*. USENIX Association, Denver, CO, 237–251.

[50] Tyler W. Thomas, Heather Lipford, Bill Chu, Justin Smith, and Emerson Murphy-Hill. 2016. What Questions Remain? An Examination of How Developers Understand an Interactive Static Analysis Tool. In *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*. USENIX Association, Denver, CO.

[51] Nikolai Philipp Tschacher. 2016. *Typosquatting in Programming Language Package Managers*. Master's thesis. University of Hamburg.

[52] Sven Türpe. 2012. Point-and-shoot Security Design: Can We Build Better Tools for Developers?. In *Proceedings of the 2012 New Security Paradigms Workshop (NSPW '12)*. ACM, New York, NY, USA, 27–42.

[53] Sven Türpe, Laura Kocksch, and Andreas Poller. 2016. Penetration Tests a Turning Point in Security Practices? Organizational Challenges and Implications in a Software Development Team. In *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*. USENIX Association, Denver, CO.

[54] C. Weir, A. Rashid, and J. Noble. 2017. I'd Like to Have an Argument, Please: Using Dialectic for Effective App Security. In *2nd European Workshop on Usable Security, EuroUSEC 2017*.

[55] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. 2016. A Look at the Dynamics of the JavaScript Package Ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, USA, 351–361.

[56] Glenn Wurster and P. C. van Oorschot. 2008. The Developer is the Enemy. In *Proceedings of the 2008 New Security Paradigms Workshop (NSPW '08)*. ACM, New York, NY, USA, 89–97.

[57] Jing Xie, Heather Lipford, and Bei-Tseng Chu. 2012. Evaluating Interactive Support for Secure Programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '12)*. ACM, New York, NY, USA, 2707–2716.

[58] Koen Yskout, Riccardo Scandariato, and Wouter Joosen. 2015. Do Security Patterns Really Help Designers?. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 292–302.

[59] Mary Ellen Zurko. 2005. *Security and Usability*. O'Reilly, Chapter IBM Lotus Notes/Domino: Embedding Security in Collaborative Applications.

[60] Mary Ellen Zurko and Richard T. Simon. 1996. User-centered Security. In *Proceedings of the 1996 Workshop on New Security Paradigms (NSPW '96)*. ACM, New York, NY, USA, 27–33.