Developers Are Not The Enemy! The need for usable security APIs

Matthew Green, Johns Hopkins University
Matthew Smith, University of Bonn, Fraunhofer FKIE

October 2016

1 Abstract

Usability problems are a major cause of many of today's IT security incidents. Despite the critical importance of securing information systems, the security mechanisms we deploy often prove too complicated, time-consuming and error-prone when placed into the hands of users. This problem has motivated the field of *usable security*. For over a decade, researchers in this field have attempted to combat these problems by conducting interdisciplinary research focusing on the root causes of the end-user problems and on the creation of security mechanisms that are compatible with ordinary users.

However, many recent security incidents were not caused by end-users, but rather by software developers making mistakes. Unfortunately, while it has become accepted that systems should be user-friendly and robust to the end-user, the prevailing attitude towards software developers amongst cryptographer library designers is that they are experts and thus should know better. This attitude persists even when considering complex areas such as cryptography and user authentication, where the typical software engineer cannot be expected to possess the domain expertise necessary to navigate all pitfalls. Unfortunately, rather than recognizing the limitations of software engineers, modern security practice has created an adversarial environment between the designers of security software and the developers who use this software to construct applications. In this article we argue that *developers are not the enemy*, and that – to strengthen security systems across the board – security professionals need to re-focus their efforts on creating developer-friendly and developer-centric approaches to assist these professionals in their complex tasks.

To illustrate our thesis, we focus on the usability of cryptographic and other security related Application Program Interfaces (APIs). Over the past several years a number of new cryptographic libraries and APIs have become available to developers. These libraries promise to greatly increase the use of cryptography on the web and in the cloud, but they often do so at a cost. In this article we propose several characteristics we believe will lead to more usable security APIs that treat normal developers, rather than cryptographers, as the primary consumer.

2 Introduction

IT security mechanisms are failing to keep pace with the threats they are facing, which increasingly exposes our systems and critical infrastructure to attacks. These failures are wide-ranging and affect home users, enterprises and governments alike. A study conducted by McAfee, Intel, and the Center for Strategic and International Studies estimates the global cost of cybercrime at around $400 billion per year. The reasons for these failures can be classified broadly into two categories: technical failures and human error.

For a long time security research has focused exclusively on the technical side of the problem, viewing the human user as "the weakest link in the chain". Recently, the research domain of usable security and privacy has developed in order to take a different stance. This field argues that *technology should adapt to its users* rather than requiring users to adapt to technology. Three seminal papers are seen as the origin of this school of thought. Zurko and Simon's: "User-Centered Security" [15], Adams and Sasse's: "Users Are Not the Enemy" [1] and Whitten and Tygar's "Why Johnny Can't Encrypt: A Usability Evaluation of

PGP 5.0" [14] all argued that users should not be seen as the problem to be dealt with, but that security experts need to communicate more with users, and adopt user-centered design approaches. To address this, the field of Usable Security and Privacy has developed to study end-user behaviour, perceptions, problems and wishes. Researchers in this field inform system administrators and software developers of the results and make concrete suggestions as to how developers and administrators could make their software and services more usable for the end-user. A classic example of usable security research is the study of users' password behaviour, producing recommendations on how administrators should set password policies so that users are encouraged to create strong yet memorable passwords.

While there are many interesting and worthwhile research questions to be answered by studying end-users and despite the fact that the earliest work in this domain [15] called for support for all involved actors,[1] current research almost entirely discounts the fact that *administrators and software developers themselves also make mistakes* and need help just as much, if not more, than end-users. Critically, while end-users normally only endanger themselves, mistakes made by administrators or developers endanger all those who rely on their work. To make matters worse, it is extremely difficult for system administrators to defend against mistakes made by developers, or for end-users to defend themselves against mistakes made by developers or administrators.

To illustrate the latter point, consider several of the most catastrophic security incidents of the past several years. Many of these incidents are not caused by failures on the part of end-users, but are the result of human error on the part of software developers and other experts. For example, in 2014 the Heartbleed and Shellshock vulnerabilities led to Internet-wide patch cycles, and brought matters of open source security into the public conversation. Each of these vulnerabilities was caused by an individual developer, and led to vulnerabilities that affected half of the Internet – and millions of users. Similarly, vulnerabilities on the part of mobile application developers led to the distribution of rootkits to million of Apple users in China,[2] and to the deployment of vulnerable cryptography in thousands of mobile applications [7, 3].

These examples clearly demonstrate that developers are not free from human error. Despite having a much higher level of expertise and training than average end-users, the tasks developers have to deal with are also significantly more complex than those faced by the average end-user. It is therefore understandable that these experts make mistakes. Nonetheless, a recent overview of the research domain by Garfinkel and Lipford highlights the fact that very little work has been done into researching the human factors involving these actors [6]. To make matters worse, the current common attitude towards developers is similar to that faced by end-users when the usable security research domain was created. Namely, that *"Developers are the weakest link and they have to do a better job and make fewer mistakes".* In this work we challenge this concept, by extending the seminal USEC expression "Users Are Not The Enemy" with the statement "Developers Are Not The Enemy Either."

Specifically, we argue there are many areas where developers could benefit from support, e.g. safer programing languages, more usable security libraries, better security testing tools, and so on. In this article we will focus on area in which we believe a great amount of benefit can be generated for relatively little effort. One of the main tasks of developers is to write program code and it is common practice to integrate security code through the use of application programming interfaces (APIs). We argue that improving the usability of these APIs is a first and important step towards creating more developer friendly security.

To make our argument more concrete, in this article we will focus on a specific form of security API: the interface to cryptographic library software. Cryptographic libraries are collections of software routines that provide cryptographic operations such as encryption, digital signing, and secure connection establishment. We focus on cryptographic libraries for two specific reasons. First, cryptographic libraries appear to be uniquely prone to misuse by developers; as we illustrate below, even subtle misuse will often produce a catastrophic security failure. Moreover, due to the more frequent deployment of cryptography within applications, these libraries are increasingly being adopted by software developers who do not possess domain expertise in cryptography. These two factors have driven a number of recent, high-profile security

[1] Indeed, Zurko and Simon [15] specifically called out the need to focus on developers; however, this recommendation has not been consistently followed in later work, with some notable exceptions [5].

[2] See e.g., the XCodeGhost malware that was delivered to millions of users due to application developers downloading a malicious distribution of Apple's developer tools.

failures [7, 3].

2.1 A brief overview of Cryptographic APIs

While cryptography has a long history in military and specialised industry usage, the widespread deployment of cryptographic software dates to the 1990s. This period saw the development of open source tools such as Zimmermann's Pretty Good Privacy (PGP), and several early cryptographic libraries including Guttman's cryptlib and Young's SSLeay library, the latter of which forms the basis for modern SSL/TLS libraries such as RSA BSAFE, OpenSSL and Google's BoringSSL.

Shortly after these libraries were released, library authors began to observe a trouble pattern of algorithm misuse among the library's consumers. For example, several years after releasing cryptlib, Peter Guttman noted that "most crypto software is written with the assumption that the user knows what they're doing, and will choose the most appropriate algorithm and mode of operation, carefully manage key generation and secure key storage, employ the crypto in a suitably safe manner, and do a great many other things that require fairly detailed crypto knowledge. However, since most implementers are everyday programmers .. the inevitable result is the creation of products with genuine naugahyde[3] crypto." More succinctly, Guttman concluded that "the determined programmer can produce snake oil using any crypto tools" [8].

Despite these early warnings, modern cryptographic software has learned surprisingly little from the lessons of the past. By and large, today's cryptographic APIs still require software developers to possess a high degree of domain expertise in cryptography. Indeed, this warning is sometimes broadcast by the software itself: as a warning to developers, the entirety of the usable W3C Web Crypto API is embedded within in a namespace entitled SubtleCrypto.

2.2 Examples of the cost of ignoring developers

Despite the good intentions of library authors, the strategy of warning and educating software developers appears thus far to be largely a failure. Several recent large-scale analyses of deployed applications have shown cryptographic library use, and *misuse* to be widespread [7, 3], resulting in consequences ranging from denial of service to total loss of security. Here we give a couple of high-level examples of the more serious examples.

Failure to validate TLS certificates. Numerous applications rely on the Transport Layer Security (TLS) protocol to secure network communications with websites and back-end services. Authentication in TLS relies on certificates, which must be carefully verified to ensure that the remote party is the expected participant. Unfortunately, recent large-scale studies [7, 3] show that significant percentage of applications fail to properly validate certificates, which potentially unwinds the security of TLS. Detailed analysis shows that many of the vulnerabilities are the resulting of confusing cryptographic library APIs and permissive security settings misused by developers.

Use of insecure encryption modes. Many cryptographic libraries provide complex encryption APIs that require expertise on the part of developers, and are correspondingly easy for developers to misuse – often to catastrophic effects. This has led to a number of incidents in which insecure modes were used, allowing information leakage and other failures.

Use of insecure random number generators. Security protocols often rely on the availability of unpredictable random numbers. These must be provided via a cryptographically-secure pseudorandom number generator. However, many applications persist in using insecure or improperly-seeded random generators, often to the detriment of application security.

[3] Naugahyde is a brand of artificial leather

3 Usable crypto APIs

Over a decade's worth of usable security research has shown us that creating usable security mechanism is preferable to educating end-user in the use of complex and confusing software. Despite this, it remains common to blame developers for mistakes resulting from library misuse. This stems from a sentiment that holds that software developers are professionals and therefore should not require help. However, the primary objective of most developers – like most end-users – is not security and thus their capacity, interest and skill to deal with security issues is limited and they would benefit from more usable APIs.

Outside the field of cryptography, there is currently a great deal of effort being put into the usability of APIs. Entire books have been written about the topic [10, 13, 9], and a number of lightweight heuristics have been promulgated. For example, Joshua Bloch [2] offered seven rules for software APIs. These include simple recommendations, such as the ease of learning an API, making APIs that are difficult to misuse, and powerful enough to satisfy requirements, among others.

Bloch's rules offer good general advice, however they do not fully address the specific pitfalls of security/crypto APIs. In the remainder of this section we adapt and extend the Bloch heuristic to tailor it to the case of crypto APIs, and discuss examples of where current APIs are getting this wrong. Wherever possible we also present examples of APIs that exemplify our recommendations. Sadly, we did not find many examples in the latter category.

3.1 Ten Principles for creating usable, secure crypto APIs

In this section we outline our core recommendation. This consists of ten principles for constructing usable, secure crypto APIs. Many of these principles draw on and extend corresponding principles of Bloch [2] with specific application to cryptography APIs. In the following sections we will elaborate on these principles, and provide examples of how they can enhance the security of cryptographic library software.

Integrate crypto functionality into standard APIs so regular developers do not have to interact with crypto APIs in the first place We believe that this first rule should be considered the golden rule for cryptographic integration. Investing the extra effort to integrate crypto APIs into non-security related APIs, in such a way that the regular developer does not have to interact with the crypto API at all, goes a long way to ensuring the the crypto code is used and is used correctly.

A good example for this is the `URL` class of the `Java.net` API. When using this API, all the developer needs to do to create a TLS-secured connection is to pass a `https` URL to the `URL` class. As long as the developer is satisfied with the default behaviour not a single line of security related code needs to be added to use a secure connection. This solution does not solve all security problems – we will see later on that this approach still has pitfalls – but the general idea is the right one.

A negative example in this category is the secure storage of passwords. None of the popular programming languages or frameworks offer a transparent mechanism to securely store (e.g., hash or encrypt) user passwords. In nearly every environment, application developers must write code to select the hashing algorithms, create the salt, choose encryption strength and tie the resulting construction together properly. A popular tutorial on how to securely store passwords using Java [4] consists of 40 lines of code, in which several critical design choices are made, such as selecting `PBKDF2WithHmacSHA1` as the hashing algorithm, setting the key length to 160 bit, setting the number of iterations to 20,000, setting the random number generator to `SHA1PRNG` and setting the length of the salt to 64 bit. The ramifications of these setting are likely not understood by the majority of developers who copy and paste this code into their applications.[5] While adding this functionality to standard storage classes is not without risk, we argue that these risks are outweighed by the risks of forcing developers to deal with this themselves.

[4] http://www.javacodegeeks.com/2012/05/secure-password-storage-donts-dos-and.html

[5] And this assumes that developers find a valid tutorial in the first place. Several tutorials have recommended that users simply encode passwords using "Base64 encryption", an encoding scheme that provides no security whatsoever. See e.g., https://3v4l.org/BZ7rC

Sufficiently powerful to satisfy both security and non-security requirements It is important that designers of security APIs remember that most developers using their API will do so because they ultimately have a *non-security* goal with security requirements in mind. The API should be designed in such a way that it is powerful enough to satisfy all the requirements. The `URL` example from above provides an good example of a good API that did not provide sufficient capabilities. While the integration of the `https` logic into the `URL` class is exemplary, the API was not powerful enough to fulfil all the developers requirements - particularly the non-security requirements. In conducting a large scale analysis of `https` code in Android [3], we found widespread misuse of the `https` functionality that endangered millions of users.[3] In follow up work we interviewed developers, and discovered that the root cause for the insecure code was that the API was not powerful enough to fulfil the developers requirements. This forced the developers to improvise and write their own code, which introduced the vulnerabilities. Interviewing just a handful of developers provided a list of only 4 simple requirements, which if met, would have been capable of preventing all the vulnerabilities we found.[4] This kind of interview-based gathering of requirement is something we highly recommend during the development of APIs. Ensuring that the API is powerful enough out of the box to fulfil the developers'requirements is the best way to stop them tinkering and breaking something.

Easy to learn, even without cryptographic expertise It is unrealistic to expect non-crypto experts to understand the cryptographic background of the APIs they are using. Consequently APIs should strive to be designed in a way that developers do not need to understand the background in order to correctly use the API. Carrying on the `https` example from above, in the user study we conducted to ascertain the root causes of insecure use of the `https` functionality we interacted with one developer who had made a coding error concerning the verification of SSL certificates. The error led to the code accepting any valid certificate, making it vulnerable to active man-in-the-middle attacks. When we informed the developer of the vulnerability, the developer did not accept there was a problem, because he inspected the traffic of his app using Wireshark which showed him, that the traffic was encrypted. The developer did not have the necessary expertise to understand that the nature of the attack meant that, even thought the traffic was encrypted, it might be encrypted under the key of an attacker and thus not secure. The password example from above is another example where most developers lack the knowledge needed to make the right choices. Few outside of the cryptographic community will understand the differences between MD5, SHA1, SHA512 or PBKDF2WithHmacSHA1 that are some of the options Java developers can choose from. It is desirable that developers should not need to have a background in the cryptographic protocols they are using. This should be hidden by the API.

Don't break the developer's paradigm Developers often have a limited understanding of cryptography that is limited to certain paradigms. These typically include secret key encryption, public key encryption, and other functions such as digital signatures. When using a public key encryption scheme, it is expected that developers will understand that they must obtain the public key of the user to encrypt to, and then employ an algorithm to transform messages into ciphertext. However, some recent attempts like the NaCL library — that are primarily aimed at providing a simple and misuse-resistant API to software developers — take a different approach that requires both a secret key of the user *and* a public key of the receiver. While the resulting function is secure enough as written in NaCl, the resulting function has led to the development of entirely new adaptations of NaCl that provide the required public-key only functionality. It is unclear whether the community is harmed from such additional implementations, but the confusion resulting from such alternatives may lead to insecure results.

Easy to use, even without documentation Developers like end-users do not like to read manuals before getting started. If the API is not self-explanatory or worse gives the false impression that it is, developers will make dangerous mistakes. A good example of where this went wrong is the the OpenSSL error handling code. The following quote is taken from the official documentation:

"Most OpenSSL functions will return an integer to indicate success or failure. Typically a function will return 1 on success or 0 on error. All return codes should be checked and handled as appropriate.

```
if (some_verify_function())
    /* signature successful */
```

Figure 1: Intuitive but incorrect error checking code.

```
if (1 != some_verify_function())
    /* signature successful */
```

Figure 2: Correct error checking code.

Note that not all of the libcrypto functions return 0 for error and 1 for success. There are exceptions which can trip up the unwary. For example if you want to check a signature with some functions you get 1 if the signature is correct, 0 if it is not correct and -1 if something bad happened like a memory allocation failure."

It is easy to see that without reading the documentation it is very likely that developers will handle errors incorrectly. A very common way to handle errors is shown in figure 1. However, due to the idiosyncratic nature of OpenSSLs error handling code the correct way to check for errors is shown in figure 2. This tripped up the OpenSSL developers and led to CVE-2008-5077.

Following on from the password example above, the choice of algorithm is made by passing a string value to a factory method. While this offers a lot of flexibility, it also forces the developer to consult the documentation to find out which values are valid. The API documentation for the `SecretKeyFactory` class does not even list the options, but refers to the `Java Cryptography Architecture Standard Algorithm Name Documentation`[6]. This documentation offers an exhaustive list of all cryptographic algorithm names, but gives little guide on when and where to use them leaving the entire burden on the developer. It is little surprise that so many password database compromises still turn up plain-text or poorly hashed passwords.

Hard to misuse. Incorrect use should lead to visible errors. One of the most important differences between security and non-security APIs is that errors made when using a non-security API will be more likely to impede visible functionality and thus be caught during regular testing. Security errors are often invisible and are only caught through adversarial testing - which is harder and more time consuming to execute than regular testing. Consequently security APIs should pay special attention to preventing incorrect use and making error visible. Our running `https` example is ideal for showing this problem. Many developers who ended up deploying insecure apps did so because they were confronted by a SSL verification exception which prevented their app from opening a `URL` connection for security reasons. The reason for the exception was that the endpoint they were connecting to did not have a certificate that validated under Androids roots-of-trust. They dealt with this by using a number of different `TrustManager` implementations - all of which turned validation off entirely. Based on our interviews with the developers some did so without realising that this fix not only made the exception go away but also seriously weakened the security of their app. Other did so knowingly, usually because they wanted to use self-signed certificates during development and forgot to remove the testing code before deploying their apps. In both cases the code compiles without any errors and the apps also run without any problems. The incorrect use of the `TrustManager` only becomes apparent during adversarial testing. While it will be hard to create security APIs which cannot be misused API designers should make it as hard as possible to do so. This can be done in several ways. Firstly, if the guidelines mentioned above are following the correct use should intuitively present itself to the developer and custom tinkering which often leads to errors is not necessary. Secondly, it is recommended that API developers create

[6] http://docs.oracle.com/javase/7/docs/technotes/guides/security/StandardNames.html

failsafes that implement checks wherever possible to catch unsafe use of the API and cause visible errors.

A related issue is that *it should be hard to circumvent security related errors.* Our running `https` example again serves to illustrate this. It was very easy for developers to circumvent the SSL validation error by turning validation off entirely. Critically it was possible to circumvent the security part while still proceeding with the primary part, i.e. opening the Internet connection. While it might be desirable to have this option, it should require a reasonable amount of effort and the security implications must be made exceedingly clear to the developer. If in any way possible it should be less desirable to circumvent the error than to fix the cause. With one noteable exception. We suggest that security APIs offer a testing/development mode, in which errors can be circumvented easily.

Defaults should be safe and never ambiguous. A fundamental principle of usable security is to avoid providing users with default behaviors that are either ambiguous or unsafe. Unfortunately this lesson has not been universally adopted in the setting of cryptographic APIs. A simple example of this phenomenon can be seen in the Java API for performing symmetric encryption. When instantiating the `javax.crypto.Cipher` class in the Java security API, the user is invited to specify a detailed description of cipher name and mode of operation and padding scheme. Alternatively, the user can simply specify the cipher name (e.g., `AES`) and accept the defaults given by the underlying implementation. This second usage is quite common, as indicated by searches of open source code repositories. The challenge in this API is that there are many implementations of these ciphers, provided by different Java Cryptography Extension (JCE) Providers. In some implementations, the default block cipher mode is ECB, a mode of operation that is well known to leak information about the structure of repetitive or low-entropy plaintexts. While the need for safe and unambiguous defaults seems obvious, experience shows that it cannot be emphasized enough.

Testing Mode. Security mechanisms often come with a complexity or performance penalty. While this is a necessary trade-off in production use, it is legitimate and understandable if developers wish to be able to test their software with reduced or no security for convenience. No security API we know of caters to this wish, which leads to developers having to write custom code to turn off security features during development. Our interviews with Android and iOS developers brought up several cases where such testing code was forgotten and apps got deployed containing the testing code that turned security off entirely. Thus, we recommend that security and crypto API offer dedicated support for development and testing purposes. Another example where this would be beneficial is the way encrpytion APIs deal with initialisation vectors. Many cryptographic algorithms require that an initialisation vector (IV) with a secure random number be used. Since it is critical that this number not be predictable or worse constant it begs the question why APIs allow the developers to use weak random number generators or even constants. The answer is that for testing purposes this can make sense. If a developer needs to check the output of a given method this would be made more difficult if randomness is involved. However, the security implications of enabling this are grave and many serious vulnerabilities have been introduced by developers choosing bad IVs. Including a dedicated testing mode can help mitigate these problems. So as not to make the testing mode into an easy way to circumvent errors in a production environment, it is advisable to tie the testing mode to specific machine IDs so if code accidentally gets deployed in testing mode it will lead to visible error on deployment.

Easy to read and maintain code that uses it/Updatability Bloch recommends that API designers should try and ensure that it is easy to maintain code that uses the API. In the security context this is particularly important since it is essential that security code be kept up to date. While programs that utilise out dated non-security APIs age more or less gracefully - programs that use out dated security APIs become a liability. Take the password example from above. The tutorial referenced was published in 2012. None of the security settings in that tutorial would be considered recommendable today - only three years later. It is particularly critical that security parameters such as the number of iterations of the hashing algorithm are set by the developer - who will usually hard code them. Since

algorithms and parameters change frequently due to vulnerabilities being discovered and to counter the increased capabilities of attackers this is a suboptimal solution, since it forces all developers to update their code on a regular basis. We argue that API design should plan ahead for these easily foreseeable updates and make life as easy as possible for developers to update their software. One simple solution to this is to use configuration files instead of hard coding security parameters, however it might be worth considering whether security APIs need to be more pro-active and offer security updates similar to operating systems.

Assist with/handle end-user interaction It is fairly common for security APIs to provide functionality that in some circumstances will terminate with an error for security reasons, e.g. in the presence of an attack - or more likely a misconfiguration. In these cases it is also fairly common for this error to be passed up through the layers of the software to the end-user who has to decide how to proceed. Both these common occurrences lie in the nature of security code. However, it is also common for APIs to leave the entire burden of developing reaction strategies to the developer using the API, e.g. they have to develop the case statements and more critically the error messages and options to interact with the end-user. This is a bad state of affairs for several reasons. Firstly, most developers using a security API do not have a firm grasp on the cryptographic or security background and thus would be hard pressed to explain to the end-user what went wrong. Secondly, even with a solid understanding of what went wrong, designing good warning messages is an art to itself. For instance all the browser vendors have been tweaking and improving their SSL warning messages and have had entire teams working on the issue. Usable security researcher have also worked on this problem area extensively [12, 11] However, SSL APIs offer no assistance. Every app or software developer must invest the effort to design their own warning messages. This fact was bemoaned by several of the Android and iOS developers we interviewed. While the general purpose nature of APIs makes designing the actual UI infeasable in most cases, offering ready made explanation texts and interaction recommendations in the API for developers to build on would be a very sensible feature to build into APIs.

4 Conclusions

In this work, we examined the paradigm of *developer friendly security* and considered the implications of this paradigm for a specific class of applications: cryptographic library software. We further proposed a series of heuristics intended to aid library developers in producing software that will minimize the possibility of developer misuse. We believe that implementing these proposals across many large-scale cryptographic APIs will produce a dramatic improvement in security with minimal effort.

References

[1] Anne Adams and Martina Angela Sasse. Users are not the enemy. *Communications of the ACM*, 42(12):40–46, 1999.

[2] Joshua Bloch. *How to Design a Good API and Why it Matters*. LCSD Keynote, 2005.

[3] Sascha Fahl, Marian Harbach, Marian Harbach, Thomas Muders, Matthew Smith, Lars Baumgärtner, and Bernd Freisleben. Why Eve and Mallory love Android: An analysis of android SSL (in)security. In *CCS '12: Proceedings of the 2012 ACM conference on Computer and communications security*. ACM Request Permissions, October 2012.

[4] Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. Rethinking SSL development in an appified world. pages 49–60, 2013.

[5] Ivan Flechais, M. Angela Sasse, and Stephen M. V. Hailes. Bringing security home: A process for developing secure and usable systems. In *Proceedings of the 2003 Workshop on New Security Paradigms*, NSPW '03, pages 49–57, New York, NY, USA, 2003. ACM.

[6] S Garfinkel and H Lipford. Usable Security: History, Themes, and Challenges. *10.2200/S00594ED1V01Y201408SPT011*.

[7] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: validating SSL certificates in non-browser software. In *CCS '12: Proceedings of the 2012 ACM conference on Computer and communications security*. ACM Request Permissions, October 2012.

[8] Peter Gutmann. Lessons learned in implementing and deploying crypto software. In *Proceedings of the 11th USENIX Security Symposium*, pages 315–325, Berkeley, CA, USA, 2002. USENIX Association.

[9] Kirsten L. Hunter. *Irresistible APIs: Designing Web APIs That Developers Will Love*. Manning, 2016.

[10] Martin Reddy. *API Design for C++*. Academic Press, 2011.

[11] Andreas Sotirakopoulos, Kirstie Hawkey, and Konstantin Beznosov. On the Challenges in Usable Security Lab Studies: Lessons Learned from Replicating a Study on SSL Warnings. In *the Seventh Symposium*, pages 1–18, New York, New York, USA, 2011. ACM Press.

[12] J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L.F. Cranor. Crying wolf: An empirical study of SSL warning effectiveness. *Proceedings of the 18th conference on USENIX security symposium*, pages 399–416, 2009.

[13] Jaroslav Tulach. *Practical API Design: Confessions of a Java Framework Architect*. Apress, 2012.

[14] A. Whitten and J.D. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. *Proceedings of the 8th USENIX Security Symposium*, 99, 1999.

[15] Mary Ellen Zurko and Richard T Simon. User-centered security. In *NSPW '96: Proceedings of the 1996 workshop on New security paradigms*. ACM, September 1996.