Developers Are Not the Enemy!

The Need for Usable Security APIs

Matthew Green | Johns Hopkins University
Matthew Smith | University of Bonn and Fraunhofer FKIE

Modern security practice has created an adversarial relationship between security software designers and developers. But developers aren't the enemy. To strengthen security systems across the board, security professionals must focus on creating developer-friendly and developer-centric approaches.

IT security mechanisms are failing to keep pace with the threats they face, increasingly exposing our systems and critical infrastructures to attacks. These failures are wide ranging and affect home users, enterprises, and governments alike. A 2014 study conducted by McAfee, Intel, and the Center for Strategic and International Studies estimates cybercrime's global cost to be US$400 billion per year.[1] The reasons for these failures can be classified broadly as either technical failures or human error.

For a long time, security research focused exclusively on the problem's technical side, viewing the human user as "the weakest link in the chain." However, the relatively new research domain of usable security and privacy takes a different stance: technology should adapt to its users rather than require users to adapt to technology. Three seminal papers—Mary Ellen Zurko and Richard Simon's "User-Centered Security,"[2] Anne Adams and M. Angela Sasse's "Users Are Not the Enemy,"[3] and Alma Whitten and J.D. Tygar's "Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0"[4]—originated this school of thought. All argued that security experts shouldn't see users as problems to be dealt with; rather, they must communicate more with users and adopt user-centered design approaches.

The Developers' Role in Usable Security and Privacy

The usable security and privacy field studies end-user behavior, perceptions, problems, and wishes. Its researchers inform system administrators and software developers of the results and make concrete suggestions as to how developers and administrators can make their software and services more functional for end users. A classic example of usable security research is the study of users' password behavior, which has produced recommendations on how administrators should set policies that enable users to create strong yet memorable passwords.

There are many interesting, worthwhile research questions to be answered by studying end users. However, despite the earliest work in this domain calling for support for all involved actors—particularly developers[2]—current research almost entirely discounts the fact that administrators and software developers also make mistakes and need help as much, if not more, than end users (Ivan Flechais and his colleagues' work is one notable exception.[5]) Critically, whereas end users normally only endanger themselves, if administrators or developers make mistakes, they endanger all

 Copublished by the IEEE Computer and Reliability Societies

those relying on their work. To make matters worse, it's extremely difficult for system administrators to defend against developer mistakes, or for end users to defend against developer or administrator mistakes.

Consider several recent catastrophic security incidents. Many of these incidents weren't caused by failures on the part of end users but as the result of human error on the part of software developers and other experts. For example, the Heartbleed and Shellshock vulnerabilities led to Internet-wide patch cycles in 2014 and brought open source security into the public conversation. Each of these vulnerabilities was caused by an individual developer and affected half the Internet—and millions of users.[6] Similarly, vulnerabilities caused by mobile application developers led to the distribution of rootkits to millions of Apple users in China (for example, the XCodeGhost malware delivered to users as a result of application developers downloading malicious Apple developer tools) and the deployment of vulnerable cryptography in thousands of mobile applications.[7,8]

These examples clearly demonstrate that developers aren't free from human error. Although developers have a much higher level of expertise and training than average end users, they deal with tasks that are significantly more complex than those faced by end users, so mistakes are understandable. Nonetheless, Simson Garfinkel and Heather Lipford's recent review highlights the lack of research on the human factors involving these actors.[6] Moreover, today's attitude toward developers reflects the previous attitude toward users when the usable security research domain was created: namely, that developers are the weakest link and must do a better job and make fewer mistakes.

We challenge this concept by extending the seminal usable security expression "users are not the enemy" with the statement "developers are not the enemy either." Specifically, we argue that developers could benefit from support in many areas, including safer programming languages, more usable security libraries, and better security testing tools. In this article, we focus on application programming interfaces (APIs), an area in which a great amount of benefit can be generated for relatively little effort. A key task for developers is writing program code, and these developers commonly integrate security code through the use of APIs. Improving these APIs' usability is a first, important step toward creating more developer-friendly security.

A Brief Overview of Cryptographic APIs

To make our argument more concrete, we 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 reasons. First, cryptographic libraries appear to be uniquely prone to misuse by developers; as we'll illustrate, even subtle misuse can produce a catastrophic security failure. Second, cryptography is increasingly deployed within applications, but the application developers adopting these libraries often don't possess domain expertise in cryptography. These two factors have driven several recent, high-profile security failures.[7,8]

Although cryptography has long been used in the military and specialized industry, widespread deployment of cryptographic software dates only to the 1990s. This period saw the development of open source tools such as Philip Zimmermann's Pretty Good Privacy (PGP) and early cryptographic libraries such as Peter Guttman's cryptlib and Eric Andrew Young's SSLeay library. SSLeay is the basis for modern SSL/Transport Layer Security (TLS) libraries such as RSA BSAFE, OpenSSL, and Google's BoringSSL.

Shortly after these libraries were released, their authors began to observe a troubling pattern of algorithm misuse among the libraries' consumers. For example, several years after releasing cryptlib, 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 [an artificial leather brand] crypto." Guttman succinctly concluded that "the determined programmer can produce snake oil using any crypto tools."[9]

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

The Cost of Ignoring Developers

Despite library authors' good intentions, their strategy of warning and educating software developers appears thus far to be largely a failure. Recent large-scale analyses of deployed applications have shown cryptographic library use—and misuse—to be widespread,[7,8] with consequences ranging from denial of service to total loss of security. Here we give some high-level examples of the more serious cases.

- *Failure to validate TLS certificates.* Numerous applications rely on the 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 show that a significant percentage of applications fail to properly validate certificates,[7,8] which potentially unwinds the security of TLS. Detailed analysis shows that many vulnerabilities are due to 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 with catastrophic effects. This has led to numerous 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, which are provided via cryptographically secure pseudorandom number generators. However, many applications persist in using insecure or improperly seeded random generators, often to the detriment of application security.

Usable Crypto APIs

More than a decade's worth of usable security research has shown us that creating usable security mechanisms is preferable to educating end users in the use of complex, confusing software. Despite this, developers are still commonly blamed for mistakes resulting from library misuse. This stems from the sentiment that software developers are professionals and therefore shouldn't require help. However, the primary objective of most developers—like most end users—isn't security; thus, developers have limited capacity, interest, and skills for dealing with security issues and would benefit from more usable APIs.

Outside the field of cryptography, a great deal of effort is being put into API usability. Entire books have been written about the topic,[10–12] and several lightweight heuristics have been promulgated. For example, Joshua Bloch offered seven rules for software APIs, including simple recommendations for making it easier to learn APIs, as well as for making APIs that are difficult to misuse yet powerful enough to satisfy requirements.[13]

Bloch's rules offer good general advice; however, they don't fully address the specific pitfalls of security/crypto APIs. Thus, we've adapted and extend the Bloch heuristic,[13] tailoring it to the case of crypto APIs, and discuss examples of where current APIs are getting this wrong. Wherever possible, we present APIs that exemplify our recommendations, although sadly we didn't find many examples.

Ten Principles for Creating Usable and Secure Crypto APIs

Our core recommendation consists of 10 principles for constructing usable, secure crypto APIs:

- integrate crypto functionality into standard APIs so regular developers don't have to interact with crypto APIs in the first place;
- design APIs that are sufficiently powerful to satisfy both security and nonsecurity requirements;
- make APIs easy to learn, even without cryptographic expertise;
- don't break the developer's paradigm;
- make APIs easy to use, even without documentation;
- make APIs hard to misuse, with incorrect use leading to visible errors;
- ensure that defaults are safe and never ambiguous;
- include a testing mode;
- ensure that APIs are easy to read, and that it's easy to maintain the code that uses them (updatability); and
- allow APIs to assist with and/or handle end-user interactions.

Integrate crypto functionality into APIs. This should be the golden rule for cryptographic integration. Investing the extra effort to integrate crypto APIs into nonsecurity-related APIs—in such a way that regular developers don't have to interact with the crypto API at all—goes a long way to ensuring that the crypto code is not only used but used correctly.

A good example is the URL class of the Java.net API. When using this API, all developers must do to create a TLS-secured connection is pass an https URL to the URL class. As long as the developers are satisfied with the default behavior, they don't need to add a single line of security-related code to use a secure connection. This solution won't solve all security problems—we'll discuss some pitfalls later—but the general idea is right.

A negative example in this category is secure password storage. None of the popular programming languages or frameworks offer a transparent mechanism to securely store (for example, 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 securely storing passwords using Java (www.javacodegeeks.com/2012/05/secure-password-storage-donts-dos-and.html) 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 bits, setting the number of hash iterations to 20,000, setting the random number generator to SHA1PRNG, and setting the salt length to 64 bits. These settings' ramifications are likely not understood by most developers, who simply copy and paste the code into their applications. (And this assumes that developers find a valid tutorial in the first place. Several tutorials recommend that users merely encode passwords using "Base64 encryption," an encoding scheme that provides no security whatsoever; see https://3v4l.org/BZ7rC.) Although adding this functionality to standard storage classes isn't without risk, we argue that these risks are outweighed by those posed by forcing developers to deal with such issues themselves.

> **Developers could benefit from support in many areas, including safer programming languages, more usable security libraries, and better security testing tools.**

Design APIs that satisfy security and nonsecurity needs. Security API designers must remember that most developers using their API ultimately have a nonsecurity goal but must still consider security requirements. The API should be designed such that it's powerful enough to satisfy all the requirements. The previous URL example is an instance of a good API that didn't provide sufficient capabilities. Although the integration of the https logic into the URL class was exemplary, the API wasn't powerful enough to fulfill all the developers' requirements—particularly the nonsecurity requirements. In conducting a large-scale analysis of https code in Android, we found widespread misuse of the https functionality that endangered millions of users.[8] In follow-up work, we interviewed developers and discovered that the insecure code's root cause was that the API wasn't powerful enough to fulfill the developers' requirements.[14] This forced the developers to improvise their own code, which introduced vulnerabilities. Interviewing just a handful of developers provided a list of four simple requirements that, if met, would have prevented all the vulnerabilities found. We highly recommend this sort of interview-based gathering of requirements during API development. Ensuring that the API is powerful enough out of the box to fulfill developers' requirements is the best way to prevent developers from tinkering and breaking something.

APIs should be easy to learn. It's unrealistic to expect nonexperts to understand the cryptographic background of the APIs they're using. Consequently, APIs should be designed so that developers don't need to understand the background to correctly use them.

In our user study, we interacted with a developer who'd made a coding error concerning SSL certificate verification.[14] The error led to the code accepting any valid certificate, making it vulnerable to active man-in-the-middle attacks, in which an attacker replaces the legitimate recipient's public key with the attacker's public key. The data is still encrypted, but now under a key the attacker knows. When we informed the developer of the vulnerability, he didn't accept that there was a problem because he'd inspected his app's traffic using Wireshark, which showed the traffic as encrypted. The developer didn't have the necessary expertise to understand the nature of the attack.

The password example is another instance of most developers lacking the knowledge needed to make the right choices. Few outside the cryptographic community will understand the differences between MD5, SHA1, SHA512, or PBKDF2WithHmacSHA1, which are some of the options Java developers can choose from. Developers shouldn't need a background in the cryptographic protocols they're using—this should be hidden by the API.

Don't break the developer's paradigm. Developers' understanding of cryptography is often 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, developers are expected to understand that they must obtain the user's public key to encrypt to and then employ an algorithm to transform messages into ciphertext. However, recent attempts such as the NaCl library—primarily aimed at providing a simple and misuse-resistant API to software developers—take a different approach that requires both a user secret key and a receiver public key. Although the resulting function is secure enough as written in NaCl, it has led to the development of entirely new NaCl adaptations that provide the required public-key only functionality. For example, a derivative library called libsodium adds a "sealing" API that uses the underlying NaCl functions to implement this function. It's unclear whether such additional implementations harm the community, but the confusion resulting from such alternatives might lead to insecure results.

```
if (some_verify_function())

/* signature successful */
```

Figure 1. How developers might commonly check for OpenSSL errors.

```
if (1 != some_verify_function())

/* signature successful */
```

Figure 2. How, because of the idiosyncratic nature of OpenSSL's error-handling code, developers should actually check for errors.

APIs should be easy to use. Developers, like end users, don't like to read manuals before getting started. If the API isn't self-explanatory—or worse, gives the false impression that it is—developers will make dangerous mistakes. A good example appears in 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. 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's easy to see that without reading the documentation, developers will likely handle errors incorrectly. Figure 1 shows a very common way to handle errors. However, because of the idiosyncratic nature of OpenSSL's error-handling code, the correct way to check for errors is shown in Figure 2. This ambiguity tripped up even the OpenSSL developers, and led to the CVE-2008-5077 vulnerability.

Following on from the aforementioned Java password example, the algorithm is chosen by passing a string value to a "factory" method. This offers great flexibility, but it also forces the developer to consult the documentation to find out which values are valid. The API documentation for the SecretKeyFactory class doesn't even list the options but rather refers developers to the "Java Cryptography Architecture Standard Algorithm Name Documentation" (http://docs.oracle.com/javase/7/docs/technotes/guides/security/StandardNames.html)—which offers an exhaustive list of all cryptographic algorithm names but gives little guidance on when and where to use them. Therefore, it's unsurprising that so many password database compromises still turn up plaintext or poorly hashed passwords.

APIs should be hard to misuse. One of the most important differences between security and nonsecurity APIs is that errors made when using a nonsecurity API are 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, the design of security APIs should pay special attention to preventing incorrect use and to making errors visible.

Our running https example illustrates this problem well. Many developers who ended up deploying insecure apps did so because they encountered an SSL verification exception that prevented their app from opening a URL connection. The exception occurred because the endpoint they were connecting to didn't have a valid certificate under Android's roots-of-trust. The developers dealt with this by using various TrustManager implementations—all of which turned validation off entirely. Some developers did so not realizing that although this fix made the exception go away, it also seriously weakened their app's security. Others did so knowingly, usually because they wanted to use self-signed certificates during development and simply forgot to remove the testing code before deploying their apps in production. In both cases, the code compiles without any errors and the apps run without any problems. Incorrect use of the TrustManager only becomes apparent during adversarial testing.

API designers should make it as hard as possible for security APIs to be misused. This can be done in several ways. First, if our recommendations are followed, the correct use should intuitively present itself to the developer, who then won't perform the customizations that often lead to errors. Second, API developers should create "failsafes" that implement checks wherever possible to catch unsafe API use and make errors visible.

As a related requirement, security-related errors should be hard to circumvent. Our https example again serves to illustrate this. Developers could easily 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, that is, opening the Internet connection. Although this option might be desirable, it should require a reasonable amount of effort and the security implications should be made exceedingly clear to the developer. It should be less desirable to circumvent the error than to

fix the cause, with one notable exception: security APIs should offer a testing or development mode in which errors can be circumvented easily.

Make defaults safe and unambiguous. A fundamental principle of usable security is to avoid providing users with default behaviors that are either ambiguous or unsafe. Unfortunately, this lesson hasn't been universally adopted in cryptographic APIs. A simple example is the Java API for performing symmetric encryption. When instantiating the javax.crypto.Cipher class in the Java security API, users are invited to specify a detailed description of the cipher name and mode of operation and padding scheme. Alternatively, users might simply specify the cipher name (for example, "AES") and accept the defaults chosen by the underlying implementation. This second usage is quite common, as indicated by searches of open source code repositories. This API's challenge 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 Electronic Codebook Mode, an operation mode known for leaking information about the structure of repetitive or low-entropy plaintexts. Although the need for safe and unambiguous defaults seems obvious, experience shows that it can't be emphasized enough.

> **Implementing our proposals across large-scale cryptographic APIs could produce dramatically improved security with minimal effort.**

Include a testing mode. Security mechanisms often come with a complexity or performance penalty. While this is a necessary tradeoff 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.[14] 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 encryption APIs deal with initialization vectors (IVs). Many cryptographic algorithms require that an IV with a secure random number be used. Because it's 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. It's more difficult for a developer to check the output of a given method 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. To avoid making the testing mode an easy way to circumvent errors in a production environment, it's advisable to tie the testing mode to specific machine IDs; if code accidentally gets deployed in testing mode, it will lead to a visible error on deployment.

Ensure APIs are easy to read and update. Bloch recommends that API designers ensure that it's easy to maintain the code that uses the API. In the security context, this is particularly important because it's essential that security code be kept up to date. While programs that utilize outdated nonsecurity APIs age more or less gracefully, programs that use outdated security APIs become a liability. Take the password example again. The tutorial referenced was published in 2012. Not one of the security settings in that tutorial would be considered best practice today—only three years later. It's worthy of particular note that when left to developers, security parameters such as the number of iterations of the hashing algorithm will usually be hard coded. Because algorithms and parameters change frequently due to vulnerabilities being discovered and the need to counter attackers' increased capabilities, this solution is suboptimal, forcing all developers to update their code regularly. We argue that API design should plan ahead for easily foreseeable updates and make it as easy as possible for developers to update their software. One simple solution is to use configuration files instead of hard coding security parameters; however, it might be worth considering whether security APIs need to be more proactive and offer security updates similar to OSs.

APIs should interact with end users. Security APIs commonly provide functionality that might terminate with an error for security reasons, such as the presence of an attack, or more likely, a misconfiguration. In these cases, it's also common for this error to be passed up

through the software layers to the end user, who must decide how to proceed. Both of these occurrences are the nature of security code. However, it's also common for APIs to leave the entire burden of developing reaction strategies to the developer using the API, who must, for example, develop case statements and—more critically—the error messages and options for interacting with end users. This is a bad state of affairs for several reasons.

First, most developers using a security API don't have a firm grasp on the cryptographic or security background and thus would be hard pressed to explain to end users what went wrong. Second, designing good warning messages is an art unto itself. For instance, browser vendors have entire teams working to tweak and improve their SSL warning messages. Usable security researchers have also worked extensively on this problem.[15,16] However, SSL APIs offer no assistance. All app or software developers must invest the effort to design their own warning messages. This fact was bemoaned by several of the Android and iOS developers we interviewed. Although APIs' general purpose nature makes designing the actual UI infeasible 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.

In this work, we examined the paradigm of developer-friendly security, specifically considering its implications for cryptographic library software. We proposed a series of heuristics to help library developers produce software that minimizes the possibility of developer misuse. Implementing these proposals across large-scale cryptographic APIs could produce dramatically improved security with minimal effort. ■

References

1. *Net Losses: Estimating the Global Cost of Cybercrime*, report, McAfee, Intel, Center for Strategic and Int'l Studies, June 2014; www.mcafee.com/us/resources/reports/rp-economic-impact-cybercrime2.pdf.
2. M.E. Zurko and R.T. Simon, "User-Centered Security," *Proc. Workshop New Security Paradigms* (NSPW 96), 1996, pp. 27–33.
3. A. Adams and M.A. Sasse, "Users Are Not the Enemy," *Comm. ACM*, vol. 42, no. 12, 1999, pp. 40–46.
4. A. Whitten and J.D. Tygar, "Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0," *Proc 8th USENIX Security Symp.* (SSYM 99), 1999, p. 14.
5. I. Flechais, M.A. Sasse, and S.M. Hailes, "Bringing Security Home: A Process for Developing Secure and Usable Systems," *Proc. Workshop New Security Paradigms* (NSPW 03), 2003, pp. 49–57.
6. S. Garfinkel and H.R. Lipford, *Usable Security: History, Themes, and Challenges*, Morgan & Claypool, 2014.
7. M. Georgiev et al., "The Most Dangerous Code in the World: Validating SSL Certificates in Non-browser Software," *Proc. ACM Conf. Computer and Communications Security* (CCS 12), 2012, pp. 38–49.
8. S. Fahl et al., "Why Eve and Mallory Love Android: An Analysis of Android SSL (In)security," *Proc. ACM Conf. Computer and Communications Security* (CCS 12), 2012, pp. 50–61.
9. P. Gutmann, "Lessons Learned in Implementing and Deploying Crypto Software," *Proc. 11th USENIX Security Symp.* (USENIX 02), 2002, pp. 315–325.
10. J. Tulach, *Practical API Design: Confessions of a Java Framework Architect*, Apress, 2012.
11. K.L. Hunter, *Irresistible APIs: Designing Web APIs that Developers Will Love*, Manning, 2016.
12. M. Reddy, *API Design for C++*, Academic Press, 2011.
13. J. Bloch, "How to Design a Good API and Why It Matters," *Companion to the 21st ACM SIGPLAN Symp. Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA 05), 2005, pp. 505–507.
14. S. Fahl et al., "Rethinking SSL Development in an Appified World," *Proc. ACM SIGSAC Conf. Computer & Communications Security* (CCS 13), 2013, pp. 49–60.
15. J. Sunshine et al., "Crying Wolf: An Empirical Study of SSL Warning Effectiveness," *Proc. 18th Conf. USENIX Security Symp.* (SSYM 09), 2009, pp. 399–416.
16. A. Sotirakopoulos, K. Hawkey, and K. Beznosov, "On the Challenges in Usable Security Lab Studies: Lessons Learned from Replicating a Study on SSL Warnings," *Proc. 7th Symp. Usable Privacy and Security* (SOUPS 11), 2011, article 3.

Matthew Green is an assistant professor of computer science at the Johns Hopkins University Information Security Institute. His research interests include applied cryptography, privacy-enhanced storage systems, and anonymous cryptocurrencies. Green led the team that developed the first anonymous cryptocurrencies, Zerocoin and Zerocash. He received a PhD in computer science from Johns Hopkins University. Contact him at mgreen@cs.jhu.edu.

Matthew Smith is a professor of usable security and privacy at the University of Bonn and the Fraunhofer FKIE. His research focuses on human factors of security and privacy mechanisms, including SSL and network security, authentication, mobile and app security, and usable security for developers and administrators. Smith received a Diplom with distinction in computer science and electrical engineering from the University of Siegen, and a PhD with distinction in computer science from Philipps Universität Marburg. Contact him at smith@cs.uni-bonn.de.