Developer-Resistant Cryptography

Kelsey Cairns and Graham Steel

Abstract

"Properly implemented strong crypto systems are one of the few things that you can rely on" - *Edward Snowden*. So why is mass surveillance so successful? One (big) problem is endpoint security. Another is that strong crypto systems are sufficiently difficult to implement that often either mistakes are made resulting in catastrophic loss of security, or cryptography is not used at all. What can we do to make cryptography easier to use and more resistant to developer errors?

1 Introduction

Our world is one of constant connectivity. Smart phones, laptops and tablets allow us to access and share information at any time from almost any location. The data we send – often personal and private information – is handled by numerous applications, each of which has been shaped by numerous developers. Alarmingly, the level of expertise of each of these developers is unknown to us. A single mistake from one of them could expose our private data to unfriendly observers.

Unfortunately, studies of application code suggest that developers make more errors than we like to think. Multiple studies targeting Android apps reveal alarming numbers of apps with incorrectly deployed cryptography. Attacks and intrusions are actively made possible through developer error. Both professional and amateur apps contain vulnerabilities exposing personal information. These findings indicate a general gap between the knowledge level of many software developers and the understanding necessary to correctly implement cryptography using available APIs. With the Internet full of untrustworthy characters, we cannot afford to be giving information away so easily. This paper approaches this problem from the perspective that developers are not perfect and will make mistakes, but by studying common problems, APIs can be designed to increase the chances that developers will make the correct choices.

2 Difficulties With APIs

The field of cryptography is inherently difficult. Cryptographic API development involves narrowing a large, complex field into a small set of usable functions. Unfortunately, these APIs are often far from simple. For example, the

SSL/TLS API contains hundreds of functions for a variety of uses: key generation and management, signature algorithms, ciphers, cipher modes, padding schemes, etc [4]. Simply knowing which ciphers are considered secure is not sufficient. Something as simple as the wrong padding scheme could cause vulnerabilities. All too often it is unclear which combinations are secure and which are not. To complicate matters, default algorithms are not always the most secure, requiring developers to recognize when and where algorithms need to be specified.

Problems with obscure cryptographic APIs are compounded by a lack of education for developers. Security best practices are not necessarily part of every developer's education. In addition, employers may be hesitant to invest in training, preferring to take risks in order to ship products more quickly. In other cases, developers may have the knowledge to implement correct security but simply forget after working on more exciting tasks first.

Large numbers of incorrectly developed applications have been exposed by multiple studies on Android apps. In 2012, Fahl et al. tested more than 13,000 apps for incorrect use of SSL/TLS [2]. Of the apps using SSL/TLS, 17% included a potential vulnerability. The most common error was failure to check certificates and simply reporting them valid. Other errors included accepting incorrect certificates or certificates signed by untrusted certificate authorities. Due to the popularity of some of these apps, the number of potentially compromised devices was estimated between 39 and 185 million.

In 2013, study by Egele et al. revealed even more startling figures [1]. In this study, six rules were defined which, if broken, indicated the use of insecure protocols. More than 88% of the 11,000 apps analyzed broke at least one rule. Of the rule-breaking apps, most would break not just one, but multiple rules. Some of these errors were attributed to negligence, for example test code included in release versions. However, in most cases it appears developers unknowingly created insecure apps.

Android apps have provided a new wealth of applications to study, but the problem of misused cryptography is much older. In 2002, Peter Gutmann wrote about several types of insecurities commonly found in professionally developed software [3]. It seems safe to conclude that incorrect cryptography is an ongoing problem, with no single API specifically at fault. This problem seems likely to continue without the design of better securities APIs.

3 Looking Forward

To combat developer induced security flaws, we consider the gap between apparent developer knowledge and the knowledge needed to create secure applications. We can tackle this problem from both sides: the human side and technological side. This section presents some ideas towards minimizing the knowledge gap, bearing in mind the reality that even knowledgeable developers are capable of occasional errors.

The human aspect can be improved through better education for developers.

Sadly, this approach is unlikely to be a complete solution. It is unreasonable to expect a developer to be a security expert when most of their time is spent on other aspects of software design. One helpful step would be to ensure up-to-date tutorials and thorough documentation so that developers who handled security only occasionally would always have an accurate reference.

Rather than relying on developer improvement for the complete solution, tackling the problem from the technological side could go a long way towards reducing developer error. Taking this approach, we attempt to design APIs that developers can intuitively use correctly. Each common vulnerability provides valuable insight, pinpointing where APIs improvements would be most valuable.

One common source of confusion is the availability of insecure functions. Such functions or combinations might be required for backwards compatibility but should require extra effort to deploy. In addition, thought should be given to designs which would allow defaults and easily accessible functions to be updated between API versions. As cryptography advances, APIs should be updated accordingly, without inducing new weaknesses in applications developed with respect to earlier versions.

Another common error is the use of hard coded values as keys, salt values, or seeds for random number generators. This error seems likely to have two causes: either debugging code accidentally going into production or complete developer misunderstanding. One avenue to explore would be API design where any algorithm requiring a fresh random value would generate it on its own rather than trusting the developer to supply one. The newly generated value could be returned from the algorithm along with any other computed values. A debugging flag might be used to prevent randomness for testing purposes, allowing developers to test applications while making it easier to transition from development to production stages. Relieving developers of some of the responsibilities involved in key generation prevents attacks that take advantage of hard-coded values.

The lack of SSL/TLS certificate checking found by Fahl et al. was a less prolific error, but a very grave one. For handling certificates, APIs should be very carefully crafted, minimizing the amount of code a developer must write to verify a certificate. At the same time, the whole verification step must be very hard to overlook. Additionally, generalized error handling could minimize the temptation to ignore unauthorized certificates. Even if the intent is only to ignore the error temporarily, it may be forgotten. With PKI used so prevalently in today's networks, secure verification and avoidance of man-in-the-middle attacks is extremely important.

These ideas are only examples of how common errors could inspire the design of future APIs. Another example is W3C's WebCrypto API which will include a smaller subset API specifically designed to be used easily by web developers [5]. The "High-Level API" aims to eliminate the expertise necessary to work with the full API, subsequently eliminating mistakes that inexperienced developers might make. Projects like these are steps towards more usable APIs and potentially much more secure application development in general.

4 Conclusion

Application security is caught in a strange conflict: cryptography is increasingly necessary to protect personal data, yet quite difficult to implement correctly. The problem is compounded by complex APIs and the lack of expertise among software developers in general. To decrease the number of security holes caused by developers, we should not only do everything possible to educate developers, but also make APIs easier to use. By examining errors commonly found in today's applications we can find the weaknesses of today's APIs. Using what we learn, we can design APIs that are more intuitive and easier to use correctly, decreasing the overall chance of human error.

References

[1] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 73–84. ACM, 2013.

[2] Sascha Fahl, 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 *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 50–61. ACM, 2012.

[3] Peter Gutmann. Lessons learned in implementing and deploying crypto software. In *Usenix Security Symposium*, pages 315–325, 2002.

[4] OpenSSL. Openssl: Document, ssl(3), 2006. `http://www.openssl.org/docs/ssl/ssl.html`

[5] Ryan Sleevi, David Dahl. Web Cryptography API. W3C Working Draft (Work in progress.) `http://www.w3.org/TR/WebCryptoAPI/`