Toward Semantic Cryptography APIs

Arvind Kumar Pandey, Assistant Professor

Department of Computer Science, Arka Jain University, Jamshedpur, Jharkhand, India

Email Id- arvind.p@arkajainuniversity.ac.in

ABSTRACT: While there are many established cryptographic frameworks that have been used to create sophisticated systems, developers often use them improperly, resulting in security problems. This is due to a number of issues, including the expectation that framework users understand security attacks and defense, as well as the subtle impact of various low-level parameters.) The need to take into account information from outside the system to ensure security (e.g., TLS certificate revocations). The need to disable security checks frequently during development and testing, as some security checks are disabled by default. We offer design principles for cryptography APIs that are semantically relevant to developers and can be reliably implemented on top of current frameworks. We also suggest the Regulator design pattern for integrating security-critical external information, as well as management hooks for isolating security workarounds required during development and testing. Our API is a starting step toward finding the appropriate balance between limiting developers' security choices while still allowing them the freedom required for sophisticated cryptographic applications.

KEYWORDS: API, Cryptography, Encryption, Framework, Security.

1. INTRODUCTION

Oracle JSSE, IBM JSSE, BouncyCastle, and OpenSSL are among the advanced cryptographic frameworks presently available for creating secure client-server communications, storing data securely, and processing payments. These frameworks are built by cryptography specialists and contain state-of-the-art algorithms. Their code has been audited and evaluated using static analysis tools, and they are used to create real-world applications with high security. Unfortunately, when using these frameworks, software developers frequently make critical errors such as exchanging keys without authenticating the endpoint, storing sensitive information in cleartext or with weak protection, using parameters for block ciphers that are known to be insecure and using encryption keys that are known to be insecure.

Simplified cryptographic APIs, secure default settings for cryptographic algorithm parameters, and utilizing static analysis tools to identify problems related to cryptography abuse are some of the solutions suggested for this problem. These solutions ignore a more basic issue: existing cryptography frameworks erode abstraction barriers by failing to encapsulate all framework-specific information and expecting developers to be aware of security threats and responses. Different cryptographic frameworks, for example, can behave differently when providing the same functionality: a TLS handshake will fail in an IBM JSSE implementation if a trusted, but expired, certificate is provided, whereas the handshake will succeed in an Oracle JSSE implementation because the developer is expected to check for expired certificates[1].

Empirical findings of cryptography abuse are included in the linked work, as are efforts to avoid such misuse by simplifying the use of cryptographic APIs. In an empirical investigation of cryptographic abuse in Android apps, Egele discovered that 88 percent of Android applications that utilize cryptographic APIs make at least one error. According to Reaves improper certificate validation, storing login credentials in plain text, and employing inadequate authentication methods such as do-it-yourself encryption were all found in 46 Android apps that conduct financial transactions[2]–[4].

Many commonly used apps, including Amazon's EC2 Java library, PayPal's merchant SDK, shopping carts like osCOmmerce, Ubercart, and Chase mobile banking, all conduct faulty certificate validation, according to Georgiev. The researchers discovered that the underlying causes of these flaws are poorly designed SSL implementation APIs (such as JSSE, OpenSSL, and GnuTLS) and data-transport libraries (such as cURL), which provide developers with a bewildering variety of settings and choices[5]–[7].

Root causes of SSL development, according to Fahl, are not only irresponsible developers, but also limits and problems with the existing SSL development paradigm. Participants who used Stack Overflow generated substantially less secure code than those who used official instructions or published books, according to Acar. We contribute to this body of work by finding systematic variations in behavior across major cryptography frameworks while implementing the same task. Our primary emphasis, however, is on providing a solution to these issues.

b) Simpler cryptographic API use. The NaCl cryptography library provided simplified APIs with the goal of avoiding some of the common misuse patterns seen in other cryptographic libraries. For authenticated encryption and digital signatures, NaCl offers all-in-one crypto box and crypto sign APIs, which replace OpenSSL's sequences of library calls. To avoid developers developing their own SSL code, Fahl suggest changing the Android OS to offer the major SSL use patterns as a service that can be added to applications through settings. Instead of developing a new library or service, we overlay a semantic layer on top of existing libraries, allowing us to offer consistent APIs across libraries, programming languages, and platforms.

OpenCCE, a tool for managing software product lines, is the most closely similar to ours. It is because many cryptographic solutions are made up of combinations of common cryptographic algorithms that are parameterized at compile time. OpenCCE assists developers in selecting suitable algorithms and generates Java code as well as a use protocol. This method necessitates the use of static analysis to track code changes over time in order to verify that the use protocol is not broken. Furthermore, the code generated is dependent on the library used (for example, minor variations in how SSL hostname verification is handled in various JSSE libraries may lead to vulnerabilities when switching libraries) and does not account for the requirement to integrate external data at runtime. We propose portable semantic APIs instead of static analysis tools.

Full disk encryption is a kind of storage hardware encryption that is helpful for devices that may be physically lost or stolen and needs the encryption key to be stored. It can only be used for data storage, not transmission. Instead of separate storage and communication functions, a single function for both interfaces would be ideal.

Part of the issue might be solved by adhering to API design best practices, but other difficulties are unique to the cryptography domain. The security of cryptographic applications, in particular, is often reliant on information from outside the system. Because of the performance of current technology, the SHA-1 cryptographic hash algorithm is no longer deemed safe and compromised TLS certificates are revoked and reissued to avoid man-in-the-middle attacks [8][9].

In order to utilize cryptography safely, applications must perform runtime checks that consider such external information, but they must also have the freedom to choose the best method for integrating this information. An application may, for example, download certificate revocation lists (CRLs), use the Online Certificate Status Protocol (OCSP), or implement OCSP stapling to verify the revocation status of TLS certificates. There is presently no consensus on the optimal approach, and the decision will most likely be platform dependant. Another domain-specific issue is that, since security means that some activities are forbidden, developers often need a method to circumvent security checks in the development environment in order to test all code paths. When a developer begins developing an SSL application, for example, the code raises errors owing to the lack of a certificate or the usage of a self-signed certificate. Many of these developers will therefore skip SSL certificate validation in order to keep developing and testing their code. While these workarounds serve a valid function in the development environment, they are sometimes implemented in production because developers are unaware of the security risks associated with them [10][11].

➢ *We offer three contributions to solve these issues:*

- We offer design principles for cryptographic libraries' semantic APIs, which reveal security choices without needing in-depth understanding of attacks and countermeasures. We provide a proof-of-concept implementation of secure communication and storage APIs to show that these APIs may be implemented consistently on top of current frameworks without the need for additional cryptographic protocols or algorithms.
- We present Regulator, a generic design pattern for transparently integrating data from external trustworthy sources. We also suggest three different ways to implement this design.
- To distinguish the development and production environments, we offer compile-time checks. This enables a clear distinction between security workarounds that should not be used in production and those that should.

1.1 Semantic APIS:

Authentication and authorization services, ecommerce SDKs and integrated shopping carts, and mobile payment systems all use cryptographic APIs, and developers need the freedom to implement the unique protocols needed in these applications.

However, we think that just a few people on the team should make security choices. These programmers are known as security engineers. The remainder of the developers, referred to as functionality engineers, should be allowed to concentrate on the application logic, and their design decisions should have no effect on security.

This may be accomplished by offering APIs for functionality engineers that are semantically relevant, such as secureConnect. secureSend and secureReceive are used to establish safe communication channels, whereas secureWrite and secureRead are used to securely store data. Associated functionalities for transmitting and storing data without authentication or encryption are also included in the APIs. The secure* functions should always handle sensitive data, according to the API implementation. This may be accomplished by asking security engineers to provide an isConfidential function that takes a data object as an argument and returns a boolean result indicating if the data is secret. By default, the function returns true, explicitly whitelisting non-sensitive data.

Instead of transmitting sensitive data across an unsecured channel, functions that are not intended to offer security call isConfidential and return an error [12].

1.2 Engineers that specialize in functionality:

1.2.1 Communicate Interface:

The secure* functions would be utilized in the same way as their insecure equivalents from the standpoint of a functionality engineer.

secureConnect/secureSend are appropriate for transmitting sensitive information such as login credentials, Social Security numbers, credit card details, and so on. The feature engineer just has to provide the address addr and the data msg to send a message, and the API implementation should conceal the complexity of setting up a TLS connection (e.g. endpoint authentication, cipher suite negotiation, certificate validation). SecureSend should always be used to send confidential communications, according to the API implementation. When isConfidential(msg) returns true, the send implementation produces an error to prevent feature engineers from inadvertently transmitting sensitive data via an unsecured channel. The security engineers define msg as an instance of a particular superclass that categorizes all data as sensitive or non-sensitive [13].

This separates the job of defining whether data is confidential from the task of implementing network communications, ensuring that the network programmer does not jeopardize security by using send when secureSend should be used.

1.2.2 Storage Interface:

A write provides for unencrypted writing of a value (or a file) to the storage system, while secureWrite encrypts and hashes the file and authenticates the storage system before transferring data to it, ensuring integrity and secrecy. A functionality engineer simply has to provide the data to be saved and a filespec that defines the storage location, regardless of whether the data is sensitive or not.

The security engineers must define an isConfidential function, similar to the Communicate interface, that verifies whether the data is really non-sensitive, separating the data sensitivity checks from the application logic.

Data may be transmitted as a single packet per connection on both interfaces, or the connection can be kept open until all data packets have been sent, which is a performance-security tradeoff. If each data packet is labelled using the isConfidential function in the latter scenario, it will assist to enhance security.

Engineers in charge of security: Security engineers, unlike functionality engineers, design more sophisticated interaction protocols (for example, authentication and authorisation, online purchasing, and payment processing). As a result, these engineers must have a deeper grasp of security concepts and are in charge of providing the functionality developers with understandable APIs.

1.2.3 connect/send:

The connect method establishes a communication socket, while the send function sends the message to the specified address. The data may be transmitted in plaintext in this case. This function, as previously stated, calls the isConfidential function to verify that data transmitted over the network is not sensitive.

1.2.4 secureConnect/secureSend:

The secureConnect method establishes a secure communication connection, for example, by using Java's HttpsUrlConnection for HTTPS. All SSL checks that are not handled by the underlying libraries are handled by this function. If the library developer uses an Oracle JSSE library, for example, this method will check for certificate expiration. Before sending the message to the destination, the function conducts the required checks for certificate revocation.

1.3 Managing Security Checks During Testing and Development:

Developers require a mechanism to indicate that these workarounds should not be included in production versions when they deactivate security checks in the development environment, for example, by using self-signed certificates to circumvent certificate validation checks. This separation should be enforced at build time, so that production releases do not have to deal with the cost of workaround checks. When developing with self-signed certificates, for example, developers may have distinct validation criteria for certificates. TestKeystore, which may include self-signed certificates, would be checked by the validation criteria for the test environment. The production environment's validation criteria would check ProdKeyStore, which only accepts genuine certificates.

Because software projects usually have distinct build configurations for development and production, limiting security workarounds to the development build environment is a logical approach to protect yourself. A Maven build profile, for example, may define the characteristics of the two build environments, including environment variables that determine which keystore to utilize. The keystore is selected conditionally in the source code depending on the environment variable. This creates a clear boundary between environments, lowering the chance of security flaws making their way into production versions.

2. DISCUSSION

The majority of reported cryptographic API abuses may be attributed to the insufficient abstractions given to developers without a background in security or cryptography. However, knowing the security choices that developers must make is required to achieve a successful separation of concerns. For example, we should not expect developers to know the conditions under which the SHA-1 hash function can be used securely (it is currently not recommended for digital signature generation, but it is allowed for all other applications), but we should expect them to determine the sensitivity level of the data handled by their code. Other security choices may be more difficult to see. Developers, in particular, have a legitimate need to disable security checks during development and testing, and they must be able to choose the mechanism for retrieving information about revoked TLS certificates, as there is currently no consensus on the best method, and the choice is likely to be platform dependent.

Our effort is a first step toward assisting developers in making fewer cryptographic errors, since other applications may need more protection. However, we highlight two domain-specific challenges: how to include external information at run time and how to provide compile-time checks to eliminate the necessity for security workarounds during development. This emphasizes the need of going beyond excellent API design to solve the issue of cryptographic errors. Providing excellent documentation on the cryptographic system to developers is also essential, although it is likely insufficient.

A controlled experiment with two teams of programmers, comparable to Yskout is one possible assessment method. The main issue is determining the effect of our solutions on the security of the resultant code (rather than the productivity of the programmers), since establishing meaningful security measurements is challenging. Such an experiment would very certainly require a systematic method of identifying the cryptographic: flaws introduced by each coder, such as having a panel of experts examine each participant's code individually.

3. CONCLUSION

To build safe applications, modern cryptographic frameworks such as Oracle JSSE, IBM JSSE, BouncyCastle, and OpenSSL are frequently utilized. However, software developers without a background in security or cryptography often make errors while utilizing these frameworks, which typically result in serious security flaws. We recognize the necessity to integrate external data in order to enhance security. We propose semantic APIs based on this, which enable developers to make good security choices while hiding low-level implementation specifics. Using innovative and well-known design principles, these APIs can be implemented uniformly across platforms. We suggest a Regulator design in particular for integrating external information from trustworthy sources, such as TLS certificate revocation status or knowledge of unsafe cryptographic methods and key lengths. In order to address the genuine requirement to deactivate security checks during development and testing, we additionally propose compile-time checks to isolate some workarounds to the development build environment. Finally, we explore the research possibilities that these concepts offer up.

REFERENCES

[1] A. Czeskis, M. Dietz, and D. Wallach, "Strengthening User Authentication through Opportunistic Cryptographic Identity Assertions Categories and Subject Descriptors," *Proc. 19th ACM Conf. Comput. Commun. Secur.*, 2012.

[2] H. Halpin, "The W3C web cryptography API: Motivation and overview," 2014. doi: 10.1145/2567948.2579224.

[3] M. A. Sasse and M. Smith, "The Security-Usability Tradeoff Myth [Guest editors' introduction]," *IEEE Secur. Priv.*, 2016, doi: 10.1109/MSP.2016.102.

[4] Z. Lina, "Design and Implementation of KSP on the Next Generation Cryptography API," *Phys. Procedia*, 2012, doi: 10.1016/j.phpro.2012.05.264.

[5] K. Cairns, H. Halpin, and G. Steel, "Security analysis of the W3C web cryptography API," 2016. doi: 10.1007/978-3-319-49100-4_5.

[6] J. Toldinas, R. Damasevicius, A. Venckauskas, T. Blazauskas, and J. Ceponis, "Energy consumption of cryptographic algorithms in mobile devices," *Elektron. ir Elektrotechnika*, 2014, doi: 10.5755/j01.eee.20.5.7118.

[7] J. Knudsen, "Java Cryptography," *EDPACS*, 1999, doi: 10.1201/1079/43250.27.4.19991001/30275.5.

[8] R. Tous, M. Guerrero, and J. Delgado, "Semantic web for reliable citation analysis in scholarly publishing," *Inf. Technol. Libr.*, 2011, doi: 10.6017/ital.v30i1.3042.

[9] Y. Yang, S. Zhang, J. Yang, J. Li, and Z. Li, "Targeted fully homomorphic encryption based on a Double Decryption Algorithm for polynomials," *Tsinghua Sci. Technol.*, 2014, doi: 10.1109/TST.2014.6919824.

[10] H. A. Yajam, J. Mohajeri, and M. Salmasizadeh, "Identity-based universal re-encryption for mixnets," *Secur. Commun. Networks*, 2015, doi: 10.1002/sec.1226.

[11] C. Choi, J. Choi, and P. Kim, "Ontology-based access control model for security policy reasoning in cloud computing," *J. Supercomput.*, 2014, doi: 10.1007/s11227-013-0980-1.

[12] Y. Lindell, "How To Simulate It – A Tutorial on the Simula," *Fc*, 2016.

[13] V. A. Roman'Kov, "New probabilistic public-key encryption based on the RSA cryptosystem," *Groups, Complexity, Cryptol.*, 2015, doi: 10.1515/gcc-2015-0016.