IV = 0 Security
Cryptographic Misuse of Libraries

Somak Das
Vineet Gopal
Kevin King
Amruth Venkatraman

6.857 Final Project
May 14, 2014

Abstract

While many essential cryptographic primitives have sound theoretical constructions and associated implementations, their security guarantees do not always hold in practice. We find that these security flaws do not usually exist in the libraries themselves, but instead in how developers use these libraries. We select the most popular cryptographic libraries from five programming languages (C, C++, Java, Python, and Go) and examine the properties that encourage or discourage cryptographic misuse. We then derive recommendations for library designers to follow to reduce this misuse. We find that the C++ NaCl cryptographic library is the safest. As an intermediate solution to more dangerous libraries, we present a proof-of-concept linter for PyCrypto which warns developers of potential cryptographic misuse in their programs.

1 Introduction

The average developer is not a cryptographic expert. Thus, when the time comes to implement secure protocols in an application (e.g., to protect user data), developers depend on publicly available cryptographic libraries to provide the building blocks (cryptographic primitives) they need. As illustrated in Figure 1, libraries expose these primitives through an application programming interface (API) that specifies the methods and parameters. Unfortunately, a badly designed API means that the developer's program will be insecure, breaking security attributes like confidentiality and integrity.

Consider an optional parameter "`iv`" for the PyCrypto library's symmetric-key encryption method, which just so happens to be the initialization vector for AES in CBC mode. The developer leaves it unspecified, and PyCrypto makes it zero by default. But because IV = 0,

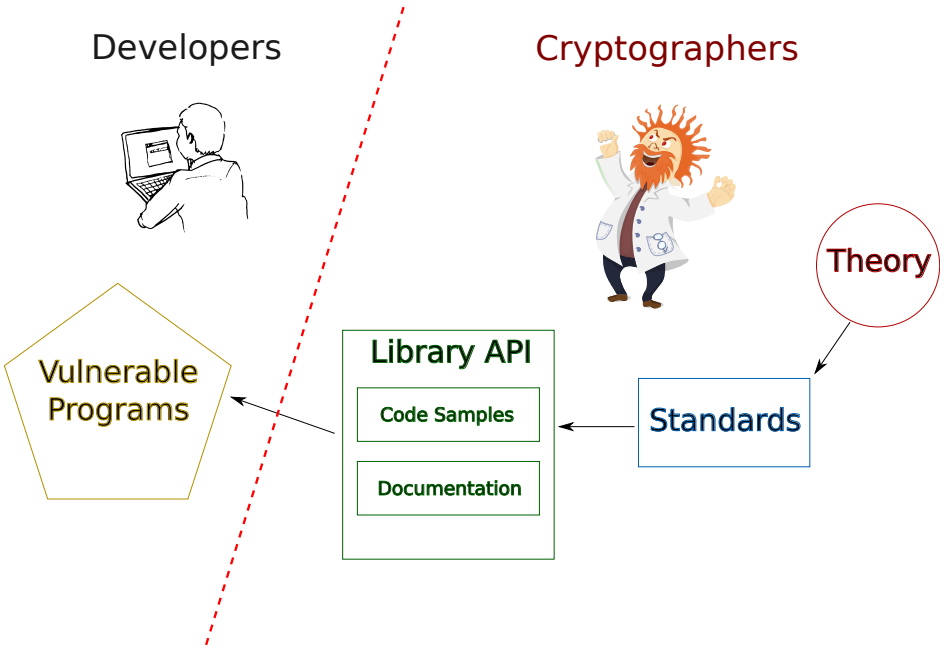

Figure 1: Cryptographic library implementations are derived from cryptographic standards, which in turn are derived from theoretical cryptography. All three of these stages are spearheaded by cryptographic experts, while the developers using these libraries are not. The highly differing backgrounds of developers and cryptographers must be bridged by cryptographic libraries that discourage cryptographic misuse. While developing programs securing user data, these developers examine (in increasing priority) the documentation, code samples, and API.

the resulting encryption is IND-CPA insecure (thus explaining our title, *zero security*). There is no warning for the developer, until an adversary compromises his whole program.

This study answers the question: *do libraries allow, or even encourage, developers to misuse them and write insecure programs?* We survey 6 libraries implementing cryptographic primitives for 5 programming languages (since different languages have different code styles and standards, which influence API design). We find 7 common issues that are handled in surprising ways by these libraries and write `pycrypto_lint` to address them. These issues are shown in Table 1.

As shown in Section 2, substantial research already exists that points out cryptographic misuse in the wild. In Section 3, we provide a brief primer on the cryptographic primitives discussed in the rest of the report. Section 4 describes our approach in selecting and evaluating each cryptographic library. Section 5 distills our analysis into seven main aspects and presents a detailed evaluation of each. In Section 6, we summarize our recommendations to cryptographic library designers based on that analysis. Section 7 presents our proof-of-concept linter for the PyCrypto library, which is intended to be an intermediate solution detecting cryptographic misuse.

Point of interest	Dangerous library ex.	Safe library ex.
IV reuse	defaults to constant value	defaults to secure value
Library defaults	insecure defaults	secure defaults
Feature bloat	unclear which methods are insecure and outdated	clear; or insecure methods are removed from API
Feature incomplete	exclude the latest methods	include the latest methods
Documentation	innavigable, excludes security discussion	clear, organized, includes security discussion
Code samples	fake, hard-coded parameters	actual, random parameters
Programming languages	return codes, no byte type	exceptions, strict typing

Table 1: 7 common issues handled in safe vs. dangerous ways by the 6 cryptographic libraries.

2 Related Work

To the best of our knowledge, we present the first study to systematically compare the APIs of cryptographic libraries across programming languages and evaluate their potential for misuse. Recently, Cairns and Steel [1] outlined their vision for developer-resistant cryptography. While they come to conclusions similar to ours, their work provides a high-level overview and does not consider specific examples of dangerous libraries.

Proposed replacements to cryptographic libraries also identify issues with older libraries. Python's cryptography is one example [2]. Another is NaCl, which compares itself to cryptlib for API simplicity [3]; we investigate this further by comparing them to 5 other libraries. NaCl also discusses security issues with previous implementations, such as timing attacks, and poor performance. We decide not to focus on security of the libraries themselves — because they are popular and have been tested in the wild (with well-publicized bugs and fixes, e.g., Heartbleed [4]), we assume that it's more common for usage of the library to be insecure than for the library to be insecure. Similarly, we believe the speed of cryptographic operations matters less if they're composed in an insecure way in the first place. This is why we did not focus on performance, even though comprehensive performance tests on standard benchmarks have been published [5, 6],

The closest study to our work is from Egele et al. [7], who checked Android applications from Google Play for cryptographic misuse and found that 88% were insecure. The Java-based Android platform uses the Java Cryptography Architecture, so we extend their study by comparing it to 5 other popular libraries and seeing if the same issues apply. There are a few differences between studying Android applications and non-Android programs. While the Android applications (as binaries) are publicly available on Google Play, sorted by usage, the non-Android programs are not. There is no central repository for these programs. Furthermore, they can be closed-source (while Android binaries can be decompiled). Therefore, we

evaluate the potential for misuse by using the libraries ourselves and searching open-source programs hosted on GitHub for examples.

Similarly, there is a distinction between CryptoLint for Android applications and the `pycrypto_lint` checker we developed for any program using PyCrypto. We focused on platforms that cannot provide decompiled bytecode. Our more general approach is to instrument the APIs and distribute a `pycrypto_lint` version of the PyCrypto library that can be used during program development. The added benefit of a runtime source checker (compared to CryptoLint's static bytecode checking) is that we can pinpoint errors with line numbers and stack traces. We are also able to compare across repeated calls to the API and flag more complex issues like repeated IVs and IVs generated without the random number generator.

3 Cryptography Primer

3.1 Symmetric Key Encryption

Block Ciphers

Block ciphers are often used in cryptography to accomplish symmetric encryption — that is, communicating data between two parties with a shared secret called a key. The block ciphers take as input a sequence of bits of a particular length (e.g., 128 bits) and return output of a set size. The key is the tool that converts the plaintext into the encrypted ciphertext. Keeping the key secret is essential. If the key ever becomes public, anyone can encrypt and decrypt messages.

Modes of Operation

In practice, most messages being encrypted are larger than the size of a block. To bridge the traditional block ciphers with variable length input, we use modes of operation. A mode of operation merely refers to the way in which input is padded and the blocks are encrypted (e.g., chained). Some of these methods are more secure than others.

Security Definitions

In practice, an encryption scheme should be Indistinguishable Under Chosen Plaintext Attack (IND-CPA) secure. This security metric is defined via an adversarial game. The adversary generates two plaintext messages, which are then encrypted using an unknown key. The adversary then tries to distinguish between the two encrypted ciphertexts. The adversary has access to an encryption oracle throughout the game, though he does not have the key itself. If the adversary can only distinguish between the two with negligible probability, the encryption scheme is said to be IND-CPA secure.

3.2 Asymmetric Key Encryption

In practice, it can be hard to establish a common key with another individual securely. This problem gets magnified when we realize that each person is likely communicating with many others and not just one. The solution can be found in asymmetric encryption. The idea is that each individual now has a secret and public key, of some reasonable length (e.g., 1024 bits). The secret key is generated at random by an involved process. The public key is derived from the secret key. Consider an individual B with public key PK_{B} and secret key SK_{B}. Individual A encrypts a message m to B using $c = E(m, PK_{\mathsf{B}})$. B can then decrypt the ciphertext to get back m by computing $m = D(c, SK_{\mathsf{B}})$.

3.3 Hash Functions

Hash functions map arbitrary long strings to a (seemingly) random fixed-length output. Hash functions are used primarily in authentication, providing a hard-to-invert primitive that is both easy to compute and public knowledge. Cryptographic hash functions are hash functions which either mathematically guarantee, or are believed to have, some security property. While hash functions backed by hard computational assumptions (such as the Discrete Logarithm Problem) exist, they are generally too inefficient for practice. Instead, heuristic bit-mixing hash functions such as SHA256 are used. NIST organizes competitions to invent new hash functions, the most recent being Keccak's SHA-3 [9]. Hash functions should be collision resistant, meaning finding two inputs which hash to the same value must be computationally hard. Some hash functions (e.g., MD5) have known collisions but are still frequently used, even in security-critical scenarios. Software engineers must stay up-to-date on the latest status of hash function security or risk severe attack vectors.

3.4 Digital Signatures

Signing a message is a way of convincing a recipient that the message has not been tampered with. Message signing usually takes as input a variable length bit sequence and outputs a relatively short sequence. The message signing protocol must ensure a number of properties:

1. No one should be able to forge another person's signature — only the owner of the PK/SK pair should be able to sign the message as himself. Reasonably, it must be computationally unfeasible to compute the secret key from the public key (as otherwise forging a signature would be easy).
2. Given the PK for the SK used to sign the message, verifying the message signature should be possible.
3. The message signing protocol should not leak any information about the private key.

4 Approach

4.1 Study of Cryptographic Libraries

In this paper, we survey six popular cryptographic libraries across five platforms (Table 2). We looked at OpenSSL in C, Crypto++, and NaCl in C++. In higher-level languages we looked at PyCrypto in Python, the Java Cryptography Architecture (with Bouncy Castle and Oracle Providers) in Java, and the Go Crypto package in Go. We chose libraries that offered lower level cryptographic primitives like RSA, block ciphers, and symmetric key encryption. We focus on libraries implementing cryptographic primitives because these libraries form the building blocks of other secure applications.

Library	Language	Year Created	Version	URL
OpenSSL	C	1998	1.0.1g	`openssl.org`
Crypto++	C++	1995	5.6.2	`cryptopp.com`
NaCl	C++	2008	2011.02.21	`nacl.cr.yp.to`
Java	Java	2007	7 SE	`docs.oracle.com/javase/7`
PyCrypto	Python	2002	2.6.0	`dlitz.net/software/pycrypto`
Go	Go	2011	1.2	`golang.org/pkg/crypto`

Table 2: 6 cryptographic libraries across 5 programming languages.

4.2 Usage

To understand the complexities of a cryptographic library, we implemented a series of basic actions using various cryptographic primitives. In each of the libraries and platforms, we implemented a public-key authenticated encryption system, using the following steps:

1. Parties A and B generate public/private key pairs.
2. A generates a secret symmetric key k.
3. A encrypts k with PK_{B}.
4. A sends B the encrypted k.
5. B decrypts the encrypted k with SK_{B}.
6. A computes ciphertext c of message m with k.
7. A signs the hash of c.
8. B verifies the signature.
9. B decrypts m using k.

This series of steps tests the developers ability to send confidential, authenticated messages. In implementing these features, we found a number of issues with the libraries. From an implementation perspective, the libraries had a number of small issues that encourage cryptographic misuse. In the next section, we explore the various shortcomings these libraries contained, and how they lead to developer misuse.

5 Points of Interest

5.1 IV Reuse

Importance to Security

As mentioned in the topics outlined above, cryptographers have developed modes of operation to handle variable length input. For these modes of operation, they all require an initial "seed" to start the encryption called the Initialization Vector (IV).

Reusing an IV value makes many modes of operation IND-CPA insecure. The IV is the only source of randomness in the encryption scheme, so a constant IV implies a constant encryption function. The adversary can easily tell if two messages are the same by comparing the ciphertexts. Randomizing the IV eliminates this problem, and is the suggested procedure when using most modes of operation. Using a randomized public IV is much easier than reestablishing a new shared secret key for each message.

In examining the libraries listed above, we found that several libraries encourage bad usage of IVs. From this point on, when referring to any code snippets, we will rewrite them in a Python-ic notation for consistency and general readability.

Case Studies

Dangerous – PyCrypto In PyCrypto, IVs are not a required parameter in the block cipher encryptions. By itself, this is not a bad standard. However, PyCrypto sets the default IV value to 0.

```
def AES(key, mode, IV = 0):
```

Developers are unlikely to supply arguments that are not required or that they don't understand. If developers use the default IV in their applications, adversaries will be able to differentiate between different encrypted messages. Inherently, the implementation of PyCrypto encourages developers to build cryptographically insecure applications.

Questionable – Go Go Crypto remedies the mistakes PyCrypto makes. Rather than encouraging lazy developer behavior, developers are required to provide an IV. This way, they are required to consider what a proper IV would be.

```
def AES(key, mode, IV):
```

However, as evidenced by code found on GitHub, people are willing to supply the same constant IV to every encryption call.

Safe – Java Java succeeds where the others libraries fail. Since the IV is ideally a random value, there is no reason that a developer should absolutely have to supply this value. Rather, it should happen under the hood and should be cryptographically secure. The Java Cryptography Architecture was developed with this philosophy, setting the IV to a securely generated random number by default.

```
def AES(key, mode, IV = SecureRandom.newRandom())
```

The lazy developer will not run into security issues using the Java Crypto AES, since the defaults are cryptographically secure.

Best Practice

Choosing an IV is a delicate process. For example, CFB only requires a non-repeating IV, while CBC requires a random one [8]. By default, libraries should generate a random IV of the proper length themselves at the encryption stage.

5.2 Library Defaults

Importance to Security

Library defaults are critical when building a cryptographic library. Developers are likely to use the defaults, as many have little-to-no knowledge of the various cryptographic schemes. Libraries can easily promote best practices by setting good defaults. We examined each of the cryptographic libraries, and found several examples of defaults that promote misuse.

Case Studies

Dangerous – PyCrypto PyCrypto uses Electronic Codebook (ECB) as its default block cipher mode for symmetric key encryption. ECB mode is generally discouraged, since it leaks the frequency of blocks in a message, as shown in Figure 2.

Now, consider a more secure cipher mode. PyCrypto has a Counter object that is required for Counter (CTR) mode. Counter values should never be reused — reusing a counter value and key can result in the actual plaintext being deciphered. PyCrypto sets the default Counter value to 1, however, instead of a random integer. This forces developers to seed the counter themselves, which results in misuse. Here is an example from a popular Python library called Beaker [10]:

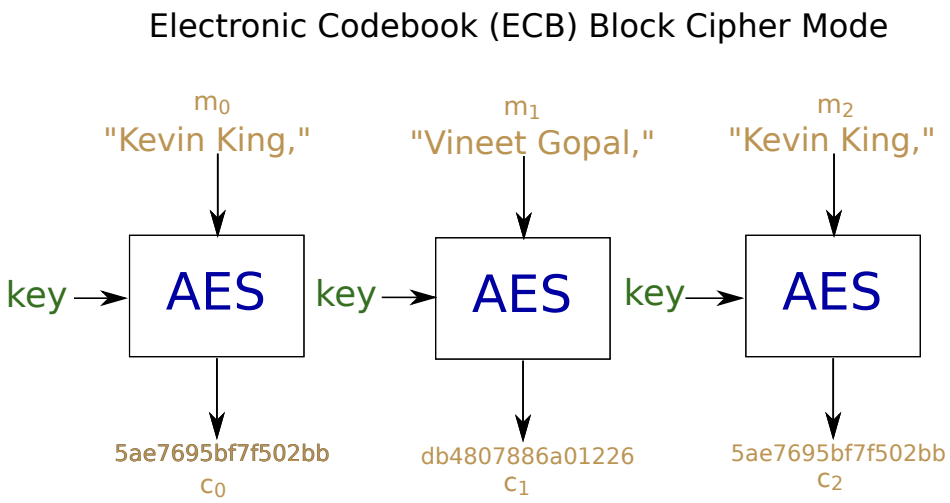

Figure 2: Electronic Codebook (ECB) cipher mode generates a ciphertext only dependent on the key and message block, and common message blocks encrypt to common ciphertexts, revealing information about the plaintext.

```
def aesEncrypt(data, key):
    counter = Counter.new(128, initial_value=0)
    cipher = AES.new(key, AES.MODE_CTR, counter=counter)
    return cipher.encrypt(data)
```

If multiple messages are encrypted using the same key, the encryption becomes very easy to break. In fact, given any message and ciphertext combo, we can decrypt all messages using this scheme. PyCrypto encourages this behavior by not setting the default as random.

Questionable – Go Go does not specify defaults for several methods, including symmetric key encryption. This forces developers to read the documentation to choose a method of encryption. This is better than offering a bad default, like PyCrypto, but still encourages misuse by putting the burden on the developer to choose methods appropriately.

Safe – NaCl NaCl uses a CTR mode as its default block cipher mode, rather than ECB. This encourages correct use of the encryption scheme by promoting a (more) secure block cipher mode as the default.

Best Practice

Libraries should always build in the most secure option as the default. For example, IND-CPA secure CBC mode with random IV should be the default cipher mode for symmetric-key encryption.

5.3 Feature Bloat

Importance to Security

Cryptographic libraries have to support a variety of developers and established protocols. For backwards compatibility, they have to implement all of the cryptographic primitives, even when they are insecure or deprecated. The libraries we examined do this in a variety of ways.

Case Studies

Dangerous – OpenSSL OpenSSL does not indicate which methods are preferred vs. deprecated. Even when developers read the documentation, they have no way of knowing which encryption methods to use, or which block cipher modes to use.

Safe – PyCrypto PyCrypto implements all of the various encryption and hashing methods, but has clear documentation of which ones are deprecated. The documentation itself suggests using AES instead of DES, and to avoid hashing methods like SHA-1. This gives developers an easy mechanism of determining which functions to use when designing a new library.

Safe – Crypto++ Crypto++ namespaces deprecated methods directly in the API. For example, to use the MD5 hash function, the developer has to write Weak1::MD5() in their code. This discourages misuse by making developers directly aware of the security implications.

Safe – NaCl, Go NaCl and Go get around feature bloat by simply removing insecure methods. Instead of offering backwards compatibility, they designed a library that forces developers to use best practices. An issue submitted to Go asking for ECB support received the response: "Why? We left ECB out intentionally: it's insecure" [11]. While this makes the library much less flexible, it does promote cryptographic security.

Best Practice

Libraries should remove outdated, insecure methods. If they must keep backwards compatibility, then they should either mark those methods insecure, in code, or emit warnings whenever the methods are called (similar to our `pycrypto_lint` solution).

5.4 Feature Incomplete

Importance to Security

Feature bloat means including old, insecure, and unnecessary primitives. On the other hand, a feature-incomplete library excludes new, secure, and necessary primitives. In particular, motivated by assertions that authentication is required for most applications [12], we focused on new schemes that provided both confidentiality and integrity. There have been recent developments in designing authenticated encryption modes for block ciphers, which provide both. An example is Galois/Counter Mode (GCM), which combines counter mode with a hash-based MAC. While NIST standardized the traditional encryption-only modes like CFB and CBC as early as 1980 [13], it only standardized GCM in 2007 [14].

Having a single cipher mode that both encrypts and authenticates is safer than letting the developer combine an encryption scheme and an authentication scheme. As an example of the latter, Encrypt-and-MAC ($E(m)\|\text{MAC}(m)$) can be IND-CPA insecure when the ciphertext is distinguishable based on the appended MAC. So, the order of operations matters: Encrypt-then-MAC is preferred to MAC-then-Encrypt and Encrypt-and-MAC [15]. Even then, Encrypt-then-MAC can be easily misused — Namprempre, Rogaway, and Shrimpton [8] show that it is insecure when the MAC doesn't cover the IV ($\text{MAC}(c)$ instead of $\text{MAC}(IV\|c)$).

Case Studies

Dangerous – PyCrypto PyCrypto's online API does not include any cipher modes for authenticated encryption, like GCM, EAX, or OCB. Proposed replacements to PyCrypto cite this as an issue they address [16]. As a result, developers must combine encryption and authentication themselves.

Safe – Java Java, as well as OpenSSL, Crypto++, and Go, provide authentication encryption modes like GCM. While the modes are not the default (instead, ECB is), they are mentioned at the end of block cipher documentation as providing confidential data and authenticity assurances.

Safe – NaCl While NaCl has not yet implemented AES in GCM mode, it provides authenticated encryption by default: the library designers packaged together their symmetric-key encryption scheme and one-time authentication scheme into a single crypto_secretbox function.

Best Practice

Libraries should stay up-to-date with the newest standards, especially when those standards improve security for developers.

5.5 Documentation

Importance to Security

Even with well-written cryptographic APIs at their disposal, developers need more than function signatures in order to accomplish their security goals. The first interaction between the developer and a cryptographic library is the library's documentation. The developer most likely has a specific feature in mind that he wishes to implement as soon as possible. If a developer jumps into using a library without understanding the underlying security concepts, the chance that he makes a mistake is much higher.

While one option is for the developer to buy a recent security textbook and read and understand the relevant chapters, very few developers have the time or patience to understand the cryptographic primitives at this level of detail. Instead, developers are likely to take the path of least resistance and use the first primitive they come across that seems to solve their problem.

In addition to educating developers on the primitives presented, a library's documentation should follow best practices of readability and completeness. The landing page of a library's documentation should quickly direct developers to the correct area of the library that solves their cryptographic problem (i.e. symmetric authenticated encryption, public key encryption, cryptographic hashing).

Case Studies

Dangerous – Crypto++ The Crypto++ library is a relatively large c++ codebase, and the documentation was generated by the widely-used doxygen documentation tool. The developers of Crypto++ made liberal use of templates. As an example, the only documentation associated with a counter mode block cipher encryption method is the following type signature:

```
typedef CipherModeFinalTemplate_CipherHolder<
   typename CIPHER::Encryption, ConcretePolicyHolder<
      Empty, AdditiveCipherTemplate<
         AbstractPolicyHolder<
            AdditiveCipherAbstractPolicy, CTR_ModePolicy >>
      >>> Encryption
```

No information is provided about how counter mode is IND-CPA secure when using a new initial counter value for every message sent.

Dangerous – Bouncy Castle Java Provider While feature-complete, the Bouncy Castle library contains approximately 1500 classes in its entirety. The resulting documentation is an innavigable JavaDoc index of these classes. Developers must guess a substring of the name of a class they are interested in in order to find the associated documentation.

Questionable – OpenSSL OpenSSL's cryptographic library contains more approachable documentation, breaking the API into Symmetric Ciphers, Public Key Cryptography and Key Agreement, Certificates, Authentication Codes, and Hash Functions. Beneath each of these categories, developers find the list of included primitives. While OpenSSL provides some notes at the bottom of the page on how DES and the ECB cipher mode are insecure, it does not recommend alternatives or direct developers to secure primitives.

Questionable – Go Go's cryptographic library contains accessible documentation similar to OpenSSL, but fails to discuss the security of any of the primitives such as DES. The documentation only references a specification number by name, forcing the developer to lookup the current accepted security of this specification.

Safe – PyCrypto PyCrypto strikes a productive balance of accessibility and education with an unbloated PyDoc repository. If a module requires extra care, a bold warning at the top of the page recommends an alternative module to use. For example, PyCrypto recommends the OAEP module over vanilla RSA.

Safe – Oracle Java Provider Oracle's own Java cryptographic library includes a complete developer guide to security. Issues regarding key storage, secure random number generation, and symmetric versus asymmetric encryption are described in an understandable manner and coupled with best practices in using each primitive.

Best Practice

Libraries should have clear documentation for developers, specifying all methods and parameters. They should carefully explain parameters that directly impact the security of the program using the methods.

5.6 Code Samples

Importance to Security

Code samples often prove to be more influential on how a developer uses a library than documentation. If a code sample using a cryptographic primitive seems to meet a developer's needs, there is a high chance that he will copy and paste the code sample into his own

source. This action becomes dangerous when code samples contain errors or unsafe practices, especially if the example lacks a comment explaining the risks. The best code samples succinctly educate a developer and incorporate safe practices.

Case Studies

Dangerous – OpenSSL While unit tests make great code samples in many types of software projects, this is not always true for cryptographic primitives. OpenSSL's AES demo hard-codes the key and initialization vector for its encryption and decryption tests, calling these hard-coded values "NIST public test vectors" at the top of the file [17]. Unfortunately, developers may see the NIST approval and believe that these are in fact secure values to use.

Questionable – PyCrypto While most of PyCrypto's code samples exhibit safe practices, some examples are simply incorrect. The code sample displaying how to sign and verify a signature using PKCS1_PSS [18] contains multiple conceptual errors, including replacing a local variable name with the global package name. If a developer were to try and use this example, it would just throw an error.

Safe – Oracle Java Provider The code samples in Oracle's extensive guide to security thoroughly explain and demonstrate safe key generation, as well as correct parameter initialization for AES block cipher modes [19]. The examples also correctly handle exceptions, providing developers with informative feedback when an error occurs in the protocol.

Safe – Go Go's native cryptographic library provides both safe examples and informative comments. When presenting AES CFB encryption, the code sample uses an obviously fake key of "example key 1234," generates a random initialization vector, and explains the nature of initialization vector security, as well as the importance of authenticating in addition to encrypting [20]. A developer that uses this example is likely to correctly generate his key and initialization vector, as well as look into authentication in addition to encryption.

Best Practice

Libraries should only feature code samples that are model examples of library usage, because that is how developers will use them.

5.7 Programming Languages

Importance to Security

Even before selecting a cryptographic library, developers decide on a primary programming language for their project. The design of the selected language has direct implications on program security, as some languages lack specific compiler or runtime checks.

Case Studies

Dangerous – C One cannot deny that some of the most important and widely-used software in the world today is written in C. C exposes more fine-grained control over the computer's memory than other programming languages, and was popularized because of its speed and basically being invented in the right place at the right time.

The first danger of C is its lack of bounds checking. Uncareful developers are susceptible to buffer overflow attacks even in code unrelated to security. C also does not strictly type inputs to functions, and actually lacks a "byte" type. Thus subtle errors can occur when the developer and designer of a library do not agree on either "char" or "unsigned char" for the type to be used to represent a byte. A chronic example of byte type disagreement occurred when a developer tried hashing a file with OpenSSL's SHA256, but saw that half of the bytes of the hash were the same [21]:

```
void sha256_hash_string (char hash[SHA256_DIGEST_LENGTH], char outputBuffer
    [65]) { ... }

// Actual (output) hash:
6dff32ffff59191f3eff6afffff06ffff1e65460d54ffff760b033fff16ff3866
// Expected hash:
6da032d0f859191f3ec46a89860694c61e65460d54f2f6760b033fa416b73866
```

This programming error yields a hash construction with approximately half of the "bits" of secure collision resistance as the correct construction. With the lack of exceptions, C also provides no standard in checking for errors (present in error codes in function return values). OpenSSL's return codes actually disagree with UNIX shell return codes, sometimes indicating success when a failure actually occurred [22].

Questionable – Python Python's bounds checking and exceptions mitigate the risk of buffer overflows and ignored errors during execution. Python does not have strict typing, however, so developers often have to determine information directly from the function parameter names.

Safe – Java, Go, C++ All of Java, Go, and C++ offer strict typing, as well as some form of native error handling. All three languages also offer bounds checking (when using std::string in C++). While the C++ and Java languages can lead to overbearing template

or generic arguments, we view interface minimalism and approachability as a responsibility of the library.

Best Practice

Libraries in non-C programming languages should use the correct byte type, check bounds, throw exceptions in case of error, and be minimalist. Libraries in C should alias unsigned char to a usable byte type, have an input length parameter to check, have standard return codes, and be minimalist.

6 Recommendations

For each point of interest that we study, we have included a best practice for library designers. We have summarized our findings in Table 3. We recommend library designers to model their libraries after NaCl, as it satisfies most of our points. Overall, we observe that while clear documentation and secure code samples are important, developers may still accidentally skip over them. So, the most crucial part is the code: the API design. The API should be easy to use (and hard to misuse) even without documentation.

NaCl exemplifies this, with a crypto_box function for public-key authenticated encryption and crypto_secretbox function for symmetric-key authenticated encryption. Its design suggests this recommendation: simplify the most common use case.

One way to do this is by having a strong default (like AES/GCM) and only enabling the less-used methods through optional parameters. Another way is to have a low-level API accompanied by a high-level one. There are unofficial wrappers for OpenSSL, but the official one, called the EVP "envelope" interface, is still complex (nearly 100 routines for just symmetric-key encryption [23]).

Point of interest	OpenSSL	Crypto++	NaCl	Java	PyCrypto	Go
IV reuse	?	?	?	✓	✗	?
Library defaults	✗	?	✓	✗	✗	?
Feature bloat	✗	✓	✓	✗	✓	✓
Feature incomplete	✓	✓	✓	✓	✗	✓
Documentation	?	✗	✓	?	✓	?
Code samples	✗	✓	✓	✓	?	✓
Programming languages	✗	✓	✓	✓	?	✓

Table 3: A summary of the points of interest per library.

7 PyCrypto Linter

The issues discussed in this paper stem from various sources — a need to maintain backwards compatibility, a desire to be flexible, a lack of developer resources, etc. Several of these libraries have wrappers that try to encourage correct use of the protocols, but these are often susceptible to feature bloat.

We propose a different solution which attacks the problem directly. Our solution is an optional linter built into the library that checks for various misuses of the library. This linter could be used in development mode, and logs any misuses to the standard error of the program.

A simple linter could detect the following issues with a program:

- Encrypted data that is not authenticated
- IV reuse (or IV's without enough entropy for CBC)
- Using DES for encryption
- Using ECB as a block cipher mode
- Not using the random number generator when needed
- Reusing counter values with the same encryption key

As a proof of concept, we implemented `pycrypto_lint`, a linter for the PyCrypto library (`https://github.com/vineetgopal/pycrypto_lint`). This linter checks for IV reuse, counter reuse, and ECB mode. When it detects an issue, it prints out the security vulnerability and the stack trace associated with it. This allows the developer to easily find and fix the problem.

This is an example of a developer using `pycrypto_lint` to check their use of cryptography:

```
>>> enc1 = AES.new(key)
    'PYCRYPTO_LINT ERROR: Default AES mode (ECB) is unsafe'
>>> enc1 = AES.new(key, MODE_CBC, IV=iv)
>>> enc2 = AES.new(key, MODE_CBC, IV=iv)
    'PYCRYPTO_LINT ERROR: Reuse of IV (should be generated by Random)'
```

This is meant to be used during development, to give developers real time feedback on whether their application is secure. Obviously, there are ways to misuse the library that cannot be checked by linter, but this will provide one more wrapper of security around the library. Such linters could easily be built for the other cryptographic packages.

8 Conclusion

In this report, we highlight the astonishing disconnect between the intended use of cryptographic primitives and how they are actually utilized in practice. We recognize that the

library which a developer selects is a significant contributing factor to whether a developer commits cryptographic misuse. We break down the relevant properties of cryptographic libraries into seven common issues and analyze dangerous, questionable, and safe methods libraries use to address these issues. Taking all of our findings into account, we consider NaCl cryptographic library to be a model library and make recommendations to all developers of cryptographic libraries. We also provide a proof-of-concept cryptographic linter for PyCrypto, `pycrypto_lint`, which warns developers at runtime of possible common instances of cryptographic misuse. We hope future libraries heed our advice and eliminate the need for any such tool, but for now provide this stopgap solution.

References

[1] Cairns, K., & Steel, G. Developer-Resistant Cryptography. https://www.w3.org/2014/strint/papers/48.pdf.

[2] Cryptography package, https://cryptography.io/.

[3] Bernstein, D. J., Lange, T., & Schwabe, P. (2011). The security impact of a new cryptographic library. *IACR Cryptology ePrint Archive*, 646.

[4] Heartbleed, http://heartbleed.com/.

[5] ECRYPT Benchmarking of Cryptographic Systems, http://bench.cr.yp.to/.

[6] Speedtest and Comparsion of Open-Source Cryptography Libraries and Compiler Flags, https://panthema.net/2008/0714-cryptography-speedtest-comparison/.

[7] Egele, M., Brumley, D., Fratantonio, Y., & Kruegel, C. (2013, November). An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (pp. 73–84). ACM.

[8] Namprempre, C., Rogaway, P., & Shrimpton, T. (2014). Reconsidering generic composition. In *Advances in Cryptology–EUROCRYPT 2014* (pp. 257–274). Springer Berlin Heidelberg.

[9] NIST Selects Winner of Secure Hash Algorithm (SHA-3) Competition, http://www.nist.gov/itl/csd/sha-100212.cfm.

[10] Beaker Python Package, https://github.com/bbangert/beaker/blob/master/beaker/crypto/pycrypto.py.

[11] Go Crypto Issues, https://code.google.com/p/go/issues/detail?id=5597.

[12] Why Encryption without Authentication is Not Secure, http://www.certesnetworks.com/newdocs/wp-authentication.html.

[13] DES Modes of Operation, http://csrc.nist.gov/publications/fips/fips81/fips81.htm.

[14] Dworkin, M. (2007). Recommendation for block cipher modes of operation: Galois/-Counter Mode (GCM) and GMAC. US Department of Commerce, National Institute of Standards and Technology. http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf.

[15] Bellare, M., & Namprempre, C. (2000). Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In Advances in Cryptology—ASIACRYPT 2000 (pp. 531-545). Springer Berlin Heidelberg. http://cseweb.ucsd.edu/~mihir/papers/oem.pdf.

[16] Cryptography Documentation, http://media.readthedocs.org/pdf/cryptography/latest/cryptography.pdf.

[17] OpenSSL AESCCM, https://github.com/openssl/openssl/blob/master/demos/evp/aesccm.c.

[18] PyCrypto - The Python Cryptography Toolkit, https://www.dlitz.net/software/pycrypto/api/current/Crypto.Signature.PKCS1_PSS-module.html.

[19] Using the Java Cryptographic Extensions, https://www.owasp.org/index.php/Using_the_Java_Cryptographic_Extensions.

[20] Go Crypto Documentation, http://golang.org/pkg/crypto/cipher/.

[21] StackOverflow, http://stackoverflow.com/questions/7853156/calculate-sha256-of-a-file-using-openssl-libcrypto-in-c.

[22] OpenSSL Project, http://openssl.6102.n7.nabble.com/openssl-verify-always-returns-0-success-to-shell-td29168.html.

[23] OpenSSL EVP_EncryptInit, https://www.openssl.org/docs/crypto/EVP_EncryptInit.html.