RE of a Solution to the problem of misuse of Cryptographic APIs

Presented By: Prasanna Kumar

Outline

- Introduction
- Problem
- Requirements for an optimal Solution
- Current Solutions
- Conclusion

Introduction

Application Programming Interfaces

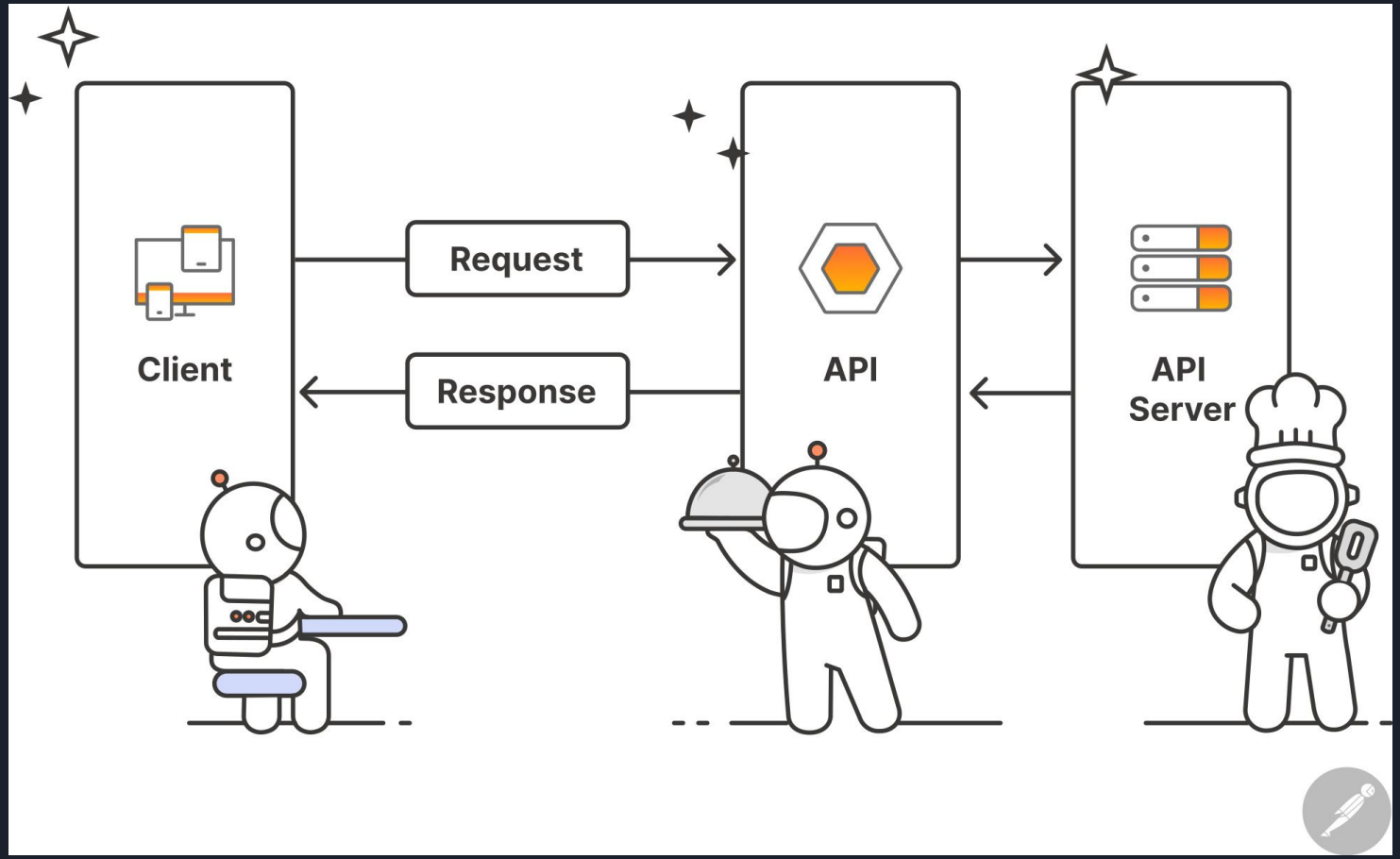
APIs or Application Programming Interfaces are simply put : mechanisms that allow two or more software components to communicate with each other

These communications take place using a set of definitions and protocols

API architecture can be thought of as a client and server model: the application sending the request is like the client and the application sending the response is the server

An example:

When we want weather updates, our phone's weather application talks to the weather software system (of some organization) via APIs and then shows us the updates


Client
Request
Response
API
API
Server

Cryptographic APIs

Cryptography is the study of methods and techniques that allow us to ensure that the content of the message are viewed only by the intended parties (sender, intended recipient, etc.)

Today, we have a lot of applications where confidentiality is important or cases where the data needs to be protected

These could range from basic password protection to hiding the content that we are accessing over the internet

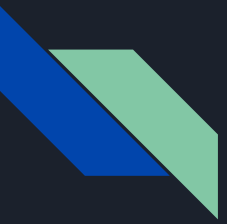
Cryptography has become an essential tool to fulfill many such objectives (like maintaining data privacy, authenticating the parties that are communicating are who they say they are, maintaining data integrity, etc)

This brings us to : Cryptographic APIs

These are basically APIs that provide cryptographic services

They are really useful and beneficial for developers

Examples of cryptographic services APIs include: Authentication APIs, Key Management APIs, Key Generation APIs, Encryption and Decryption APIs, etc



Now, a major functionality of any API is that one doesn't need to understand the inner details of its working in order to use it

Only the parts relevant to whatever is going to access the API is exposed, rest of the details aren't visible to the client that is accessing/using the API

So, Developers with no knowledge of cryptographic principles and primitives can still use cryptographic APIs to utilize cryptographic services in their application

This is what makes them really important in modern day application development environment

Problem

Misuse of Cryptographic APIs

This raises the question: is the data being secured in a sensible manner?

As we will see in the upcoming slides, the answer to this question is often : No.

A vast majority of applications actually misuse the cryptographic APIs and these can result in major security issues

For example, we could have use of insecure cryptographic primitives or the random number we use isn’t that random, etc..

Usually, this happens as the developers that are making these applications don’t have any sort of cybersecurity training and have limited knowledge of proper use of cryptographic primitives

Often, these projects need to be completed on strict deadlines

So, little to no effort is put in by the developers to improve their knowledge for long term benefits

This results in misuse of these APIs

Most of these vulnerabilities arise from programming mistakes and not the underlying algorithms

Common Categories of such Misuse in Java

Cryptographic Keys: If the adversary can easily read the cryptographic key due to it being predictable or constant, it is possible to obtain potentially sensitive information

Passwords in Password-Based Encryption: In certain APIs like “javax.crypto.spec.PBE-KeySpec API”, a user may hardcode the password that is used to generate the secret key: in this case the malicious attacker can easily predict the key if they obtain the password

Passwords in KeyStore: Similar to above, hardcoded passwords can allow adversary to get access to stored keys and certificates if the KeyStore API is being used

Credentials in String: Any sensitive information should be put in data structures such as byte or char array and cleared after use

Hostname Verifier: If not arranged properly, this makes it simpler to orchestrate URL spoofing attacks

Certificate Validation: Empty methods used in TrustManager interface to connect quickly and easily with client and remote servers (skipping certificate validation). This makes it so that the TrustManager accepts and trusts every entity and makes system vulnerable to MiTM attacks

SSL Sockets: Some incorrect implementation of SSLSocket skip the host name verification when socket is created

HTTP: We should use HTTPS if we need to encrypt sensitive information

Pseudorandom Number Generator: Generation of pseudorandom number isn't entirely random if we use java.util.Random

Seeds in PRNG: The seed that is used in java.security.SecureRandom needs to be a non deterministic random seed

Salts in PBE: Salt should be a random number so as to produce a random and unpredictable key

Mode of Operation: Electronic Codebook mode (ECB) is known to be insecure as ECB-encrypted Ciphertext can leak information about the plaintext

Initialization Vector: While CBC is more secure, we need to ensure that the IV used during encryption and decryption isn't constant/static

Iteration Count in PBE: Number of iterations used in PBE should be greater than 1000

Symmetric Cipher: Symmetric Cipher use same keys for encryption and decryption. AES is a secure alternative to DES or RC4

Asymmetric Cipher: In asymmetric cryptography, we have a public key and a private key.

Cryptographic Hash Functions: this function generates a fixed-length alphanumeric hash value commonly used in verifying integrity of message, signature, etc.. It is considered to be broken if a collision can be observed

Cryptographic MAC: HmacMD5 and HmacSHA1 are considered to be insecure as they are susceptible to collision attacks.

Requirements for an optimal Solution

Potential Requirements for any solution

Overall, we want any solution to fulfill the following requirements in order to be considered to be an efficient solution that is deployable in the real world

1) Complete: We want to be able to detect all the misuses even if they seem insignificant or aren't completely breaking security
2) Scalability: We want to be able to analyze large code-bases with minimal overhead
3) Efficiency and Automation: These are not necessarily core requirements but considering how large code-bases can get, they might be essential for any solution to be viable at a commercial level
4) Extensible: Any solution that is based on a set of rules that must be followed to maintain security should be possible to be modified with minimal work in order to incorporate more rules in the future

Current Solutions

Static Analysis

Over the years, the security community has produced a lot of impressive static analysis tools as well as dynamic code screening tools.

The difference being that static analysis doesn't require the program to execute and works on a version of the code

Many security rules that may seem abstract can be reduced to concrete program properties which can be then enforced using Static Analysis

Dynamic Tools, on the other hand, require execution of program and spend resources to trigger and detect misuse symptoms at runtime.

So, Static Analysis is usually favored.

Weakness of Static Analysis

Static Analysis tools tend to produce false alerts

False positives can greatly reduce the reliability and effectiveness of such tools

So we usually need to have a tradeoff between

a) Completeness
b) Scalability

Example of a Static Analysis tool : CryptoLint, FixDroid

CryptoLint

An Empirical Study of Cryptographic Misuse in Android Applications

The authors wanted to conduct a study on the applications found on Android in terms of their soundness when it comes to utilization of cryptographic APIs

In order to do so, they developed tool called CryptoLint

It is a light-weight static analysis tool that can check the applications for common flaws.

It takes a raw Android Binary, disassembles it and then checks for cryptographic misuses.

Defining Correct Behavior

CryptoLint uses a set of security properties to evaluate applications and these rules maybe a good starting point to start from for any framework that deals with this problem

These rules are as follows:

1) Do not use ECB mode for encryption
2) Do not use a non-random IV for CBC encryption
3) Do not use constant encryption keys
4) Do not use constant salt for PBE
5) Do not use less than 1000 iterations for PBE
6) Don’t use static seeds SecureRandom()

Results from CryptoLint

CryptoLint analyzed 11,748 applications

88% applications misused cryptographic APIs in some way

Number of Apps	Rule Violated
5,656 + 2000	Uses ECB
3,644	Uses Constant Symmetric Key
1932	Uses Constant IV
1,636	Used iteration count <1000 for PBE
1629	Static SecureRandom Seed
1,574	Uses Static Salt for PBE
1,421	No Violations

Is that enough?

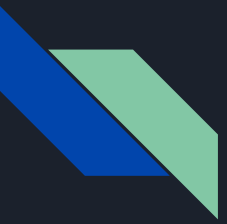
The answer is a resounding no.

As we have seen, there are a huge number of categories that are often seen to cause security flaws when using cryptographic APIs.

The reasons for this are that these APIs vary in how they work, poor documentation, the default parameters may not be optimal, etc.

So, we need to make a system that can improve over time and allows additions of rules

Any of these static analysis tools can be modified and with a proper set of ideal behavior, we may be able to achieve what we desire



Some properties may also vary depending on the implementations.

For example, we state the IV should be unique in one of the rules. However, this is not globally valid. Kerberos uses a static IV with CBC block cipher mode but also discards the first block (as it is filled with random data) and therefore, the role of IV is basically fulfilled by first block. Inferring this as valid would require far more complex systems than a simple static analysis tool can provide.

Other Potential Solutions

A Machine Learning approach

SCAT adopt basic pattern matching techniques to detect misuses.

Problem with this system is that such tools aren't really good at detecting all the misuse and rather only the ones that they are meant to check for (based on the criteria used in order to make the tool)

However, an application of existing ML techniques to build binary classifiers for such detection system has been seen to achieve about 87% detection

However, not a lot of work other than this, has been done in this field and we could explore the potential of ML for solving this issue.

Closing Thoughts

What do developers feel about this?

It has been observed that most developers don't seem to be bothered by these studies.

In particular, Ying Zhang and others, recorded several responses from developers and found that

a) Majority of the developers don't believe these vulnerabilities to be important
b) Most of them didn't want to address the reported problems
c) Almost none of them wanted to follow the tool-generated guidance
d) Mostly, this was chalked up to "we haven't seen security exploits yet" so it must be alright
e) As stated earlier, most static tools don't consider context so a lot of developers believed that the security threats detected won't have any real world security consequence

Conclusion

We have seen that cryptographic APIs are often misused by developers due to

a) Lack of Knowledge
b) Lack of proper cybersecurity training due to strict deadlines

It seems that with a framework that follows the requirements listed earlier could provide a feasible and reasonable solution

However, most existing solutions don't provide all 4 requirements due to the very nature of their implementation

Future works could work with ML or Type Checking in order to achieve all 4 goals to a reasonable level of satisfaction

References

1. What is an API? , https://aws.amazon.com/what-is/api/#seo-faq-pairs#what-is-an-api
2. What is an API? , https://www.postman.com/what-is-an-api/
3. Cryptographic Services APIs, https://www.ibm.com/docs/en/i/7.3?topic=ssw_ibm_i_73/apis/catcrypt.html
4. S. Afrose, Y. Xiao, S. Rahaman, B. P. Miller and D. Yao, "Evaluation of Static Vulnerability Detection Tools With Java Cryptographic API Benchmarks," in IEEE Transactions on Software Engineering, vol. 49, no. 2, pp. 485-497, 1 Feb. 2023, doi: 10.1109/TSE.2022.3154717.
5. Manuel Egele, David Brumley, Yanick Fratantonio, Christopher Kruegel, “An Empirical Study of Cryptographic Misuse in Android Applications”
6. Gustavo Eloi de P. Rodrigues, Alexandre M. Braga, Ricardo Dahab, “A machine learning approach to detect misuse of cryptographic APIs in source code”

7. Ying Zhang, Mahir Kabir, Ya Xiao, Danfeng (Daphne) Yao, Na Meng, Automatic Detection of Java Cryptographic API Misuses: Are We There Yet?
8. Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Danfeng (Daphne) Yao, Murat Kantarcioglu, CRYPTOGUARD: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects