Cybersecurity MSc: Applied Cryptography 2022-23

Practical Exercises: Cryptographic hashing: digests and MACs

September 8, 2022 Due date: no date

Changelog

- v1.0 - Initial version.

Introduction

In order to elaborate this laboratory guide it is required to install the Java Development Environment (JDK), Python 3 or the C compiler and development environment.

The examples provided will use both Java, Python and C. For Python you need to install the `cryptography` module. For C we will use the EVP (Digital Envelope) functions from the Crypto library that is part of OpenSSL, and for that you need to use install the packages `libssl1.0-dev` and `libssl1.1`.

1 Elements of interest for each language

1.1 Java

- java.security.**MessageDigest** An instance of the `MessageDigest` allows to calculate digests of arbitrary data. Some important methods are: `getInstance`, `update` and `digest`.
- javax.crypto.**Mac** An instance of the class `Mac` allows to calculate a MAC of arbitrary data given a key. Some important methods are: `getInstance`, `init`, `update` and `doFinal`.

1.2 Python

- The `cryptography` module is both a frontend, high-level interface for dealing with cryptography and a default backend implementation of the fundamental cryptographic primitives. The high-level interface allows explore a backend other than the default. A reference to the default backend is fetched with the function `default_backend`.
- The Python interpreter distinguishes bytes from text characters, and thus byte arrays from strings. A string is more than a byte array, it has also an encoding (e.g. UTF-8, Unicode, etc.). You can get the bytes of characters (strings) using the `bytes` function, and you can convert bytes to characters (strings) by using the method `decode` on a byte array (with a target encoding).

  By default, in cryptography we always use bits and bytes, nothing else. So, do not attempt to solve interpreter errors by changing everything to text (characters and strings)!

1.3 C

- Functions `EVP_`$\cdots$`_ex` are functions that can be parameterized with an implementation engine, whilst functions without the `_ex` suffix use the default library engine.
- Functions `EVP_DigestInit`, `EVP_DigestUpdate` and `EVP_DigestFinal` form the high-level interface for handling digest computations.
- Functions `HMAC_init`, `HMAC_update` and `HMAC_final` form the high-level interface for handling HMAC computations.
- A structure `EVP_MD_CTX` describes the current context of a digest computation, and it is created with `EVP_MD_CTX_new` and released with `EVP_MD_CTX_free`. This context is used to store the current state of the hashing process and to perform padding.
- A structure `HMAC_CTX` describes the current context of an HMAC computation, and it is created with `HMAC_CTX_new` and released with `HMAC_CTX_free`. This context is used to store the current state of the HMAC process and to perform padding.
- A structure `EVP_MD` describes generic attributes of a given digest computation. Usually it does not need to be filled manually, one can use predefined functions for that (e.g. `EVP_sha256` returns a pointer to such a structure describing a SHA-256 digest operation).

2 Avalanche effect in digest functions

Develop a program to generate the digest of random byte array. Keep that digest as a reference. Then, change a single bit in the original data, any bit, compute the digest again, and compute how many bits have change from the first one. Repeat this process for several other bits, always relatively to the same original data, and compare the resulting digests with the first one. Produce an histogram with the percentages of bits modified in each computation (you can use a counter per each percentage unit).

3 Avalanche effect in MAC functions

Change the previous program to do the same with HMAC and with bit modifications in the secret key.

4 Birthday paradox

Develop a program that computes digests from a series of fixed-size random byte arrays. Store the digests for detecting collisions and count the number of attempts until finding a collision. Compute the average number of attempts required to find a collision, in order to verify the birthday paradox.

Note: this exercise is infeasible with the current digest sizes, which are above 128 bits. So, parameterize your program to work only with a given number of bits from the resulting digests.

Tip: the birthday paradox applies to any digest function. Therefore, the results should be independent from the digest functions used. Verify it.

References

- The Java Tutorial: Security Features in Java SE
  `https://docs.oracle.com/javase/tutorial/security/index.html`
- Java Documentation: Security
  `http://docs.oracle.com/javase/8/docs/technotes/guides/security/index.html`
- Welcome to pyca/cryptography
  (`https://cryptography.io`)
- Libcrypto API
  (`https://wiki.openssl.org/index.php/Libcrypto_API`)