This repository contains all the deliverables related to the mini project of CS4035D Computer Security.

SHA256

Abstract

Cryptographic hash functions are widely used for digital signatures, password tables, message-authentication codes etc. The Secure Hash Algorithm 256 (SHA-256) and Message Digest (MD5) are examples of such functions. Despite their widespread use, few cryptographers delve into the low-level implementation details of these functions. This project aims to develop a barebone implementation of the SHA256 and MD5, using only basic programming constructs and no in-built libraries. The implementation will be tested for correctness and performance against known test vectors. Furthermore, this project will evaluate the security of SHA256 and MD5 by simulating a length extension attack on both hashing algorithms. With this, we aim to better understand the inner workings of both algorithms and their limitations.

Learnings

- Constants: The first 32 bits of the fractional parts of the cube roots of the first 64 prime numbers is defined. As per reading, we found that this approach is used in some cryptographic applications, specifically in the construction of the SHA-256 hash function. These values are used to "mix" the input message in a way that makes it difficult to reverse-engineer the original message from the output hash. The specific choice of the first 32 bits of the fractional part of the cube roots of the first 64 prime numbers was likely made for a few reasons.
    - Cube roots were chosen because they provide a good balance between randomness and predictability, making them suitable for use in a cryptographic algorithm.
    - Using the fractional part of the cube roots ensures that the resulting values are uniformly distributed between 0 and 1, which is important for maintaining the security properties of the algorithm.
    - Using the first 64 prime numbers ensures that the resulting values are large and unlikely to repeat, further enhancing the security of the algorithm.

Implementation

To run SHA256 checksum of a file:

Help menu:

```
$ python3 sha256.py -h
```

```
usage: sha256.py [-h] -f F

options:
```

```
  -h, --help  show this help message and exit
  -f F        Name of the file to find the checksum
```

Run with a test file:

```
$ python3 sha256.py -f tests/test1.pdf
```

```
7e2903a8c60bf957824c330707617e8cc32283b5287bb9362acb2f45550810c1
```

MD5

Abstract

The MD5 message-digest algorithm is a widely used hash function producing a 128-bit hash value. Despite its widespread use, few cryptographers delve into the low-level implementation details of the SHA256 checksum algorithm.

This project aims to develop a barebone implementation of the MD5 checksum, using only basic programming constructs and no in-built libraries. The implementation will be tested for correctness and performance against known test vectors.

Implementation

To run MD5 checksum of a file:

Help menu:

```
$ python3 md5.py -h
```

```
usage: md5.py [-h] -f F

options:
  -h, --help  show this help message and exit
  -f F        Name of the file to find the checksum
```

Run with a test file:

```
$ python3 md5.py -f tests/test1.pdf
```

```
e2726c121bdb725e83cab7c0166438c0
```

Testing

Using the command line utilities, `sha256sum` and `md5sum`, we can compare the sha256 hash and md5 hash generated by this implementation and verify if it matches.

Download Nessus via:

```
curl --request GET \
  --url
'https://www.tenable.com/downloads/api/v2/pages/nessus/files/Nessus-10.5.0-
ubuntu1404_amd64.deb' \
  --output 'tests/Nessus.deb'
```

```
$ ./test.sh
```

```
--- SHA256 Test ---
Passed: tests/test1.pdf  Time taken: 2.74931 seconds
Passed: tests/test2.txt  Time taken: 0.0456512 seconds
Passed: tests/Nessus.deb  Time taken: 0.0420008 seconds
-------------------

--- MD5 Test ---
Passed: tests/test1.pdf  Time taken: 0.795163 seconds
Passed: tests/test2.txt  Time taken: 0.0367074 seconds
Passed: tests/Nessus.deb  Time taken: 0.0352709 seconds
---------------
```

Here, three files are given as test cases and the hash generated by our implementations from scratch matches the one generated by the built-in utilities.

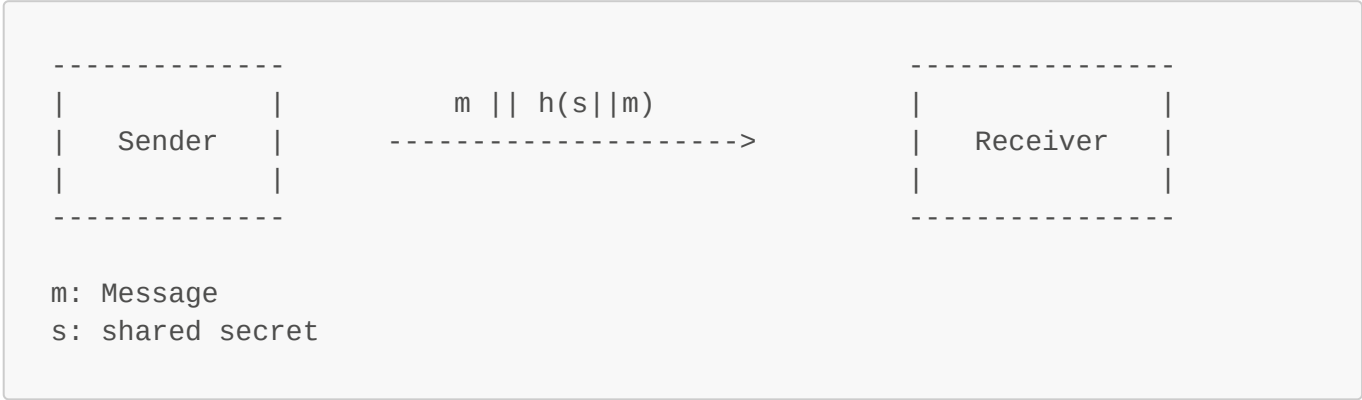
Length Extension Attack (LEA)

This is an attack on hash functions based on the Merkle-Damgard Scheme. For a given message `m`, `H(m)` can be calculated where `H` is the hash function. LEA allows us to use knowledge of `H(m)` and the `len(m)` to calculate `H(m||e)` where || represents concatenation and `e` is an extension to the message. Essentially without knowledge of m, the has of extended messages of m can be calculated. The use of hash functions susceptible to LEA can have serious security concerns.

To illustrate the use of this attack the concept Message Authentication Code and Merkle-Damgard Scheme has been explained below.

Message Authentication Code (MAC)

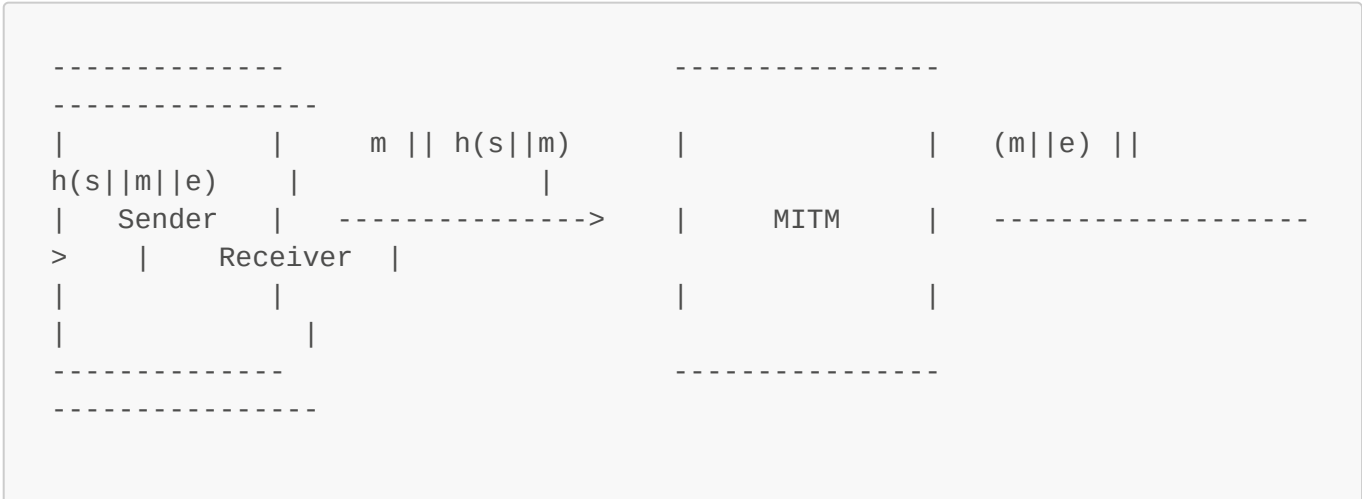
1. Used to authenticate origin and legitamacy of data (m).
2. The shared key (s) is known only to the sender and the receiver and hence they can only compute the hash.
3. The hash h(m || s) is appended to the message when transmitted.
4. The receiver can verify the origin/legitamacy of data.
5. Using LEA susceptible hash functions to generate the MAC is highly insecure.



Merkle Damgard Construction

1. This scheme is used to build a collision-resistant iterated hash function from a collision-resistant compression function.
2. The compression function is a hash function that takes n-bits and m-bits fixed- size inputs.
3. For hashing, the original message is appended with padding (0x80) and then as many 0's as necessary are added till only 8 bytes remain. These 8 bytes are used represent the size of message in bits.
4. The resulting message can thus be split into units of n-bit length. Before hashing, an m-bit digest is chosen as the initial input to the compression function, along with the first n-bit block. This digest is referred to as an initial vector or initial value.
5. The compression function outputs an m-bit digest which is passed as input to the compression function again along with the next block of n-bits.
6. This process repeats till all the blocks of the message are processed to give a final m-bit digest.
7. **Vulnerability:** The output of the hash function can be passed to the compression function along with another n-bit block to obtain the hash for the extended message.

LEA on SHA-256



```
m: Message
s: shared secret
e: extension to message
```

To run LEA Attack on SHA256:

Help menu:

```
$ python3 lea.py -h
```

```
usage: lea.py [-h] -m M -s S -e E

optional arguments:
  -h, --help  show this help message and exit
  -m M        Message to be hashed
  -s S        The secret shared by both sender and receiver
  -e E        The extension to the message
```

Run test:

```
$ python3.9 lea.py -m "user=joel&role=user" -s "hello" -e "&role=admin"
```

```
Original Message to be hashed: hellouser=joel&role=user
MAC for Original Message:
cb8616e7d5e3d5195b5dd9d7c57b4d2ed430ddba3edf5396716fdcc3916fd4b1

-------------------------------------------------

Extended Message to be hashed:
hellouser=joel&role=user\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\xc0&role=admin

MAC for Extended Message:
ff901d5e20f79d990badd0ab7ba1c94f26340ecb235c8644615d747e0c54d227
MAC for Extended Message with LEA attack:
ff901d5e20f79d990badd0ab7ba1c94f26340ecb235c8644615d747e0c54d227

-------------------------------------------------

(LEA successfull) The two hashes are identical
```

Team members

S.L. No.	Name	Roll number	GitHub ID
1	Joel Mathew Cherian	B190664CS	@JoelMathewC
2	Pavithra Rajan	B190632CS	@Pavithra-Rajan
3	Clifford Joshy	B190539CS	@clifordjoshy