Crypto Project

Based on content from UMich's EECS388

Georgia Tech

Project Overview

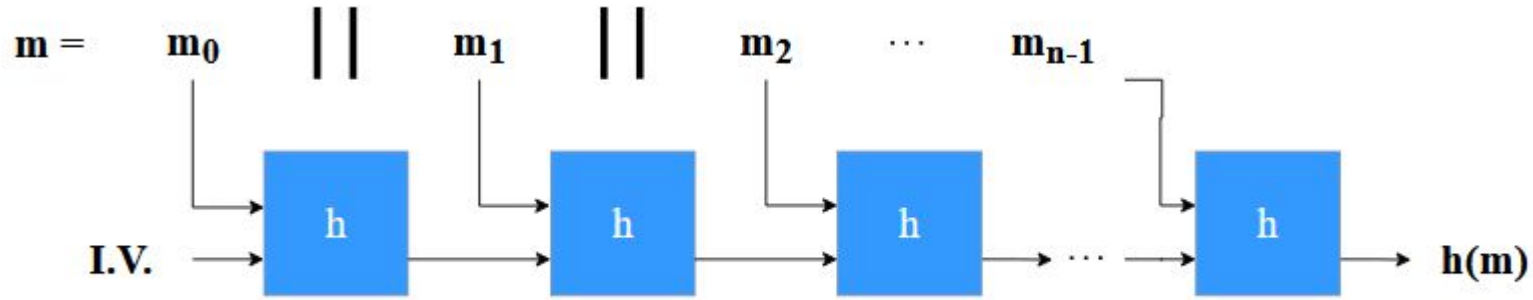
- 8 Tasks
  - Lab assignment parts 1 - 4
- Crypto Project
  - Length extension attack
  - Hash Collision
  - Padding Oracle
  - Bleichenbacher
- Webtesting
  - https://cryptoproject.gtinfosec.org/

Length Extension Attack

Merkle-Damgard Construction Introduction

- Merkle-Damgard Hash Functions (MD5, SHA-1, or SHA-2):
  - Built around a compression function *f* and maintains an internal state *s*
  - Messages are fixed-sized blocks
  - Compression function are applied to the current state and block to compute an updated state, $s_{i+1} = f(s_i, b_i)$

- Let m be the message and m_n be the message blocks
- Let h be the compression function
- I.V. is the initialization vector, a fixed constant to initialize the internal state of the hash function
- h(m) is the final digest or output of the very last application of the compression function



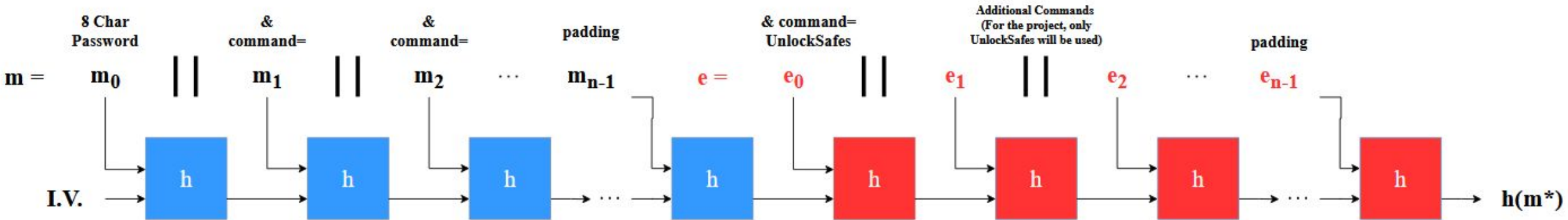
Georgia Tech

Length Extension Introduction

- Merkle-Damgard Hash Functions (MD5, SHA-1, or SHA-2) are vulnerable to Length Extension Attacks
  - If we know the hash of an n-block message, we can find the hash of longer messages by applying the compression function for each additional message block that we want to add

- For the project, the SHA-256 hash function will be used
- SHA-256 requires final message length to be 64 bytes where padding is added to ensure this
- The padding consists of the byte 0x80, followed by as many 0 bytes as necessary, followed by an 8-byte count of the number of bits in the unpadded message

Note: You will only have to compute the first set of padding (use padding() from pysha256) after the original commands, the second set of padding, after the malicious command, will be automatically calculated and appended.



Final Project Verification

- The token needed to unlock the safes for The Bank of GTInfosec will be final hash digest with the appended &command=UnlockSafes
  - token = sha256(8 char password || &command = … || padding || &command=UnlockSafes || padding)

- The url will need to be updated to match the updated token/digest
  - suffix = &command = … + padding + &command=UnlockSafes

Bleichenbacher Attack

RSA Signature Example

- RSA Signature Example:

Let the RSA Key Pair: Public Key: (e, N) Private Key: (d, N)

1) Alice wants to send a message “gtinfosec rul3z!” to Bob
2) Alice hashes the message using SHA-256
3) Alice pads the message using PKCS #1 v1.5 scheme to fit the size of RSA key
4) Alice signs the message using her private key (d) to get s = padded_hash^d (mod n)
5) Bob verifies using Alice’s public key (e), result = s^e (mod n)
6) If the result matches the padded hash, the signature is valid

RSA Signature Forgery

Simple Attacks on textbook RSA

1) Small e value such as e = 3
   - If s^e < n, then s^e does not wrap around the modulus and can be ignored

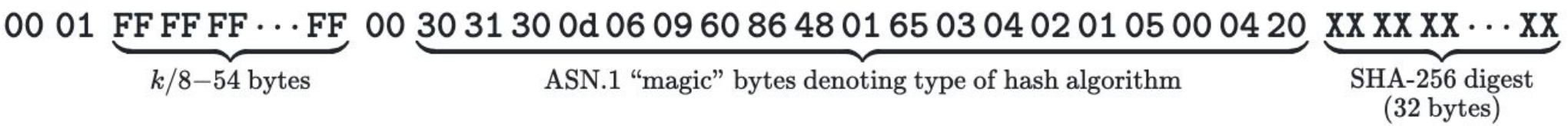
1) Deterministic and structured format of the PKCS #1 v1.5 padding scheme
   - Scheme will always have the header 0x0001ff00, magic bytes indicating hash type, and the hash of the message itself

1) Bad Signature verification
   - If the code verifying an RSA signature does not check bit by bit, a forged signature could bypass the verification (more on the next slides)

Bad Signature Verification

- With an encryption method as seen below, a simple verification would be to

1. Check the total number of bytes
2. Strip the leading 00 01 and then the FF bytes
3. Strip the 00 bytes
4. Parse the ASN and verify the SHA 256 digest of 32 bytes



Georgia Tech

Bad Signature Verification

- This method fails to:
  - Count the number of FF bytes
  - Check the location of the SHA hash
- Therefore false signatures where the arbitrary bytes placed at the end of the sequence may be verified

Georgia Tech

How Bleichenbacher Works

The Bank of GTInfosec uses e = 3 and all sender public keys are 2048 bits long

1) We want to forge a RSA signature that s^3 = malformed message
2) We can do so by creating a malformed message that is 2048 bits long, and then finding the cube root of it, s = (malformed message)^{⅓}
3) When the Bank of GTInfosec goes to verify the signature s^3, it uses a bad signature verification that only checks for the header, magic bytes, and hash digest and ignores all other bytes
4) Once it finds the necessary bytes, you will be allowed to initiate the transfer

*Note: Remember to take the ceiling and not the floor when calculating for the final cube root

Good Luck !

- Visit us during Office Hours or make a post on Ed Discussion if you have questions!
- OH Schedule is on the course website: https://gatech.fail