Cryptographic Hash Functions

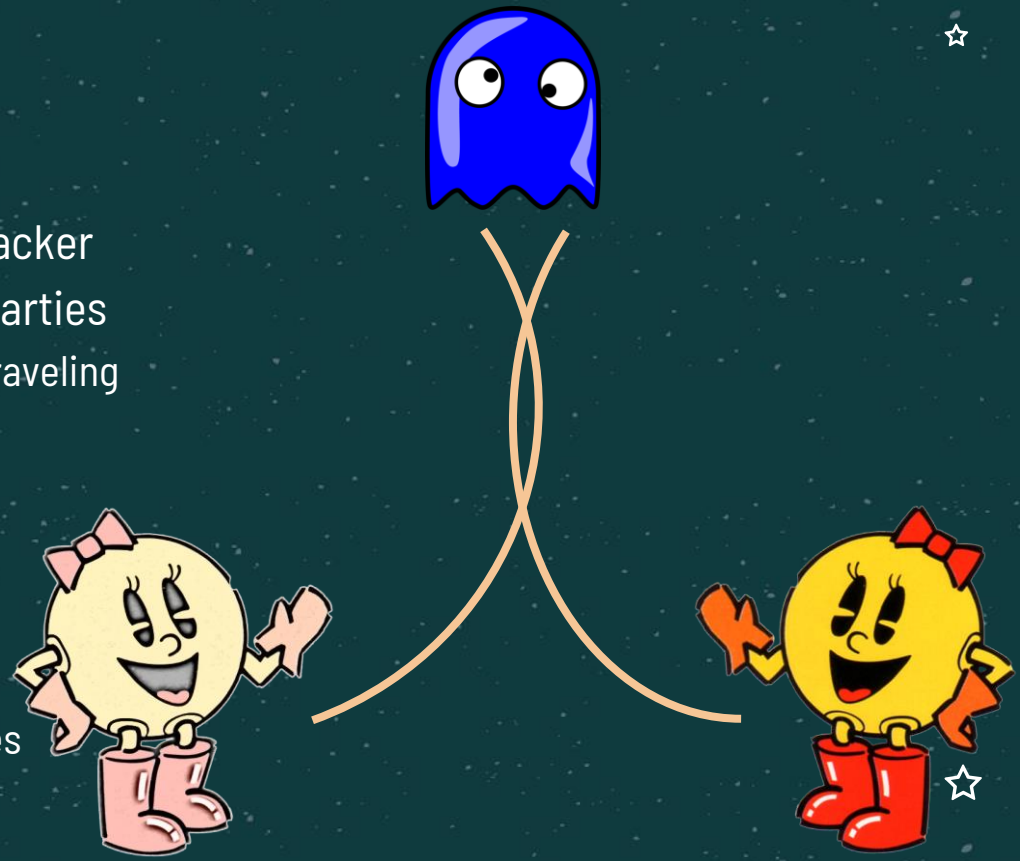
Information Security – Lecture 07

Aadil Zia Khan

Man-in-the-Middle

- A man-in-the-middle attack is when an attacker intercepts communications between two parties
  - Can secretly eavesdrop or modify traffic traveling between the two

- Attackers might use it for
  - Stealing login or personal information
  - Spy on the victim
  - Modify messages for serious consequences



Is Symmetric-key Encryption Enough?

- Symmetric-key encryption protects against attacks like eavesdropping
  - Ensures confidentiality
- Does it ensure authenticity/integrity - is the data unaltered and comes from the source?
  - Bit-flipping attack in RC4 – the message can be changed despite it being encrypted
- But what if we use strong cipher and protect the key?
  - ECB mode of encryption - if blocks of ciphertext are reordered, they will still decrypt but be reordered
- What about non-Repudiation?
  - No way of proving the source of message

Message Authentication

- Questions we need to ask to determine authenticity and integrity
  - Have the contents of the message/data been altered
  - Is the source authentic
  - Has the message been artificially delayed or replayed
  - Is the sequence relative to other messages flowing between two parties unchanged

Lets focus on the first now, and leave the second one for later

… others will be discussed much later

How Can Hash Functions Help Detect Data Modification

- A hash function is any function that can be used to map data of arbitrary size to fixed-size values
  - The values returned by a hash function are called hash values, hash codes, digests, or simply hashes
- Encryption requires a key – hashing can be done without a key
- A simple XOR hashing scheme
  - Determine how many bits would be used to represent the hash value – e.g., 8, 32, 64, 128
  - Split the data into blocks – each the size of the hash value
  - XOR the with the corresponding bits in the each blocks – i^{th} bit of each block will be XORed to get the i^{th} bit of the hash value

How Can Hash Functions Help Detect Data Modification

- Alice sends Bob data and a hash value of the data
- When Bob gets the message, he
  - Computes the hash value of the plaintext
  - Compares the calculated hash value with the one sent by Alice
- If Eve modifies the message, the hash values won't match
- If they match, assume that it is the message that Alice sent

How Can Hash Functions Help Detect Data Modification

- But wait … If Eve can modify the data, cant she modify the hash as well???
- Think about torrents
  - Suppose you download a file – how do you check its authenticity?
  - The hash value of that file is not sent by a peer – it is available on a secure website
    - Peer only sends the file or a block of it
    - Eve will have to attack both your traffic as well as the website to fool you
- It is also possible that the hash or data is encrypted – (more on this later)
  - But remember, sometimes you want to avoid encryption (maybe because it is time consuming)

XOR Hashing Scheme - Secure or Not?

- Suppose the hash value is only 8 bits long
- There will be only 256 possible values
- Several messages would have the same hash value
  - Sending one such message would fool Bob

- A better example
  - Suppose a computer does not save passwords, but instead saves a file consisting of each password's hash value
  - When you enter a password, its hash is compared with the already stored values – a match grants access
  - If there are only 256 possible hash values, the attacker can try different password combinations such that all these hash values are covered

Cryptographic Hash Functions

- Hash functions in infosec usually refer to cryptographic (or secure) hash functions
- If a hash function is secure
  - Easy to compute in one direction
  - Very difficult to compute in the other direction
  - Very difficult to find two messages that hash to the same value

Cryptographic Hash Functions

- What conditions should a hash function meet to be labeled secure?
  - Collision resistance
    - Difficult to find two messages such that both have the same hash value
  - Preimage resistance
    - Given a hash value, it is difficult to find a message that has that value
  - Second preimage resistance
    - Given a message, it is difficult to find another message with the same hash value

Cryptographic Hash Functions

- Some other properties in an ideal cryptographic hash function
  - It is deterministic
    - The same message always results in the same hash
  - It is quick to compute the hash value for any given message
  - Avalanche Effect
    - A small change to a message should change the hash value so extensively that a new hash value appears uncorrelated with the old hash value

Cryptographic Hash Functions - Uses

- Software integrity / Data integrity
- Intrusion detection / Malware detection
- Timestamping
- Message authentication
- One-time passwords
- Password verification
- Digital signature
- Proof-of-work (used in bitcoin mining)

Secure Hash Algorithm (SHA-256)

- SHA-2 (Secure Hash Algorithm 2) is a set of cryptographic hash functions designed by the United States National Security Agency and first published in 2001
  - SHA-2 family consists of six hash functions with hash values that are 224, 256, 384 or 512 bits
    - SHA-224, SHA-256, SHA-384, SHA-512
    - SHA-512/224, SHA-512/256 – the final result is truncated
- SHA-256 is used in
  - Security applications and protocols, including TLS and SSL, PGP, SSH, S/MIME, and Ipsec
  - Authenticating Debian software packages
  - Cryptocurrencies like Bitcoin for verifying transactions and calculating proof of work

SHA-256 - Preprocessing

- Begin with the original message of length L bits
- Append a single '1' bit at the least significant end
- Append K '0' bits at the least significant end, such that L + 1 + K + 64 is a multiple of 512
- Represent L using 64 bits and append at the least significant end

Total post-processed length will be a multiple of 512 bits

SHA-256 - Initialize Hash Values

- Now we create 8 hash values (h)
  - Initially these are hard-coded constants that represent the first 32 bits of the fractional parts of the square roots of the first 8 primes: 2, 3, 5, 7, 11, 13, 17, 19
  - Together these will form the final hash value

h0 := 0x6a09e667
h1 := 0xbb67ae85
h2 := 0x3c6ef372
h3 := 0xa54ff53a
h4 := 0x510e527f
h5 := 0x9b05688c
h6 := 0x1f83d9ab
h7 := 0x5be0cd19

SHA-256 - Initialize Round Constants

- Create 64 constants (k)
  - These are hard-coded constants that represent the first 32 bits of the fractional parts of the cube roots of the first 64 primes

0x428a2f98	0x71374491	0xb5c0fbcf	0xe9b5dba5	0x3956c25b	0x59f111f1	0x923f82a4	0xab1c5ed5
0xd807aa98	0x12835b01	0x243185be	0x550c7dc3	0x72be5d74	0x80deb1fe	0x9bdc06a7	0xc19bf174
0xe49b69c1	0xefbe4786	0x0fc19dc6	0x240ca1cc	0x2de92c6f	0x4a7484aa	0x5cb0a9dc	0x76f988da
0x983e5152	0xa831c66d	0xb00327c8	0xbf597fc7	0xc6e00bf3	0xd5a79147	0x06ca6351	0x14292967
0x27b70a85	0x2e1b2138	0x4d2c6dfc	0x53380d13	0x650a7354	0x766a0abb	0x81c2c92e	0x92722c85
0xa2bfe8a1	0xa81a664b	0xc24b8b70	0xc76c51a3	0xd192e819	0xd6990624	0xf40e3585	0x106aa070
0x19a4c116	0x1e376c08	0x2748774c	0x34b0bcb5	0x391c0cb3	0x4ed8aa4a	0x5b9cca4f	0x682e6ff3
0x748f82ee	0x78a5636f	0x84c87814	0x8cc70208	0x90befffa	0xa4506ceb	0xbef9a3f7	0xc67178f2

SHA-256 - What's with these Weird Numbers

- Cryptographic algorithms often need randomized constants for mixing or initialization purposes
- nothing-up-my-sleeve numbers are any numbers which, by their construction, are above suspicion of hidden properties
    - People would not feel that these numbers are there so that the government can crack the encryption
- Other possible nothing-up-my-sleeve number could be pi

SHA-256 - Blocks

- Split the bits that we obtained at the end of the preprocessing stage into 512 bit chunks

SHA-256 - Main Loop (for each chunk)

- Create an array (w) of 64 cells where each cell size is 32 bits - set all cells to zero
- Copy the 512 bit chunk into the first 16 cells
- For the remaining 48 cells - *index 16 to 63*
  - s0 = (w[i-15] rightrotate 7) xor (w[i-15] rightrotate 18) xor (w[i-15] rightshift 3)
  - s1 = (w[i- 2] rightrotate 17) xor (w[i- 2] rightrotate 19) xor (w[i- 2] rightshift 10)
  - w[i] = w[i-16] + s0 + w[i-7] + s1
- Initialize variables: a := h0 b := h1 c := h2 d := h3 e := h4 f := h5 g := h6 h := h7

All addition is modulo 2^{32}

SHA-256 - Main Loop (for each chunk)

- For all array (w) cells - *index 0 to 63*
  - S1 := (e rightrotate 6) xor (e rightrotate 11) xor (e rightrotate 25)
  - ch := (e and f) xor ((not e) and g)
  - temp1 := h + S1 + ch + k[i] + w[i]
  - S0 := (a rightrotate 2) xor (a rightrotate 13) xor (a rightrotate 22)
  - maj := (a and b) xor (a and c) xor (b and c)
  - temp2 := S0 + maj
  - h := g g := f f := e e := d + temp1
  - d := c c := b b := a a := temp1 + temp2

All addition is modulo 2^{32}

SHA-256 - Main Loop (for each chunk)

- Finally recalculate the 8 hash values
  - h0 := h0 + a
  - h1 := h1 + b
  - h2 := h2 + c
  - h3 := h3 + d
  - h4 := h4 + e
  - h5 := h5 + f
  - h6 := h6 + g
  - h7 := h7 + h

All addition is modulo 2^{32}

- You can now move on to the next chunk and repeat the process

SHA-256 - Final Hash Value

- Once all the chunks have been iterated over you can now calculate the final hash value
- hash := h0 append h1 append h2 append h3 append h4 append h5 append h6 append h7