Announcements

1. HW 3 is due next lecture.

2. HW 4 is online, due before class in 1.5 weeks.
Last time: Hash functions

This time: Hash-based MACs, authenticated encryption
Constructing a MAC from a hash function

Recall:

- Collision-resistant hash function: Unkeyed function \( H : \{0, 1\}^* \rightarrow \{0, 1\}^n \) hard to find inputs mapping to same output.

- MAC: Keyed function \( \text{Mac}_k(m) = t \), hard for adversary to construct valid \((m, t)\) pair.
Candidate MAC constructions

- \( \text{Mac}(k, m) = H(k||m) \)

- \( \text{Mac}(k, m) = H(m||k) \)

- \( \text{Mac}(k, m) = H(k||m||k) \)

- \( \text{Mac}(k_1, k_2, m) = H(k_2||H(k_1||m)) \)
Candidate MAC constructions

- **Mac**\((k, m) = H(k\|m)\)

  **Insecure.** Vulnerable to length extension attacks for Merkle-Damgård functions. Secure for SHA3 sponge.

- **Mac**\((k, m) = H(m\|k)\)

- **Mac**\((k, m) = H(k\|m\|k)\)

- **Mac**\((k_1, k_2, m) = H(k_2\|H(k_1\|m))\)
Candidate MAC constructions

- $\text{Mac}(k, m) = H(k||m)$

  **Insecure.** Vulnerable to length extension attacks for Merkle-Damgård functions. Secure for SHA3 sponge.

- $\text{Mac}(k, m) = H(m||k)$

  Ok, but vulnerable to offline collision-finding attacks against $H$.

- $\text{Mac}(k, m) = H(k||m||k)$

- $\text{Mac}(k_1, k_2, m) = H(k_2||H(k_1||m))$
Candidate MAC constructions

- $\text{Mac}(k, m) = H(k \| m)$

  **Insecure.** Vulnerable to length extension attacks for Merkle-Damgård functions. Secure for SHA3 sponge.

- $\text{Mac}(k, m) = H(m \| k)$

  Ok, but vulnerable to offline collision-finding attacks against $H$.

- $\text{Mac}(k, m) = H(k \| m \| k)$

  Ok, but nobody uses.

- $\text{Mac}(k_1, k_2, m) = H(k_2 \| H(k_1 \| m))$
Candidate MAC constructions

- **Mac**$(k, m) = H(k||m)$
  
  **Insecure.** Vulnerable to length extension attacks for Merkle-Damgård functions. Secure for SHA3 sponge.

- **Mac**$(k, m) = H(m||k)$
  
  Ok, but vulnerable to offline collision-finding attacks against $H$.

- **Mac**$(k, m) = H(k||m||k)$
  
  Ok, but nobody uses.

- **Mac**$(k_1, k_2, m) = H(k_2||H(k_1||m))$
  
  Secure, similar to HMAC.
Length extension attacks

Recall the Merkle-Damgård construction:

\[
\begin{array}{c}
\hat{m}_k = m_k || \text{pad} || \text{len}(m) \\
\text{The final output is equivalent to an intermediate state for} \\
H(m||\text{pad}||\ldots).
\end{array}
\]
Length extension attacks

**Input:** Bad MAC: \((m, H(k||m))\)

**Attack:** Forge valid bad MAC: \((m||pad||m', H(k||m||pad||m'))\)

In general, we can construct the hash \(H(m||pad||m_{new})\) for any \(m_{new}\) from only \(H(m)\) even if we don’t know \(m\).

Just need to know (or guess) \(\text{len}(m)\) to compute padding.
HMAC: A PRF for Merkle-Damgård functions

\[ F_k(m) = H(k \oplus \text{opad} || H(k \oplus \text{ipad} || m)) \]

\[ \text{ipad} = 0x36 \quad \text{opad} = 0x5C \]

Under the heuristic assumption that \( k \oplus \text{opad} \) and \( k \oplus \text{ipad} \) are “independent” keys, this is a secure PRF.

HMAC is standardized and HMAC-SHA256 is a good choice. Historically HMAC-SHA1 was also common.

\[ H(k || m) \] is a secure MAC for SHA3.
Key derivation

**Problem:** How do we get symmetric keys?

**Input:** Some data that we want to use to generate a key.
- A password
- A bunch of nonuniform random inputs from the environment
- The result of a public-key agreement (coming soon!)

**Desired output:** Uniform AES or MAC keys of the right length.

**Solutions that work in practice:**
- $H(data)$
- $HMAC_0(data)$ (better for Merkle-Damgård functions)
Subkey derivation

For a real protocol, we likely need several keys: encryption keys for each direction, MAC keys.

Once we have derived a master key $mk$ using a hash function, we can use a PRF to derive subkeys.

Examples:

- $k_{mac} = F_{mk}("MAC-KEY")$
- $k_{AB} = F_{mk}("AB-KEY")$ for Alice $\rightarrow$ Bob encryption
- $k_{BA} = F_{mk}("BA-KEY")$ for Bob $\rightarrow$ Alice encryption

If $F$ is a secure PRF, then these behave like independent keys.

HMAC is often used for this in practice.
HKDF

Standardized HMAC-based key derivation function.

**Input:** Secret $s$, optional salt $salt$

**Output:** $L$ bytes of output

**Algorithm:**
Use a HMAC function with output length $\ell$.

1. $t = \text{HMAC}_{salt}(s)$
2. $z_0 = \text{empty string}$.
3. for $i$ from 1 to $\lceil L/\ell \rceil$:
   
   $z_i = \text{HMAC}_t(z_{i-1} \| i)$
4. Output $L$ bytes of $z_1 \| \ldots$
Chosen ciphertext attacks

\[ A \xrightarrow{\text{mo, m_i}} C \]

\[ C \leftarrow \text{Enc}_x(m_i) \]

Oracle access to \( \text{Enc}_x(\cdot), \text{Dec}_x(\cdot) \)

\[ \xrightarrow{\text{b}'} A \text{ succeeds, if } b = b' \]

Definition

(Enc, Dec) is CCA-secure if \( \forall \) efficient adversaries \( A \),

\[ \Pr[A \text{ succeeds}] \leq 1/2 + \epsilon \]

IND-CCA1: Non-adaptive: Decryption oracle only queried prior to \( c \)

IND-CCA2: Adaptive: May make further calls to decryption oracle
Ciphertext Integrity

A wins if $c$ is a valid ciphertext and not queried.

**Definition**

(Enc, Dec) provides ciphertext integrity if $\Pr[A \text{ succeeds}] = \text{negligible}$. 
Authenticated Encryption

**Definition**

(Enc, Dec) provides authenticated encryption if it is CPA-secure and provides ciphertext integrity.

**Theorem**

If (Enc, Dec) provides authenticated encryption then it is CCA-secure.
Constructing Authenticated Encryption

Encrypt-then-MAC

- Encryption: \( c = \text{Enc}_{k_e}(m) \quad t = \text{Mac}_{k_m}(c) \) output \((c, t)\)
- Decryption: Input \((c, t)\).
  
  If \(\text{Verify}_{k_m}(c, t) = \text{reject} \) then output reject
  else output \(\text{Dec}_{k_e}(c)\).

Theorem

*Encrypt-then-MAC is CCA secure.*

Common implementation mistakes:

- Using the same key for encryption and MAC
- Only MACing part of the ciphertext. (e.g. omitting the IV or the data used to derive a deterministic IV)
- Outputting some plaintext before verifying integrity
MAC then Encrypt is not CCA secure

**MAC-then-encrypt**

- **Encryption:** $t = \text{Mac}_{km}(m)$, $c = \text{Enc}_{ke}(m||t)$, output $c$
- **Decryption:** Input $c$. Compute $\text{Dec}_{ke}(c) = (m||t)$
  - If $\text{Verify}_{km}(m, t) = \text{reject}$ then output reject
  - else output $m$.

MAC-then-encrypt can fail to be secure even with CPA-secure Enc and secure MAC.

SSL 3.0 vulnerable to POODLE attack.
POODLE Attack Setup

Victim is a web browser.

Victim visits evil.com.

evil.com contains Javascript causing victim to make cookie-bearing request to bank.com.

Man-in-the-middle attacker intercepts encrypted traffic between victim and bank.com and modifies ciphertext, using bank.com as a decryption oracle.
POODLE Attack Idea

SSL 3.0 uses MAC-then-encrypt with CBC mode.

c = Enc(message || MAC tag || pad)

To pad $p$ bytes, append $p - 1$ arbitrary bytes and then byte $p - 1$. (For 0 bytes, append dummy block of 15 bytes ending in 15.)

If adversary intercepts block

If last byte is 15, decryption valid, otherwise likely reject

⇒ learn byte of $m$. (Same logic as your homework.)
Authenticated encryption in practice

Fine solution: Use AES-GCM mode.

- TLS 1.3 uses authenticated encryption modes correctly.
- Older versions of TLS use MAC-then-encrypt.
- SSHv2 uses Encrypt-and-MAC. This is not generally secure but is secure for SSH’s cipher choices.
1. HW 3 due next lecture!

2. HW 4 is online!