CSE 207B: Applied Cryptography

Nadia Heninger

UCSD

Fall 2023 Lecture 7

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 $H : \{0,1\}^* \to \{0,1\}^n$ hard to find inputs mapping to same output.

- MAC: Keyed function $\mathsf{Mac}_k(m) = t$, 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

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

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

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

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

- $\mathrm{Mac}(k_1, k_2, m) = H(k_2||H(k_1||m))$

Candidate MAC constructions

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

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

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

  Ok, but vulnerable to offline collision-finding attacks against H for Merkle-Damgård functions.

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

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

Candidate MAC constructions

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

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

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

  Ok, but vulnerable to offline collision-finding attacks against H for Merkle-Damgård functions.

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

  Ok, but nobody uses.

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

Candidate MAC constructions

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

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

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

  Ok, but vulnerable to offline collision-finding attacks against H for Merkle-Damgård functions.

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

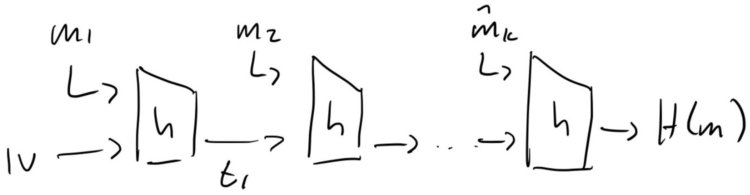
  Ok, but nobody uses.

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

  Secure for Merkle-Damgård functions, similar to HMAC.

Length extension attacks

Recall the Merkle-Damgård construction:



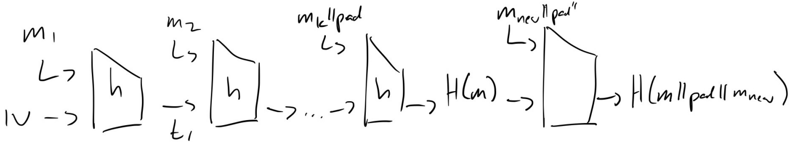
$\hat{m}_k = m_k || \text{pad} || \text{len}(m)$

The final output is equivalent to an intermediate state for $H(m||pad||...)$.

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) len(m) to compute padding.

HMAC: A PRF for Merkle-Damgård functions

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

$$\text{ipad} = 0x\overline{36} \qquad \text{opad} = 0x\overline{5C}$$

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 to derive subkeys.

Examples:

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

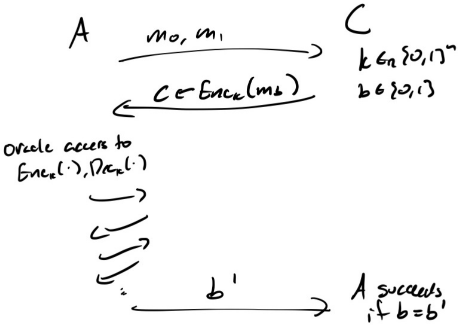
Input: Secret s, optional salt $salt$

Output: L bytes of output

Algorithm:
Use a HMAC function with output length ℓ.

1. $t = HMAC_{salt}(s)$
2. $z_0 =$ empty string.
3. for i from 1 to $\lceil L/\ell \rceil$:
   $$z_i = HMAC_t(z_{i-1} || i)$$
4. Output L bytes of $z_1 || \dots$

Chosen ciphertext attacks



Definition

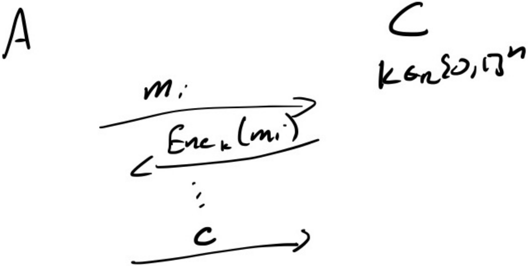
(Enc, Dec) is CCA-secure if $\forall$ efficient adversaries A,

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

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

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

Authenticated Encryption

Definition
$(\mathsf{Enc}, \mathsf{Dec})$ provides authenticated encryption if it is CPA-secure and provides ciphertext integrity.

Theorem
If $(\mathsf{Enc}, \mathsf{Dec})$ *provides authenticated encryption then it is CCA-secure.*

Constructing Authenticated Encryption

Encrypt-then-MAC

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

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: $t = \mathsf{Mac}_{k_m}(m) \quad c = \mathsf{Enc}_{k_e}(m||t) \quad$ output c
- Decryption: Input c. Compute $\mathsf{Dec}_{k_e}(c) = (m||t)$
  If $\mathsf{Verify}_{k_m}(m, t) = \texttt{reject}$ 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

$$c \quad = \quad \underbrace{\boxed{c[0]}}_{\text{IV}}\underbrace{\boxed{c[1]}\ \cdots}_{\text{encryption of m}}\underbrace{\boxed{c[\ell-1]}}_{\text{encrypted tag}}\underbrace{\boxed{c[\ell]}}_{\text{encrypted pad}}$$

Then they query decryption oracle with

$$\hat{c} \quad := \quad \boxed{c[0]}\boxed{c[1]}\ \cdots\ \boxed{c[\ell-1]}\underbrace{\boxed{c[1]}}_{\text{encrypted pad?}}$$

If last byte is 15, decryption valid, otherwise likely reject
$\implies$ 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!