EECS 388

Introduction to Computer Security

Lecture 2:

Message Integrity

August 29, 2024
Prof. Halderman

Cryptography

Cryptography
(From Greek: *kryptós+graphein*, “secret writing”)
The study of techniques for communicating securely in the presence of an adversary

***Related*: Cryptanalysis**
the study of techniques for *breaking* cryptosystems

Goals for 388: Learn how to safely use **crypto primitives** as building blocks for security

Security properties we’ll try to achieve:

- **Confidentiality**
- **Message Integrity**
- **Sender Authenticity**

Cryptographic Theory

Beautiful, highly rigorous

Proofs based on computational complexity

Cryptographic Practice

Delicate engineering

Assumptions based on empirical experience

This course gives only a brief (**and non-rigorous**) overview. Consider EECS 475 or 575 to dive deeper into cryptography!

Goal: Message Integrity

Message integrity ensures that attackers cannot modify messages *without being detected* (they can still do plenty of other bad stuff!)

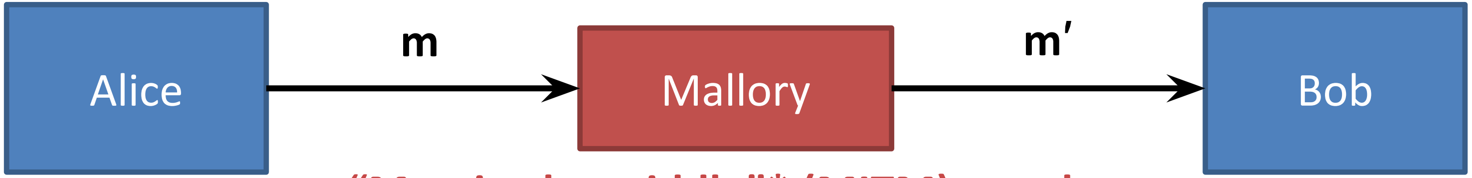
Message integrity is often *even more important* than confidentiality [Why?]

Alice wants to send message ***m*** to **Bob**

- They don't fully trust the messenger (or network) carrying the message
- They want to be sure what Bob receives is what Alice actually sent (**m′** = **m**)

Threat model:

- Mallory can see/modify/forge messages
- Mallory wants to trick Bob into accepting a message Alice didn't send



"Man-in-the-middle"* (MITM) attacker

a kind of **active attacker**

* Now sometimes called a "meddler", "manipulator", "machine", "person", or "adversary"-in-the-middle, but the original term remains the most common.

Approach: Message Verifier

1. Alice computes *verifier* **v** := ***f***(**m**)

2. 

e.g. “Attack at dawn”, 628369867…

3. Bob verifies that **v′** = ***f***(**m′**), accepts message if and only if this is true

Properties we want for ***f***() ?

Easily computable by Alice and Bob, but ***not*** easily computable by Mallory

We lose the game if Mallory can deduce ***f***(**x**) for any **x** ≠ **m**

Idea: Secret ***f***() only Alice and Bob know

Candidate ***f***(): **Random function (RF)**

Input: Fixed size (length of longest **m**)

Output: Fixed size (say, 256 bits)

Construct a giant lookup table by flipping coins for every possible input

0	→	0011111001010…
1	→	1110011010001…
2	→	0101010001010…
	…	

Pro: ***Provably secure***

[Show Mallory can’t do better than guessing]

Con: ***Completely impractical!***

[Estimate how much storage it would require]

Pseudorandom Functions

Want a function that's practical but "looks random"...

Pseudorandom function (PRF)

Let's build a PRF:

Start with a family of 2^n functions
$f_0(), f_1(), f_2(), \ldots, f_{(2^n)-1}()$
all ***known to Mallory.***

Let our verification function $v() := f_k()$
where **k** is a secret **n**-bit index ("**key**")
known only to Alice and Bob.

What makes for a suitable function family $f()$?

Security definition: A game against Mallory—

1. Choose a secret **k** and a random function $g()$
2. We flip a coin secretly to get bit **b**
3. If **b**=0, let $h() := g()$
   If **b**=1, let $h() := f_k()$
4. Mallory chooses **x**; we announce $h(\mathbf{x})$.
   Repeat step 4 as often as Mallory likes
5. Mallory guesses **b** *in polynomial time**

We say $f()$ is a **secure PRF** if Mallory can't do meaningfully better than random guessing.

*** Note the reliance on computational complexity.**
Mallory can always win slowly!
With RF, Mallory **can't possibly** learn unseen outputs.
With PRF, M. can, but at *impractical* (exponential) cost by mounting a "**brute force attack**" on **k**. [Explain?]

Using a PRF for Message Integrity

1. Let ***f***() by a secure PRF (known to everyone)
2. In advance, choose a random key **k** known to Alice and Bob but <u>not</u> Mallory
3. Alice computes **v** := $\boldsymbol{f}_{\mathbf{k}}$(**m**)
4. 

5. Bob verifies that **v′** = $\boldsymbol{f}_{\mathbf{k}}$(**m′**), accepts message if and only if this is true

If Bob accepts **m′**, then, with very high confidence, **m′** is identical to **m**. (How high? $1-1/2^n$)
[Important assumptions?]

What if Alice and Bob want to send more than one message? [Attacks?] [Solutions?]

This approach follows **Kerckhoffs's Principle**:
"A cryptosystem should remain secure even if attackers know ***everything but the key.***" [Why?]

Annoying question:
Do PRFs *actually exist*?

Annoying answer:
We don't know. (Would imply P≠NP!)

Best we can do:
Use well studied functions where we haven't spotted a problem yet

Cryptographic Hashes

First step towards a practical solution…

Cryptographic Hash Function

Fixed function ***H***(). ***No key***!

Input: arbitrary length data

Output: fixed size *digest* (**n** bits)

Properties of strong hash functions:

Preimage resistance

Given output **h**, hard to find any input **m** s.t. **h** = ***H***(**m**)

Collision resistance

Hard to find *any* pair of inputs $\mathbf{m_1}$,$\mathbf{m_2}$ s.t. $H(\mathbf{m_1})=H(\mathbf{m_2})$

Note: Collisions exist [why?], but should be hard to find

Second-preimage resistance

Given $\mathbf{m_1}$, hard to find different $\mathbf{m_2}$ s.t. $H(\mathbf{m_1}) = H(\mathbf{m_2})$

Computing hashes with **OpenSSL:**

```
$ echo "hello world 0" | openssl dgst -sha256
8bfa9a398e97152beaaf385847808ad2d828c1c7251f1a45
bc7697723827e7e7
$ echo "hello world 1" | openssl dgst -sha256
07bafbe63a0e7c57c572aedf1c228022b537e28785013d7b
e017fc78731a8cc5
```

Observe how changing even a single bit of the input produces output that appears completely unrelated.

Annoying question:

Are existing hashes *actually strong*?

Annoying answer:

We don't know.

Candidates: ~~MD5~~, ~~SHA-1~~, **SHA-256**, SHA-512, SHA-3

Hash Function Failures

MD5

Once ubiquitous … broken in 2004

Now it's easy to find collisions
(pairs of messages with the same MD5 hash)

Exploited to attack real systems

You'll do this in Project 1!

```
$ echo
d131dd02c5e6eec4693d9a0698aff95c2fcab58712467eab4004583eb8fb7f89
55ad340609f4b30283e488832571415a085125e8f7cdc99fd91dbdf280373c5b
d8823e3156348f5bae6dacd436c919c6dd53e2b487da03fd02396306d248cda0
e99f33420f577ee8ce54b67080a80d1ec69821bcb6a8839396f9652b6ff72a70
| xxd -r -p  | openssl dgst -md5
79054025255fb1a26e4bc422aef54eb4

$ echo
d131dd02c5e6eec4693d9a0698aff95c2fcab50712467eab4004583eb8fb7f89
55ad340609f4b30283e4888325f1415a085125e8f7cdc99fd91dbd7280373c5b
d8823e3156348f5bae6dacd436c919c6dd53e23487da03fd02396306d248cda0
e99f33420f577ee8ce54b67080280d1ec69821bcb6a8839396f965ab6ff72a70
| xxd -r -p  | openssl dgst -md5
79054025255fb1a26e4bc422aef54eb4
```

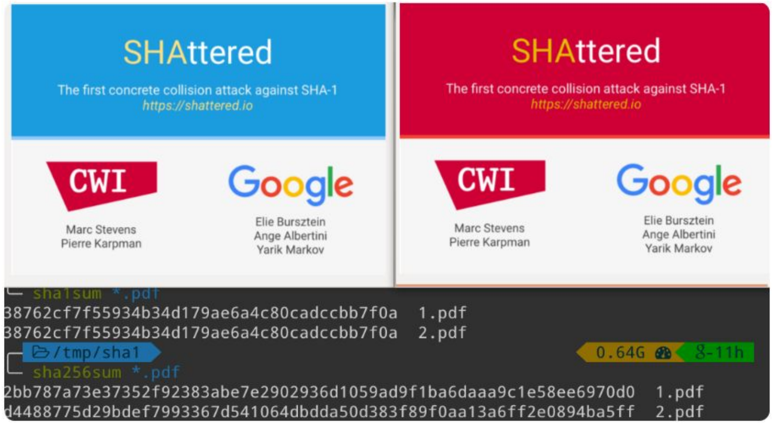
SHA-1

Once ubiquitous … broken in 2017

Rapidly being phased out

Computing first collision cost >$100,000
Today, with improvements <$10,000

Expect attacks will keep getting better!



Constructing SHA-256

SHA-256 is a widely used hash function that is currently thought to be strong

Input: arbitrary length data
Output: 256-bit digest

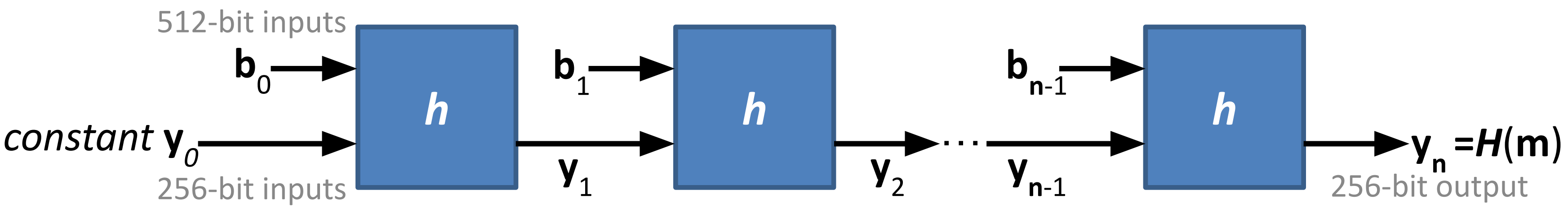
Built from a compression function ***h***:

Inputs: (256 bits, 512 bits), **Output:** 256 bits

reduces 768 bits to 256 bits using a complicated internal function (details out of scope for 388)

Uses the Merkle–Damgård (MD) construction (illustrated below) to accept arbitrary-length input by repeatedly applying ***h***():

1. Pad input **m** to the next multiple of 512 bits (adds <u>at least</u> 1 bit, uses fixed algorithm [why?]) and split into 512-bit blocks: $\mathbf{b}_0, \mathbf{b}_1, \dots \mathbf{b}_{n-1}$
2. $\mathbf{y}_0$:= <*256-bit constant*>
   $\mathbf{y}_1 := h(\mathbf{y}_0, \mathbf{b}_0) \quad \dots \quad \mathbf{y}_i := h(\mathbf{y}_{i-1}, \mathbf{b}_{i-1})$
3. Return $\mathbf{y}_n$ which is defined to be **SHA-256**(m)



MD Hash Pitfall: Length Extension Attacks

Merkle–Damgård hash functions are susceptible to **length extension attacks**:

Given **y** = ***H***(**x**) for some <u>unknown</u> **x**, attackers can calculate

z = ***H***(**x** // *padding* // **s**)

concatenation

for arbitrary **s**.

That is, given:

y = **SHA-256**(???)

An attacker can produce:

z = **SHA-256**(??? *original pad* suffix)

Note that this doesn't violate preimage, second-preimage, or collision resistance.

[But why is it a problem?]

Suppose Alice and Bob use this as a verifier:

v := **SHA-256**(**k** // **m**)

k Alice → **m**, **v** → Mallory → **m′**, **v′** → Bob **k**

1. Alice sends **m** = "Please go to the bank."
2. Mallory doesn't know **k**, but *can* apply length extension to (**m**, **v**) to calculate **v′** for:

   m′ = "Please go to the bank.[*original_pad*]
   Then transfer $10,000 to Mallory."

   (*original_pad* is some bytes beyond Mallory's control, but which the recipient might ignore)
3. Since **v′** is the correct verifier for **m′**,
   Bob will accept the modified message as valid

You'll explore how this is done in Project 1!

Practical Solution: HMAC

Message Authentication Code (MAC)

Designed to be used as a secure verifier:

Inputs: key, arbitrary length data
Output: fixed size digest (**n** bits)

HMAC construction turns any secure hash function *H*() into a MAC:

$$\mathrm{HMAC}_k(m) = H(k \oplus c_1 \,\|\, H(k \oplus c_2 \,\|\, m))$$

XOR

constant 363636…

constant 5c5c5c…

concatenation

Design protects against length extension!

Example: HMAC-SHA-256 is an HMAC constructed using **SHA-256** for *H*()

For practical purposes, we (think/hope) we can treat **HMAC-SHA-256** as a PRF

Can *reduce* PRF security of HMAC-SHA-256 to a (weaker) security property of SHA-256's compression function

```
$ echo "hi" | openssl dgst -sha256 -hmac Secr3t
8074cdfd007e5cfdc71c2c1cd393a5fefa890d7702956a13
66a155d79d1cbe77
```

At last, this gives Alice and Bob a suitable approach for protecting message integrity:

v := HMAC-SHA-256$_k$(m)

Computes and sends
v := HMAC-SHA-256$_k$(m)

Rejects message if
$v' \neq$ HMAC-SHA-256$_k$(m')

Coming Up

Reminders:

Selfies were due today

Quiz on Canvas after every lecture

Lab Assignment 1 available today, due next Thursday at 6pm

Project 1 available today; Part 1 due Sept. 12 at 6pm

Tuesday

Randomness and Pseudorandomness

Generating randomness,
PRGs, one-time pads

Thursday

Confidentiality

Simple ciphers,
AES,
block cipher modes