Message Integrity HMAC-SHA-256

1. Alice computes verifier $V := f(m)$
2. Alice $\xrightarrow{m,V}$ Mallory $\xrightarrow{m',V'}$ Bob.
3. Bob verifies $V' = f(m')$

pseudorandom functions (PRF)

Start with 2^n functions: $f_0(\,) \cdots f_{2^n-1}(\,)$

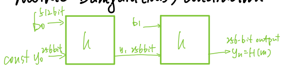
Verify with k, only know to Alice and Bob, $f_k(\,)$

Security definition:

1. Choose a secret k and random function $g(\,)$.
2. Flip a coin → b=0, $h(\,) := g(\,)$; b=1, $h(\,) := f_k(\,)$
3. Mallory choose x, we announce h(x), Mallory guess b

Hash Failures: MD5. SHA-1

Merkle-Damgard (MD) construction



Length-Extension Attack

m: 'original message' + '[original padding]'

m': 'original message' + 'original padding' + 'modified message' + '[padding]'

Solution: HMAC

$HMAC_k(m) = H(k \oplus c_1 \| H(k \oplus c_2 \| m))$ (constant)

Randomness and Pseudorandomness

Pseudorandom Generator (PRG)

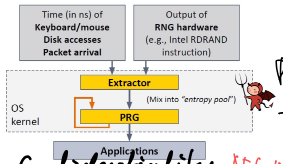
Getting Randomness

Randomness is **an input** to your program

Typically provided by the OS, via special APIs

OS continuously gathers inputs from physical sources that are hard for adversary to predict, "extracts" uniform bits from them.

[What if attacker can predict *some* of them?]



Confidentiality AES-128 in CTR mode

$C_i := P_i \oplus K_i$

Onetime pad: $P_i := C_i \oplus K_i$ *No reuse

Use PRG: $S_i := g(l)$, $C_i := P_i \oplus S_i$

$S_i := g(l)$, $P_i := C_i \oplus S_i$ *No reuse k, output

AES Advanced Encryption Standard

Padding: M MMMM 030303, a block of 0s

Cipher-block chaining (CBC) mode

"Chains" ciphertexts to obscure later ones

Choose a **random initialization vector IV**

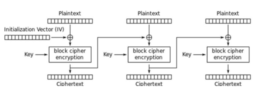
Encrypt: $c_0 := IV$; $c_i := E_k(p_i \oplus c_{i-1})$

Decrypt: $p_i := D_k(c_i) \oplus c_{i-1}$

[Why do we need the IV?]

Have to send IV with ciphertext

Can't encrypt blocks in parallel or out of order



Counter (CTR) mode

Turns a block cipher into a stream cipher

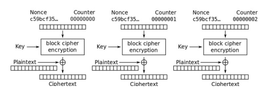
Generate **keystream** s for k and **unique nonce**:

$s := E_k(\text{nonce}\|0) \| E_k(\text{nonce}\|1) \| E_k(\text{nonce}\|2) \| \ldots$

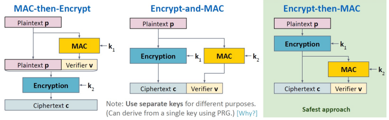
Encrypt: $c := p \oplus s$ Decrypt: $p := c \oplus s$

Benefits: Doesn't require padding. Efficient parallelism/random access

Caution: Never reuse nonce for same k!



MAC-then-Encrypt / Encrypt-and-MAC / Encrypt-then-MAC



Note: Use separate keys for different purposes. (Can derive from a single key using PRG.) [Why?]

[Which approach is safest?] Our encryption methods (so far) only secure against passive eavesdroppers. Only EtM can ensure ciphertext isn't tampered with before decryption.

"Cryptographic Doom Principle": If you perform any cryptographic operations on a message you've received before verifying the MAC, it will somehow inevitably lead to doom

Common flaw when using MAC-then-Encrypt

Suppose an implementation uses **CBC mode**. Decryption involves the following steps:

1. $m\|v\|pad := D_k(c)$
2. Check that **pad** is valid PKCS7, else raise PadError
3. Check that $v = MAC_k(m)$, else raise MacError

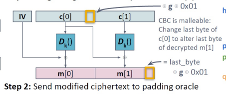
This is how TLS 1.0 worked. Seems reasonable?

Any method to distinguish these two error types (even tiny timing differences) **leaks the plaintext!**

Padding oracle: attacker submits any ciphertext and learns if last bytes of plaintext are a valid pad

Suppose attacker intercepts c, wants to learn m.

Step 1: Let g be a guess for last byte of block m[1]:



Step 2: Send modified ciphertext to padding oracle. If g = last_byte: ... padding's valid; oracle returns MacError; else: Padding is invalid*, oracle returns PadError

Step 3: Repeat with g = 0, 1, … 255 to learn last_byte. Then use a 0x02, 0x02 pad to learn next byte, etc.

Galois/Counter Mode (GCM)

Most widely used AEAD cipher mode

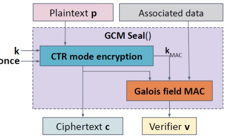
Developed by McGrew and Viega in 2004

Standardized by IETF, NIST, others

Non-generic composition [What kind?] of AES in CTR mode for encryption and a special MAC based on polynomials over finite ("Galois") fields.

Note: GCM violates principle of key separation [How?]

(We can prove it's ok, but it's delicate.)



Warning: Can construct GCM ciphertexts that decrypt (differently, but without error) under many keys

Prof. Grubbs crafted one that decrypts under 131,072 different keys

Preferred modern approach:

Warning: GCM nonce reuse is *catastrophic*! Encrypting two ciphertexts with the same (k,nonce) leaks **the plaintext and the MAC key.**

Why? Polynomial root-finding. Details interesting but beyond our scope in 388.

Authenticated encryption with associated data (AEAD)

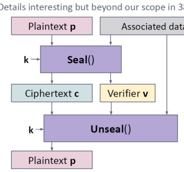
Integrity and encryption in a single primitive:

c, v := Seal(k, p, [associated_data]) encrypts plaintext **p** and returns ciphertext **c** and a verifier **v** (called a "tag")

p, err := Unseal(k, c, v, [associated_data]) returns **p** or an error if **v** does not match the supplied **c** and **associated_data**

Optional **associated_data** is covered by verifier but *not encrypted*.

Useful for binding data to its context: e.g., counter, sender ID, etc.



Examples:

AES-GCM ("Galois Counter Mode") hardware accelerated in recent CPUs

ChaCha20-Poly1305, common on mobile

The web platform

HTML: Static

DOM: Developer tool - elements dynamic

URLs (Uniform Resource Locators)

A **URL** is a string specifying a unique resource on the Web.

scheme://host:port/path?query#fragment

scheme: protocol used to access resource (https, http)

host: server's domain name or IP address

port: TCP port (default 443 for HTTPS and 80 for HTTP)

path: identifies resource to server

query: parameter string passed to server

fragment: parameter string visible only to client

Paths, Query, Fragments: % Encoding — Most punctuation and non-ASCII characters must be *escaped* by encoding each UTF-8 byte as % followed by two hex characters.

Hosts: IDN Encoding — The host field uses a different encoding for internationalized names called **IDN encoding.** Portions of the name are replaced by xn-- followed by a Punycode encoding.

GET: get the header + contents of url.

POST: send information (eg. web forms)

PUT: uploads file in entity body to path specified in URL field.

DELETE: deletes file specified in the URL field

- HTTP Response Message
  - Status line: protocol version, status code, status phrase
  - Response headers: provide information
  - Body: optional data

```
HTTP/1.1 200 OK
Connection close
Date: Thu, 06 Jan 2017 12:00:15 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Mon, 22 Jun 2006 ...
Content-Length: 6821
Content-Type: text/html
(blank line)
data data data data data ...
```

status line (protocol, status code, status phrase); header lines; data

- HTTP Request Message
  - Request line: method, resource, and protocol version

```
GET /somedir/page.htm HTTP/1.1
Host: www.someschool.edu
User-agent: Mozilla/4.0
Connection: close
Accept-language: fr
(blank line)
```

request line; header lines; carriage return line feed indicates end of message

1. **Allowed by SOP:**
   - A script on https://example.com/page1 can freely access and modify resources from https://example.com/page2 because both pages share the same origin (same protocol, domain, and port).
2. **Blocked by SOP:**
   - A script on https://example.com **cannot** access resources from https://anotherdomain.com because they have different domains.
   - A script on https://example.com **cannot** access resources from http://example.com because they have different protocols (https vs http).
   - A script on http://example.com:80 **cannot access resources from** http://example.com:8080 because they have different ports (80 vs 8080).

Diffie-Hellman key exchange

Alice $A: g^a \bmod p$ → Bob $c = g^{ab} \bmod p$; $B: g^b \bmod p$

can protect passive eavsdropping, can't protect MITM attack

RSA key exchange

1. get p, q 2. $N = pq$ 3. choose e, prime to $(p-1)(q-1)$
4. choose d. $ed \equiv 1 \bmod (p-1)(q-1)$

Public key (e,N) Private key (d,N)

S: $s = m^d \bmod N$; E: $c = m^e \bmod N$; V: $m \stackrel{?}{=} s^e \bmod N$

Public key encryption: $m = c^d \bmod N$ Digital signatures:

Textbook RSA: e=3, $c = m^e < N \Rightarrow m = c^{\frac{1}{3}}$

Image allowed: The site can load the image, but cannot access image's pixel data (colors, dimensions), unless Cross Origin Resource Sharing

Cookies

To prevent eavsdropper: Secure. To prevent DOM: HttpOnly

Setting a cookie: A site can set a cookie for its own domain or any *parent domain*, as long as the parent domain is not a **public suffix**.

Example: login.site.com attempts to set cookie

login.site.com	allowed
site.com	allowed
other.site.com	prohibited
different.com	prohibited
com	prohibited (public suffix)

Reading a cookie: A site can read cookies set for its own domain or **any parent domains.***

Example: login.site.com can read cookies for

login.site.com	yes
site.com	yes
other.site.com	no
different.com	no

Caution: Cookies also specify a *path* on the site, but (without HttpOnly) it's for efficiency only. DOM origins aren't isolated by path, so scripts can read cookies set for any path in the origin.

Suppose cookie set for x.com/b. Then x.com/a can do: alert(frames[0].document.cookie);

* If domain= attribute is unset, some browsers disallow

Caution: You don't know which domain set a cookie when you receive it.

Example: club.eecs.umich.edu can set cookies for umich.edu (since latter isn't a public suffix).

Cookie 1: name = mycookie, value = mycookievalue, domain = login.site.com, path = /

Cookie 2: name = cookie2, value = mycookievalue, domain = site.com, path = /

Cookie 3: name = cookie3, value = mycookievalue, domain = site.com, path = /my/home

	Cookie 1	Cookie 2	Cookie 3
checkout.site.com	No	Yes	No
login.site.com	Yes	Yes	No
login.site.com/my/home	Yes	Yes	Yes
site.com/my	No	Yes	No

CSRF

```
<html>
  <img src="//bank.com/transfer?to=attacker&amount=10000">
</html>
```

GET /transfer?to=attacker&amount=10000 — Cookie: AuthToken=39e839f928ab79 → bank.com

```
<form name=f method=post action="//bank.com/transfer">
  <input type="hidden" name="to" value="attacker">
  <input type="hidden" name="amount" value="10000">
</form>
<script>document.f.submit();</script>
```

POST /transfer HTTP/1.1 — Cookie: AuthToken=39e839f928ab79 — to=attacker&amount=10000 → bank.com

CSRF Defense: need mechanism to ensure requests are authentic

referer: identify where visitors are coming from

secret token: * session dependent

SameSite cookies: strict: not any cookies from third-party site

LAX: read relevant cookies, but not allow post

SQL injection

' OR 1=1 -- '; DROP TABLE users --

Solution: paramaterized SQL

benefit 1. no need to sanitize input 2. speed up

XSS injection

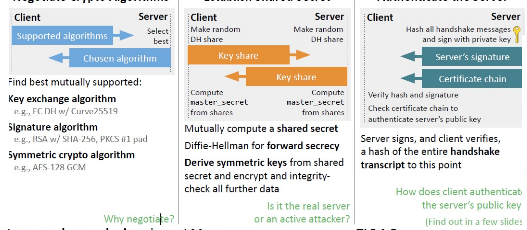
Reflected XSS: Site echoes inputs back to user without properly escaping them

Stored XSS: Site stores and displays user content without properly escaping it.

Defense: old: input validation, output escaping; modern: content security policy: tightly specifying what scripts are allowed to execute

HTTP

HTTP / TLS / TCP — HTTP / TLS / TCP — transport layer security



Find best mutually supported:

Key exchange algorithm e.g., EC DH w/ Curve25519

Signature algorithm e.g., RSA w/ SHA-256, PKCS #1 pad

Symmetric crypto algorithm e.g., AES-128 GCM

Why negotiate?

Mutually compute a **shared secret**

Diffie-Hellman for **forward secrecy**

Derive symmetric keys from shared secret and provide integrity-check all further data

Is it the real server or an active attacker?

Server signs, and client verifies, a hash of the entire **handshake transcript** to this point

How does client authenticate the server's public key? (Find out in a few slides)

A **network round trip** takes ≈100 ms.

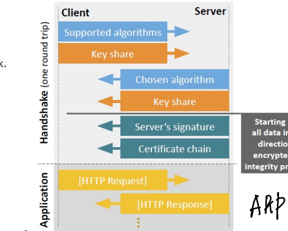
To **minimize latency**, TLS 1.3 handshake works in one round trip.

Clever design: Client *guesses* which key exchange algorithm the server will pick. (If guessed wrong, must add another round-trip to send a new key share.)

Make sure you understand how TLS 1.3 achieves these properties:

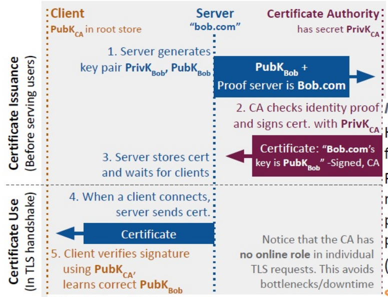
- **Confidentiality** (AEAD ciphers)
- **Message Integrity** (AEAD ciphers)
- **Authentication** of server by client (Public key crypto and certificates)

TLS 1.3



Public key and certificates

Three parties:



CAs sometimes issue **intermediate CA** certificates, which lend permission to sign further certificates

Used to:

- Delegate trust to other CAs
- Use separate key for issuing certs. from long-term root key stored offline [Why?]

Servers provide a **certificate chain**. Client verifies signature in each link of the chain, back to a trusted root CA certificate the client trusts

certificate chain

Subject:	eecs388.org
Public Key:	98:98:58:eb:ec:cb:b6:77:81:…
Signed by:	Let's Encrypt Intermediate CA R11

Use to verify

Subject:	Let's Encrypt Intermediate CA R11
Public Key:	ba:87:bc:5c:1b:00:39:cb:ca:…
Signed by:	ISRG Root CA X1

Certificates provided by server during TLS handshake

Use to verify

Subject:	ISRG Root CA X1
Public Key:	ad:e8:24:73:f4:14:37:f3:9b:…
Signed by:	ISRG Root CA X1

Trusted root CA certificate, built into client. **Self-signed**

HTTP Attacks

1. Fooling users: { homographs use punycode to block IDN homograph; stripping HTTP Strict Transport Security; phishing Google safe browsing / phishing resistant auth }
2. Problems with site design: mixed content, cookies (HTTPS, Secure), server name indication (use VPN / Tor)
3. CA weakness { Fooling Validation no-use of some username; multi-perspective validation; Attack on CA: revoke CA }
4. bugs in TLS use formal verification techniques
5. TLS Protocol Vulnerabilities
6. Server Vulnerabilities.

Common protocol vulnerabilities:

Plaintext transmission
Passive attackers can eavesdrop on unencrypted communication.

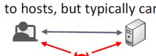
No source authentication
The source address on packets you receive can be spoofed, can't trust it.

No cryptographic integrity
Protocols don't prevent data modification. (Checksums stop data corruption, not attacks.)

No built-in bandwidth control
Attacks can deny service by flooding hosts or the network itself with traffic.

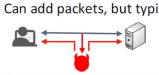
Network threat models:

Off-path attacker. **Network participant.** Can talk to hosts, but typically can't see victim's packets.



Most attackers, initially

On-path attacker. Sees *copy* of victim's packets. Can add packets, but typically **can't** change/block.



WiFi sniffing or Ethernet passive tap

In-path attacker. MITM. Can see, add, change, or block victim's packets.

Obtain via ARP spoofing, BGP hijacking, DNS attacks

IP Spoofing: Fake source address, reach destination and return to fake source.

ARP Spoofing: Host X can fine IP traffic between hosts A and B to flow through X:

1. claim IPA is at attacker MX
2. claim IPB is at attacker MX
3. Re-send IPA's traffic to MA, and vice versa

Border Gateway Protocol

BGP hijacking: compromised ISP can announce someone else's network.

Defense: RPKI, cryptographic method

Network Address Translation (NAT)

Issue: How to allow many devices at home?

Home routers assign each device an address from Private Address Range (via DHCP).

Router rewrites Internet-bound packets to replace private source address with its public IP address. Recognizes response packets and rewrites in reverse.

(This complicates hosting servers at home.)

Security benefit: Devices at private IPs not directly addressable from the output world.

TCP

TCP on-path: easy

TCP off-path: need to guess seq num

DNS

dns hijacking hack into home router

host file hijacking change OS hosts

DNS monitoring/blocking fake responses

off-path DNS cache poisoning Send a lot of responses

Solution: randomized UDP source port; randomized capitalization in query; encrypted DNS