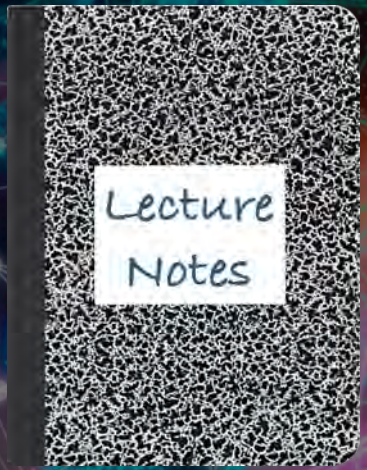
CS 419 – Computer Security

Week 3 Recitation

Integrity Review



What We'll Cover

- Vocabulary from the lecture
- Reading a real certificate chain and a real signature check
- The limits of each mechanism
- Questions to check your own understanding

Integrity Vocabulary Review

Four Mechanisms

Hash function → digest
Public function, no key.
Anyone can compute a digest.

MAC → authentication tag
Shared key.
Both parties can compute and forge.

Signature
Private key signs, public key verifies.

Certificate
A signed statement binding a key to a name.

The distinguishing question among these:
Who can produce a value that verifies the data?

Anyone, both parties, one party, or an authority vouching for one party.

The Three Resistance Properties

Preimage
Given a digest, find any message producing it.

Second preimage
Given a message, find another with the same digest.

Collision
Find any two messages that share a digest.

Work required against an ideal n-bit digest:

2^n	2^n	$2^{n/2}$

Why the third is cheaper (but still infeasible for a good hash function)
Every new message is compared against every message already tried, so the number of pairs grows as the square of the number of attempts.

How Verification Works

Verifying a Download

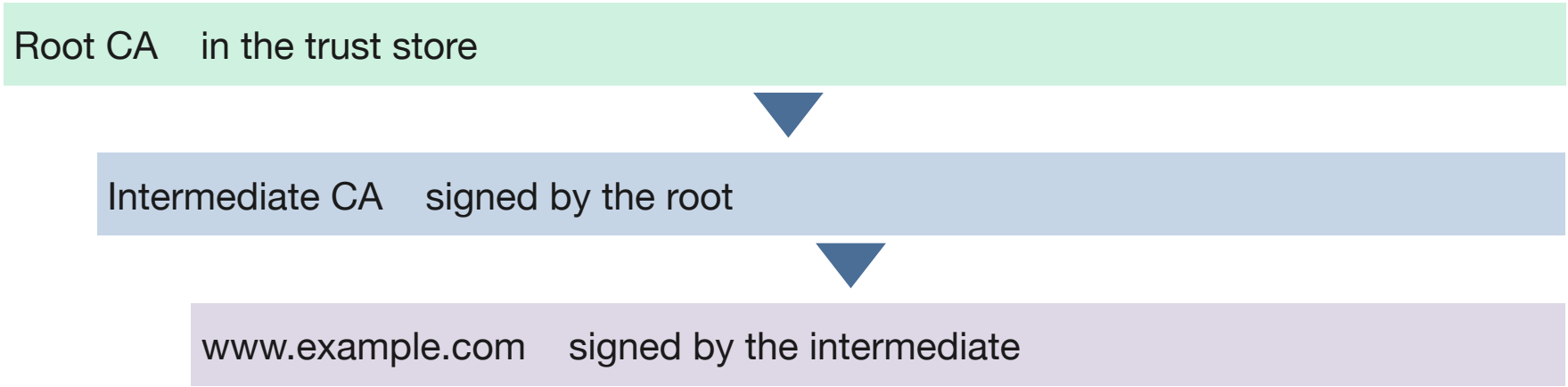
Verifying a downloaded package, step by step:

1. Establish the publisher's key:
   Check its certificate chain, or use a key already installed
2. Check the manifest signature with that key
3. Compute the digest of the file you received
4. Compare your digest with the one in the signed manifest

A signature check means nothing until you know whose key it is
An attacker who controls the download server can replace the file, the digest, and the signature, but not the key your system already trusts.

Reading a Certificate Chain

What a browser shows for a certificate chain:



At each step
Signature, dates, and what the issuer may sign.

Once
The name, and proof of the private key.

What a Certificate Contains

Subject
The identity the certificate describes.

Subject public key
The key being bound to that identity.

Issuer
The authority that signed this certificate.

Validity period
The dates outside which it is not accepted.

Extensions
Names covered, and whether the key may sign certificates.

Signature
The issuer's signature over all of the above.

Everything above the signature is what the signature protects

Example Certificate: www.rutgers.edu

Some fields in the X.509 certificate of rutgers.edu

Subject	**Org:** Rutgers, The State University of New Jersey **Common name:** www.rutgers.edu	Identifies **who** owns this certificate
Issuer Name	InCommon RSA Server CA 2	Identifies the certificate authority (CA)
Signature algorithm	SHA-256 with RSA Encryption	What algorithm is used to sign this certificate
Not valid before:	January 25, 2026	Validity period
Not valid after:	January 26, 2027	Validity period
Public key algorithm	RSA Encryption	What type of public key does Rutgers use?
Public key	**512 bytes:** C5 20 02 02 12 13 …	One part of the RSA key
Public key Exponent	65537	2nd part of the RSA key
Key size	4096 bits	Size of the key

Signature	
Signature	384 bytes: 82 4F A2 7F …
Algorithm	SHA-256 ECDSA
Timestamp	January 26, 2026 3:11pm

Example Certificate: www.google.edu

A certificate fetched from www.google.com (fields are shortened).

Subject: CN = www.google.com

Subject Alternative Names: www.google.com, *.google.com, youtu.be, ...

Public Key: RSA 2048 bits, exponent 65537

Issuer: C = US, O = Google Trust Services, CN = WR2

Validity: about three months

Extensions: CA:FALSE, serverAuth, CRL and OCSP locations, SCTs

Run it yourself

```
openssl s_client -connect www.google.com:443 -servername www.google.com
```

Google's Chain, End to End

GTS Root R1 Google Trust Services in the trust store

WR2 intermediate, CA:TRUE, signed by the root

www.google.com CA:FALSE, signed by WR2

The server sends
Its own certificate and the intermediate.

The browser supplies
The root, from its own store.

The Limits of Each Mechanism

Hash alone
Fails when the attacker controls the content and the digest.

MAC
Fails for one publisher and many verifiers.

Signature
Fails when nobody can say whose key it is.

Certificate
Fails when the signed build was already malicious.

Each mechanism answers the previous failure
Knowing what each one does **not** do is a better guide to choosing between them than knowing what each one does.

MAC Against Signature

	MAC	Signature
Keys	one shared secret	private and public
Who verifies	key holders only	anyone
Who can forge	any key holder	nobody
Non-repudiation	no	yes
Speed	fast	slower by orders

A TLS connection uses both: a signature once, a MAC on every record

Matching Defenses to Attacks

All three produce software that verifies correctly

The difference is what failed, and that determines which defense helps.

Stolen key

A cryptographic mechanism failed.

Defense: keys in hardware.

Compromised build

Cryptography was perfect.

Defense: reproducible builds.

Compromised dependency

Nothing was tested.

Defense: review and pinning.

Signing proves the last link

Every defense here protects a step that happens before the signature exists.

Why Key Exchange Uses Diffie-Hellman

What Diffie-Hellman does

Two parties who have never met end up holding the same secret, and that secret never goes over the network.

Neither side chooses the secret

The secret is the result of the exchange. Even a compromised random number generator on one side does not break security.

New keys can be created quickly

A private value is one random number and a public value is one exponentiation, so creating a new pair per connection costs almost nothing.

Nothing to decrypt later

No message contains the key, so an eavesdropper who records the back-and-forth exchange seens only the two public values.

It does not authenticate anybody

An active attacker can run the exchange as well. We rely on signatures and certificates to do authentication

Compare this with sending a key encrypted under an RSA public key

Common Confusions

- A hash is **not** encryption, and there is nothing to reverse
- A MAC is **not** a signature, because both parties can produce one
- Signing is **not** encrypting with the private key
- A self-signed root is trusted because it is installed, not because it is signed
- A valid signature says nothing about whether the code is safe

The End