Chapter 13: Security

Operating Systems

- Security goals and threats
- Attackers and defense in depth
- Encryption, hashes, signatures, PKI, TLS
- Protection domains and the access matrix
- ACLs and capabilities, least privilege
- Bell-LaPadula, Biba, covert channels
- Authentication: passwords, MFA, passkeys
- Software exploits and defenses
- Insider attacks
- Malware
- Sandboxes, containers, SELinux
- Secure boot and side channels

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/os/slides/Chapter_13_Security.html

Learning Outcomes

Goals and threats

Distinguish confidentiality, integrity and availability, attack types and attacker profiles.

Cryptography

Explain symmetric and asymmetric encryption, hash functions, digital signatures, certificates and the TLS handshake.

Protection

Read the access matrix as ACLs and as capabilities; apply least privilege, setuid and Linux capabilities.

Formal models

Apply the Bell-LaPadula and Biba rules; explain why covert channels cannot be closed.

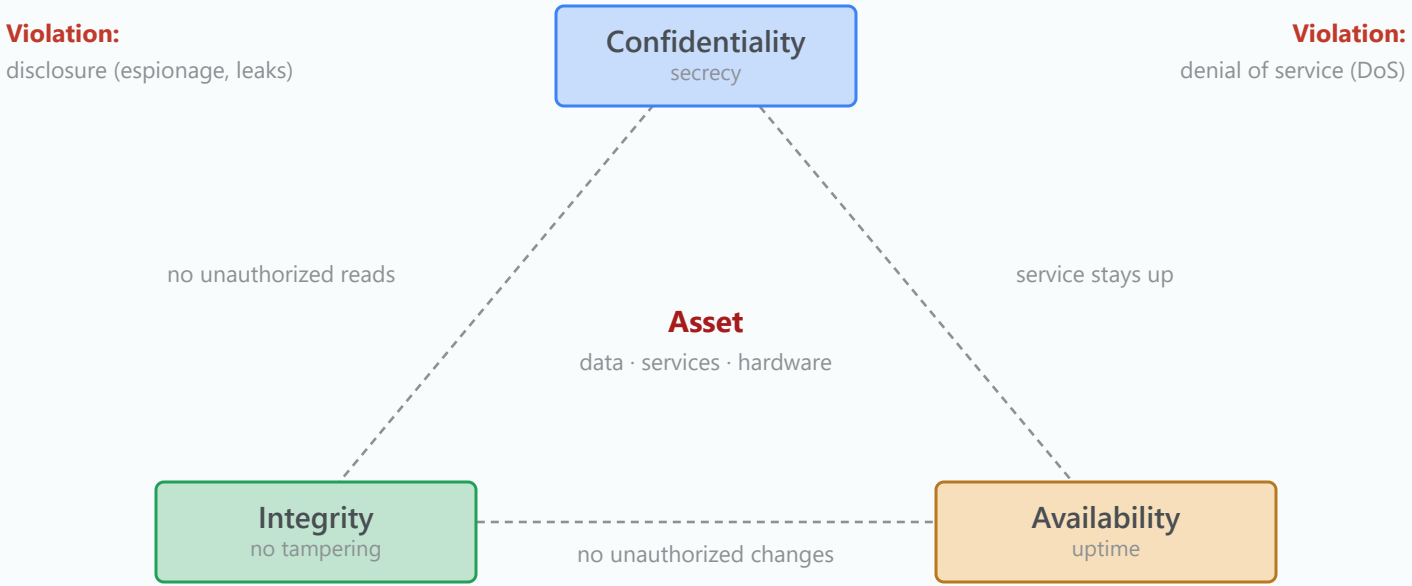
Authentication

Compare salted slow password hashes, MFA, TOTP and FIDO2 passkeys.

Attacks and defenses

Recognize memory-safety bugs, TOCTOU and malware; canaries, NX, ASLR, CFI, sandboxes and secure boot.

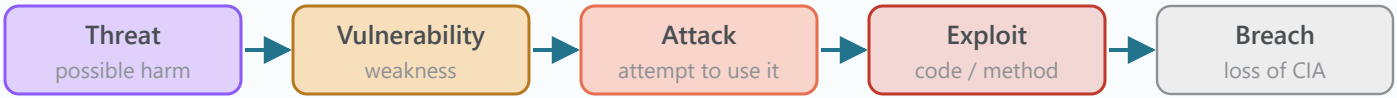
Security Goals: The CIA Triad



Confidentiality keeps data secret, **integrity** keeps it unaltered, **availability** keeps it usable. Often added: **authenticity** and **accountability** (nonrepudiation, audit logs).

Threats and Breach Categories

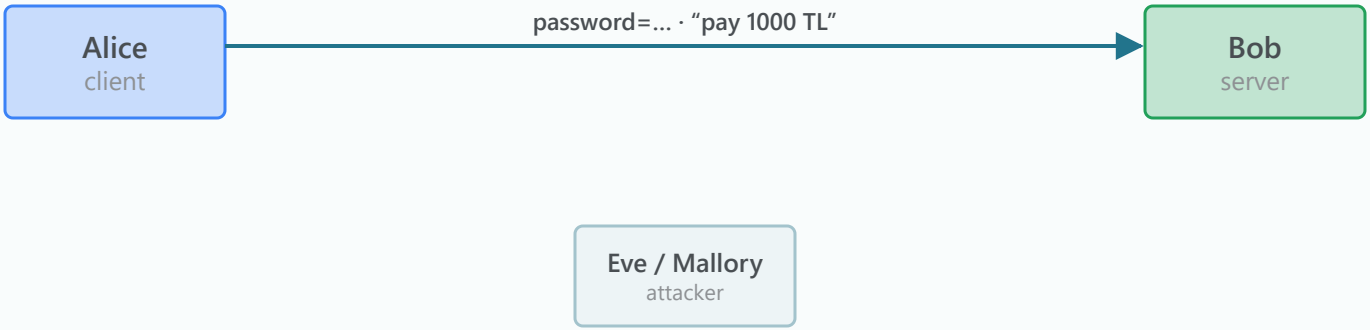
Category	Definition	Example
Breach of confidentiality	Unauthorized reading of data	A credit-card database is leaked
Breach of integrity	Unauthorized modification of data	Grades tampered with in a records system
Breach of availability	Destruction of data or service	Ransomware encrypts the disk
Theft of service	Unauthorized use of resources	Crypto mining on a hijacked server
Denial of service (DoS/DDoS)	Preventing legitimate use	SYN flood from a botnet
Masquerading	Pretending to be someone else	A password stolen by phishing
Privilege escalation	Gaining more rights than granted	Becoming root through a bug in a setuid program



patches fix vulnerabilities, authentication stops attacks, canaries/NX stop exploits, backups limit the damage

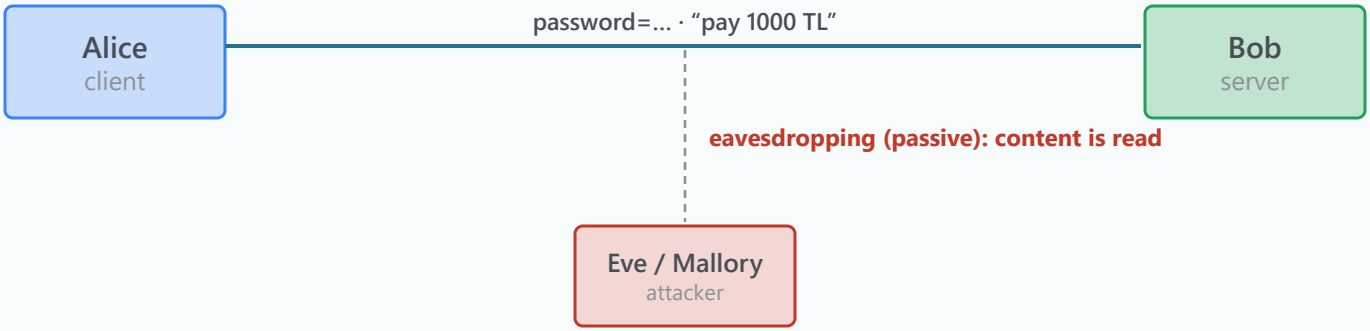
A **threat** is potential harm, a **vulnerability** is a weakness that makes it possible, an **attack** is an attempt to use the vulnerability, and an **exploit** is the code or method that does it.

Network Attacks: Eavesdropping, Replay, Man in the Middle



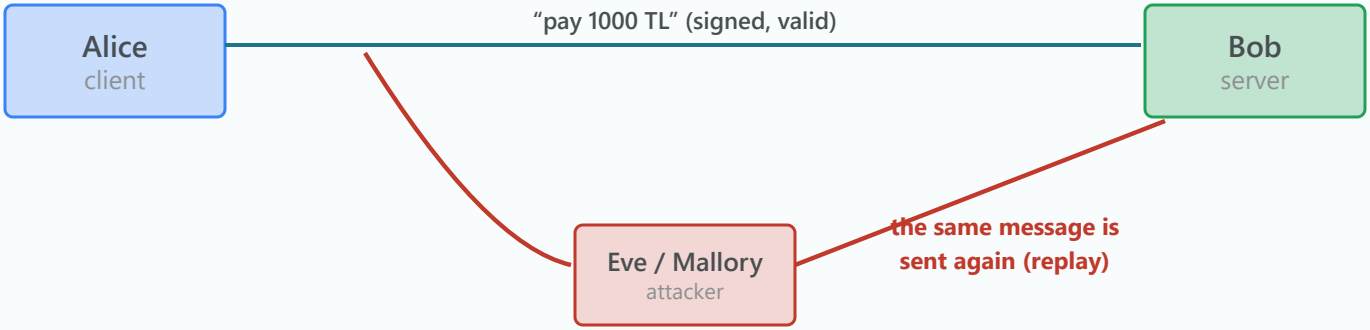
Normal traffic between Alice and Bob. If the channel is unencrypted and unauthenticated, every attack below is possible.

Network Attacks: Eavesdropping, Replay, Man in the Middle



Eavesdropping: a passive attacker copies the traffic → breach of confidentiality. Fix: encryption (TLS).

Network Attacks: Eavesdropping, Replay, Man in the Middle



Replay attack: even an encrypted, signed message can be resent. Fix: a nonce, a timestamp, a sequence number.

Network Attacks: Eavesdropping, Replay, Man in the Middle



Man in the middle (MITM): the attacker poses as the other party to both sides. Fix: verify the peer's identity with a certificate.

Network Attacks: Eavesdropping, Replay, Man in the Middle



Session hijacking: the session token issued after authentication is stolen. Fix: HTTPS only, `Secure` / `HttpOnly` cookies, short lifetimes.

Attackers and Accidental Data Loss

Curious insider

Little technical skill; “peeks” at others' email or files. Unethical, often illegal.

Malicious employee

Abuses their authority: data theft, sabotage, logic bombs. The hardest group to detect.

Cybercriminal

Profit-driven: ransomware, identity and card theft, botnets for hire. The most common threat today.

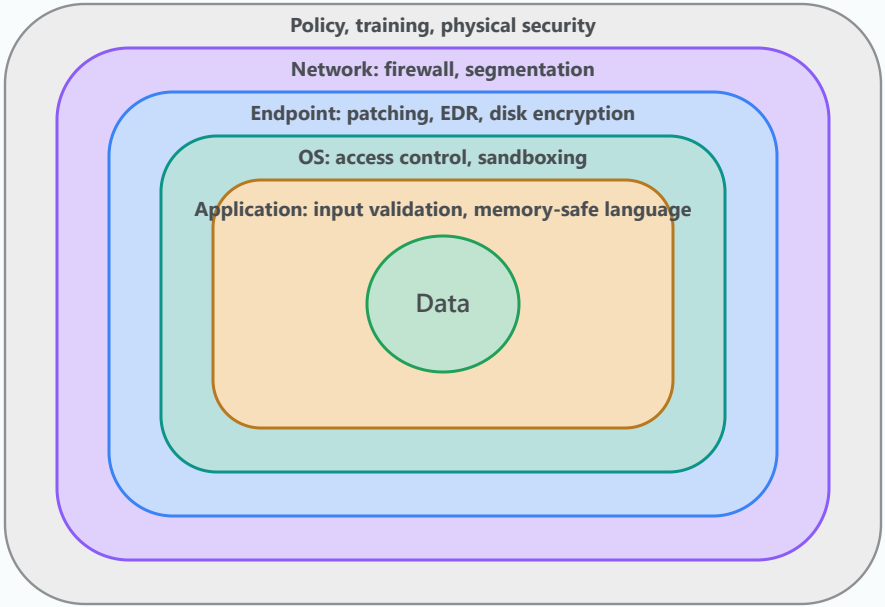
State-sponsored groups (APT)

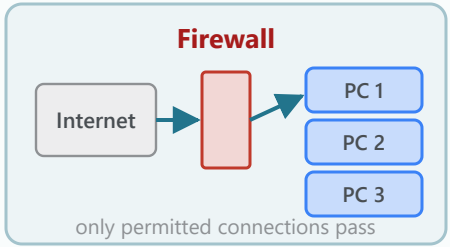
Military and commercial espionage, sabotage of critical infrastructure; zero-day exploits and supply-chain attacks.

Accidental data loss	Example	Countermeasure
Acts of nature	fire, flood, earthquake (and rats gnawing tapes)	geographically separate backup
Hardware/software error	disk failure, CPU fault, software bug	RAID, checksums, versioning
Human error	wrong command, formatting the wrong disk, lost device	3-2-1 backups, encrypted disks

Accidents lose more data than attacks do. 3-2-1: 3 copies, 2 different media, 1 copy offsite and preferably immutable — which also helps against ransomware.

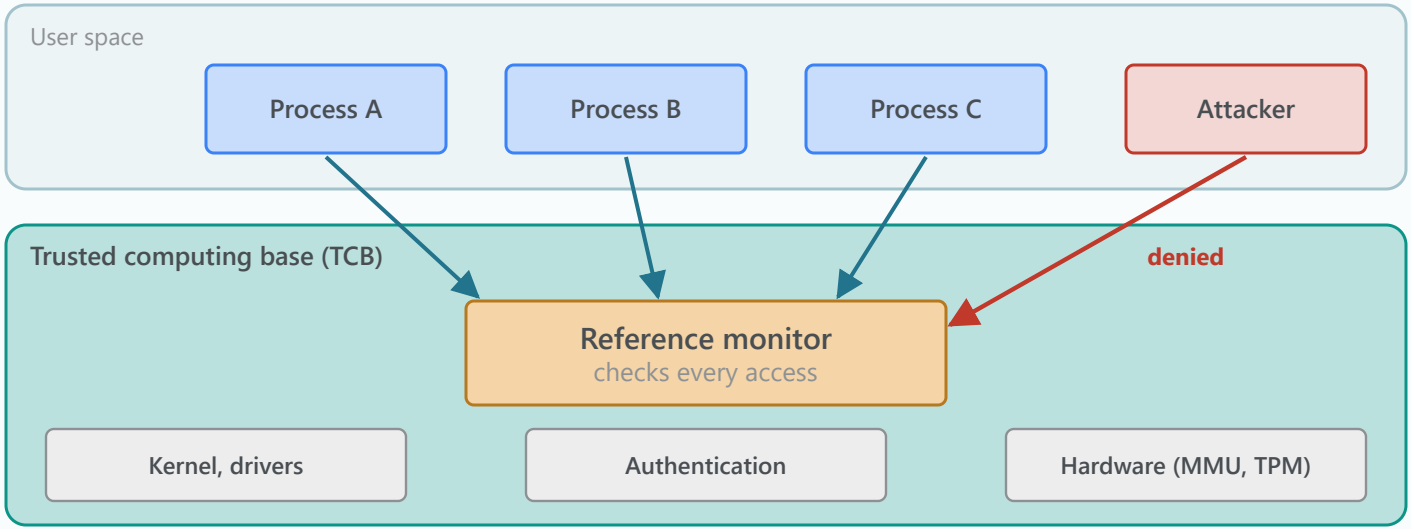
Defense in Depth and Good Practice





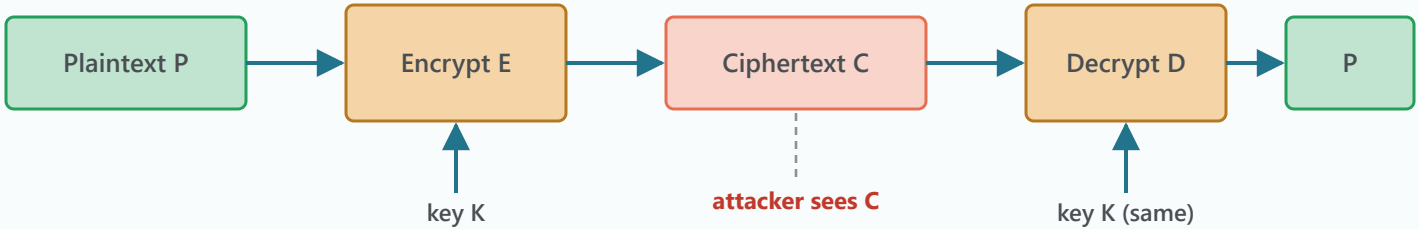
- **Patching**: apply security updates quickly (most attacks use known vulnerabilities).
- **Strong authentication** and least-privilege access control; **logging** and monitoring.
- Regular **penetration testing**; **training** users against phishing; zero trust: the inside of the network is not trusted either.

The Trusted Computing Base (TCB)



A reference monitor must meet three conditions: it is invoked on **every** access (cannot be bypassed), it is **tamperproof**, and it is **small** enough to verify. The bigger the TCB, the likelier a bug — hence the interest in microkernels and formally verified kernels such as seL4.

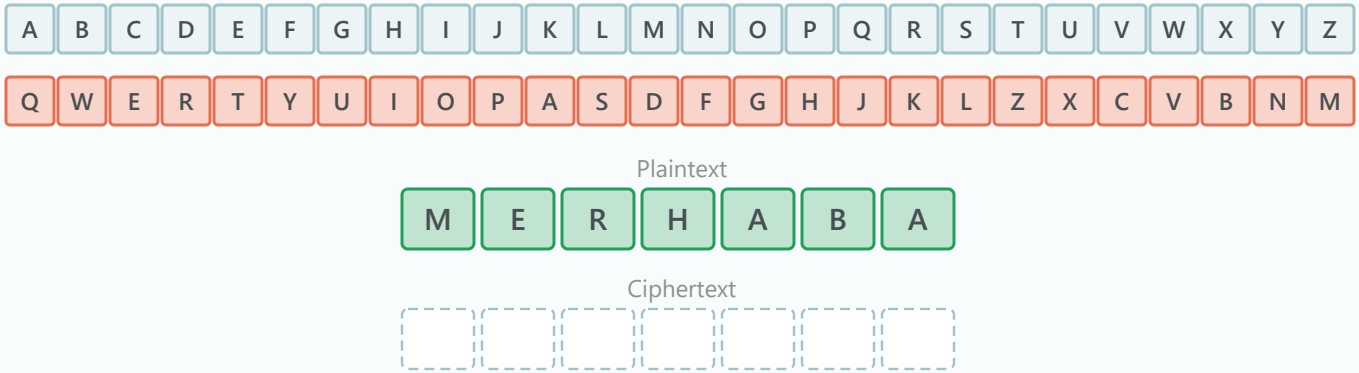
Encryption Basics and Secret-Key Cryptography



$$C = E(P, K) \cdot P = D(C, K)$$

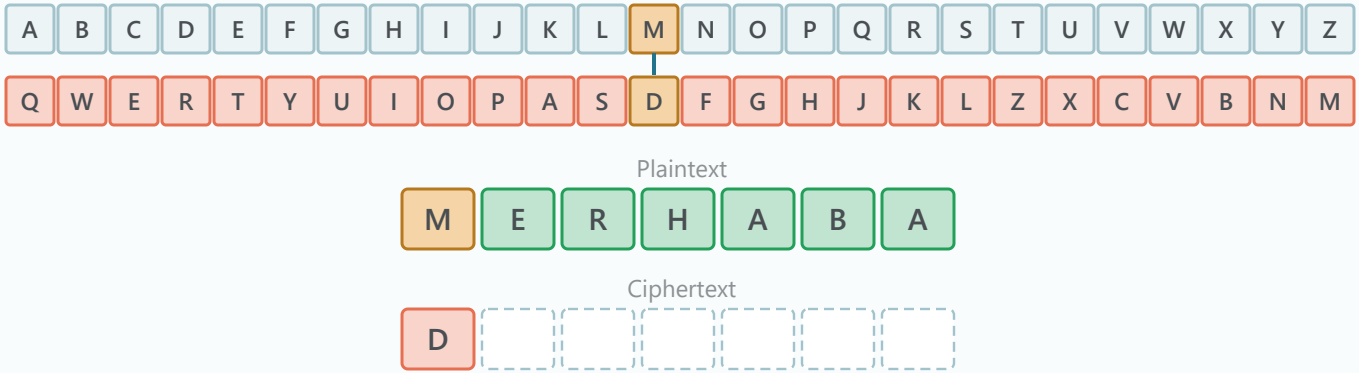
Symmetric (secret-key) encryption: the same K encrypts and decrypts. **Kerckhoffs's principle**: the algorithm may be public; security must rest only on keeping the key secret.

Encryption Basics and Secret-Key Cryptography



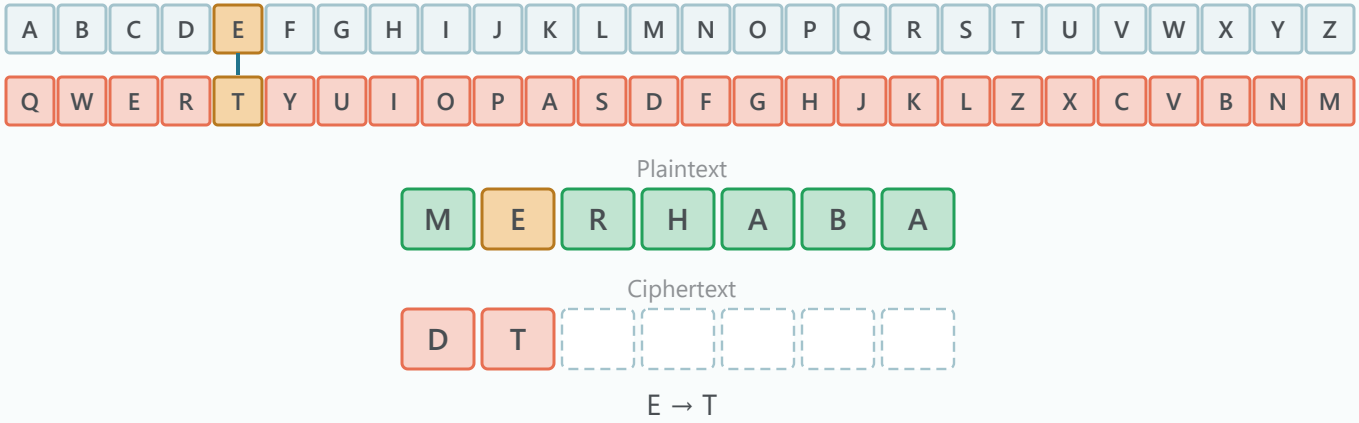
Monoalphabetic substitution: each letter maps to a fixed letter; the key is this 26-letter permutation ($26! \approx 4 \times 10^{26}$ possible keys).

Encryption Basics and Secret-Key Cryptography

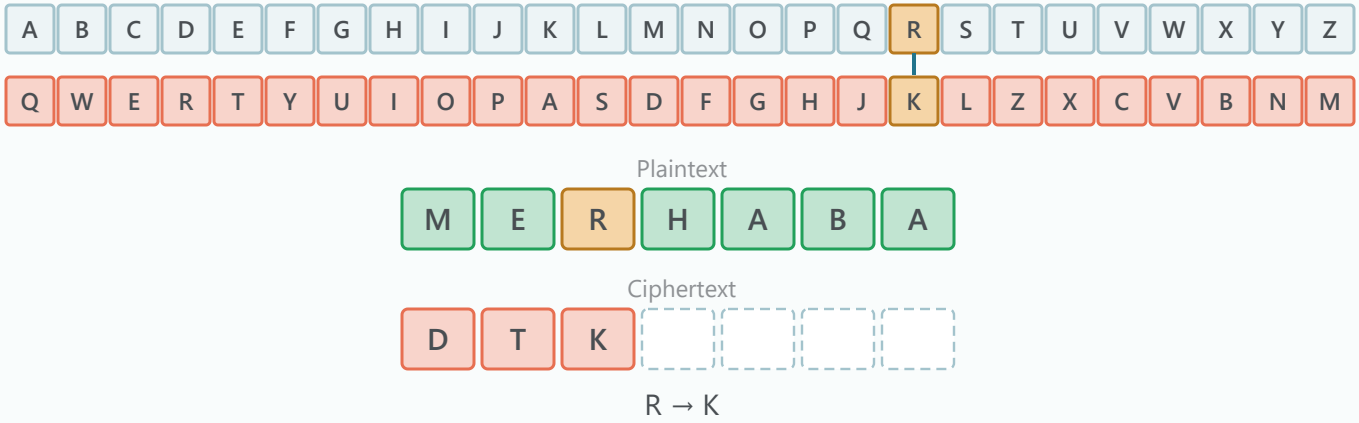


M → D: the letter under M in the table is D.

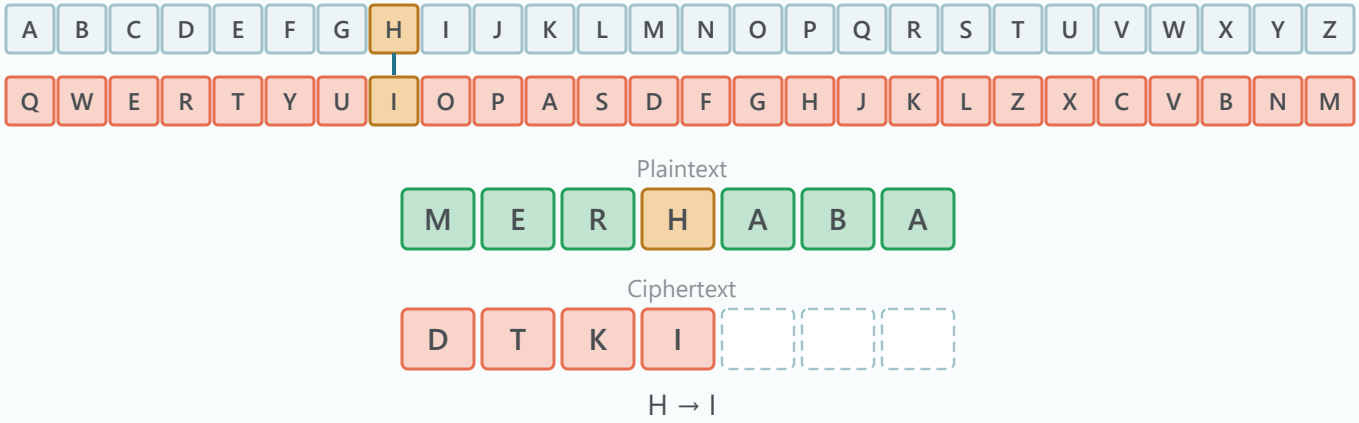
Encryption Basics and Secret-Key Cryptography



Encryption Basics and Secret-Key Cryptography



Encryption Basics and Secret-Key Cryptography



Encryption Basics and Secret-Key Cryptography

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Q	W	E	R	T	Y	U	I	O	P	A	S	D	F	G	H	J	K	L	Z	X	C	V	B	N	M

Plaintext

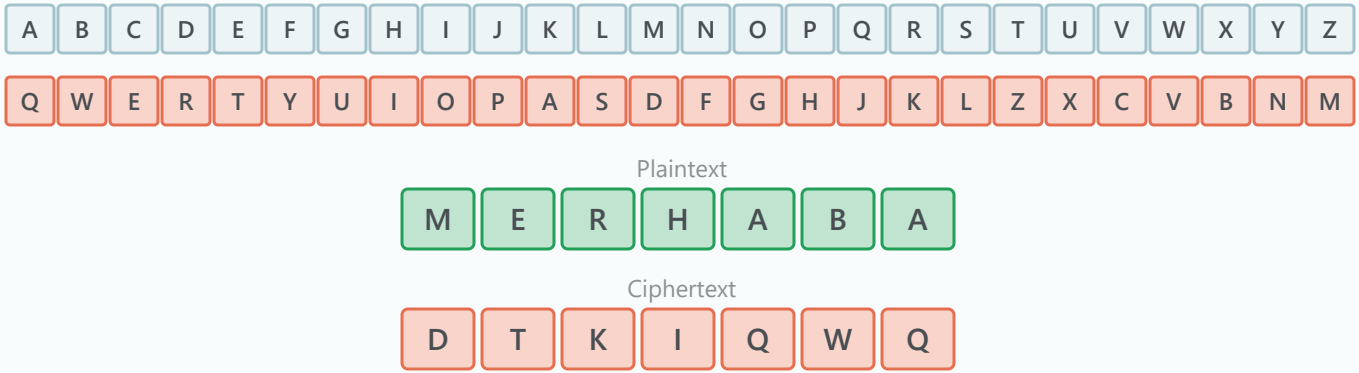
M	E	R	H	A	B	A

Ciphertext

D	T	K	I	Q	W	Q

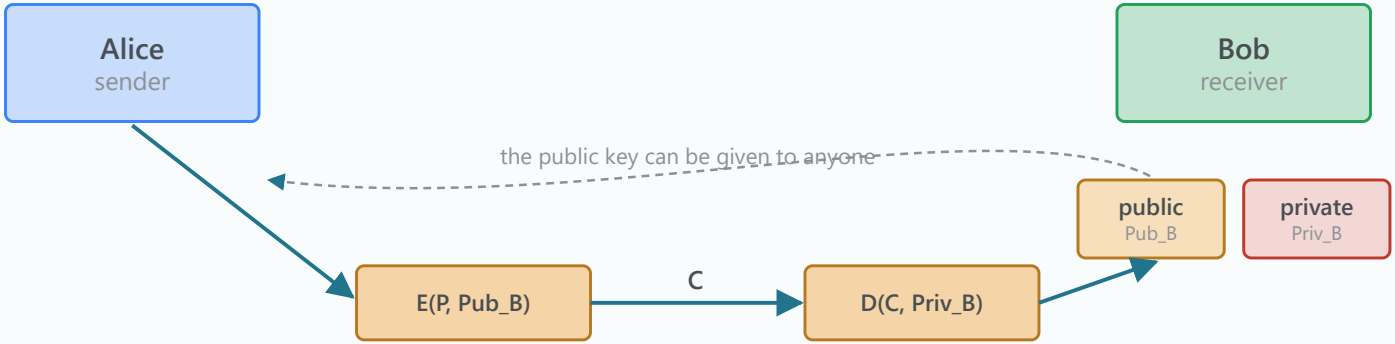
Result: MERHABA (“hello”) → **DTKIQWQ**. Both A's became Q: letter frequencies survive.

Encryption Basics and Secret-Key Cryptography



The key space is huge, yet **frequency analysis** (in English E, T and A dominate) breaks the cipher with a few lines of text. Today: **AES-256-GCM**, **ChaCha20-Poly1305** (encryption + integrity together).

Public-Key Cryptography

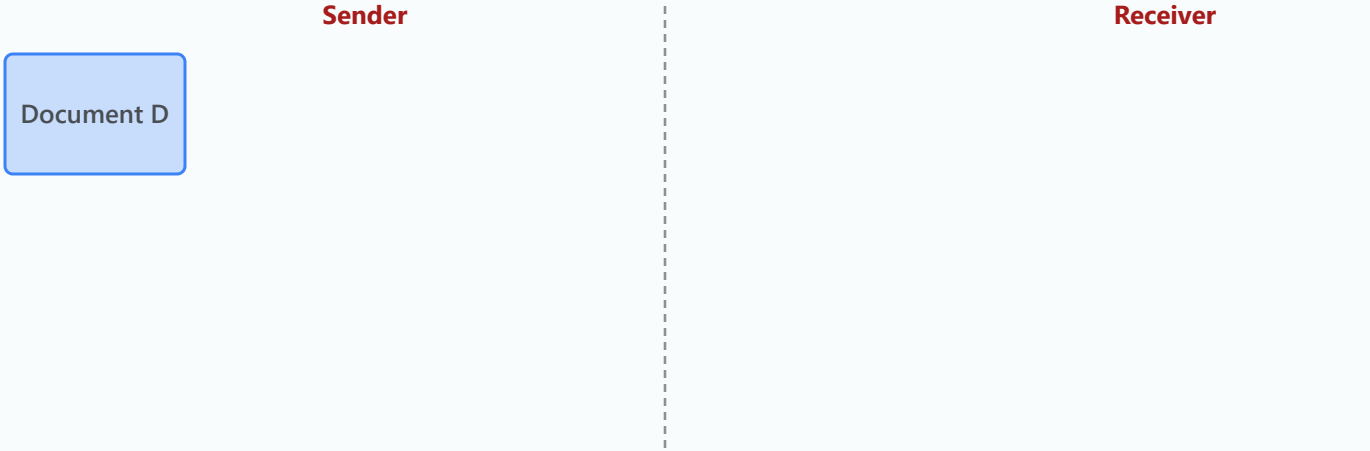


What Pub_B encrypts, only Priv_B decrypts.

	Symmetric	**Asymmetric (public key)**
Key	One, shared	Public + private pair
Speed	Very fast (AES-NI)	100–1000× slower
Use	Bulk data	Key exchange, signatures
Example	AES, ChaCha20	RSA, ECDH/X25519, Ed25519, ML-KEM

Asymmetric encryption rests on **one-way trapdoor functions**: multiplying two large primes is easy, factoring the product (RSA) or taking a discrete logarithm (ECC) is hard. Quantum-resistant ML-KEM was standardized in 2024 (FIPS 203).

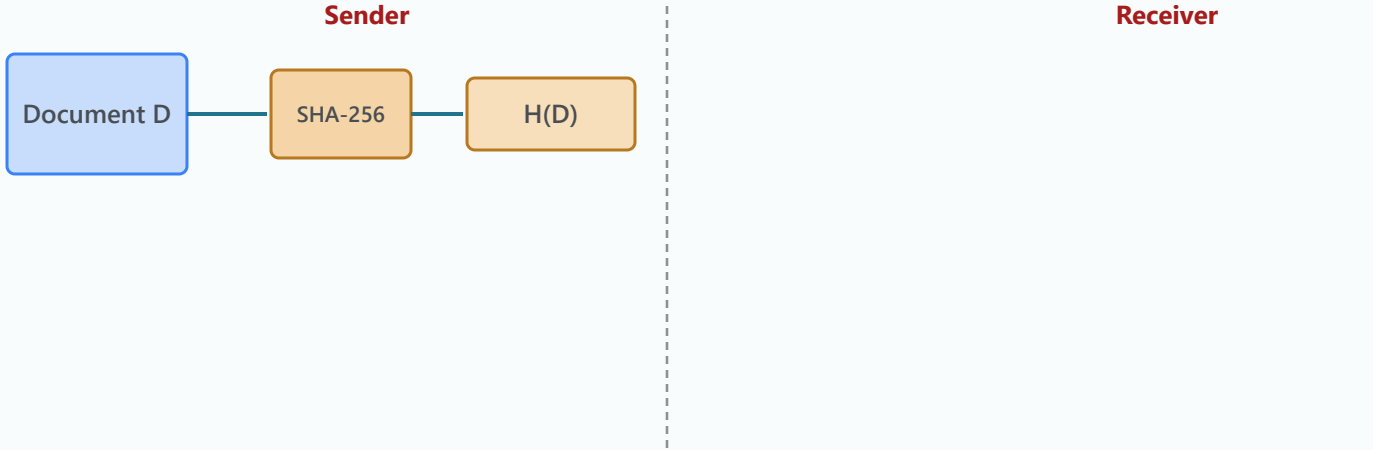
Hash Functions and Digital Signatures



The sender wants to deliver document D unaltered and provably from them (secrecy is not required).

A hash is not compression; an **unsigned** hash proves nothing on its own (the attacker can change the hash along with the document).

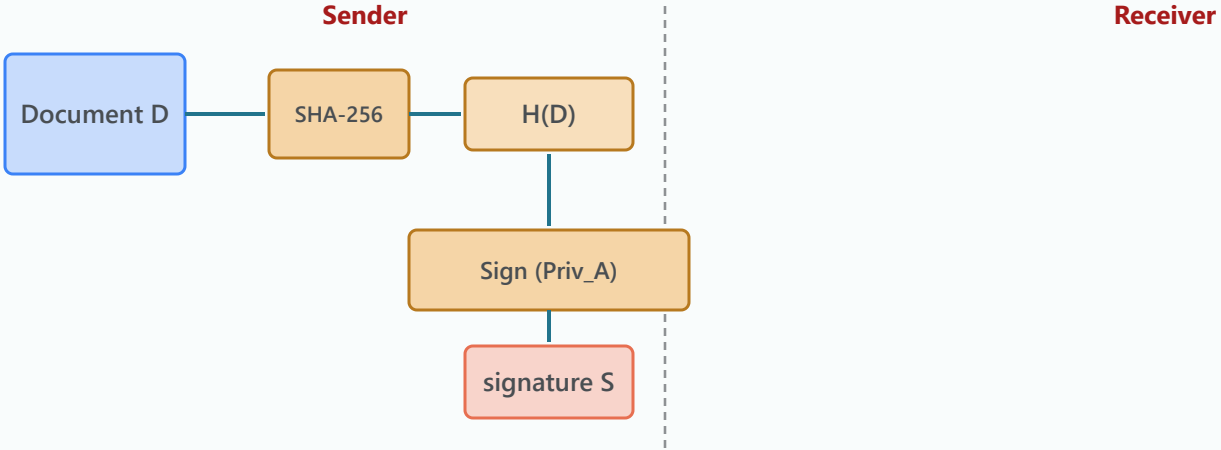
Hash Functions and Digital Signatures



A **hash function** (SHA-256, SHA-3) turns input of any size into a fixed 256-bit digest: one-way and collision-resistant; flipping one bit changes half the digest. MD5 and SHA-1 are broken.

A hash is not compression; an **unsigned** hash proves nothing on its own (the attacker can change the hash along with the document).

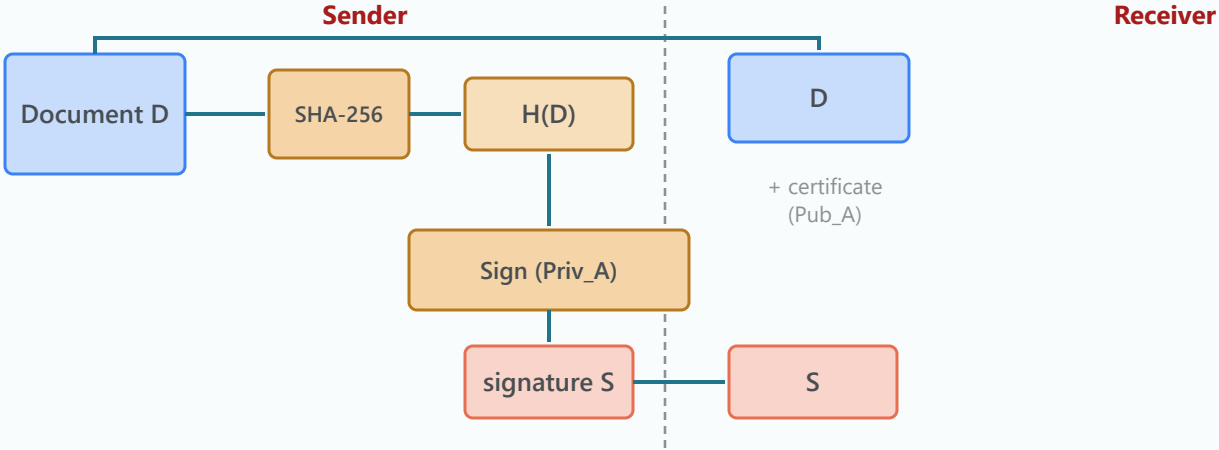
Hash Functions and Digital Signatures



The digest is signed **with the sender's private key** (RSA-PSS, ECDSA, Ed25519). Only the holder of Priv_A can do this.

A hash is not compression; an **unsigned** hash proves nothing on its own (the attacker can change the hash along with the document).

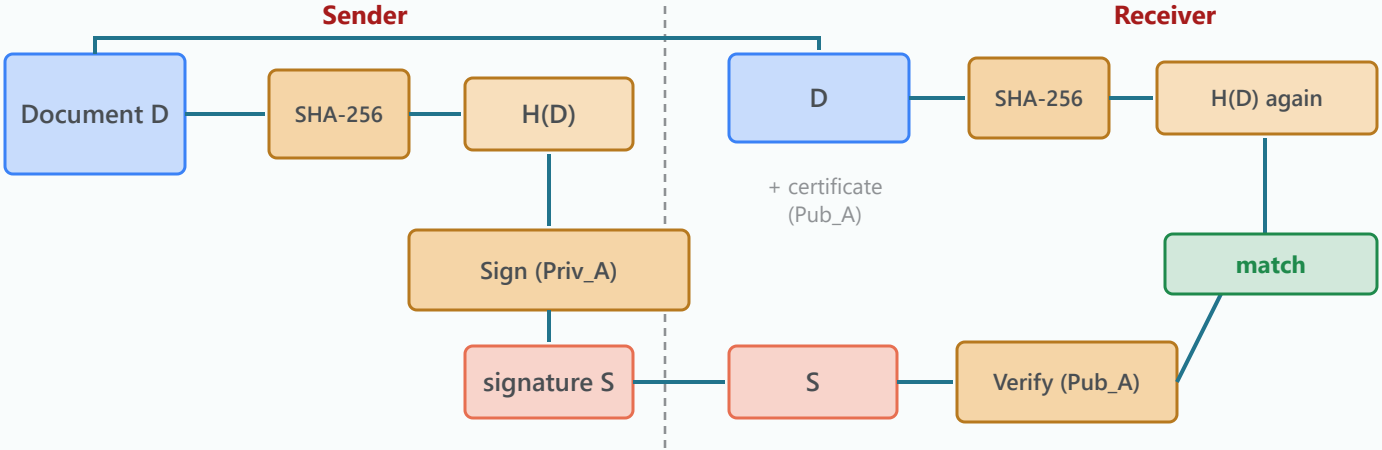
Hash Functions and Digital Signatures



The document, the signature and a certificate carrying the sender's public key are sent together.

A hash is not compression; an **unsigned** hash proves nothing on its own (the attacker can change the hash along with the document).

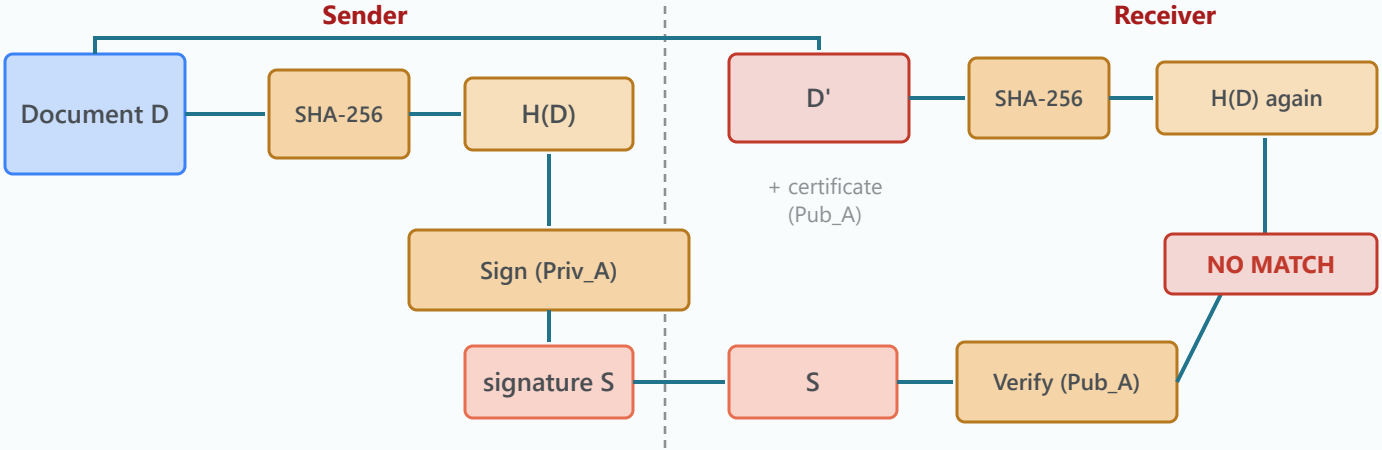
Hash Functions and Digital Signatures



The receiver computes the digest itself and verifies the signature with Pub_A: if they match, the document is **intact** and came **from Alice** (nonrepudiation).

A hash is not compression; an **unsigned** hash proves nothing on its own (the attacker can change the hash along with the document).

Hash Functions and Digital Signatures

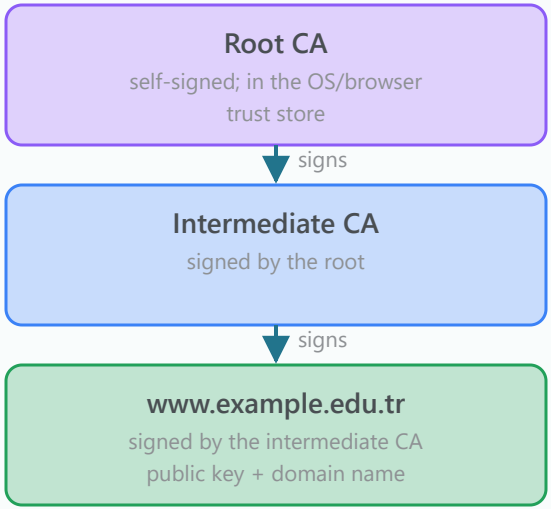


If a single byte changes in transit (D′), the digest no longer matches and verification fails. The attacker cannot make a new valid signature without Priv_A.

A hash is not compression; an **unsigned** hash proves nothing on its own (the attacker can change the hash along with the document).

CRYPTOGRAPHY

Certificates and Public-Key Infrastructure (PKI)



Browser verification

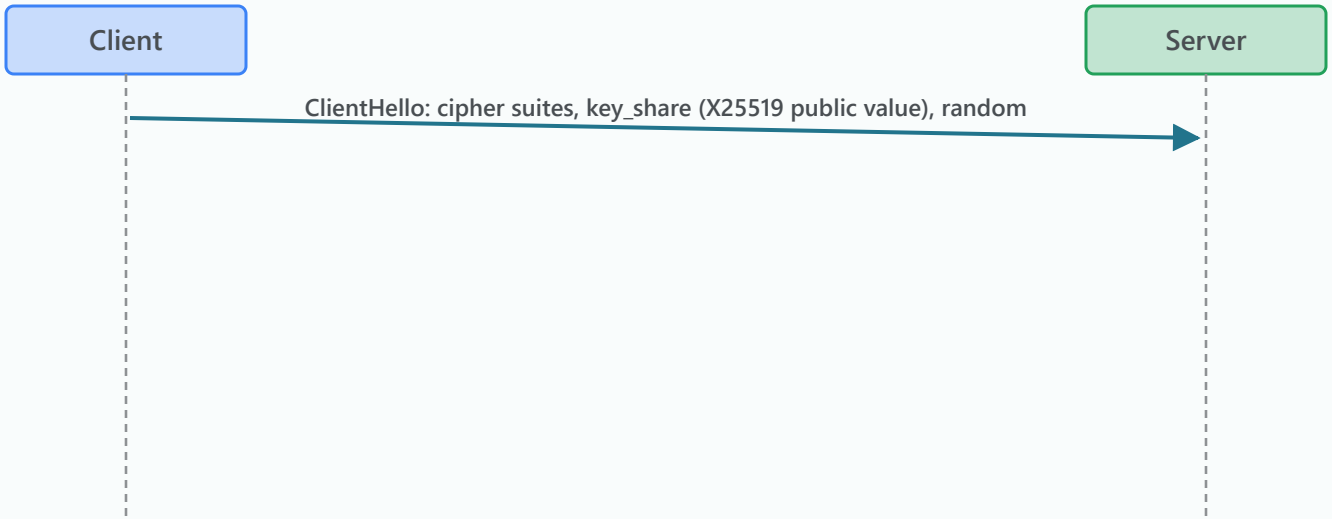
1. Server cert + intermediate CA arrive
2. Signatures verified along the chain
3. Is the root in the trust store?
4. Domain name matches? Expiry?
5. Revocation (OCSP/CRL) and CT log
6. All good → connection trusted

Forged certificate (MITM)

not signed by a trusted CA, so no chain can be built → warning

A **certificate** (X.509) = public key + identity (domain name) + validity period + CA signature. It answers the question of whether a public key really belongs to its claimed owner. Thanks to Let's Encrypt (ACME), certificates are free and last 90 days; lifetimes are set to shrink gradually to 47 days.

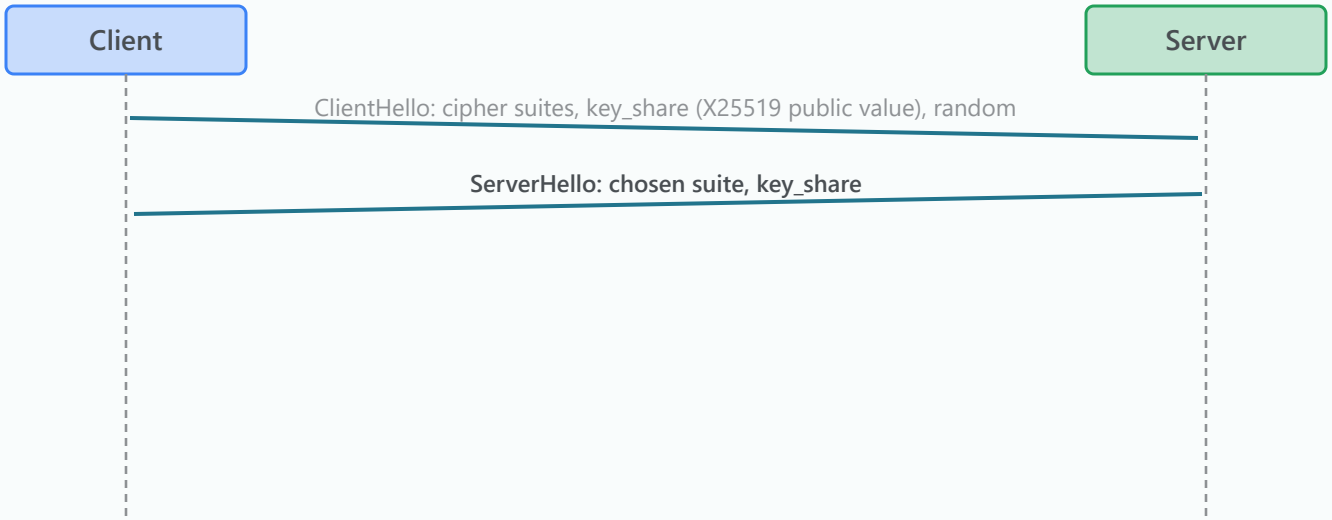
The TLS 1.3 Handshake (Simplified)



{ } = sent encrypted (green)

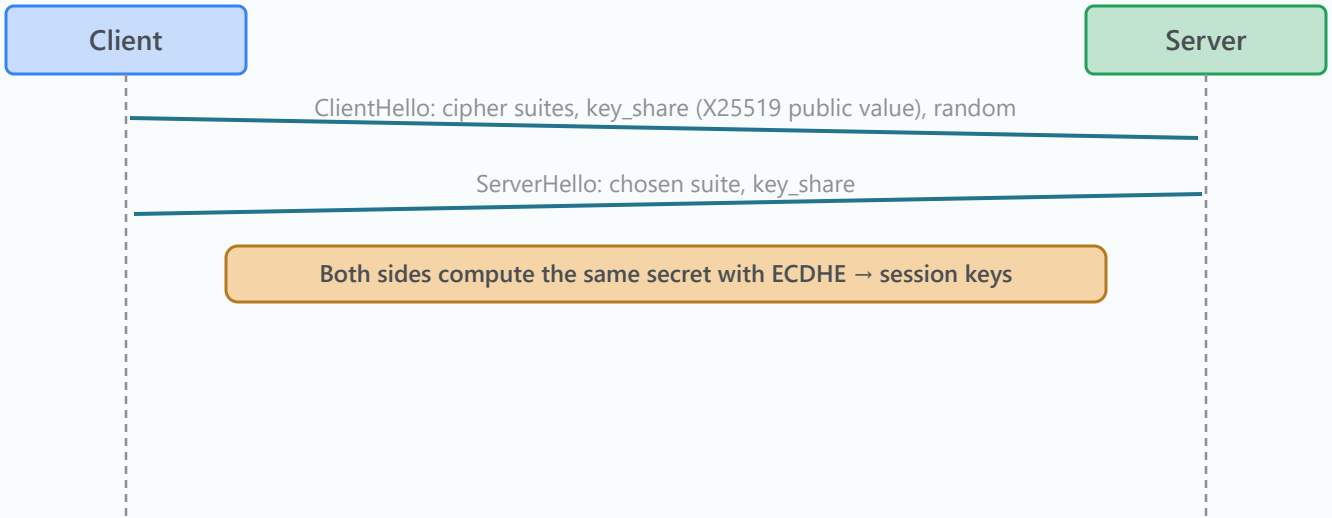
In its first message the client sends the algorithms it supports and an **ephemeral** Diffie-Hellman public value.

The TLS 1.3 Handshake (Simplified)



The server picks one and sends its own DH public value. Everything so far is in the clear.

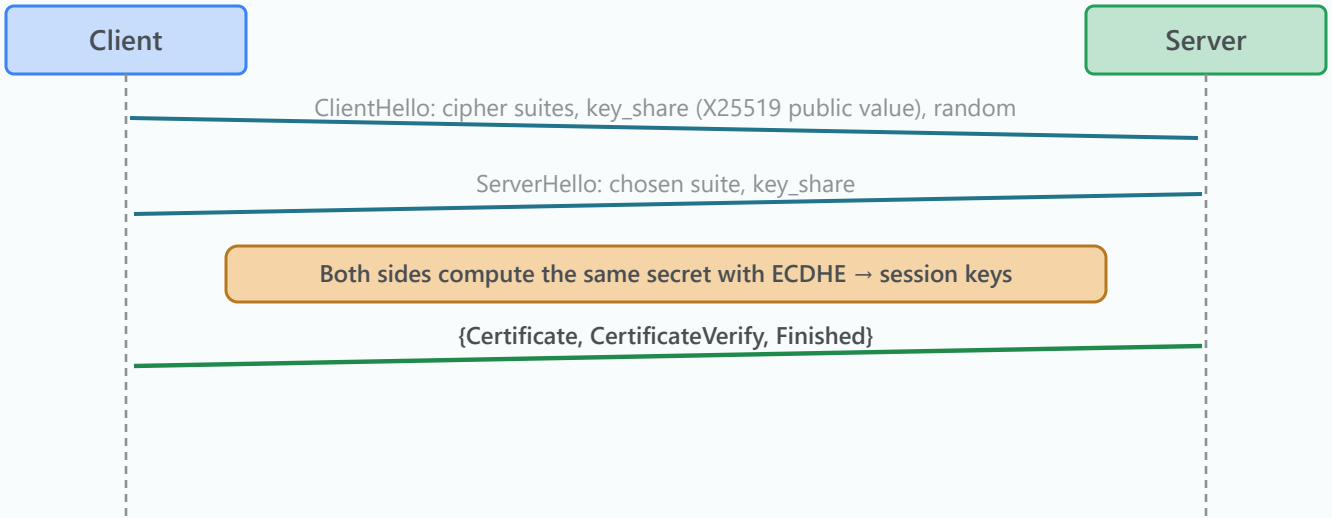
The TLS 1.3 Handshake (Simplified)



{ } = sent encrypted (green)

Each side combines the peer's public value with its own secret to compute **the same shared secret**; the secret never crosses the network.

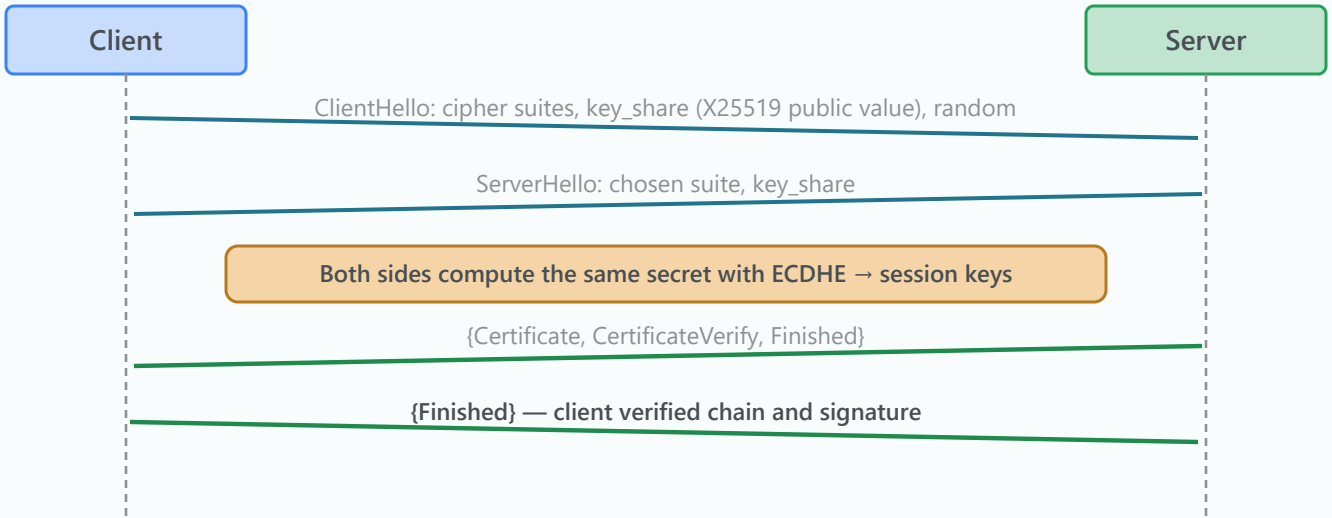
The TLS 1.3 Handshake (Simplified)



{ } = sent encrypted (green)

The server sends its certificate and **signs the handshake transcript with its private key** (CertificateVerify): a MITM cannot produce this signature.

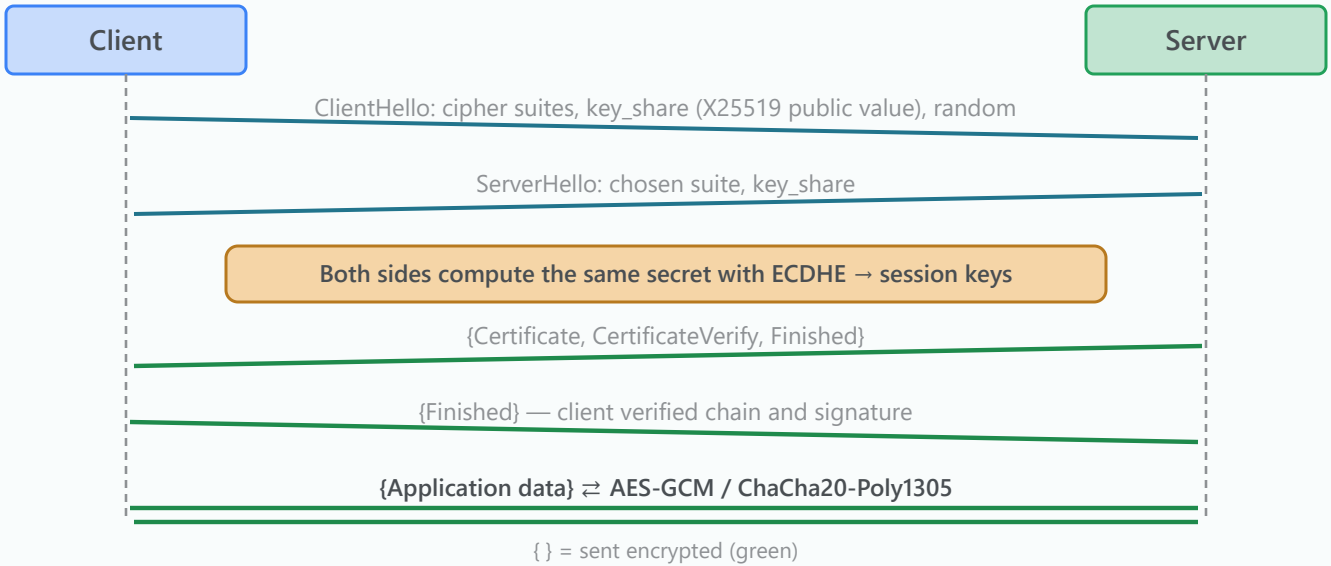
The TLS 1.3 Handshake (Simplified)



{ } = sent encrypted (green)

The client verifies the chain and signature, and its Finished confirms the handshake was not tampered with. Just **one round trip** (1-RTT).

The TLS 1.3 Handshake (Simplified)



Application data flows under symmetric, authenticated encryption. Ephemeral keys give **forward secrecy**: even if the private key is stolen later, old traffic cannot be decrypted.

Diffie-Hellman Key Exchange: Code Trace

```
 1 long modpow(long g, long e, long p) {
 2     long r = 1;
 3     while (e > 0) {
 4         if (e & 1) r = r * g % p;
 5         g = g * g % p;
 6         e >>= 1;
 7     }
 8     return r;
 9 }
10 int main(void) {
11     /* public: p = 23, g = 5 */
12     long A  = modpow(5, 6, 23);
13     long B  = modpow(5, 15, 23);
14     long sA = modpow(B, 6, 23);
15     long sB = modpow(A, 15, 23);
16 }
```

Alice	network (everyone sees)	Bob
a = 6 (secret)	p = 23, g = 5	b = 15 (secret)
s = ?	A = ?	s = ?
	B = ?	

modpow(5, 6, 23)

e: 1 1 0 r = 1 g = 5

$e = 6 = 110_2$; bits processed right to left

Alice has picked secret **a = 6**; her public value A = 5^6 mod 23 is being computed. Square-and-multiply scans the exponent's bits right to left; r = 1.

Step 1 / 8

VARIABLES

g = 5 e = 6 p = 23 r = 1

OUTPUT

Diffie-Hellman Key Exchange: Code Trace

```
1  long modpow(long g, long e, long p) {
2      long r = 1;
▶      while (e > 0) {
▶          if (e & 1) r = r * g % p;
▶          g = g * g % p;
▶          e >>= 1;
7      }
8      return r;
9  }
10 int main(void) {
11     /* public: p = 23, g = 5 */
12     long A  = modpow(5, 6, 23);
13     long B  = modpow(5, 15, 23);
14     long sA = modpow(B, 6, 23);
15     long sB = modpow(A, 15, 23);
16 }
```

Alice	network (everyone sees)	Bob
a = 6 (secret)	p = 23, g = 5	b = 15 (secret)
s = ?	A = ?	s = ?
	B = ?	

modpow(5, 6, 23)

e: 1 1 0 r = 1 g = 2

bit 0 = 0 → r unchanged

e = 6 is even: r stays. The base is squared: g = 5·5 mod 23 = **2**; e = 3.

Step 2 / 8

VARIABLES

g = 2 e = 3 p = 23 r = 1

OUTPUT

Diffie-Hellman Key Exchange: Code Trace

```
1  long modpow(long g, long e, long p) {
2      long r = 1;
▶      while (e > 0) {
▶          if (e & 1) r = r * g % p;
▶          g = g * g % p;
▶          e >>= 1;
7      }
8      return r;
9  }
10 int main(void) {
11     /* public: p = 23, g = 5 */
12     long A  = modpow(5, 6, 23);
13     long B  = modpow(5, 15, 23);
14     long sA = modpow(B, 6, 23);
15     long sB = modpow(A, 15, 23);
16 }
```

Alice	network (everyone sees)	Bob
a = 6 (secret)	p = 23, g = 5	b = 15 (secret)
s = ?	A = ?	s = ?
	B = ?	

modpow(5, 6, 23)

e: 1 1 0 r = 2 g = 4

bit 1 = 1 → r = r·g mod p

e = 3 is odd: r = 1·2 = **2**. g = 2·2 = 4; e = 1.

Step 3 / 8

VARIABLES

g = 4 e = 1 p = 23 r = 2

OUTPUT

Diffie-Hellman Key Exchange: Code Trace

```
1  long modpow(long g, long e, long p) {
2      long r = 1;
▶      while (e > 0) {
▶          if (e & 1) r = r * g % p;
▶          g = g * g % p;
▶          e >>= 1;
7      }
8      return r;
9  }
10 int main(void) {
11     /* public: p = 23, g = 5 */
12     long A  = modpow(5, 6, 23);
13     long B  = modpow(5, 15, 23);
14     long sA = modpow(B, 6, 23);
15     long sB = modpow(A, 15, 23);
16 }
```

Alice	network (everyone sees)	Bob
a = 6 (secret)	p = 23, g = 5	b = 15 (secret)
s = ?	A = ?	s = ?
	B = ?	

modpow(5, 6, 23)

e: 1 1 0 r = 8 g = 16

bit 2 = 1 → r = r·g mod p

e = 1 is odd: r = 2·4 = **8**. g = 16; e = 0, the loop ends. 3 rounds instead of 6 multiplications: an n-bit exponent takes only n rounds.

Step 4 / 8

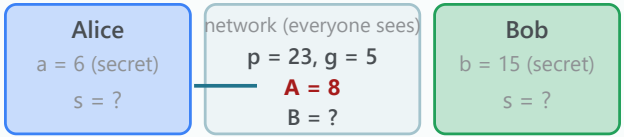
VARIABLES

g = 16 e = 0 p = 23 r = 8

OUTPUT

Diffie-Hellman Key Exchange: Code Trace

```
1  long modpow(long g, long e, long p) {
2      long r = 1;
3      while (e > 0) {
4          if (e & 1) r = r * g % p;
5          g = g * g % p;
6          e >>= 1;
7      }
▶      return r;
9  }
10 int main(void) {
11     /* public: p = 23, g = 5 */
▶      long A  = modpow(5, 6, 23);
13     long B  = modpow(5, 15, 23);
14     long sA = modpow(B, 6, 23);
15     long sB = modpow(A, 15, 23);
16 }
```



A = modpow(5, 6, 23)

A = 8

Alice sends A to Bob over the network

The function returns **8**: A = 8. This value is public; a = 6 never leaves Alice.

Step 5 / 8

VARIABLES

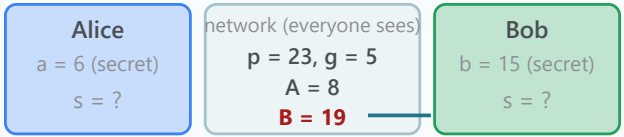
A = 8

OUTPUT

```
A = 8
```

Diffie-Hellman Key Exchange: Code Trace

```
1  long modpow(long g, long e, long p) {
2      long r = 1;
3      while (e > 0) {
4          if (e & 1) r = r * g % p;
5          g = g * g % p;
6          e >>= 1;
7      }
8      return r;
9  }
10 int main(void) {
11     /* public: p = 23, g = 5 */
12     long A  = modpow(5, 6, 23);
▶      long B  = modpow(5, 15, 23);
14     long sA = modpow(B, 6, 23);
15     long sB = modpow(A, 15, 23);
16 }
```



B = modpow(5, 15, 23)

e = 15 = 1111_2: 4 rounds → B = 19

Bob sends B to Alice over the network

Bob runs the same function with his secret b = 15: 5^{15} mod 23 = **19**. B = 19 travels to Alice.

Step 6 / 8

VARIABLES

A = **8** B = **19**

OUTPUT

```
A = 8
B = 19
```

Diffie-Hellman Key Exchange: Code Trace

```
1  long modpow(long g, long e, long p) {
2      long r = 1;
3      while (e > 0) {
4          if (e & 1) r = r * g % p;
5          g = g * g % p;
6          e >>= 1;
7      }
8      return r;
9  }
10 int main(void) {
11     /* public: p = 23, g = 5 */
12     long A  = modpow(5, 6, 23);
13     long B  = modpow(5, 15, 23);
▶      long sA = modpow(B, 6, 23);
15     long sB = modpow(A, 15, 23);
16 }
```

Alice	network (everyone sees)	Bob
a = 6 (secret)	p = 23, g = 5	b = 15 (secret)
s = 2	A = 8	s = ?
	B = 19	

sA = modpow(19, 6, 23)

Alice: B^a mod p = 19^6 mod 23 = 2

the shared secret never crossed the network

Alice raises Bob's public value to her secret exponent: 19^6 mod 23 = **2**.

Step 7 / 8

VARIABLES

A = **8** B = **19** sA = **2**

OUTPUT

```
A = 8
B = 19
sA = 2
```

Diffie-Hellman Key Exchange: Code Trace

```
 1 long modpow(long g, long e, long p) {
 2     long r = 1;
 3     while (e > 0) {
 4         if (e & 1) r = r * g % p;
 5         g = g * g % p;
 6         e >>= 1;
 7     }
 8     return r;
 9 }
10 int main(void) {
11     /* public: p = 23, g = 5 */
12     long A  = modpow(5, 6, 23);
13     long B  = modpow(5, 15, 23);
14     long sA = modpow(B, 6, 23);
▶      long sB = modpow(A, 15, 23);
16 }
```

Alice	network (everyone sees)	Bob
a = 6 (secret)	p = 23, g = 5	b = 15 (secret)
s = 2	A = 8	**s = 2**
	B = 19	

sB = modpow(8, 15, 23)

Bob: A^b mod p = 8^{15} mod 23 = 2

both sides found 5^(6·15) mod 23 = 2

Bob also gets 8^{15} mod 23 = **2**: both computed 5^(a·b) mod p. An eavesdropper sees only p, g, A and B; finding a is the discrete logarithm problem (in practice a 2048-bit p or X25519).

Step 8 / 8

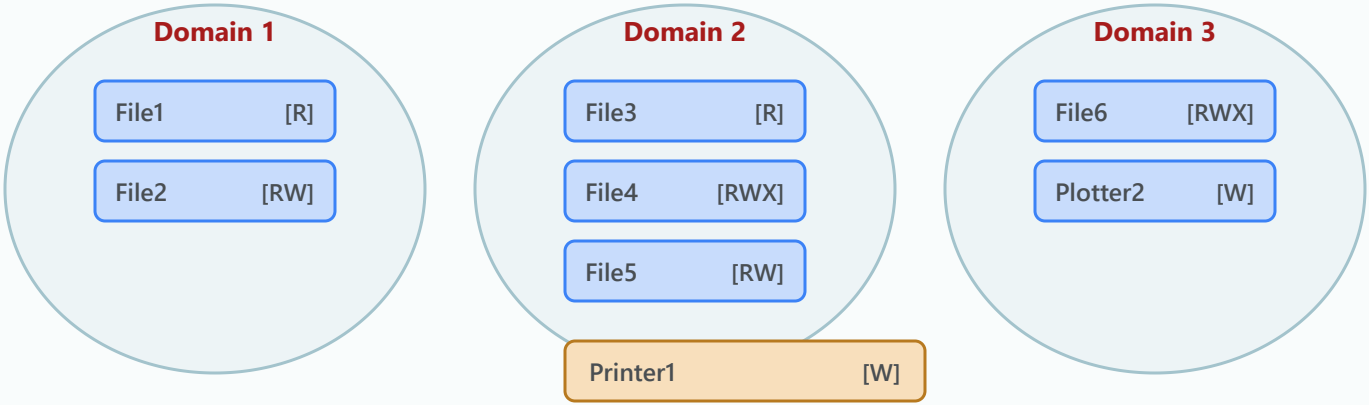
VARIABLES

A = **8** B = **19** sA = **2** sB = **2**

OUTPUT

```
A = 8
B = 19
sA = 2
sB = 2
```

Protection Domains



Printer1 is in both D2 and D3: objects can be shared between domains

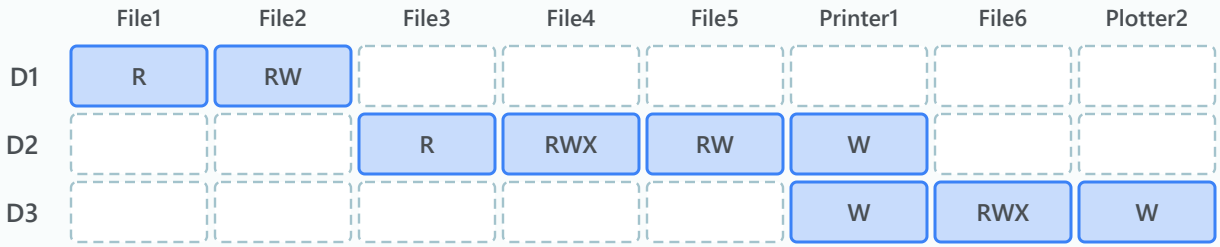
- A **domain** is a set of (object, rights) pairs. At any moment a process runs in one domain.
- UNIX/Linux: domain = (UID, set of GIDs); kernel mode and user mode are also two separate domains.
- Domain switching: running a `setuid` program, entering the kernel through a system call.

The Protection Matrix

	File1	File2	File3	File4	File5	Printer1	File6	Plotter2	D1	D2	D3
D1	R	RW								**Enter**	
D2			R	RWX	RW	W					
D3						W	RWX	W			

- Row = domain, column = object, cell = rights. Domains can be objects too: **Enter** in cell D1 → D2 says a process in D1 may switch to D2.
- The matrix is very **sparse**: thousands of users × millions of files. It is never stored whole; either the columns (ACLs) or the rows (capabilities) are stored.
- **Authorized and unauthorized states**: every state of the matrix is checked against the security policy. If policy says Eve must not read a file, yet an R appears in that cell after some sequence of commands, the system has entered an **unauthorized** state.
- Harrison-Ruzzo-Ullman (HRU, 1976): in general, "can a right leak?" is **undecidable**, which is why restricted models (BLP, Biba) are used.

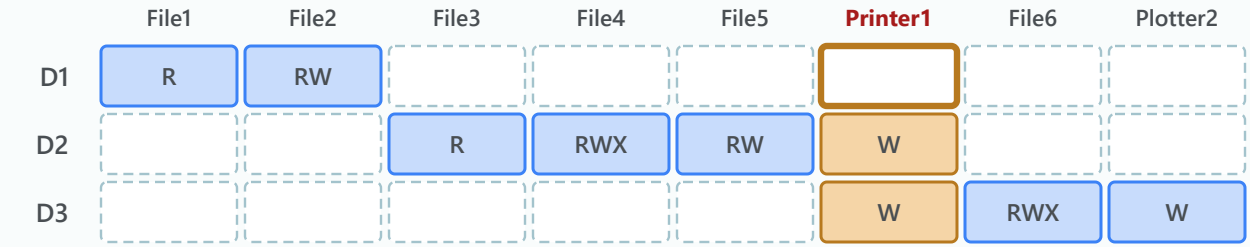
One Matrix, Two Forms: ACLs and Capabilities



15 of 24 cells are empty: the matrix is sparse

The protection matrix of three domains and eight objects.

One Matrix, Two Forms: ACLs and Capabilities

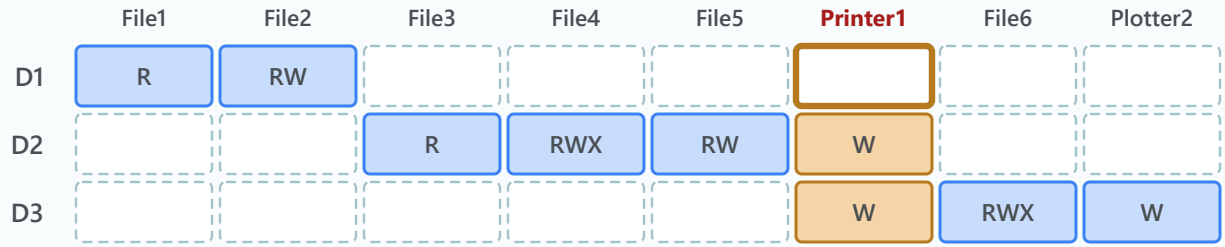


ACL = column (stored with the object)

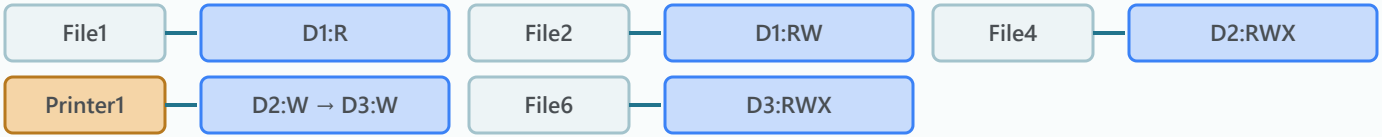


Access control list (ACL): a **column** of the matrix, stored with the object. Printer1's ACL: D2 may write, D3 may write.

One Matrix, Two Forms: ACLs and Capabilities

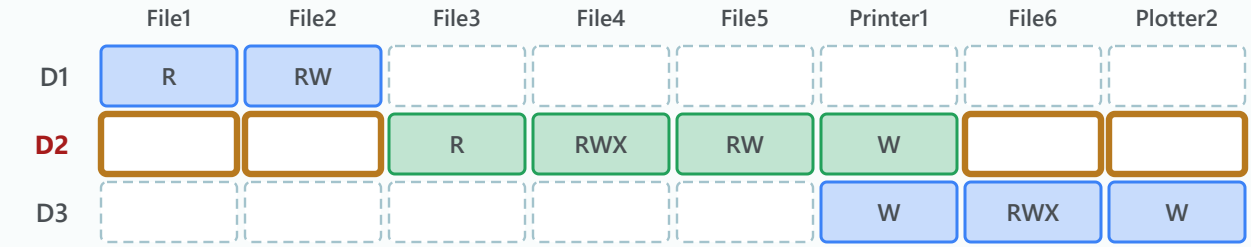


ACL = column (stored with the object)

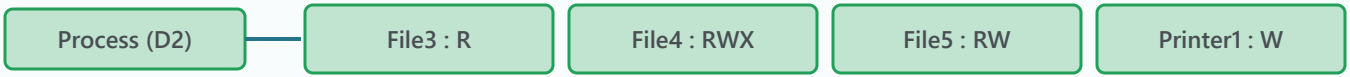


Each object's nonempty entries become its ACL. On access, the kernel looks up the process's identity in the object's ACL.

One Matrix, Two Forms: ACLs and Capabilities

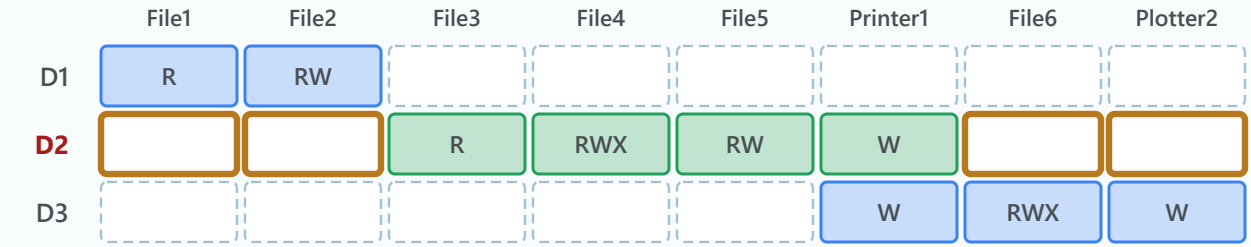


Capability list = row (kept with the process / in the kernel)

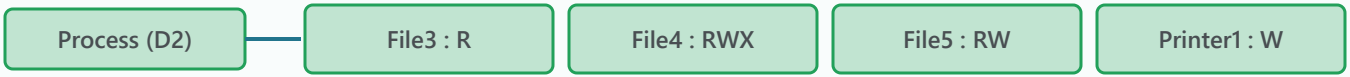


Capability list (C-list): a **row** of the matrix, belonging to the domain (process). D2's capabilities: File3:R, File4:RWX, File5:RW, Printer1:W.

One Matrix, Two Forms: ACLs and Capabilities



Capability list = row (kept with the process / in the kernel)



To access an object the process presents the capability; no name lookup or ACL scan is needed

Comparison: ACLs make “who can access this object?” and **revocation** easy; capabilities make **delegation** and fast checks easy, but revoking every copy is hard.

Access Control Lists in Practice

```
$ ls -l report.txt
-rw-r----- 1 ali accounting 4096 report.txt
 rw-  owner   r--  group  ---  other

$ setfacl -m u:ayse:rw report.txt
$ getfacl report.txt
user::rw-
user:ayse:rw-
group::r--
mask::rw-
other::---
```

Group	A	B	C	D
Group 1	R	RWX	—	—
Group 2	W	—	—	—
Group 3	—	—	X	R
Group 4	—	—	—	—
Admin	RWX	RWX	RWX	RWX

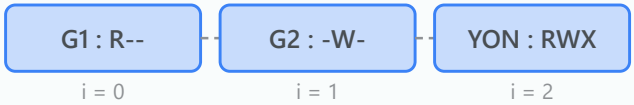
- The 9 rwx bits are fast but coarse; an ACL adds named users/groups, and `mask` caps their maximum rights (`+` in `ls`).
- Pro: easy to understand, easy per-object revocation.
- Con: hard to manage at scale; a list scan on every access (eased by caching).
- NTFS DACL: ordered allow/deny entries (SID + access mask), explicit denies first, inheritance from folders (`icacls`). POSIX, NFSv4 and NTFS ACLs do not map exactly onto each other.

Access Control List: Code Trace

```
1  /* ACL of file A: R=4 W=2 X=1 */
▶  struct entry { int group, rights; } acl[] =
▶      {{G1, 4}, {G2, 2}, {ADM, 7}};
4
5  int access(int group, int req) {
6      for (int i = 0; i < 3; i++)
7          if (acl[i].group == group) {
8              int h = acl[i].rights;
9              return (h & req) == req;
10         }
11     return 0;    /* not listed: denied */
12 }
```

request: —

acl[] (stored with file A)



File A's ACL is column A of the protection matrix: G1 reads, G2 writes, the admin can do anything. G3 and G4 are not listed; empty cells are not stored.

Step 1 / 7

VARIABLES

`entries = 3`

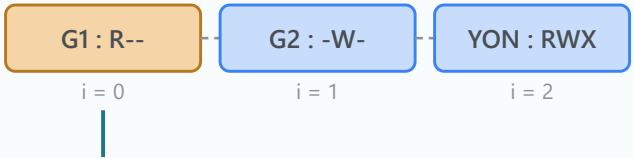
OUTPUT

Access Control List: Code Trace

```
1  /* ACL of file A: R=4 W=2 X=1 */
2  struct entry { int group, rights; } acl[] =
3      {{G1, 4}, {G2, 2}, {ADM, 7}};
4
▶  int access(int group, int req) {
▶      for (int i = 0; i < 3; i++)
▶          if (acl[i].group == group) {
8              int h = acl[i].rights;
9              return (h & req) == req;
10         }
11     return 0;    /* not listed: denied */
12 }
```

request: G2 · W (2)

acl[] (stored with file A)



access(G2, W): the entry at i = 0 belongs to G1, so the group does not match; the scan continues.

Step 2 / 7

VARIABLES

`group = G2` `req = W` `i = 0`

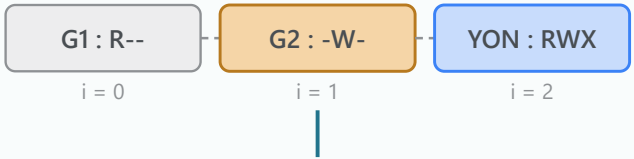
OUTPUT

Access Control List: Code Trace

```
1  /* ACL of file A: R=4 W=2 X=1 */
2  struct entry { int group, rights; } acl[] =
3      {{G1, 4}, {G2, 2}, {ADM, 7}};
4
5  int access(int group, int req) {
6      for (int i = 0; i < 3; i++)
▶          if (acl[i].group == group) {
▶              int h = acl[i].rights;
▶              return (h & req) == req;
10         }
11     return 0;    /* not listed: denied */
12 }
```

request: G2 · W (2) **allow**

acl[] (stored with file A)



G2 found at i = 1: h = 2. Since (2 & 2) == 2, access is **allowed**.

Step 3 / 7

VARIABLES

`group = G2` `req = W` `i = 1` `h = -W-`

OUTPUT

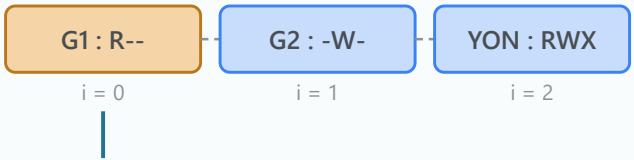
```
G2 W: allow
```

Access Control List: Code Trace

```
1  /* ACL of file A: R=4 W=2 X=1 */
2  struct entry { int group, rights; } acl[] =
3      {{G1, 4}, {G2, 2}, {ADM, 7}};
4
5  int access(int group, int req) {
▶      for (int i = 0; i < 3; i++)
▶          if (acl[i].group == group) {
▶              int h = acl[i].rights;
▶              return (h & req) == req;
10         }
11     return 0;    /* not listed: denied */
12 }
```

request: G1 · W (2) **deny**

acl[] (stored with file A)



access(G1, W): the first entry matches, h = R. 4 & 2 = 0 ≠ 2: **denied**. The decision is made at the first matching entry; the list is not scanned further.

Step 4 / 7

VARIABLES

`group = G1` `req = W` `i = 0` `h = R--`

OUTPUT

```
G2 W: allow
G1 W: deny
```

Access Control List: Code Trace

```
1  /* ACL of file A: R=4 W=2 X=1 */
2  struct entry { int group, rights; } acl[] =
3      {{G1, 4}, {G2, 2}, {ADM, 7}};
4
5  int access(int group, int req) {
▶      for (int i = 0; i < 3; i++)
▶          if (acl[i].group == group) {
8              int h = acl[i].rights;
9              return (h & req) == req;
10         }
11     return 0;    /* not listed: denied */
12 }
```

request: G4 · R (4)

acl[] (stored with file A)



access(G4, R): G4 is in no entry, so the kernel scans the whole list. That is the cost of every access with long ACLs.

Step 5 / 7

VARIABLES

`group = G4` `req = R` `i = 2`

OUTPUT

```
G2 W: allow
G1 W: deny
```

Access Control List: Code Trace

```
1  /* ACL of file A: R=4 W=2 X=1 */
2  struct entry { int group, rights; } acl[] =
3      {{G1, 4}, {G2, 2}, {ADM, 7}};
4
5  int access(int group, int req) {
6      for (int i = 0; i < 3; i++)
7          if (acl[i].group == group) {
8              int h = acl[i].rights;
9              return (h & req) == req;
10         }
▶      return 0;    /* not listed: denied */
12 }
```

request: G4 · R (4) **deny**

acl[] (stored with file A)



The loop ends with no match: it returns 0. **Fail-safe default**: anything not explicitly allowed is forbidden.

Step 6 / 7

VARIABLES

`group = G4` `req = R` `i = 3`

OUTPUT

```
G2 W: allow
G1 W: deny
G4 R: deny
```

Access Control List: Code Trace

```
1  /* ACL of file A: R=4 W=2 X=1 */
2  struct entry { int group, rights; } acl[] =
3      {{G1, 4}, {G2, 2}, {ADM, 7}};
4
5  int access(int group, int req) {
▶      for (int i = 0; i < 3; i++)
▶          if (acl[i].group == group) {
▶              int h = acl[i].rights;
▶              return (h & req) == req;
10         }
11     return 0;    /* not listed: denied */
12 }
```

request: ADM · R|X (5) **allow**

acl[] (stored with file A)



access(ADM, R|X): matches at i = 2, h = 7. 7 & 5 = 5: **allowed**. Several rights are checked in one request; access is granted only if all are present.

Step 7 / 7

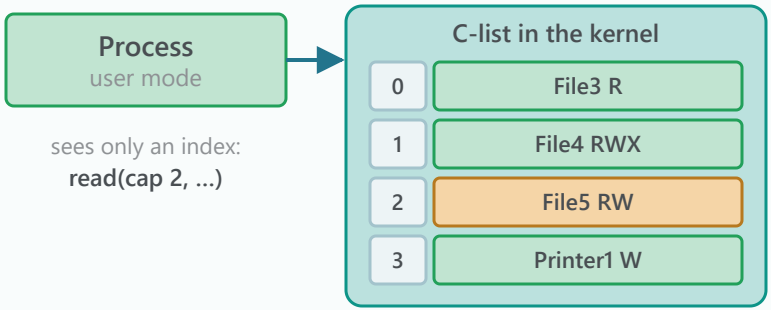
VARIABLES

`group = ADM` `req = RX` `i = 2` `h = RWX`

OUTPUT

```
G2 W: allow
G1 W: deny
G4 R: deny
ADM RX: allow
```

Capability Lists



Cryptographically protected capability

Server	Object	Rights	f(Object, Rights, Secret)

Cannot be forged even if users hold it: f is one-way (e.g. HMAC); only the server knows the secret (Amoeba).

Copy capability
A new capability for the same object (delegation).

Copy object
A new object and a new capability for it.

Remove capability
The entry is deleted from the list; the object stays.

Destroy object
Object and capability are deleted for good.

Capabilities must be unforgeable: kept in the kernel (accessed by index), protected by tagged hardware (CHERI), or cryptographically signed. Everyday examples: UNIX **file descriptors** (no permission recheck after open), FreeBSD Capsicum, seL4, Fuchsia.

Least Privilege and setuid

```
▶ /* /usr/bin/passwd  -rwsr-xr-x root */
2 int main(void) {
3     uid_t me = getuid();
4     int fd = open("/etc/shadow", O_RDWR);
5     update_own_entry(fd, me);
6     close(fd);
7     setresuid(me, me, me);
8     /* ... rest of the work ... */
9     fd = open("/etc/shadow", O_RDWR);
10    if (fd < 0) perror("open");
11    else close(fd);
12    return 0;
13 }
```



shell: ali, unprivileged

Ali (uid 1000) runs `passwd`. The file is owned by root and its **setuid** bit (s) is set.

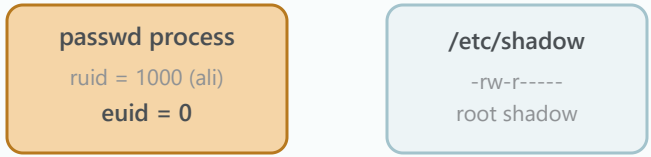
Step 1 / 6

VARIABLES

`ruid = 1000` `euid = 1000` `fd = –`

Least Privilege and setuid

```
1  /* /usr/bin/passwd  -rwsr-xr-x root */
2  int main(void) {
3      uid_t me = getuid();
4      int fd = open("/etc/shadow", O_RDWR);
5      update_own_entry(fd, me);
6      close(fd);
7      setresuid(me, me, me);
8      /* ... rest of the work ... */
9      fd = open("/etc/shadow", O_RDWR);
10     if (fd < 0) perror("open");
11     else close(fd);
12     return 0;
13 }
```



execve: euid ← file owner

During `execve` the effective ID becomes the file owner: **euid = 0**. The real ID (ruid) is still Ali.

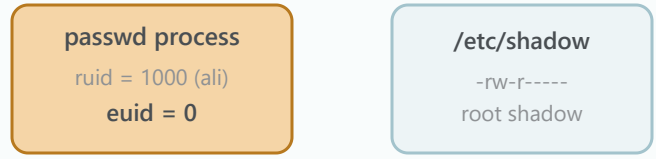
Step 2 / 6

VARIABLES

`ruid = 1000` `euid = 0` `fd = –`

Least Privilege and setuid

```
1  /* /usr/bin/passwd  -rwsr-xr-x root */
2  int main(void) {
▶      uid_t me = getuid();
4      int fd = open("/etc/shadow", O_RDWR);
5      update_own_entry(fd, me);
6      close(fd);
7      setresuid(me, me, me);
8      /* ... rest of the work ... */
9      fd = open("/etc/shadow", O_RDWR);
10     if (fd < 0) perror("open");
11     else close(fd);
12     return 0;
13 }
```



getuid() → 1000

`getuid()` returns the real ID: the program knows on whose behalf it runs and changes only that user's line.

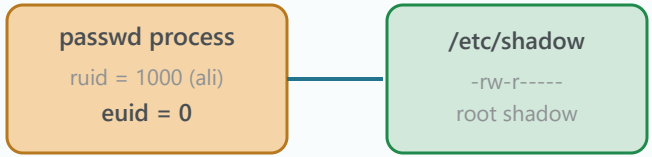
Step 3 / 6

VARIABLES

`ruid = 1000` `euid = 0` `me = 1000` `fd = –`

Least Privilege and setuid

```
1  /* /usr/bin/passwd  -rwsr-xr-x root */
2  int main(void) {
3      uid_t me = getuid();
▶      int fd = open("/etc/shadow", O_RDWR);
▶      update_own_entry(fd, me);
6      close(fd);
7      setresuid(me, me, me);
8      /* ... rest of the work ... */
9      fd = open("/etc/shadow", O_RDWR);
10     if (fd < 0) perror("open");
11     else close(fd);
12     return 0;
13 }
```



allowed (euid 0)

The kernel checks access against the **effective** ID: euid 0, so the file opens.

Step 4 / 6

VARIABLES

`ruid = 1000` `euid = 0` `me = 1000` `fd = 3`

Least Privilege and setuid

```
1  /* /usr/bin/passwd  -rwsr-xr-x root */
2  int main(void) {
3      uid_t me = getuid();
4      int fd = open("/etc/shadow", O_RDWR);
5      update_own_entry(fd, me);
▶      close(fd);
▶      setresuid(me, me, me);
8      /* ... rest of the work ... */
9      fd = open("/etc/shadow", O_RDWR);
10     if (fd < 0) perror("open");
11     else close(fd);
12     return 0;
13 }
```



privilege dropped for good

As soon as the job is done, `setresuid` sets the real, effective and saved IDs all to 1000: **least privilege**, with no way back.

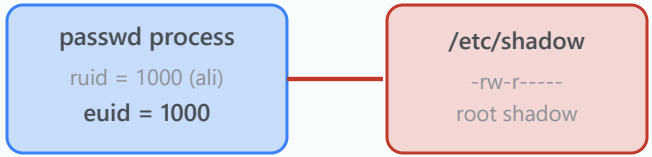
Step 5 / 6

VARIABLES

`ruid = 1000` `euid = 1000` `me = 1000` `fd = –`

Least Privilege and setuid

```
1  /* /usr/bin/passwd  -rwsr-xr-x root */
2  int main(void) {
3      uid_t me = getuid();
4      int fd = open("/etc/shadow", O_RDWR);
5      update_own_entry(fd, me);
6      close(fd);
7      setresuid(me, me, me);
8      /* ... rest of the work ... */
▶      fd = open("/etc/shadow", O_RDWR);
▶      if (fd < 0) perror("open");
11     else close(fd);
12     return 0;
13 }
```



EACCES: permission denied

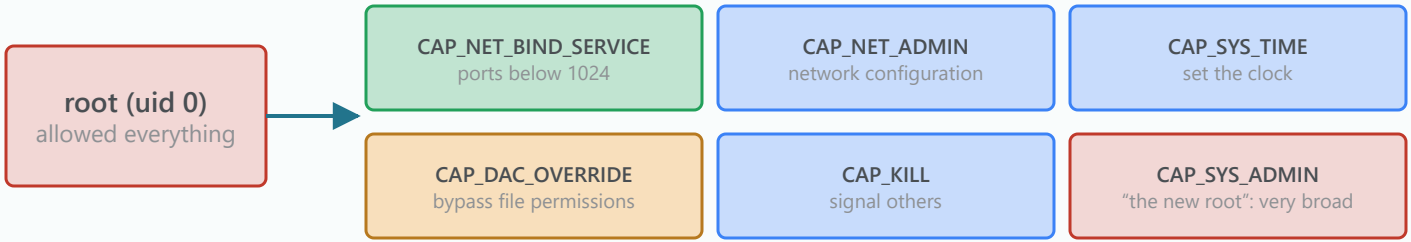
Even if a bug in the rest of the program is exploited, the attacker is no longer root. Setuid-root programs are the classic target of privilege-escalation attacks.

Step 6 / 6

VARIABLES

`ruid = 1000` `euid = 1000` `me = 1000` `fd = -1`

Linux Capabilities and Design Principles



≈ 41 capabilities; each thread has permitted, effective, inheritable and bounding sets

```
$ sudo setcap cap_net_bind_service=+ep /usr/local/bin/websrv
$ getcap /usr/local/bin/websrv
/usr/local/bin/websrv cap_net_bind_service=ep     # may bind port 80, no other root rights
```

- **Least privilege** (Saltzer & Schroeder, 1975): every component should run with the smallest set of rights that gets its job done, so a compromise does limited damage.
- Other principles: fail-safe defaults (anything not explicitly allowed is forbidden), **complete mediation** (every access is checked), open design, separation of privilege, psychological acceptability.
- In practice: privilege separation (OpenSSH: a small root process + an unprivileged child), `systemd` `CapabilityBoundingSet=`, `NoNewPrivileges=`.

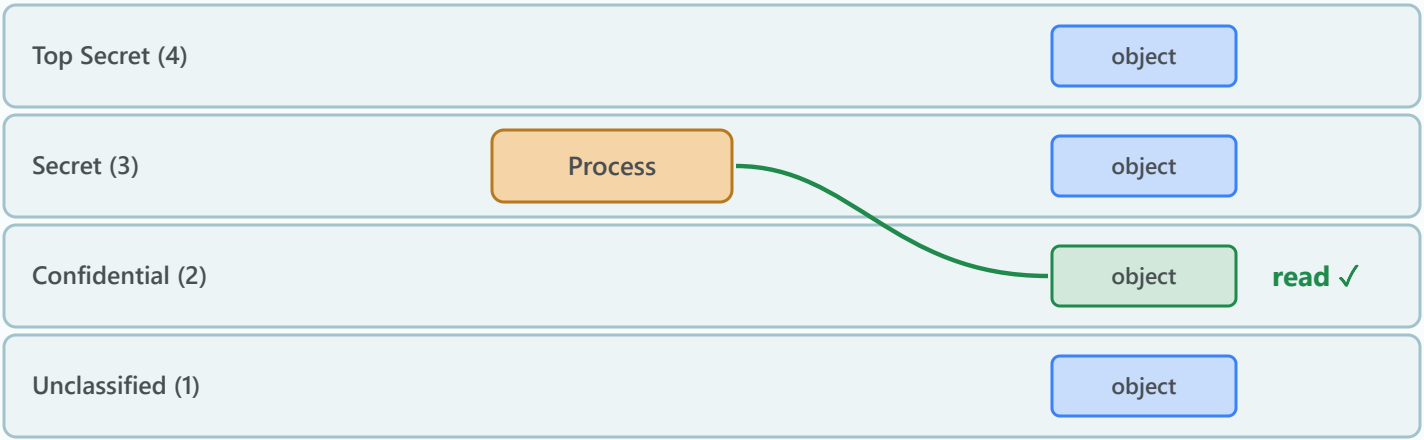
Bell-LaPadula: No Read Up, No Write Down



The process runs with “Secret” clearance

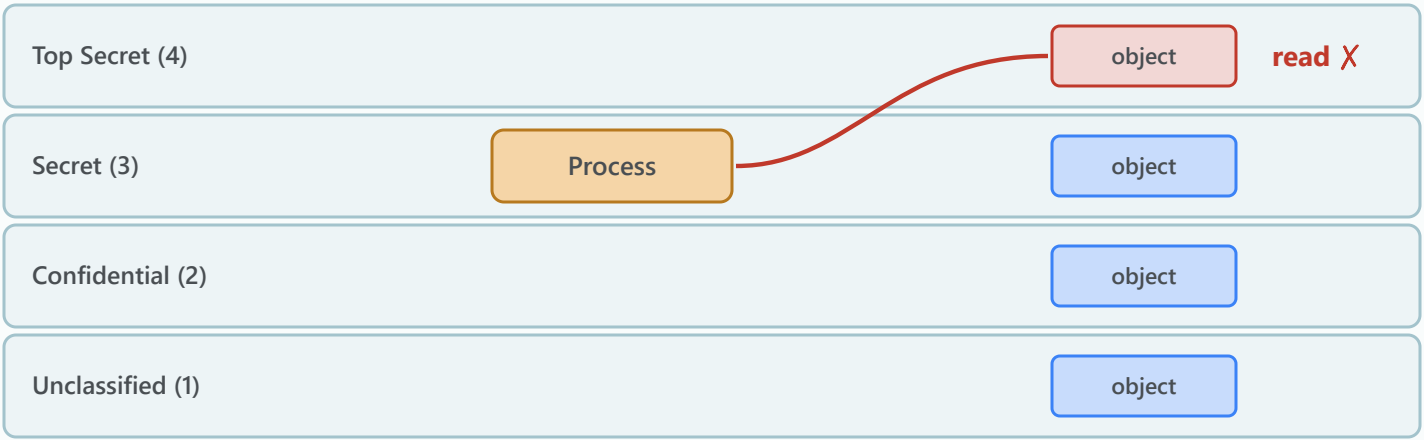
Military classification levels; every process and object has a level. The goal: information must not **leak downward** (confidentiality).

Bell-LaPadula: No Read Up, No Write Down



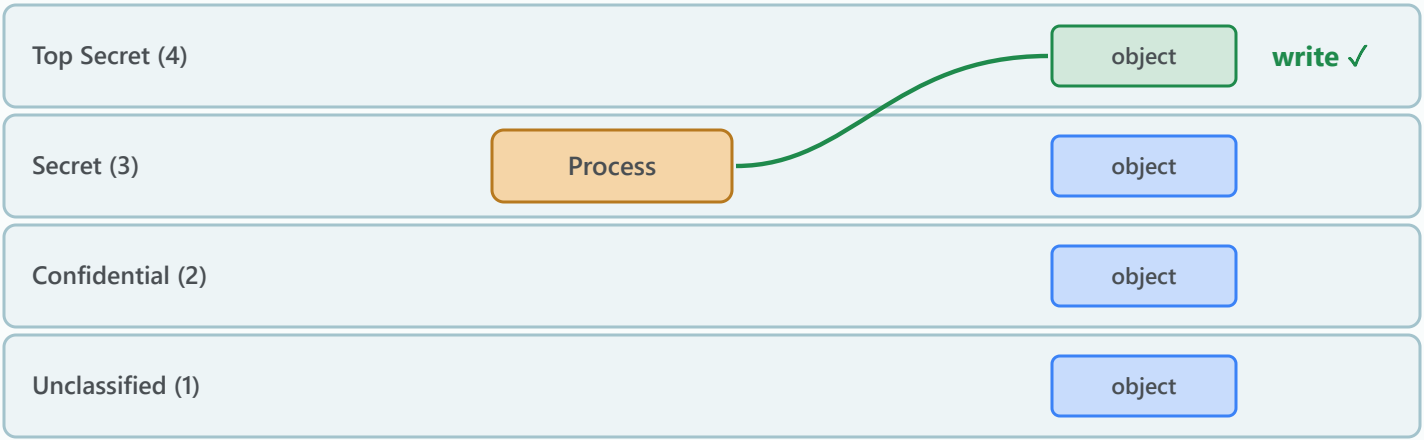
Simple security property: a process at level k may **read** only objects at levels ≤ k. Reading down is allowed.

Bell-LaPadula: No Read Up, No Write Down



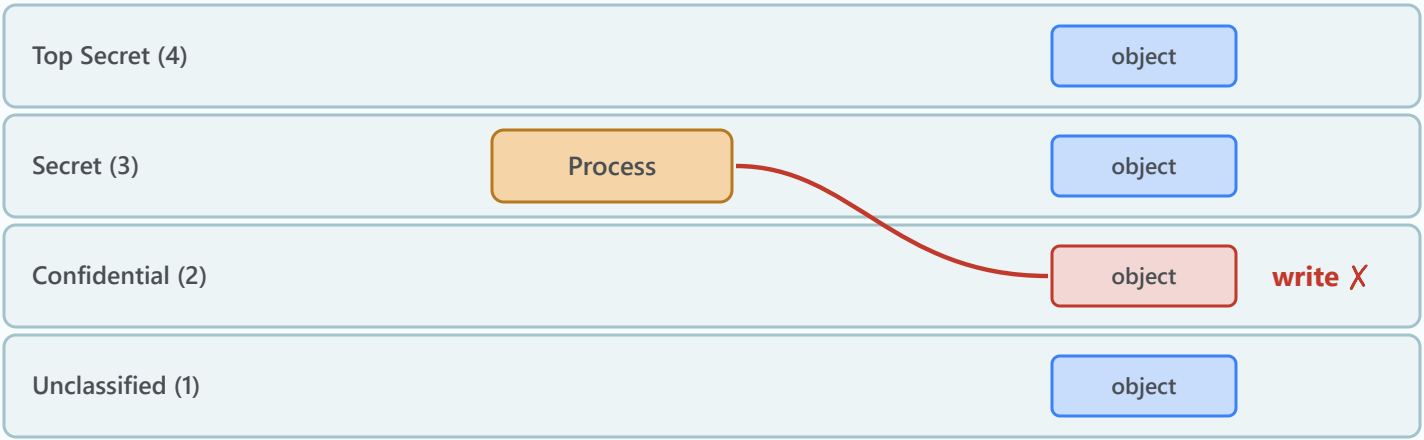
No read up: a process cleared for Secret cannot read a Top Secret document.

Bell-LaPadula: No Read Up, No Write Down



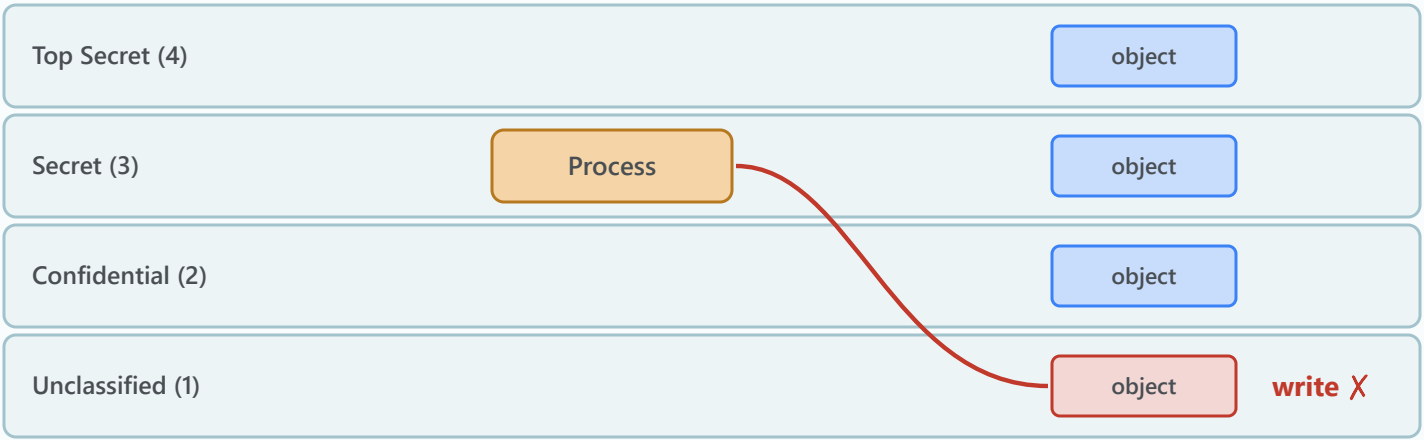
* **property**: a process at level k may **write** only at levels ≥ k. Writing up does not break secrecy (usually a “blind” append).

Bell-LaPadula: No Read Up, No Write Down



No write down: otherwise Secret information could leak by being copied into a Confidential file.

Bell-LaPadula: No Read Up, No Write Down



even a Trojan horse running inside a Secret process cannot write down

The payoff: even a **Trojan horse** the user runs cannot move information down, because the rule depends on the system, not on the user's good intentions (mandatory access control, MAC).

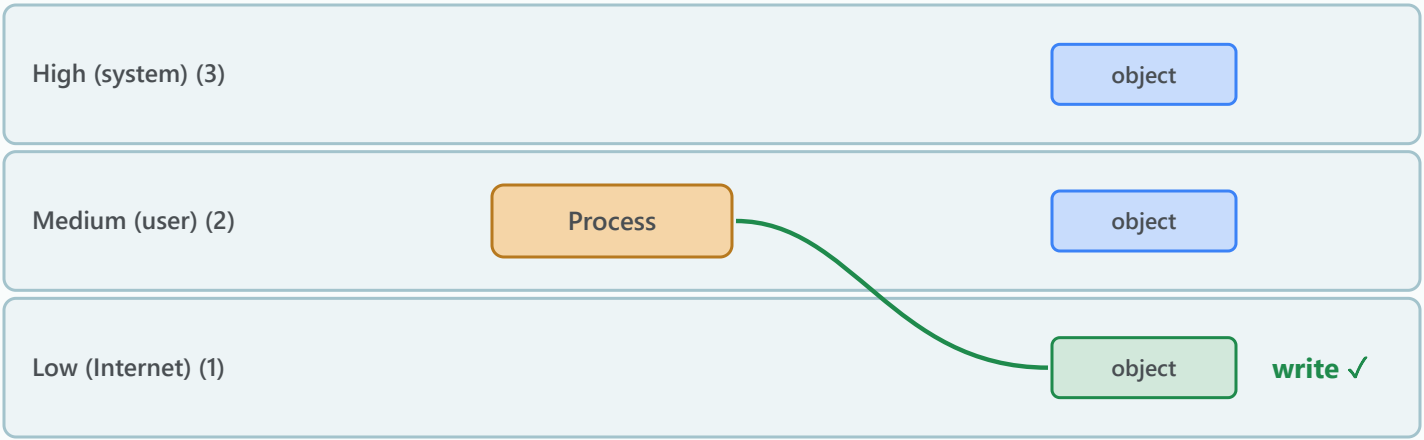
Biba: No Read Down, No Write Up



The process runs at medium integrity

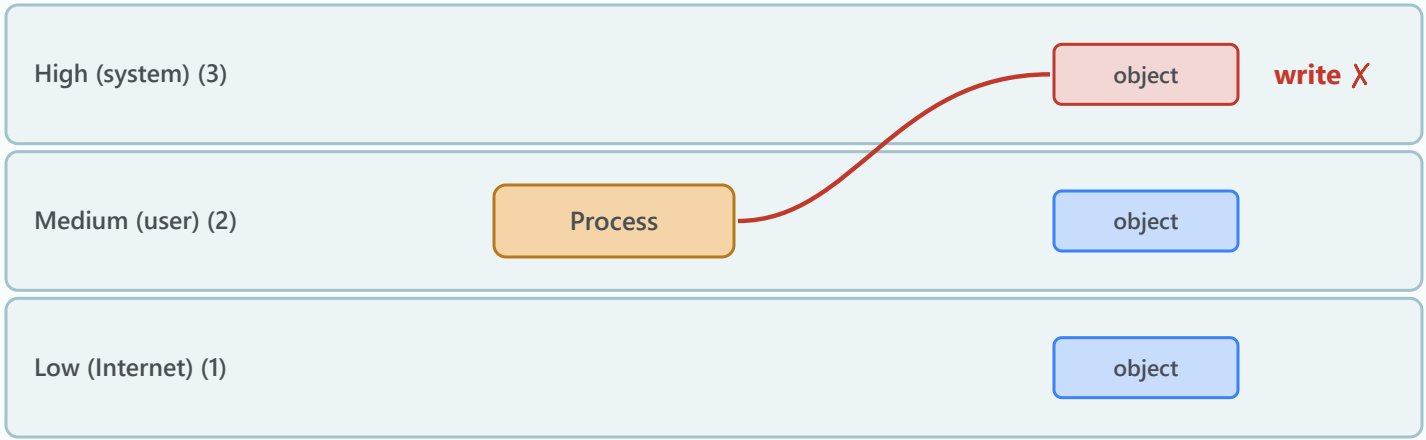
Biba (1977) mirrors BLP for **integrity**: levels express trustworthiness. The goal: untrusted data must not **contaminate upward**.

Biba: No Read Down, No Write Up



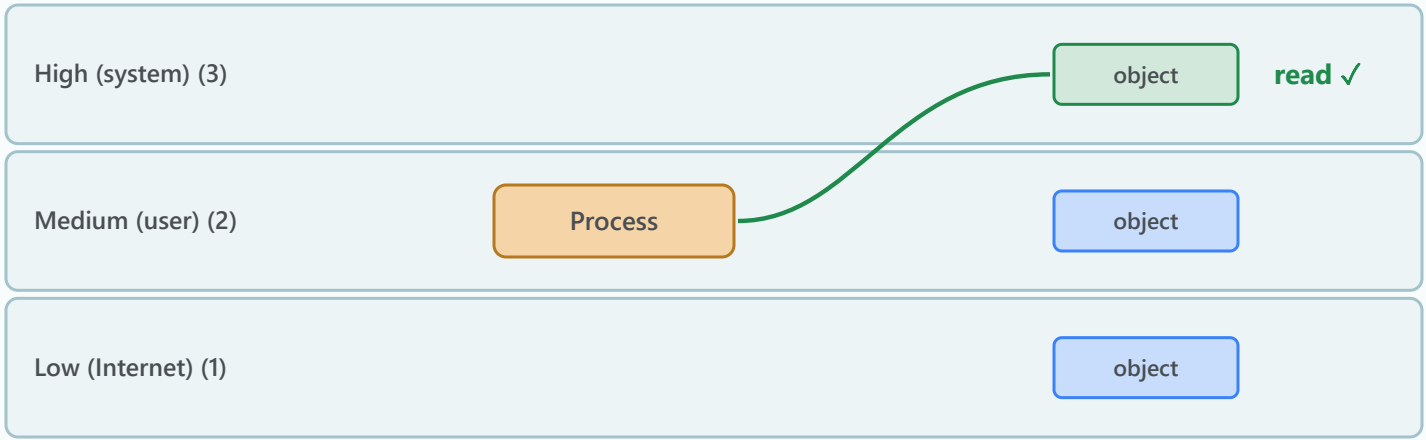
* **integrity property**: a process at level k may **write** only at levels ≤ k.

Biba: No Read Down, No Write Up



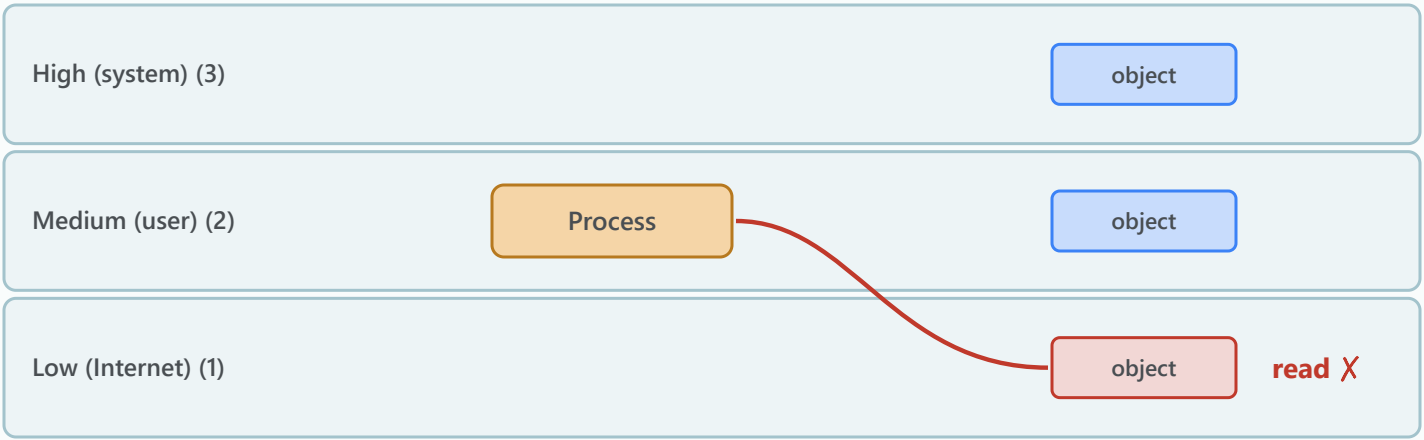
No write up: a user process cannot modify system files.

Biba: No Read Down, No Write Up



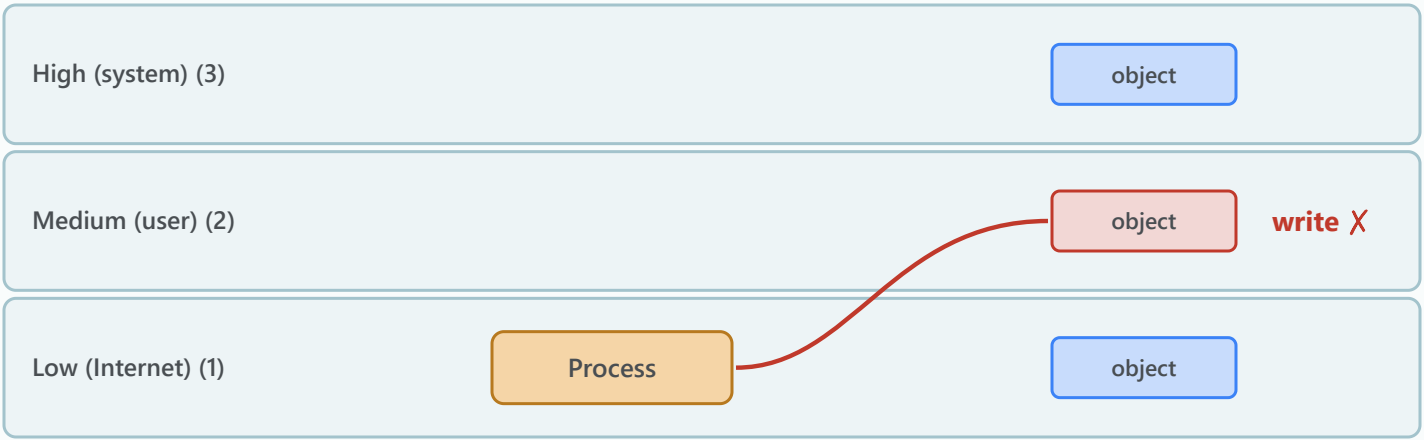
Simple integrity property: a process may **read** only from levels ≥ k; reading trusted data is fine.

Biba: No Read Down, No Write Up



No read down: a process that read untrusted data from the Internet and acted on it would be “contaminated”.

Biba: No Read Down, No Write Up

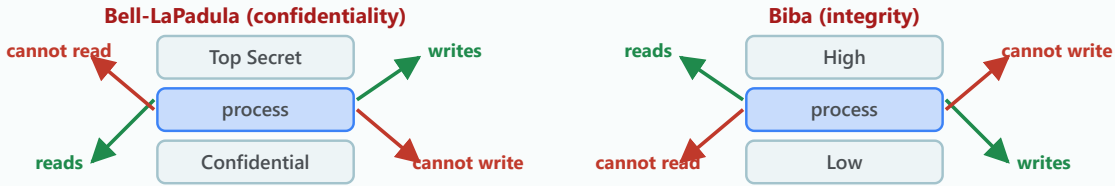


a low-integrity browser process cannot write to user files

A real example: Windows **Mandatory Integrity Control** (MIC). Browser and protected-mode processes run at Low and cannot write to Medium-level documents (no write up).

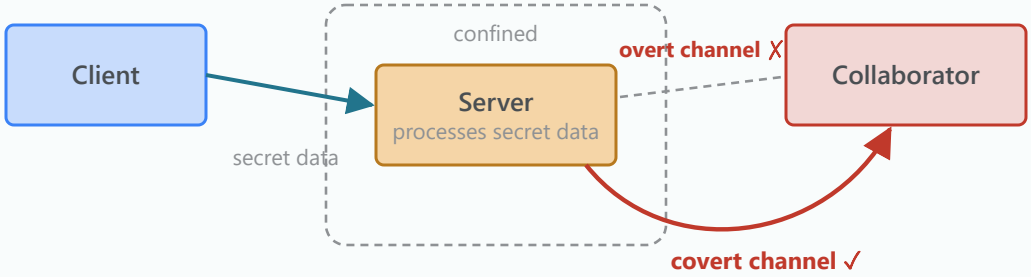
Comparing Bell-LaPadula and Biba

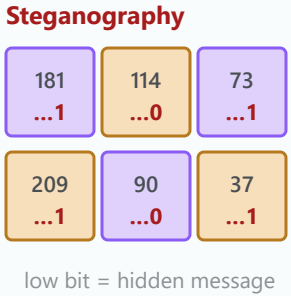
	Bell-LaPadula (1973)	Biba (1977)
Protects	Confidentiality	Integrity
Read	Down allowed, **no up**	Up allowed, **no down**
Write	Up allowed, **no down**	Down allowed, **no up**
Prevents	Information leaking down	Tainted data spreading up
Today	SELinux MLS, military systems	Windows MIC, FreeBSD/Linux integrity policies



- Used together, the two models leave a process talking almost only to its own level: one protects confidentiality, the other integrity, and they conflict.
- Naming: in BLP the **simple security property** forbids reading up and the *** property** forbids writing down; in Biba (1977) the **simple integrity property** forbids reading down and the *** integrity property** forbids writing up.
- Neither model stops **covert channels**: the rules govern only explicit reads and writes.

Covert Channels and Steganography





Storage channel

A change in some shared state: a file exists or not, locked or unlocked, disk fullness.

Timing channel

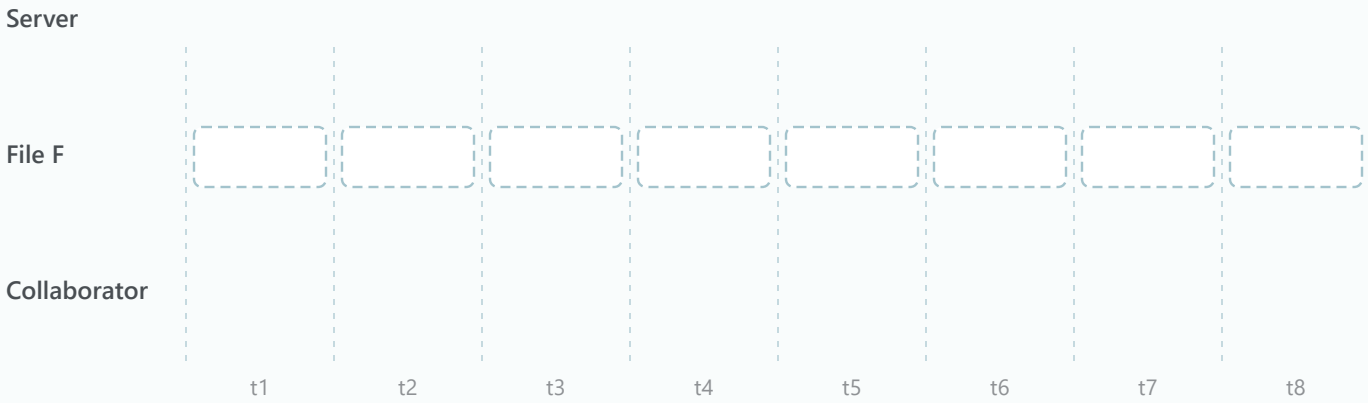
The timing of events: heavy CPU use = 1, idling = 0; cache latency.

Resource channel

Use of a shared resource: page-fault rate, network traffic volume.

Covert channel: an information path the designer never intended for communication, which bypasses the security policy. Lampson's confinement problem (1973): closing them all is practically impossible, though their bandwidth can be reduced. Steganography: text (say, Shakespeare's plays) can be hidden in the low bits of the pixels of a photo of three zebras; the image looks unchanged.

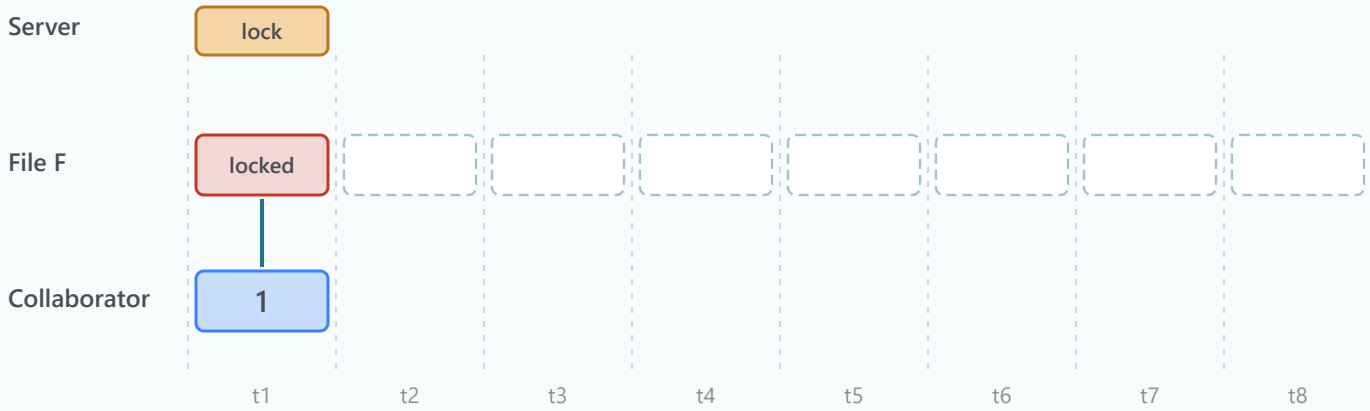
A Covert Channel Using File Locks



Bits received: —

The server and collaborator agree in advance: in each clock interval, F **locked means 1**, **unlocked means 0**. No read or write takes place; F's contents never change.

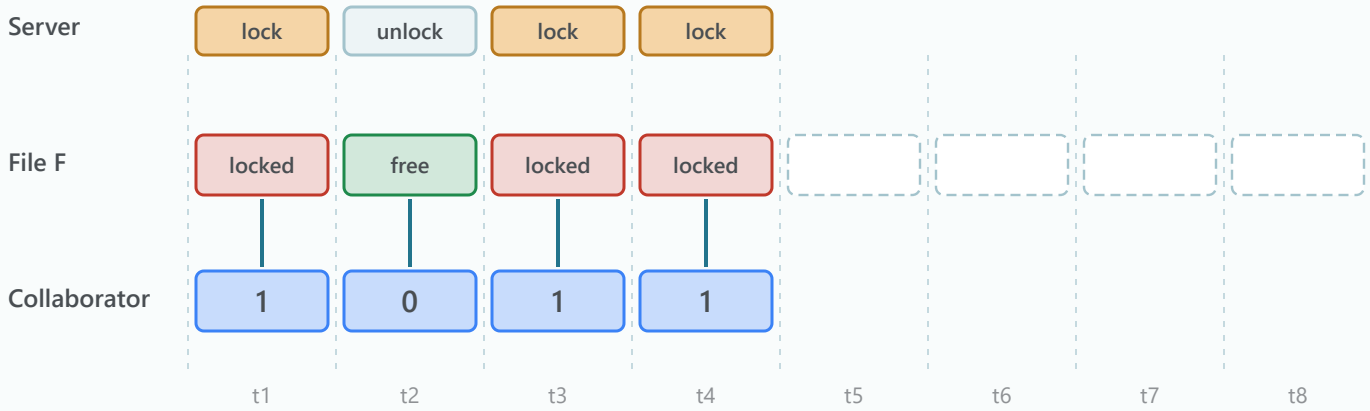
A Covert Channel Using File Locks



Bits received: 1

The server locks F at t1; the collaborator tries to lock it, fails → 1.

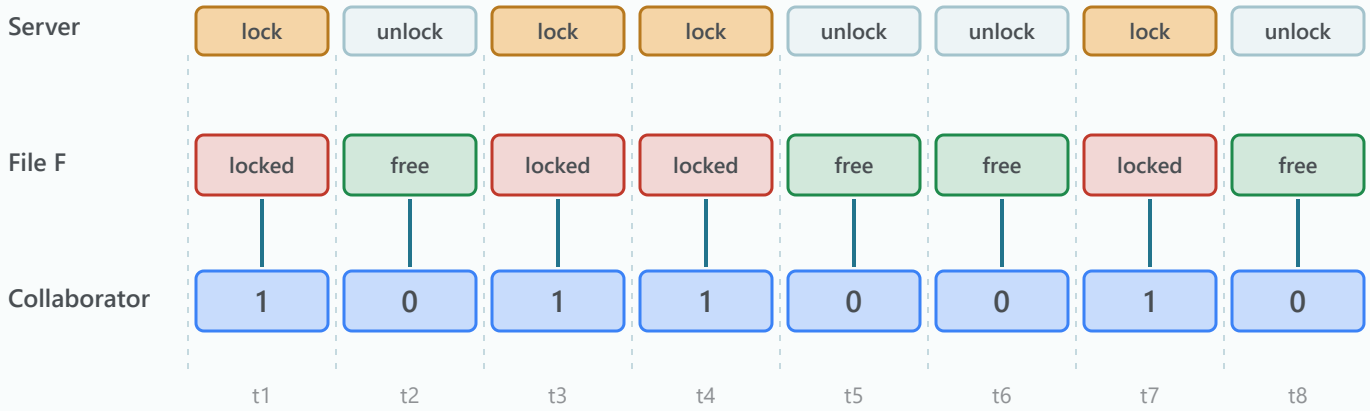
A Covert Channel Using File Locks



Bits received: 1011

The first four bits the same way: 1 0 1 1.

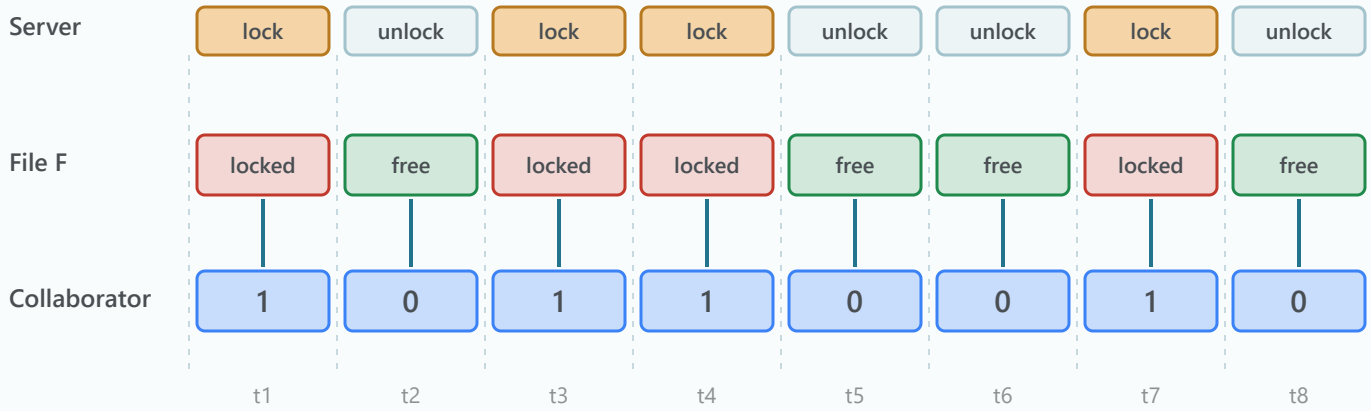
A Covert Channel Using File Locks



Bits received: 10110010

One byte in eight intervals: 10110010 (0xB2). At 100 intervals per second that is ≈ 100 bit/s; a password or key leaks in seconds.

A Covert Channel Using File Locks



Bits received: 10110010

Countermeasures: hide lock state from others, add noise and delay, partition resources by level. The bandwidth drops but never reaches zero.

Authentication Factors

Something you know

Password, PIN, security question. Cheap, but can be stolen, guessed and reused.

Something you have

Phone, smart card, FIDO2 key, TOTP generator. Can be stolen, but not copied remotely.

Something you are

Fingerprint, face, iris. Easy, but unchangeable; once leaked, burned for good.

Where you are / how you act

Location, IP, typing rhythm, device fingerprint. Usually an extra risk signal.

Multifactor authentication (MFA): **at least two** factors from different categories. “Password + security question” is not MFA (both are knowledge). MFA is the single most effective defense against stolen passwords.

Challenge-response authentication asks the user preregistered questions; but answers like “mother's maiden name” or “first pet” can be found on social media, so it is considered weak today.

Password Storage: Salt and Slow Hashes

```
# Registration / password change
salt = os.urandom(16)
h = argon2(password, salt)
store(user, salt, h)

# Login attempt
salt, h = lookup(user)
g = argon2(attempt, salt)
if constant_eq(g, h): ok()
```



Passwords are **never** stored in the clear. Two of the three users picked the same password (people often do).

Step 1 / 4

VARIABLES

—

Plain SHA-256/MD5 is unfit for passwords: they are fast, exactly what you do not want for passwords. 1970s UNIX used a 2-character salt + slowed-down DES (crypt); the idea is the same, the parameters far stronger today. After a breach, passwords must still be reset.

Password Storage: Salt and Slow Hashes

```
1 # Registration / password change
▶ salt = os.urandom(16)
3 h = argon2(password, salt)
4 store(user, salt, h)
5
6 # Login attempt
7 salt, h = lookup(user)
8 g = argon2(attempt, salt)
9 if constant_eq(g, h): ok()
```



A plain (unsalted) hash is not enough: the same password gives the same hash; the attacker spots repeats and cracks them all at once with a **rainbow table**.

Step 2 / 4

VARIABLES

—

Plain SHA-256/MD5 is unfit for passwords: they are fast, exactly what you do not want for passwords. 1970s UNIX used a 2-character salt + slowed-down DES (crypt); the idea is the same, the parameters far stronger today. After a breach, passwords must still be reset.

Password Storage: Salt and Slow Hashes

```
1 # Registration / password change
▶ salt = os.urandom(16)
▶ h = argon2(password, salt)
4 store(user, salt, h)
5
6 # Login attempt
7 salt, h = lookup(user)
8 g = argon2(attempt, salt)
9 if constant_eq(g, h): ok()
```



Salt: a random, nonsecret value per user is added to the password. The same password is now a different input; precomputation is useless.

Step 3 / 4

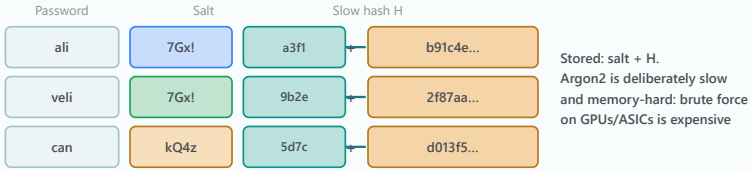
VARIABLES

—

Plain SHA-256/MD5 is unfit for passwords: they are fast, exactly what you do not want for passwords. 1970s UNIX used a 2-character salt + slowed-down DES (crypt); the idea is the same, the parameters far stronger today. After a breach, passwords must still be reset.

Password Storage: Salt and Slow Hashes

```
1 # Registration / password change
2 salt = os.urandom(16)
3 h = argon2(password, salt)
4 store(user, salt, h)
5
6 # Login attempt
7 salt, h = lookup(user)
▶ g = argon2(attempt, salt)
▶ if constant_eq(g, h): ok()
```



A **slow, memory-hard hash**: Argon2id (or scrypt, bcrypt). One check takes milliseconds; billions of guesses take years. A **constant-time** comparison during verification prevents timing leaks.

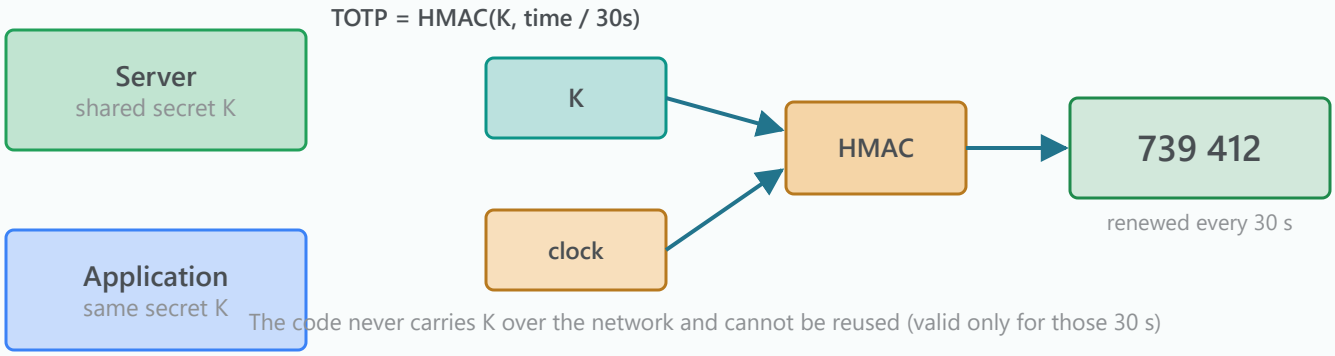
Step 4 / 4

VARIABLES

—

Plain SHA-256/MD5 is unfit for passwords: they are fast, exactly what you do not want for passwords. 1970s UNIX used a 2-character salt + slowed-down DES (crypt); the idea is the same, the parameters far stronger today. After a breach, passwords must still be reset.

MFA and Time-Based One-Time Passwords (TOTP)



SMS codes

Common but weak: they can be hijacked by SIM swapping and SS7 attacks. Better than nothing.

TOTP (authenticator app)

RFC 6238; works offline. Phishable: the user may type the code into a fake site.

Push approval

Easy, but open to “MFA fatigue” attacks. Number matching helps.

Generating a TOTP Code: Code Trace

```
▶ uint32_t totp(uint8_t *K, uint64_t t) {
2     uint64_t T = t / 30;     /* 30 s */
3     uint8_t h[20];
4     hmac_sha1(K, T, h);
5     int o = h[19] & 0x0F;    /* offset */
6     uint32_t v = (h[o] & 0x7F) << 24
7                | h[o+1] << 16
8                | h[o+2] << 8
9                | h[o+3];
10    return v % 1000000;      /* 6 digits */
11 }
```

h[0..19]: not computed yet

K: 20-byte secret shared by QR code at enrollment

t = 59: Unix time (seconds)

The phone and the server know the same secret **K** and the same time. RFC 6238's test key ("12345678901234567890") at t = 59 s.

Step 1 / 7

VARIABLES

`t = 59` `K = 20 bytes`

OUTPUT

Generating a TOTP Code: Code Trace

```
1  uint32_t totp(uint8_t *K, uint64_t t) {
▶      uint64_t T = t / 30;     /* 30 s */
3      uint8_t h[20];
4      hmac_sha1(K, T, h);
5      int o = h[19] & 0x0F;    /* offset */
6      uint32_t v = (h[o] & 0x7F) << 24
7                 | h[o+1] << 16
8                 | h[o+2] << 8
9                 | h[o+3];
10     return v % 1000000;      /* 6 digits */
11 }
```

h[0..19]: not computed yet

T = 59 / 30 = 1

0–29 s → 0, 30–59 s → 1, 60–89 s → 2

Time is cut into 30-second steps: T = **1**. Every moment within a step gives the same code.

Step 2 / 7

VARIABLES

`t = 59` `T = 1`

OUTPUT

Generating a TOTP Code: Code Trace

```
1  uint32_t totp(uint8_t *K, uint64_t t) {
2      uint64_t T = t / 30;     /* 30 s */
▶      uint8_t h[20];
▶      hmac_sha1(K, T, h);
5      int o = h[19] & 0x0F;    /* offset */
6      uint32_t v = (h[o] & 0x7F) << 24
7                 | h[o+1] << 16
8                 | h[o+2] << 8
9                 | h[o+3];
10     return v % 1000000;      /* 6 digits */
11 }
```

0	1	2	3	4	5	6	7	8	9
75	a4	8a	19	d4	cb	e1	00	64	4e
10	11	12	13	14	15	16	17	18	19
8a	c1	39	7e	ea	74	7a	2d	33	ab

h = HMAC-SHA1(K, T)

20 bytes; no one without K can produce it

T, as an 8-byte number, goes through HMAC-SHA1 with K: a 20-byte digest. Incrementing T by one gives a completely different digest.

Step 3 / 7

VARIABLES

`T = 1` `h[19] = 0xab`

OUTPUT

Generating a TOTP Code: Code Trace

```
1  uint32_t totp(uint8_t *K, uint64_t t) {
2      uint64_t T = t / 30;     /* 30 s */
3      uint8_t h[20];
4      hmac_sha1(K, T, h);
▶      int o = h[19] & 0x0F;    /* offset */
6      uint32_t v = (h[o] & 0x7F) << 24
7                 | h[o+1] << 16
8                 | h[o+2] << 8
9                 | h[o+3];
10     return v % 1000000;      /* 6 digits */
11 }
```

0	1	2	3	4	5	6	7	8	9
75	a4	8a	19	d4	cb	e1	00	64	4e
10	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
8a	c1	39	7e	ea	74	7a	2d	33	**ab**

h[19] = 0xab → low 4 bits = 11

offset o = 11

Dynamic truncation: the low nibble of the last byte (0xab & 0x0F) picks the offset: **o = 11**.

Step 4 / 7

VARIABLES

`T = 1` `h[19] = 0xab` `o = 11`

OUTPUT

Generating a TOTP Code: Code Trace

```
1  uint32_t totp(uint8_t *K, uint64_t t) {
2      uint64_t T = t / 30;     /* 30 s */
3      uint8_t h[20];
4      hmac_sha1(K, T, h);
5      int o = h[19] & 0x0F;    /* offset */
▶      uint32_t v = (h[o] & 0x7F) << 24
▶                 | h[o+1] << 16
▶                 | h[o+2] << 8
▶                 | h[o+3];
10     return v % 1000000;      /* 6 digits */
11 }
```

0	1	2	3	4	5	6	7	8	9
75	a4	8a	19	d4	cb	e1	00	64	4e
10	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
8a	**c1**	**39**	**7e**	**ea**	74	7a	2d	33	**ab**

h[11..14] = c1 39 7e ea

top bit cleared → v = 1094287082

The four bytes h[11]...h[14] form a 32-bit number; clearing the top bit (& 0x7F) avoids sign issues: v = **1094287082**.

Step 5 / 7

VARIABLES

`T = 1` `o = 11` `v = 1094287082`

OUTPUT

Generating a TOTP Code: Code Trace

```
1  uint32_t totp(uint8_t *K, uint64_t t) {
2      uint64_t T = t / 30;     /* 30 s */
3      uint8_t h[20];
4      hmac_sha1(K, T, h);
5      int o = h[19] & 0x0F;    /* offset */
6      uint32_t v = (h[o] & 0x7F) << 24
7                 | h[o+1] << 16
8                 | h[o+2] << 8
9                 | h[o+3];
▶      return v % 1000000;      /* 6 digits */
11 }
```

0	1	2	3	4	5	6	7	8	9
75	a4	8a	19	d4	cb	e1	00	64	4e
10	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
8a	c1	39	7e	ea	74	7a	2d	33	ab

1094287082 mod 1000000

code = 287082

The last 6 digits: **287082**. The phone shows this; the server does the same computation and compares.

Step 6 / 7

VARIABLES

`T = 1` `v = 1094287082` `code = 287082`

OUTPUT

```
t=59: 287082
```

Generating a TOTP Code: Code Trace

```
1  uint32_t totp(uint8_t *K, uint64_t t) {
▶      uint64_t T = t / 30;     /* 30 s */
3      uint8_t h[20];
▶      hmac_sha1(K, T, h);
▶      int o = h[19] & 0x0F;    /* offset */
6      uint32_t v = (h[o] & 0x7F) << 24
7                 | h[o+1] << 16
8                 | h[o+2] << 8
9                 | h[o+3];
▶      return v % 1000000;      /* 6 digits */
11 }
```

0	1	2	3	4	5	6	7	8	9
0b	ac	b7	fa	08	2f	ef	30	78	22
10	11	12	13	14	15	16	17	18	19
11	93	8b	c1	c5	e7	04	16	ff	44

t = 75 → T = 2, o = 4

code = 359152

30 seconds later T = 2: the digest and offset change and the code becomes **359152**. A stolen old code goes stale within seconds; servers usually accept T − 1 … T + 1 to allow for clock drift.

Step 7 / 7

VARIABLES

`t = 75` `T = 2` `o = 4` `code = 359152`

OUTPUT

```
t=59: 287082
t=75: 359152
```

AUTHENTICATION

FIDO2, WebAuthn and Passkeys



The signature includes the site's origin: a response given to a fake site is invalid on the real one → unphishable

- **FIDO2/WebAuthn** and **passkeys**: a public/private key pair per site. The private key **never leaves** the device (TPM, Secure Enclave, security key).
- The server stores only the **public key**: useless even if leaked. No shared secret, no password to reuse.
- The signature is bound to the page's **origin** → **phishing-resistant**. Passkeys can sync across platforms; they were designed to replace passwords.

Biometric Authentication

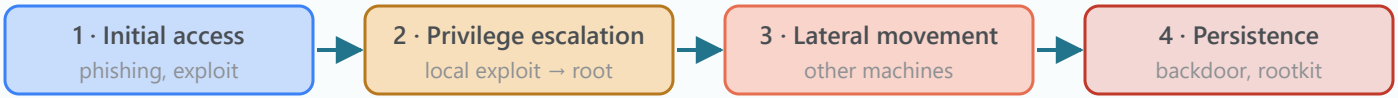
Metric	Description
False acceptance (FAR)	Rate of accepting a stranger — a security risk
False rejection (FRR)	Rate of rejecting the real user — a usability problem
Liveness test	Against photos, masks, fake fingerprints (liveness)
Irrevocability	A leaked fingerprint cannot be changed; a password can

- Biometrics are good as an **identity**, bad as a **secret**: faces and fingerprints are not secret.
- Proper use: locally on the device, as the factor that unlocks secure hardware (Face ID → Secure Enclave). The raw template must never leave the device.
- Not suitable alone for remote authentication; usually combined with “something you have”.

Early devices that measured finger length are historical; today's systems use fingerprint ridge details (minutiae), 3D faces and iris patterns, and hardware-backed liveness testing is a must.

How Do Attackers Get In?

Route	Description	Countermeasure
Phishing	Harvesting passwords with a fake login page	FIDO2, user training
Credential stuffing	Trying leaked passwords on other sites	Unique passwords, MFA
Brute force / dictionary	Guessing weak passwords	Slow hashes, rate limits, lockout
Unpatched vulnerability	Using a known flaw	Fast patch management
Social engineering	Fooling the help desk	Process and verification
Default/weak credentials	admin/admin, open ports	Hardening

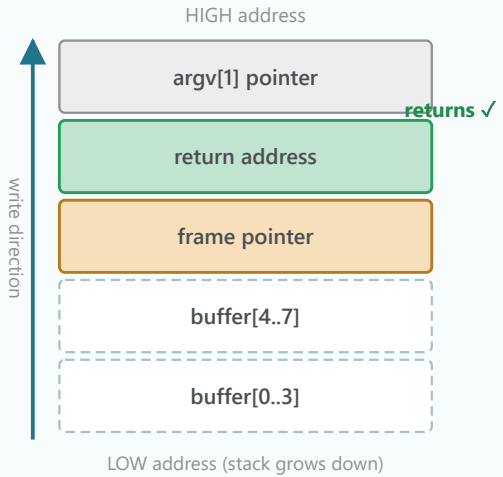


MFA and FIDO2 · patching and least privilege · network segmentation · EDR and secure boot: a defense at every link

An attack is usually a chain: initial access by phishing → **privilege escalation** through a local exploit → lateral movement → persistence (a backdoor). Defenses must be layered too; one broken link should not hand over the system.

Stack Buffer Overflow: The Frame Step by Step

```
1 /* small, deliberately buggy example */
▶ void A(char *in) {
▶     char buffer[8];
4     strcpy(buffer, in);   // NO bounds check
5     return;
6 }
7 A(argv[1]);              // in: user input
```



Each function call opens a **frame** on the stack: local variables, the saved frame pointer and the **return address** (where the CPU goes when the function ends).

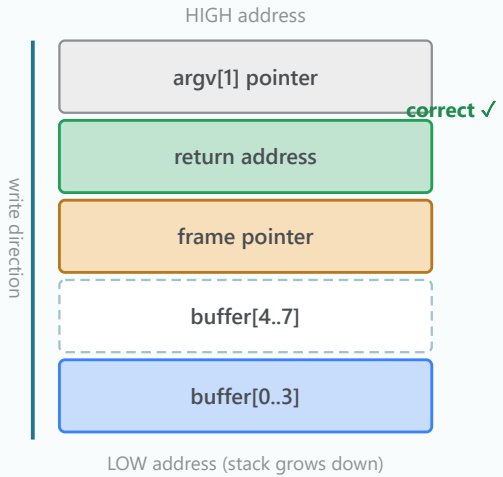
Step 1 / 4

VARIABLES

—

Stack Buffer Overflow: The Frame Step by Step

```
1 /* small, deliberately buggy example */
2 void A(char *in) {
3     char buffer[8];
▶     strcpy(buffer, in);   // NO bounds check
5     return;
6 }
7 A(argv[1]);              // in: user input
```



As long as the input fits, all is well: `strcpy` writes at most eight bytes, terminating `\0` included, into `buffer`.

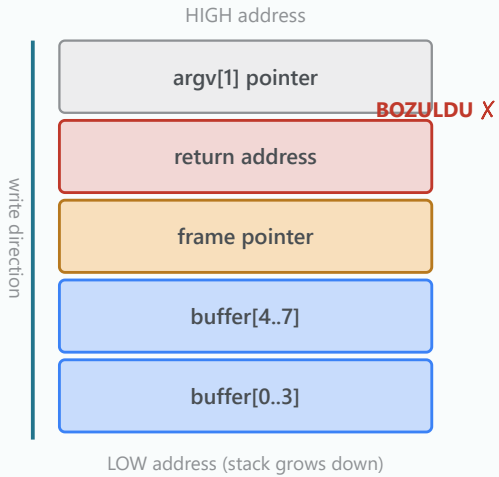
Step 2 / 4

VARIABLES

—

Stack Buffer Overflow: The Frame Step by Step

```
1 /* small, deliberately buggy example */
2 void A(char *in) {
3     char buffer[8];
▶     strcpy(buffer, in);   // NO bounds check
5     return;
6 }
7 A(argv[1]);              // in: user input
```



Overflow: `strcpy` does not check the length, so extra bytes spill into neighboring cells — most dangerously, over the **return address**.

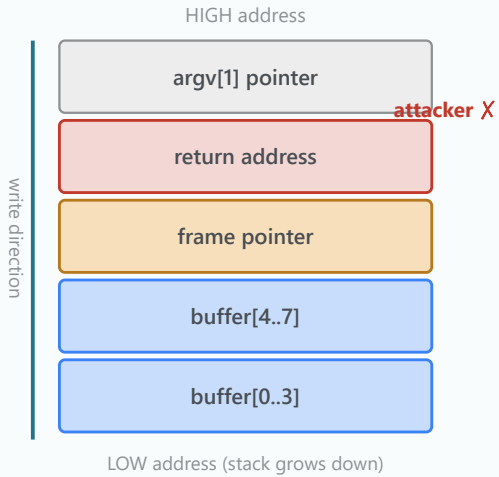
Step 3 / 4

VARIABLES

—

Stack Buffer Overflow: The Frame Step by Step

```
1 /* small, deliberately buggy example */
2 void A(char *in) {
3     char buffer[8];
4     strcpy(buffer, in);   // NO bounds check
▶     return;
6 }
7 A(argv[1]);              // in: user input
```



When the function `return` s, the CPU jumps to the corrupted address. If the attacker can choose it, they **hijack the flow of execution**. That is why untrusted input is never copied without a bounds check.

Step 4 / 4

VARIABLES

—

Defenses Against Buffer Overflows

Stack canary

A random value before the return address; if it is corrupted before the return, the program is stopped (SSP).

DEP / NX

Data pages are marked nonexecutable; code injected onto the stack cannot run.

ASLR

Stack, heap and libraries at random addresses, so the attacker cannot know the target address. Used with PIE.

CFI

Control-flow integrity: indirect jumps may reach only valid targets.

Shadow stack / CET

A hardware-protected copy of the return address: Intel CET, ARM GCS. (ARM PAC/BTI are separate protections.)

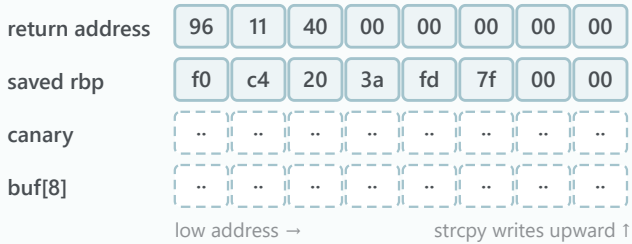
Memory-safe language

Rust, Go, Java, C#: bounds and lifetime checks in the compiler/runtime. The most fundamental fix.

ROP and return-to-libc: once NX blocks injected code, attackers run **existing** code pieces (`system()` in libc, or short “gadget” sequences) by chaining return addresses. CFI, shadow stacks and ASLR are combined to make these attacks hard — no single defense suffices.

Stack Canary: Code Trace

```
1 /* the compiler adds the canary lines */
▶ void copy_in(const char *input) {
3     long canary = __stack_chk_guard;
4     char buf[8];
5     strcpy(buf, input);
6     if (canary != __stack_chk_guard)
7         __stack_chk_fail();
8 }
9 /* input: "AAAAAAAABBBBBBBB" */
```



call: return address and rbp pushed

copy_in was called: the return address (0x401196) and the caller's frame pointer are on the stack. The input is 16 characters; buf is 8 bytes.

Step 1 / 7

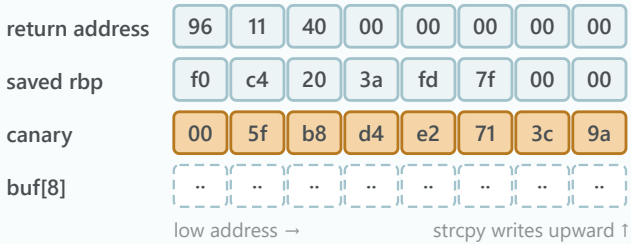
VARIABLES

`input = 16 chars`

OUTPUT

Stack Canary: Code Trace

```
1 /* the compiler adds the canary lines */
2 void copy_in(const char *input) {
▶     long canary = __stack_chk_guard;
▶     char buf[8];
5     strcpy(buf, input);
6     if (canary != __stack_chk_guard)
7         __stack_chk_fail();
8 }
9 /* input: "AAAAAAAABBBBBBBB" */
```



canary = random; first byte 00

Compiler-inserted prologue code places a value chosen at random at process start between buf and the saved rbp. Its first byte is 0x00, so string functions can neither read and leak it nor write it verbatim.

Step 2 / 7

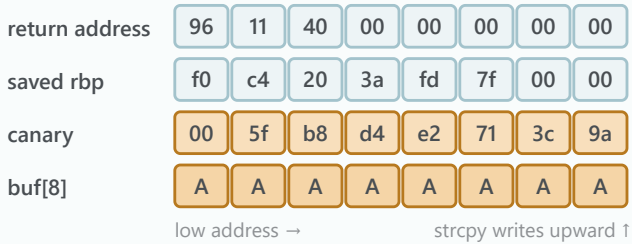
VARIABLES

`canary = 0x9a3c71e2d4b85f00`

OUTPUT

Stack Canary: Code Trace

```
1 /* the compiler adds the canary lines */
2 void copy_in(const char *input) {
3     long canary = __stack_chk_guard;
4     char buf[8];
▶     strcpy(buf, input);
6     if (canary != __stack_chk_guard)
7         __stack_chk_fail();
8 }
9 /* input: "AAAAAAAABBBBBBBB" */
```



strcpy: 8 bytes fit in buf

strcpy knows no length; it copies up to the \0 in the source. The first 8 bytes (A) fill buf.

Step 3 / 7

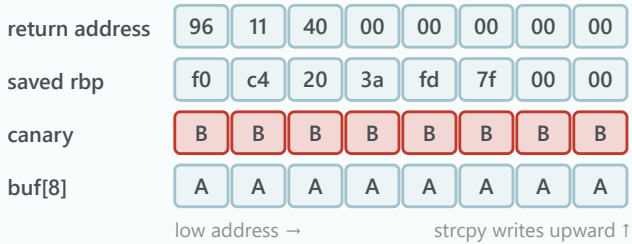
VARIABLES

`canary = 0x9a3c71e2d4b85f00` `written = 8`

OUTPUT

Stack Canary: Code Trace

```
1 /* the compiler adds the canary lines */
2 void copy_in(const char *input) {
3     long canary = __stack_chk_guard;
4     char buf[8];
▶     strcpy(buf, input);
6     if (canary != __stack_chk_guard)
7         __stack_chk_fail();
8 }
9 /* input: "AAAAAAAABBBBBBBB" */
```



overflow: canary overwritten

The next 8 bytes (B) land outside buf, on top of the canary: the canary is now 0x4242424242424242.

Step 4 / 7

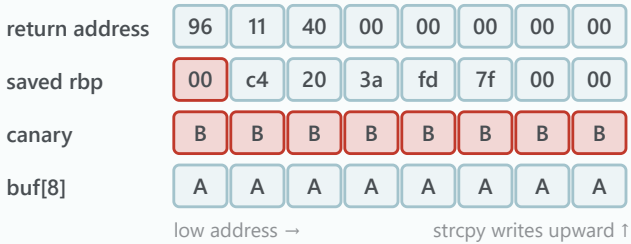
VARIABLES

`canary = 0x4242424242424242` `written = 16`

OUTPUT

Stack Canary: Code Trace

```
1 /* the compiler adds the canary lines */
2 void copy_in(const char *input) {
3     long canary = __stack_chk_guard;
4     char buf[8];
▶     strcpy(buf, input);
6     if (canary != __stack_chk_guard)
7         __stack_chk_fail();
8 }
9 /* input: "AAAAAAAABBBBBBBB" */
```



terminating \0 spilled into saved rbp

The terminating \0 corrupts the first byte of the saved rbp. A slightly longer input would reach the return address next — but the only way there is through the canary.

Step 5 / 7

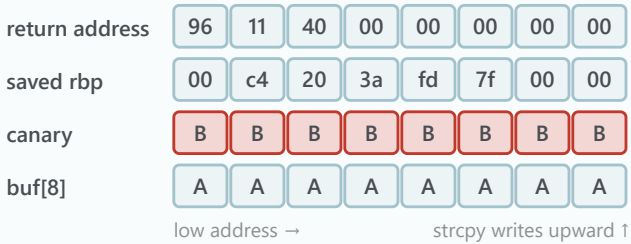
VARIABLES

`canary = 0x4242424242424242` `written = 17`

OUTPUT

Stack Canary: Code Trace

```
1 /* the compiler adds the canary lines */
2 void copy_in(const char *input) {
3     long canary = __stack_chk_guard;
4     char buf[8];
5     strcpy(buf, input);
▶     if (canary != __stack_chk_guard)
7         __stack_chk_fail();
8 }
9 /* input: "AAAAAAAABBBBBBBB" */
```



0x4242424242424242 ≠ 0x9a3c71e2d4b85f00

Before returning, the epilogue compares the canary on the stack with the value from process start: they differ.

Step 6 / 7

VARIABLES

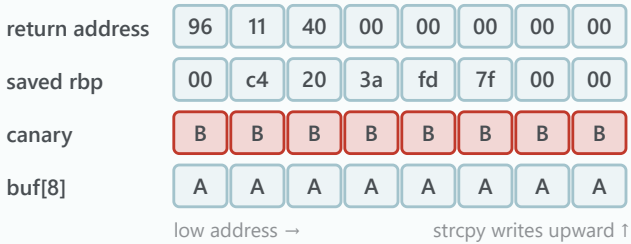
`canary = 0x42424242424242`

`guard = 0x9a3c71e2d4b85f00`

OUTPUT

Stack Canary: Code Trace

```
1 /* the compiler adds the canary lines */
2 void copy_in(const char *input) {
3     long canary = __stack_chk_guard;
4     char buf[8];
5     strcpy(buf, input);
6     if (canary != __stack_chk_guard)
▶         __stack_chk_fail();
8 }
9 /* input: "AAAAAAAABBBBBBBB" */
```



return never runs: the process is stopped

__stack_chk_fail prints the message and kills the process with SIGABRT. The corrupted frame is never returned to; the attack becomes a crash (lost availability, but no code execution).

Step 7 / 7

VARIABLES

`canary = 0x4242424242424242`

`result = SIGABRT`

OUTPUT

```
*** stack smashing detected ***: terminated
```

Other Memory and Injection Bugs

Vulnerability	Why it is dangerous	Countermeasure
Format string	`printf(user)` — reads/writes memory via %n/%x	`printf("%s", user)`
Integer overflow	`len = a + b` wraps around, causing a small allocation	Checked arithmetic, size types
Use after free	Accessing freed memory (UAF)	Ownership (Rust), smart pointers, quarantine
Double free	The same block freed twice; the heap is corrupted	NULL after free; a hardened allocator
Out-of-bounds read	Memory leaks like Heartbleed	Bounds checks, fuzzing
Command/SQL injection	Input gets mixed into code/queries	Parameterized queries, escaping, allowlists

The common root cause: **data mixed with code or control**, and untrusted input used unchecked. Defense: validate input, keep data apart from code; memory-safe languages and **fuzzing** catch bugs early.

The TOCTOU Race Condition

```
1 /* TOCTOU: a setuid-root helper */
▶ if (access(path, W_OK) == 0) {   // CHECK
3     fd = open(path, O_WRONLY);     // USE
4     write(fd, data, n);
5 }
```



The program first **checks** with `access()` : “may ali write this file?” `/tmp/f` is an ordinary, permitted file. The answer is yes.

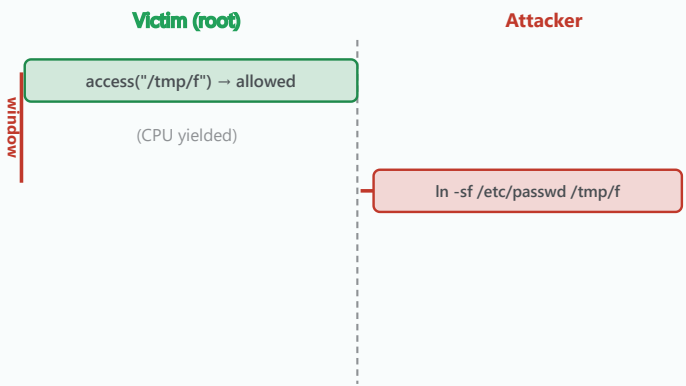
Step 1 / 5

VARIABLES

—

The TOCTOU Race Condition

```
1 /* TOCTOU: a setuid-root helper */
▶ if (access(path, W_OK) == 0) {   // CHECK
3     fd = open(path, O_WRONLY);     // USE
4     write(fd, data, n);
5 }
```



There is a small **time window** between check and use. If the program loses the CPU right here...

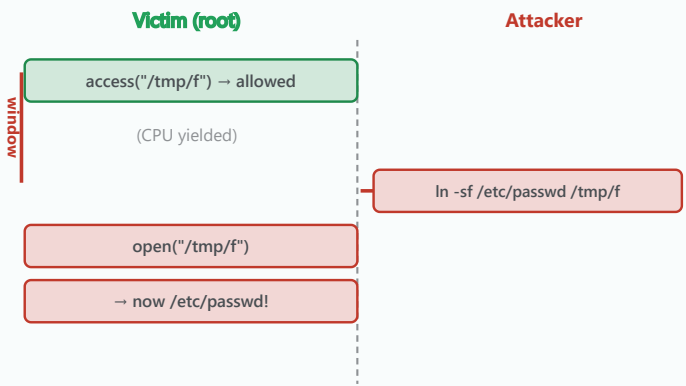
Step 2 / 5

VARIABLES

—

The TOCTOU Race Condition

```
1 /* TOCTOU: a setuid-root helper */
▶ if (access(path, W_OK) == 0) {   // CHECK
▶     fd = open(path, O_WRONLY);     // USE
4     write(fd, data, n);
5 }
```



The attacker deletes `/tmp/f` and puts a **symbolic link** to `/etc/passwd` in its place. Same file name, different target.

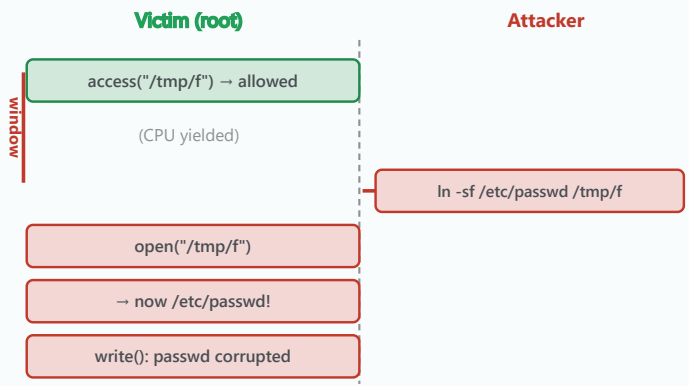
Step 3 / 5

VARIABLES

—

The TOCTOU Race Condition

```
1 /* TOCTOU: a setuid-root helper */
2 if (access(path, W_OK) == 0) {   // CHECK
      fd = open(path, O_WRONLY);     // USE
      write(fd, data, n);
5 }
```



`open()` now opens the **new target**, not the name: `/etc/passwd`, with root rights. **Time of check ≠ time of use** (TOCTOU).

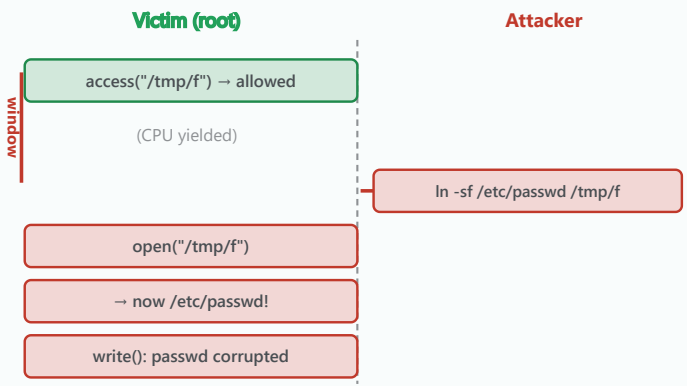
Step 4 / 5

VARIABLES

—

The TOCTOU Race Condition

```
1 /* TOCTOU: a setuid-root helper */
▶ if (access(path, W_OK) == 0) {   // CHECK
▶     fd = open(path, O_WRONLY);     // USE
4     write(fd, data, n);
5 }
```



The fix: never resolve the name twice. Open the file once (`open` + `O_NOFOLLOW`), then check **the same descriptor** with `fstat` / `faccessat` ; or use `openat` and drop privileges.

Step 5 / 5

VARIABLES

—

Insider Attacks: Logic Bombs, Back Doors

(a) Normal code

```
if (user_ok(name, pw))
grant_access();
else
deny();
```

(b) Code with a back door

```
if (user_ok(name, pw)
|| eq(name, "zzz"))
grant_access();
else
deny();
```

"zzz" always gets in

Logic bomb

Malicious code triggered by a condition (a date, being fired). Often planted by a disgruntled employee.

Back door

A hidden entry that bypasses authentication. Can also be slipped into the supply chain.

Login spoofing

A fake login screen collects the user name and password, then hands off to the real one.

Against login spoofing, a **secure attention key**: in Windows, Ctrl+Alt+Del always summons the real kernel login screen, and no user program can imitate it. Against back doors: code review, two-person approval, reproducible builds.

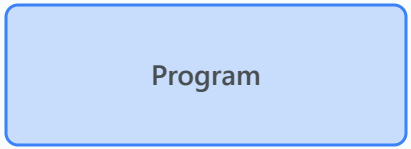
Types of Malware

Type	Spread	Goal / behavior
Virus	Attaches itself to another program; infects when the user runs it	Corrupts files, spreads
Worm	Self-propagating over the network; no user needed	Spreads fast, consumes resources
Trojan horse	Hidden inside a useful-looking program	Opens back doors, steals data
Ransomware	Phishing, exploits, RDP	Encrypts files, demands a ransom
Rootkit	Installed after gaining privileges	Hides itself and other malware
Spyware	Trojans, bundled software, fake extensions	Collects data, sends it out
Bot / botnet	Worm + back door	DDoS, spam, mining on remote command

The lines are blurry: modern malware usually combines several types (e.g. a bot that spreads like a worm, hides with a rootkit and carries ransomware). Spyware may change the browser home page or search engine, add toolbars and produce an unstoppable stream of ads.

Parasitic Viruses

(a) clean program



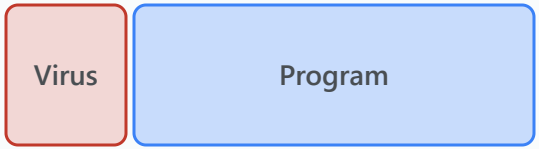
The entry point is redirected to the virus; the virus runs, then jumps to the real program

A clean executable: the real program code from start to end.

Other kinds: companion viruses (a .com file that runs before the .exe of the same name), memory-resident, boot sector, device driver, macro (in documents) and source code viruses. Most are historical; modern malware uses scripts, in-memory (fileless) and supply-chain routes.

Parasitic Viruses

(b) prepended



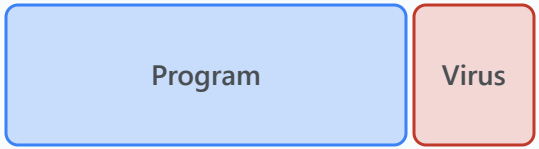
The entry point is redirected to the virus; the virus runs, then jumps to the real program

Prepending virus: the virus code goes at the start and the entry point is redirected to it.

Other kinds: companion viruses (a .com file that runs before the .exe of the same name), memory-resident, boot sector, device driver, macro (in documents) and source code viruses. Most are historical; modern malware uses scripts, in-memory (fileless) and supply-chain routes.

Parasitic Viruses

(c) appended



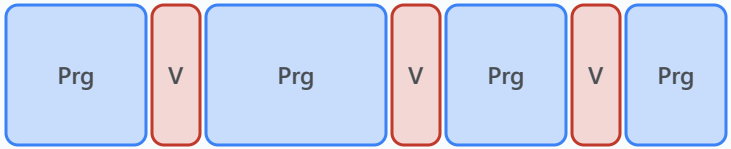
The entry point is redirected to the virus; the virus runs, then jumps to the real program

Appending virus: the virus goes at the end, with a jump to it placed at the start (most common).

Other kinds: companion viruses (a .com file that runs before the .exe of the same name), memory-resident, boot sector, device driver, macro (in documents) and source code viruses. Most are historical; modern malware uses scripts, in-memory (fileless) and supply-chain routes.

Parasitic Viruses

(d) spread over cavities

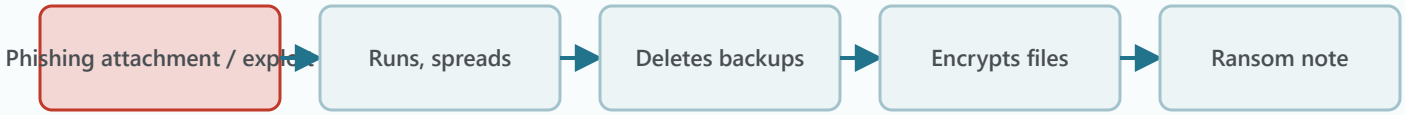


The entry point is redirected to the virus; the virus runs, then jumps to the real program

Cavity virus: scatters itself over unused gaps in the program; the file size does not change, which makes it harder to detect.

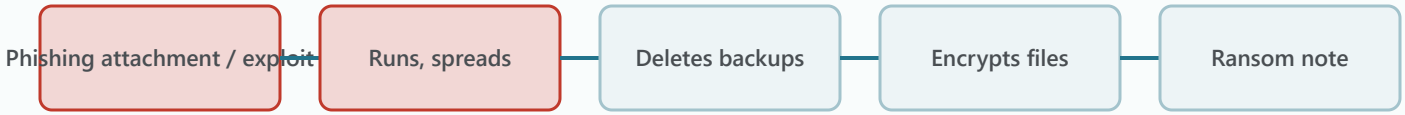
Other kinds: companion viruses (a .com file that runs before the .exe of the same name), memory-resident, boot sector, device driver, macro (in documents) and source code viruses. Most are historical; modern malware uses scripts, in-memory (fileless) and supply-chain routes.

Ransomware



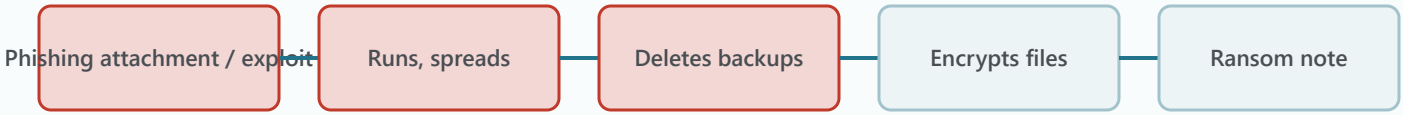
Initial access usually comes from a phishing attachment, an exposed service or stolen RDP credentials.

Ransomware



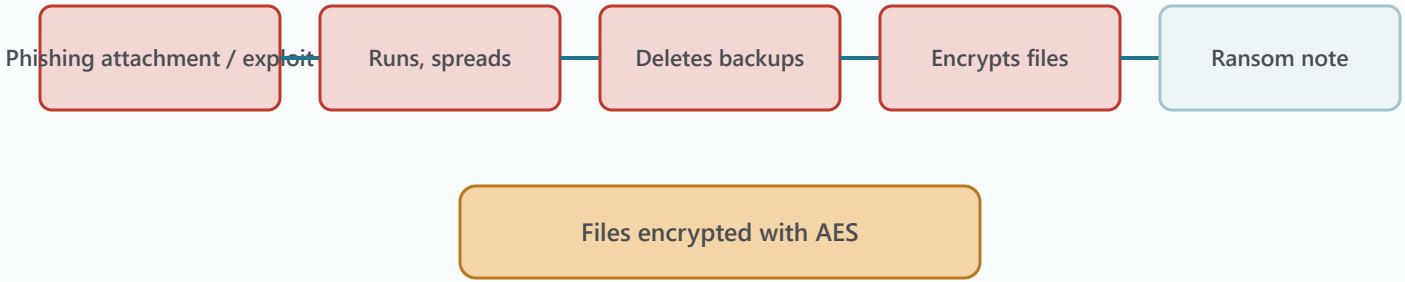
The malware runs and moves laterally to reach as many machines as it can.

Ransomware



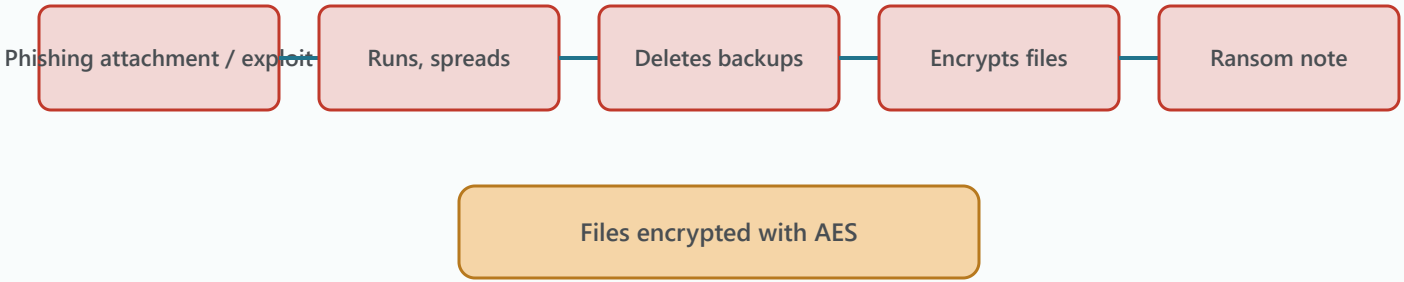
To block recovery it **deletes shadow copies and backups** (hence the need for offline/immutable backups).

Ransomware



Files are locked with strong symmetric encryption; the key goes to the attacker's server.

Ransomware

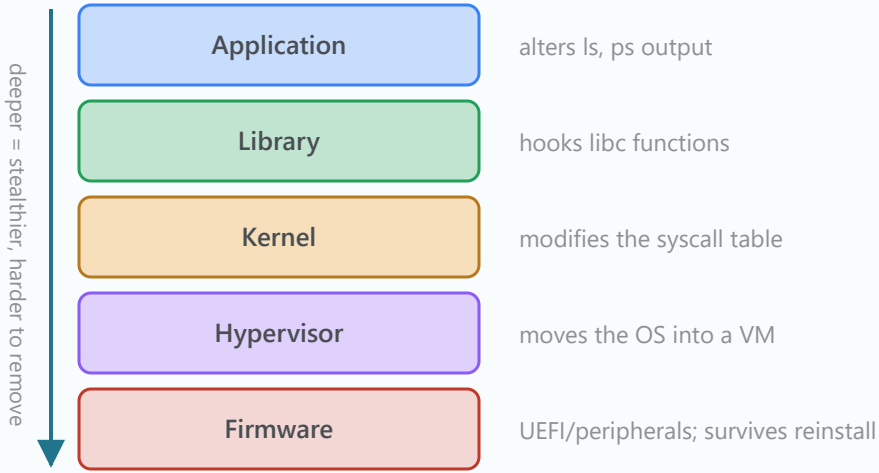


Paying ≠ a guaranteed key; backups + patches are the only reliable defense

The ransom note appears. Paying does not guarantee the key and encourages new attacks. Defense: 3-2-1 backups, patching, MFA, segmentation, EDR.

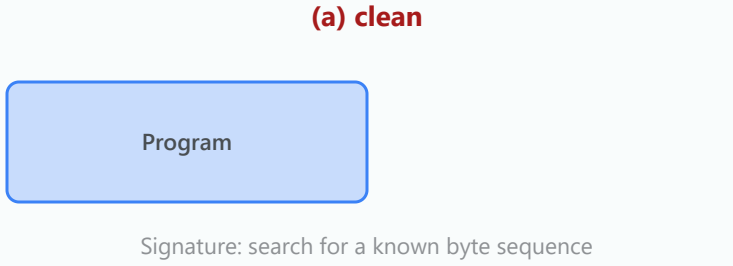
MALWARE

Rootkits and Stealth



A **rootkit** hides its presence: it conceals files, processes and network connections from the OS. The deeper it sits, the stronger it is, but the harder to install. Defenses at the kernel and below: signed drivers, **secure boot**, measured boot (TPM) and scanning from outside (offline).

Antivirus and Evasion Techniques



Step 1 / 5

Signature scanning
Known patterns; fast, few false alarms, misses the new.

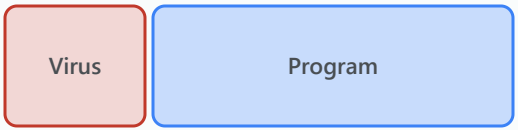
Integrity checking
Stores file hashes and notices changes.

Behavioral / heuristic
Watches for suspicious actions (encryption, injection).

Sandbox + cloud
Runs the suspect in an isolated environment and observes.

Antivirus and Evasion Techniques

(b) infected



The signature catches the virus

Signature scanning: look for the byte sequences (signatures) of known viruses. Fast, but catches only what is known.

Step 2 / 5

Signature scanning	Integrity checking	Behavioral / heuristic	Sandbox + cloud
Known patterns; fast, few false alarms, misses the new.	Stores file hashes and notices changes.	Watches for suspicious actions (encryption, injection).	Runs the suspect in an isolated environment and observes.

Antivirus and Evasion Techniques

(c) compressed



Compression hides the signature; the scanner must unpack and look

If the virus **compresses** itself the signature is invisible; the scanner has to unpack the file and scan inside.

Signature scanning
Known patterns; fast, few false alarms, misses the new.

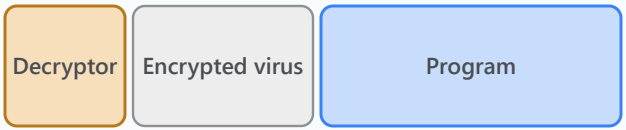
Integrity checking
Stores file hashes and notices changes.

Behavioral / heuristic
Watches for suspicious actions (encryption, injection).

Sandbox + cloud
Runs the suspect in an isolated environment and observes.

Antivirus and Evasion Techniques

(d) encrypted virus



Encrypted body: only the decryptor is constant, so its signature is sought

Encrypted virus: the body is encrypted and only a small decryptor is in the clear; the scanner looks for the decryptor's signature.

Signature scanning
Known patterns; fast, few false alarms, misses the new.

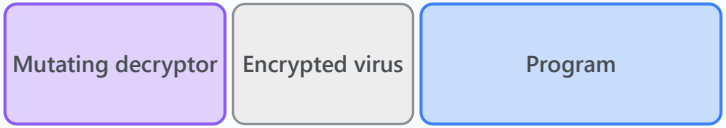
Integrity checking
Stores file hashes and notices changes.

Behavioral / heuristic
Watches for suspicious actions (encryption, injection).

Sandbox + cloud
Runs the suspect in an isolated environment and observes.

Antivirus and Evasion Techniques

(e) polymorphic



Polymorphic: the decryptor differs in every copy → signatures fail; behavior analysis is needed

Polymorphic virus: even the decryptor changes in every copy. Signatures are not enough → **heuristic/behavioral** analysis, running it in a **sandbox** and watching, is needed.

Step 5 / 5

Signature scanning
Known patterns; fast, few false alarms, misses the new.

Integrity checking
Stores file hashes and notices changes.

Behavioral / heuristic
Watches for suspicious actions (encryption, injection).

Sandbox + cloud
Runs the suspect in an isolated environment and observes.

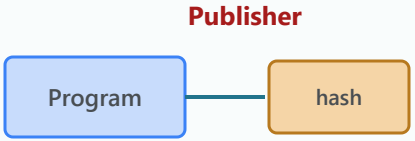
Code Signing

Publisher

Operating system

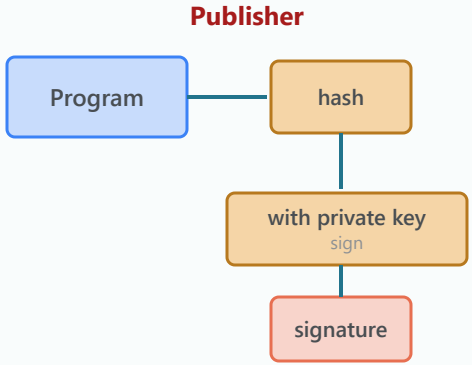
The publisher computes the program's **hash** (the same as a digital signature, this time applied to code).

Code Signing



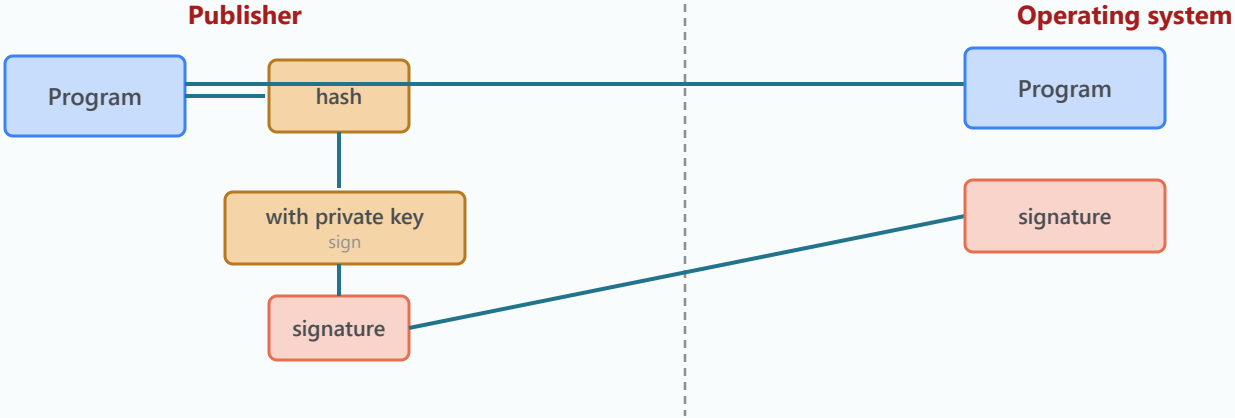
The hash is **signed with the publisher's private key**; the signature ships with the program.

Code Signing



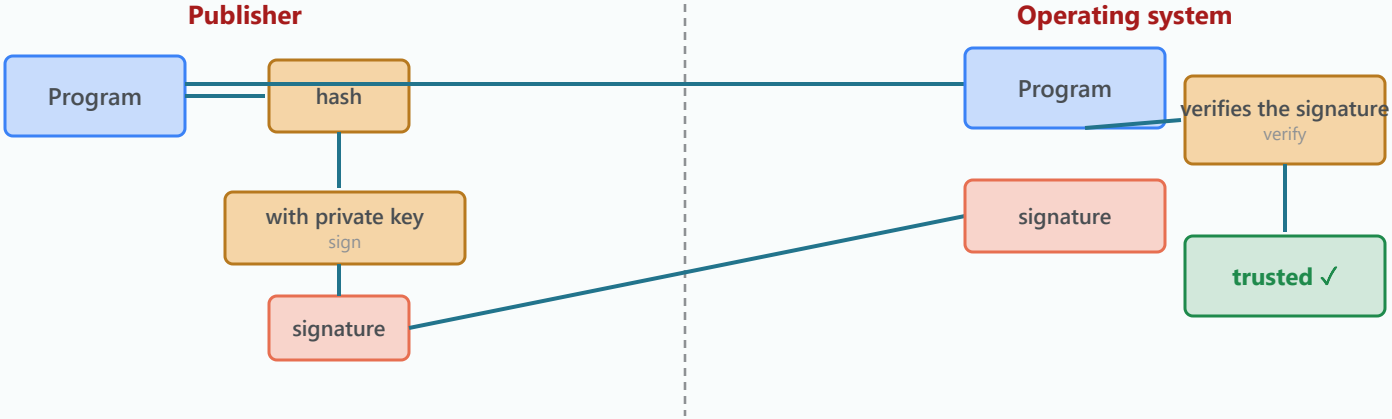
The user receives the program and signature; the publisher's certificate carries its public key.

Code Signing



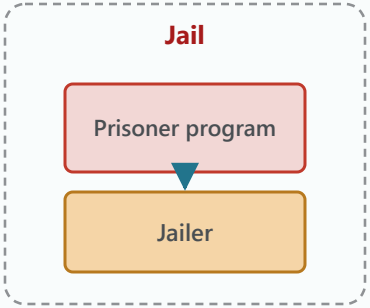
The OS recomputes the hash and verifies the signature with the public key; if they match, the program is **unmodified** and from that publisher.

Code Signing



App stores, driver signing, `apt` / `dnf` package signatures and platform “signed code only” policies all rest on this.

Sandboxing and Confinement



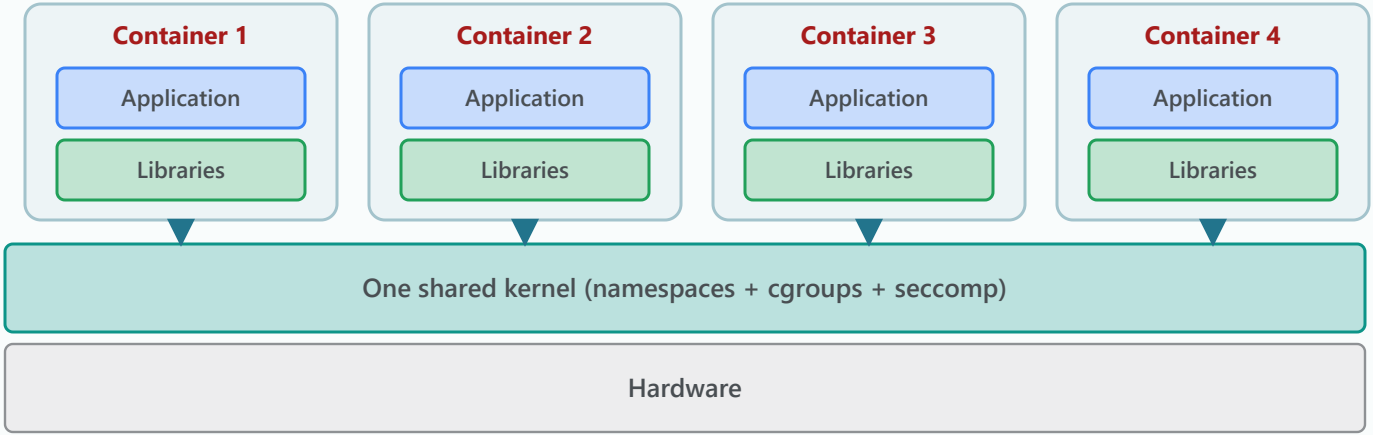
Every system call
is checked by the jailer;
unauthorized ones
are blocked

Modern sandbox

- seccomp-bpf: allowed system calls
- namespaces: separate PID/net/file views
- cgroups: CPU/memory limits
- capabilities dropped
- SELinux/AppArmor profile

Sandboxing: run untrusted code in a restricted environment to limit the damage it can do. Browser tabs, mobile apps and server daemons all run sandboxed today. The Linux building blocks: `seccomp`, namespaces, cgroups and mandatory access control.

Containers and Isolation



	Container	Virtual machine
Isolation	Kernel features	Hardware + hypervisor
Overhead	Low, starts in seconds	High, separate kernel
Attack surface	Kernel bug = escape risk	Stronger boundary

Mandatory Access Control: SELinux and AppArmor

DAC (discretionary)

Permissions granted by the owner (rwx, ACLs). Users can change the permissions of their own files — and so can malware.

MAC (mandatory)

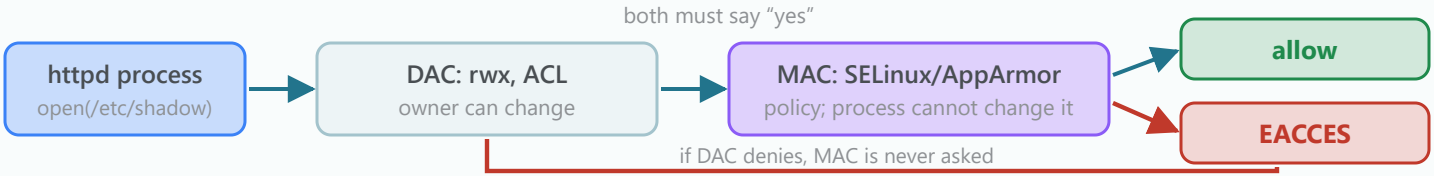
The system/administrator sets the policy; a process cannot loosen its own restrictions. BLP/Biba, SELinux, AppArmor.

SELinux

A label on every subject and object; fine-grained rules through type enforcement. Powerful but complex.

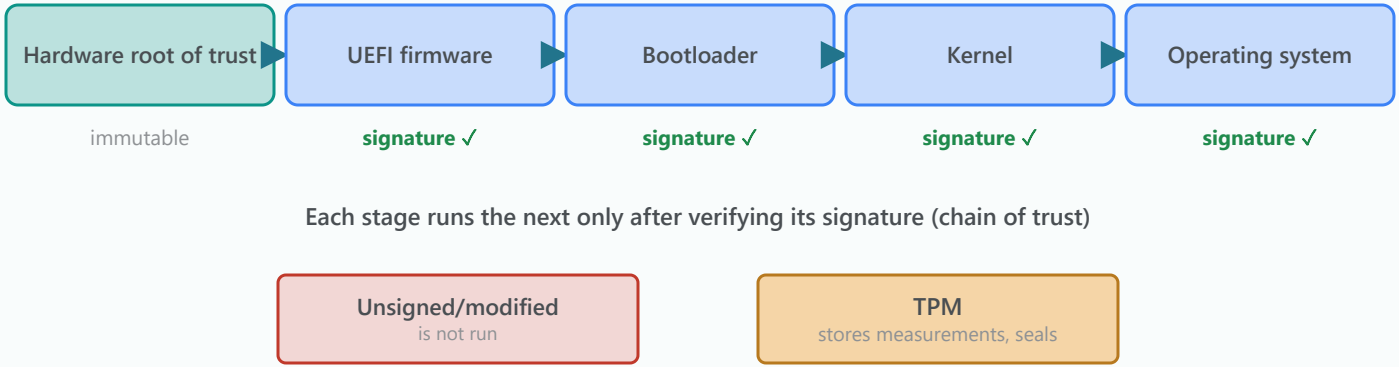
AppArmor

Path-based profiles; a per-application allowlist. Easier to write and understand.



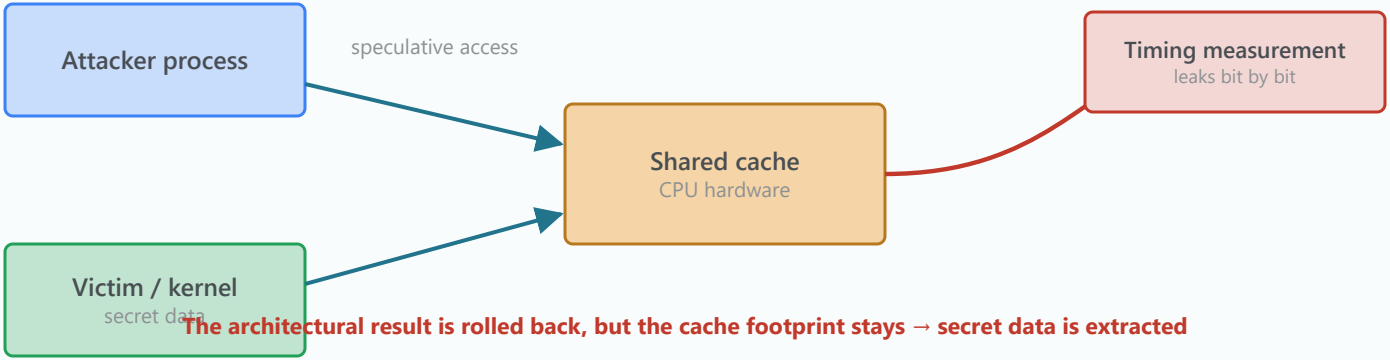
Mandatory access control (MAC) confines even a compromised process to what its profile allows: if a web server is exploited, the attacker reaches only the files that profile names. Android runs entirely in SELinux enforcing mode.

Secure Boot and the TPM



Secure boot: each layer verifies the next one's signature, anchored in a hardware root of trust. The **TPM** (Trusted Platform Module) stores hashes of boot components (PCRs); with **measured boot** the disk-encryption key is released only if the system is in the expected state (remote attestation). Rootkits and bootkits try to break this chain.

Spectre, Meltdown and Side Channels



A **side channel** leaks data not directly but through an indirect physical effect (time, power, cache, sound). **Meltdown** and **Spectre** (2018) abuse CPUs' **speculative execution**: the footprint a canceled operation leaves in the cache is read by timing, breaking isolation. Mitigations: kernel page-table isolation (KPTI), microcode, compiler barriers — most at a performance cost.

Summary

Topic	Key idea	Today
Goals	Confidentiality, integrity, availability	Zero trust, + authenticity/accountability
Cryptography	Symmetric + asymmetric + hash + signature	AES-GCM, X25519, Ed25519, TLS 1.3, ML-KEM
Access control	Matrix → ACLs or capabilities	Least privilege, Linux capabilities, MAC
Formal models	BLP confidentiality, Biba integrity	SELinux MLS, Windows MIC
Authentication	Salted slow hashes + MFA	Argon2, FIDO2/passkeys
Memory safety	An overflow corrupts the return address	Canary+NX+ASLR+CFI, Rust
Isolation	Sandboxes limit damage	seccomp, namespaces, containers
Hardware	Chain of trust and measurement	Secure boot, TPM; side channels

Review Questions

1. Explain the three parts of the CIA triad, with an example breach of each.
2. Compare the ACL and capability forms of the same access matrix: in which is revocation easy, and in which delegation? Why?
3. Write the Bell-LaPadula and Biba rules (read/write directions). Which protects confidentiality, which integrity?
4. Why store passwords with salted Argon2/bcrypt rather than plain SHA-256? What are the separate roles of the salt and the slowness?
5. Why are FIDO2/passkeys phishing-resistant compared with password + TOTP?
6. How is the return address hijacked in a stack buffer overflow? At which stage do ASLR, NX and the stack canary make it harder?
7. Describe a TOCTOU race with the `access()` / `open()` example and give a fix.
8. Distinguish a covert storage channel from a timing channel with examples. Why can they never be fully closed?
9. Explain the secure boot chain of trust and the TPM's role. Where does it stop a rootkit?
10. What is the fundamental difference between container and VM isolation? In which is a kernel bug more dangerous?

▶ **Answers**

References

- A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4th ed. (2014) / 5th ed. (2023), Pearson — Chapter 9: Security.
- A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10th ed., Wiley, 2018 — Chapter 16: Security; Chapter 17: Protection.
- R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — security chapters.
- R. Anderson, *Security Engineering*, 3rd ed., Wiley, 2020.
- J. H. Saltzer, M. D. Schroeder, “The Protection of Information in Computer Systems”, 1975 (least privilege and design principles).
- D. E. Bell, L. J. LaPadula (1973); K. J. Biba (1977) — formal security models. B. W. Lampson, “A Note on the Confinement Problem”, 1973.
- RFC 8446 (TLS 1.3), RFC 6238 (TOTP), RFC 9106 (Argon2); FIDO2/WebAuthn (W3C); NIST FIPS 203 (ML-KEM).
- Aleph One, “Smashing the Stack for Fun and Profit”, Phrack 49, 1996; Kocher et al., “Spectre”; Lipp et al., “Meltdown”, 2018.

The End

Chapter 13: Security

Previous: Chapter 12: Multiprocessor Systems · Next: Chapter 14: Operating-System Design