

Attacks and Vulnerabilities

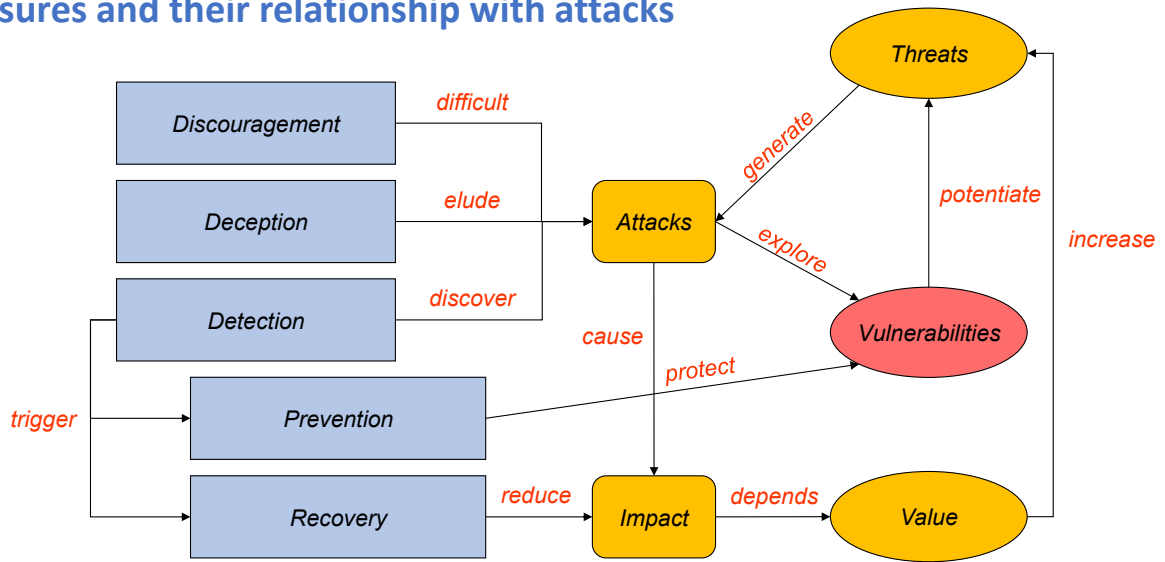
SIO

deti universidade de aveiro
departamento de eletrónica,
telecomunicações e informática

This lecture is the pivot of the course. Until now we have reasoned about information security from the top down: assets, the CIA triad, risk, policies and controls. From here on we reason from the point of view of the adversary, because that is the only perspective that tells us whether our controls are placed in the right places. Two words carry the whole deck: an attack is an action taken by an agent, while a vulnerability is a state of a system. Attacks exploit vulnerabilities, but they also exploit people and processes, which is why some of the most damaging incidents in this lecture involve no defective code at all.

Information Security

Measures and their relationship with attacks



This diagram is a causal map, and it should be read as a graph rather than as a list. The verbs printed on the arrows are the actual content: threat actors generate attacks, attacks explore vulnerabilities, vulnerabilities potentiate attacks, attacks trigger impact, and the five families of measures act on the loop by reducing, preventing, detecting, deceiving and recovering. Two relations deserve attention because students usually invert them. Vulnerabilities depend on the value of the target, not only on the care of the developers: high-value and widely deployed software attracts both attackers and researchers, so it shows more published weaknesses. And impact is not a property of an attack but a product of attack and context: the same defect in a test VM and in a hospital's imaging system produce incomparable damage.

Take a single chain and follow it through the map, because exam questions are usually chains of this type. A university runs a mail platform with a flaw that allows authentication bypass (vulnerability); an organised crime group specialising in business email compromise (threat actor) explores it, triggers a payload, and diverts invoice payments (attack and impact); the university loses money, must notify under GDPR, and loses trust (impact measured in euros, time and reputation). The measures then intervene at different points: patching removes the vulnerability, authentication hardening raises the cost of exploring it, detection shortens the time between intrusion and discovery, deception wastes the attacker's time, backups and incident response limit the impact. No single measure closes the loop.

The five measure families are not alternatives, they are layers with different time horizons and different failure modes. Prevention removes the possibility of the attack and is the cheapest when done at design time, but it can never be complete. Discouragement works on the attacker's expected utility: visible punishment, prosecution, forensic capability. Detection does not reduce the probability of an attempt, it reduces the time an attacker spends undetected, which is the variable that most determines final damage. Deception (honeypots, honeytokens) creates cheap signals about attacker behaviour that would otherwise be invisible in noise. Recovery accepts failure and buys resilience: without it, one successful attack ends the organisation.

Finally, notice that the map has no arrow that goes from measures to the disappearance of threats. Security is not solved, it is only displaced: making one route expensive moves the attacker to another, which is why the lecture on measures is followed by the lecture on vulnerabilities, and why the second half of this deck is about tracking, naming and managing weaknesses over time.

Measures (and some tools)

- **Discouragement**

- Punishment
 - Legal restrictions
 - Forensic evidences
- Security barriers
 - Firewalls
 - Authentication
 - Secure communication
 - Sandboxing

- **Detection**

- Intrusion detection system
 - e.g. Zeek, Suricata
- Auditing
- Forensic break-in analysis

- **Discouragement**

- Honeypots / honeynets
- Forensic follow-up

- **Prevention**

- Restrictive policies
 - e.g. least privilege principle
- Vulnerability scanning
 - e.g. SAST tools
- Vulnerability patching
 - e.g. regular updates

- **Recovery**

- Backups
- Redundant systems
- Forensic recovery

Discouragement is often misunderstood as 'education campaigns', but its technical core is making the attacker's expected cost exceed the expected gain. Legal restrictions matter because they change the attacker's risk, not the attack's difficulty: the Council of Europe's Budapest Convention (2001) criminalises illegal access, interference and sabotage and obliges cross-border cooperation. Deterrence only works against actors who fear identification, so it is effective against opportunistic attackers and largely ineffective against crime companies with jurisdiction shopping or against state actors. Forensic evidence is the enforcement arm of discouragement: without retained logs, chain of custody, disk images and time synchronisation, prosecution is impossible, so log retention is a security control with legal value, not merely an operational one.

The barrier family is what most people picture as 'security product': firewalls and segmentation (stateful, next-generation, web application firewalls), authentication (passwords plus factors, ideally phishing-resistant ones such as FIDO2/WebAuthn, with step-up for privileged actions), secure communication (TLS 1.3 with forward secrecy, IPsec or WireGuard tunnels, mutual TLS between services, and the unglamorous work of key and certificate lifecycle management), and sandboxing (containers, VMs, browser site isolation, application sandboxes, seccomp/AppArmor, and read-only filesystems). The common principle is containment: assume a component will be compromised and limit what it can reach. That is why sandboxing survived as a control while perimeter-only thinking did not.

Detection is where the tool names in the slide come from. Network intrusion detection reads traffic and raises signals: Zeek is not really an alerting engine but a framework that turns traffic into structured logs and lets you script protocol analytics, which makes it superb for hunt and forensics; Suricata matches traffic against signature rulesets (community and commercial) and can run in inline mode as an intrusion prevention system. Host and enterprise detection comes from auditing (auditd and journald on Linux, Windows Event Log, Sysmon, cloud control-plane logs) aggregated in a SIEM, plus EDR telemetry of processes, loads and network sockets. Forensic break-in analysis is the disciplined tail of the family: acquire before you remediate, image memory and disk with documented hashes, build a timeline, and only then rebuild. An organisation that rebuilds first loses the ability to answer 'how did they get in, and are we sure they are gone?'.

Prevention is cheapest at design time and most effective when it removes whole classes of attack. Restrictive policies such as least privilege and default-deny mean a compromised account or process can reach almost nothing; unprivileged and just-in-time administrative access, plus privileged access management, keep the blast radius small. Vulnerability scanning before release (static analysis of source, dependency and composition analysis against known-vulnerability databases, secrets scanning) finds weaknesses while they are still cheap, and patching keeps the installed base current: predictable rings, canary systems, staged rollouts, explicit end-of-life policies for unsupported software. Recovery is the acceptance that the rest will fail: backups with immutable or air-gapped copies, restores rehearsed on a calendar rather than trusted, redundant systems and degraded modes so the service survives the loss of a site, and forensic recovery so the evidence survives the rebuild.

Threats and Attacks

- Threat Actors **explore Vulnerabilities**
 - They will trigger and action, send a crafted payload, to disrupt CIA and existing policies
- Threat Actors also conduct Attacks **without** clear **Software Vulnerabilities**
 - **Targeting people, processes and resources**
 - Out of the scope of Information Security, but relevant to the security of Organizations
- The number vulnerabilities depends on Value and Security Posture
 - More popular software may have higher number of **vulnerabilities discovered**
 - Also software with more higher maturity (more tests) may show more vulnerabilities
 - Unknown software can look good, while being broken

	Product Name	Vendor Name	Product Type	Number of Vulnerabilities
1	Debian Linux	Debian	OS	8799
2	Android	Google	OS	7169
3	Linux Kernel	Linux	OS	5486
4	Fedora	Fedoraproject	OS	5116
5	Ubuntu Linux	Canonical	OS	4094
6	Windows Server 2016	Microsoft	OS	3661
7	Chrome	Google	Application	3504
8	Iphone Os	Apple	OS	3437
9	Windows Server 2019	Microsoft	OS	3217
10	Mac Os X	Apple	OS	3206
11	Windows Server 2012	Microsoft	OS	3059
12	Windows 10	Microsoft	OS	3031
13	Windows Server 2008	Microsoft	OS	2983
14	Firefox	Mozilla	Application	2667
15	Windows 7	Microsoft	OS	2370
16	Windows 8.1	Microsoft	OS	2217
17	Windows Rt 8.1	Microsoft	OS	2017
18	Enterprise Linux Desktop	Redhat	OS	1925

<https://www.cvedetails.com/top-50-products.php?year=0>

Threat actors are not a single species, and their differences explain the differences in the attacks we will study. Script kiddies run other people's tools with low capability and random motivation, and are defeated by basic hygiene. Organised crime is professional, patient, financially motivated, and industrialised: phishing-as-a-service, initial access brokers, affiliates, ransomware infrastructure. Hacktivists seek attention and political damage, often through defacement and denial of service. Insiders act with legitimate credentials. Nation-state operators (usually labelled APT, advanced persistent threat, a marketing term more than a technical one) have long timelines, custom tooling and patience, and they are the ones that most often hold zero-days. Motivation and capability determine what a vulnerability is worth, which is why the same flaw is a nuisance for a personal blog and a catastrophe for a bank.

The slide's definition of an attack is deliberately action-centred: a threat actor explores a vulnerability, then triggers an action or sends a crafted payload, disrupting confidentiality, integrity or availability and existing policies. The phrase 'existing policies' is important: some intrusions violate only organisational rules and no technical control at all, for example a privileged engineer exporting a client database to a personal laptop, or an employee disabling MFA on a shared account under pressure. That is why the slide insists that many attacks target people, processes and resources rather than software, and why they fall partly outside information security as a technical discipline: they belong to governance, physical security, HR and internal control. A useful operational model is the sequence reconnaissance, weaponisation, delivery, exploitation, installation, command and control, actions on objectives, refined by MITRE ATT&CK into tactics and techniques that map directly onto detection rules.

The second half of the slide is a warning about statistics. The number of vulnerabilities published for a product is not a measurement of its quality; it is a measurement of how much it is used and how much it is looked at. Popular targets dominate the public rankings, and mature codebases that are continuously fuzzed and reviewed also keep producing findings, which is a sign of health: the bugs are found before the attackers do. The chart in the figure is a product ranking of published vulnerabilities, and it is populated by the usual suspects, large operating systems, ubiquitous open-source components, browsers, office suites and network equipment, together with industrial vendors whose products are now networked. Read such a ranking as 'where research goes', never as 'where risk is', because risk also needs installed base and exposure.

The inverse trap is the one that actually hurts: 'unknown software can look good, while being broken'. A small proprietary protocol stack, a device firmware from a vendor with no security contact, a library with three users and no bug tracker may show zero CVEs for a decade and be a catalogues of memory-safety errors. Absence of reports reflects absence of eyes.

Common Attacks and Threats

Only some... more here: <https://owasp.org/www-community/attacks/>

Denial Of
Service

MiTM

Phishing

Ransomware

Password

Injection

Malware

Insider
Threats

This slide sets the menu, and the menu is a taxonomy rather than a theory. Taxonomies exist so that practitioners can name things consistently: the same incident reported by a technician, a lawyer, an insurer and a regulator should be recognisable as one object. The OWASP community page referenced in the slide collects hundreds of named attacks, and other catalogues approach the same territory from other angles, for example ENISA's threat taxonomy for policy-level discussion, MITRE ATT&CK for behaviour and detection engineering, and CWE for code-level weaknesses. None of them is a partition of reality: ransomware is often delivered by phishing and sometimes through an injection in a public application; a denial of service is frequently a smokescreen while another team works inside the network. Overlap is not a flaw, it is the price of describing human activity.

The selection criterion used here is prevalence times consequence, which is why these families are present in every industry report on breaches, year after year, with variations in order: attacks that use stolen credentials and socially engineered messages, attacks that exhaust resources, attacks that abuse a trusted position, and attacks that inject unintended code. What is missing from the menu is equally instructive: physical intrusion, theft of equipment, wire taps, legal compulsion, and the growing field of abuse of legitimate functionality such as insider data exfiltration through sanctioned cloud storage. Nothing in this deck is exhaustive, and a student who leaves believing the list is closed has misunderstood the point.

For each family, keep the same analytical template, because the real life rewards structure over memory.

Prerequisites: what must already be true for the attack to be possible (open service, untrained user, flat network, unpatched component).

Mechanism: what actually consumes resources, alters data or reveals information.

Consequences: which of confidentiality, integrity and availability are hit, and in what order.

Indicators: which artefacts remain (log lines, traffic anomalies, unexpected processes, new certificates, changed files).

Controls: which of the measure families from the previous slide reduce probability, reduce dwell time or limit damage.

If you can produce those five columns for all eight families, you have the second half of the course ready to use, because CVEs, CWEs and the OWASP Top 10 are simply the tools that let you reason about the prerequisites column.

Denial of Service (DoS)

- Attacker overwhelms the resources of a system to the point where it is unable to reply to legitimate service requests.
 - Explores **software/system/architecture vulnerabilities**
 - Overwhelms **server providing the service**
 - Overwhelms dependent services such as the Authentication or Database servers
 - Frequently executed as a DDoS – Distributed DoS
 - Example: HTTP/2 Rapid Reset with multi-Tbps IoT attacks
- Impact: Clients are unable to access a service
 - Financial, brand and operational damage (e.g. Denial of Wallet)
 - Popular in relevant moments (exams, elections, public events)
 - Popular due to the low cost and low complexity

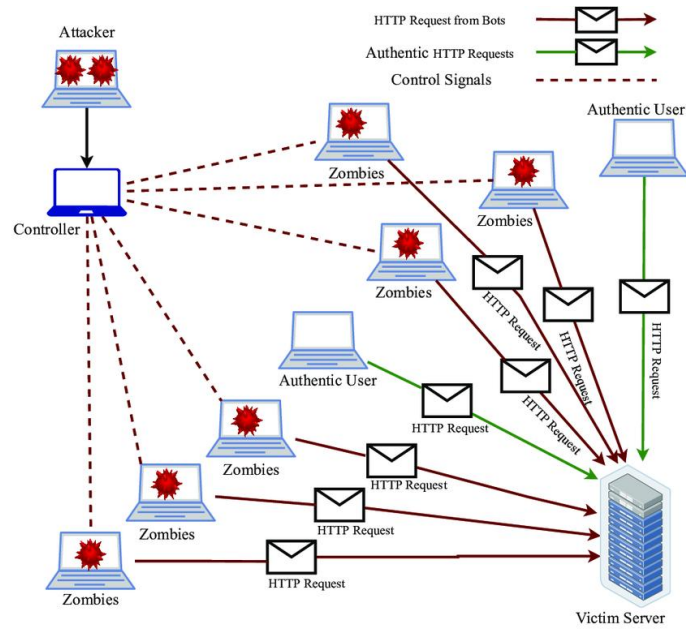
A denial of service is the cheapest attack in the catalogue, and that is precisely why it is everywhere. The mechanism is a resource that runs out before the attacker's budget does: network bandwidth, connection table entries in a firewall or load balancer, worker threads, file descriptors, memory, database connections, DNS resolution capacity, or the CPU cost of an apparently innocent operation such as image resizing, password hashing or regular expression matching.

Two amplification tricks dominate the network layer. Reflection sends requests to a third-party server with the victim's address forged as the source, so the reply goes to the victim; amplification multiplies bytes because the reply is much larger than the request. Open DNS resolvers, NTP's monlist command, memcached (whose amplification factor reaches five orders of magnitude), SSDP, CLDAP and character generators have all been used for this. Botnets of compromised cameras, routers and set-top boxes, of which Mirai in 2016 was the archetype, turned this into terabit-per-second assaults on infrastructure providers and, famously, on a DNS operator, taking large parts of a region's web offline because a name could not be resolved. Since then, reported record attacks have grown by an order of magnitude, and the countermeasures are largely architectural: anycast and distributed scrubbing capacity, over-provisioning, and the removal of amplifying open resolvers.

The slide mentions HTTP/2 Rapid Reset, which is worth understanding in detail because it inverts every assumption of an application-layer defence. A client may open many streams in one HTTP/2 connection and, in legitimate use, cancel them; the specification allows resetting a stream immediately after sending headers. Attackers exploited that by opening streams and resetting them in a burst, so servers allocated and released resources at enormous rates; the traffic was protocol-valid, the connection count stayed low, and conventional rate limiting and web application firewalls saw nothing wrong. It became CVE-2023-44487 and affected major web servers, proxies and language runtimes, and it produced attacks measured in tens of millions of requests per second. Mitigations were not filters but limits: cap concurrent streams per connection, shorten timeouts, disable HTTP/2 where unneeded, and add computational challenges at the edge.

Do not dismiss denial of service as 'just unavailability'. The impact column in the slide is honest: a service that fails during exams, elections, ticket sales, year-end or a product launch does reputational and contractual damage out of proportion to the technical simplicity, and customers rarely distinguish an attack from an outage they should have prevented. Denial of Wallet weaponises the economics of elasticity: an architecture that auto-scales to absorb attacks also buys the attacker capacity, so abusive traffic against metered services such as pay-per-call APIs, transcoding, GPU inference or SMS gateways produces a bill instead of a crash. Finally, denial of service is often a companion, not the objective: industry studies have repeatedly found DDoS activity alongside intrusions, because flooding a monitoring pipeline and a busy help desk is an effective way to keep attention away from the real door. That is why dependent services belong in the analysis: exhausting an identity provider, a directory, a certificate service or a database can lock out every system that trusts it, without ever touching those systems.

Denial of Service (DoS)



MiTM – Man in The Middle Attacks

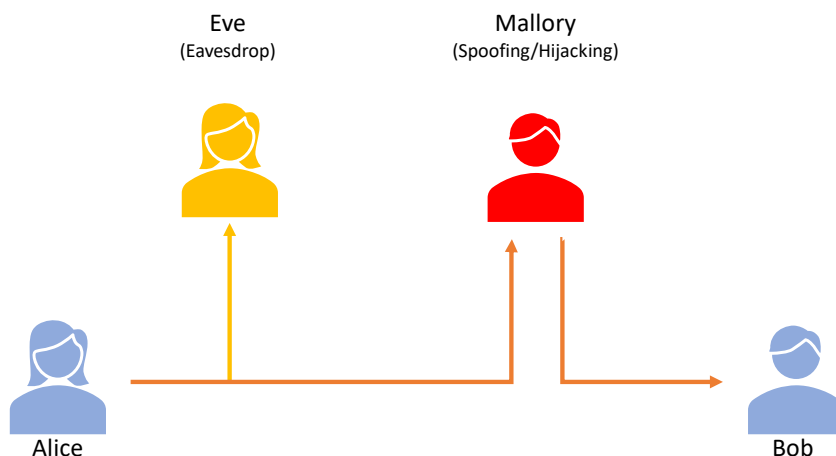
- Attacker puts themselves in the **middle of two communicating parties**
 - Using social engineering, misconfigurations or vulnerabilities
 - **Eavesdropping**: attacker passively listens to traffic
 - **Spoofing**: attacker fakes responses to questions (e.g. DNS Spoofing)
 - **Hijacking**: attacker actively mangles the communication
 - May steal data, inject packets or divert the communication
- Impact: CIA Triad is compromised
 - Communications not confidential
 - Communications payloads are manipulated/changed
 - Sessions are blocked or data is omitted

Man-in-the-middle attacks are best understood as a positioning problem rather than a protocol bug. The attacker does not need to break cryptography; they need to be in the path, after which the parties talk to each other through them. Positioning can be achieved on a local network by abusing the address resolution between IP and MAC (ARP replies are unauthenticated by design, so a host can be told that the attacker is the default gateway), by running a rogue DHCP server, by broadcasting an evil-twin access point with the same name as the legitimate corporate or conference network, by abusing switching equipment with MAC table flooding that forces the switch to behave like a hub, on the Internet by announcing someone else's prefixes in BGP, and inside an application by DNS manipulation, malicious browser extensions, malware, or a certificate issued by a trusted certification authority that should not have issued it.

The three modes in the slide differ in visibility and in what they cost the defender. Eavesdropping is passive and leaves almost no trace on either endpoint; against plaintext protocols it is trivial, which is the entire justification for the 'HTTPS everywhere' movement, and it remains possible even against encrypted links as traffic analysis (who talks to whom, when, how much). Spoofing means answering a question before the legitimate answer arrives, and any protocol whose trust model is 'whoever replied first, and replies look plausible' is vulnerable: DNS is the canonical example, which is why DNSSEC, DNS over TLS and DNS over HTTPS matter, though DNSSEC authenticates data from the zone and does not by itself protect a client from a lying local resolver. Hijacking is the aggressive mode: the attacker relays, modifies or blocks traffic, injects content into pages, captures credentials, plays session tokens, or silently redirects. With a valid position the attacker is not an observer, they are an endpoint, and they can show each party whatever they like.

The consequences for the CIA triad are therefore simultaneous rather than selective. Confidentiality is lost because the attacker sees the plaintext the endpoints exchange. Integrity is lost because payloads can be rewritten: credentials replaced, links swapped for malicious ones, invoice bank details edited, or a script injected into a page for every visitor. Availability is lost because the attacker can inject resets, delay or drop messages, and starve the session. History supplies the cautionary tales: a compromise of a certification authority in 2011 produced certificates used to intercept mail traffic of a national population; in 2018 a large social network stored full HTTPS URLs in its search index and thereby exposed session cookies of millions of users to internal and third-party systems; in 2023 a state actor was reported to have hijacked routing prefixes to divert traffic destined for well-known websites. Each of these required no cryptographic break, only position and trust.

MiTM – Man in The Middle Attacks



https://en.wikipedia.org/wiki/Alice_and_Bob

The characters in these diagrams are the standard notation of protocol thinking. Alice and Bob are the two honest parties; Eve is the passive eavesdropper, from 'eavesdropping'; Mallory is the active attacker who may alter, inject and fabricate messages, sometimes also called Trudy, from 'intruder'; Carol, Dave and others appear when the protocol has more participants. The convention comes from the 1978 paper by Needham and Thompson that introduced key distribution with named characters, and it survived because it lets you reason about a protocol without a single line of code or a single vendor name. When you read a protocol description as a sequence of Alice to Bob messages, the natural next question is always the same: which of these messages could be sent by someone else, and how would Alice know?

Read the panels as an escalation. The first is the reference case, a direct channel, which exists mainly as a hypothetical. The second shows passive interception: the same messages flow, Eve keeps a copy, and nothing on either endpoint changes; this is why detection of eavesdropping is a property of the network, not of the hosts. The third shows impersonation: Eve speaks to Bob pretending to be Alice and to Alice pretending to be Bob, so one session becomes two independent sessions controlled by the attacker; from that moment the attacker decides what each party sees, and can keep both parties in a consistent illusion indefinitely. The fourth shows alteration and disruption, where Mallory rewrites content, withholds messages or injects new ones, which is what turns an interception into fraud, malware distribution or censorship.

Now ask what changes when the channel is encrypted. With authenticated end-to-end encryption, a man in the middle loses the plaintext and the ability to modify payloads, but does not disappear: they can still block traffic, force a downgrade if the implementation accepts fallback, terminate the connection and hope the user clicks through a warning, or present a certificate that the client accepts because a trusted authority vouched for it. This is where real-world failures cluster: rogue or over-permissive certificate authorities, corporate TLS interception proxies that decrypt and re-encrypt traffic at a security gateway (common in email gateways, endpoint antivirus and web filters), and ad-injecting proxies shipped preinstalled on consumer devices. Interception proxies are legitimate tools with a heavy trust cost, and they quietly reintroduce exactly the assumption end-to-end encryption was meant to remove: that the intermediate boxes are honest.

Phishing Attack

- **Attacker uses fraudulent messages to trick victims**
 - Objective: provide information, exposing, download malware, pay for something
 - Social Engineering attack exploring human vulnerabilities
 - Messages resort to urgency, fear, curiosity, authority, greed
 - Subtypes:
 - **Spear Phishing**: crafted to trick a specific person
 - **Whaling**: senior executives/role-holders
 - **Smishing**: Uses SMS
 - **Vishing**: Uses phone calls
 - **Deepfake voice/audio vishing**: AI-generated phishing
- **Impact: Financial loss, damage to public image, compromise of other systems**

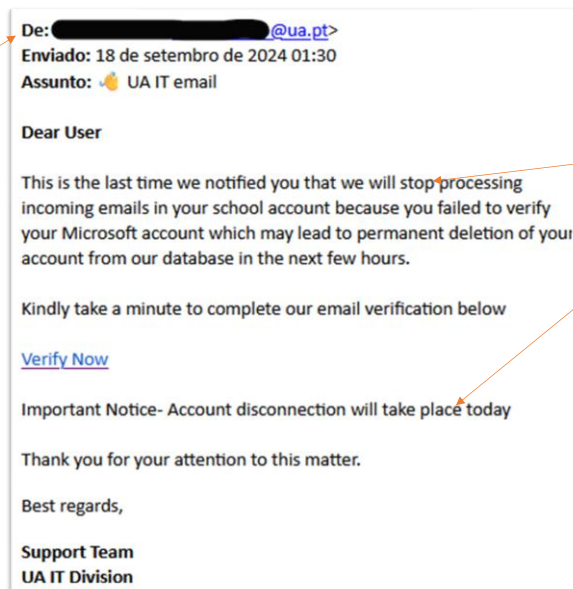
Phishing succeeds because it attacks a decision, not a system. The psychological levers listed in the slide are the classic influence mechanisms studied long before cybersecurity: authority (a message that looks like it comes from the finance director, the tax office, the registrar, the platform's own security team), urgency and scarcity ('your account will be suspended in 24 hours', 'the payment deadline expires today'), fear (a compliance violation, a disciplinary process, a suspended salary), curiosity (a shared document, an invoice you do not recognise, a delivery notice), greed ('refund', 'crypto giveaway', 'you have been overpaid'), and social proof ('your colleagues from the department have already confirmed'). All of them have one function: to compress the time available for reflection and to make the recipient act on the message's own framing. Notice that none of them requires a technical flaw; the only vulnerability being explored is attention under load.

The subtype names are useful because each one changes the economics of the attack. Bulk phishing sprays thousands of recipients and wins on volume, with conversion rates measured in thousandths of a percent. Spear phishing invests in open-source intelligence about one person, referencing real projects, real colleagues, real documents, and it typically arrives at a moment when the target is busy or travelling. Whaling aims at executives and role-holders, where a single approval moves large sums; the family of attacks known as business email compromise, in which the attacker impersonates a supplier or an executive to redirect a payment, consistently produces the largest reported financial losses of any internet crime category, well above malware in the annual FBI reporting. Smishing and QR phishing move the interaction to a channel with no mail gateway, no link rewriting and no sandbox, on a device where the user has no security awareness at all. Vishing calls the victim directly, often with cloned caller ID and a convincing script, and increasingly with cloned or synthesised voice; a single convincing phone call asking to 'confirm the transfer by approving the prompt you just received' has been enough to authorise fraud.

The most important technical development in this area is that modern phishing is often a proxy rather than a page. Attack kits host a faithful clone of a real login page behind a reverse proxy that relays everything to the genuine identity provider, so the victim types the correct password and even answers a one-time code: the kit replays it, obtains a session from the real service, and hands the attacker a valid session cookie. These are called adversary-in-the-middle kits, and they defeat one-factor authentication and most out-of-band codes in real time. Related variants abuse legitimate platforms: malicious OAuth consent screens that ask for mailbox access and look identical to a well-known application's permission prompt, device-code flows where the victim is asked to enter a code the attacker already holds, shared documents on real collaboration platforms (which is why 'it was a link from SharePoint' does not mean safe), and comments on public code hosts or social networks to host the lure. Deepfake audio and video are the newest increment, mostly used to add credibility to a vishing call or a video call asking for an urgent transfer or a password reset, and detection is unreliable.

Phishing Attack

Authority and trust



Urgency and fear

The annotated message is worth dissecting line by line, because the elements that make it convincing are the same elements a defender can learn to recognise and that a security awareness session should point at. Urgency and fear are manufactured through consequences that fall on the reader (loss of access, loss of money, a penalty, a deadline), and through a false sense of precision: a reference number, a date, an account fragment, a system name in small print. Authority is manufactured through identity: a sender who appears to be a person with power over the reader, official logos, formal register, legal or compliance vocabulary, and the institutional design language copied pixel by pixel from the real organisation.

When you examine any message, the questions that actually matter are structural rather than cosmetic. Is the sender domain one the organisation owns, and does it pass alignment? (a display name can say anything it likes, and 'on behalf of' headers are perfectly legal). Was the reader expecting this interaction, and would the real process work this way? Does the message ask for something that a real institution would ask by email at all, such as a password, an approval, a payment to new bank details, or the confirmation of a login attempt? Where does the link actually go, and where does it go after the redirects, which you can inspect by hovering, by copying the URL without clicking, or by asking a colleague to check? Is there any independent way to confirm the request through a channel the attacker does not control?

Note also what the annotation implies about detection technology. A message with no malicious attachment, no known malicious URL, no exploit and a well-formed sender can be perfectly clean for every filter in the chain, because nothing in it is technically malicious; it is only malicious in combination with a gullible or distracted human and a process that allows the request to be honoured. This is why phishing defence is a process problem as much as a technical one. Financial processes need dual control and a callback on a known number before changing payment details. Identity processes need an approval path that cannot be satisfied from inside the message itself. Helpdesk processes need identity verification that does not rely on the caller knowing a phone number or an employee ID.

A last point for discussion: the responsibility question. Students often conclude that users are the weak link, and then stop thinking. In a well-run organisation the answer is that the user is one control among many, and the ones that don't depend on attention should carry most of the load: phishing-resistant factors, blocking at the transport and DNS layers, restricting what an authenticated session can do, limiting the blast radius of a compromised mailbox, and rehearsing the incident so the person who did report knows exactly what to do in the following five minutes. Awareness training has measurable value in reporting rates, and very little value in click rates; that asymmetry should shape how the training budget is spent.

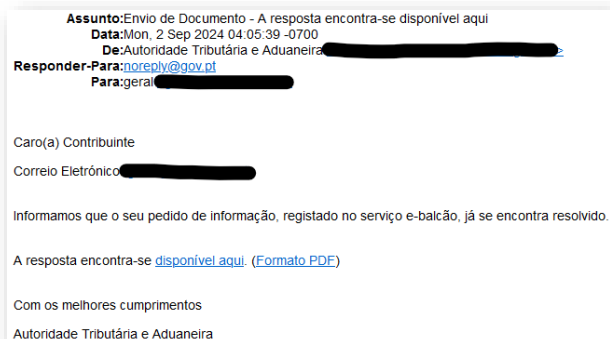
- Infect systems with malicious software
 - Using social engineering or **software vulnerabilities**
 - Variations:
 - **Virus**: Require some host to exist (binary file, document)
 - **Worm**: Isolated program that can run without others
 - **Trojan**: Disguised of another application (popular with keygens/cracks)
 - Initial payload is usually a dropper to further stages
 - Recently, it is a preconfigured Remote Access Tool
- Impact: Financial loss, information loss, compromise of other systems, participation in attacks
 - Infostealers are a dominant trend (Bank Fraud)

The three classical categories are distinguished by how they obtain a host and how they move, and the distinction still matters for containment. A virus needs a host object: it attaches to or replaces a binary, a document macro, a boot sector, a script in a build pipeline, and it propagates when that object is executed or opened. A worm is self-contained and propagates by itself across a network path, needing no user action beyond the existence of a reachable service; that single property makes worms an availability and containment problem rather than a user-education problem, and it is why historical worms spread around the world in minutes. A Trojan does not propagate at all: it is a disguise, delivered once, usually with the user's cooperation, because the user wanted whatever was on the other side of the download.

The slide's remark about keygens, cracks and cracked licences deserves emphasis, because it is one of the clearest examples of a self-selected victim population. Software that is already pirated arrives with no integrity expectations, no code signing to verify, and often with the user actively disabling antivirus because the crack 'triggers a false positive'. The same pattern appears in fake installers for popular applications, malicious browser extensions, fake updates (a browser plugin marketed as a video downloader, a fake Teams or Zoom installer during remote-work peaks), fake tech-support software, and poisoned open-source packages: typosquatted names in package registries, dependency confusion where an internal package name is published publicly, and malicious GitHub Actions or container images. In all these cases the delivery vector is not a vulnerability in the product but a trust decision made by a person or a build system.

The 'initial payload is a dropper' point describes the standard structure of modern intrusions. The first stage is small and disposable: a macro document, a shortcut that runs a shell command, a signed but abused binary, an installer. Its only job is to fetch and set up what comes next, which decouples the delivery from the objective and lets the operator change tools between the intrusion and the payload. The most common settled form today is a preconfigured remote access tool: something that gives interactive control, persistence, a channel out, and the ability to run whatever the operator decides later. Some of these tools are legitimate commercial products abused after licence theft or cracking, which makes them extremely hard to block cleanly, since the same binary appears on real customer machines. Add to this the trend of fileless operation, where the payload lives in memory and executes through legitimate interpreters and system utilities, and the appearance of vulnerable-but-signed drivers used to disable protections, and you see why 'detect the malicious file' is a losing strategy.

Consequences are broader than the infected machine, and the slide lists the four that matter. Financial loss from fraud and from the response itself. Information loss when credentials, mail and documents are collected. Compromise of other systems, because a single workstation is a foothold in a directory with paths to servers. And participation in attacks, where the infected machine becomes a proxy for relaying phishing, a scraper, a botnet node or a launch point against third parties, which turns the incident into your liability towards other organisations. Infostealers, which harvest everything with a session: browser profiles, cookies, saved passwords, autofill, wallet files, SSH keys, MFA extensions, screenshots, are currently the dominant commodity, precisely because they monetise at scale.



Notificação Eletrónica

Exmo(a). Senhor(a),

Vimos, pela presente, comunicar a V. Ex.^a que, através do endereço geral@... foi identificado um registo de dívida ativa junto da Autoridade Tributária e Aduaneira, no montante global de **5.941,58 EUR**.

Para consultar os detalhes do processo e proceder à respetiva regularização, deverá aceder através da ligação infra:

 **Consultar Documento**

Esta notificação eletrónica tem validade legal nos termos do Decreto-Lei n.º 93/2025. A falta de regularização poderá originar acréscimos legais previstos na legislação em vigor.

Autoridade Tributária e Aduaneira – Sistema de Notificação Eletrónica
Comunicação automática. Por favor, não responda a este endereço de correio eletrónico.

• <https://www.joesandbox.com/analysis/1513069>

<https://www.joesandbox.com/analysis/1783749>

João Paulo Barraca, André Zúquete

SIO

13

The two screenshots are related to sandbox analysis reports, and they illustrate the standard workflow for dealing with suspicious files when you cannot and must not open them on your own machine. A sandbox is an instrumented, isolated environment where the sample is executed and everything it does is recorded: processes created and their command lines, files written and dropped, registry changes, network attempts (including domains that fail to resolve, which is often the most valuable artefact), mutexes, scheduled tasks, services, and injected code. Analysis is both static, before execution, and dynamic, during and after execution. Static analysis gives you the file's shape: hashes, cryptographic signatures and their issuer, compile timestamps, embedded strings, imported functions and APIs that hint at capability (debugging, key capture, credential reading, network sockets), packed or obfuscated sections, and version metadata that may be spoofed.

The value of the report view shown here is not the pretty graph but the indicators it lets you harvest. From a single execution you can extract hashes of the sample and of its dropped files, domains and IP addresses with the times contacted, URLs used for downloads, certificate details of command-and-control endpoints, wallet addresses in ransom notes, and behavioural signatures that can be turned into detection rules. Those indicators are then correlated: the same infrastructure may appear in other samples, in threat intelligence feeds, in your own proxy or DNS logs, which answers the question that matters most after an alert, 'is this the only machine that saw it, and how long ago?'. Modern sandboxes also map observed behaviour onto a tactic-and-technique matrix automatically, which lets the analyst talk to the incident response team in the language of initial access, persistence, privilege escalation, defence evasion and exfiltration instead of a list of syscalls.

Two families of caveats belong in any honest lecture on malware analysis. The first is that sandboxes are increasingly avoided by the sample. Malware routinely checks for the tell-tale traces of virtualisation and analysis before doing anything interesting: the presence of virtualisation drivers and guest tools, disk size and total installed entropy, mouse movement and keyboard activity, the number of processes, the existence of specific usernames or screen-recording tools, sleep delays that outlast a short analysis window, and even checks on the ambient network. A sample that exits immediately tells you it was analysed, and nothing else. The second caveat is safety and legality: real malware analysis requires isolated networks without route to production, restricted egress so that a live sample cannot be used to attack a third party from your infrastructure, and evidence handling if the analysis is destined for legal or disciplinary proceedings.

Ransomware

- **Malware that blocks access to system until a ransom is paid**
 - Explores social engineering or **software vulnerabilities**
 - Social engineering such as phishing
 - Software vulnerabilities allow fast propagation across systems
 - Facilitated in systems with no defenses or perimeter defenses
 - **Cryptolockers:** Encrypts user data and requests a ransom to release a key
 - Double Extortion: Data is exfiltrated and may be released to the public if there is no payment
 - Triple Extortion: All above plus blackmailing other companies/clients or launching secondary attacks
 - Frequently operated in a Ransomware-as-a-Service model, with initial access broker markets
- **Impact: Denial of Resources, Data loss, Severe disruption**
 - Considered as extremely dangerous as operations may be disrupted for a long time
 - Recovery will require 3-2-1 with immutable/air-gapped copies and tested restores
 - WORM backups (Write Once, Read Many)

Ransomware is the industrialised combination of everything else in this lecture, which is why it deserves its own slide and why it is considered the most dangerous category. The technical core, encrypting data and demanding payment, is almost trivial cryptography: a symmetric key per file or per volume (fast), wrapped by an asymmetric key pair so that only the attacker's private key can recover it. The difficulty for the victim is that there is usually no exploitable weakness in the crypto: recoverability depends on having a key or a backup, not on the encryption being weak. What makes ransomware a business is everything around the cipher, and that 'everything' is what the bullet list describes: initial access, privilege escalation, lateral movement, data staging and theft, destruction of defences, and negotiation.

Initial access comes from the same doors as every other major incident: a phished credential, an exposed remote desktop service with a weak password, a vulnerable edge device such as a VPN concentrator or a remote access product, an unpatched public application, or the purchase of access from somebody who already had it. That last route deserves attention because it explains the speed of modern incidents: an entire market of initial access brokers sells footholds, sometimes weeks old, with mapped Active Directory and stolen credentials included. From there the operator escalates, harvests credentials from memory and from stored configurations, moves laterally over administrative shares and remote services, disables endpoint protection and volume shadow copies so previous versions cannot be used, and locates backups before encrypting anything. Speed matters because the defenders' clock is the detection time: the well-known statistics on dwell time show that the intruders who reach encryption had often been inside for days or weeks, and had already been seen by tools whose alerts nobody closed.

The extortion models escalate because the market competes. Single extortion encrypts and asks for money, but victims with backups can refuse. Double extortion exfiltrates first and threatens publication, which means refusing to pay now carries a cost of its own: personal data, medical records, client contracts, internal correspondence. Triple extortion extends pressure to the parties around the victim: clients, partners and suppliers receive notices that their data is in the attacker's hands, or are attacked themselves, and secondary denial of service may be used while negotiations stall; in some variants the attacker also leaks employee data or contacts journalists, because a reputational clock is more effective than a technical one. Ransomware-as-a-service structures the whole supply chain, with affiliates doing the intrusion, developers taking a share of each ransom, and shared infrastructure for leaking sites, negotiation chat and even 'customer support' desks. The consequence is that the threat is not a small number of groups but a diffuse ecosystem with onboarding materials, and that dismantling one group does not reduce the number of operators looking for a foothold.

Password

- Attacks targeting the discovery of passwords
 - Explore social behavior (password reuse) and **software vulnerabilities**
 - Some common types:
 - Brute force: login attempts testing all possibilities
 - Dictionary attack: testing common words
 - Spraying: one/few passwords against many accounts
 - Keylogging: intercepting keys using malware
 - Infostealers: generic log stealers that sell data in Combo Lists
 - Credential phishing and Token theft
- Impact: financial loss, system compromise, information loss, impersonation

1. 123456
2. admin
3. 12345678
4. 123456789
5. 1234
6. 12345
7. password
8. 123
9. Aa123456
10. 1234567890
11. UNKNOWN
12. 1234567
13. 123123
14. 111111
15. Password
16. 12345678910
17. 000000
18. Admin123
19. *****
20. user

Most common passwords

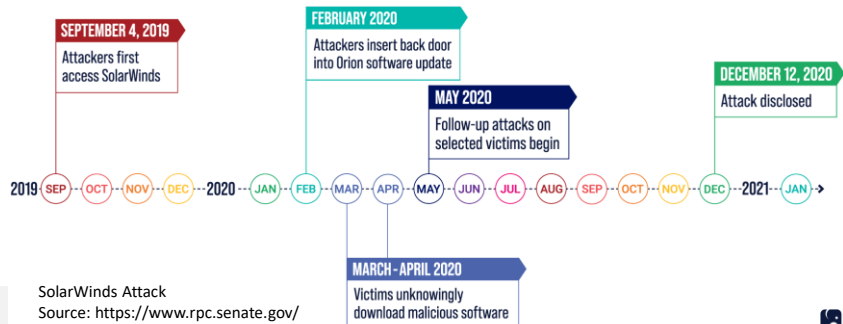
Passwords persist because they are the only authentication factor most systems have ever truly controlled: something you know, cheap to issue, cheap to reset, and impossible to physically steal in bulk. Their weakness is that they are shared secrets with no way to limit the number of guesses, no revocation other than rotation, and a user population that optimises for recall across dozens of accounts. The list in the slide is a top-of-frequency list from breached corpora, and its content is a lesson in itself: sequences, the word 'password' and its variants, keyboard patterns, default account names, and the interesting entry meaning 'unknown or empty'. Users in every country behave alike under cognitive load, which is why a list like this is a legitimate first filter for any organisation, and why '123456' remains near the top two decades after it was first observed.

Attackers exploit the human pattern rather than trying random strings. Dictionary attacks test words and their mutations (letter substitutions, appended years, capitalisation, common suffixes), and rule-based engines expand a hundred million base passwords into billions of candidates cheaply. Credential stuffing does not guess at all: it replays username and password pairs from other sites' breaches, betting on reuse, and it succeeds at disturbing rates because a user who reused a password on a forum that was breached in 2015 has almost certainly reused it somewhere that matters. Password spraying inverts the target set on purpose: one or two common passwords against thousands of accounts, which evades the per-account lockout that defenders configure, and which is the standard technique against Outlook Web Access, VPNs and remote desktop gateways. Keylogging malware and infostealers bypass the password entirely by taking what the browser or the operating system keeps for convenience: saved credentials, but more importantly live session cookies and refresh tokens, which are valid even when the password is strong and even after rotation if sessions are not revoked.

The organisational policy questions follow from the same physics. NIST's guidance abandoned most composition rules and expiry mandates, because forced rotation and complexity requirements produce written-down, minimally-varied passwords; instead it recommends generous minimum length (length dominates entropy per character), checking candidate passwords against corpora of breached and common passwords at the moment of creation, screening user accounts against breach data, and allowing passphrases and password managers. A

Insider and Supplier Threats

- An insider that uses their authorized access or understanding of an organization to harm that organization
 - **Collaborators**: disgruntled, subverted or simply malicious
 - **Suppliers** or **contractors**: create a variant named: Supply Chain Attacks
 - Explore weaknesses in a Perimeter Defense Model
 - Insiders have wide access to resources
 - Can be used to escalate attack to **other organizations**
 - Compromising a software company will potentially compromise their clients
- Impact: Brand, Information, Total disruption



João Paulo Barraca, André Zúquete

Supplier and contractor access turns the insider problem into a graph problem, because a supplier's access is another organisation's insider. Every managed service provider, outsourced helpdesk, integrator with remote support tooling, audit firm with data access, and cloud service with administrative reach is a path from a third party's security posture into your systems, and the aggregate effect is that your effective security level is that of your least well-managed critical supplier. Historical incidents make the point better than any policy: a large retail breach began with credentials obtained from a third-party vendor for climate control equipment; a helpdesk impersonation attack against a support contractor of an identity platform led to customer data being taken; and a compromised managed service provider delivered ransomware to more than a thousand of its clients at once, because its remote management tool was the trusted path into all of them. The lesson is not 'avoid suppliers' but 'bound supplier access narrowly, monitor it as first-class traffic, and ask the hard questions in procurement'.

Injection

- Exploration of a vulnerability allowing Injection of code into a program or query
 - Code is later executed in server or other clients
 - Code is an SQL statement, Javascript/python/bash/powershell/html/css code, Binary instruction, **LLM Prompt...**
 - Targets Databases, Web applications, binary applications...
 - Due to improper handling of untrusted data which is accepted and later used
- Impact: data loss, total system compromise
 - Specific technique: Remote Code Injection – System runs new malicious code provided by the attacker
 - Popular attacks: Log4Shell, xz-utils takeover, MOVEit Breach

Query: `SELECT * FROM users WHERE username = '%u' AND pass='%p'`

Arguments: `u="admin" and p="qwerty"`

Result: `SELECT * FROM users WHERE username = 'admin' AND pass= 'qwerty'`

But if: `u="admin' or '1'='1' - "`

Result: `SELECT * FROM users WHERE username = 'admin' or '1'='1' -- ' AND pass...`

Always true

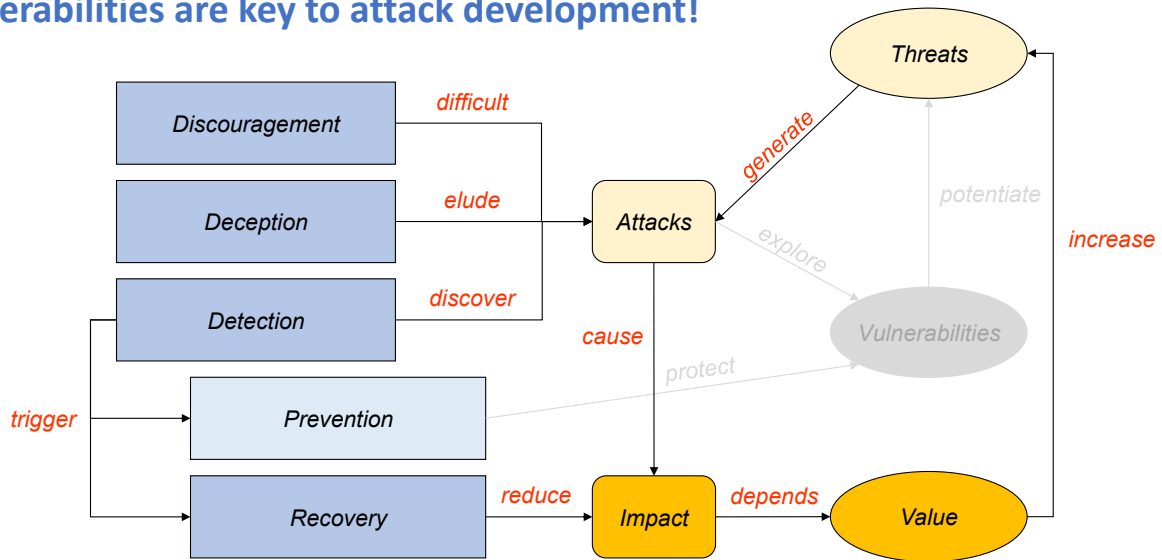
Comment.
Ignores what follows

Every injection attack has the same root cause, and it is worth stating once so it can be recognised everywhere: the system builds a message for another interpreter by concatenating trusted instructions with untrusted data, and the receiving interpreter cannot tell which is which. Once the data can change the structure of the message, the attacker's intent becomes part of the program. That single sentence covers SQL, command execution, LDAP and XPath queries, NoSQL operators, HTML and JavaScript in a page (the family of cross-site scripting), XML entity expansion, template languages, deserialised object streams, binary payloads that overwrite a return address, and prompt injection in a large language model, where untrusted text is concatenated with the instructions that govern the assistant. The interpreter changes, the failure does not. The local remedy differs with the sink: for the browser, context-aware output encoding with a content security policy and httpOnly and SameSite cookies as a backstop; for shells, calling the process with an argument vector rather than a command string; for XML parsers, disabling entity resolution; for object streams, refusing to deserialise untrusted data at all; for binaries, compiler and platform hardening.

Work the example in the slide until you can do it in your head. The query wraps the user name and password in quotes. Supplied normally, the values become literals and authentication depends on both matching. If the user name is submitted as admin followed by a single quote, an 'or' condition that is always true, and then a comment marker, the string closes the literal, adds a condition that every row satisfies, and comments out the rest of the statement, so the query returns rows without knowing any password. From there the escalation in real engagements is mechanical: discovering which columns are returned to the page and can be used to display data, extracting the schema and the tables, reading files or writing them where the database user has rights, and in some configurations obtaining code execution on the host. Even when the result is not visible, the attacker still has a channel: boolean-blind testing, where the page differs between true and false answers, timing, where a delay betrays a true condition, and out-of-band channels such as DNS lookups that carry data out to an attacker-controlled domain.

Information Security

Vulnerabilities are key to attack development!



The same map is shown again, now with the vulnerabilities node highlighted, and the emphasis is deliberate: if you want to prevent attacks, **vulnerabilities** are the highest-leverage node in the graph, because they are the only one of the four that you can legitimately remove from your own systems. You cannot delete a threat actor, you cannot un-happen the value of your assets, and impact is a consequence rather than a cause. But you can decide not to run the component, to patch it this week rather than next quarter, to disable the feature nobody uses, and to design the system so that a class of weaknesses cannot exist. Every defensive budget allocated to vulnerabilities buys more than the same budget spent chasing threats.

The map also implies something uncomfortable: as you become more attractive, you become more vulnerable in the statistics. Value attracts research attention, and research attention produces published findings, so a growing organisation, a product that gains market share, or an institution that starts paying bounties will see its numbers rise. The rise is not proof of degradation; it is proof of visibility. The genuinely bad signal is different: a rising count of unpatched findings in the installed base, a rising time-to-patch, and a rising count of findings in components you did not know you were running. Those three are measurable from your own inventory, and they belong on management dashboards, whereas the total number of CVEs in the world is only useful as background noise.

Finally, think about the two directions of time in this diagram. Attacks explore vulnerabilities that exist today; vulnerabilities in the products you are about to deploy are created today, in design and in code. Defence against tomorrow's attacks is therefore partly a development discipline (secure design, memory safety, authentication and authorisation by construction, dependency hygiene, testing), and partly an operations discipline (inventory, prioritisation, patching, monitoring for exploitation). The rest of this lecture follows exactly that split: first the naming and tracking machinery that makes vulnerabilities discussable objects, then the weaknesses and the catalogues that let us reason about causes rather than symptoms.

Vulnerability tracking

- During the development cycle, vulnerabilities are handled as bugs
 - May have been handled by a security team or not
 - May have a **security classification**, **priority** and **time to be handled**
- When software is available, vulnerabilities are **also tracked at a wider scale**
 - For every system and software publicly available
- Public tracking helps...
 - focusing the discussion around the **same issue**
 - Ex: a dependency that is used in multiple applications or distributions
 - defenders to easily **test their systems**, enhancing the security
 - (attackers to easily know what vulnerability can be used to a given system)

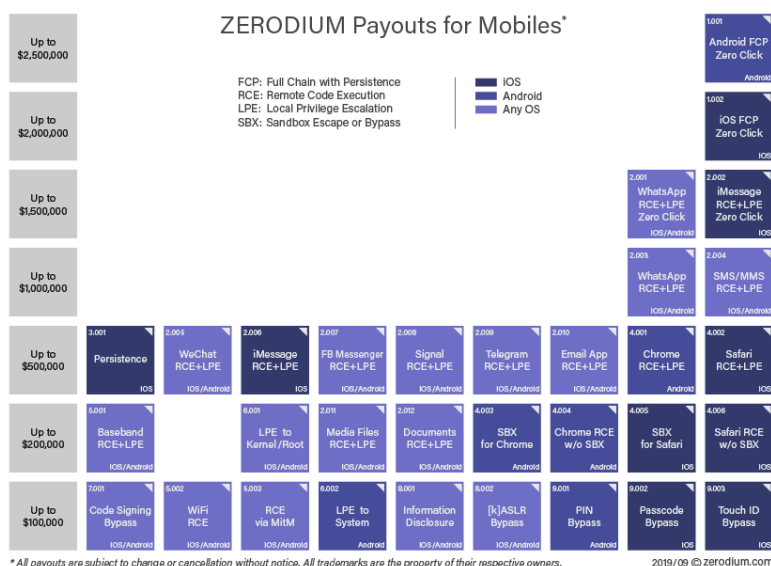
Inside a development organisation, a security defect is a record in a tracker, indistinguishable in form from any other defect, and that similarity creates the classic incentive problem. A crash in a rare path competes with the feature deadline and usually loses; a vulnerability competes with the same deadline but has no visible user, so it is deferred until it becomes embarrassing. Mature organisations break the tie with process: a security champions or review function that triages reported issues, a severity scheme with defined meaning, a 'bug bar' that states which severities must be fixed before a release can ship, embargo handling so the fix and the disclosure are synchronised, and time-bound service levels with escalation. Where there is no security team, the report arrives as a support ticket, and the outcome is unpredictable; where there is one, the interesting question becomes who owns the risk decision to defer, and whether that decision is written down somewhere a customer or an auditor could find it.

Once software is published, the tracking problem changes character: the same defect now exists in many places at once, in many versions, in many combinations, and the parties involved do not share a database. This is the coordination problem that public identifiers solve. A dependency used by a thousand applications and ten distributions produces, without a common name, a thousand conversations that cannot be joined: a thousand advisories, a thousand scanner rules, a thousand customer questions phrased differently, and no way to measure who has fixed it. With a common identifier, an advisory, a patch, a scanner signature, a compliance check and an internal asset query can all point at the same object, and the ecosystem can compute the things managers actually need, such as 'how many of our systems still run the affected version'.

The slide lists the benefits of public tracking, and the honest version includes the cost: the same list is a shopping guide. An attacker with an identifier can find the affected versions, read the analysis, locate a proof of concept, and build an exploit. Two observations decide the balance in practice. First, secrecy does not remove the exploit, it only removes the patch: the exploit market exists precisely because unpublished weaknesses are traded, and uncoordinated silence favours whoever is already looking. Second, the asymmetry between defenders and attackers is not knowledge but time: the attacker only needs one path, whereas the defender needs all of them, so public knowledge helps only if the organisation acts on it. Public tracking converts a defensive advantage that would otherwise be theoretical (you can be patched before you are attacked) into a manageable obligation, and it is also the substrate of every compliance and insurance question you will ever answer.

Vulnerability tracking

There is even a market



João Paulo Barraca, André Zúquete

SIO

21

The economic fact behind this slide is simple: a previously unknown weakness has market value, and value attracts supply. On the legal side of the market sit vendor bug bounty programmes and platforms, which pay for responsible reports and whose prize tiers have tracked the escalation of what is considered valuable: from a few hundred dollars for a web bug to tens of thousands for a remote code execution, and to seven figures for a full, zero-click chain on a mobile platform. Public contests have made the prices explicit for two decades, and one long-running contest alone has paid out hundreds of millions of dollars to researchers, which is a striking statistic when you consider that it purchases the right to see a vulnerability before anyone else does.

On the other side of the market sit intermediaries whose business is arbitrage: buying weaknesses quietly and reselling them to clients who will not publish them. Leaked correspondence from commercial surveillance vendors has shown published price lists by platform and by capability, together with support contracts, exploit updates and even after-sales arrangements for keeping implants working across OS upgrades. Reported prices for a zero-click mobile compromise have reached seven figures, and the clients are law enforcement, intelligence agencies and, in documented cases, authoritarian regimes whose use of the tooling against journalists and opposition figures became a diplomatic problem. Export-control regimes responded first by classifying intrusion software as controlled dual-use goods, and later by tightening the rules at European level, but the practical reality is that a small number of researchers and a handful of brokers determine which weaknesses are patched first.

There is also a much larger, much less glamorous market that touches every organisation directly. Stolen credentials sold in bulk, infected machines rented as access, initial access sold to ransomware crews, malware leased as a service with a support desk, phishing kits rented with a dashboard, log-cleaning and hosting services, and leak-site infrastructure offered to affiliates. This is the market that turns an unpatched VPN appliance or a weak remote desktop password into a commodity with a price list, and it explains the operational speed of modern incidents: the intruder does not need to be good at intrusion, because intrusion is procured. Understanding this changes how you argue for controls internally. 'Patch the edge devices' is not an abstract hygiene recommendation; it is a refusal to participate in a market where your perimeter is being advertised for sale.

CVE - Common Vulnerabilities and Exposures

What it is

- Dictionary of **publicly known information security vulnerabilities and exposures**
 - For vulnerability management
 - For patch management
 - For vulnerability alerting
 - For intrusion detection
- Uses **common identifiers** for the same issue in given application (e.g: CVE-2024-1234)
 - Enable data exchange between security products
 - Provide a baseline index point for evaluating coverage of tools and services.
 - Details about a vulnerability can be kept private
 - Part of responsible disclosure: until owner provides a fix

It is important to be precise about what CVE is, because the name suggests a knowledge base and it is closer to a registry of names. A CVE entry is an identifier, a short description, and references. It is deliberately thin: it does not contain a guaranteed technical analysis, a severity score, a fix, a workaround, or a statement about whether you are affected. Those live elsewhere, in the advisories of the vendor that made the software, in the national vulnerability databases that enrich entries with scores and configuration details, in distribution security notices, and in the researcher's own write-up. The value of the thin layer is exactly what makes it scalable and interoperable: a short, stable name that anything in the ecosystem can cite, from a patch note to a contract to a scanner rule.

Consider the four uses in the slide from the point of view of the tools that implement them. Vulnerability management means matching what you run against what is known, which is the matching problem: your inventory must express products and versions in a way that can be compared with the affected ranges in the advisory, and that comparison breaks on backported fixes, on vendors that keep a version number unchanged while applying security patches, on components renamed or re-vendored inside larger products, and on the difference between 'installed' and 'reachable'. Patch management means knowing what to install and when, and identifiers let you build service levels and measure them. Alerting means telling the right team about the right system at the right time; a name is the join key. Intrusion detection means writing a signature for a known weakness, and rules have historically referenced CVE identifiers precisely so an analyst who sees an alert can read what it means and what is patched.

The 'baseline index point for evaluating coverage of tools' is a subtle and very useful property. If a product claims to protect against known vulnerabilities, the only way to test the claim is to ask which identifiers it knows, and to compare across products on the same reference. That turns procurement into something closer to measurement: this scanner detects this fraction of the identifiers relevant to our estate, this endpoint product's signatures cover these, this patch service supports these platforms. It also explains a failure mode: when the reference layer degrades, everything downstream degrades. If the enrichment of identifiers into scores and configurations slows down, scanner feeds slow, prioritisation loses its inputs, and organisations fall back to guessing from names and dates, which is exactly what happened when the main national enrichment workload grew far beyond the staff available to process it.

The final bullet is the one that reconciles CVE with responsible disclosure. An identifier can be reserved before anything is public: the researcher or vendor obtains a name, keeps the details private while the fix is prepared, and the record becomes published when the advisory is issued. Embargoes exist so that the public appearance of a name coincides with the availability of a patch, and their duration is negotiated rather than absolute. There are legitimate reasons to keep a record private beyond the vendor's own fix: active exploitation in the wild, victims who need warning before the wider announcement, and weaknesses in systems whose failure endangers people. The general principle is that the identifier is a synchronisation signal for the ecosystem, and every decision about when to pull the trigger on publication is a trade between the safety of silence and the safety of information.

CVE Identifiers

...aka CVE names, CVE numbers, CVE Program Records

- Unique, common identifiers for publicly known information security vulnerabilities
 - Published on acceptance
 - Can be corrected/withdrawn afterwards
- Format
 - CVE identifier number (CVE-Year-Number)
 - Brief description of the vulnerability or exposure
 - References to extra information

🚩 CVE-2024-23934 Detail

RECEIVED

This vulnerability has been received by the NVD and has not been analyzed.

Description

Sony XAV-AXS500 WMV/ASF Parsing Stack-based Buffer Overflow Remote Code Execution Vulnerability. This vulnerability allows remote attackers to execute arbitrary code on affected installations of Sony XAV-AXS500 devices. User interaction is required to exploit this vulnerability in that the target must visit a malicious page or open a malicious file. The specific flaw exists within the parsing of WMV/ASF files. A crafted Extended Content Description Object in a WMV media file can trigger an overflow of a fixed-length stack-based buffer. An attacker can leverage this vulnerability to execute code in the context of the device. . Was ZDI-CAN-22994.

Metrics CVSS Version 4.0 CVSS Version 3.x CVSS Version 2.0

NVD enrichment efforts reference publicly available information to associate vector strings. CVSS information contributed by other sources is also displayed.

CVSS 3.x Severity and Vector Strings:

 NIST: NVD

Base Score: **9.8**

NVD assessment not yet provided.

 CNA: Automotive Security Research Group (ASRG)

Base Score: **9.8**

Vector: CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:H

References to Advisories, Solutions, and Tools

By selecting these links, you will be leaving NIST's website. We have provided these links to other web sites because they may have information that would be of interest to you. No inferences should be drawn on account of other sites being referenced, or not, from this page. There may be other web sites that are more appropriate for your purpose. NIST does not necessarily endorse the views expressed, or concur with the facts presented on these sites. Further, NIST does not endorse any commercial products that may be mentioned on these sites. Please address comments about this page to nvd@nist.gov.

Hyperlink	Resource
https://www.sony.com/electronics/support/mobile-cd-players-digital-media-players-xav-series/xav-ax500/software/00274156	
https://www.zerodayinitiative.com/advisories/ZDI-24-875/	

The syntax carries more information than it appears to. The year component is the year in which the identifier was reserved, not the year the bug was introduced, discovered or exploited, so it is entirely normal to see a recent identifier for code written a decade ago, and normal for a numbering authority to reserve a block of names in advance. The numeric part is a counter within that year, padded to a minimum width and allowed to grow past four digits when needed; it is not a sequence of importance, and the numeric value has no meaning about severity or about the order of publication. Any heuristic that treats 'higher number means newer problem' is wrong, and any argument that a CVE with a small number is less serious is also wrong.

The lifecycle states matter more than the format. An identifier moves from reserved to published, and it can end up rejected, which happens when the issue turns out not to be a security vulnerability, when it duplicates an existing record, or when the assignment was withdrawn. Rejected records continue to circulate in feeds, blog posts and vendor advisories that were drafted before the rejection, and they cause real confusion, including in compliance reports. A record can also be marked disputed, which is the mechanism that lets the party affected by the report state publicly that it disagrees with the classification, for example arguing that the behaviour does not violate the product's security policy, or that the reporting party published before the agreed date. Corrections are possible after publication, and a record's description can be rewritten; the identifier stays, which is the point of a stable name.

Reading a record well is a small skill worth practising. Look first for who assigned it: the vendor's own numbering authority, a distribution, a coordinated disclosure centre, or a third-party researcher organisation. The assigner determines how much care went into the description, whether the affected version ranges are trustworthy, and whether the fix information will ever appear. Look for machine-readable fields: the affected products and version ranges, the weakness classification drawn from the catalogue of weaknesses, and the score if one was provided by the assigner. Then follow the references, because the substantive material is usually there: the vendor advisory with the fixed version, the distribution notices with their backported fix versions, the researcher's blog post with the analysis and, sometimes, the proof of concept.

And the most important reading rule: a CVE is a claim about the world, not a measurement of your situation. Deciding whether it applies to you requires knowing what you run, how it is configured, whether the vulnerable code path is compiled in or reachable, whether a compensating control already blocks it, and what would happen if it were exploited. That last judgement is yours and not the database's, and it is exactly the judgement that the structured 'exploitability' statements now used in software bill of materials documents were designed to record, so that a supplier can say 'that identifier does not apply to this product in this configuration' in a form a tool can consume. Without such statements, the only two options are to patch everything blindly or to argue by email.

Definition: Vulnerability

A mistake in software that can be directly used by an attacker to gain access to a system or network

- A mistake is a vulnerability if it allows an attacker to use it to violate a reasonable security policy for that system
 - This excludes entirely "open" security policies in which all users are trusted, or where there is no consideration of risk to the system
- A CVE vulnerability is a state in a computing system (or set of systems) that either:
 - Allows an attacker to execute commands as another user
 - Allows an attacker to access data that is contrary to the specified access restrictions for that data
 - Allows an attacker to pose as another entity
 - Allows an attacker to conduct a denial of service

MITRE/CVE-project definition

The definition is doing three jobs at once, and separating them makes it much easier to apply. First, it ties 'vulnerability' to a policy: a mistake is a vulnerability only if it allows a violation of a reasonable security policy for that system. This is why the same behaviour can be a vulnerability in one product and a feature in another. Reading another user's data is a vulnerability in a multi-user operating system; a single-user tool with no confidentiality model has no such policy to violate. A local administrative user being able to read anything is not a vulnerability, because the policy already grants that. A missing restriction in a system whose documented model promises that restriction is a vulnerability, even if nobody can exploit it today.

Second, the definition excludes 'entirely open' policies, which is the sentence that prevents the term from collapsing into every imperfection. If a device is designed to be opened and used by anyone present, or if a service is explicitly intended to be anonymous and unauthenticated, then behaviours that would be alarming elsewhere are simply the design. The exclusion also warns against the opposite abuse: declaring a policy 'open' after the fact in order to argue that an uncomfortable finding is not a vulnerability. That is why the qualification is 'reasonable' and why the assignment process expects the affected product's own security model to be described, not invented at the moment of the dispute.

Third, the four states in the CVE-oriented formulation are a practical triage test, and they are worth memorising as a checklist because they cover the space surprisingly well. Executing commands as another user (any code-execution or privilege issue that lands you as somebody else). Accessing data contrary to the access restrictions for that data (read disclosures, information leaks, directory traversal, mis-scoped cloud buckets, secrets in a log). Posing as another entity (spoofed identity, forged tokens, authentication bypass, weak session handling, or the ability to impersonate a service). Conducting a denial of service (any failure mode that removes availability from those entitled to it, including resource exhaustion, amplification abuse and control-plane crashes). If a candidate finding satisfies none of the four, ask hard questions about what security property it touches at all; if it satisfies one, ask who can reach it and what they become.

Definition: Exposure

A configuration issue or a mistake in software allowing access to information or capabilities used as a stepping-stone into a system or network

- A configuration issue or a mistake is an exposure **if it does not directly allow compromise**
 - But could be an important component of a successful attack, and is a violation of a reasonable security policy
- An exposure describes a state in a computing system (or set of systems) that is not a vulnerability, but either:
 - Allows an attacker to conduct information gathering activities
 - Allows an attacker to hide activities
 - Includes a capability that behaves as expected, but can be easily compromised
 - Is a primary point of entry that an attacker may attempt to use to gain access to the system or data
 - Is considered a problem by some reasonable security policy

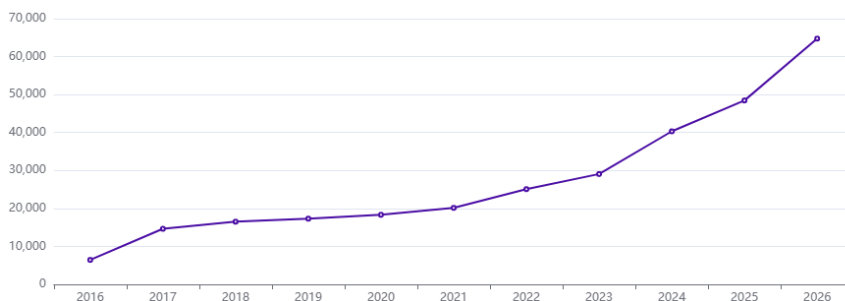
MITRE/CVE-project definition

An exposure is the weaker, more numerous and often cheaper-to-fix cousin of a vulnerability, and the difference is worth being able to articulate under pressure. A vulnerability lets an attacker do something they could not otherwise do. An exposure does not, by itself, allow compromise, but it violates a reasonable security policy and it helps: it reveals, it hides, it invites, or it makes the eventual compromise easier. Version banners and error pages that disclose the exact product and build, directory listings, administrative interfaces reachable from the wrong network, unused services left running, debug endpoints left enabled, default credentials that are technically 'the customer's fault', stale accounts of former employees, test systems accidentally published, unencrypted internal protocols, credentials committed to public repositories, and the absence of logging that would have recorded the intrusion are all exposures.

The four characteristics in the definition map onto attacker needs rather than onto technical categories, which is how you should learn them. Exposures that enable information gathering are reconnaissance aids, and they compress the most expensive phase of an operation, the mapping of what exists and how it behaves. Exposures that enable hiding are the ones that turn an intrusion into a persistent one: no logging, log rotation that deletes evidence, a shared service account used by everyone, or a permitted outbound channel through which traffic looks normal. Exposures whose capability 'behaves as expected but is easily compromised' are the saddest category, and they include legitimate remote administration tools, backup interfaces, and automation accounts with broad rights and weak authentication. And the 'primary point of entry' category is an admission that attack surface itself is a risk factor: an internet-facing component is an exposure even before anyone finds a bug in it.

There is an arithmetic to exposures that you should internalise, because it changes how you read a hardening report. Exposures multiply. Each reconnaissance aid raises the probability that a real vulnerability will be found in your deployment; each hidden channel raises the probability that a compromise will not be noticed; each convenient entry point raises the number of attackers who can try. So a configuration with a dozen individually unimportant exposures is not mildly risky, it is structurally risky, and the correction is not to argue about whether each one is 'really' a vulnerability, but to reduce the product of probabilities by removing the aids. This is also why the industry created hardening benchmarks and configuration baselines, why compliance frameworks devote whole sections to configuration, and why a modern practice area, exposure management, exists as a discipline separate from patching.

CVE number statistics (from cvedetails.com)



New/Updated CVEs



912 CVEs created, 1,905 CVEs updated since yesterday
44,399 CVEs created, 8,291 CVEs updated in the last 7 days
13,103 CVEs created, 21,067 CVEs updated in the last 30 days

Known exploited vulnerabilities

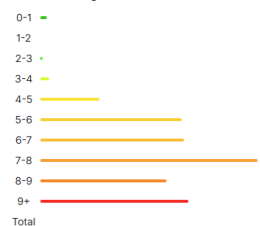
Time Period	Count
Since yesterday	1
Last 7 days	15
Last 30 days	45

Recent EPSS score changes

Score Change	Count
>5%	4
>10%	0
>50%	0

Distribution of vulnerabilities by CVSS scores

CVSS Score Range



Vulnerabilities

Weighted Average CVSS Score: 7.6

* For CVEs published in the last 10 years

João Paulo Barraca, André Zúquete

SIO

26

Charts of published vulnerability counts have a strong narrative: the curve rises, and in recent years it rises steeply, with record years followed by record years. Some of that growth is real in the sense that there is more software, more of it is networked, and more of it is inspected. Components that used to be isolated are now reachable: industrial controllers, building management, medical devices, vehicles, home equipment. Development produces more code, more often, with more reuse, so every defect in a widely copied component is counted once and felt thousands of times. And inspection capacity has grown, through automated fuzzing at industrial scale, coverage-guided fuzzing of parsers and kernels, static analysis integrated into continuous integration, and a large population of researchers paid by bounty programmes to look.

Finally, a caution about using these charts to compare products or to argue with a vendor. A product with many published vulnerabilities and a fast, transparent fix process is usually a better choice than one with no published vulnerabilities and no security contact, because the first is measurable and the second is not. Compare vendors on responsiveness and support lifetime rather than on defect counts, and read per-product rankings as an indicator of research attention. In a risk register, the number to report is not 'how many vulnerabilities exist' but 'how many affect us, how many are reachable, how many are known to be exploited, and what is our median time to close them'. Those four questions are answerable from your own data, and they are the reason the rest of this lecture is about identifiers, catalogues and processes rather than about statistics.

Vulnerabilities and Software

- The number of vulnerabilities **always increases**
 - Even if it is solved for a given software, it is still present in older (non updated) versions
 - May be relevant in systems **without updates** or due to **downgrade attacks**
- Vulnerabilities are a **common aspect of software**
 - **They are not only bugs, as there is an impact!**
 - There should be a process to handle them
 - Vendors: track vulnerabilities and issue fixes to clients
 - Clients: be informed about vulnerabilities and apply updates
 - Not all vulnerabilities can be corrected.
 - Sometimes only the attack is mitigated (e.g. segmenting the network, or disabling a feature)

The first bullet sounds like pessimism and is really arithmetic. Defect rates in software are roughly proportional to the amount of change and the amount of code written by people who do not fully understand it, which is every codebase on earth. If a project produces a security-relevant defect per some volume of new or modified code, then the cumulative count of published defects can only go up, and the only variables are the rate of change, the quality of the review and the intensity of external inspection. Adding a fix to version 1.4 does not remove the defect from the millions of installations of 1.2, which is why public counts are cumulative and why a vendor's past record predicts nothing about the current build.

The installed base is therefore the real subject of this slide. End-of-life systems continue to run for years: operating systems long past support still power tills, laboratory instruments, industrial controllers and hospital equipment whose certification makes an upgrade a project of its own; embedded devices ship with a kernel version frozen at build time and are updated rarely or never; mobile fleets are fragmented across carriers and generations. Attackers are excellent economists in this respect, and they attack the version that is actually deployed. Downgrade attacks add a twist, because even a patched system can be pushed back to its insecure behaviour: compatibility negotiation exists so that two peers can agree on a version, and attackers have repeatedly exploited it to insist on the older, weaker one, which is why fallback paths have been removed and why negotiation now carries signals that refuse a downgrade to a version the client knows to be unsafe. Any feature that says 'if the secure option fails, try the insecure one' is a vulnerability with a delay attached.

The distinction between a bug and a vulnerability is worth stating plainly, because it is the difference between a reliability conversation and a risk conversation. A bug makes the software do something wrong; a vulnerability makes the software do something that a policy forbids, and that 'whoever can trigger it' element is what creates impact. Both are defects, but only one needs an adversary model, and that is why vulnerabilities can be latent for years and then suddenly matter, often because a new class of attacker acquired the capability or the motivation. The organisational consequences are the two obligations on the slide: vendors need a security process that finds or receives defects, tracks them, issues fixes with clear affected ranges and fixed versions, and states support lifetimes honestly; clients need a process that consumes that information, because a fix that nobody installs is a purchase of good intentions. Note that the two processes fail together: the most common pattern in breaches is a fix that existed and an estate that did not take it.

Not everything can be corrected, and pretending otherwise produces dishonest registers. Some weaknesses are in hardware, in protocols that everyone has deployed, in a component whose source nobody owns, or in a product for which the vendor no longer exists. In those cases the honest options are mitigations: remove the component, disable the feature or the code path, restrict who and what can reach it, segment the network so that a compromise cannot proceed, put the system behind a filtering proxy, add monitoring tuned to the specific exploitation behaviour, or accept the risk with a documented owner and a review date. A compensating control is not a synonym for 'we did nothing'; it is a written statement of what the residual risk is, what limits it, and when somebody will revisit it. That discipline, more than any tool in this lecture, is what distinguishes an organisation that manages vulnerabilities from one that merely accumulates them.

Zero Day (or Zero Hour) Attack/Threat

- Attack using vulnerabilities which are:
 - Unknown to others
 - Undisclosed to the software vendor
- Occurs **at the day zero** of the knowledge about those vulnerabilities
 - For which no security fix is available
- A single “day zero” may exist for months/years
 - Known to attackers, unknown to others
 - Frequently part of attack arsenal
 - Traded around in specific markets



A vulnerability is 'zero day' not because it is exotic but because the defender's clock has not started: the weakness is unknown to the vendor, undisclosed, unpatched, and therefore unfixable at the moment of use. The same expression is also used for the other end of the timeline, the day a piece of exploit code or a patch becomes public, which is why you should read 'zero-day' in reports as 'a weakness being exploited with no available fix'. Once a fix exists, the same weakness in the same code becomes an 'N-day' problem: exploitation then depends on how fast the affected population installs the update, which is why an exploit against a two-month-old fix can still be enormously successful.

The most important idea in the slide is that a single zero day may exist for months or years. The exposure period of a weakness is set by whoever finds it first: a researcher who reports it privately, a fuzzer that has been hitting the same target for a year, or an operator who bought or built the exploit and uses it quietly. Historically, several of the most consequential intrusions used weaknesses that were later patched as if by magic, because the vendor found out only when the intrusion was investigated. The asymmetry is brutal and structural: the defender's knowledge of a weakness usually arrives with a patch, while the attacker's knowledge arrives with a working capability, and the interval between the two is the window of maximum risk.

Vulnerability Disclosure

- Disclosure of new vulnerabilities should be **coordinated** with the vendor
 - Typical Coordination:
 1. Describe vulnerability to vendor
 2. Vendor starts the correction process and agrees on a timeline
 3. Updates are issued and a CVE entry is created (Vulnerability is made public)
 4. Clients update the software, deploy protections or mitigate the impact
 5. The community discuss the root cause of the issue
- Vital to prevent Zero Day attacks
 - Clients will be (mostly) fixed when the vulnerability becomes public
- Requires collaboration from vendors
 - Relevant Legal frameworks: EU Cyber Resilience Act: vendor vulnerability-reporting duties
 - in force since 11 Sep 2026

Coordinated disclosure is an agreement about time, and it is worth understanding why the agreement exists at all. If a researcher publishes immediately, the vendor has no fix, customers get an unpatched warning, and attackers get a specification; the community's memory of this pattern is a set of incidents where disclosure and exploitation were separated by days. If instead the researcher stays silent indefinitely, or sells quietly, defenders get nothing and the vendor is not even aware that it has a problem. The compromise is a private channel, a bounded embargo, and publication synchronised with the availability of a fix, so that the majority of the population is protected at roughly the moment the information becomes useful to attackers.

The sequence in the slide deserves attention to its failure points. The initial report must contain enough detail for the vendor to reproduce the problem, which is why researchers provide proof-of-concept code, environment details and impact analysis, and why the absence of any of these three produces a stalled exchange. The vendor must acknowledge and reproduce, which requires a team with the authority to stop a release; many organisations discover they lack this only when they receive their first serious report. A timeline is agreed, patches are built and tested, and a CVE is published, ideally with an advisory that names affected versions and fixed versions and thanks the reporter. Then the unglamorous part decides the real-world impact: customers install, or do not, and the security community publishes analysis of the root cause, which is where the catalogue of weaknesses and the fix patterns become reusable knowledge. Note that the last step is the one that turns a single incident into collective learning.

The standard timeline in practice is around ninety days, with variations: some programmes use shorter windows for simpler fixes and longer ones for architectural rework, some publish on the agreed date regardless of the fix's completeness, and some grant extensions only against demonstrable progress. Ninety days sounds generous until you remember that it assumes the clock started when the vendor was told, and that the researcher may have known for months. The debate about what happens if the vendor does not act is unresolved: options are publishing anyway, partial disclosure, coordinated release with other affected parties, or handing the problem to a CERT-type intermediary that can coordinate many vendors at once. Intermediaries exist precisely for these cases, and their value is that a single affected party cannot force a coordination failure on the others.

Vulnerability detection

- Specific tools can detect vulnerabilities
 - Exploiting **known vulnerabilities**
 - Testing known **vulnerability patterns**
 - e.g., buffer overflow, SQL injection, XSS, etc.
- Specific tools can replicate known attacks
 - Use **known exploits** for known vulnerabilities
 - e.g.: EternalBlue exploit used by WannaCry targeting Microsoft SMBv1 (MS17-010, CVE-2017-0144)
 - Can be used to **implement countermeasures**
- It is vital to assess the robustness of production systems and applications
 - Auditing service often provided by third-party companies

Vulnerability detection means asking, with tools, whether the weaknesses we know about are present in the systems we run. The two families of techniques differ in how they decide. Pattern-based checks look for evidence: a banner that reports a version, a file hash, a missing patch identifier in a registry, a configuration value, a certificate characteristic, the absence of a header. Exploit-based checks prove the point by attempting the interaction, safely or not, and reporting success. The second is more trustworthy and more dangerous, and the trade-off between the two is the first design decision in any scanning programme.

The named example is instructive because it compresses several lessons into one line. The vulnerability was a flaw in an ancient file-sharing protocol implementation that allowed remote code execution over the network without authentication; the vendor published an advisory and a fix in March 2017, and the identifiers record both the vendor's bulletin number and the CVE. In April the exploit was leaked by third parties, and in May a ransomware campaign used it to spread automatically between machines, turning a single infected laptop into an organisation-wide event; a second, far more destructive campaign the following month combined the same exploit with credential theft and with a compromise of an update channel, and its damage was measured in billions. The lessons are the ones this lecture keeps repeating: an unpatched month is a real exposure, an obsolete protocol still enabled is a hole that no firewall rule fully closes, an exploit once public becomes a permanent fixture of the threat landscape, and the same capability is used by criminals, by defenders and by intelligence agencies without distinction.

The third point in the slide, that detection enables countermeasures, is where the value lies. Once you can prove which systems are exposed, you can patch them, disable the feature, block the protocol, isolate the segment, or add detection for the exploitation behaviour; without that proof, all four are guesses. Scanning also establishes the baseline that makes change visible: an inventory of what is listening on which port, refreshed on a schedule and compared with what was there last month, detects the forgotten service and the accidental exposure that no alert would otherwise describe. It is also the only way to answer the question that matters after an emergency announcement: do we run this, anywhere, including in that system everyone is afraid to touch?

Vulnerability detection

- Can be applied to:
 - Source code (static analysis)
 - ZAP, Veracode, LLMs, AppScan Source, ...
 - Running application (dynamic analysis)
 - Valgrind, AppScan Standard ...
 - Externally as a remote client:
 - OpenVAS, Metasploit, ...
- Should not be blindly applied to production systems!
 - Potential data loss/corruption
 - Potential DoS
 - Potential illegal activity

The three positions describe where the tool sits relative to the software, and each sees a different part of the problem. Static analysis reads the source or the binaries without executing anything: it follows paths through the code, tracks how untrusted input reaches dangerous operations, and reports candidate flaws together with the location and the trace. Its advantage is coverage of code that is never run during a test, including dead paths, error paths and rarely-used administrative functions, and the fact that it fits into a pipeline before release. Its disadvantages are false positives, the difficulty of judging whether a candidate is reachable in a real configuration, and its blindness to runtime and infrastructure problems such as a misconfigured server, a leaked key, or a missing authorisation check that depends on data.

Dynamic analysis exercises the running application, ideally with realistic accounts, roles and data, and looks for failures from the outside and, with instrumented runners, from the inside. Its strengths are the opposite of static analysis's: it proves that something is reachable and exploitable, and it catches environment problems. Its weaknesses are coverage, since only the paths you exercise are tested, and the need for a representative environment, credentials and data. A proxy-based scanner that crawls an application is the most familiar form, and it is worth knowing that it will never find the class of problems that dominate real breaches in access control: the tool can discover that an endpoint exists, but not that the endpoint should have refused this particular user. Between the two sit techniques that combine them, instrumenting the application while it is exercised so that the runtime data can confirm or kill a static candidate, and software composition analysis, which does not look at your code at all but at the dependencies you shipped, and which is the only technique that scales to the majority of a modern application.

Fuzzing deserves a mention whenever this list is discussed, because it has become the most productive discovery method for memory-safety and parser bugs. A fuzzer generates inputs, observes which ones reach new code, and mutates towards the interesting ones; coverage-guided fuzzing of parsers, decompressors, kernels and language runtimes has found more of these defects than any human effort, and industrial programmes now run fuzzers continuously against open-source infrastructure, which is a large part of why the statistics on published vulnerabilities have grown. Related, and worth discussing honestly, is the use of large language models to review code and to generate candidate findings: they are useful at scale for surfacing suspicious patterns and for triaging huge codebases, they produce plausible errors, and they have already contributed to real discoveries in recent research; but they inherit the classic problem of static analysis, which is that a candidate is not a vulnerability until a human with context decides otherwise.

The warning at the bottom of the slide should be treated as an operational rule, and the reasons are concrete. Intrusive checks can crash fragile software: old network gear, printers, industrial controllers, building systems and medical devices have been knocked over by probes that a modern server would shrug off, and scanners have taken out services by exhausting connection tables or by sending malformed packets that trigger a fault.

Vulnerability management

- Discussing and fixing vulnerabilities is important, yet insufficient
 - They will just keep appearing, non-stop
- Vital to discuss the **root mistake** of each vulnerability
 - So that it can be fixed, **preventing future vulnerabilities**
- Vulnerabilities exist because of **Anti-patterns**
 - Wrong or fragile implementation of logic structures
 - Which exist because of **lack of training, wrongly defined features, wrong design, wrong processes...**

Symptoms / Vulnerabilities



Mistakes / Anti-patterns

The arrow in this figure is the most underrated idea in the lecture: vulnerabilities are symptoms, and the mistake that produced them is the disease. A single memory-copying pattern can generate dozens of findings across a codebase, a missing authorisation check can appear in every new endpoint because the framework makes it optional, a string-built query recurs wherever a developer needed a filter box, and hard-coded credentials keep coming back because the documentation shows that as the working example. If you respond to each finding individually, you will be busy forever and nothing will change; the count will keep rising because the generating pattern is untouched. This is why the arrow points from mistakes and anti-patterns to symptoms and vulnerabilities, and not the other way round.

Operationalising this is a specific and learnable discipline. When a serious finding is fixed, ask what class of mistake it is, and then search the entire codebase for the class rather than the instance, using structural search or a query written for the pattern rather than a grep for the string. Write a test that would have caught it, so the class cannot silently return. Encode the pattern as a rule in the pipeline so that new instances are blocked at review time. Check whether the pattern exists in other systems owned by other teams, since the same developer, the same template, or the same internal framework usually means the same defect elsewhere. Then ask the harder question in the retro, without blaming: what made this mistake easy to make and hard to notice, and what would make it impossible to make? Answers are usually structural, such as a framework that requires an explicit authorisation call instead of granting it by default, a build with no memory-safety checks enabled, or a documented 'how to get it working quickly' recipe that everyone copies.

The class of anti-patterns is remarkably stable across industries, which is why the next slide's catalogue exists. Unbounded copies and unchecked indexes into buffers; queries and commands assembled by concatenating strings; hand-rolled cryptography; secrets embedded in code and configuration; authentication treated as sufficient where authorisation is required, especially on new endpoints and in APIs; unsafe deserialisation of untrusted data; time-of-check-to-time-of-use races between a validation and the action; error handling that swallows failures, including authorisation failures, so that the safe path becomes the broken one; and default configurations that are insecure because the demo needed to work. None of these are exotic, all of them are preventable by making the secure path the easy path, and every one of them is cheaper to prevent at design time than to patch in production, which is the entire argument for secure-by-design work.

CWE - Common Weakness Enumeration

- **Common language for discussing, finding and dealing with the causes of software security vulnerabilities**
 - Found in code, design, or system architecture
 - Each individual CWE represents a single weakness type
 - Currently maintained by the MITRE Corporation
 - A detailed CWE list is currently available at the MITRE website
 - The list provides a detailed definition for each individual CWE
- **Individual CWEs are held within a hierarchical structure**
 - CWEs at higher levels provide a broad overview of a vulnerability type
 - Can have many children CWEs associated with them
 - CWEs at deeper levels provide a finer granularity
 - Usually have fewer or no children CWEs

Symptoms / Vulnerabilities
Represented as CVE



Mistakes / Anti-patterns
Represented as CWE

The catalogue of weaknesses fills the gap between the two levels at which people talk about security. Engineers talk about a specific crash in a specific function; managers talk about risk in a specific product. Neither level is reusable. This catalogue gives a middle vocabulary: a finite, named, hierarchical set of weakness types, each with a definition, examples, likely consequences, detection approaches, mitigations, and relationships to other weaknesses. It is maintained as a public community effort, has grown continuously since it started, and is now embedded in tooling to the point that most static analysis products emit identifiers from it and most advisories cite at least one.

The hierarchy is not a taxonomy for its own sake; the levels correspond to how much context you have. Broad categories at the top organise discussion and let you say 'this is an issue of resource management' without pretending to precision. Mid-level entries describe a weakness class, such as improper input validation or a classic memory-safety error, and are useful for code review guidance and for searching a codebase for the pattern. Deep entries describe concrete, implementation-shaped mistakes that a tool can look for directly, and typically have few or no children. A single vulnerability may be described at several levels at once, which is why the catalogue is organised as a graph with multiple views rather than as a single tree: different audiences need different resolutions, and the same object must be findable from both directions.

The relationship with identifiers for individual vulnerabilities is what makes this practically valuable, and it is exactly the arrow in the figure. A CVE names one instance, in one product, with one fix; the weakness classification says what kind of mistake it was. When instances are tagged with their weakness type, the ecosystem can do the analysis that individual records never allow: which classes dominate in a given language or a given kind of component, which classes lead to the most severe outcomes, which classes our own code produces, and which classes our tooling detects. Coverage is imperfect, many records are still unclassified or mis-classified, and a single instance may have several contributing weaknesses, so treat the mapping as a starting hypothesis rather than as data. There is also the complementary machine-readable form, in which a supplier states whether a given product is affected by a given weakness or instance, which is how a bill of materials answers 'are we affected?' without a phone call.

CWE-416: Use After Free

<https://cwe.mitre.org/data/definitions/416.html>

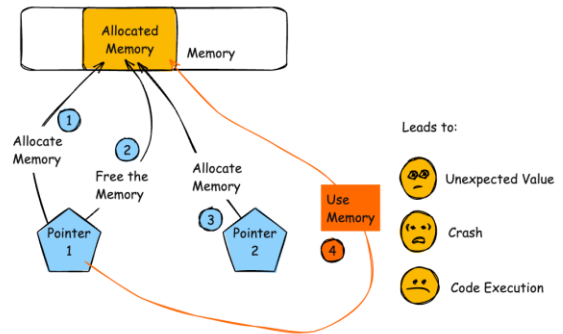
- The product **reuses or references memory after it has been freed**.
 - At some point afterward, the memory may be allocated again and saved in another pointer, while the original pointer references a location somewhere within the new allocation. Any operations using the original pointer are no longer valid because the memory "belongs" to the code that operates on the new pointer.

- Mitigation Phase: Architecture and Design Strategy:

- **Language Selection:** Choose a language that provides automatic memory management.

- Mitigation Phase: Implementation Strategy:

- **Attack Surface Reduction:** When freeing pointers, be sure to set them to NULL once they are freed. However, the utilization of multiple or complex data structures may lower the usefulness of this strategy.
- **Effectiveness:** Defense in Depth



To understand the diagram, you need only two facts about manual memory management. Allocating and freeing do not move or erase anything: they update the allocator's bookkeeping about which regions of the heap are available. And a pointer is just an address with no connection to the life of the object it once referred to. So after a region is freed, the pointer still 'points' at something, but the meaning of the bytes there is no longer under the code that holds the pointer. When the allocator later hands that same region to a different allocation, possibly of a different type, the old pointer now describes the new object's memory, and any read or write through it is an operation on somebody else's data.

That confusion is exploitable rather than merely a crash. If the new object contains data an attacker controls, for example content received from the network, the old code will interpret attacker-controlled bytes as the fields of the type it expects, which can produce out-of-range offsets or forged function addresses. Attackers shape this deliberately: they influence which allocations happen and in what order, so that the freed slot is reused by something they choose, a technique usually described as heap grooming or spraying. The classic escalation is overwriting something executable, such as a function pointer or a table of virtual methods, so that when the old code calls through it, control transfers to attacker-chosen code; with a read primitive instead, the same condition leaks pointers or secrets that defeat address-space randomisation. Because the timing and the allocation pattern matter, these bugs are intermittent and difficult to reproduce, which is exactly how a use-after-free survives code review.

The mitigations quoted from the catalogue are worth commenting on individually, because their effectiveness differs by an order of magnitude. Choosing a language with automatic memory management, or with ownership checked at compile time, removes the entire class rather than one instance: garbage-collected and ownership-checked languages make this family of defects structurally impossible, which is why whole-industry recommendations now exist for migrating security-critical components away from manual memory management, and why large system components have been rewritten over the past years. Nulling a pointer after freeing is genuinely useful but partial: it turns a use-after-free into a predictable null dereference for the one pointer you remembered, and it does nothing for other references to the same memory, for copies of the pointer in other structures, for containers that keep iterators, or for the double-free variant, and its usefulness drops quickly in complex data structures. Between these extremes sit reference counting, smart pointers with clear ownership rules, arena and region allocation with lifetime discipline, object pools with generation counters, and allocator hardening that makes reuse patterns less predictable.

10 most common vulnerability types found in real systems

- Reviewed **every ~4 years** from real world security assessments
- Each type contains multiple CWEs to be prevented
- Industry can focus on the most common problems
 - Improve training, testing and awareness on this areas
 - Improve toolkits, languages and frameworks
 - Create detection and defenses against typical vulnerabilities

[A01:2025 - Broken Access Control](#)

[A02:2025 - Security Misconfiguration](#)

[A03:2025 - Software Supply Chain Failures](#)

[A04:2025 - Cryptographic Failures](#)

[A05:2025 - Injection](#)

[A06:2025 - Insecure Design](#)

[A07:2025 - Authentication Failures](#)

[A08:2025 - Software or Data Integrity Failures](#)

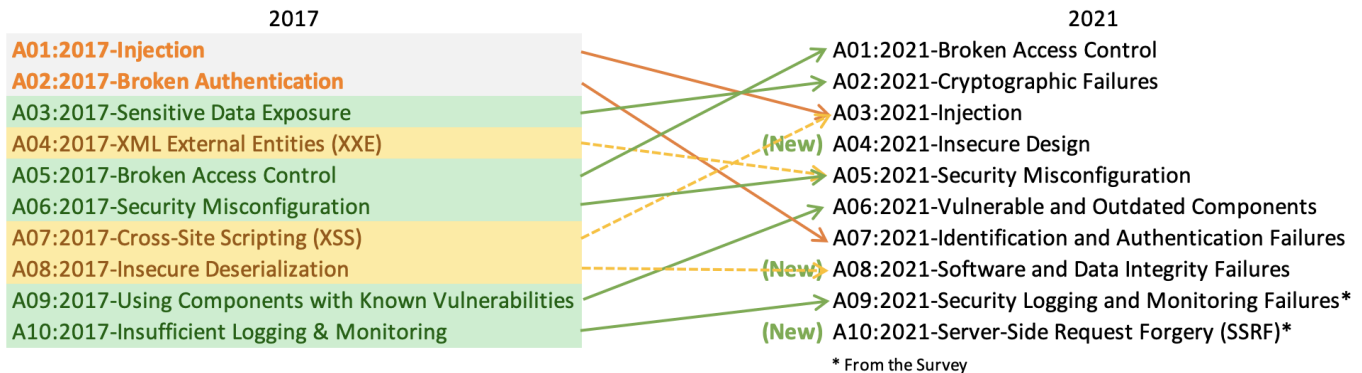
[A09:2025 - Security Logging and Alerting Failures](#)

[A10:2025 - Mishandling of Exceptional Conditions](#)

OWASP is an open, volunteer-driven community organised as a non-profit foundation, and its output is a body of freely usable material on web application security: implementation guidance, threat-modeling requirements, testing guides, verification standards for applications and mobile, a maturity model for organisations, and deliberately vulnerable applications used for training and for benchmarking tools. The value of this output is that it is common ground: a developer, an auditor, a supplier and a buyer can point at the same page instead of negotiating definitions. The Top 10 is the most visible of these documents, and the fact that it is a communication instrument rather than a technical standard is the key to using it well.

The list is built from data rather than from opinion alone: aggregated telemetry from vulnerability-scanning and application-security vendors covering very large numbers of real systems, combined with expert review of what should be included and how items should be grouped. Each entry is deliberately coarse, because it is a category rather than a bug: it names the problem, describes the risks and the typical failure modes, points to the weakness types that implement it, and lists prevention measures. That is why the slide says each type contains multiple weaknesses: an item such as injection or cryptographic failure is a family, and the family is where the training, the tooling rules and the review checklists operate. Because the ranking comes from what is found in real deployments, it is a better predictor of what you will find in your own estate than a list built from what researchers find impressive.

The 2025 edition is worth reading as a document of its time. Access control remains first, which tells you that after two decades the most common defect is the application correctly identifying a user and then failing to check whether that user may perform this action on this object, the failure that dominates API-heavy and multi-tenant systems. Misconfiguration moved up, reflecting infrastructure deployed as code and cloud services whose defaults nobody owns. A new item on software supply chain failures absorbs what earlier editions listed as integrity and third-party component problems, which is the direct consequence of a decade of incidents in which the compromise moved through build systems, package registries, update channels and signing keys rather than through the application. Injection moved down but not away, and now explicitly includes cross-site scripting, an acknowledgement that the two are one failure of separating data from instructions. Authentication failures, logging and alerting failures, and a new item on mishandling of exceptional conditions round out a picture in which several entries are about operations and observability rather than about code.



The history of the list is the best available fossil record of twenty years of defensive progress, which is why the slide's caption is about mistakes being prevented while others arise. Injection was first for several consecutive editions, and its gradual decline in rank is not a story of attackers losing interest but of the fix becoming structural: parameterised queries and prepared statements moved into frameworks and object-relational mappers, so writing an injectable query became the unusual choice rather than the default. Cross-site scripting fell for the same reason, as templating systems began escaping output automatically and browsers added protections, yet it survives wherever a developer writes into a page from a value, which is why it was eventually folded into the injection category rather than removed.

Other movements record architectural migrations rather than progress. A category describing exposed sensitive data was renamed into one describing cryptographic failures, which is a shift from describing the symptom to describing the cause, and it reframed the responsibility from 'data leaked' to 'you did not protect it in transit and at rest'. Broken access control went from the bottom of the list to the top, because the industry moved from monolithic pages to APIs, single-page clients and multi-tenant services where authorisation is bespoke per endpoint and cannot be delegated to a library. Items about third-party components and integrity of pipelines appeared as dependency reuse exploded and then as build systems became the target, ending in the current supply-chain framing. Categories about logging and monitoring, and now about mishandling of exceptional conditions, appeared as practitioners realised that a failure which is silent is a failure that becomes an intrusion: an exception that swallows an authorisation error, a fail-open default, or a crash that leaves a service in an insecure state.