OWASP Top 10

Conor Rynne[*] Oihana Gallastegi [†]

April 3, 2024

Introduction

Welcome to OWASP Barcelona. Barcelona has a thriving tech start up community, full of people with amazing ideas, and the skills to make them happen.

But being able to do things and being able to do things in the right way can be very different things. Sure, what you've made might technically work, but it might be working in a way that puts your company at risk.

It might expose your customer database. It might allow your customers to obtain your services for free. It might even give people the ability to disrupt your service or steal payment information.

This talk is about the most common pitfalls and failings of web applications today, and how you can avoid them in your projects. This is mostly related to web applications; mobile applications and web APIs have their own considerations.

[*]Original author [English]
[†]Translation [Spanish]

Contents

1 A01:2021 - Broken Access Control

Access controls exist to ensure that users are only able to do what they are supposed to be able to do, and cannot act outside of these. After all, you wouldn't want one user to be able to see another user's private information, or even be able to delete them from your service entirely!

When deciding on what users are supposed to be able to do, there are two golden rules:

1. **Principle of Least Privilege** - This is where you define all the abilities a user should have and assign them only the permissions to do those things. For example, for an article writer, you might want to give them permissions *only* to create and modify their own articles.

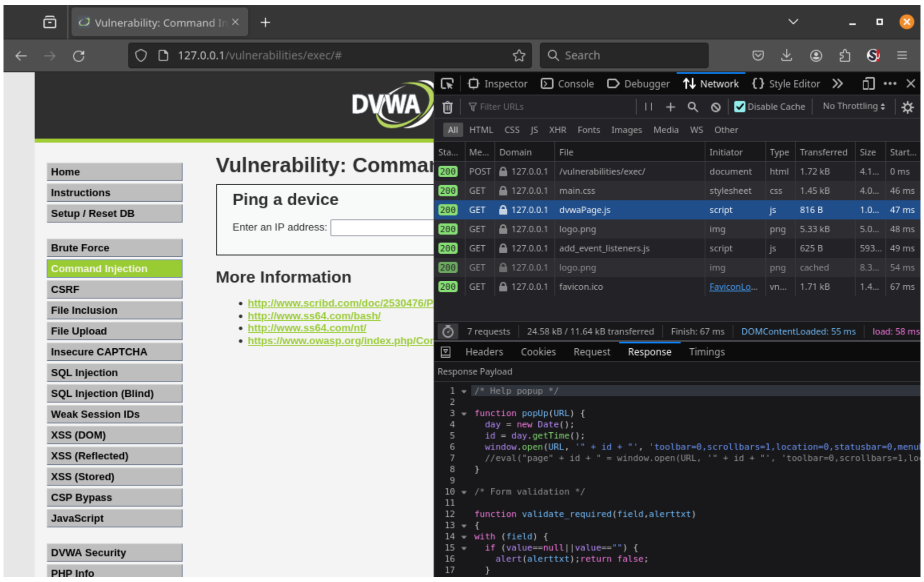
2. **Deny by Default** - If a user is not explicitly given a permission, the application should act as though they don't have that permission at all.

Enforcement

Even if you do have these permissions and rules written out, it is important that enforcement is implemented. It is no good saying 'only administrators can access administrative functions' if your administrative functions do not check that the user utilising them is indeed an administrator.

With that said, there are good ways to do enforcement and there are bad ways. There are two more golden rules to keep in mind:

1. **Do not send unnecessary data to the user. It can all be read.** - Even if you have locally run JavaScript code to ensure that only the data the user is supposed to see is displayed in the web browser, the user can see any the raw data you send them if they so choose. This ability is included with all modern web browsers as part of their Development Consoles.



Sta...	Me...	Domain	File	Initiator	Type	Transferred	Size	Start...
200	POST	127.0.0.1	/vulnerabilities/exec/	document	html	1.72 kB	4.1...	0 ms
200	GET	127.0.0.1	main.css	stylesheet	css	1.45 kB	4.0...	46 ms
200	GET	127.0.0.1	dvwaPage.js	script	js	816 B	1.0...	47 ms
200	GET	127.0.0.1	logo.png	img	png	5.33 kB	5.0...	48 ms
200	GET	127.0.0.1	add_event_listeners.js	script	js	625 B	593...	49 ms
200	GET	127.0.0.1	logo.png	img	png	cached	8.3...	54 ms
200	GET	127.0.0.1	favicon.ico	FaviconLo...	vn...	1.71 kB	1.4...	67 ms

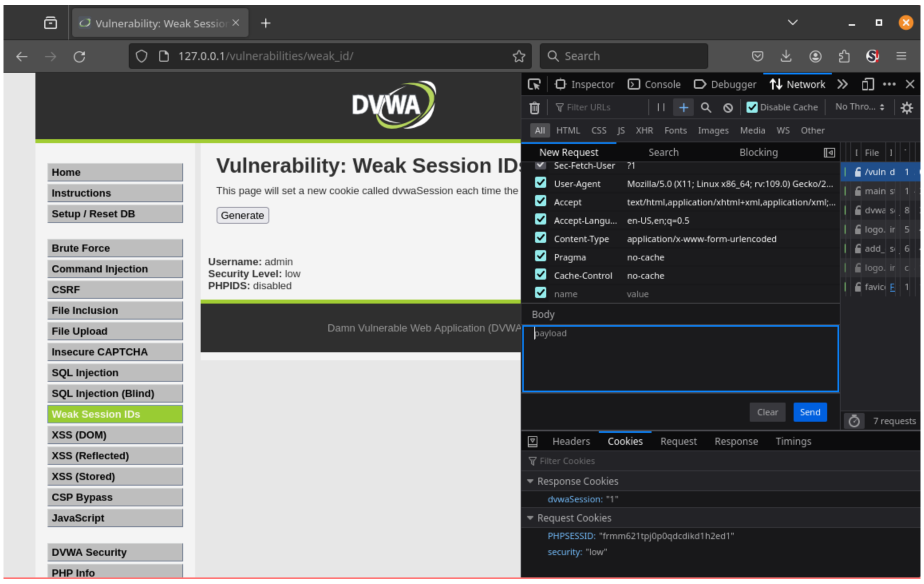
7 requests | 24.58 kB / 11.64 kB transferred | Finish: 67 ms | DOMContentLoaded: 55 ms | load: 58 ms

Response Payload

```
/* Help popup */

function popUp(URL) {
  day = new Date();
  id = day.getTime();
  window.open(URL, '" + id + "', 'toolbar=0,scrollbars=1,location=0,statusbar=0,menu
  //eval("page" + id + " = window.open(URL, '" + id + "', 'toolbar=0,scrollbars=1,lo
}

/* Form validation */

function validate_required(field,alerttxt)
{
with (field) {
  if (value==null||value=="") {
    alert(alerttxt);return false;
  }
```

The request viewer in Mozilla Firefox

2. **Do not blindly trust any data sent by the user. It can all be manipulated.** - Likewise, just because you do not give the option to adjust certain bits of data in your web application, doesn't mean they don't have the ability to do so manually. This ability is also included with all modern web browsers as part of their Development Consoles.



This page will set a new cookie called dvwaSession each time the

Username: admin
Security Level: low
PHPIDS: disabled

Sec-Fetch-User	?1
User-Agent	Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/2...
Accept	text/html,application/xhtml+xml,application/xml;...
Accept-Langu...	en-US,en;q=0.5
Content-Type	application/x-www-form-urlencoded
Pragma	no-cache
Cache-Control	no-cache
name	value

Body

payload

Response Cookies
dvwaSession: "1"

Request Cookies
PHPSESSID: "frmm621tpj0p0qdcdikd1h2ed1"
security: "low"

The request editor in Mozilla Firefox

HTTP CORS

A HTTP Cross-Origin Resource Sharing (CORS) policy can help you restrict what resources your applications can use.

It does not, however, prevent direct access, nor is it particularly effective against non-web browser HTTP clients such as cURL, where users can readily edit the HTTP headers sent in requests.

CORS policies are sent to the browser via HTTP headers, specifically the `Access-Control-Allow-Origin` header. Access to specific HTTP verbs (like GET and POST) and the restriction of other HTTP headers can also be handled with the `Access-Control-Allow-Methods` and `Access-Control-Allow-Headers` headers respectively.

With regards to how to build a secure policy, again the golden rules of **Principle of Least Privilege** and **Deny by Default** come into use. That is, you will want to restrict access to only sites that should be using those resources and denying everything else.

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://www.mydomain.com
Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Allow-Headers: Authorization, Content-Type
Content-Type: application/json
...
```

An example HTTP response with CORS policy headers

Cookies and Session Data

Many websites use local storage mechanisms such as cookies and local databases. One of the most common uses of these is to store a session token for logged in users. When a web browser submits a session token and a web page identifies that token as belonging to a logged in user named X, the web page will do the rest of its processing knowing that the user of the web browser is the user named X.

To make this a secure means of session management, it is imperative that the session token cannot be easily guessed. That is, it needs to be a completely random value that an attacker isn't likely to guess. Most modern frameworks handle this for you, and if you have access to this, it is better to use it; the mechanisms provided by web application frameworks are usually far more tested and hardened than anything you can realistically create.

But if that isn't an option and you do need to implement it yourself, it is important to use a good, cryptographically-secure pseudo-random number generator as a basis for session token generation.

Many issues arise from using predictable or guessable data such as time or user IDs. Since user IDs are static, and time is predictable (ie: one minute after after 12:47 it will be 12:48), these are not good candidates for use as a session token, even when put through a hashing algorithm.

Encrypted Java Web Tokens (JWT) can be useful here, as JWT encryption can be used as a verification that the data has not been tampered with. However, bear in mind that convention JWT encryption is useful *only* for verification purposes. Because it also includes a clear-text (aka: not encrypted) copy of the data, the information is not secure against someone reading it.

Prevention

The summary of methods one can use to prevent these issues is:

- Only give users the minimum amount of permissions they need to do their legitimate tasks.
- Explicitly deny access to everything else.
- Don't send unnecessary data to the user.
- Don't blindly trust any data from the user.
- When it comes to session management, don't create your own solution where it isn't necessary.

2 A02:2021 – Cryptographic Failures

Encryption has become an important part of most of our interactions online. It assures us that our information is not being intercepted, and that communications are not being modified. But as with all things cybersecurity, there are right ways to do things and many, many wrong ways to do things.

Inappropriate Transmission in Cleartext

The first way to do encryption wrong is to not do it at all, especially with regards to communications that involve sensitive information. Ideally all pages should be encrypted in the modern day, but when it comes to login details or Personally Identifiable Information (PII) it becomes an outright requirement, often by law.

Protocols such as HTTP, FTP and Telnet do not support encryption at all. Instead more secure counterparts should be used such as HTTPS, SFTP and SSH. SMTP traffic is not always encrypted, and care should be taken to ensure that encryption is mandatory on such services.

It is advisable in the case of web browsers to enforce the use of encrypted HTTPS traffic via the use of HTTP Strict Transport Security (HSTS). This can ensure that if an encrypted connection cannot be made, the connection will fail.

Inappropriate Storage in Cleartext

Encrypting traffic is all well and good, but if your systems are breached, or your hard drives are stolen, storing sensitive details in clear, unencrypted text will do even more damage.

Full Disk Encryption (FDE) encrypts your entire disk and requires an encryption key before any of it can be used. This is quite useful against physical theft, but is less useful on an already running system; once it is plugged in and the key is given, any user or process on the currently running machine can access the data on the FDE encrypted drive, including malicious programs or SQL databases complying with malicious SQL queries.

Database encryption encrypts an entire SQL database. While this might protect against other malicious programs on the system, it will still yield the data when complying with a malicious SQL query performed by an attacker.

Encrypting the data *inside* the database will yield better results when it comes to protecting against attacks like SQL injection. Unlike the last two encryption schemes, there is no one-click way to do this, and this is instead something built into your application.

However, when it comes to credentials and passwords, it is better to avoid using encryption and instead use a hashing algorithm. The main difference between an encryption algorithm and a hashing algorithm, is that with the latter it is impossible to recover the original data. Hashing algorithms can, however, be used to check if two bits of data are identical; with a good hashing algorithm, no two non-identical bits of data will produce the same hash, the output when the data is processed via a hashing algorithm.

This makes hashes an ideal method of storing credentials. Instead of storing the passwords, you store the hash and compare the hashed password of a recent login attempt against it. However, this can still have issues if you leave it as is.

A hashing algorithm will consistently produce the same hash when given an identical bit of data, and this is true of passwords as well. Thus there are efforts to create dictionaries of hashes that correspond to given passwords. These are called rainbow tables, and they make the deriving of passwords from hashes very easy. To combat this, it is common practice to insert a consistent piece of junk data into the password before hashing it.

This is called a 'salt' and you can even store it alongside the user's password in the SQL database. However, for added security, you can also add a salt that originates outside of the database, such that someone with only database access cannot access it.

Weak Algorithms and Protocols

A common problem many sites have is that they use or accept ageing algorithms and protocols when it comes to setting up encrypted traffic. Older algorithms and protocols often have vulnerabilities that make unauthorised decryption of messages much easier.

While offering support for older protocols and algorithms might seem like a good idea for compatibility with older devices, in practice it allows attackers to force users to use the weakest supported ones. This is because with SSL and TLS, the two most common protocols used for encrypted traffic on networks, the first few communications are about deciding what encryption algorithms to use, and are thus unencrypted and susceptible to manipulation. If an attacker can force communications to use algorithms such as RC4, the encryption could be broken in minutes!

Key secrecy

This may sound obvious, but secret encryption keys are most useful when they are... secret. The more people know about them, the more of a liability the use of these keys can be. Just as you wouldn't want to use 'admin:password' on your router, you don't want to use default encryption keys in any of your devices or products either.

Additionally, if your developed software, private or public, uses encryption keys at any point, you will wish to keep them out of your code repositories. Many code repository services such as Azure DevOps and GitHub provide options to store required encryption keys separately.

Caching

Many web browsers and services cache data to prevent having to reload data. The caching of sensitive personal data can be very problematic, however, as it can allow other users of, for example, a public computer, to view said sensitive information of a prior user.

This can be controlled via the use of HTTP headers. Specifically, to disable caching, you would want to use:

```
Cache-Control: no-cache, no-store, must-revalidate
Pragma: no-cache
Expires: 0
```

The HTTP headers used to disable caching

3 A03:2021 – Injection

Imagine for a moment that you had something like a GameShark that, instead of messing with your video games, it messed with things around you in real life. It could set prices in a shop to whatever you wanted, make billboards magically show whatever you wanted to everyone, make a company give you its entire customer credit card database, or if you so wanted, it could basically cripple a business you didn't like.

Against applications that don't defend themselves adequately, injection attacks *are* that GameShark device. Injection attacks allow an attacker to insert their own code into your applications, and the effects can range from the modification of site parameters (like prices), the disclosure of sensitive data, to outright control of your web servers and even clients.

These attacks usually work because the application running the code, be it your SQL server or a client's web browser, cannot differentiate between legitimate code and user inputted data.

There are many types of injection attacks, so we will go over the most common ones.

Cross Site Scripting

Cross Site Scripting (XSS) is the process of injecting JavaScript code into a web application. This code then runs on the client's machine. While this may not immediately seem dangerous for the server, it can be extremely dangerous for your reputation as a company; at this point, the attacker has nearly complete control over what your user can see and interact with on-screen.

If your session storage isn't secured against JavaScript access, they can even stay undetected and instead send the client's session data over to them, allowing them to log into your application as the affected user without a password.

The main defence against this is the filtering of inputted data. The filter would replace things like special characters with their HTML notation equivalents. '>' would become '&gt;' in the code, which would then appear as '>' in the web browser, but would not have any of the properties that would make the browser confuse it with part of the actual HTML code. This is an oversimplification, however, and the filtering process is very complex due to the large amount of possible bypasses available, so it is recommended to use the filter function from your web framework or a software library instead.

Server Side Template Injection

Many modern web application frameworks come with a templating engine. The way it is generally supposed to work is that the template file, containing the static HTML code as well as placeholders for dynamic content, is loaded from a file, the dynamic content generated, and then the templating engine adds it in to the final file delivered to the client.

Problems arise when processes other than the templating engine make modifications to the base template. If dynamically generated content is added to the base template before the templating engine can work, an attacker can insert their own code for the templating engine to execute. Depending on the underlying technologies involved, this can grant an attacker Remote Code Execution capabilities. This is where they can effectively control your server.

Tests for Server Side Template Injection (SSTI) vulnerabilities vary depending on the underlying technologies used, but a good collection can be found at `https://book.hacktricks.xyz/pentesting-web/ssti-server-side-template-injection`.

The best defence against this is to not let any processes other than the templating engine insert dynamic content into templates.

SQL Injection

This technique is considered a classic among attackers, and made famous by the Anonymous hacktivist group. SQL commands, or queries, are written commands, not entirely unlike command-line. For example, a basic password check could look like:

```
SELECT username FROM users WHERE username = '<inputted username>' AND
    password = '<the submitted password, hashed>'
```

Boilerplate simple SQL code for logging in

where the web-server and dynamic web page technology would add in the actual username and hashed password obtained from the user. If the user were to enter the credentials for the user 'bob' and the password 'hunter2', the command would look like:

```
SELECT username FROM users WHERE username = 'bob' AND password = '84
    ca85078d6fa3a9b01dae0242938a9b71c9c6920f8d790505cad7a7' # The SHA2
    -224 hash for 'hunter2'
```

How the SQL code looks for the user 'bob' with the password 'hunter2'

But what if I were a malicious user simply trying to get access to Bob's account? Then I might instead put my username as:

`bob' WHERE 1=1;#`

And the code would look like this:

```sql
SELECT username FROM users WHERE username = 'bob' WHERE 1=1;#' AND
    password = '<irrelevant>'
```

What I have essentially done here is rewritten the SQL query to completely disregard the password. For reference, the ';' character finishes a statement in SQL and the '#' character is considered a comment marker; anything to the right of that symbol will be ignored by the SQL server.

This works because the web-server is blindly filling in the blanks of the SQL query directly with our inputs, and the SQL server cannot differentiate between what is code and what is user input. For this reason, we never use direct queries with user inputs but instead use parameterised queries. Some SQL servers have different names for it, but the general idea is that it is a stored query that accepts parameters. Unlike with the direct query approach, any data passed as a parameter is marked as input data and never executed as code unless you use it in an `eval()` command (don't do this!)

How to fix

- Filter all inputs. Do not blindly trust any user inputs.
- If your web framework has template functionality, make sure that it's templating engine is the only thing inserting dynamic code into your pages
- When performing database queries with user data, always use parameterised queries

4 A04:2021 – Insecure Design

Insecure design is a much more nebulous problem. Essentially, it stems from your applications being developed and designed around insecure processes or with little thought being given about the security of the data.

The problem with doing things this way is that security should be a part of the design process, and trying to implement security after the fact is expensive, time-consuming and less effective.

An example of this might be if you wanted to take card data, but then completely forget to put in any sort of provisions for protecting said card data. So you build your application without the requisite security in mind, and then eventually you realise you have to rethink your whole architecture, at least to comply with PCI-DSS, the main security standard governing the storage of card data. And there are strong penalties if you store card data without complying with PCI-DSS, even if you aren't attacked!

It is imperative that the security needs of your applications be given proper consideration throughout all stages of designing and development of your application. Consider what your threats are, understand your security requirements, create processes to fulfil those requirements and use those in your applications.

As hinted at earlier, there is more to worry about than simply if you are attacked. if due care is not taken with customer PII, there may be legal consequences as well; Leakage of customer PII can result in heavy fines, courtesy of GDPR legislation. Implementing card payments without complying with PCI-DSS will damage your relations with Visa and MasterCard, to the point where they will not let you accept their cards in your payment processor anymore. And this is before we touch upon the other data privacy laws in other countries where you may be operating.

If you are creating your own software, you should adopt a Secure Software Development Lifecycle (SSDLC). The OWASP Foundation has created the Software Assurance Maturity Model (SAMM) to help with this. You can find more information at `https://owaspsamm.org/`.

Applications on shared resources should be resource-limited so that in the event of an attack, the application isn't bringing other services offline. You may also wish to consider the use of load balancers and redundant installations.

5 A05:2021 – Security Misconfiguration

So we have our product, and our shiny new toys, the next step is to configure them. Unfortunately this can be easier said than done, as proper configuration requires an understanding of the software/devices capabilities. The default configuration of most retail software/devices is to prioritise out-of-the-box operability and usability over security. This can lead to a number of issues including:

- Unnecessary features are enabled or installed - It is worth remembering that the more features you have active and the more services that are listening on ports, the more possible attack vectors an attacker may have.

- Default credentials - These are usually well documented and/or easy to guess.

- Missing appropriate security hardening - It is up to you to ensure that only the required services and features are running, and that those that are have been protected as well as is possible, including the testing of security configurations.

Developer tools can also be problematic with regards to their defaults, as they tend to prioritise helping developers pinpoint issues by providing lots of information. This is great in a developer environment but in production, this same information can be used by an attacker to try and get onto your systems.

- Ensure that debug modes are turned off in production, and that error messages are providing as little information as possible.

- Ensure that developer environments are set to connect to code repositories and other requisite systems in a secure manner.

- Ensure that web application frameworks are set to secure standards, turning off all debug modes, replacing default SSL keys and credentials and disabling any unnecessary services and features.

The hardening process should be documented for future reference. Even if you don't use that particular product in future, it should still be able to serve as a guideline for the secure setting up of a similar device.

6 A06:2021 – Vulnerable and Outdated Components

Software is written by humans and just like everything made by humans, it can have its flaws. Some of those flaws can have cybersecurity consequences ranging form the disclosure of information all the way to giving attackers complete control of a machine. Therefore it is vitally important that organisations have a regular patching procedure for all devices they use. From workstations and servers all the way to routers.

This applies to software made in-house too. Unless you're building everything from scratch, you are probably using some kind of third party library or interpreter to help create and run the software. Third party libraries have weaknesses just like any other type of software, however your application is likely to inherit said weaknesses if it is using a vulnerable version of the library. Keeping libraries up-to-date isn't as simple, however, as libraries also change their syntax and functions, requiring you to update the code in your application as well to ensure it works with newer versions.

7 A07:2021 – Identification and Authentication Failures

The protection of processes to authenticate users is of critical importance to the protection of your application and its users. Common flaws can include:

- Lack of protection against credential stuffing - the process of taking a list of known credentials, from such places as a leak of another website, and testing them on other applications such as your own.

- Allowing the use of brute-force attacks - where an attacker tries to guess the password to a user or list of users very rapidly.

- Allows the use of weak passwords.

- Does not encrypt traffic or hash passwords properly.

- Has no or improper multi-factor authentication

- Exposes session identifiers in the URL

- Does not properly invalidate session tokens or authentication tokens. They should be invalidated after logout or an extended period of time.

The solutions for these problems include:

- The use of multi-factor authentication. You can use Time-based One-Time Passwords via an authenticator app such as Google Authenticator on a smartphone. This will prevent attackers from being able to automatically be able to use credential stuffing attacks.

- Locking out an account temporarily after a few incorrect attempts - This will prevent or greatly hinder attackers from being able to use automated tools to try and guess passwords.

- Ensure registration, credential recovery pathways and API endpoints are hardened against account enumeration attacks by ensuring that authentication failure messages do not reveal the presence of that account.

- Log all failures and alert administrators if attacks are suspected.

- Use the built in session management tools of your web application frameworks

8 A08:2021 – Software and Data Integrity Failures

Software and data integrity failures are when there are security shortcomings in how items like dependencies, critical data, software updates and build processes are handled.

If, for example, your application relies on dependencies hosted by an untrusted third party, someone in control of that dependency could inject malicious code into your application by simply modifying that external dependency. If your build process involves an automated Continuous Integration/Continuous Deployment (CI/CD) mechanism, it is vital to ensure that the build process is protected from interference from rogue employees. Otherwise they could modify the build process to inject malicious code without said malicious code ever touching the repository. As many pieces of software now also include auto-update functionality, if there is no verification process, then attackers could potentially create a malicious 'update' that inserts malicious code into your applications.

The main ways of preventing this are:

- Use digital signatures or similar mechanisms to verify updates.
- Remove modification rights to build process configurations in CI/CD software for all except those who truly need it (project leads, for example).
- You can use tools such as OWASP Dependency Check or OWASP CycloneDX to check for vulnerabilities within your dependencias.
- Make sure all web app dependencies are either hosted by yourself or with a party you trust.
- If you must send unsigned or unencrypted serialised data to a client, ensure that there is an integrity check mechanism to scan the incoming data.

9 A09:2021 – Security Logging and Monitoring Failures

Logging is an important part of detecting breaches. Without it, you simply have no idea what is going on in your systems. As always, however, there are right ways and wrong ways to implement such a core functionality. In order to get the best results out of your logging system, it must:

- Log sensitive and auditable events such as login attempts (whether successful for failed), credit card payments (NB: card data itself should never be logged) or other important actions.
- Filter any user inputs, where relevant.
- Generate clear and immediately understandable log messages for warnings and errors.
- Be routinely monitored for suspicious activity.
- Have appropriate alerting thresholds. Setting it to only log on confirmed attacks may omit needed data to see other attacks that cannot be so easily confirmed server-side. However, having it log every single little thing will swamp your monitoring team and impede their efforts.
- Have an appropriate escalation system in place.
- Be triggered upon penetration testing activity, such as actions performed by OWASP ZAP.
- Escalate, detect and alert for ongoing attacks in real-time.

Log data should also not be available to the user, and you may be leaking sensitive information if they are. An incident response and recovery plan should be formulated to help deal with issues in the event that they arise.

10 A10:2021 – Server Side Request Forgery (SSRF)

Sever Side Request Forgery (SSRF) is a specific kind of attack that can allow an attacker to see and interact with systems that they are not supposed to, typically because they are behind a firewall or a port that the user isn't normally supposed to be able to contact.

The way a vulnerable service would normally work is like this:

1. The client sends a request to the server. The request data includes the specific endpoint that needs to be contacted by the server.

2. The server then sends a request to the endpoint indicated, possibly including data from the original request in step 1.

3. The endpoint sends data back to the client, and the server sends this data, modified or otherwise, back to the client.

The problem begins when the client changes the API endpoint variable from something like `/api/status` to `http://127.0.0.1:3000`. The server then tries to contact this address, essentially port 3000 on the same system. This can be used to port scan the systems (the attacker can try to scan other machines on the network like this too), and if a valid HTTP service is found on the system, can identify what is running on those ports. This allows an attacker to accurately map your internal network even behind a firewall or VPN.

There are a number of ways to fix this:

- **From a the network:**

  - Ensure communications between the web application and internal network systems go through an internal firewall. Set default-deny rules to prevent any unwanted communications to and from internal systems.

  - In that same firewall, only allow communications to and from intended targets.

  - Use network segmentation to prevent the ability to communicate with unintended systems.

  - Establish a lifecycle, ownership and regular review of firewall rules.

  - Log accepted and blocked network flows from the firewalls.

- **From the application:**

  - Whitelist all allowed communication endpoints. Check against a map or dictionary object of known endpoints and use the endpoints from that object.

- Filter all user inputs.

- Don't send raw responses to clients, even for JavaScript to parse.

- Disable HTTP redirection.

- Be aware of the URL consistency to avoid attacks such as DNS rebinding and "time of check, time of use" (TOCTOU) race conditions.

- DON'T use a deny-list approach (where the application has a list of endpoints to specifically deny). Attackers have multiple ways to bypass these.