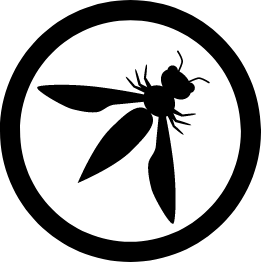
**OWASP Top Ten**



Calvin Hong – 431518

Semester 6 – RB03

**Table of contents**

[Introduction 3](#_Toc105543232)

[What is OWASP? 3](#_Toc105543233)

[The Current OWASP Top 10 3](#_Toc105543234)

[A01 – Broken Access Control 4](#_Toc105543235)

[A02: Cryptographic Failures 6](#_Toc105543236)

[A03: Injection 8](#_Toc105543237)

[A04: Insecure Design 9](#_Toc105543238)

[A05: Security Misconfiguration 10](#_Toc105543239)

[A06: Vulnerable and Outdated Components 11](#_Toc105543240)

[A07: Identification and Authentication Failures 12](#_Toc105543241)

[A08: Software and Data Integrity Failures 13](#_Toc105543242)

[A09: Security Logging and Monitoring Failures 14](#_Toc105543243)

[A10: Server-Side Request Forgery (SSRF) 15](#_Toc105543244)

[Which security precaution would be most beneficial for our project? 16](#_Toc105543245)

# Introduction

The purpose of this document is to list common security pitfalls in software development. For this, we have investigated the Open Web Application Security Project as a reference to the most pressing security concerns in current days. These concerns will be listed below, along with the potential damage they can cause to a system, as well as the solutions for it.

# What is OWASP?

The Open Web Application Security Project (OWASP) is an international non-profit organization dedicated to web application security. One of OWASP’s core principles is that all their materials be freely available and easily accessible on their website, making it possible for anyone to improve their own web application security. The materials they offer include documentation, tools, videos, and forums. Their best-known project is the OWASP Top 10, which will be the core of this investigation.

# The Current OWASP Top 10

Each topic in the top 10 will be investigated in a descending order from the first place, specifying the description of the issue, its’ change in position compared to the previous iteration, example attack scenarios and a solution. See below a table of the latest ranking of the highest security concerns in web development:

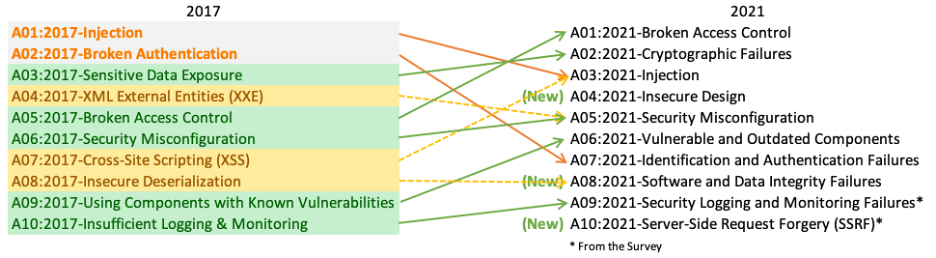


Fig. 1-1: OWASP Top 10 Table, 2017 and 2021

# A01 – Broken Access Control

## Description

Access control is a fundamental component of data security that dictates the permissions of users and what resources they can access and what they cannot. Developers use different methods of authentication and authorization. As such, broken access control is when the IT domain’s access control component is not properly set up; it is prone to attacks where a user can execute actions past their intended permissions. Failures typically lead to unauthorized information disclosure, modification, or destruction of all data or performing a business function outside the user's limits.

Common access control vulnerabilities include:

* Violation of the principle of least privilege or deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone.
* Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool modifying API requests.
* Permitting viewing or editing someone else's account, by providing its unique identifier (insecure direct object references)
* Accessing API with missing access controls for POST, PUT and DELETE.
* Elevation of privilege. Acting as a user without being logged in or acting as an admin when logged in as a user.
* Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, or a cookie or hidden field manipulated to elevate privileges or abusing JWT invalidation.
* CORS misconfiguration allows API access from unauthorized/untrusted origins.
* Force browsing to authenticated pages as an unauthenticated user or to privileged pages as a standard user.

## Position

Broken Access Control has quickly moved up in security concern from fifth place in 2017 to first place currently. Research from OWASP shows that around 94% of applications were tested for some sort of broken access control, with incidents happening around 4% on average. The severity of these attacks also warrants a high position, as someone with elevated privileges can obtain personal information, and even remove it from databases.

## Example Scenarios

Scenario #1

The application uses unverified data in a SQL call that is accessing account information:

 pstmt.setString(1, request.getParameter("acct"));

 ResultSet results = pstmt.executeQuery();

An attacker simply modifies the browser's 'acct' parameter to send whatever account number they want. If not correctly verified, the attacker can access any user's account.

  https://example.com/app/accountInfo?acct=notmyacct

Scenario #2

An attacker simply forces browsers to target URLs. Admin rights are required for access to the admin page.

 https://example.com/app/getappInfo

 https://example.com/app/admin\_getappInfo

If an unauthenticated user or a non-admin can access the admin page, this is a security flaw.

## Solutions

Preventing this attack is done by implementing access control in an environment where the user cannot manipulate metadata or access control checks. Denying requests from outside resources, unless authorized, will stop external requests from tampering with data.

Further precautions include:

* Implementing access control mechanisms once and re-using them throughout the application, including minimizing Cross-Origin Resource Sharing (CORS) usage.
* Unique application business limit requirements should be enforced by domain models.
* Disabling web server directory listing and ensuring file metadata (e.g.,. git) and backup files are not present within web roots.
* Logging access control failures and alert admins when appropriate (e.g., repeated failures).
* Rate limiting API and controller access to minimize the harm from automated attack tooling.
* Stateful session identifiers should be invalidated on the server after logout. Stateless JWT tokens should rather be short-lived so that the window of opportunity for an attacker is minimized. For longer lived JWTs it's highly recommended to follow the OAuth standards to revoke access.
* Model access controls should enforce record ownership rather than accepting that the user can create, read, update, or delete any record.

## Relation

Broken access control is very important in this project. Failure to prevent this security detail could lead to loss of information and massive losses of money. In the ideal situation, all of the solutions mentioned above will be implemented, but unfortunately this is not possible in the timeframe given.

# A02: Cryptographic Failures

## Description

A cryptographic failure is the inefficient encryption of sensitive data, along with non-compliance with the GDPR (General Data Protection Regulation) policy. For example, health records, business information, passwords, etc. have to be properly protected.

Common vulnerabilities include:

* Data being passed along in clear text form
* Outdated or compromised cryptographic algorithms
* Encryption not enforced, missing HTTP headers
* Use of deprecated hashing functions

## Position

Cryptographic Failures have gone up by one position since 2017 as Sensitive Data Exposure. Renaming was done to reflect more on the root cause rather than a direct symptom. The average incidence rate is 4%, with 79% of applications being covered.

## Example Scenarios

Scenario #1

An application encrypts credit card numbers in a database using automatic database encryption. However, this data is automatically decrypted when retrieved, allowing a SQL injection flaw to retrieve credit card numbers in clear text.

Scenario #2

An attacker monitors network traffic (e.g., at an insecure wireless network), downgrades connections from HTTPS to HTTP, intercepts requests, and steals the user's session cookie. The attacker then replays this cookie and hijacks the user's (authenticated) session, accessing or modifying the user's private data. Instead of the above they could alter all transported data, e.g., the recipient of a money transfer.

Scenario #3

The password database uses unsalted or simple hashes to store everyone's passwords. A file upload flaw allows an attacker to retrieve the password database. All the unsalted hashes can be exposed with a rainbow table of pre-calculated hashes. Hashes generated by simple or fast hash functions may be cracked by GPUs, even if they were salted.

## Solutions

Doing the following can improve cryptographic security:

* Classify transmitted data to identify which is sensitive according to privacy laws like GDPR.
* If sensitive data is unnecessary to store, do not store it.
* Privacy laws require data older than 2 years to be removed from databases.
* Encrypt all data in transit with secure protocols such as TLS with forward secrecy (FS) ciphers, cipher prioritization by the server, and secure parameters. Enforce encryption using directives like HTTP Strict Transport Security (HSTS). Do not use legacy protocols such as FTP and SMTP for transporting sensitive data.
* Disable caching for responses that contain sensitive data.
* Apply required security controls as per the data classification.
* Always use authenticated encryption instead of just encryption.
* Ensure that cryptographic randomness is used where appropriate, and that it has not been seeded in a predictable way or with low entropy. Most modern APIs do not require the developer to seed the CSPRNG to get security.
* Avoid deprecated encryptors.

## Relation

Cryptographic Failures is in this project not handled by our system. The auth system used is MetaMask. MetaMask is the link to the crypto wallet the users have. Of course, the solutions are still good to be implemented, but this is not possible in the timeframe given.

# A03: Injection

## Description

An injection is the act of sending a command directly to the database of an application without permission. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) injection. An application is vulnerable to an attack when:

* User-supplied data is not validated, filtered, or sanitized by the application.
* Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter.
* Hostile data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records.
* Hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures.

## Position

Injections have gone down from the first position to third. The incidence rate is on average 3% with a 19% maximum. Since 2017, more safeguards have been put in place to ensure no database commands go through the API.

## Example Scenarios

Scenario #1

An application uses untrusted data in the construction of the following vulnerable SQL call:

String query = "SELECT \\* FROM accounts WHERE custID='" +   request.getParameter("id") + "'";

Scenario #2:

Similarly, an application’s blind trust in frameworks may result in queries that are still vulnerable, (e.g., Hibernate Query Language (HQL):

Query HQLQuery = session.createQuery("FROM accounts WHERE custID='" + request.getParameter("id") + "'");

In both cases, the attacker modifies the ‘id’ parameter value in their browser to send: ‘or ‘1’=’1. For example:

  http://example.com/app/accountView?id=' or '1'='1

This changes the meaning of both queries to return all the records from the accounts table. More dangerous attacks could modify or delete data or even invoke stored procedures.

## Solutions

Preventing an injection can be done via the following approaches:

* Using a safe API, which avoids using the interpreter entirely, provides a parameterized interface, or migrates to Object Relational Mapping Tools (ORMs). This avoids injections using EXECUTE IMMEDIATE or exec() as those can still inject a command even when parameterized.
* Using positive server-side input validation. This is not a complete defense, as many applications require special characters, such as text areas or APIs for mobile applications.
* Using LIMIT and other SQL controls within queries to prevent mass disclosure of records in case of SQL injection.

## Relation to Our Project

Injection is a big problem in the system, but its luckily a problem that is easily solvable by implementing checks. The solutions above will be implemented in the project.

# A04: Insecure Design

## Description

Insecure design is a broad category representing different weaknesses, expressed as “missing or ineffective control design.” Insecure design is not the source for all other Top 10 risk categories. There is a difference between insecure design and insecure implementation. A secure design can still have implementation defects leading to vulnerabilities that may be exploited. An insecure design cannot be fixed by a perfect implementation as by definition, needed security controls were never created to defend against specific attacks.

## Position

Insecure Design is a new topic since the latest edition at the time of writing. Its average incidence rate is 3%, with a maximum of 30%.

## Example Scenarios

Scenario #1

A cinema chain allows group booking discounts and has a maximum of fifteen attendees before requiring a deposit. Attackers could threaten to model this flow and test if they could book six hundred seats and all cinemas at once in a few requests, causing a massive loss of income.

Scenario #2

A retail chain’s e-commerce website does not have protection against bots run by scalpers buying high-end video cards to resell auction websites. This creates terrible publicity for the video card makers and retail chain owners and enduring bad blood with enthusiasts who cannot obtain these cards at any price. Careful anti-bot design and domain logic rules, such as purchases made within a few seconds of availability, might identify inauthentic purchases and rejected such transactions

## Solutions

An application’s design can be further improved and secured with the following methods:

* Establish and use a secure development lifecycle with AppSec professionals to help evaluate and design security and privacy-related controls.
* Establish and use a library of secure design patterns or paved road ready to use components.
* Use threat modeling for critical authentication, access control, business logic, and key flows.
* Integrate security language and controls into user stories.
* Integrate plausibility checks at each tier of your application (from frontend to backend).
* Write unit and integration tests to validate that all critical flows are resistant to the threat model. Compile use-cases *and* misuse-cases for each tier of your application.
* Segregate tier layers on the system and network layers depending on the exposure and protection needs.
* Segregate tenants robustly by design throughout all tiers.
* Limit resource consumption by user or service.

## Relation

In the initial design of the system, some of the points above have been thought about. However, at the time of creating this document, the finish of the project is nearing. Security points will still be picked up here and there, but the main focus will be somewhere else like the LO’s.

# A05: Security Misconfiguration

## Description

Security misconfiguration is inaccurately handled security in any part of the application stack, or improper configuration.

## Position

Security Misconfigurations have gone up from 6. They have an average incidence rate of 4.5% with a maximum of ~20%.

## Example Scenarios

Scenario #1

The application server comes with sample applications not removed from the production server. These sample applications have known security flaws attackers use to compromise the server. Suppose one of these applications is the admin console, and default accounts weren't changed. In that case, the attacker logs in with default passwords and takes over.

Scenario #2

Directory listing is not disabled on the server. An attacker discovers they can simply list directories. The attacker finds and downloads the compiled Java classes, which they decompile and reverse engineer to view the code. The attacker then finds a severe access control flaw in the application.

Scenario #3

The application server's configuration allows detailed error messages, e.g., stack traces, to be returned to users. This potentially exposes sensitive information or underlying flaws such as component versions that are known to be vulnerable.

Scenario #4

A cloud service provider (CSP) has default sharing permissions open to the Internet by other CSP users. This allows sensitive data stored within cloud storage to be accessed.

## Solutions

Secure installation processes should be implemented, including:

* A repeatable hardening process makes it fast and easy to deploy another environment that is appropriately locked down. Development, QA, and production environments should all be configured identically, with different credentials used in each environment. This process should be automated to minimize the effort required to set up a new secure environment.
* A minimal platform without any unnecessary features, components, documentation, and samples. Remove or do not install unused features and frameworks.
* A task to review and update the configurations appropriate to all security notes, updates, and patches as part of the patch management process. Review cloud storage permissions.
* A segmented application architecture provides effective and secure separation between components or tenants, with segmentation, containerization, or cloud security groups (ACLs).
* Sending security directives to clients, e.g., Security Headers.
* An automated process to verify the effectiveness of the configurations and settings in all environments.

## Relation

To ensure security is properly structured and configured, configurations will be regularly tested. Also our backend has a microservice setup, so all services are segmented and can be containerized.

# A06: Vulnerable and Outdated Components

## Description

If an application is using outdated or deprecated dependencies, it is vulnerable to an attack, depending on the component. This is a common issue if developers do not test for compatibility of updated or patched libraries and scanning for vulnerabilities is not done often.

## Position

Vulnerable and Outdated Components have gone up from number 9. Interestingly, this is number 2 according to an OWASP-organized top 10 community survey. It has an average incidence rate of 8%, with a maximum of 28%.

## Example Scenarios

Scenario #1

Components typically run with the same privileges as the application itself, so flaws in any component can result in serious impact. Such flaws can be accidental (e.g., coding error) or intentional (e.g., a backdoor in a component). Some example exploitable component vulnerabilities discovered are:

* CVE-2017-5638, a Struts 2 remote code execution vulnerability that enables the execution of arbitrary code on the server, has been blamed for several breaches.
* While the internet of things (IoT) is frequently difficult or impossible to patch, the importance of patching them can be great (e.g., biomedical devices).

There are automated tools to help attackers find unpatched or misconfigured systems. For example, the Shodan IoT search engine can help you find devices that still suffer from Heartbleed vulnerability patched in April 2014.

## Solutions

There should be a patch management process in place to:

* Remove unused dependencies, unnecessary features, components, files, and documentation.
* Continuously inventory the versions of both client-side and server-side components (e.g., frameworks, libraries) and their dependencies using tools like versions, OWASP Dependency Check, retire.js, etc. Continuously monitor sources like Common Vulnerability and Exposures (CVE) and National Vulnerability Database (NVD) for vulnerabilities in the components. Use software composition analysis tools to automate the process. Subscribe to email alerts for security vulnerabilities related to components you use.
* Only obtain components from official sources over secure links. Prefer signed packages to reduce the chance of including a modified, malicious component (See A08:2021-Software and Data Integrity Failures).
* Monitor for libraries and components that are unmaintained or do not create security patches for older versions. If patching is not possible, consider deploying a virtual patch to monitor, detect, or protect against the discovered issue.

## Relation

This project will use several dependencies, both in the backend and frontend. The dependencies will be looked through and will be updated whenever necessary. The credibility and secureness of each dependency will be looked at if there is any time left over.

# A07: Identification and Authentication Failures

## Description

Confirmation of the user's identity, authentication, and session management is critical to protect against authentication-related attacks. This topic is concerned with weak authentication that does not solidify the mentioned security concerns. An authentication is weak if it:

* Permits automated attacks like credential stuffing, or weak passwords like “password1”.
* Stores passwords in unencrypted plain text or with weak hashing.
* Exposes the session ID in the URL.
* Does not correctly validate session ID’s.

## Position

Identification and Authentication failures have gone down from 2nd place in 2017, currently with an average incidence rate of ~2.55% with a maximum of 14%

## Example Scenarios

Scenario #1

Credential stuffing, the use of lists of known passwords, is a common attack. Suppose an application does not implement automated threat or credential stuffing protection. In that case, the application can be used as a password oracle to determine if the credentials are valid.

Scenario #2

Most authentication attacks occur due to the continued use of passwords as a sole factor. Once considered best practices, password rotation and complexity requirements encourage users to use and reuse weak passwords. Organizations are recommended to stop these practices per NIST 800-63 and use multi-factor authentication.

Scenario #3

Application session timeouts are not set correctly. A user uses a public computer to access an application. Instead of selecting "logout," the user simply closes the browser tab and walks away. An attacker uses the same browser an hour later, and the user is still authenticated.

## Solutions

* Where possible, implement multi-factor authentication to prevent automated credential stuffing, brute force, and stolen credential reuse attacks.
* Do not ship or deploy with any default credentials or secrets, particularly for admin users.
* Implement checks for weak passwords.
* Ensure registration, credential recovery, and API pathways are hardened against account enumeration attacks by using the same messages for all outcomes.
* Limit or increasingly delay failed login attempts, but be careful not to create a denial of service scenario. Log all failures and alert administrators when credential stuffing, brute force, or other attacks are detected.
* Use a server-side, secure, built-in session manager that generates a new random session ID with high entropy after login. Session identifier should not be in the URL, should be securely stored, and invalidated after logout, idle, and absolute timeouts.

## Relation

The project will use MetaMask for authentication. Each account has a crypto wallet with an address. MetaMask does not have any major hacking incidents at the time of writing.

# A08: Software and Data Integrity Failures

## Description

Software and data integrity failures relate to code and infrastructure that does not protect against integrity violations. An example of this is where an application relies upon plugins, libraries, or modules from untrusted sources, repositories, and content delivery networks (CDNs). Another example is an insecure CI/CD pipeline, which will not be able to properly confirm if a system is secure enough to defend itself from malicious code or unauthorized access.

## Position

Software and Data Integrity Failure is a new topic since the latest edition at the time of writing.

## Example Scenarios

Scenario #1

Many home routers, set-top boxes, device firmware, and others do not verify updates via signed firmware. Unsigned firmware is a growing target for attackers and is expected to only get worse. This is a major concern as many times there is no mechanism to remediate other than to fix in a future version and wait for previous versions to age out.

Scenario #2

Nation-states have been known to attack update mechanisms, with a recent notable attack being the SolarWinds Orion attack. The company that develops the software had secure build and update integrity processes. Still, these were able to be subverted, and for several months, the firm distributed a highly targeted malicious update to more than 18,000 organizations, of which around 100 or so were affected. This is one of the most far-reaching and most significant breaches of this nature in history

## Solutions

* Use digital signatures or similar mechanisms to verify the software or data is from the expected source and has not been altered.
* Ensure libraries and dependencies (npm, Maven, etc.) are consuming trusted repositories.
* Ensure that your CI/CD pipeline has proper segregation, configuration, and access control to ensure the integrity of the code flowing through the build and deploy processes.
* Ensure that unsigned or unencrypted serialized data is not sent to untrusted clients without some form of integrity check or digital signature to detect tampering or replay of the serialized data

## Relation

For our project security we expect to contain very few dependencies, of which all are trusted. We are planning to develop a CI/CD pipeline for each microservice. That being said, we already have planned security precautions for access control, segregation and configuration, as seen in previous topics. A digital signature is considered to validate untrusted clients, but for now this topic is a low priority.

# A09: Security Logging and Monitoring Failures

## Description

This topic refers to ineffective logging and monitoring, which is a detriment in the case of an active breach. Without proper monitoring systems, a breach cannot be detected and without proper logging, the source of the attack is unknown. An attack like this can occur any time:

* Auditable events, such as logins, failed logins, and high-value transactions, are not logged.
* Warnings and errors generate no, inadequate, or unclear log messages.
* Logs of applications and APIs are not monitored for suspicious activity.
* Logs are only stored locally.
* Appropriate alerting thresholds and response escalation processes are not in place or effective.
* Penetration testing and scans by dynamic application security testing (DAST) tools (such as OWASP ZAP) do not trigger alerts.
* The application cannot detect, escalate, or alert for active attacks in real-time or near real-time.

## Position

Security Logging and Monitoring Failures have gone up by one position. Interestingly, it scored third in the Top 10 Community survey in 2017. This issue is expected to fall in positions as monitoring technology improves. It has an average incidence rate of 6% with a maximum of ~20%.

## Example Scenarios

Scenario #1

A major Indian airline had a data breach involving more than ten years' worth of personal data of millions of passengers, including passport and credit card data. The data breach occurred at a third-party cloud hosting provider, who notified the airline of the breach after some time.

Scenario #2

A major European airline suffered a GDPR reportable breach. The breach was reportedly caused by payment application security vulnerabilities exploited by attackers, who harvested more than 400,000 customer payment records. The airline was fined 20 million pounds as a result by the privacy regulator.

## Solutions

* Ensure all login, access control, and server-side input validation failures can be logged with sufficient user context to identify suspicious or malicious accounts and held for enough time to allow delayed forensic analysis.
* Ensure that logs are generated in a format that log management solutions can easily consume.
* Ensure log data is encoded correctly to prevent injections or attacks on the logging or monitoring systems.
* Ensure high-value transactions have an audit trail with integrity controls to prevent tampering or deletion, such as append-only database tables or similar.
* DevSecOps teams should establish effective monitoring and alerting such that suspicious activities are detected and responded to quickly.

## Relation

We are planning to have an event sourcing database in place, so all login and access control commands are saved. This way the system has a single point of truth where admins know the cause of a crash, and since it is kept in a database, it can be easily managed and queried.  Unfortunately, we do not have the time to implement effective monitoring and we do not have enough manpower to assign a DevSecOps position to a person.

# A10: Server-Side Request Forgery (SSRF)

## Description

SSRF flaws occur whenever a web application is fetching a remote resource without validating the user-supplied URL. This allows an attacker to create a crafter request to the system, while bypassing any firewalls. As modern web applications provide end-users with convenient features, fetching a URL becomes a common scenario. As a result, the incidence of SSRF is increasing.

## Position

SSRF is a new topic since the latest edition at the time of writing. It has an average incidence rate of 2.7% with a maximum not yet reached.

## Example Scenarios

Scenario #1

Compromise internal services – The attacker can abuse internal services to conduct further attacks such as Remote Code Execution (RCE) or Denial of Service (DoS).

Scenario #2

Accessing metadata storage of cloud services – Most cloud providers have metadata storage such as http://169.254.169.254/. An attacker can read the metadata to gain sensitive information. This security issue can also be used to gain sensitive information.

## Solutions

* Remote access functionality can be put into separate layers from the network, which will leave less information accessible for an SSRF.
* Enforce “deny by default” firewall policies or network access control rules to block all but essential intranet traffic.
* Sanitize user-input data from the application layer.
* Do not send raw responses to clients
* Disable HTTP redirects

## Relation

Due to the structure of our project, layers are separated from each other, so an SSRF attack would be less detrimental. Furthermore, we plan to put in place strict policies with CORS and unknown networks, as well as sanitize input data for other topics (ex. injection).

# Which security precaution would be most beneficial for our project?

The reason for this investigation is to specify what concerns we as a group want to focus on the most to ensure the safety of clients’ information and our application’s robustness. Of course, it would be best to account for all of these security concerns, but unfortunately time constraints do not allow for this.

As such, a priority table has been made to display what we believe is best to prioritize for our project. Our choices are made based on the case of our project, as well as the incidence rate, severity and how much time would be invested in implementing a solution:

|  |  |  |
| --- | --- | --- |
| OWASP ID | OWASP Name | Priority |
| A01 | Broken Access Control | HIGH |
| A02 | Cryptographic Failures | HIGH |
| A03 | Injection | HIGH |
| A04 | Insecure Design | MEDIUM |
| A05 | Security Misconfiguration | LOW |
| A06 | Vulnerable and Outdated Components | LOW |
| A07 | Identification and Authentication Failures | MEDIUM |
| A08 | Software and Data Integrity Failures | LOW |
| A09 | Security Logging and Monitoring Failures | LOW |
| A10 | Server-Side Request Forgery | MEDIUM |