CRYPTO M5 (p-1)

techworldthink • March 20, 2022

Common web application security vulnerabilities

Injection flaws

Injection flaws result from a classic failure to filter untrusted input. It can happen when you pass unfiltered data to the SQL server (SQL injection), to the browser (XSS), to the LDAP server (LDAP injection), or anywhere else. The problem here is that the attacker can inject commands to these entities, resulting in loss of data and hijacking clients' browsers.

The good news is that protecting against injection is "simply" a matter of filtering your input properly and thinking about whether an input can be trusted. But the bad news is that *all* input needs to be properly filtered, unless it can unquestionably be trusted

A SQL injection attack consists of insertion or "injection" of a SQL query via the input data from the client to the application.

A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to effect the execution of predefined SQL commands.

Here is a basic HTML login form with two inputs: `username` and `password`.

```
<form method="post" action="/login">
<input name="username" type="text" id="username">
<input name="password" type="password" id="password">
</form>
```

The common way for the `/login` to work is by building a database query. If the variables `$request.username` and `$request.password` are requested directly from the user's input, this can be compromised.

```
SELECT id
FROM Users
WHERE username = '$request.username'
AND password = '$request.password'
```

For example, if a user inserts `admin' or 1=1 --` as the username, he/she will bypass the login form without providing a valid username/password combination.

```
SELECT id
FROM Users
WHERE username = 'admin' or 1=1--
AND password = 'request.password'
```

The issue is that the `'` in the `username` closes out the `username` field, then the `--` starts a SQL comment causing the database server to ignore the rest of the string. As the inputs of the web application are not sanitized, the query has been modified in a malicious way.

How to Prevent SQL Injection

The source of the problem of SQL Injection (the most important injection risk) is based on SQL queries that use untrusted data without the use of parametrized queries (without `PreparedStatement` in Java environments).

First of all Hdiv minimizes the existence of untrusted data thanks to the web information flow control system that avoids the manipulation of the data generated on the server side. This architecture minimizes the risk to just the new data generated legally from editable form elements. It's important to note that even using `PreparedStatement` if the query is based on untrusted data generated previously at server side (for instance the identification id of an item within a list) it's possible to exist a SQL Injection risk.

Although `PreparedStatement` solves the most of the cases, there are some SQL keywords that can not be used with `PreparedStatement`, such as `ORDER BY`. In these cases, you have to concatenate the column name and the order to the SQL

query but only after verifying that the column name and order are valid in this context and sanitising them to counter any attempt of SQL Injection attack.

Broken authentication

Broken authentication is #7 on the latest (2021) OWASP Top 10 list. Broken authentication is typically caused by poorly implemented authentication and session management functions. Broken authentication attacks aim to take over one or more accounts giving the attacker the same privileges as the attacked user. Authentication is “broken” when attackers are able to compromise passwords, keys or session tokens, user account information, and other details to assume user identities.

Due to poor design and implementation of identity and access controls, the prevalence of broken authentication is widespread. Common risk factors include:

Predictable login credentials

User authentication credentials that are not protected when stored

Session IDs exposed in the URL (e.g., URL rewriting)

Session IDs vulnerable to session fixation attacks

Session value that does not time out or get invalidated after logout

Session IDs that are not rotated after successful login

Passwords, session IDs, and other credentials sent over unencrypted connections

Sensitive data exposure

Sensitive Data Exposure occurs when an organization unknowingly exposes sensitive data or when a security incident leads to the accidental or unlawful destruction, loss, alteration, or unauthorized disclosure of, or access to sensitive data. Such Data exposure may occur as a result of inadequate protection of a database, misconfigurations when bringing up new instances of datastores, inappropriate usage of data systems and more.

Such Data exposure may occur as a result of inadequate protection of a database, misconfigurations when bringing up new instances of datastores, inappropriate usage

of data systems and more.

Sensitive Data Exposure can of the following three types:

- Confidentiality Breach: where there is unauthorized or accidental disclosure of, or access to, sensitive data.
- Integrity Breach: where there is an unauthorized or accidental alteration of sensitive data.
- Availability Breach: where there is an unauthorised or accidental loss of access to, or destruction of, sensitive data. This will include both the permanent and temporary loss of sensitive data.

Organizations that collect sensitive data are responsible for its protection and failure to do so can lead to heavy fines and penalties.

How to Protect Yourself From Data Exposure?

- Catalog Data

In order to protect their consumers data, organizations need to make sure they keep track of all the data stored within their systems and perform an audit. This will give them a clear picture of owners, locations, security and governance measures enabled on the data.

- Assess Risks Associated to Data

In order to protect data, organizations need to have a clear understanding of the data risk and allocate budgets & resources for risk mitigation activities accordingly. The more sensitive the data is, the higher the risk of harm will be. Even a small amount of highly sensitive data can have a high impact on data subjects.

- Appropriate security controls

Organizations must have appropriate security controls in place to avoid the occurrence of sensitive data exposures as well as to limit their impacts on data subjects.

- Instant Action

Organizations must have an effective breach response mechanism in place to immediately respond to sensitive data exposure.

XML External Entities (XXE)

XML external entity injection (also known as XXE) is a web security vulnerability that allows an attacker to interfere with an application's processing of XML data. It often allows an attacker to view files on the application server filesystem, and to interact with any back-end or external systems that the application itself can access.

In some situations, an attacker can escalate an XXE attack to compromise the underlying server or other back-end infrastructure, by leveraging the XXE vulnerability to perform server-side request forgery (SSRF) attacks.

Some applications use the XML format to transmit data between the browser and the server. Applications that do this virtually always use a standard library or platform API to process the XML data on the server. XXE vulnerabilities arise because the XML specification contains various potentially dangerous features, and standard parsers support these features even if they are not normally used by the application.

There are various types of XXE attacks:

- Exploiting XXE to retrieve files, where an external entity is defined containing the contents of a file, and returned in the application's response.
- Exploiting XXE to perform SSRF attacks, where an external entity is defined based on a URL to a back-end system.
- Exploiting blind XXE exfiltrate data out-of-band, where sensitive data is transmitted from the application server to a system that the attacker controls.
- Exploiting blind XXE to retrieve data via error messages, where the attacker can trigger a parsing error message containing sensitive data.

How to prevent XXE vulnerabilities

Virtually all XXE vulnerabilities arise because the application's XML parsing library supports potentially dangerous XML features that the application does not need or

intend to use. The easiest and most effective way to prevent XXE attacks is to disable those features.

Generally, it is sufficient to disable resolution of external entities and disable support for `XInclude`. This can usually be done via configuration options or by programmatically overriding default behavior. Consult the documentation for your XML parsing library or API for details about how to disable unnecessary capabilities.

Broken access control

Access control enforces policy such that users cannot act outside of 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.

How to Prevent

Access control is only effective in trusted server-side code or server-less API, where the attacker cannot modify the access control check or metadata.

- Except for public resources, deny by default.
- Implement access control mechanisms once and re-use them throughout the application, including minimizing Cross-Origin Resource Sharing (CORS) usage.
- Model access controls should enforce record ownership rather than accepting that the user can create, read, update, or delete any record.
- Unique application business limit requirements should be enforced by domain models.
- Disable web server directory listing and ensure file metadata (e.g., .git) and backup files are not present within web roots.
- Log access control failures, alert admins when appropriate (e.g., repeated failures).
- Rate limit 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 highy recommended to follow the OAuth standards to revoke access.

Security misconfiguration

The application might be vulnerable if the application is:

- Missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services.
- Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, or privileges).
- Default accounts and their passwords are still enabled and unchanged.

- Error handling reveals stack traces or other overly informative error messages to users.
- For upgraded systems, the latest security features are disabled or not configured securely.
- The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values.
- The server does not send security headers or directives, or they are not set to secure values.
- The software is out of date or vulnerable

Without a concerted, repeatable application security configuration process, systems are at a higher risk.

How to Prevent

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 (e.g., S3 bucket 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.

Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted websites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it.

An attacker can use XSS to send a malicious script to an unsuspecting user. The end user's browser has no way to know that the script should not be trusted, and will execute the script. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser and used with that site. These scripts can even rewrite the content of the HTML page. For more details on the different types of XSS flaws

Cross-Site Scripting (XSS) attacks occur when:

1. Data enters a Web application through an untrusted source, most frequently a web request.
2. The data is included in dynamic content that is sent to a web user without being validated for malicious content.

The malicious content sent to the web browser often takes the form of a segment of JavaScript, but may also include HTML, Flash, or any other type of code that the browser may execute. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data, like cookies or other session information, to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site.

The consequence of an XSS attack is the same regardless of whether it is stored or reflected (or DOM Based). The difference is in how the payload arrives at the server. Do not be fooled into thinking that a "read-only" or "brochureware" site is not vulnerable to serious reflected XSS attacks. XSS can cause a variety of problems for the end user that range in severity from an annoyance to complete account compromise. The most severe XSS attacks involve disclosure of the user's session

cookie, allowing an attacker to hijack the user's session and take over the account. Other damaging attacks include the disclosure of end user files, installation of Trojan horse programs, redirect the user to some other page or site, or modify presentation of content. An XSS vulnerability allowing an attacker to modify a press release or news item could affect a company's stock price or lessen consumer confidence. An XSS vulnerability on a pharmaceutical site could allow an attacker to modify dosage information resulting in an overdose

Insecure deserialization

Serialization is the process of converting complex data structures, such as objects and their fields, into a "flatter" format that can be sent and received as a sequential stream of bytes. Serializing data makes it much simpler to:

- Write complex data to inter-process memory, a file, or a database
- Send complex data, for example, over a network, between different components of an application, or in an API call

Crucially, when serializing an object, its state is also persisted. In other words, the object's attributes are preserved, along with their assigned values.

Deserialization is the process of restoring this byte stream to a fully functional replica of the original object, in the exact state as when it was serialized. The website's logic can then interact with this deserialized object, just like it would with any other object.

Many programming languages offer native support for serialization. Exactly how objects are serialized depends on the language. Some languages serialize objects into binary formats, whereas others use different string formats, with varying degrees of human readability. Note that all of the original object's attributes are stored in the serialized data stream, including any private fields. To prevent a field from being serialized, it must be explicitly marked as "transient" in the class declaration.

Be aware that when working with different programming languages, serialization may be referred to as marshalling (Ruby) or pickling (Python). These terms are synonymous with "serialization" in this context.

Insecure deserialization is when user-controllable data is deserialized by a website. This potentially enables an attacker to manipulate serialized objects in order to pass harmful data into the application code.

It is even possible to replace a serialized object with an object of an entirely different class. Alarmingly, objects of any class that is available to the website will be deserialized and instantiated, regardless of which class was expected. For this reason, insecure deserialization is sometimes known as an "object injection" vulnerability.

An object of an unexpected class might cause an exception. By this time, however, the damage may already be done. Many deserialization-based attacks are completed **before** deserialization is finished. This means that the deserialization process itself can initiate an attack, even if the website's own functionality does not directly interact with the malicious object. For this reason, websites whose logic is based on strongly typed languages can also be vulnerable to these techniques.

Insecure deserialization typically arises because there is a general lack of understanding of how dangerous deserializing user-controllable data can be. Ideally, user input should never be deserialized at all.

However, sometimes website owners think they are safe because they implement some form of additional check on the deserialized data. This approach is often ineffective because it is virtually impossible to implement validation or sanitization to account for every eventuality. These checks are also fundamentally flawed as they rely on checking the data after it has been deserialized, which in many cases will be too late to prevent the attack.

Vulnerabilities may also arise because deserialized objects are often assumed to be trustworthy. Especially when using languages with a binary serialization format, developers might think that users cannot read or manipulate the data effectively. However, while it may require more effort, it is just as possible for an attacker to exploit binary serialized objects as it is to exploit string-based formats.

Deserialization-based attacks are also made possible due to the number of dependencies that exist in modern websites. A typical site might implement many different libraries, which each have their own dependencies as well. This creates a massive pool of classes and methods that is difficult to manage securely. As an attacker can create instances of any of these classes, it is hard to predict which methods can be invoked on the malicious data. This is especially true if an attacker is able to chain together a long series of unexpected method invocations, passing data

into a sink that is completely unrelated to the initial source. It is, therefore, almost impossible to anticipate the flow of malicious data and plug every potential hole.

In short, it can be argued that it is not possible to securely deserialize untrusted input.

The impact of insecure deserialization can be very severe because it provides an entry point to a massively increased attack surface. It allows an attacker to reuse existing application code in harmful ways, resulting in numerous other vulnerabilities, often remote code execution.

Even in cases where remote code execution is not possible, insecure deserialization can lead to privilege escalation, arbitrary file access, and denial-of-service attacks

How to prevent insecure deserialization vulnerabilities

Generally speaking, deserialization of user input should be avoided unless absolutely necessary. The high severity of exploits that it potentially enables, and the difficulty in protecting against them, outweigh the benefits in many cases.

If you do need to deserialize data from untrusted sources, incorporate robust measures to make sure that the data has not been tampered with. For example, you could implement a digital signature to check the integrity of the data. However, remember that any checks must take place **before** beginning the deserialization process. Otherwise, they are of little use.

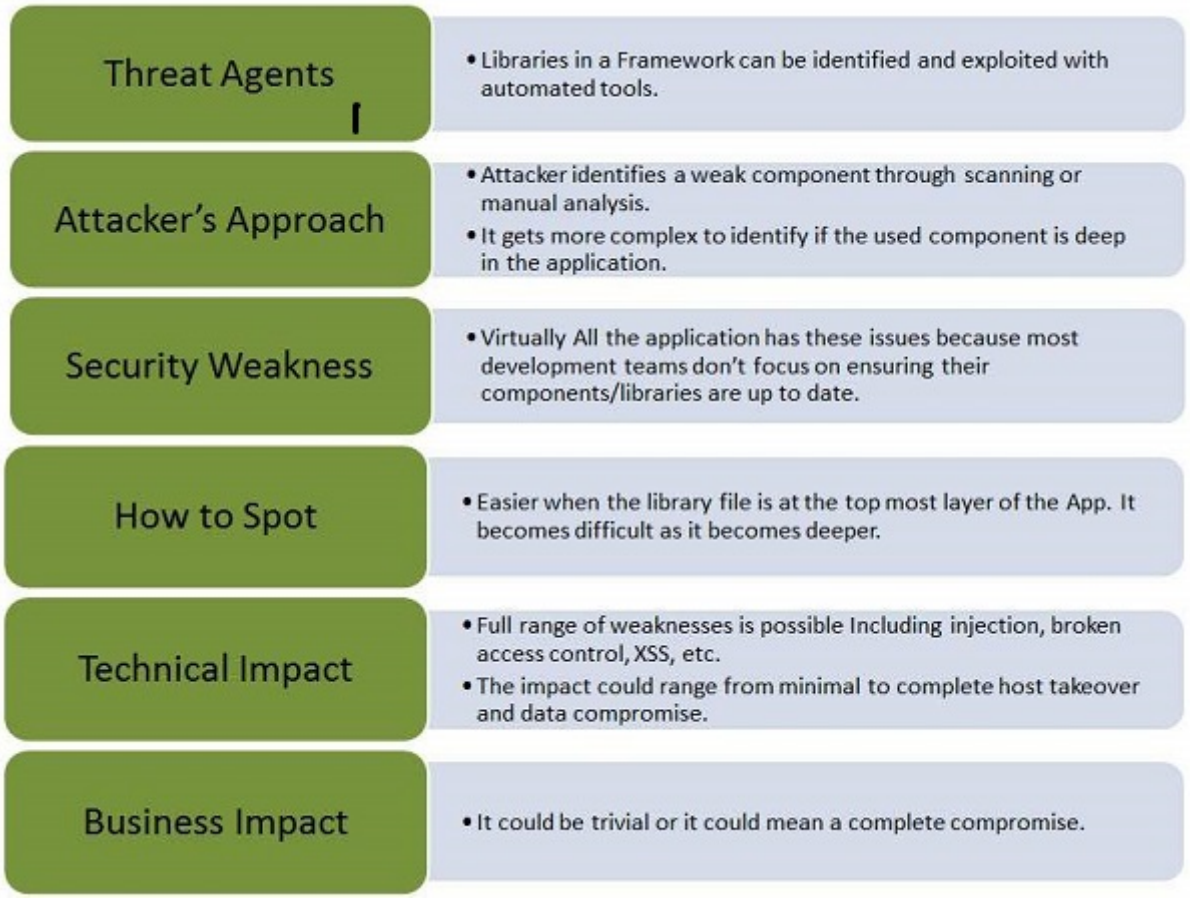
If possible, you should avoid using generic deserialization features altogether. Serialized data from these methods contains all attributes of the original object, including private fields that potentially contain sensitive information. Instead, you could create your own class-specific serialization methods so that you can at least control which fields are exposed.

Finally, remember that the vulnerability is the deserialization of user input, not the presence of gadget chains that subsequently handle the data. Don't rely on trying to eliminate gadget chains that you identify during testing. It is impractical to try and plug them all due to the web of cross-library dependencies that almost certainly exist on your website. At any given time, publicly documented memory corruption exploits are also a factor, meaning that your application may be vulnerable regardless.

Using components with known vulnerabilities

This kind of threat occurs when the components such as libraries and frameworks used within the app almost always execute with full privileges. If a vulnerable component is exploited, it makes the hacker's job easier to cause a serious data loss or server takeover.

Let us understand Threat Agents, Attack Vectors, Security Weakness, Technical Impact and Business Impacts of this flaw with the help of simple diagram.



using_components_with_known_vulnerabilities

Example

The following examples are of using components with known vulnerabilities −

- Attackers can invoke any web service with full permission by failing to provide an identity token.
- Remote-code execution with Expression Language injection vulnerability is introduced through the Spring Framework for Java based apps.

Preventive Mechanisms

- Identify all components and the versions that are being used in the webapps not just restricted to database/frameworks.

- Keep all the components such as public databases, project mailing lists etc. up to date.
- Add security wrappers around components that are vulnerable in nature.

Insufficient logging & monitoring

Logs give visibility into an organization's activities. Logs and audit trails generated enables an organization to troubleshoot, track events, detect incidents and maintain regulatory requirements. Insufficient logging and monitoring is, missing security critical information logs or lack of proper log format, context, storage, security and timely response to detect an incident or breach.

According to the 2020 IBM breach report, the average time to detect and contain a data breach is 280 days. Logs are an important part of incident response. Organization may be blindsided to a breach which can go undetected with irreparable regulatory, financial and legal issues. Proper log management will ensure faster breach detection and mitigation that will save business time, money and reputation.

How does Insufficient Logging & Monitoring attacks impact business?

Confidentiality: Logs contain sensitive information and that can be accessed by an attacker

Integrity: Allowing unsanitized input to log files, attackers might tamper with log files and corrupt, inject unexpected inputs, change entries (CWE-117).

Availability: Logging everything can overload the system causing denial of service, business disruption could happen due to security incident or breach

Non repudiation: The source of the attack may not be traceable and may lead to system compromise, future attacks

Accountability: Missing audit trails

Security incidents could be mitigated with proper log collection and monitoring. Sufficient logging can even mitigate APTs, ransomwares, malwares, insider threat,

DOS, dns attacks etc.

How to prevent a Insufficient Logging & Monitoring attack?

- Perform a baseline of logs needed for business which includes access logs, failed logins, suspicious or anomalous activities, network, endpoints, cloud etc.
- Log formatted properly and context of logs is clearly understood
- Have a centralized log management system where all logs are collected in one place like a SIEM tool integrated with real time reporting, heuristics and visualization tools
- Synchronize time (UTC)
- Secure the logs
- Store the logs in accordance with the compliance and business requirements
- Properly monitor user activity, anomalous behavior with automation and alerting
- Log review should be closely monitored
- Logs should not be deleted or modified
- Integrate SIEM with SOC to improve threat detection and visibility
- Legacy systems to cloud environments must be continuously monitored
- Anomalous activity or any incident must be timely reported and action must be taken
- Have an incident response plan following NIST 800-61 rev2 or later
- Follow standards NIST 800-92, CIS control 6 and ISO27001
- Perform pentesting and DAST tools to check to see where insufficient logging and monitoring has occurred