web application vulnerabilities

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.

Description

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.

SQL Injection

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 affect the execution of predefined SQL commands.

Description

SQL injection attack occurs when:

1. An unintended data enters a program from an untrusted source.
2. The data is used to dynamically construct a SQL query

The main consequences are:

- **Confidentiality**: Since SQL databases generally hold sensitive data, loss of confidentiality is a frequent problem with SQL Injection vulnerabilities.
- **Authentication**: If poor SQL commands are used to check user names and passwords, it may be possible to connect to a system as another user with no previous knowledge of the password.
- **Authorization**: If authorization information is held in a SQL database, it may be possible to change this information through the successful exploitation of a SQL Injection vulnerability.
- **Integrity**: Just as it may be possible to read sensitive information, it is also possible to make changes or even delete this information with a SQL Injection attack.

Cross Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. With a little help of social engineering (such as sending a link via email or chat), an attacker may trick the users of a web application into executing actions of the attacker's choosing. If the victim is a normal user, a successful CSRF attack can force the user to perform state changing requests like transferring funds, changing their email address, and so forth. If the victim is an administrative account, CSRF can compromise the entire web application.

Description

CSRF is an attack that tricks the victim into submitting a malicious request. It inherits the identity and privileges of the victim to perform an undesired function on the victim's behalf (though note that this is not true of login CSRF, a special form of the attack described below). For most sites, browser requests automatically include any credentials associated with the site, such as the user's session cookie, IP address, Windows domain credentials, and so forth. Therefore, if the user is currently authenticated to the site, the site will have no way to distinguish between the forged request sent by the victim and a legitimate request sent by the victim.

CSRF attacks target functionality that causes a state change on the server, such as changing the victim's email address or password, or purchasing something. Forcing the victim to retrieve data doesn't benefit an attacker because the attacker doesn't receive the response, the victim does. As such, CSRF attacks target state-changing requests.

An attacker can use CSRF to obtain the victim's private data via a special form of the attack, known as login CSRF. The attacker forces a non-authenticated user to log in to an account the attacker controls. If the victim does not realize this, they may add personal data—such as credit card information—to the account. The attacker can then log back into the account to view this data, along with the victim's activity history on the web application.

It's sometimes possible to store the CSRF attack on the vulnerable site itself. Such vulnerabilities are called “stored CSRF flaws". This can be accomplished by simply storing an IMG or IFRAME tag in a field that accepts HTML, or by a more complex cross-site scripting attack. If the attacker can store a CSRF attack in the site, the severity of the attack is amplified. In particular, the likelihood is increased because the victim is more likely to view the page containing the attack than some random page on the Internet. The likelihood is also increased because the victim is sure to be authenticated to the site already.

Server Side Request Forgery

In a Server-Side Request Forgery (SSRF) attack, the attacker can abuse functionality on the server to read or update internal resources. The attacker can supply or modify a URL which the code running on the server will read or submit data to, and by carefully selecting the URLs, the attacker may be able to read server configuration such as AWS metadata, connect to internal services like http enabled databases or perform post requests towards internal services which are not intended to be exposed.

Description

The target application may have functionality for importing data from a URL, publishing data to a URL or otherwise reading data from a URL that can be tampered with. The attacker modifies the calls to this functionality by supplying a completely different URL or by manipulating how URLs are built (path traversal etc.).

When the manipulated request goes to the server, the server-side code picks up the manipulated URL and tries to read data to the manipulated URL. By selecting target URLs the attacker may be able to read data from services that are not directly exposed on the internet:

- Cloud server meta-data - Cloud services such as AWS provide a REST interface on http://169.254.169.254/ where important configuration and sometimes even authentication keys can be extracted.
- Database HTTP interfaces - NoSQL databases such as MongoDB provide REST interfaces on HTTP ports. If the database is expected to only be available internally, authentication may be disabled and the attacker can extract data.
- Internal REST interfaces.
- Files - The attacker may be able to read files using file:// URIs.

The attacker may also use this functionality to import untrusted data into code that expects to only read data from trusted sources, and as such circumvent input validation

XML External Entity (XXE) Processing

Description

An *XML External Entity* attack is a type of attack against an application that parses XML input. This attack occurs when XML input containing a reference to an external entity is processed by a weakly configured XML parser. This attack may lead to the disclosure of confidential data, denial of service, server side request forgery, port scanning from the perspective of the machine where the parser is located, and other system impacts.

The XML 1.0 standard defines the structure of an XML document. The standard defines a concept called an entity, which is a storage unit of some type. There are a few different types of entities, external general/parameter parsed entities often shortened to **external entities**, that can access local or remote content via a declared system identifier. The system identifier is assumed to be a URI that can be dereferenced (accessed) by the XML processor when processing the entity. The XML processor then replaces occurrences of the named external entity with the contents dereferenced by the system identifier. If the system identifier contains tainted data and the XML processor dereferences this tainted data, the XML processor may disclose confidential information normally not accessible by the application. Similar attack vectors apply the usage of external DTDs, external stylesheets, external schemas, etc. which, when included, allow similar external resource inclusion style attacks.

Attacks can include disclosing local files, which may contain sensitive data such as passwords or private user data, using file: schemes or relative paths in the system identifier. Since the attack occurs relative to the application processing the XML document, an attacker may use this trusted application to pivot to other internal systems, possibly disclosing other internal content via http(s) requests or launching a CSRF attack to any unprotected internal services. In some situations, an XML processor library that is vulnerable to client-side memory corruption issues may be exploited by dereferencing a malicious URI, possibly allowing arbitrary code execution under the application account. Other attacks can access local resources that may not stop returning data, possibly impacting application availability if too many threads or processes are not released.

Note that the application does not need to explicitly return the response to the attacker for it to be vulnerable to information disclosures. An attacker can leverage DNS information to exfiltrate data through subdomain names to a DNS server that they control.

Insecure Direct Object References

Insecure Direct Object References (IDOR) occur when an application provides direct access to objects based on user-supplied input. As a result of this vulnerability attackers can bypass authorization and access resources in the system directly, for example database records or files. Insecure Direct Object References allow attackers to bypass authorization and access resources directly by modifying the value of a parameter used to directly point to an object. Such resources can be database entries belonging to other users, files in the system, and more. This is caused by the fact that the application takes user supplied input and uses it to retrieve an object without performing sufficient authorization checks.

Directory traversal

Directory traversal (also known as file path traversal) is a web security vulnerability that allows an attacker to read arbitrary files on the server that is running an application. This might include application code and data, credentials for back-end systems, and sensitive operating system files. In some cases, an attacker might be able to write to arbitrary files on the server, allowing them to modify application data or behavior, and ultimately take full control of the server.

Static application security testing

Static application security testing (**SAST**) is used to secure software by reviewing the source code of the software to identify sources of vulnerabilities. Although the process of statically analyzing the source code has existed as long as computers have existed, the technique spread to security in the late 90s and the first public discussion of SQL injection in 1998 when Web applications integrated new technologies like JavaScript and Flash.

Unlike dynamic application security testing (DAST) tools for black-box testing of application functionality, SAST tools focus on the code content of the application, white-box testing. A SAST tool scans the source code of applications and its components to identify potential security vulnerabilities in their software and architecture. Static analysis tools can detect an estimated 50% of existing security vulnerabilities.

In the software development life cycle (SDLC), SAST is performed early in the development process and at code level, and also when all pieces of code and components are put together in a consistent testing environment. SAST is also used for software quality assurance. even if the many resulting false-positive impede its adoption by developers

SAST tools are integrated into the development process to help development teams as they are primarily focusing on developing and delivering software respecting requested specifications. SAST tools, like other security tools, focus on reducing the risk of downtime of applications or that private information stored in applications will not be compromised.

For the year of 2018, the Privacy Rights Clearinghouse database shows that more than 612 million records have been compromised by hacking.

Session fixation

Description

Session Fixation is an attack that permits an attacker to hijack a valid user session. The attack explores a limitation in the way the web application manages the session ID, more specifically the vulnerable web application. When authenticating a user, it doesn't assign a new session ID, making it possible to use an existent session ID. The attack consists of obtaining a valid session ID (e.g. by connecting to the application), inducing a user to authenticate himself with that session ID, and then hijacking the user-validated session by the knowledge of the used session ID. The attacker has to provide a legitimate Web application session ID and try to make the victim's browser use it.
The session fixation attack is not a class of Session Hijacking, which steals the established session between the client and the Web Server after the user logs in. Instead, the Session Fixation attack fixes an established session on the victim's browser, so the attack starts before the user logs in.
There are several techniques to execute the attack; it depends on how the Web application deals with session tokens. Below are some of the most common techniques:
• Session token in the URL argument: The Session ID is sent to the victim in a hyperlink and the victim accesses the site through the malicious URL.

• Session token in a hidden form field: In this method, the victim must be tricked to authenticate in the target Web Server, using a login form developed for the attacker. The form could be hosted in the evil web server or directly in html formatted e-mail.
• Session ID in a cookie:
o Client-side script
Most browsers support the execution of client-side scripting. In this case, the aggressor could use attacks of code injection as the XSS (Cross-site scripting) attack to insert a malicious code in the hyperlink sent to the victim and fix a Session ID in its cookie. Using the function document.cookie, the browser which executes the command becomes capable of fixing values inside of the cookie that it will use to keep a session between the client and the Web Application.

<META>

tag

<META>

tag also is considered a code injection attack, however, different from the XSS attack where undesirable scripts can be disabled, or the execution can be denied. The attack using this method becomes much more efficient because it's impossible to disable the processing of these tags in the browsers.
o HTTP header response
This method explores the server response to fix the Session ID in the victim's browser. Including the parameter Set-Cookie in the HTTP header response, the attacker is able to insert the value of Session ID in the cookie and sends it to the victim's browser.

Interactive Application Security Testing

IAST (interactive application security testing) is an application security testing method that tests the application while the app is run by an automated test, human tester, or any activity "interacting" with the application functionality.
The core of an IAST tool is sensor modules, software libraries included in the application code. These sensor modules keep track of application behavior while the interactive tests are running. If a vulnerability is detected, an alert will be sent.
The process and feedback are done in real time in your integrated development environment (IDE), continuous integration (CI) environment, or quality assurance, or while in production. The sensors have access to:

- Entire code
- Dataflow and control flow
- System configuration data
- Web components
- Back-end connection data

Examples of such vulnerabilities could be hardcoding API keys in cleartext, not sanitizing your users inputs, or using connections without SSL encryption.