

Web Application Protocols

1.1 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 directives 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., `"` ; → `"`). 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.

1.2 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., `View Products`). 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 to 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.

1.3 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: wagh-bank.com
Content-Length: 106
servlet=TransferFunds&method=confirmTransfer&fromAccount=10372918&to
Account=3910852&amount=291.23&Submit=0k
```

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.

1.4 Lab 2

Task 1 - PHP Reading

Consider the PHP code contained in the file `task_1.php` and answer the following questions:

- (1) What does the code do? Try to understand it by grouping code lines (e.g., "lines 7-11 do X, lines 13-21 do Y").
- (2) What is the role of superglobal variables in the code?
- (3) `preg_match` is a very important function in PHP. Can you explain what the regular expression in lines 37-39 do?

As a side note, you will find some instructions and commands that are not contained in the slides. This is done on purpose to make you navigate the PHP documentation. Feel also free to test the code on a sandbox to see how it works.

Solution. This PHP snippet retrieves form data submitted via POST (username, address, comments, and a "sent" flag) and stores it in variables for further processing. It also captures the current script's path in `$self`, likely to redisplay the form on the same page, and initializes an empty error message variable (`$errmsg`) for potential validation feedback.

```
PHP
$self = $_SERVER['PHP_SELF'];
$username = $_POST['username'];
$useraddr = $_POST['useraddr'];
$comments = $_POST['comments'];
$sent = $_POST['sent'];
$errmsg = "";
```

This PHP code dynamically builds an HTML form as a string stored in the variable `$form`. The form submits to the same script (`$self`) using the POST method. It includes fields for Name, Email, and Comments, pre-filled with values from previous submissions, and it ends with a submit button named `sent` with the label "Send Form".

```
PHP
$form = "<form action=\"\$self\" method=\"post\">";
    $form.="Name:<input type=\"text\" name=\"username\"";
    $form.=" size=\"30\" value=\"\$username\" >";
    $form.="Email:<input type=\"text\" name=\"useraddr\"";
    $form.=" size=\"30\" value=\"\$useraddr\">";
    $form.="Comments:<textarea name=\"comments\" >";
    $form.="\$comments</textarea><br/>";
    $form.="<input type=\"submit\" name=\"sent\" value=\"Send Form\">";
    $form.="</form>";
```

The following PHP block performs form validation only if the form was submitted. It assumes the form has been submitted and starts by setting `$valid=true`. It checks whether the name, email, and comments fields are non-empty. If any are empty, it appends an error message to `$errmsg` and marks the input as invalid (`$valid=false`).

```
PHP
if($sent) {
    $valid=true;

    if(!$username) {$errmsg.="Enter your name...<br />"; $valid = false;}

    if(!$useraddr) {
        $errmsg.="Enter your email address...<br />"; $valid = false;
    }

    if(!$comments) {$errmsg.="Enter your comments...<br />"; $valid = false;}

    $useraddr = trim($useraddr);
    $_name = "/^[~!#$%&\'*+\\.\\"/>

```

This final block handles the outcome of the validation. If the form data is invalid, it displays any error messages (`$errmsg`) followed by the form again, pre-filled with the user's previous input, so they can correct mistakes. If the data is valid, it sends an email to `php@h.com`, with the subject "Feedback from [username]", with the message being the user's comment, and the headers are set to send an HTML-formatted email with ISO-8859-1 encoding and the user's email as the sender. If the `mail()` function succeeds, it shows a success message thanking the user.

```

PHP
if($valid != true) {
    echo($errmsg.$form);
}
else {
    $to = "php@h.com";
    $re = "Feedback from $username";
    $msg = $comments;
    $headers = "MIME-Version: 1.0\r\n";
    $headers .= "Content-type: text/html;";
    $headers .= "charset=\"iso-8859-1\"\r\n";
    $headers .= "From: $useraddr \r\n";
    if(mail($to,$re,$msg, $headers)) {
        echo("Your comments have been sent - thanks $username");
    }
}

```

Task 2 - Basic Web Hacking (Part II)

Solve NATAS levels 6-8. To do so, you must first complete levels 1-5 of Lab 1. Focus in particular on describing the actions performed by the PHP code in level 8.

Solution. These solutions require various web application penetration testing techniques to uncover hidden data and bypass access controls.

- (1) *Level 6* (`bmg8SvU1LizuWjx3y7xkNERkHxGre0GS`): The source code reveals the PHP script, which includes a file at the path `includes/secret.inc`. Navigating directly to this path reveals the secret key: `FOEIUWGHFEEUHOFUOIU`. Entering this secret into the search bar completes the challenge and provides the password.
- (2) *Level 7* (`xcoXLmzMkoIP9D7hlgPlh9XD70gLAe5Q`): The page initially shows two non-informative links. Inspection with Burp reveals a comment containing a hint about the password location: `/etc/natas_webpass/natas8`. Attempting to access this file directly via the URL fails. However, by clicking one of the provided links, the URL structure changes to `index.php?page=`, indicating a Local File Inclusion vulnerability. By setting the page parameter to the full path `index.php?page=/etc/natas_webpass/natas8`, the contents of the password file are displayed.
- (3) *Level 8* (`ZE1ck82lmdGIoEr1hQgWND6j2Wzz6b6t`): The page presents a search bar, and the key to the solution is found by inspecting the source PHP code. The code contains the following function used to encode the secret:

```

PHP
encodeSecret($secret) = bin2hex(strrev(base64_encode($secret)))

```

To find the original secret, one must simply reverse the process: first hex-decode the encoded string, then reverse the resulting string, and finally Base64-decode the result. Applying this reversal process reveals the secret: `oubWYf2kBq`, which, when entered into the search bar, unlocks the final password.

Task 3 - Request/Responses with Python

While Burp is an excellent tool for intercepting requests and responses, Python can be really useful to easily manipulate requests and responses, thus providing similar functionalities to Burp Proxy and Repeater. Find the flag on this page by using Python 3 requests:

`http://mercury.picoctf.net:27177/`.

I 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.

2.1 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

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

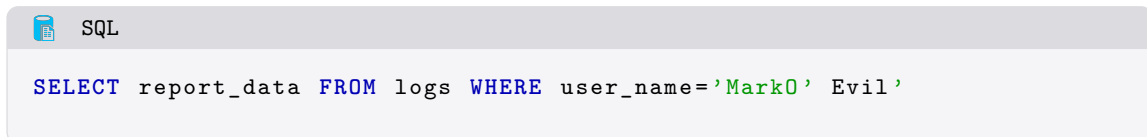
SELECT * FROM users WHERE username='Markus' AND password=' ' OR 1 = 1 --'
```

- (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 now 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 `Mark0' Evil`. The application may correctly escape the quote before storing it, resulting in `Mark0" 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 `Mark0' Evil`, a query like the following will break the SQL syntax and potentially allow the attacker to inject their code:



```
SELECT report_data FROM logs WHERE user_name='Mark0' 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.

2.2 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. If a web page uses an external parameter to construct the code passed to `eval()`, it introduces a high-risk vulnerability.

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

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
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.

2.3 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.

2.4 Lab 4

Task 2 - Command Injection

The task is composed of two parts:

- (1) Complete the Command Injection tasks of the DVWA at low, medium and high difficulty. The goal is to print the `/etc/passwd` file stored in the server.
- (2) Solve Natas levels 9 and 10.

Solution.

- (1) *Level 9* (`t7I5VHvpa14sJTUGV0cbEsbYfFP2dm0u`): This level is composed of a search bar and a php source code. By inspecting the source code we can see that there is no protection against command injection, so a simple `; cat /etc/natas_webpass/natas10` prints the password to screen.
- (2) *Level 10* (`UJdqkK1pTu6VLt9UHWAgRZz6sVUZ3lEk`): This level the same as the one before, but it now filters the symbols `|`, `;` and `&`. However, we can still use the `grep` function to look at the entire contents of the `/etc/natas_webpass/natas11` directory using `.* /etc/natas_webpass/natas11 #`. This shows a few entries, and the last one is the password we need.

Task 3 - PHP and Cookie Encryption

Consider the PHP code of Natas level 11. You are required to describe the following accurately:

- (1) The functionality of each function of the code.
- (2) How the cookie is encrypted.
- (3) Why the employed encryption is weak and how you can crack it. What kind of decryption function is required?

Finally, try to solve Natas 11 and provide the password for the next level.

Solution. The main flow of the PHP code is the following. In here, the variable `data` is loaded with the function `loadData()`. Then, the input from the user, once sanitized, is used to change the background color. When this is done, data is saved with the function `saveData()`.

```
PHP
$data = loadData($defaultdata);

if(array_key_exists("bgcolor",$_REQUEST)) {
    if (preg_match('/^#(?:[a-f\d]{6})$/i', $_REQUEST['bgcolor'])) {
        $data['bgcolor'] = $_REQUEST['bgcolor'];
    }
}

saveData($data);
```

In the next portion of code, we can take a look at the `loadData` function. The function gets access to the global `$_COOKIE` superglobal, which holds all cookies sent by the browser. If the browser sends a cookie called `"data"`, the function proceeds to decode and validate it. The cookie value is Base64-decoded, xor-encrypted and json decoded. Once this is done, the data is updated with the user inserted data and returned.

```
PHP
function loadData($def) {
    global $_COOKIE;
    $mydata = $def;
    if(array_key_exists("data", $_COOKIE)) {
        $tempdata =
            json_decode(xor_encrypt(base64_decode($_COOKIE["data"])), true);
        if(is_array($tempdata) && array_key_exists("showpassword",
            $tempdata) && array_key_exists("bgcolor", $tempdata)) {
            if (preg_match('/^#(?:[a-f\d]{6})$/i', $tempdata['bgcolor'])) {
                $mydata['showpassword'] = $tempdata['showpassword'];
                $mydata['bgcolor'] = $tempdata['bgcolor'];
            }
        }
    }
    return $mydata;
}
```

Finally, the function that does the actual encryption. This function uses a repeating XOR key to "encrypt" cookies, but it's insecure because XOR with a known or guessed plaintext easily reveals the key, allowing attackers to forge cookies.

```
PHP

function xor_encrypt($in) {
    $key = '<ensored>';
    $text = $in;
    $outText = '';

    // Iterate through each character
    for($i=0;$i<strlen($text);$i++) {
        $outText .= $text[$i] ^ $key[$i % strlen($key)];
    }

    return $outText;
}
```

Now, for the actual solution. Since we know the default data value for the cookie, which is "showpassword"=>"no", "bgcolor"=>"#ffffff", we can take a look at the current cookie value from Burp: HmYkBwozJw4WNyAAFyB1VUcqOE1JZjUIBis7ABdmbU1GIjEJAyIxTRg