

# Web & Browser Security

## Chapter Three: Cookies, Sessions, XSS

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



How does the webapp know who you are.  
And how does the attacker abuse that?

# 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**

# Act One



# Cookies

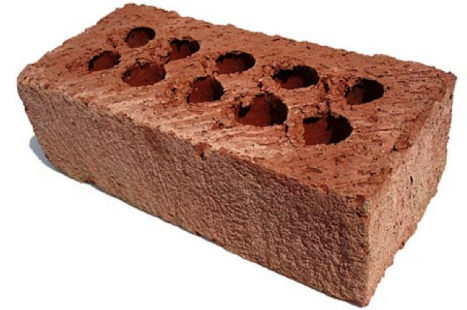
# Questions

- 1) You hear “cookies”, what comes to mind?
- 2) What do we need them for?
- 3) What are they actually used for?



# History of Sessions & Cookies

- Basically, HTTP was never meant to
  - Serve & cater to individualized Web Applications
  - Be able to contain authentication logic
- It was made to be...
  - Great for static sites, transport hypertext
  - Linking other static sites, transport hypertext
  - Want Applications? Use Applets! There, solved it :)
- HTTP is stateless, nothing but a set of headers and a body
  - No sequence numbers, no integrity checks
  - We saw already how simple all that is
- Now, how can we get HTTP to deliver a full fledged web application?
- **Well, we need to patch something on top!**



# Remember, Request... (1/2)

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

Upgrade-Insecure-Requests: 1



No authentication here, nothing to see.

# And Response (2/2)

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

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

Nothing here either...



<!doctype html>

<html lang="de" class="no-js html--rwd">

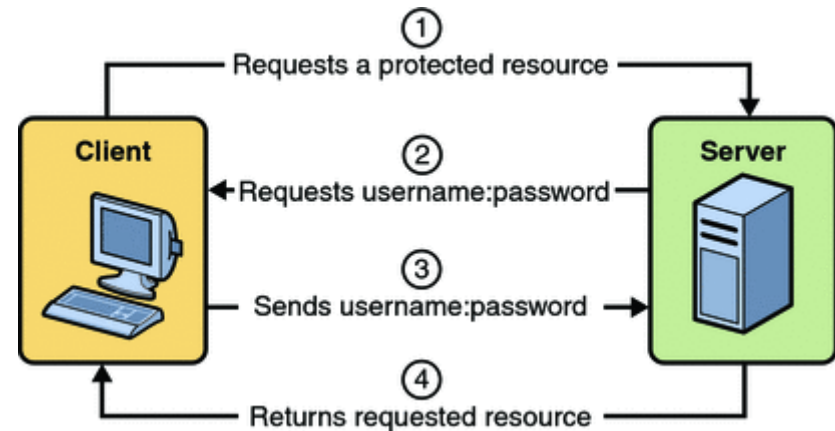
<!--<![endif]-->

<head><!-- Google Tag Manager -->

<script>(function ...

# Authentication in a Stateless World

- First we have HTTP Basic Auth
  - Base64 encoded data in HTTP header
  - Or user-name & password via URL
  - <http://foo:bar@ohnoe.com>
- Then, URL Tokens, yikes, PHPSESSID
- Then we have Cookies to handle the info
  - „Invented“ by Lou Montulli, 1994
  - Client and Server first share a secret,
  - Client stores the secret locally
  - Server uses the secret to identify a client
- **But let's wait with that for a second!**





# Request (1/4)

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

Upgrade-Insecure-Requests: 1

Name: Value

# Angry Response (2/4)

**HTTP/1.1 401 Unauthorized**

Server: nginx

Date: Sat, 26 Oct 2019 10:48:00 GMT

Content-Type: text/html

Content-Length: 188

Connection: close

WWW-Authenticate: Basic realm="Restricted Content"

<html>

<head><title>401 Authorization Required</title></head>

<body bgcolor="white">

<center><h1>401 Authorization Required</h1></center>

<hr><center>nginx</center>

</body>

</html>

# And the next Request (3/4)

GET / HTTP/1.1

Host: www.test.de

User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86\_64; rv:70.0) Gecko/20100101 Firefox/70.0

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

Upgrade-Insecure-Requests: 1

**Authorization: Basic YXVkaXRpOmRlbW8=**

# Happy Response (4/4)

**HTTP/1.1 200 OK**

Server: nginx

Date: Sat, 26 Oct 2019 10:48:00 GMT

Content-Type: text/html

Content-Length: 188

Connection: close

<html>**Yay :D**</html>

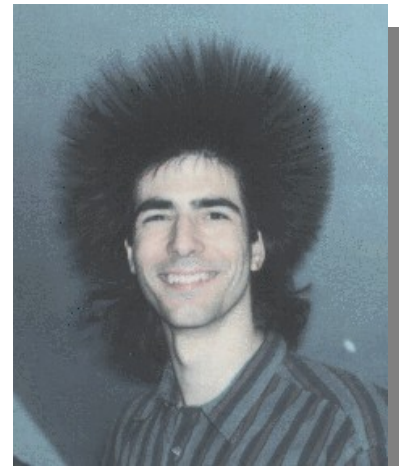
# Questions

- 1) What are existing problems with HTTP Basic Auth
- 2) Is there anything like HTTP “Advanced” Auth available instead?
- 3) And what is the problem with that?



# So, well, Cookies

- It became clear that HTTP will not take care of authentication
  - And quite frankly, why would it. Not its job.
- So it needed to be implemented some other way
- The **endpoints**, here **browser** and **server**, needed to take care of it
- And Lou Montulli came up with a proposal
  - It was labeled “Netscape HTTP Cookies”
  - Remember, the first unnoticed then demonized tool
- Let the endpoints take care my exchanging secret values
  - Just like servers did with “magic cookies”
  - Check that out here <https://is.gd/yp5OT1>
- And voila, problem solved.



# Looking at Cookies & Sessions

- It's really really simple, at least it seems so
  - Browser sends a request (i.e. username + password)
  - Server says “Oh it's you! Here's a secret”
    - Set-Cookie: SESSID=123456;
    - Or why not Set-Cookie2: SESSID=123456;
  - Browser receives and stores the secret
  - Browser sends the secret with next request
    - Cookie: SESSID=123456;
  - Server says “Oh, hi again!”
- But then again, the devil is in the detail
- **Let's have a closer look at how this works!**



# Remember, Request (1/3)

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

Upgrade-Insecure-Requests: 1

Name: Value



# And Response (2/3)

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: wt3\_eid=%3B368664393966025%7C2155671910572043192%232156199509891465343**

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 ...

# And another Request (3/3)

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%232156199509891465343

Upgrade-Insecure-Requests: 1

# Intellectual Property



- US5774670A
  - <https://is.gd/PdJH3P>
- US6374359B1
  - <https://is.gd/DhXd5Y>

# A modern Cookie

Set-Cookie:

**\_\_cfduid**=d7ecb6e558c890d6dad2bd2832ecf23731572088040;  
**expires**=Sun, 25-Oct-20 11:07:20 GMT; **path**=/;  
**domain**=.blog.codinghorror.com; **HttpOnly**

Cookie:

**\_\_cfduid**=da927454c2cf12e545769b9cae14001111572088040

or

Cookie: **prov**=0daa9c51-7287-217c-b88f-fa6954041fe4;  
**\_ga**=GA1.2.1368922457.1559295967;  
**\_\_gads**=ID=ece53c1356ba707f:T=1559295967:S=ALNI\_MZm6YD  
zP3JouygKotpNCpXzknKwfA; **\_\_qca**=P0-1610552880-  
1559295967862

# Cookies: Too Complicated?

- Now it gets complicated. Because cookies have flags and states. Several ones in fact.
  - **Domain;** that's the domain the cookie is set for. Note that it sets to `.blog.codinghorror.com` – not `blog.codinghorror.com`
  - **Path;** Is the cookie only set for a specific path on the domains it's meant for?
  - **Secure;** Is the cookie being sent over HTTP or only HTTPS?
  - **HttpOnly;** can or can not JavaScript access the cookie?
  - **Expires;** the date when it falls apart. Can be a date in the future, or even a date in the past.
  - **Max-Age;** kinda the same things as above, not supported by all browsers though

# More Cookie Dough

- New browsers understand so called SameSite cookies.
  - They work really well as CSRF protection.
  - But more on that later. Same for the new Cookie Prefixes.
- Cookies can be set from sub-domains
  - We will come to that next
- Cookies can be “stuffed”, overwritten or otherwise be influences.
  - We will come to that very soon
- Cookies can be abused to create a 414 Error
  - We will also come to that very soon

**Same Origin Policy != Cookie Policy**

# About the dot

- Remember the Set-Cookie header from earlier?
  - Set-Cookie:  
`__cfduid=d7ecb6e558c890d6dad2bd2832ecf23731572088040; expires=Sun, 25-Oct-20 11:07:20 GMT; path=/; domain=.blog.codinghorror.com; HttpOnly`
- Let's look at an important part, the domain flag
  - `domain=.blog.codinghorror.com;`
- What does that mean? Why the dot?



“The leading dot means that the cookie is valid for subdomains as well; nevertheless recent HTTP specifications (RFC 6265) changed this rule so modern browsers should not care about the leading dot. The dot may be needed by old browser implementing the deprecated RFC 2109.”

<https://tools.ietf.org/html/rfc6265#section-4.1.2.3>

# About the dot

- Wait, so back then this cookie flag...
  - `domain=.blog.codinghorror.com;`
- And this cookie flag were different?
  - `domain=blog.codinghorror.com;`
- But now they are not anymore?
- And they actually both work for subdomains?
  - `evil.blog.codinghorror.com`
  - `foo.bar.codinghorror.com`
- Correct. So, how about this case?
  - `verysecure.com`
  - `blog.verysecure.com`

“That said, you CAN limit the cookie to just the host by **not setting the cookie's domain at all** (or setting to empty string).

That, strangely, will set the cookie for just the host (example.com) and not any of its subdomains.”

<https://stackoverflow.com/questions/9618217/what-does-the-dot-prefix-in-the-cookie-domain-mean>

# So what does that mean?

- Web Security also means Cookie Security
- Developers have to know how to set cookies right
- Developers also need to know how to set up their domains correctly
  - `verysecure.com`
  - `blog.verysecure.com`
  - This is likely doomed to fail
- One of the safest ways is **not** to set the domain flag
  - Despite that being seemingly wrong
- Now let's look at more cookie related attacks

# Questions

- 1)What was the SOP again and how did it work?
- 2)What is the difference between SOP and Cookie Policy?
- 3)And what is the problem with that?



# Cookie Stealing

- That's the easiest attack, just steal it
- There is various ways to do so
  - Use Cross-Site Scripting, we will get there soon
  - Use MitM Attacks and just look at the headers
  - Try to get them to be echoed somewhere
  - Try to get them mixed into external requests
  - Social Engineer people into giving them away
- This is often also referred to as session hijacking or cookie hijacking

# Cookie Poisoning

- Primary goal is usually not to steal them
- This isn't always an attack carried out against users
- But rather a direct attack against servers
  - Rails web applications have had issues here
- So, how does the attack work?
- Note: **Can** be carried out against users too, i.e. to impersonate them on the server



# Cookie-based XSS

- Duh.
- Technically a Subclass of Cookie Poisoning.
- Just this time targeted against the user.
- Any idea what this means?

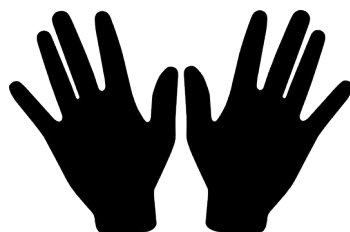


# Cookie DoS

- Most web servers have limits regarding the maximal request length
- This can be demonstrated very very easily
- Just go to any website and append many slashes to the URL
- **Let's have a look!**
- We will get a 414 error, as expected, no content, “DoS”
- Cookie Poisoning and Cookie DoS can be partners in crime
  - Look here for instance <https://is.gd/60fy7c>

# Really Useful Cookie DoS

- Now, doing what we just did is not really interesting in real life
- But how can we abuse that?
- For an **actual** attack?



# How do we prevent these?

- Cookie Stealing
  - Make sure they are set to use the HttpOnly flag
  - Make sure they are set to use the Secure flag
- Cookie Poisoning
  - Don't work with cookie headers on the server
  - If you have to, validate integrity, don't leak secrets
  - Don't echo stuff from the cookie without care
- Cookie DoS
  - Phew, maybe ask the LB to detect overlong cookies and purge them?
  - Maybe avoid having GET or POST go into COOKIE?
  - It sort of depends... really.
- Usually website owners forget **at least** one of those

# Behold. The Golden Cookie!

- **name=value; Path=/; HTTPOnly; Secure;**
  - <https://is.gd/GLEGGP>
- **name=value; Path=/; HTTPOnly; Secure;**  
**SameSite=strict**
  - <https://is.gd/lliARu>
- **\_\_Host-name=value; Path=/; HTTPOnly; Secure;**  
**SameSite=strict**
  - <https://is.gd/3Gsjpg>

# Cookie Security Checklist

- **Don't**

- Put anything from GET or POST into cookies
- If you really have to, check length before doing it
- Allow users to change cookie data that is used by the server
- Echo them anywhere with some server side script
- Echo them anywhere in the DOM to avoid persistent XSS

- **Do**

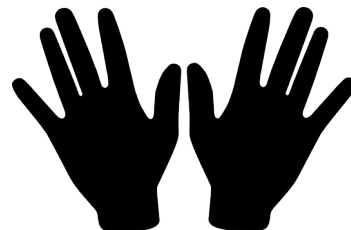
- Use all available security flags that make sense
- Make sure the server realizes cookies are untrusted
- Make sure to scope them correctly
- Less is more. Legitimate websites rarely need more than one.

# Beyond Cookies: HTTP State Tokens

- Mike West is, again, working on deprecating cookies
- Like, for real this time. And appears to be about time.
- The project is called **HTTP State Tokens**
  - <https://github.com/mikewest/http-state-tokens>
- They avoid weird SOP “Bypasses” & Artifacts
- They give users more control
- But are harder to implement
- Also, certainly not everyone likes them

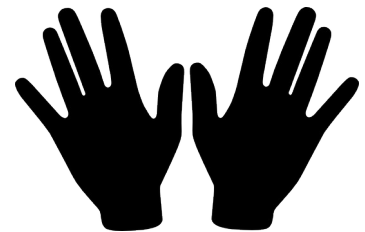
# Questions

- 1) How can we summarize the state of HTTP Cookies in present times?
- 2) What are possible alternatives to HTTP Cookies (we can already use)?
- 3) Why did none of the alternatives take off?



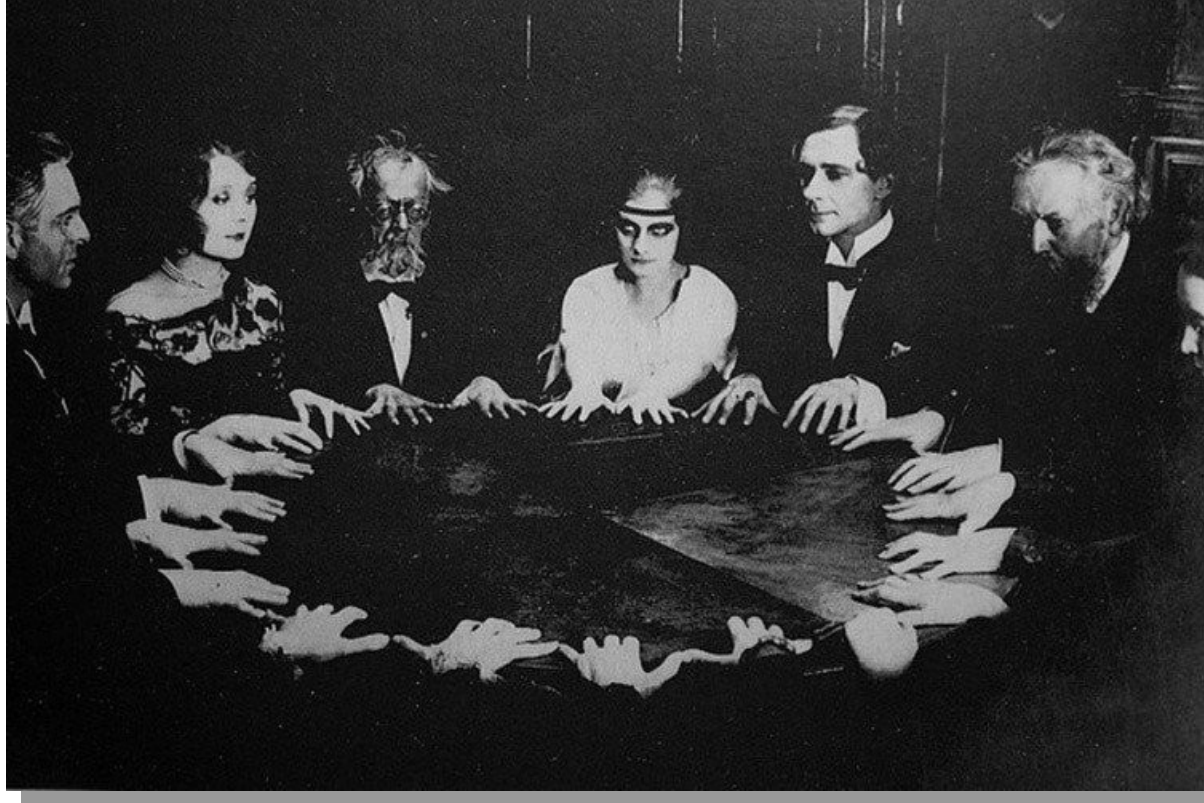
# Hands-On Time!

- **Cookies & XSS**, let's try to use cookies to get XSS and use XSS to get cookie content.
  - <https://is.gd/Jrzwl8>
  - <https://is.gd/lt7ME7>
  - 15 minutes time for each exercise





# Act Two



# Sessions

# What are Sessions

“a session is a temporary and interactive information interchange between two or more communicating devices, or between a computer and user (see login session). A session is established at a certain point in time, and then ‘torn down’ - brought to an end - at some later point”

<https://is.gd/xJTDs7>

# So, in other Words

- A session stores user-related information on usually the server
- This can contain anything, from a few bytes to megabytes of data
- Usually they also come with a session ID
- And often that session ID is also stored in the Cookies
- So browser and server can work together to recognize the user

# Individualized Storage

- They have to be stored somewhere
  - Maybe in the /tmp folder (still happens quite a lot)
  - Maybe in a database (great when you have SQL Injection)
  - Maybe in a safer location
  - Maybe on a different system in the backend
- They need to expire at some point
- They should ideally be encrypted
- They can contain personal and sensitive info

# Attacks against Sessions

- Session Hijacking
  - Wait, this is the same cookie hijacking, no?
  - Yep, mostly that is the case – just a clear focus on hijacking a user’s session
  - The attacker wants to learn the users session ID and make it her own. Done.
- Session Fixation
  - More interesting and more complex.
  - The attacker wants to make their session be used by the victim
  - Then by knowing it, be able to “hijack” it an impersonate
- Less common attacks
  - Session information stored in insecure locations
  - Bad randomness used for session IDs, low entropy, guessable IDs
  - Bad operators comparing IDs and cookie values, i.e. PHP’s “==”

# Session Fixation

- Attacker doesn't want to steal a session ID but still impersonate
- So, the attacker has to provide a valid Session ID and make the victim accept it
  - For instance, by sending a URL via mail, ICQ or alike that contains the attacker's session ID
  - Think [https://url/?SESSID=known\\_to\\_attacker](https://url/?SESSID=known_to_attacker)
- If the victim clicks the link and then logs in, and if the server does not create a new session ID...
- Then the attacker knows the session ID and impersonates the victim. Easy.

# Session Security Checklist

- **Don't**

- Put anything from GET or POST into the session ID
- Use shitty comparison operators like “==”
- Use low entropy or guessable parts in the session ID
- Store it in insecure locations like for instance /tmp

- **Do**

- Store it where only the web application itself can reach it
- Let the frameworks handle the session logic
- Invalidate sessions after logout
- Regenerate sessions after login

# Questions

- 1) How can we attack a session?
- 2) How do we best protect a session?
- 3) Is there any alternatives to sessions as we know them?





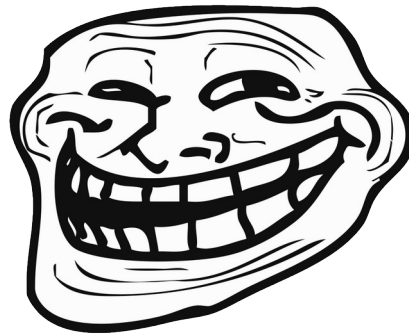
# Act Four

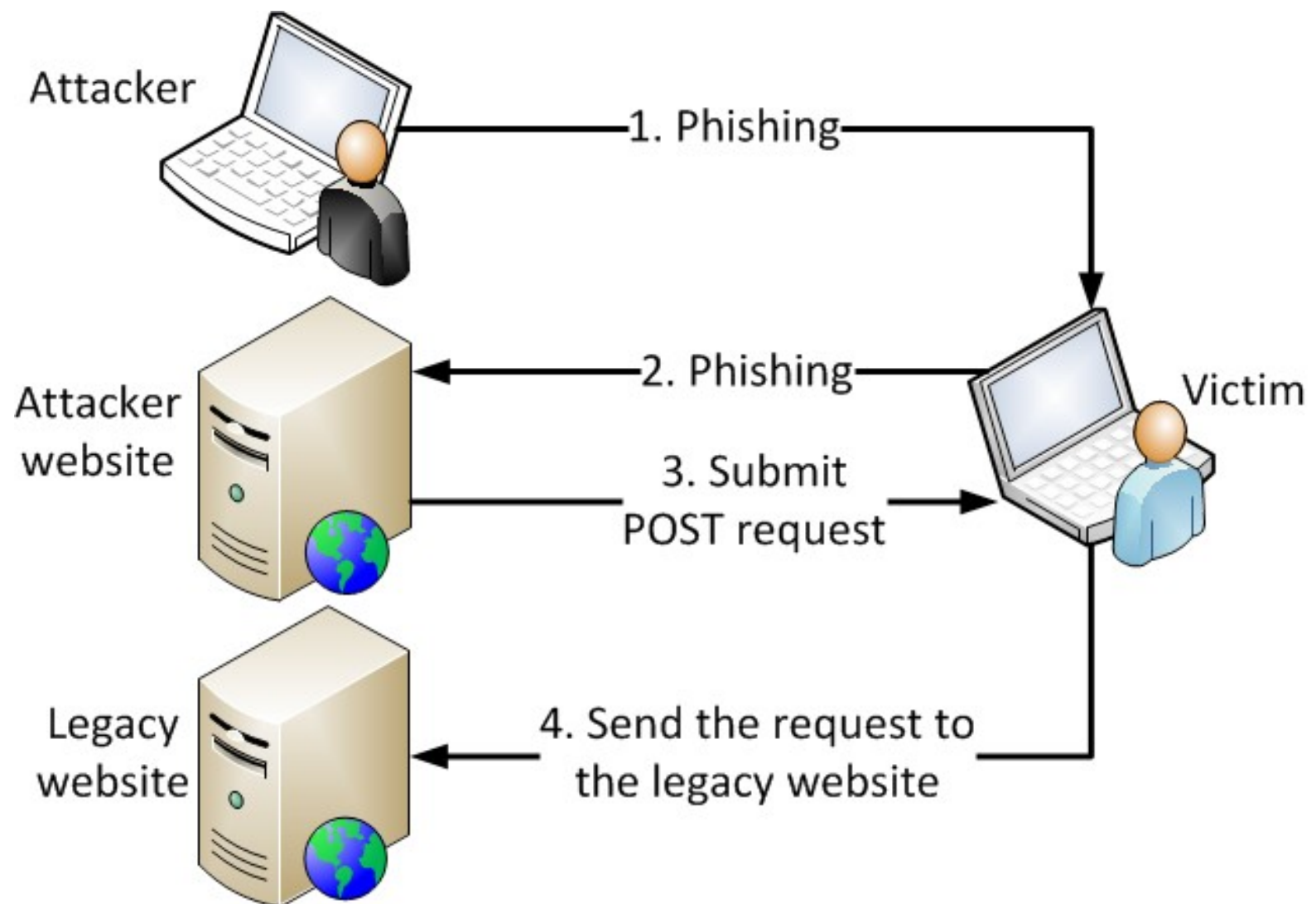


# CSRF

# Let's first discuss CSRF in the wild

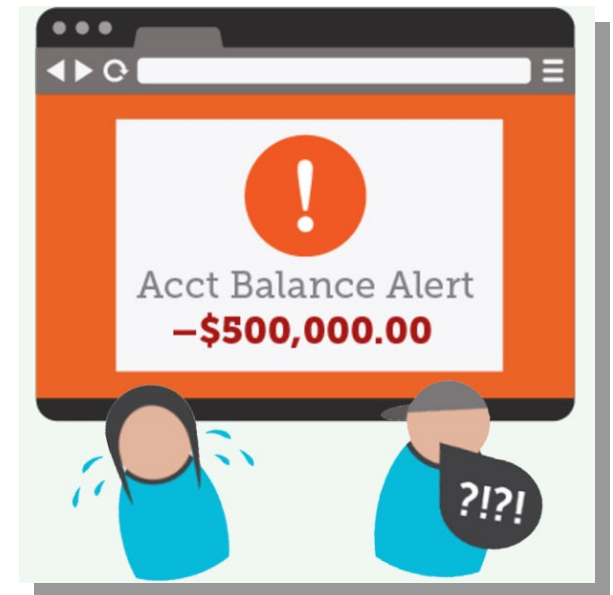
- Imagine a common online Forum, like phpBB for example. Or anything comparable.
- You can post texts and links and images.
- And you want to perform a prank against the Admin.
- **What do you do?**





# The Browser's need to send Cookies

- So, basically the CSRF problem is based on the lack of state in HTTP.
- Or the problem, that websites are doing much more than the protocol could foresee.
- And now, almost 20 years too late, we see browser-assisted mitigations
  - Remember SameSite cookies? Anyway..
- But, bottom line: no problem for requests that “read” data (SOP protects from reading)
- But huge problem for requests that write data! Like non-idempotent GET, POST or others.



# CSRF Case Study for a Pinboard

- Imagine an online pinboard application
- Users can pin notes on there.
- For shopping lists, reminders, etc.
- And attackers can do so as well!
- With terrifying consequences.



# What the attacker can do

- The attacker can
  - Send POST requests across domains, cookies will be attached
  - Send uploads across domains (POST + multipart/form-data)
  - Send an AJAX request across domain. Can send, but not read (and we don't wanna read)
  - Use `navigator.sendBeacon()` to perform uploads across domains with “virtual” files (i.e. XSS payload in filename, works so often)
  - Use brute-force to enumerate object IDs or bad tokens and nonces
  - Analyze the cryptographic quality of the token off-line, see if patterns appear or the token might be guessable
  - Check if the software verifies the token in unsafe manners, likely by using “==” instead of “===” or alike
- **And all just to do something on behalf of the logged in user.**

# What CSRF can do

- **It really depends**
- **CSRF can do what the application can do**
  - Delete a user?
  - Finalize a sale?
  - Reset the server?
  - Get sometime into jail?
- **It really depends**
  - CSRF is easy to carry out
  - Hard to fix holistically
  - Can have highly critical impact
  - And it basically a HTTP/Browser design bug

# Mitigation against CSRF

- **Don't**

- Bind “non-idempotent” actions to GET requests. It's against the RFC
- Do with GET or POST what you should do with PUT, DELETE. They are safer across domains (see CORS spec)
- Allow POST request bodies to be guessable, use a token!
- Rely on double-submit cookie patterns when you have possible sub-domain XSS

- **Do**

- Create a smart token that is bound to the user, maybe to the action and maybe to the time (so it can expire)
- Use “cryptography” wisely to create your tokens
  - # = separator character
  - nonce = random value
  - secret = sha256(userid + sessionid)
  - CSRF-Token = base64(nonce##hmac-sha256(nonce, secret))



# Request Security Checklist

- **Don't**

- Do with GET what needs a POST. Semantics matter!
- Do anything state-changing without protection.
- Trust that checking referrer is enough. It's not.
- Trust that checking other header fields is okay, it's not.

- **Do**

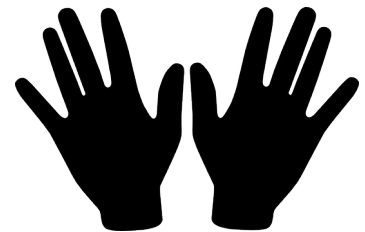
- Protect every non-idempotent action with a token
- Generate the tokens cleverly
- Avoid them leaking via GET, XS Search or alike

# But wait, SameSite by default..

- Hold on, isn't CSRF pretty much dead by now?
  - Chrome and Firefox now set SameSite lax by default
  - At least all cookies that have no SameSite flag
  - This means, we are done here, no? No more CSRF just so?
  - Correct! But. Let's think about cookie policy vs. SOP
- Can we still bypass that?
- And, if so, under **which condition?**

# Hands-On Time!

- **CSRF Examples**, let's try to see how CSRF looks in action.
  - <https://is.gd/bkjHB6>
  - <https://is.gd/zjYqam>
  - <https://is.gd/o6DmQJ>
  - 10 minutes time for each exercise



# Act Five



# XSS

# Attack with a very wrong Name

- Cross-Site Scripting (XSS) – we all love it (<http://is.gd/ufoaOm>)
- Technically, XSS is the wrong name. Still coming from the “Age of Frames”
- One website/domain „scripts“ another one
  - Kinda wrong, right? “Script injection” might fit better
  - Maybe this explains why it's still not defeated?
- What are the attacker's goals:
  - Performing DOM manipulations and (same-site) Phishing
  - Accomplishing a CSRF-token leak and taking control
  - Reading plain-text passwords, abusing form managers
  - Steal CPU/GPU time by running miners
  - Steal data, manipulate accounts, profit
  - Persistence via `localStorage` or Service Workers



# How to even XSS?

- What are levers to trigger XSS?
  - HTML injections, attribute and event-handlers
  - Script injections, DOMXSS, Flash XSS
  - Or even Plugin XSS and Cross Application Scripting

# Indeed easy as Pie



- Probably one of the “easiest to get” attack types
- Trained chicken can find XSS.
- How to do it? Somehow inject active HTML into a website
  - Breaking attributes, injecting new ones
  - Injecting entire new HTML elements
  - Breaking JavaScript strings and alike
  - Injecting styles and CSS
- Very easy to test for too:
  - `"' "><s>000`
  - `</script>'"/]});alert(1)//`
  - `<plaintext>`
  - Or why not a polyglot <https://is.gd/p1qWwo>



# Remember, Request (1/3)

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

Upgrade-Insecure-Requests: 1

Name: Value



# Benign Request

GET /?user=Susan 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

Upgrade-Insecure-Requests: 1

Name: Value

# Benign 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

Date: Mon, 07 Oct 2019 15:31:25 GMT

Connection: close

Content-Length: 68377

<!doctype html>

<html lang="de" class="no-js html--rwd">

<head></head>

<body>Hello, **Susan**!</body>

</html>

# “Evil” Request

GET **?user=<script>alert(1)</script>** 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

Upgrade-Insecure-Requests: 1

Name: Value

# XSS'd 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

Date: Mon, 07 Oct 2019 15:31:25 GMT

Connection: close

Content-Length: 68377

<!doctype html>

<html lang="de" class="no-js html--rwd">

<head></head>

<body>Hello, **<script>alert(1)</script>**!</body>

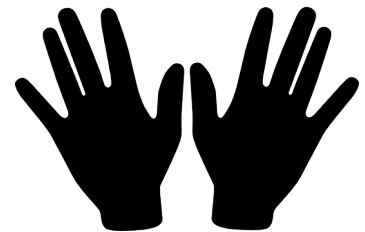
</html>

# How to even XSS?

- What are levers to trigger XSS?
  - **HTML injections**, Attributes and Event-Handlers
  - Script injections, DOMXSS, even Flash XSS
  - Or even Plugin XSS and Cross Application Scripting

# Hands-On Time!

- Isn't XSS the best? Let's inject some HTML, JS and the likes and see where we end up
  - <https://is.gd/7koyEs>
  - <https://is.gd/dA219x>
  - <https://is.gd/h47NyL>
  - 10 minutes time for each exercise



# Incomplete Anti-XSS Checklist

- **Don't**

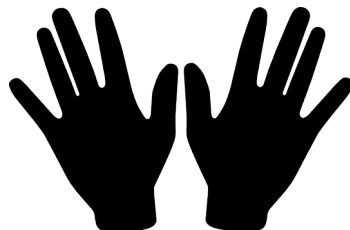
- Echo stuff from GET, POST or alike without encoding
- Store stuff without encoding or filtering
- Mess up with different **contexts**

- **Do**

- Encode if you echo stuff inside HTML
- Escape things if you echo stuff inside script
- Make all user controlled data treated securely
- Lean what “securely” even means

# Questions

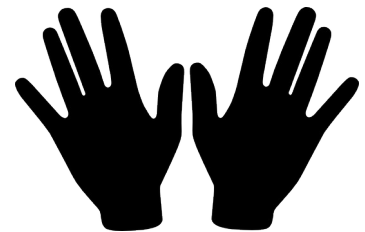
- 1) What should XSS actually be called?
- 2) Isn't encoding anything to entities just enough?
- 3) How about solving XSS with regular expressions?





# Hands-On Time!

- **More XSS?** Let's now make it a bit harder and see where we end up
  - <https://is.gd/JAnDBS>
  - <https://is.gd/f9QUUp0>
  - <https://is.gd/OoX8m9>
  - 10 minutes time for each exercise



# Chapter Three: Done

- Thanks a lot!
- Soon, more.
- Any questions? Ping me.
  - [mario@cure53.de](mailto:mario@cure53.de)