Madrid
18-19.10.2013
www.codemotionworld.com

R. Luque & J. San Leandro

Grails vs XSS

Defending Grails against XSS attacks

@rafael_luque - Osoco ***@rydnr - Ventura24***

http://goo.gl/uVadCh

Plugins You can find out about all the publicly available Grails plugins.

<iframe width="9C

<iframe width="9C

FILTERS CLEAR FILTER

- All Plugins
- Featured
- Top Installed
- Popular
- Recently Updated
- Newest
- Supported by SpringSource
- Pending Plugins

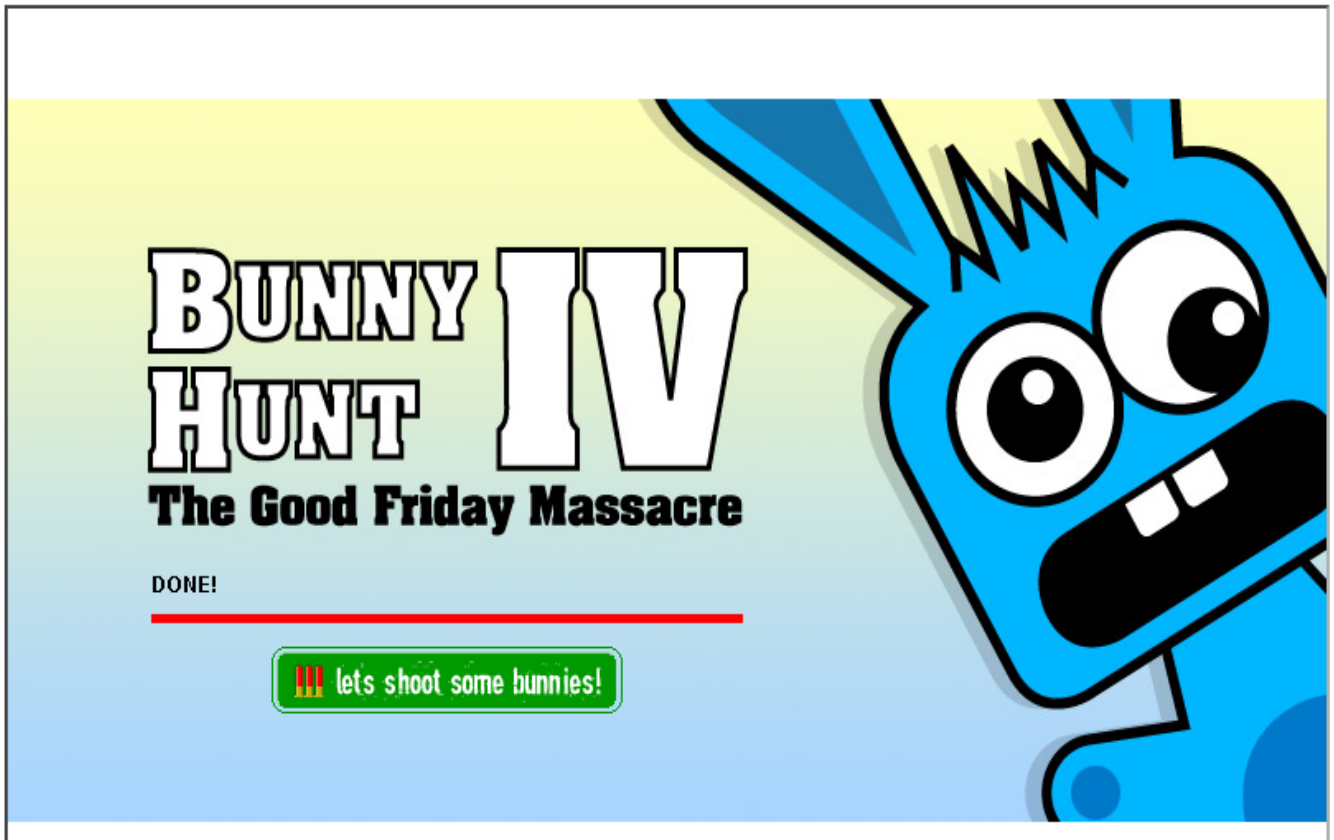
POPULAR TAGS

- Javascript
- Ajax
- Security
- Persistence
- Database

Nothing matched your query -

Oh no, Grails.org was PWN3D!!

Don't panic! This is a simple reflected XSS used as a proof of concept in our talk about Grails and XSS prevention.



XSS Intro

XSS concepts

R. Luque & J. San Leandro

- What's a XSS
- XSS Types: Reflected, stored, DOM-based.
- Famous XSS attacks: Samy worm, MrBean defacement, ...

XSS threats

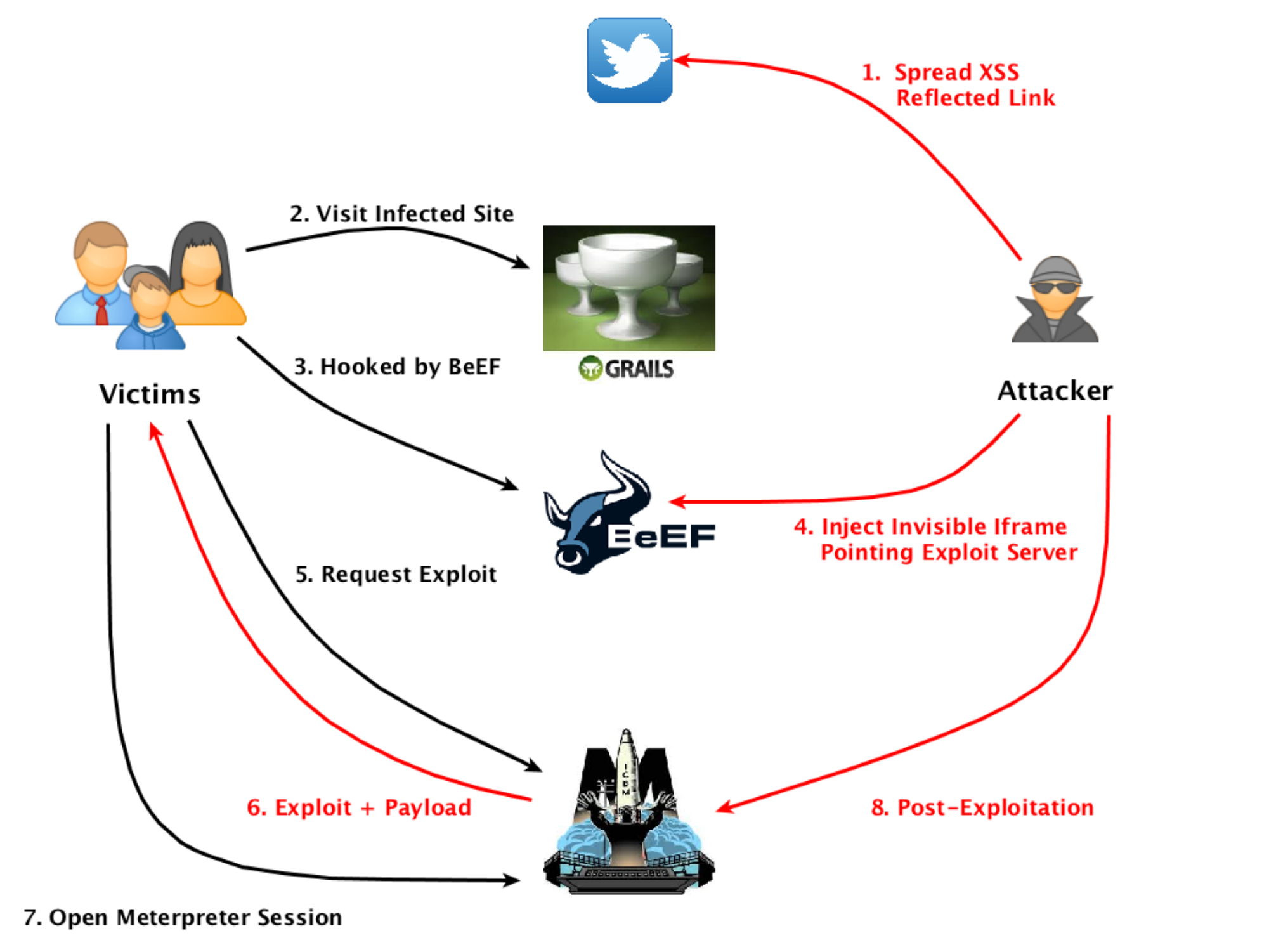
R. Luque & J. San Leandro

- Interface defacement
- Session hijacking
- Click hijacking
- Malware infection
- Your PC may be joined to the horde of zombies in a BotNet.

Following the white rabbit...

he white rab

Something more than a joke...


1. Spread XSS
Reflected Link
2. Visit Infected Site
3. Hooked by BeEF
Victims
GRAILS
Attacker
BeEF
4. Inject Invisible Iframe
Pointing Exploit Server
5. Request Exploit
6. Exploit + Payload
8. Post-Exploitation
7. Open Meterpreter Session

Responsibilities: Why is this still an issue?

Commercial software

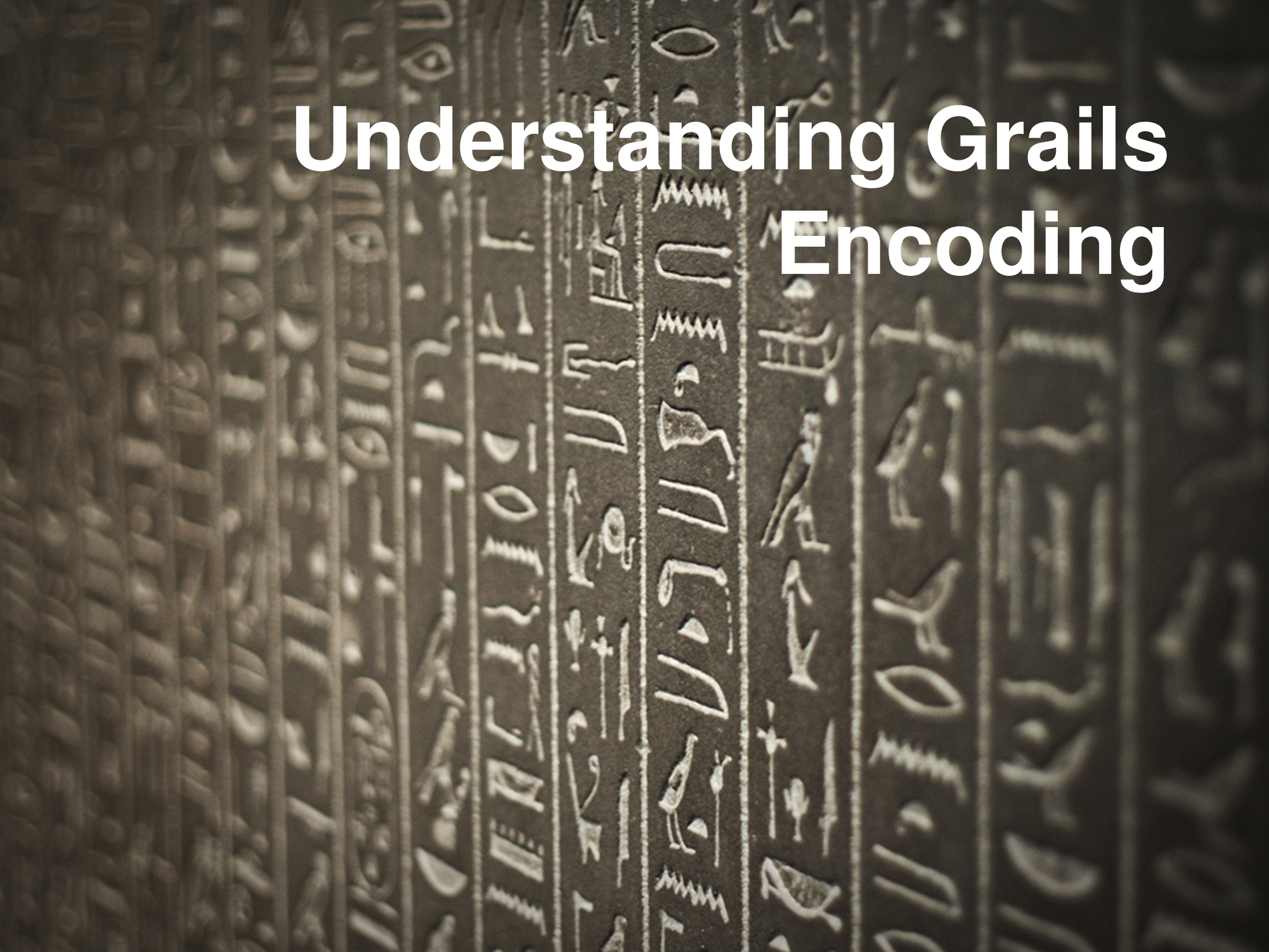
R. Luque & J. San Leandro

- XSS is not known for business stakeholders
- For most people, security means attacking your servers
- Developers don’t pay enough attention

Do your homework

R. Luque & J. San Leandro

- Raise awareness
- Practice with security tools
- Promote defensive coding
- Improve monitoring



Understanding Grails Encoding

Grails Pre-2.3 Gotchas

#1: Built-in default codec

#1: Built-in default codec

```
grails.views.default.codec
```

#1: Built-in default codec *is none!*

```
grails.views.default.codec = ''none''
```

#1: Built-in default codec *is none!*

Problems

You have to escape explicitly every untrusted data:

```
encodeAsHTML()
encodeAsJavaScript()
encodeAsURL()
```

#1: Built-in default codec *is none!*

Problems

High likelihood of XSS vulnerabilities in production.

E.g. Grails.org website.

#1: Built-in default codec *is none!*

Problems

Double-encoding prevention over *Security by default*.

#1: Built-in default codec *is none!*

Solution

Change default codec to HTML:

```
grails.views.default.codec = ''html''
```

#2: Inconsistent behaviour

Apply codec

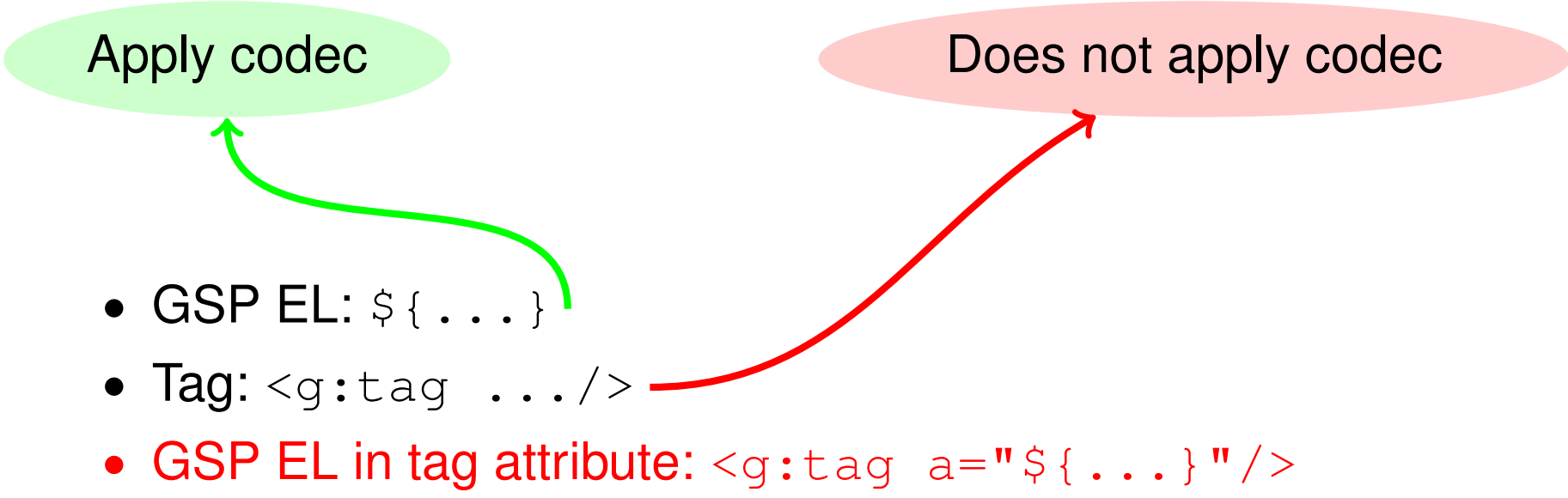
Does not apply codec

- GSP EL: `${...}`

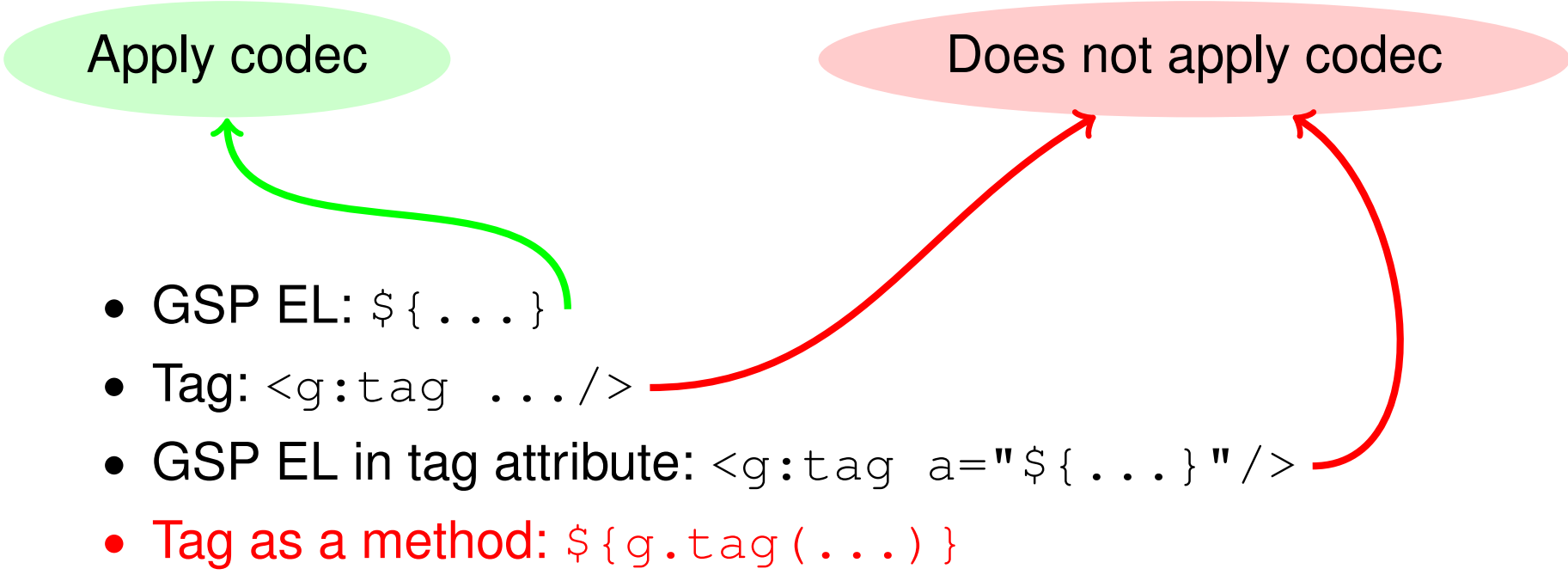
#2: Inconsistent behaviour

- GSP EL: `${...}`
- Tag: `<g:tag .../>`

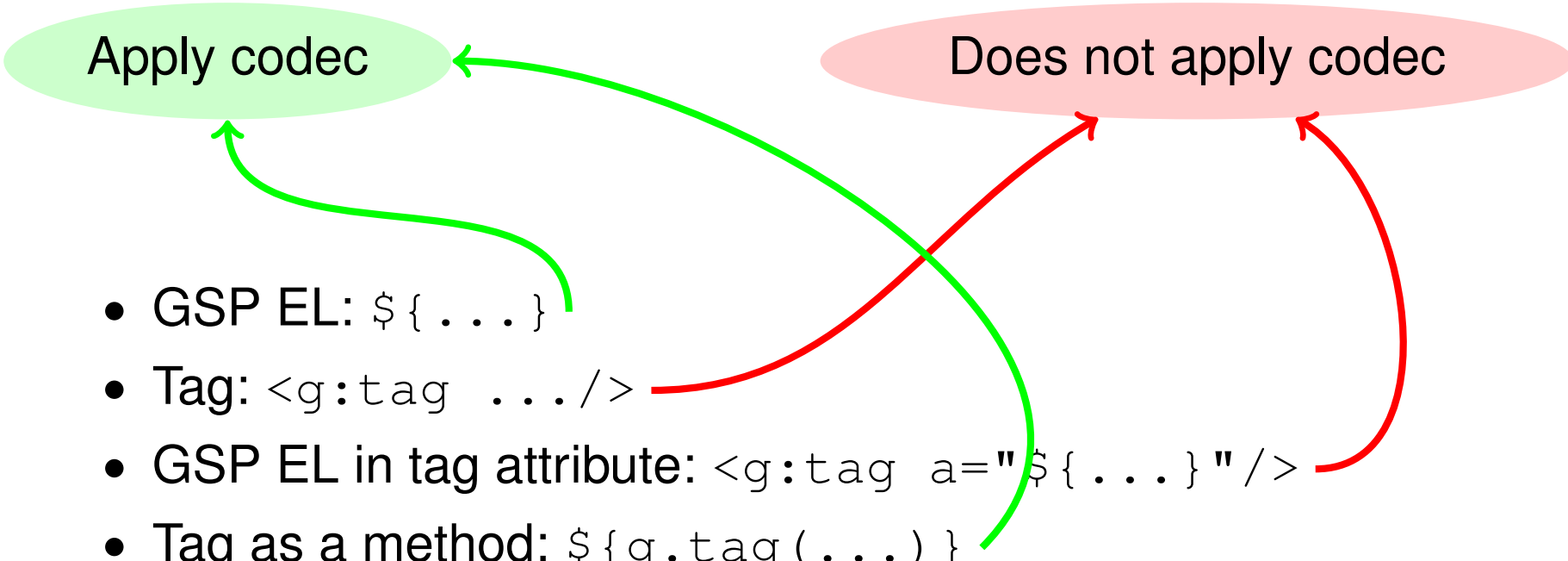
#2: Inconsistent behaviour



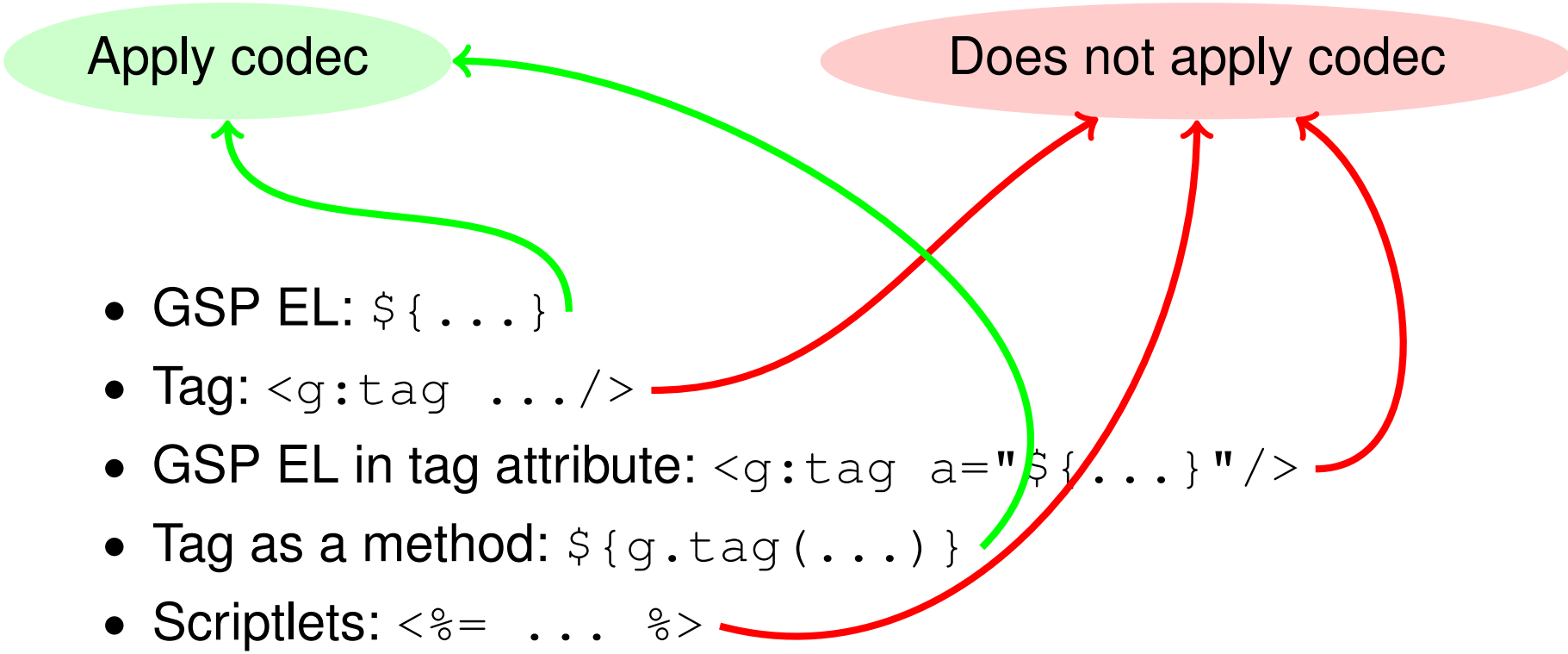
#2: Inconsistent behaviour



#2: Inconsistent behaviour



#2: Inconsistent behaviour



#3: Tag output is not escaped

Problems

Review the tags you use to make sure they encode their output or have options for this (e.g. `encodeAs` attribute).

#3: Tag output is not escaped

Problems

Review the tags from plugins you use.

#3: Tag output is not escaped

Problems

Review the tags you invoke as methods in Controllers.

#3: Tag output is not escaped

Problems

Don’t trust Grails core tags, they have inconsistent behaviour. E.g:

```
<g:fieldValue /> // HTML-encoded
<g:message />    // NO HTML-encoded
```

#3: Tag output is not escaped

Solutions

If tag implementation doesn't encode, add it explicitly or invoke it as a method inside a GSP expression:

```
<g:message ... encodeAs=''HTML''/>
${g.message(...)}
g.message(...).encodeAsHTML()
```

#4: g:message doesn't escape arguments

Problems

With default codec set to HTML the following XSS attack vector works:

```
<g:message code='welcome' args='[params.user]'/>

where:
   welcome = Hi {0}!
   params.user = <script>alert('pwnd')</script>
```

#4: g:message doesn't escape arguments

Solutions

Upgrade to a Grails version with the issue (GRAILS-7170) fixed:

2.0.5, 2.1.5, 2.2.2, 2.3-M1

#4: g:message doesn't escape arguments

Solutions

Escape explicitly or invoke the tag inside a GSP expression:

```
<g:message code='welcome' args='[params.user]'
    encodeAs='HTML'/>

${g.message(code:'welcome', args:[params.user])}
```

#5: One codec is not enough

You MUST use the escape syntax for the context of the HTML document you're putting untrusted data into:

- HTML
- JavaScript
- URL
- CSS

#5: One codec is not enough

HTML entity encoding doesn't work if you're using untrusted data inside a <script>, or an event handler attribute like onmouseover, or inside CSS, or in a URL.

#5: One codec is not enough

Problems

You can override the default codec for a page, but not to switch the codec for each context:

```
<%@page defaultCodec='CODEC' %>
```

#5: One codec is not enough

Problems

How to manage GSPs with mixed encoding requirements?

#5: One codec is not enough

Solutions

Turn off default codec for that page and use `encodeAsJavaScript()` and `encodeAsHTML()` explicitly everywhere.

#5: One codec is not enough

Solutions

Extract the JavaScript fragment to a GSP tag encoding as JavaScript.

Grails 2.3 Encoding Enhancements

#1: New configuration more *secure by default*

#1: New configuration more security by default

```
grails {
  views {
    gsp {
      encoding = 'UTF-8'
      htmlcodec = 'xml' // use xml escaping instead of HTML4
      codecs {
        expression = 'html' // escapes values inside ${}
        scriptlet = 'html' // escapes output from scriptlets in GSPs
        taglib = 'none' // escapes output from taglibs
        staticparts = 'none' // escapes output from static templates
      }
    }
    // escapes all not-encoded output at final stage of outputting
    filteringCodecForContentType {
      //'text/html' = 'html'
    }
  }
}
```

#2: Finer-grained control of codecs

Control the codecs used per plugin:

```
pluginName.grails.views.gsp.codecs.expression = 'CODEC'
```

#2: Finer-grained control of codecs

Control the codecs used per page:

```
<%@ expressionCodec='CODEC' %>
```

#2: Finer-grained control of codecs

Control the default codec used by a tag library:

```
static defaultEncodeAs = 'HTML'
```

Or on a per tag basis:

```
static encodeAsForTags = [tagName: 'HTML']
```

#2: Finer-grained control of codecs

Add support for an optional `encodeAs` attribute to all tags automatically:

```
<my:tag arg='foo.bar' encodeAs='JavaScript'/>
```

#3: Context-sensitive encoding switching

Tag `withCodec('CODEC', Closure)` to switch the current default codec, pushing and popping a default codec stack.

```
out.println '<script type=''text/javascript''>'
withCodec(``JavaScript'') {
    out << body()
}
out.println()
out.println '</script>'
```

#3: Context-sensitive encoding switching

Core tags like `<g:javascript/>` and `<r:script/>` automatically set an appropriate codec.

#4: Raw output

When you do not wish to encode a value, you can use the `raw()` method.

```
${raw(book.title)}
```

It's available in GSPs, controllers and tag libraries.

#5: Default encoding for all output

You can configure Grails to encode all output at the end of a response.

#5: Default encoding for all output

```
grails {
  views {
    gsp {
      codecs {
        expression = 'html' // escapes values inside ${}
        scriptlet = 'html' // escapes output from scriptlets in GSPs
        taglib = 'none' // escapes output from taglibs
        staticparts = 'raw' // escapes output from static templates
      }
    }
    // escapes all not-encoded output at final stage of outputting
    filteringCodecForContentType {
      'text/html' = 'html'
    }
  }
}
```

Don’t Trust Plugins

Plugins are part of your app

R. Luque & J. San Leandro

- Grails plugins are not security audited
- Grails plugins are part of your application's attack surface
- Review plugins to make sure they encode, and if they don't you should JIRA the authors immediately, and fork and patch to fix your app quickly.

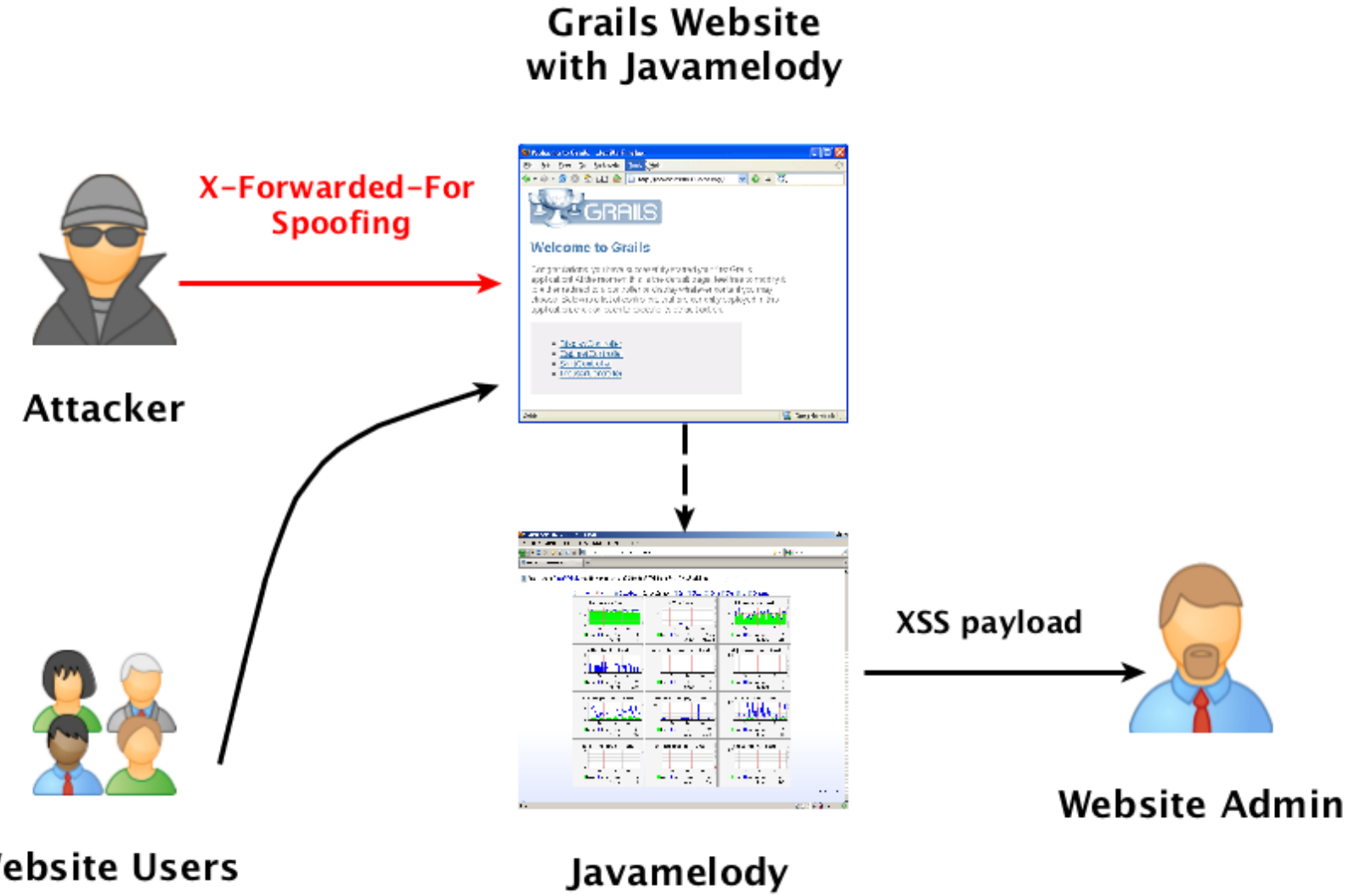
E.g. Javamelody vulnerability

R. Luque & J. San Leandro

- CVE-2013-4378 vulnerability reported.
- Allows **blind XSS** attack via `X-Forwarded-For` header spoofing.
- The attack target is the admin's browser.
- Fixed in the last release (1.47).
- You should upgrade ASAP.

Demo: Javamelody XSSed

R. Luque & J. San Leandro



Solutions: What options do we have?

Think like an attacker

R. Luque & J. San Leandro

- According to your grails version
- Find unescaped values
- Use fuzzers
- Read and understand Samy code
- Review OWASP XSS cheatsheets

Be aware

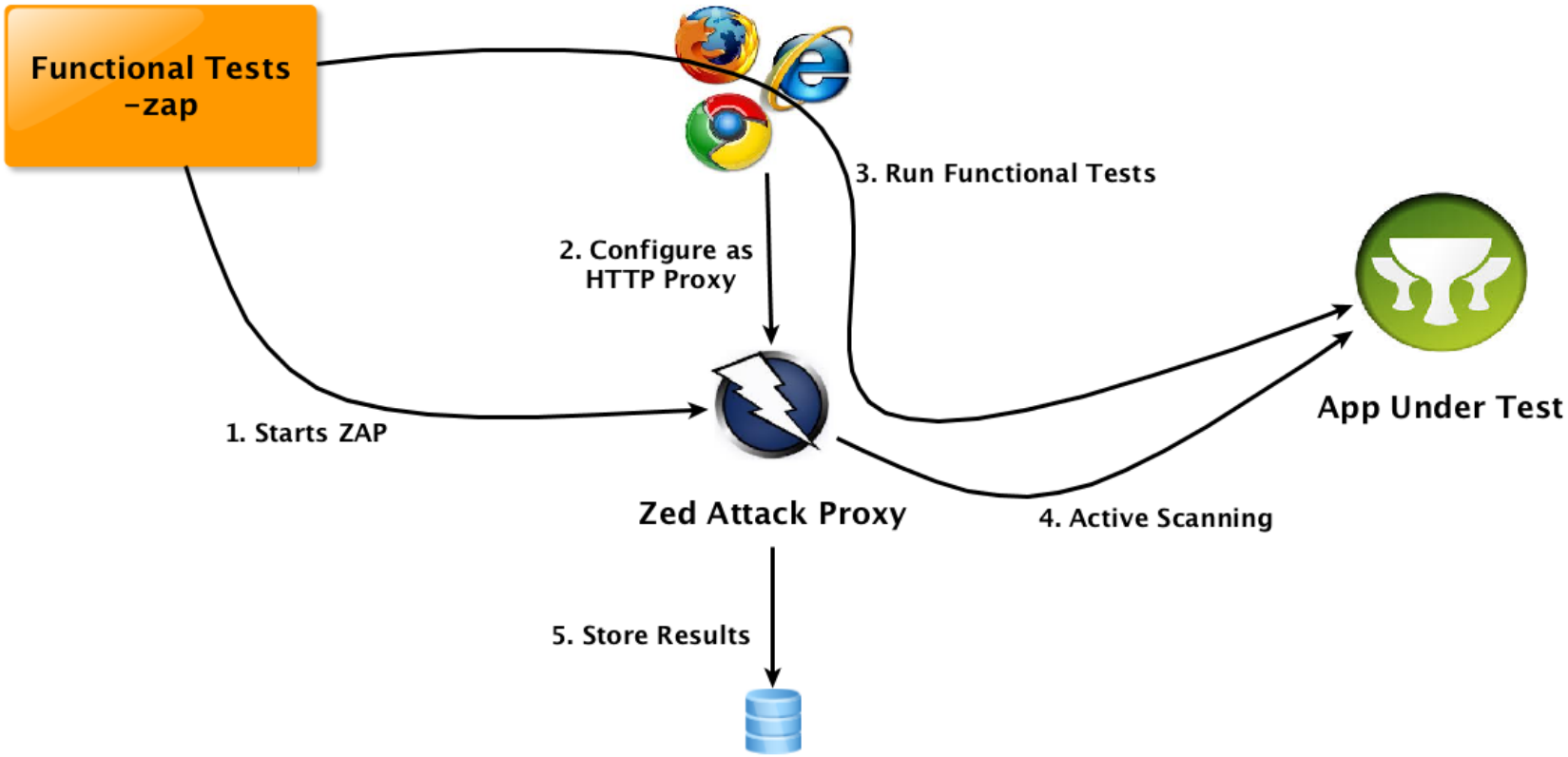
R. Luque & J. San Leandro

- Review your Grails app to double-check how all dynamic content gets escaped
- Monitor for suspicious traffic
- Spread the knowledge
- Review available security plugins for Grails

Automated security tests

R. Luque & J. San Leandro

- **ZAP Security Tests plugin** for Grails.



Application firewalls

R. Luque & J. San Leandro

- Enable common, safe rules

Application firewalls

R. Luque & J. San Leandro

- Enable common, safe rules
- Log unexpected traffic

Application firewalls

R. Luque & J. San Leandro

- Enable common, safe rules
- Log unexpected traffic
- Don’t fool yourself

- CSP: Content Security Policy
- Adds headers to disable default behavior
  - inline Javascript
  - dynamic code evaluation
- Still a Candidate Recommendation of W3C

Conclusions: Grails can defeat XSS

Grails

R. Luque & J. San Leandro

- Is able to defend our application from XSS attacks

Grails

R. Luque & J. San Leandro

- Is able to defend our application from XSS attacks
- But we need to pay attention to the details

Grails

R. Luque & J. San Leandro

- Is able to defend our application from XSS attacks
- But we need to pay attention to the details
- Upgrade to 2.3 ASAP

Grails

R. Luque & J. San Leandro

- Is able to defend our application from XSS attacks
- But we need to pay attention to the details
- Upgrade to 2.3 ASAP
- Pay attention to XSS

XSS

R. Luque & J. San Leandro

- Is much more dangerous than defacement jokes

XSS

R. Luque & J. San Leandro

- Is much more dangerous than defacement jokes
- The browsers are the actual target

XSS

R. Luque & J. San Leandro

- Is much more dangerous than defacement jokes
- The browsers are the actual target
- Difficult to monitor

XSS

R. Luque & J. San Leandro

- Is much more dangerous than defacement jokes
- The browsers are the actual target
- Difficult to monitor
- Unconfortable counter-measures in the browser: NoScript, Request Policy

Wake up

R. Luque & J. San Leandro

- Write secure applications by default

Wake up

R. Luque & J. San Leandro

- Write secure applications by default
- Get yourself used with Metasploit, Burp, ZAP

Wake up

R. Luque & J. San Leandro

- Write secure applications by default
- Get yourself used with Metasploit, Burp, ZAP
- Spread the word both horizontally and vertically

Picture credits

R. Luque & J. San Leandro

- **Cover**:
  `http://www.flickr.com/photos/usairforce/`
  CC by-nc
- **White rabbit**:
  `http://www.flickr.com/photos/alles-banane/5849593440`
  CC by-sa-nc
- **Hieroglyphs**:
  `http://www.flickr.com/photos/59372146@N00`
  CC by-sa-nc
- **Zombies**:
  `http://www.flickr.com/photos/aeviin/4986897433`
  CC by-sa-nc

Madrid
18-19.10.2013
www.codemotionworld.com

R. Luque & J. San Leandro

Grails vs XSS

Defending Grails against XSS attacks

@rafael_luque - Osoco ***@rydnr - Ventura24***