Using Instrumentation to Optimize Application Security

The road to Self-Protecting Software

Girish Nair, CISSP, CSSLP
Solutions Architect
Current state of Application Security

PAIN POINTS

1. Delays
2. Inconvenience
3. Unfriendly
4. ...
An early application exploit

• In 1988, Morris Worm exploited a buffer overflow in Unix to spread from machine to machine.
• 3 decades later... buffer overflows still exist.
• Is the developer responsible for the security breach?

Don’t blame the developer!
Possible ways to prevent it?

• One could argue that the technology failed him.
• Why is C/C++ susceptible to buffer overflow attacks?
• Java & .NET developers don’t create this problem. Are they better developers?

The JVM and CLR protects against such exploits.
Self-Protecting Application

Ordinary Insecure Application

AGENT
Adds missing security capabilities at runtime without changing existing code...

Self-Protecting Application
4. The use of measuring instruments to monitor and control a process. It is the art and science of measurement and control of process variables within a production, laboratory, or manufacturing area.
Source instrumentation

Inject simple static method call
Binary Instrumentation

- Widely used
  - CPU Performance
  - Memory
  - Logging
  - Security
  - …

- Lots of libraries
  - ASM (Java)
  - BCEL (Java)
  - Javassist (Java)
  - MBEL (.NET)
  - RAIL (.NET)
  - …
Dynamic Binary Instrumentation!

Binary code is enhanced as it loads.
1. `java -javaagent:security.jar`

2. `premain()`

3. `addTransformer()`

4. Loading original classes ...

5. `transform()`

6. Instrumented classes with Security

Diagram:
- JVM
  - ClassLoader
  - Class A
  - Class B
  - Class C
- Agent
  - ClassFile Transformer

Flow:
- Class A
- Class B
- Class C
Sensors are Woven
Instrumentation in Action

1. Add agent -javaagent:security.jar
2. Agent instruments running application
3. Agent blocks attacks and finds vulnerabilities
4. Dashboard provides visibility and control
Types of Sensors

**Vulnerability Sensors**
- Verify Security Configuration
- Verify Library Versions
- Verify Library Vulnerabilities
- Verify Control Flow Patterns
- **Verify Data Flow Patterns**
- Verify Coding Patterns

**Discovery Sensors**
- Identify Architecture
- Identify Connections
- Identify Security Controls
- Profile Application
- Report Technologies In Use
- Measure Lines of Code
What Does a Vulnerability Look Like?

c conn = pool.getConnection();
String sql = "select * from user where 
  username='" + username + "' and 
  password='" + password + ";"
stmt = conn.createStatement();
rs = stmt.executeQuery(sql);
if (rs.next()) {
  loggedIn = true;
  out.println("Successfully logged in");
} else {
  out.println("Invalid credentials");
}
It’s a “Path” Through Your Code!
<table>
<thead>
<tr>
<th>Source</th>
<th>$a = \text{request.getParameter(&quot;foo&quot;)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Flow</td>
<td>$b = a + \text{&quot;bar&quot;;}$</td>
</tr>
<tr>
<td>Control (Validation)</td>
<td>$\text{pattern.matches(&quot;[a-zA-Z0-9 ]&quot;)}$;</td>
</tr>
<tr>
<td>Data Flow</td>
<td>$c = b.replaceAll(&quot;foo&quot;,&quot;bar&quot;);$</td>
</tr>
<tr>
<td>Data Flow</td>
<td>$d = c.getBytes();$</td>
</tr>
<tr>
<td>Data Flow</td>
<td>$e = \text{new String}(d, &quot;UTF-8&quot;);$</td>
</tr>
<tr>
<td>Control (Encoding)</td>
<td>$f = \text{ESAPI.encodeForSQL}(e);$</td>
</tr>
<tr>
<td>Trigger</td>
<td>$\text{stmt.exec(&quot;SELECT * FROM &quot; + f);}$</td>
</tr>
</tbody>
</table>
Tagging

```
“user-input”    a = request.getParameter("foo")

b = a + "bar";

“limited-chars” pattern.matches("[a-zA-Z0-9 ]");

c = b.replaceAll("foo","bar");

“encoded”      d = ESAPI.encodeForSQL(c);

stmt.exec("SELECT * FROM " + d);
```

Safe!
Data flow analysis (aka clusterbomb)

HTTP Request

- getParameter("foo")
- Header
- Header
- Cookie
- URL Parameter
- URL Parameter
- Form Parameter
- Form Parameter
- ...

split("","")
append("<")
htmlEncode()
append(chunk1)
chunk3&lt;chunk1
html-encoded
cross-site
?

chunk1
chunk2
chunk3
chunk4
chunk5
...

Fine-Grained Tracking

```java
foo
a = request.getParameter("foo")

foobarrow
b = a + "bar" + a;

pattern.matches("[a-zA-Z0-9 ]");

foorfoo
c = b.replaceAll("ob", "");

foafoto

d = ESAPI.encodeForSQL(c);

SELECT * FROM 'foarfro'
stmt.exec("SELECT * FROM " + d);
```
### Agent capabilities

<table>
<thead>
<tr>
<th>Stack</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apps and APIs</td>
<td>• Analyze configuration files</td>
</tr>
<tr>
<td>3rd Party Libraries</td>
<td>• Analyze loaded libraries</td>
</tr>
<tr>
<td>3rd Party Frameworks</td>
<td>• Analyze HTTP request</td>
</tr>
<tr>
<td>Application Platform</td>
<td>• Analyze HTTP response</td>
</tr>
<tr>
<td>Container Runtime</td>
<td>• Analyze Backend connections</td>
</tr>
<tr>
<td>Container OS</td>
<td>• Report hardcode credentials</td>
</tr>
<tr>
<td>Physical Host or VM</td>
<td>• Report on weak ciphers</td>
</tr>
<tr>
<td>Your standard</td>
<td>• Report injection flaws</td>
</tr>
<tr>
<td></td>
<td>application stack(s)</td>
</tr>
</tbody>
</table>

#### Examples...

- Analyze configuration files
- Analyze loaded libraries
- Analyze HTTP request
- Analyze HTTP response
- Analyze Backend connections
- Report hardcode credentials
- Report on weak ciphers
- Report injection flaws
- Report vulnerable libraries
- Zero touch logging
- Deploy virtual patches
- Block attacks
# Library analysis – 3rd party / CVEs

- **68 known vulnerabilities** (across 28 libraries)
- **133.71M lines of library code**
- **1429 libraries** (28 Vulnerable 382 Stale)

## Average Known Library Grade

<table>
<thead>
<tr>
<th>Library</th>
<th>Grade</th>
<th>Libraries</th>
<th>CVE Date</th>
<th>Out of Date</th>
<th>Licence</th>
<th>Diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>commons-dbcp-1.3.jar</td>
<td>A</td>
<td>OpenCms</td>
<td>0.0</td>
<td>1.3 (02/07/2010)</td>
<td>1.4 (02/07/2010)</td>
<td>19/62</td>
</tr>
<tr>
<td>commons-digester-1.8.jar</td>
<td>D</td>
<td>OpenCms</td>
<td>0.0</td>
<td>1.8 (12/03/2006)</td>
<td>2.1 (09/24/2010)</td>
<td>20/100</td>
</tr>
<tr>
<td>commons-email-1.2.jar</td>
<td>F</td>
<td>OpenCms</td>
<td>0.0</td>
<td>1.2 (11/01/2009)</td>
<td>1.4 (05/18/2015)</td>
<td>2/10</td>
</tr>
<tr>
<td>commons-fileupload-1.2.2.jar</td>
<td>F</td>
<td>OpenCms</td>
<td>1.0</td>
<td>1.2 (07/29/2010)</td>
<td>1.3 (06/18/2016)</td>
<td>36/44</td>
</tr>
<tr>
<td>commons-httpclient-3.1.jar</td>
<td>F</td>
<td>OpenCms</td>
<td>4.0</td>
<td>3.1 (08/21/2007)</td>
<td>3.1 (08/21/2007)</td>
<td>2/167</td>
</tr>
<tr>
<td>commons-io-1.4.jar</td>
<td>F</td>
<td>OpenCms</td>
<td>0.0</td>
<td>1.4 (01/20/2008)</td>
<td>2.5 (04/30/2016)</td>
<td>0/76</td>
</tr>
<tr>
<td>commons-lang-2.4.jar</td>
<td>C</td>
<td>OpenCms</td>
<td>0.0</td>
<td>2.4 (03/19/2008)</td>
<td>2.6 (01/16/2011)</td>
<td>2/127</td>
</tr>
</tbody>
</table>
## Data Flow analysis - SQL injection

### SQL Injection from "uname" Parameter, "pass" Parameter on "login.jsp" page

_CRITICAL_ First Detected: 12/28/2016 10:52 AM  |  Status: Reported  |  ID: UOVBY-LWLN-NFVQ-095Q

<table>
<thead>
<tr>
<th>Source</th>
<th>Code</th>
<th>Database Query</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DANGEROUS DATA RECEIVED</strong></td>
<td><code>str = facade.getParameter('uname')</code></td>
<td><code>girish</code></td>
</tr>
<tr>
<td></td>
<td><code>at service() @ login.jsp:3</code></td>
<td></td>
</tr>
<tr>
<td><strong>DATA FLOWED FROM PARAMETER TO OBJECT</strong></td>
<td><code>sb = sb.append('girish')</code></td>
<td><code>select * from members where uname='girish'</code></td>
</tr>
<tr>
<td></td>
<td><code>at service() @ HttpJspBase.java:70</code></td>
<td></td>
</tr>
<tr>
<td><strong>DANGEROUS DATA RECEIVED</strong></td>
<td><code>str = facade.getParameter('pass')</code></td>
<td><code>girish</code></td>
</tr>
<tr>
<td></td>
<td><code>at service() @ login.jsp:4</code></td>
<td></td>
</tr>
<tr>
<td><strong>DATA FLOWED FROM PARAMETER TO OBJECT</strong></td>
<td><code>sb = sb.append('girish')</code></td>
<td><code>select * from members where uname='girish' and pass='girish'</code></td>
</tr>
<tr>
<td></td>
<td><code>at service() @ HttpJspBase.java:70</code></td>
<td></td>
</tr>
<tr>
<td><strong>DATA FLOWED FROM OBJECT TO RETURN VALUE</strong></td>
<td><code>str = sb.toString()</code></td>
<td><code>select * from members where uname='girish' and pass='girish'</code></td>
</tr>
<tr>
<td></td>
<td><code>at service() @ HttpJspBase.java:70</code></td>
<td></td>
</tr>
<tr>
<td><strong>RULE VIOLATION DETECTED</strong></td>
<td><code>rs = impl.executeQuery('select * from members where...girish' and pass='girish')</code></td>
<td><code>select * from members where uname='girish' and pass='girish'</code></td>
</tr>
<tr>
<td></td>
<td><code>at service() @ HttpJspBase.java:70</code></td>
<td></td>
</tr>
</tbody>
</table>
HTTP response analysis – Clickjacking

<table>
<thead>
<tr>
<th>Overview</th>
<th>How to Fix</th>
<th>Notes</th>
<th>Discussion</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Application</strong></td>
<td><strong>Status</strong></td>
<td><strong>Environments</strong></td>
<td></td>
</tr>
</tbody>
</table>

| DNN | REPORTED | Development |
| 17 Feb 2017 | First Detected |

| 26+ days | 26 days |
| Window of Exposure | Since Last Detected |

We observed 1 page without sufficient anti-clickjacking controls:

/DNN/Install/InstallWizard.aspx

The application doesn't apply anti-framing controls to the given page. Not applying these controls will allow it to be framed by other websites. This can present a few security issues, but the chief concern is that framed pages may be used in Clickjacking attacks.

Clickjacking (also called UI Redress) is an attack whereby an attacker can trick a user into clicking on something different from what it appears they are clicking on. Most attacks will first build a page that entices the user to click on it - for scandalous pictures, monetary rewards, and other promises. The buttons and links on this page are carefully placed so that they overlap with buttons, links or other items on a second page that gets layered on top of the first one.

The problem for the victim in this scenario is that the link that they'd like to click for the promised rewards on the evil page is actually "beneath" a translucent, second page which has been framed into the HTML document. Thus, when they intend to click on "Collect Your Prize!", they're really clicking on "Transfer Your Money!", or similar.

Numerous real-life Clickjacking attacks have caused damage to Adobe Flash Player, and numerous websites including Facebook, Twitter, YouTube, and others.

There are generally two controls that prevent framing of a web page: the X-Frame-Options header, and so-called frame-busting JavaScript. Details about these controls can be found in the Remediation tab. Neither of these anti-framing controls were found in the HTTP response to the following URL(s):
Configuration analysis – Authentication mode

SSL Not Required For Forms Authentication in \web.config

The configuration in \web.config was configured to use forms authentication and `requireSSL` was not set to `true` in the following authentication section:

```
144:     <!-- Forms or Windows authentication -->
145:     <authentication mode="Forms">
146:     <forms name=".DNNETNUKE" protection="All" timeout="60" cookieless="UseCookies"/>
147:     </authentication>
148:     <!--
```

When `requireSSL` is `true`, an SSL connection is required for forms authentication and the forms authentication cookie will have the 'secure' flag which prevents browsers from sending the cookie across unencrypted connections.

Neither of these protections are used when `requireSSL` is `false`. An attacker could eavesdrop on forms authentication requests sent over HTTP and learn users' credentials as well as the users' forms authentication cookies, leading to the compromise of user accounts.
Execution analysis – Cipher initialization

The code:

```java
org.owasp.webgoat.plugin.EncodingLesson#createContent(), line 243

...obtained a handle to the encryption algorithm seen here, which is considered insecure:

cipher = javax.crypto.Cipher.getInstance("PBEWithMD5AndDES/CBC/PKCS5Padding")

The encryption algorithm used, PBEWithMD5AndDES/CBC/PKCS5Padding, has been found by researchers to be unsafe for protecting sensitive data with today’s technology.
```
Blocking attacks

Sensors woven into running application

Security context assembled within agent

Developer
Tester
User
Attacker

Controller
Validation
Session
Business Logic
Data Layer
SQL API

Database

HTTP Request
Validation Tags
Data Tracking
Data Parsing
Escaping Tags
Query

✓ Vulnerability?
✓ Attack?
GET /foo?name=%20or%20%20'1'='1 HTTP/1.0

Three problems:
1) Bottleneck
2) No context
3) Impedance

stmt.execute("select * from table where id ='1' or '1'='1"");
RASP performance – same as code

<table>
<thead>
<tr>
<th>WebGoat</th>
<th>RASP Processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Typical traffic</td>
<td>50 microseconds</td>
</tr>
<tr>
<td>Mixed traffic</td>
<td>170 microseconds</td>
</tr>
<tr>
<td>Heavy attack traffic</td>
<td>230 microseconds</td>
</tr>
</tbody>
</table>

- Number of applications doesn’t matter
- No bottleneck on either bandwidth or CPU

millionths of a second
Accuracy, Automation and Scalability

You can’t scale appsec without **highly accurate tools**
(both true positives and true negatives)

Because inaccuracies **require** experts...

...and experts don’t scale.
Concluding Remarks

Instrumentation enables ...

- Application security in parallel (background),
- continuously across entire portfolio,
- without scans and bottlenecks,
- on modern software architecture
- at breakneck speed of development.
Thank you!

Girish Nair
girish@contrastsecurity.com

http://contrastsecurity.com