What do you mean my appsec tools don’t work on APIs?!!

Jeff Williams
@planetlevel
A HISTORY OF APPLICATION SECURITY AUTOMATION

Development (find vulnerabilities)

- SAST (Static AppSec Testing)
- DAST (Dynamic AppSec Testing)
- IAST (Interactive AppSec Testing)

Operations (block attacks)

- WAF (Web Application Firewall)
- IDS/IPS (Intrusion Detection/Prevention System)
- RASP (Runtime Application Self-Protection)

Unified Agent IAST and RASP
How long has your organization been providing or developing APIs?

*EVERYONE* USES APIS

*SmartBear State of API 2016 Survey*
APIS ARE THE NEW “HIDDEN FIELD”

Automated Access
- mobile
- thing
- desktop
- server

Account Hijacking
Direct Object References
Corruption
Injection
Denial of Service
Information Theft
...

Business Capabilities
- code
- sql
- nosql
- files
- actions

Similar threats, similar vulnerabilities – testing is all different
WHAT TECHNIQUE SHOULD YOU USE TO TEST APIS?

Pentest
Dynamic (DAST)
JavaSnoop

Proxy

Code Review
Static (SAST)
Instrumentation (IAST)

Client(s)

API(s)
VERIFYING API SECURITY - SIX PROBLEMS

1. Protocols
2. Data Parsers
3. API Specifications
4. Frameworks and Data Flow
5: AuthN and AuthZ
6: Futures and Promises

Why is automated verification of APIs different than plain old web applications?
PROBLEM 1: PROTOCOLS

• Protocol could be multiple nested protocols, or even custom protocol
  – Websocket, XMPP, AMQP, MQTT
  – Without a lot of work, we can’t scan, intercept, or block custom protocols

```javascript
var connection = new WebSocket('ws://html5rocks.websocket.org/echo',
  ['soap', 'xmpp']);
```

• Intercepting is now really hard
  – Could spend your whole pentest just getting proxy
  – Especially difficult with SSL
  – IOS: Setup reverse proxy, modify trust store, reverse/recompile with forward proxy, use iOS kill switch to avoid DNS pinning, etc…
GWT TRAFFIC

5|0|12|http://192.168.0.1:9080/gwt_test/|5E2238F4BEC12E99ABC5F1B1D661286C|
PersonName|java.lang.Integer/374839484|CustomObjParam1|CustomObjParam2|
CustomObjParam3|1|2|3|4|2|5|6|5|2|7|200|8|7|200|8|6|9|200|10|11|12|10|
PROBLEM 2: DATA PARSERS

• Denial of Service (DOS) – entity expansion bombs
  – Billion laughs attack

• eXternal Entity (XXE)
  – DOCTYPE causes parser to include sensitive data

• Server Side Request Forgery (SSRF)
  – DOCTYPE causes parser to emit malicious HTTP requests

• Untrusted Deserialization
  – Deserializing arbitrary classes enables remote code execution (RCE)

• Standard: URL, JSON, GWT, serialized objects, etc…

• Custom
It is quite difficult to predict what concrete JAXP factory implementation will be loaded without actually creating an instance because the process for selecting an implementation.

From the Official JAXP FAQ (Question 14):

When an application wants to create a new JAXP `DocumentBuilderFactory` instance, it calls the static method `DocumentBuilderFactory.newInstance()`. This causes a search for the name of a concrete subclass of `DocumentBuilderFactory` using the following order:

1. The value of a system property like `javax.xml.parsers.DocumentBuilderFactory` if it exists and is accessible.
2. The contents of the file `$JAVA_HOME/jre/lib/jaxp.properties` if it exists.
3. The Jar Service Provider discovery mechanism specified in the Jar File Specification. A jar file can have a resource (i.e. an embedded file) such as `META-INF/services/javax.xml.parsers.DocumentBuilderFactory` containing the name of the concrete class to instantiate.
4. The fallback platform default implementation.
PROBLEM 3: API SPECIFICATIONS

• Can your tool understand the API?
  – RMI, CORBA, IDL
  – WSDL, WADL, DTD, Slate
  – Swagger, Blueprint, RAML
  – Annotations
  – None?

• What do you send?
  – How do you make API run?

• Most specifications don’t tell you anything about security
apis: [
  - {
    path: "/pet/{petId}",
    operations: [
      - {
          method: "DELETE",
          summary: "Deletes a pet",
          notes: "",
          type: "void",
          nickname: "deletePet",
          authorizations: {
            oauth2: {
              scope: "write:pets",
              description: "modify pets in your account"
            }
          }
        },
    parameters: [
      - {
          name: "petId",
          description: "Pet id to delete",
          required: true,
          type: "string",
          paramType: "path",
          allowMultiple: false
        }
    ]
  }
]

the point of attack

HTTP Method: Are other methods handled correctly?

Oauth 2.0: are tokens enforced and validated correctly?

Is access validated? Are ids sequential? Injection point?, etc

What if we send multiple? Or none at all?
### Problem 4: Frameworks Obscure Data/Control Flow

#### RESTful Web Service

```java
@RestController
class GreetingController {
    private static final String template = "Hello, %s!";
    private final AtomicLong counter = new AtomicLong(0);

    @RequestMapping("/greeting")
    public Greeting greeting(@RequestParam(value="name", defaultValue="World")
        return new Greeting(counter.incrementAndGet(),
            String.format(template, name));
}
```

http://localhost:8080/greeting?name=User

```
{"id":1,"content":"Hello, User!"}
```

---

<table>
<thead>
<tr>
<th>Framework</th>
<th>Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Axis</td>
<td>Apache Wink</td>
</tr>
<tr>
<td>Apache Axis2</td>
<td>Apache Tuscany</td>
</tr>
<tr>
<td>Apache CXF</td>
<td>Cuubez</td>
</tr>
<tr>
<td>Jersey</td>
<td>Everest</td>
</tr>
<tr>
<td>Web Services Interoperability Technology</td>
<td>RESTX</td>
</tr>
<tr>
<td>Web Services Invocation Framework</td>
<td>RestExpress</td>
</tr>
<tr>
<td>XFire became Apache CXF</td>
<td>restSQL</td>
</tr>
<tr>
<td>XML Interface for Network Services</td>
<td>Sparkjava</td>
</tr>
<tr>
<td>JAX-RS</td>
<td>Retrofit</td>
</tr>
<tr>
<td>JAX-WS</td>
<td>Swagger</td>
</tr>
<tr>
<td>Spring WS</td>
<td>Spring Web MVC</td>
</tr>
<tr>
<td>Sun Metro</td>
<td>Jooby</td>
</tr>
<tr>
<td>Restfulie</td>
<td>Ninja</td>
</tr>
<tr>
<td>Restlet</td>
<td>Jodd</td>
</tr>
<tr>
<td>Dropwizard (Jetty, Jersey, Jackson)</td>
<td>Jhipster</td>
</tr>
<tr>
<td>Jackson</td>
<td>Javalite</td>
</tr>
<tr>
<td>RESTEasy</td>
<td>Ratpack</td>
</tr>
<tr>
<td>Play</td>
<td>Vert.x</td>
</tr>
</tbody>
</table>
The calls to `request.getParameter()` and other untrusted sources are no longer in the source code anywhere!
INTERMISSION
A HISTORY OF APPLICATION SECURITY AUTOMATION

Development (find vulnerabilities):
- SAST (Static AppSec Testing)
- DAST (Dynamic AppSec Testing)
- IAST (Interactive AppSec Testing)

Operations (block attacks):
- WAF (Web Application Firewall)
- IDS/IPS (Intrusion Detection/Prevention System)
- RASP (Runtime Application Self-Protection)

Unified Agent: IAST and RASP
OWASP Benchmark

• Extensive test suite to measure appsec tools
• Same tests for SAST, DAST, IAST, {WAF, RASP}
• Scientifically test capabilities of tools

Sponsored by DHS
THE OWASP BENCHMARK PROJECT

<table>
<thead>
<tr>
<th>Vulnerability Category</th>
<th>True Vulnerabilities</th>
<th>False Vulnerabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Command Injection</td>
<td>1802</td>
<td>906</td>
</tr>
<tr>
<td>2 Cross Site Scripting</td>
<td>1540</td>
<td>1909</td>
</tr>
<tr>
<td>3 Insecure Cookie</td>
<td>201</td>
<td>215</td>
</tr>
<tr>
<td>4 LDAP Injection</td>
<td>521</td>
<td>215</td>
</tr>
<tr>
<td>5 Path Traversal</td>
<td>1706</td>
<td>924</td>
</tr>
<tr>
<td>6 SQL Injection</td>
<td>2297</td>
<td>1232</td>
</tr>
<tr>
<td>7 Trust Boundary Violation</td>
<td>505</td>
<td>220</td>
</tr>
<tr>
<td>8 Weak Encryption Algorithm</td>
<td>720</td>
<td>720</td>
</tr>
<tr>
<td>9 Weak Hash Algorithm</td>
<td>714</td>
<td>707</td>
</tr>
<tr>
<td>10 Weak Random Number</td>
<td>1612</td>
<td>2028</td>
</tr>
<tr>
<td>11 XPath Injection</td>
<td>217</td>
<td>130</td>
</tr>
<tr>
<td>Totals</td>
<td>11835</td>
<td>9206</td>
</tr>
</tbody>
</table>

https://www.owasp.org/index.php/Benchmark

21,041 test cases
TP and FP
Free and open
Totally reproducible
This is the easy easy stuff…
FREE, OPEN, AND REPRODUCIBLE FOR ANYONE

1. Test Suite

Source Code
- Findbugs
- HP Fortify
- PMD
- IBM AppScan Source
- Veracode
- CheckMarx
- Synopsys Coverity
- Parasoft
- SonarQube

Running App
- AppScan Dynamic
- Burp Suite Pro
- OWASP ZAP
- Arachni
- Acunetix
- Rapid7 AppSpider
- HP WebInspect
- WhiteHat
- Netsparker
- Google Skipfish
- w3af

2. Scorecard Generator

Expected Results

Tool Reports with Actual Results

3. Benchmark Scorecard(s)
Every false alarm costs an organization the opportunity to eliminate real vulnerabilities. It’s trivial to write a tool that reports *everything* and it’s trivial to write a tool that reports *nothing*. Ideal AppSec tools remain far from ideal in practice.
The results are fascinating…
- Wildly varying results for different tools
- Some tools are amazing at certain things
- Some tools have inexplicable weaknesses

Driving improvement…
- Fortify, Checkmarx, Contrast, Arachni, ZAP, FindSecBugs, and SonarQube have already made improvements to their tools
## ADDING API TEST CASES TO OWASP BENCHMARK

<table>
<thead>
<tr>
<th>Vulnerability Category</th>
<th>True Vulnerabilities</th>
<th>False Vulnerabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Command Injection</td>
<td>56</td>
<td>56</td>
</tr>
<tr>
<td>2 Insecure Cookie</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>3 LDAP Injection</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>4 Path Traversal</td>
<td>56</td>
<td>56</td>
</tr>
<tr>
<td>5 SQL Injection</td>
<td>116</td>
<td>116</td>
</tr>
<tr>
<td>6 Trust Boundary Violation</td>
<td>20</td>
<td>60</td>
</tr>
<tr>
<td>7 Weak Encryption Algorithm</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>8 Weak Hash Algorithm</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>9 Weak Random Number</td>
<td>160</td>
<td>128</td>
</tr>
<tr>
<td>10 XPath Injection</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td><strong>Totals</strong></td>
<td><strong>576</strong></td>
<td><strong>584</strong></td>
</tr>
</tbody>
</table>

We are contributing over 1,100 new API test cases to the OWASP Benchmark Project.

Total lines: 132,451

No XSS
package org.owasp.benchmark.service.testcode;

import java.io.IOException;

@RestController
public class BenchmarkTest00072 {

    @RequestMapping(value = "/BenchmarkTest00072", method = RequestMethod.POST)
    public @ResponseBody Person createOrder(@RequestBody Person model, HttpServletRequest req, HttpServletResponse resp) {
        String param = model.getName();
        if (param == null) param = "";
        String cmd = org.owasp.benchmark.helpers.Utils.getOSCommandString("echo");
        String[] argsEnv = {"message"};
        Runtime r = Runtime.getRuntime();
        try {
            Process p = r.exec(cmd + param, argsEnv);
            org.owasp.benchmark.helpers.Utils.printOSCommandResults(p, resp);
        } catch (IOException e) {
            throw new ServletException("Problem executing cmdi", e);
        }
        return model;
    }
}
PRELIMINARY RESULTS

STATIC TOOL 1

Total vulnerabilities reported: 3,393
PRELIMINARY RESULTS

STATIC TOOL 2

Total vulnerabilities reported: 560,000
Contrast analyzes APIs from the inside.

Contrast IAST/RASP Agent instruments your application with sensors that protect against both vulnerabilities and attacks.

All agents report to Contrast TeamServer to protect the entire application portfolio in parallel.
DETECTING AND BLOCKING BOTH ATTACKS AND VULNERABILITIES

Sensors woven into running application

Security context assembled within Contrast Agent

Controller
Validation
Session
Business Logic
Data Layer
SQL API

Developer
Tester
User
Attacker

Database

HTTP Request
Validation Tags
Data Tracking
Data Parsing
Escaping Tags
Query

✓ Vulnerability?
✓ Attack?
ACCURACY: IAST/RASP HAS AN UNFAIR ADVANTAGE

IAST/RASP provide full visibility into running application
PRELIMINARY RESULTS

CONTRAST

Total vulnerabilities reported: 576
KEY TAKEAWAYS

• APIs are a large and increasing part of your attack surface
  – Legacy SAST and DAST tools don’t work on APIs
  – Manual penetration testing and code review don’t work on APIs

• Are you sure your APIs are protected?

• Contrast’s instrumentation-based approach is effective
  – Both traditional web applications and APIs

• Use the OWASP Benchmark Project when choosing tools
SOFTWARE TRENDS CHALLENGING SAST/DAST/WAF

Explosive growth in libraries and frameworks

- SAST can’t handle scale and complexity of supply chain

Microservices, APIs, REST/XML services

- SAST and DAST can’t handle API and web service complexity

Rapidly growing use of cloud and containers

- WAF can’t handle infrastructure deployment pace and complexity

High speed software development

- SAST, DAST, and WAF all require experts in the critical path
PROBLEM 4: DATA FLOW

• Data flow is critical!
  – Primary cause of missed vulnerabilities and false positives
  – JSON -> parser -> binding -> bean -> query -> SQL Injection
  – Websocket -> byte[] -> custom parser -> filename -> Path Traversal

• Complex data flow is hard to track
  – Sources, sinks, and propagators are now deep in libraries
  – Every framework works differently

• Tools
  – Instrumentation (IAST) has highly accurate data flow, regardless of frameworks
ABUSING URLS

• Authentication http://user:pass@example.com/foo/bar
• HTTP Parameter Pollution (HPP)
• Faking the method with _method parameter
• NoSQL REST APIs follow their schema
• Every service has its own approach

* http://www.slideshare.net/DinisCruz/res-ting-on-your-laurels-will-get-you-powned4-3
DIFFERENCE 5: AUTHENTICATION AND AUTHORIZATION

• Many techniques…
  – HTTP BASIC authentication over HTTPS
  – Cookies and session management
  – Token in HTTP headers (e.g. OAuth 2.0)
  – Query Authentication with additional signature parameters

• Often custom implementation

• Tools
  – All tools require custom auth’n and auth’z rules
DIFFERENCE 6: FUTURES AND PROMISES

• APIs are far more likely to take advantage of concurrency
  – Netty, Play, Grizzly are all heavily multithreaded
  – Adds complexity to control and data flows

• Tools
  – Dynamic (DAST) and Interactive (IAST) tools avoid these problems