SECURITY OBSERVABILITY 101: THINKING INSIDE THE BOX!

JEFF WILLIAMS, COFOUNDER AND CTO
@PLANETLEVEL

OWASP CHARLOTTE – OCT 2021
96% of applications have at least one vulnerability

The average software project introduces 2-3 new vulnerabilities every month

The average application has 30+ vulnerabilities and 2+ high or critical flaws in open source libraries

Average application codebase:
– 20% is custom code
– 6% is OSS that actually runs
– 74% is never used

Only 14% of libraries are the latest version

The average application is attacked over 13,000 times a month

Zero applications were not attacked every single month

99% of attacks do not connect with their intended vulnerability

Attacks on all vulnerabilities are trending up over the last 12 months
INSTRUMENTATION CHANGES EVERYTHING
INSTRUMENTATION IS EASY

Agent
Transformer
Loader

Platform

Transformer adds security sensors to code

Instrumented Code

Run instrumented code!

Original Code
ADD IN SENSORS TO REVEAL SECURITY

USE INSTRUMENTATION TO DEFINE SENSORS THAT MAKE SOFTWARE BEHAVIOR OBSERVABLE

WIN!

“RUNTIME REALITY” FULLY ASSEMBLED AND RUNNING APPS/APIS ARE THE ONLY SOURCE OF TRUTH

INSTRUMENTATION PROVIDES CONTINUOUS ACCURATE REALTIME TELEMETRY

USE INSTRUMENTATION TO DEFINE SENSORS THAT MAKE SOFTWARE BEHAVIOR OBSERVABLE
THE JAVA OBSERVABILITY TOOLKIT (JOT)
FREE OPEN SOURCE INSTRUMENTATION

https://github.com/planetlevel/jot
RIDICULOUSLY SIMPLE EXAMPLE: WEAK SQL QUERIES

sensors:
- name: "get-unsafe-queries"
  description: "Identifies unparameterized database queries"
  methods:
  - "java.sql.Statement.executeQuery"
  - "java.sql.Statement.addBatch"
- "java.sql.Statement.executeUpdate"
  excludes:
  - "java.sql.PreparedStatement" # not vulnerable subclass
  - "#ARGS"

```
$ export JAVA_TOOL_OPTIONS="-javaagent:jot-0.9.jar=rules/usql.jot"

[JOR get-unsafe-queries] com.acme.ticketbook.Database.updateUnsafe
(Database.java:173) [INSERT INTO tickets(name,city,cc,ticket) VALUES('Arshan Dabirsiaigi', 'Baltimore', '/j7B2e38B93GMJNNTVRMXD2JAYE
f76', '18002')]  
[JOR get-unsafe-queries] com.acme.ticketbook.Database.updateUnsafe
(Database.java:173) [INSERT INTO tickets(name,city,cc,ticket) VALUES('Harold McGinnis', 'Philadelphia', 'uWtJbTHcGaGF/bvouf9w5WcVwSa
2Avr', '18003')]  
[JOR get-unsafe-queries] com.acme.ticketbook.Database.updateUnsafe
(Database.java:173) [INSERT INTO tickets(name,city,cc,ticket) VALUES('Chris Schmidt', 'Denver', 'JFEu+fcb7lwvRj3KX1DD0wrsqmpDrPvn', '18004')]  
[JOR get-unsafe-queries] com.acme.ticketbook.Database.queryUnsafe
(Database.java:151) [SELECT * FROM tickets]
TRACE-10004(1)  
[JOR get-unsafe-queries] com.acme.ticketbook.Database.queryUnsafe
(Database.java:151) [SELECT * FROM tickets]
TRACE-10006(1)  
[JOR get-unsafe-queries] com.acme.ticketbook.Database.queryUnsafe
(Database.java:151) [SELECT * FROM tickets WHERE ticket='JOT FTW']
```
WHAT ENCRYPTION IS HAPPENING?

In JOT, a “capture” is a “spring expression” (SPEL) that allows you to extract data using references to objects in the running app/API.

- #P0 is the first parameter to the method
- #OBJ is the object itself
- #RET is the return value from the method

You can call methods on these references!!!
sensors:
  - name: "get-routes"
    description: "Identifies the route for this HTTP request"
    methods:
    - "javax.servlet.Servlet.service"
    captures:
    - "#P0.getRequestURI()"
  - name: "get-users"
    description: "Identifies user names"
    methods:
    - "javax.servlet.Servlet.service"
    captures:
    - "#P0.getRemoteUser() ?: "Guest""
  - name: "get-role"
    description: "Identifies roles"
    methods:
    - "javax.servlet.ServletRequest.isUserInRole"
    captures:
    - "#P0"

reports:
  - name: "Test Coverage Matrix"
    type: "compare"
    rows: "get-routes"
    cols: "get-users"
  - name: "Access Control Matrix"
    type: "compare"
    rows: "get-routes"
    cols: "get-role"

VERIFYING ACCESS CONTROL?

$ export JAVA_TOOL_OPTIONS="-javaagent:jot-0.9.1.jar=jots/access.jot"

<table>
<thead>
<tr>
<th>Test Coverage Matrix</th>
<th>Guest</th>
<th>UserB</th>
<th>UserC</th>
<th>UserD</th>
<th>UserE</th>
</tr>
</thead>
<tbody>
<tr>
<td>/ticketbook/accessA.jsp</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>/ticketbook/accessB.jsp</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>/ticketbook/accessC.jsp</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>/ticketbook/accessD.jsp</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>/ticketbook/accessE.jsp</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>/ticketbook/architecture.jsp</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/ticketbook/cmd.jsp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/ticketbook/forward.jsp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/ticketbook/hash.jsp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/ticketbook/redirect.jsp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/ticketbook/xss.jsp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/ticketbook/xxe.jsp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Access Control Matrix</th>
<th>RoleA</th>
<th>RoleB</th>
<th>RoleC</th>
<th>RoleD</th>
<th>RoleE</th>
</tr>
</thead>
<tbody>
<tr>
<td>/ticketbook/accessA.jsp</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/ticketbook/accessB.jsp</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/ticketbook/accessC.jsp</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>/ticketbook/accessE.jsp</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
CHEATING AT PENTESTS FOR FUN AND PROFIT

SQL Injection

EL Injection

Command Injection
Define this “scope” and add an exception. Now your normal test cases fail for security reasons if your sensor fires!
HOW CAN I PREVENT EXPRESSION LANGUAGE INJECTION FROM BEING EXPLOITED?

Runtime Protection!
**IAST**
Interactive Application Security Testing is simply using instrumentation to detect vulnerabilities.
USE IT IN DEVELOPMENT.

**RASP**
Runtime Application Self-Protection is simply using instrumentation to detect attacks and prevent exploits.
USE IT IN PRODUCTION
THE MOVE TO MODERN SOFTWARE SECURITY

LEGACY

SCAN AND FIREWALL MODEL

- Disruptive, bottleneck
- Can’t keep up, even with army of experts
- After the fact, inaccurate
- Snapshot in time
- Tool soup, security silos

MODERN

EMBEDDED MODEL

- Embedded, works in flow, frictionless
- Force multiplier, no experts required
- Direct observation, instant feedback
- Continuous, always-on
- One platform across dev, sec, ops
Security observability accelerates innovation

Security observability (inside - out)

Instrumentation (IAST, SCA, RASP)

Self-testing

Custom code
Libraries
Frameworks
App server
Runtime

Self-protecting

Vulnerabilities

Instant feedback

Attacks

Continuous automated security testing and exploit prevention

Flow
Business value + continuous assurance
OBSERVABILITY YIELDS A 17X IMPROVEMENT IN MTTR OVER SCANNING

MEAN TIME TO REMEDIATE

% OF VULNERABILITIES REMAINING

DAYS SINCE DISCOVERY

- SAST
- IAST
- IAST (BELOW AVERAGE BACKLOG)

- 29% CLOSED IN 30 DAYS
- MEDIAN CLOSED IN 7 DAYS
- MEDIAN CLOSED IN 121 DAYS
- 61% CLOSED IN 30 DAYS
- 75% CLOSED IN 8 DAYS
- 90% CLOSED IN 1 YEAR
- 95% CLOSED IN 1 YEAR
- 71% CLOSED IN 1 YEAR
“How to Vulnerability”

https://www.linkedin.com/pulse/how-vulnerability-jeff-williams
“Making Security in a Software Factory”

https://www.linkedin.com/pulse/making-security-software-factory-jeff-williams/