Dynamic Taint Propagation

Finding Vulnerabilities Without Attacking

Brian Chess / Jacob West

Fortify Software

2.21.08

Overview

- Motivation
- Dynamic taint propagation
- Sources of inaccuracy
- Integrating with QA
- Related work
- Parting thoughts

MOTIVATION

Existential Quantification

$\exists$

"there exists"

There exists a vulnerability (again).

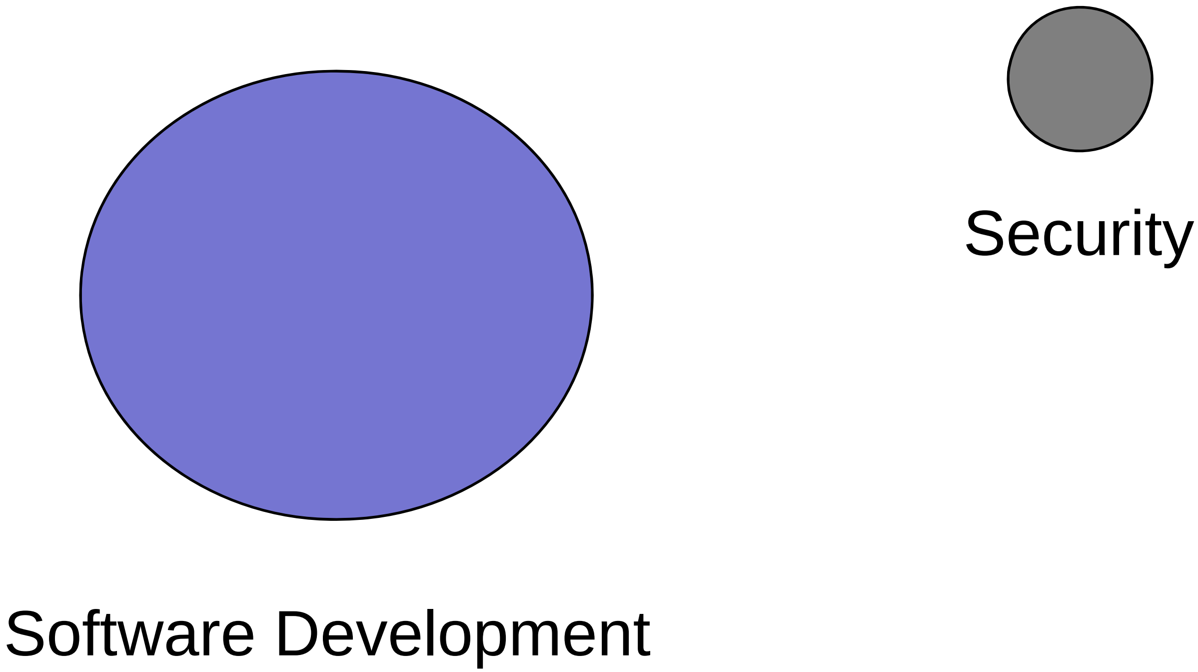
Universal Quantification

$\forall$

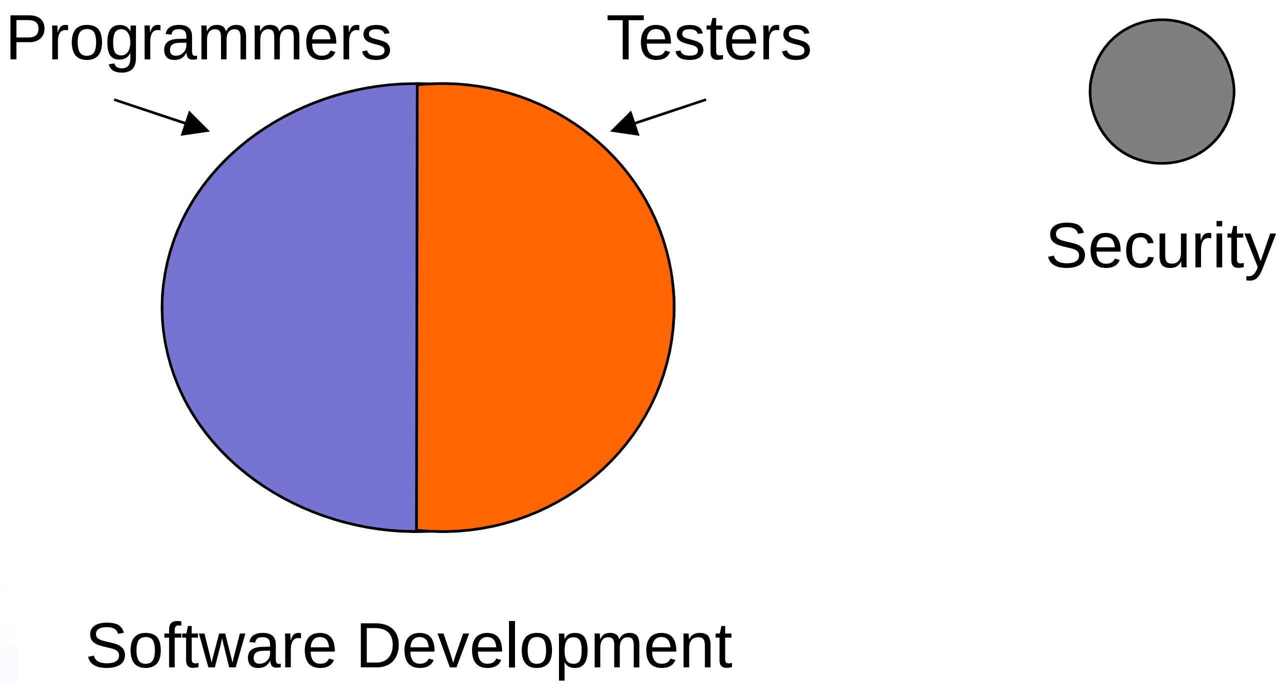
"for all"

For all bad things that might happen, the program is safe.

Security vs. Software Development



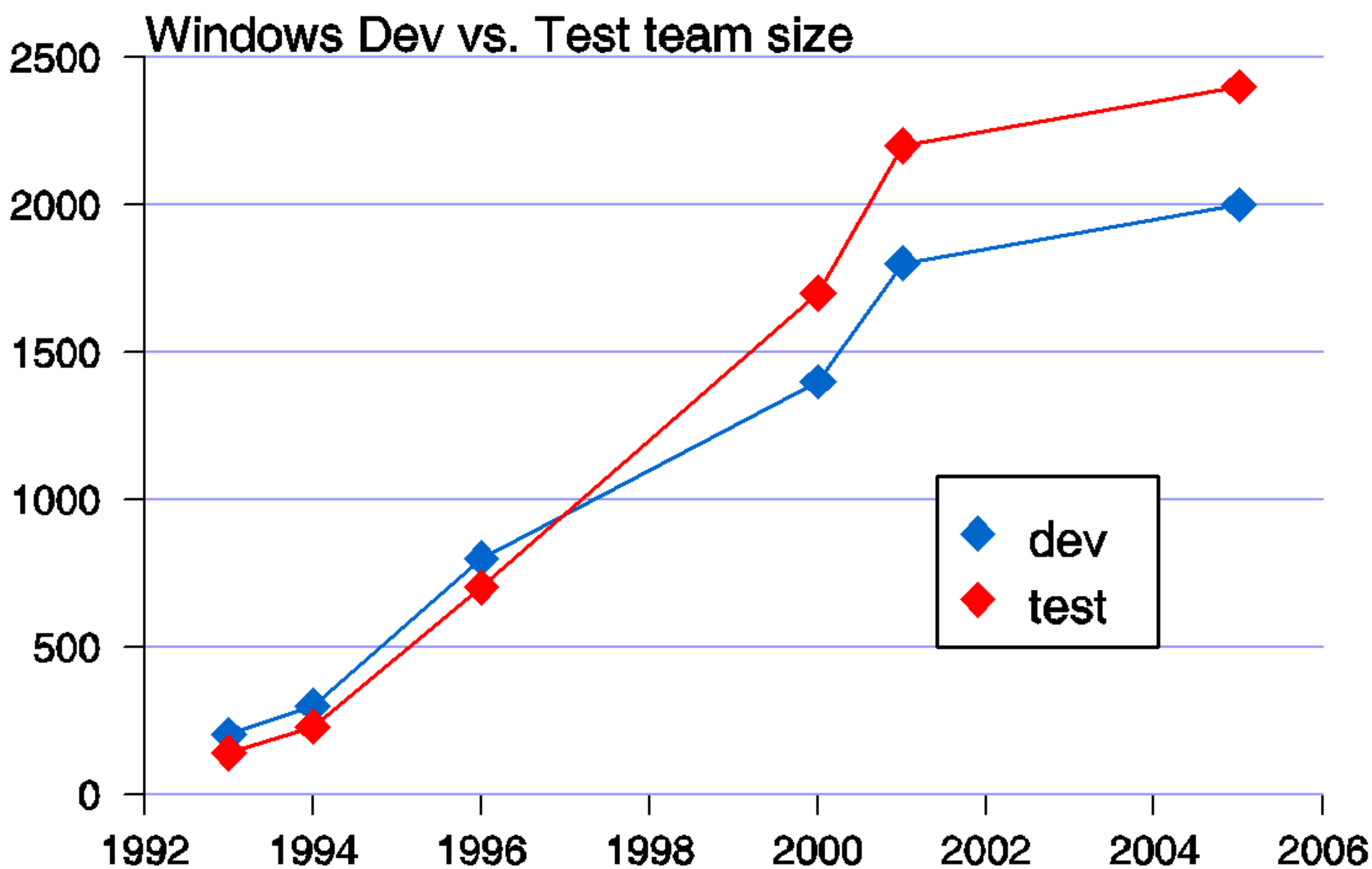
Security vs. Software Development



Are you going to give me Yet Another Lecture About Static Analysis (YALASA)?

- No
- Focus on QA
- Using static analysis requires understanding code

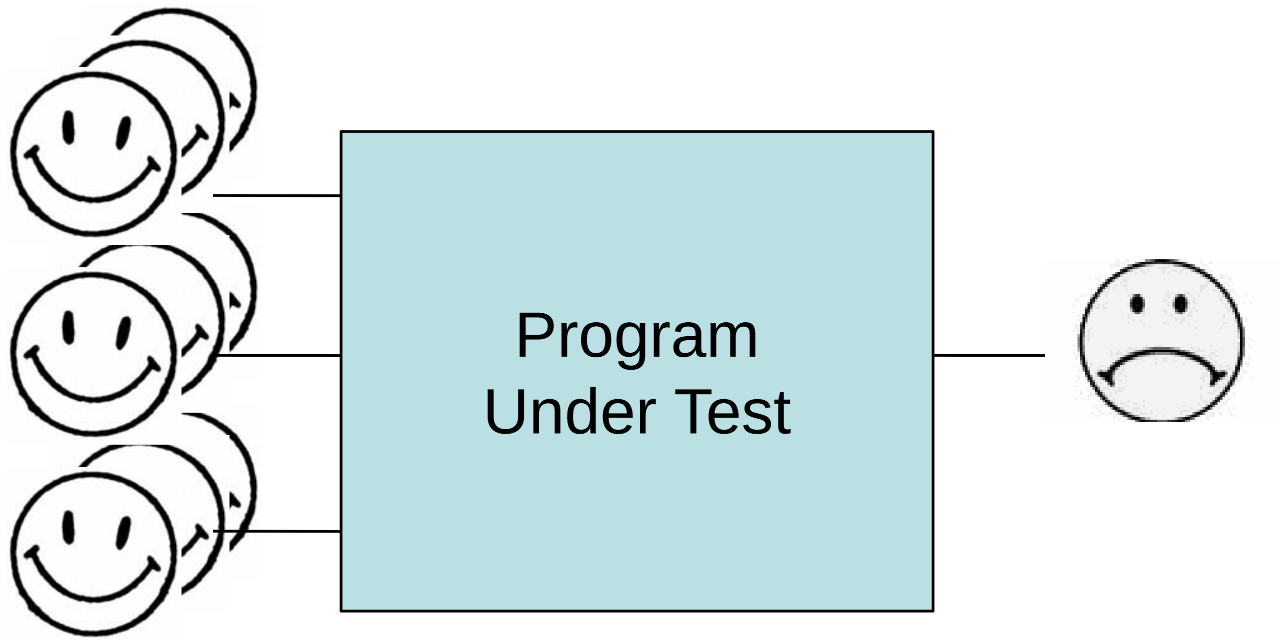
Team Sizes at Microsoft



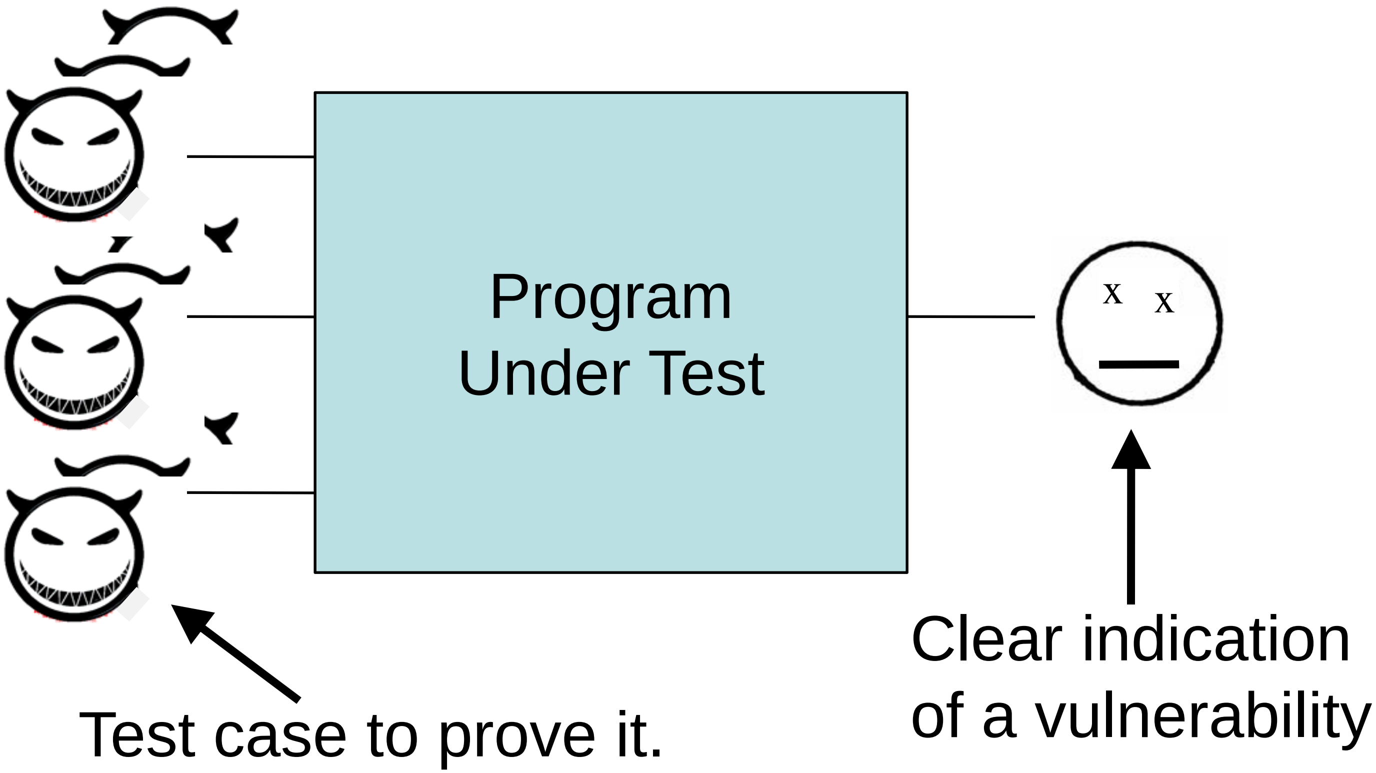
QA Testers vs. Security Testers

Functional Testers	Security Testers
Know the program.	Know security.
Need high functional coverage.	Need to find at least one vulnerability.
Lots of time and resources (comparatively).	Often arrive at the party late and are asked to leave early.

Typical Software Testing

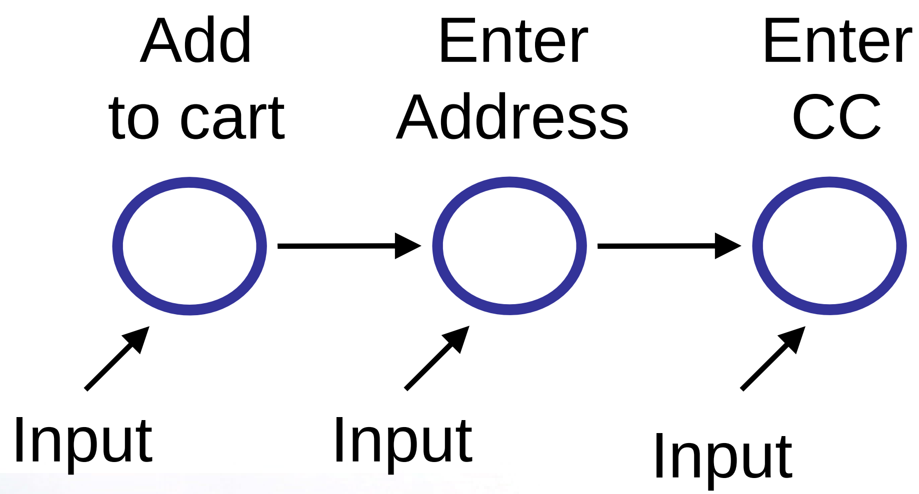


Typical Security Testing



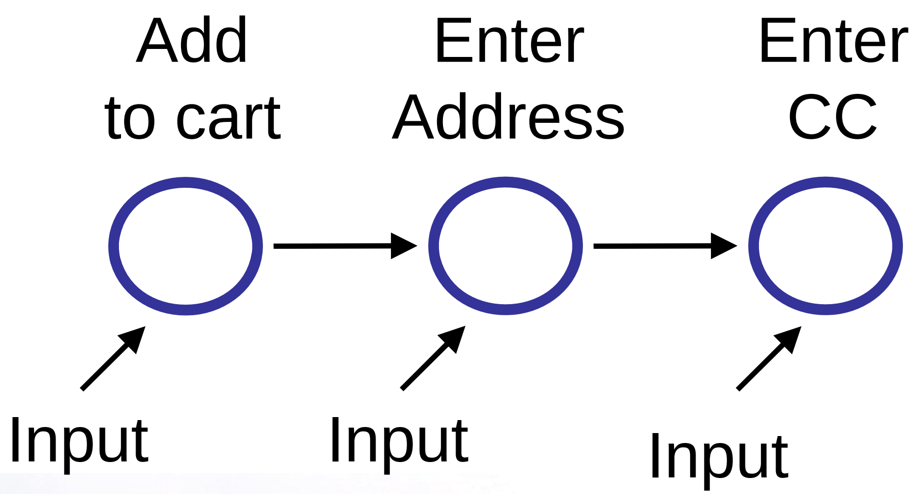
Fault Injection Failings

- Bad input derails normal program flow
- Cannot mutate functional tests and retain coverage



Fault Injection Failings

- Result: bad test coverage
- Result: missed vulnerabilities



Problem Summary

- QA has, security team lacks:
  - Good test coverage
  - Time and resources
- Security team has, QA lacks:
  - Security clue

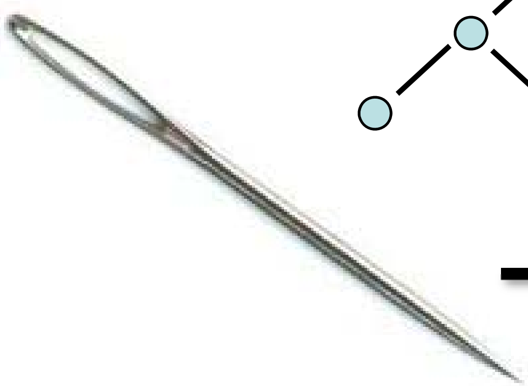
Involve QA in Security

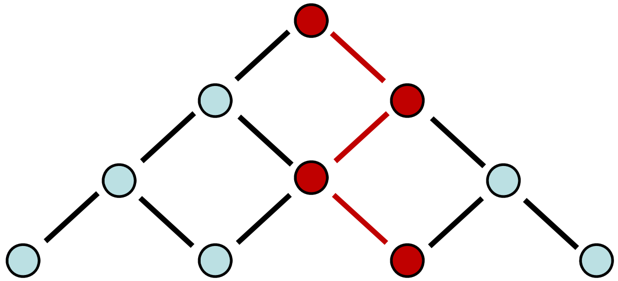
- Ease of use
  - Favor false negatives over false positives
  - Expect security team to test too
- Leverage existing QA tests
  - Achieve high coverage
  - Must be transformed into security tests

DYNAMIC TAINT PROPAGATION

Dynamic Taint Propagation

- Follow untrusted data and identify points where they are misused





Example: SQL Injection

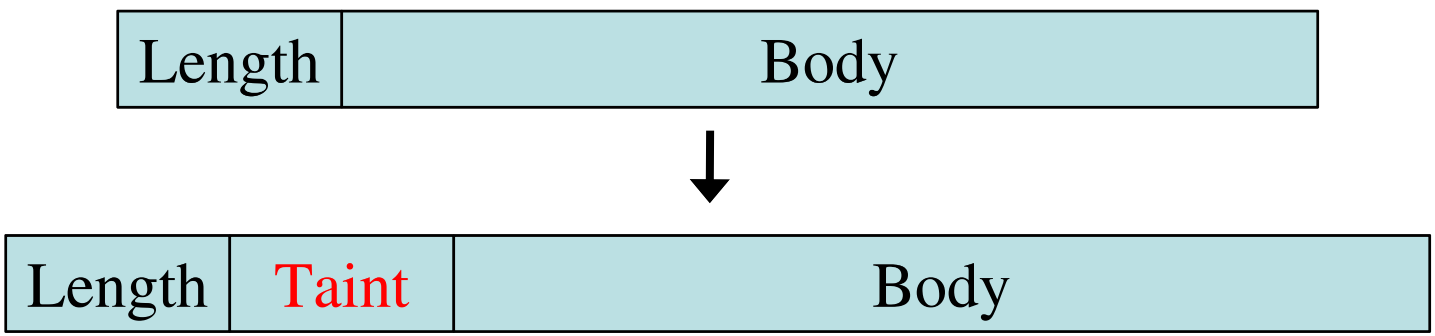
```
...
user = request.getParameter("user");
try {
 sql = "SELECT * FROM users " +
       "WHERE id='" + user + "'";
 stmt.executeQuery(sql);
}
...
```

Tracking Taint

1. Associate taint marker with untrusted input as it enters the program
2. Propagate markers when string values are copied or concatenated
3. Report vulnerabilities when tainted strings are passed to sensitive sinks

Java: Foundation

- Add taint storage to **`java.lang.String`**



Java: Foundation

- **`StringBuilder`** and **`StringBuffer`** propagate taint markers appropriately

Untainted	+	Untainted	=	Untainted
Untainted	+	Tainted	=	Tainted
Tainted	+	Tainted	=	Tainted

Java: Sources

- Instrument methods that introduce input to set taint markers, such as:
  - **`HttpServletRequest.getParameter()`**
  - **`PreparedStatement.executeQuery()`**
  - **`FileReader.read()`**
  - **`System.getenv()`**
  - **`...`**

Java: Sinks

- Instrument sensitive methods to check for taint marker before executing, such as:
  - **`Statement.executeQuery()`**
  - **`JspWriter.print()`**
  - **`new File()`**
  - **`Runtime.exec()`**
  - **`...`**

Example: SQL Injection

```
user = request.getParameter("user");
TaintUtil.setTaint(user, 1);
try {
  sql = "SELECT * FROM users " +
        "WHERE id='" + user + "'";
  TaintUtil.setTaint(sql,user.getTaint());
  TaintUtil.checkTaint(sql);
  stmt.executeQuery(sql);
}
```

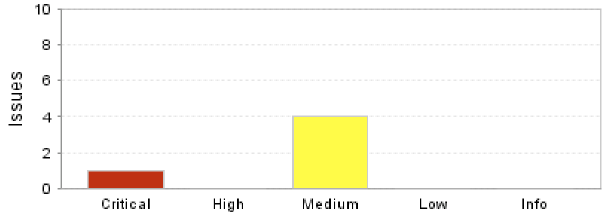
Results Overview

Current Run Clear Pause New Run [Export to Fortify Manager] [Import Configs] Events File: Browse... Import Events

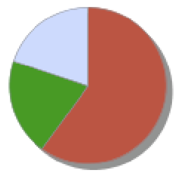
Name: Random Status: In Progress

Security Issues

Issues by Severity



Issues by Category



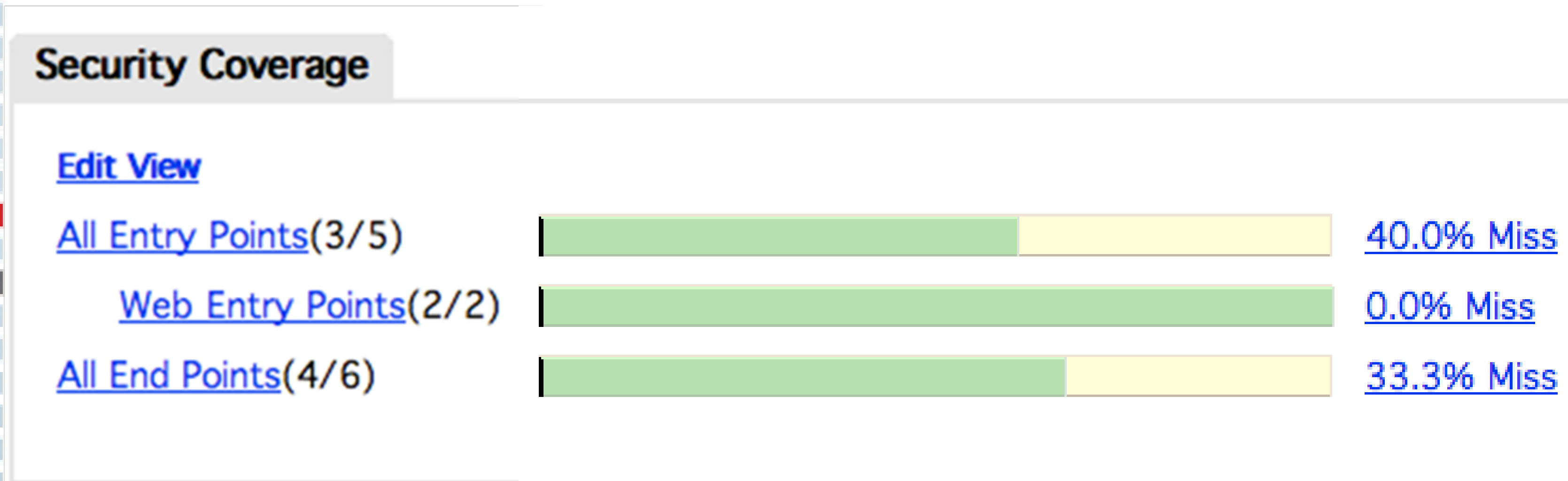
Log Forging: 3 SQL Injection: 1 Unhandled Exception: 1

Security Coverage

Edit View

All Entry Points(3/5)		40.0% Miss
Web Entry Points(2/2)		0.0% Miss
All End Points(4/6)		33.3% Miss

Security Coverage



SQL Injection Issue

Search:

✗	Run	is	SPLC:Random
✗	Category	is	SQL Injection

View/Edit Application View Options

Displaying 1 out of 12 events.

Group By: Entry Point File submit

Expand All Collapse All

Change all displayed events:

Suppress ALL

Unsuppress ALL

Events: 1 total

Category	Entry Point Type	End Point Type	Issues
SQL Injection	Web	Database	1

Entry Point File

org.apache.coyote.tomcat5.CoyoteRequestFacade:295

Entry Point Method	End Point File	URL	Audit Status	Verified Status	Details
String[] org.apache.coyote.tomcat5.CoyoteRequest.getParameterValues(String)	splc.ItemService: 201	/splc/listMyItems.do	Under Review	✓	View

Source

SQL Injection : Detected a SQL Injection issue where external taint reached a database sink

URL: http://localhost/splc/listMyItems.do

Entry Point: Web Input

File:	org.apache.coyote.tomcat5.CoyoteRequestFacade:295
Method:	String[] org.apache.coyote.tomcat5.CoyoteRequest.getParameterValues(String)
Method Arguments:	• bean.quantity

Sink

End Point: Database

File:	com.order.splc.ItemService:201
Method:	ResultSet java.sql.Statement.executeQuery(String)
Trigger:	*Method Argument* Value: `select id, account, sku, quantity, price, ccno, description from`
Stack Trace:	
HTTP Request:	

Where is the Problem?

Severity	Category	URL
Critical	SQL Injection	/splc/listMyItems.do

Class	Line
com.order.splc.ItemService	196

Query	Stack Trace
select * from item where item name = ‘adam‘ and ...	java.lang.Throwable at StackTrace$FirstNested$SecondNested. <init>(StackTrace.java:267) at StackTrace$FirstNested. <init>(StackTrace.java:256) at StackTrace. <init>(StackTrace.java:246) at StackTrace. main(StackTrace.java:70)

Instrumentation

- Instrument JRE classes once
- Two ways to instrument program:
  - Compile-time
    - Rewrite the program's class files on disk
  - Runtime
    - Augment class loader to rewrite program

Aspect-Oriented Programming

- Express cross-cutting concerns independently from logic (aspects)
- Open source frameworks
  - AspectJ (Java)
  - AspectDNG (.NET)
- Could build home-brew instrumentation on top of bytecode library (BCEL, ASM)

Example

```
public aspect SQLInjectionCore extends ... {
 //Statement
 pointcut sqlInjectionStatement(String sql):
  (call(ResultSet Statement+.executeQuery(String))
  && args(sql))
  ...
}
```

Instrument Inside or Outside?

- Inside function body
  - Lower instrumentation cost
- Outside function call
  - Lower runtime cost / better reporting

Types of Taint

- Track distinct sources of untrusted input
  - Report XSS on data from the Web or database, but not from the file system
- Distinguish between different sources when reporting vulnerabilities
  - Prioritize remotely exploitable vulnerabilites

Java: Foundation – Round 2

- Add taint storage and source information to **`java.lang.String`** storage

Length	Taint	Body

↓

Length	Taint	Source	Body

Writing Rules

- Identifying the right methods is critical
  - Missing just one source or sink can be fatal
- Leverage experience from static analysis
  - Knowledge of security-relevant APIs

Going Wrong

SOURCES OF INACCURACY

Types of Inaccuracy

- False positives: erroneous bug reports
  - Painful for tool user
- False negatives: unreported bugs
  - Uh oh

False Positives: Unrecognized Input Validation

```
user = request.getParameter("user");
if (!InputUtil.alphaOnly(user)) {
  return false;
}
try {
  sql = "SELECT * FROM users " +
        "WHERE id='" + user + "'";
  stmt.executeQuery(sql);
}
```

False Positives: Impossible Ctl Flow Paths

- Paths that regular data can take that malicious data cannot take
- Solution: cleanse rules
  - Remove taint when String is input to a regular expression, compared to static string, etc

Countering False Positives: Bug Verification

- Training wheels for security testers
- Show which inputs to attack
- Suggest attack data
- Monitor call sites to determine if attack succeeds

False Negatives

- Taint can go where we cannot follow
  - String decomposition
  - Native code
  - Written to file or database and read back
- Bad cleanse rules
- Poor test coverage

False Negatives: String Decomposition

```
StringBuffer sb = new StringBuffer();
for (int i=0; i<tainted.length(); i++){
  sb.append(tainted.charAt(i));
}
String untainted = sb.toString();
return untainted;
```

False Negatives: Insufficient Input Validation

```
user = request.getParameter("user");
if (!InputUtil.alphaOnly(user)) {
  return false;
}
try {
  sql = "SELECT * FROM users " +
        "WHERE id='" + user + "'";
  stmt.executeQuery(sql);
}
```

False Negatives: Poor Test Coverage

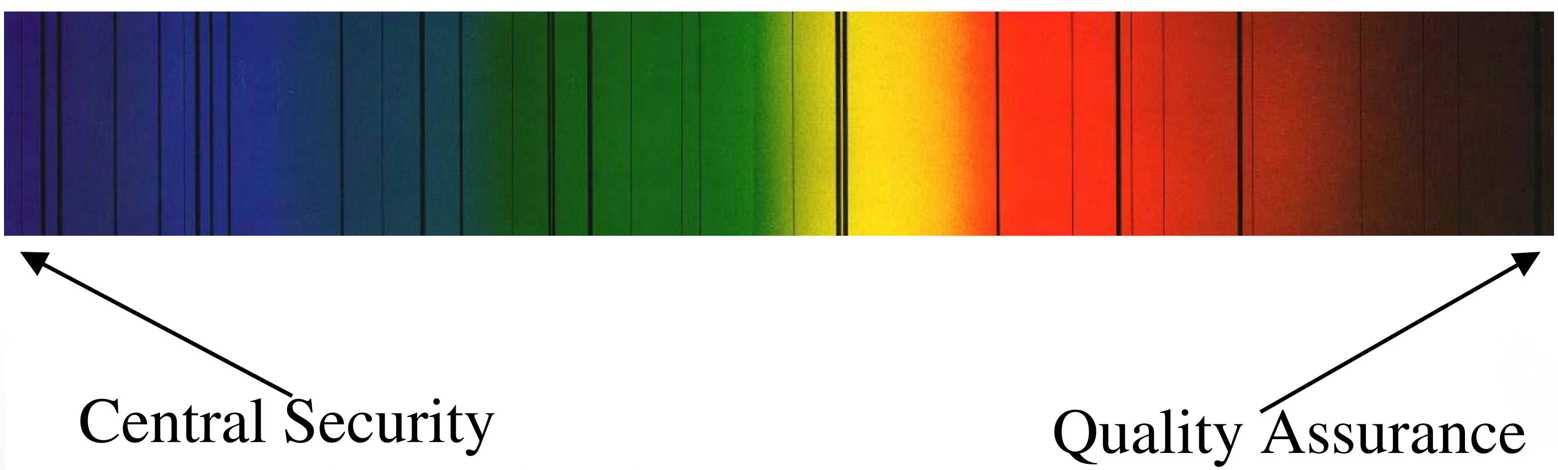
- Only looks at paths that are executed
- Bad QA Testing == Bad Security Testing

Practical Considerations

INTEGRATING WITH QA

In Practice

- Deployment may involve more or less involvement from central security team



Deployment Activities

Central Security		Quality Assurance
←	Instrumentation	→
	🔒	**Functional testing**
←	Triage and Verification	→
←	Reporting bugs	→

Instrumentation

- Either QA or Security
- Key considerations
  - Cover program behavior
  - Cover security threats

Functional Testing

- QA
- Key considerations
  - Maximize coverage (existing goal)
  - Security knowledge not required

Triage and Verification

- Either QA or Security
- Key considerations
  - Understand issues in program context
  - Security knowledge
    - Hand-holding to create "exploits"
    - Different bugs to different auditors
    - Targeted training

Reporting Bugs

- Either QA or Security
- Key considerations
  - Bug reporting conventions / protocols
  - Solid remediation advice

Other people's business

RELATED WORK

Related Work

- Perl
- Taint propagation for Java
- Constraint propagation for C
- Fine-grained taint propagation for C
- Taint propagation for PHP

Perl

```
#!/usr/bin/perl -T
my $arg=shift;
system($arg);

 > Insecure $ENV{PATH }
```

Perl

```
#!/usr/bin/perl -T
my $arg=shift;
$ENV{PATH} = "/bin";
system($arg);
```

```
 > Insecure dependency in system
   while running with -T switch
```

Perl

- Automatically removes taint when string is used in regex
- Meant for active defense, not bug finding, so error messages are less than ideal

Taint Propagation for Java

- Haldar, Chandra, Franz (UC Irvine) ACSAC ‘05
- Taints Java String objects
- Active protection, not bug detection
- Notion of taint flags, but no impl

Constraint Propagation for C

- Larsen and Austin (U Michigan) USENIX ‘03
- Keep track of symbolic constraints on input while program is running
- Spot bugs where input is under-constrained
- Found multiple bugs in OpenSSH

Constraint Propagation for C

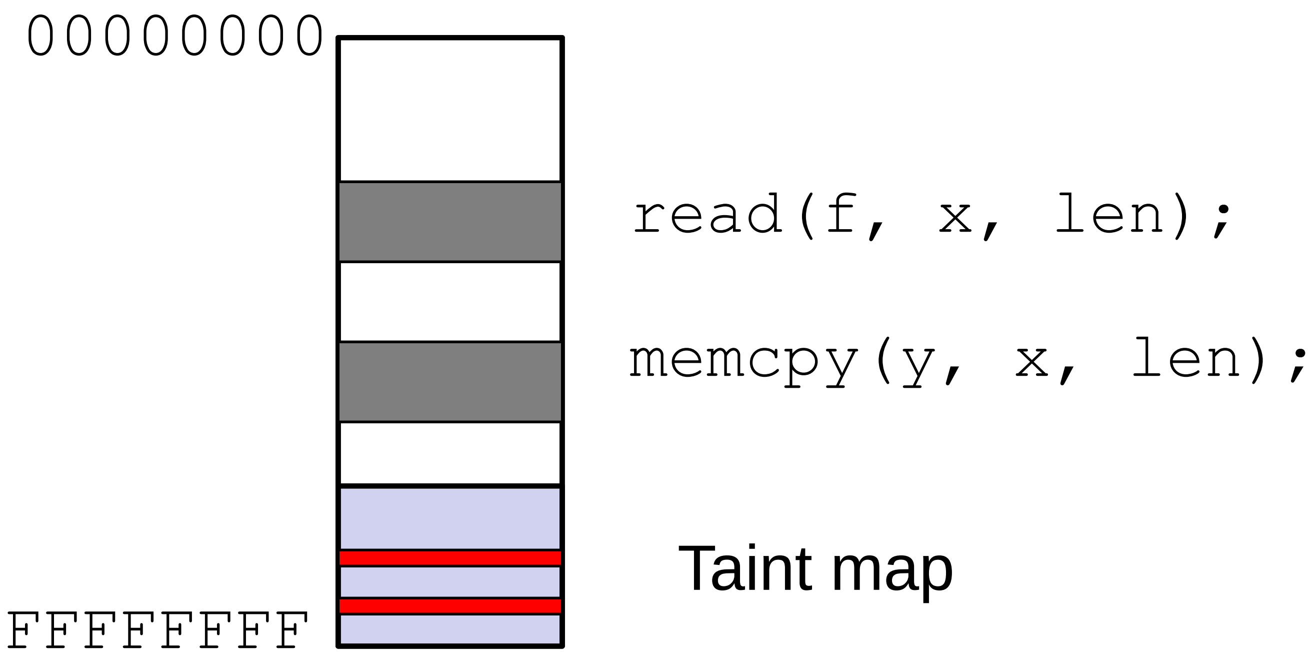
Code	Concrete Execution	Symbolic Execution
`unsigned int x;`		
`int array[5];`		
`scanf("%d", &x);`	$x = 2$	$0 \leq x \leq \infty$
`if (x > 4) die();`	$x = 2$	$0 \leq x \leq 4$
`x++;`	$x = 3$	$0 \leq x \leq 5$
`array[x]= 0;`	OK	ERROR!

Fine-grained Taint Propagation

- Xu, Bhatkar, Sekar (Stony Brook), USENIX ‘06
- Keep explicit taint state for every byte in the program
- Requires large chunk of program address space
- Clever optimizations make performance penalty bearable in many cases

Fine-grained Taint Propagation

Program address space



Fine-grained Taint Propagation

- Can detect most injection attacks
  - Buffer overflow, format string attacks, SQL injection, command injection
- Works for interpreted languages with native interpreters (PHP).

PHP

- Easier to do fine-grained analysis
  - all program data represented with native data structures
- Augment interpreter to propagate taint
- Small performance penalty
- Core GRASP
- Our vote: build it into the std interpreter

Static Analysis (YALASA)

- Advantage
  - can simulate execution of all possible paths
- Disadvantage
  - necessarily less precise
  - does not know which paths are likely and which are unlikely

SUMMARY

Conclusions

- Security is coming to QA!
- Lessons from security in development
  - Target process steps at strengths
  - Designs tools for the right audience
  - Use targeted training to bolster capabilities

Questions?

Brian Chess
brian@fortify.com

Jacob West
jacob@fortify.com