Security Anomalies and Continuous Vulnerability Detection

Luciano Sampaio - lsampaio@inf.puc-rio.br
Alessandro Garcia - afgarcia@inf.puc-rio.br

LES | DI |PUC-Rio - Brazil

OPUS Research Group

Introduction

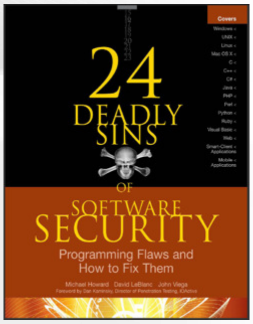
- Secure programming is the practice of writing programs that are resistant to attacks by malicious people or programs

✓ Check for SQL Injection. ✓ Check for Cookie Poisoning.
✗ Check for XSS. ✓ …

- Security vulnerability or just vulnerability is a flaw that can be exploited to allow an attacker to cause unintended operations
- 86% of audited websites has at least 1 serious security vulnerability (WhiteHat - 2013)

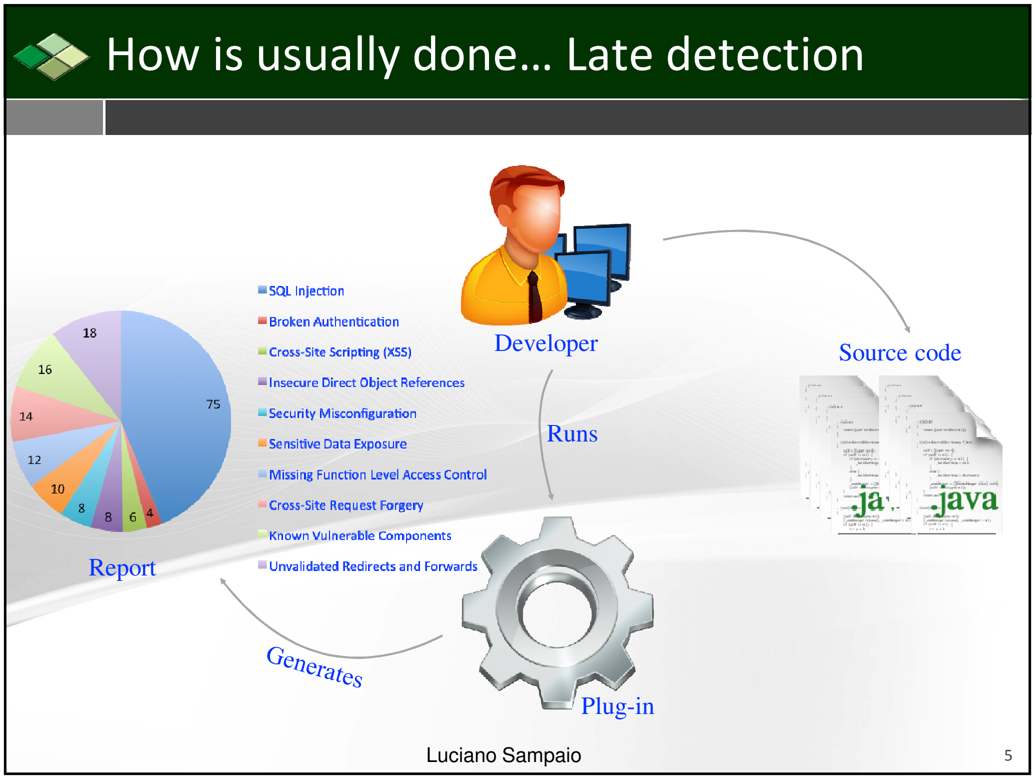
Top 10 vulnerabilities

- Open Web Application Security Project (OWASP)
  - 2003, 2004, 2007, 2010, 2013
- OWASP Top 10 – 2013
  - From 8 datasets from 7 companies with over 500,000 vulnerabilities
  - 01 – (SQL/Command) Injection
  - 02 – Broken Authentication and Session Management
  - 03 – Cross-Site Scripting (XSS)
  - 04 – Insecure Direct Object References
  - 05 – Security Misconfiguration
  - 06 – Sensitive Data Exposure
  - 07 – Missing Function Level Access Control
  - 08 – Cross-Site Request Forgery (CSRF)
  - 09 – Using Known Vulnerable Components
  - 10 – Unvalidated Redirects and Forwards



Static analysis can help

- SSVChecker, FindBug, ASIDE, Lapse+, CodePro Analytics, CodeProfiler, JeSS and AppScan IBM
- Key characteristics
  - Late detection
  - Pattern matching



How is usually done… Late detection

Developer

Source code

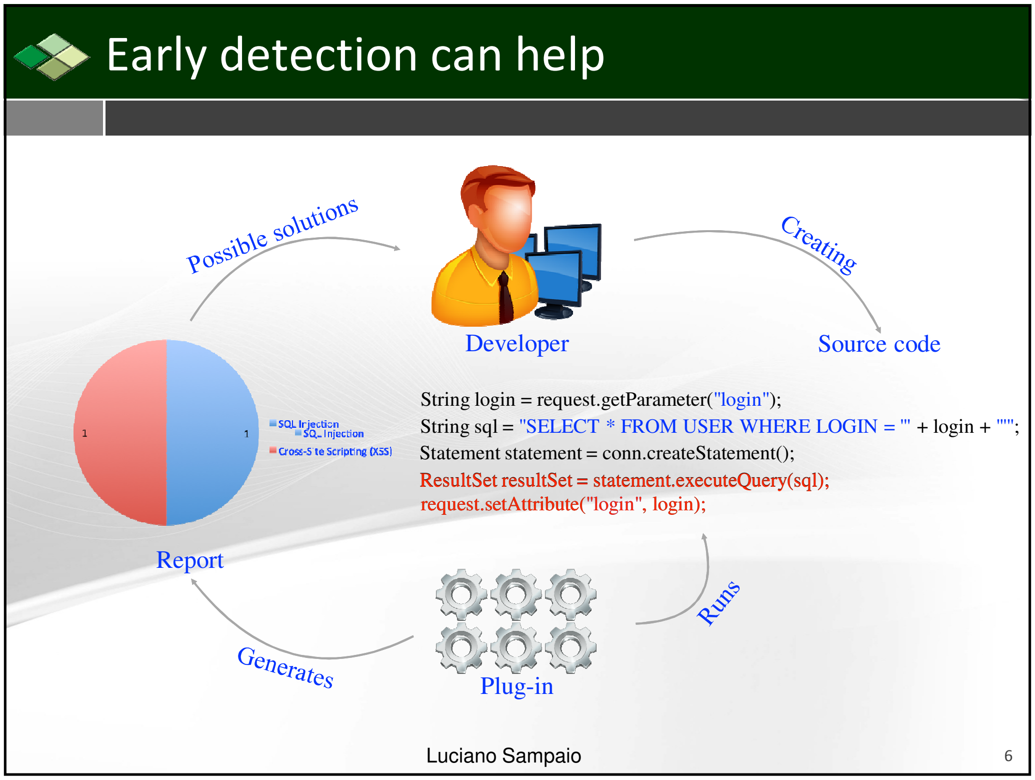
Runs

Plug-in

Generates

Report

75
18
16
14
12
10
8
8
6
4
🟦 SQL Injection
🟥 Broken Authentication
🟩 Cross-Site Scripting (XSS)
🟪 Insecure Direct Object References
🟦 Security Misconfiguration
🟧 Sensitive Data Exposure
🟦 Missing Function Level Access Control
🟥 Cross-Site Request Forgery
🟩 Known Vulnerable Components
🟪 Unvalidated Redirects and Forwards
java
java



Early detection can help

Possible solutions

Developer

Creating

Source code

```
String login = request.getParameter("login");
String sql = "SELECT * FROM USER WHERE LOGIN = '" + login + "'";
Statement statement = conn.createStatement();
ResultSet resultSet = statement.executeQuery(sql);
request.setAttribute("login", login);
```

Runs

Plug-in

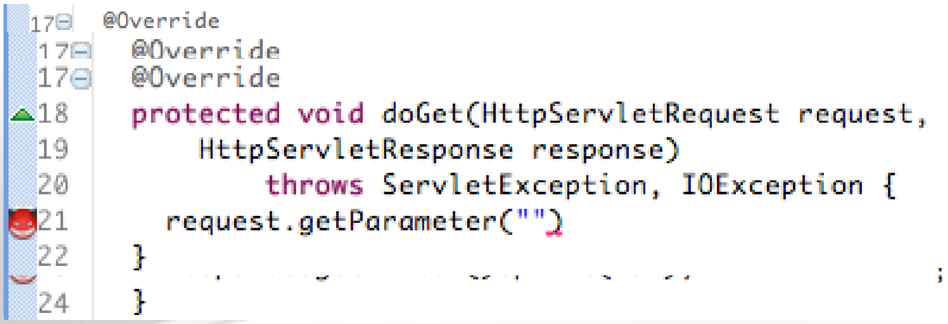
Generates

Report

Early detection requirements

- High accurate detection technique
  - Rate of false positives
- The usage of time and resources can not disturb the developer

Early detection using low accurate technique

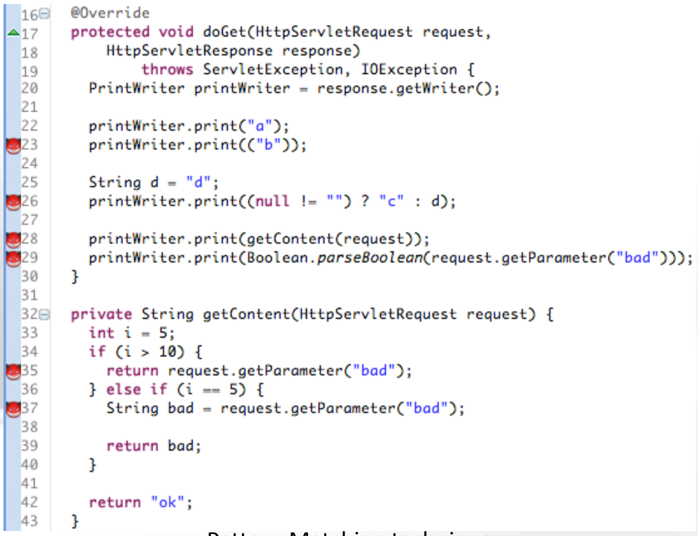


```
@Override
  @Override
  @Override
  protected void doGet(HttpServletRequest request,
      HttpServletResponse response)
          throws ServletException, IOException {
    request.getParameter("")
  }
  }
```

Frequently used technique

- Pattern matching is a technique for checking if a pattern matches a given sequence of tokens (letters, numbers, punctuation, and certain symbols)
  - 20/30% of false positives - (Nadeem 2012)

Pattern matching example



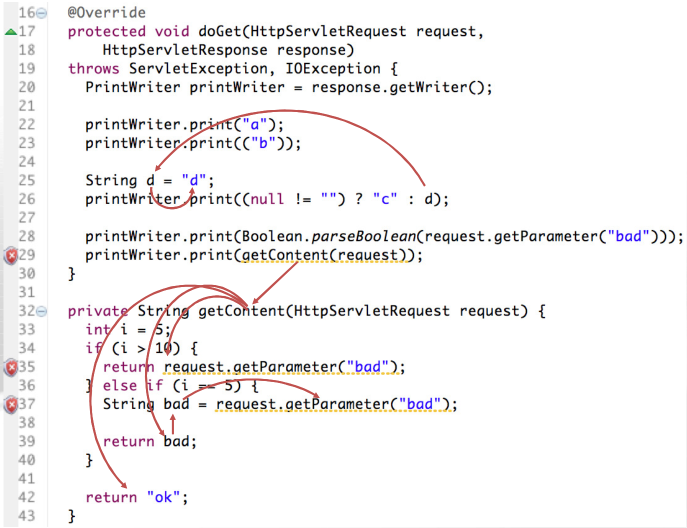
```
@Override
protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
        throws ServletException, IOException {
  PrintWriter printWriter = response.getWriter();

  printWriter.print("a");
  printWriter.print(("b"));

  String d = "d";
  printWriter.print((null != "") ? "c" : d);

  printWriter.print(getContent(request));
  printWriter.print(Boolean.parseBoolean(request.getParameter("bad")));
}

private String getContent(HttpServletRequest request) {
  int i = 5;
  if (i > 10) {
    return request.getParameter("bad");
  } else if (i == 5) {
    String bad = request.getParameter("bad");

    return bad;
  }

  return "ok";
}
```

Pattern Matching technique

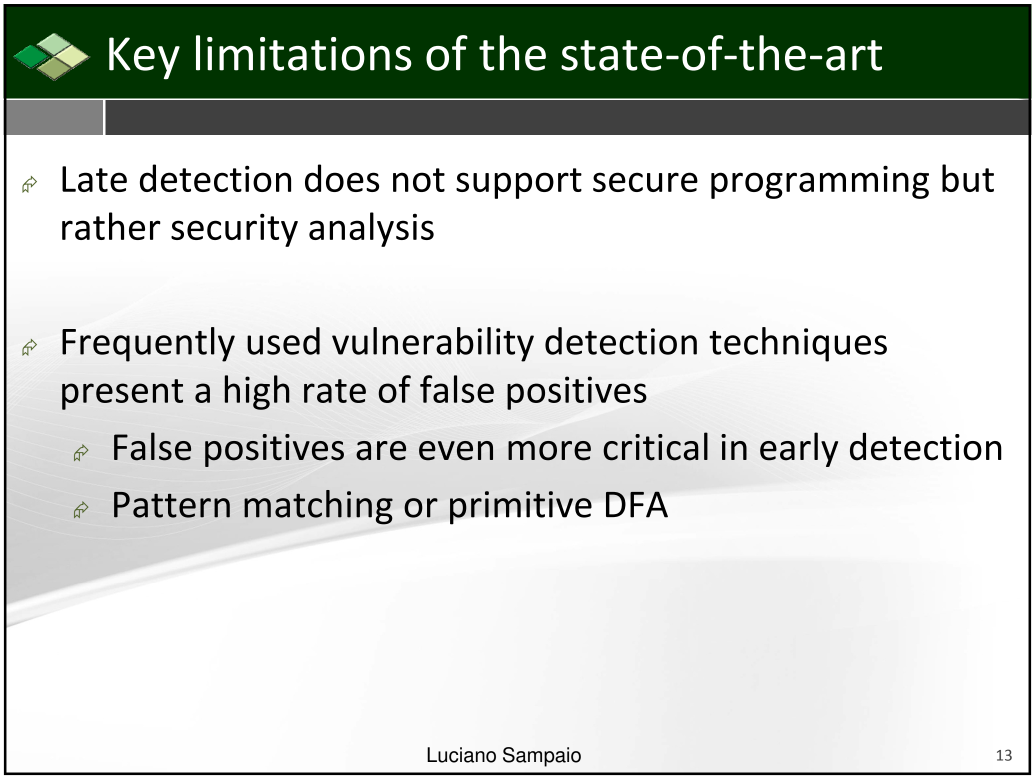
Data-flow analysis as an alternative

- Data-flow analysis (DFA) is a technique for gathering information about the possible set of values calculated at various points in a computer program
- Originally created and commonly used for implementing optimizations on compilers
- Currently, there is only one solution that uses DFA to detect security vulnerabilities (CodePro)

Data-flow analysis example

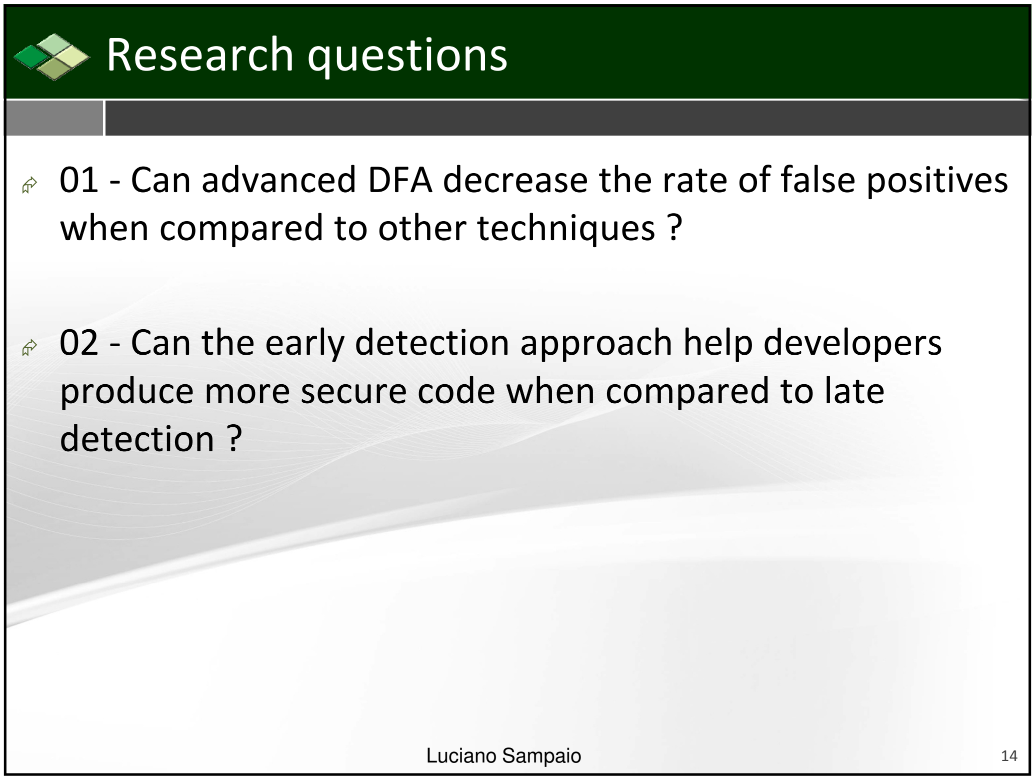


```
@Override
protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
throws ServletException, IOException {
  PrintWriter printWriter = response.getWriter();

  printWriter.print("a");
  printWriter.print(("b"));

  String d = "d";
  printWriter.print((null != "") ? "c" : d);

  printWriter.print(Boolean.parseBoolean(request.getParameter("bad")));
  printWriter.print(getContent(request));
}

private String getContent(HttpServletRequest request) {
  int i = 5;
  if (i > 10) {
    return request.getParameter("bad");
  } else if (i == 5) {
    String bad = request.getParameter("bad");

    return bad;
  }

  return "ok";
}
```



Key limitations of the state-of-the-art

- Late detection does not support secure programming but rather security analysis
- Frequently used vulnerability detection techniques present a high rate of false positives
  - False positives are even more critical in early detection
  - Pattern matching or primitive DFA



Research questions

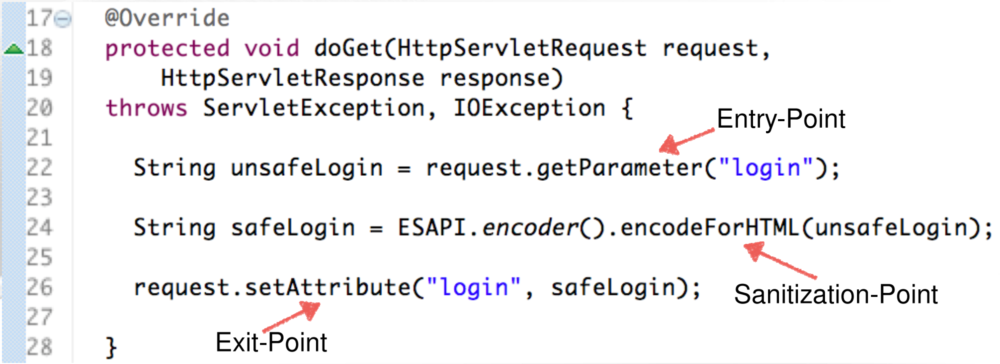
- 01 - Can advanced DFA decrease the rate of false positives when compared to other techniques ?
- 02 - Can the early detection approach help developers produce more secure code when compared to late detection ?

Proposed solutions

- Propose to support a change from the default behavior of late detection to early detection
- Propose new heuristics using a technique named context-sensitive data flow analysis
  - Pattern matching
  - Context-insensitive data flow analysis
- Designed and implemented a prototype
- Performed 2 empirical studies

How to identify something as unsafe/safe ?

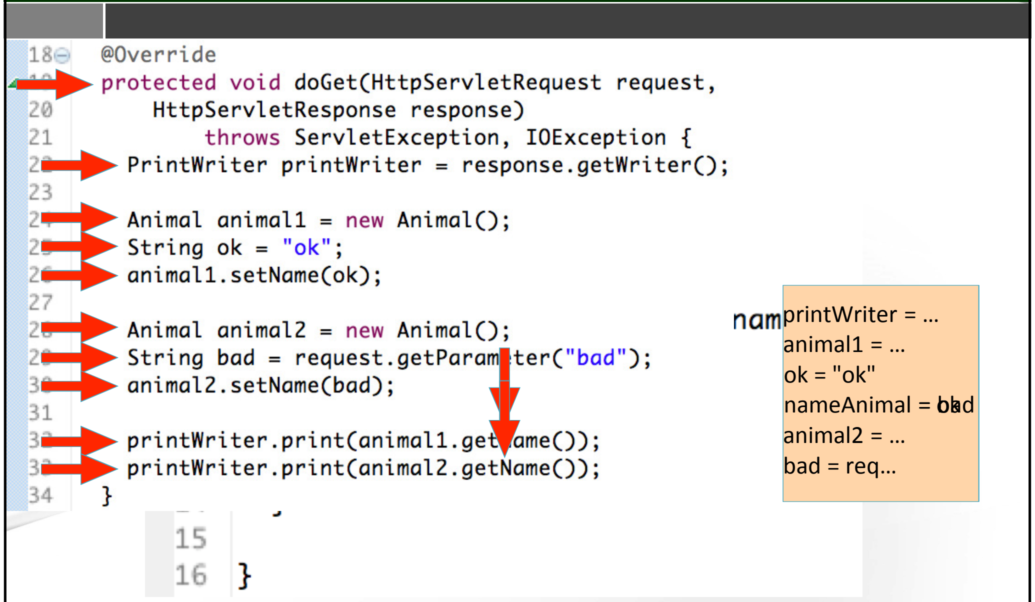
- Entry-Point
- Sanitization-Point
- Exit-Point



```
@Override
protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
throws ServletException, IOException {

  String unsafeLogin = request.getParameter("login");

  String safeLogin = ESAPI.encoder().encodeForHTML(unsafeLogin);

  request.setAttribute("login", safeLogin);

}
```

Data Flow Analysis >>

Context Insensitive



```
@Override
protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
            throws ServletException, IOException {
    PrintWriter printWriter = response.getWriter();

    Animal animal1 = new Animal();
    String ok = "ok";
    animal1.setName(ok);

    Animal animal2 = new Animal();
    String bad = request.getParameter("bad");
    animal2.setName(bad);

    printWriter.print(animal1.getName());
    printWriter.print(animal2.getName());
}

}
```

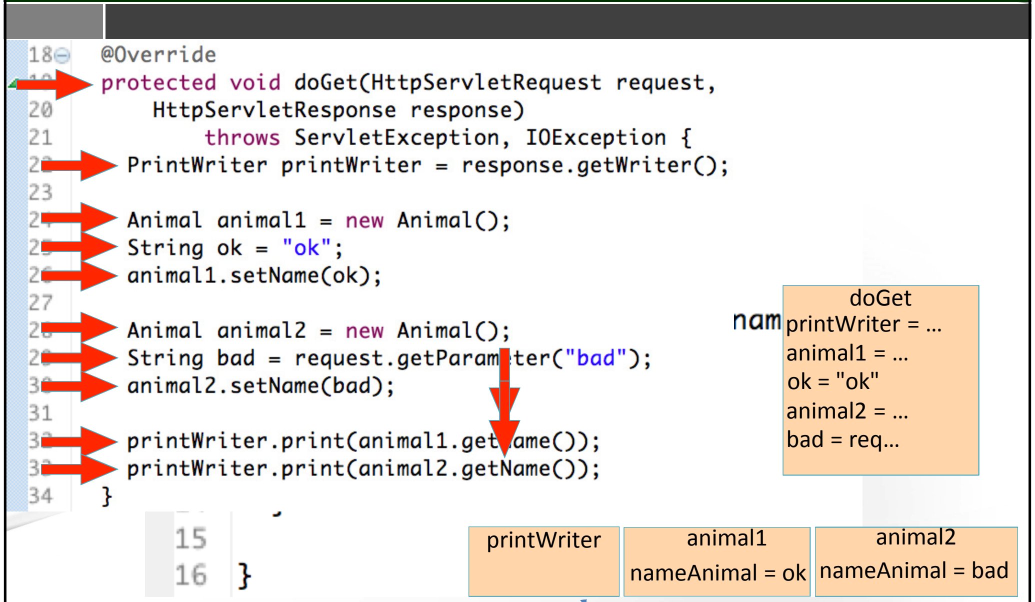
nam

printWriter = ...
animal1 = ...
ok = "ok"
nameAnimal = bad
animal2 = ...
bad = req...

Luciano Sampaio

Data Flow Analysis >>

Context Sensitive



```
@Override
protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
            throws ServletException, IOException {
    PrintWriter printWriter = response.getWriter();

    Animal animal1 = new Animal();
    String ok = "ok";
    animal1.setName(ok);

    Animal animal2 = new Animal();
    String bad = request.getParameter("bad");
    animal2.setName(bad);

    printWriter.print(animal1.getName());
    printWriter.print(animal2.getName());
}

}
```

nam

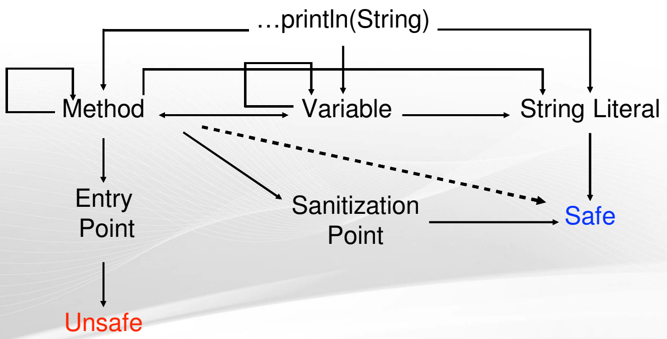
doGet
printWriter = ...
animal1 = ...
ok = "ok"
animal2 = ...
bad = req...

printWriter	animal1 nameAnimal = ok	animal2 nameAnimal = bad

Contexts

Luciano Sampaio

Our proposed heuristics


…println(String)

Method

Variable

String Literal

Entry Point

Sanitization Point

Safe

Unsafe

```
String a = request.getParameter("a");
System.out.println(a);
String a = sanitize(request.getParameter("a"));
System.out.println(a);
System.out.println("abcd");
String a = "abcd";
System.out.println(a);
```

Data Flow Analysis >>

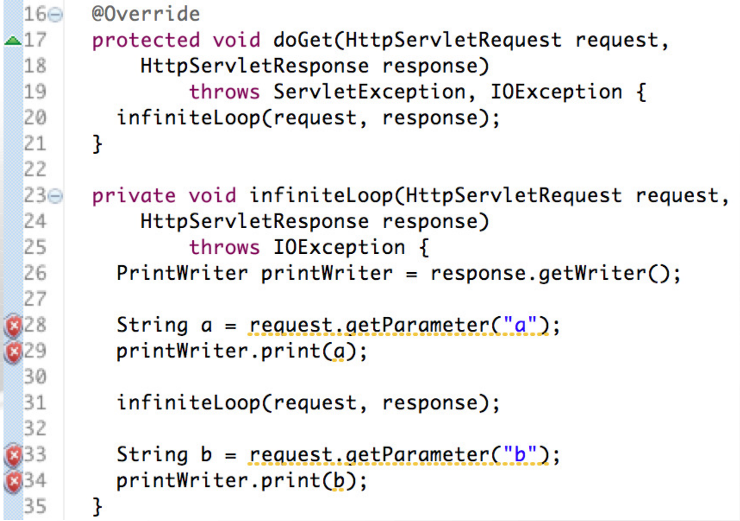
Supported vulnerabilities

- Vulnerabilities that stem from program input and output not being properly validated are recognized as being the most common ones
- These vulnerabilities are not dependent on how they are implemented

- 01 - Command Injection
- 02 - Cookie Poisoning
- 03 - Cross-Site Scripting (XSS)
- 04 - HTTP Response Splitting
- 05 - LDAP Injection
- 06 - Log Forging
- 07 - Path Traversal
- 08 - Reflection Injection
- 09 - Security Misconfiguration
- 10 - SQL Injection
- 11 - XPath Injection

Data Flow Analysis >>

New type of problems - Infinite Loop

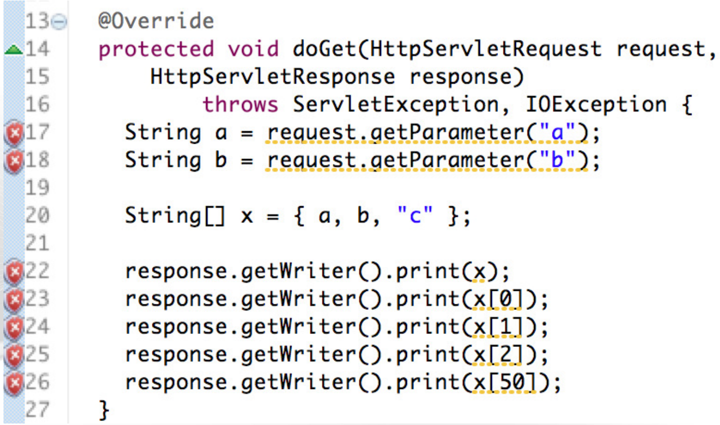


```
@Override
protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
        throws ServletException, IOException {
  infiniteLoop(request, response);
}

private void infiniteLoop(HttpServletRequest request,
    HttpServletResponse response)
        throws IOException {
  PrintWriter printWriter = response.getWriter();

  String a = request.getParameter("a");
  printWriter.print(a);

  infiniteLoop(request, response);

  String b = request.getParameter("b");
  printWriter.print(b);
}
```

Data Flow Analysis >>

Current limitations of our implementation

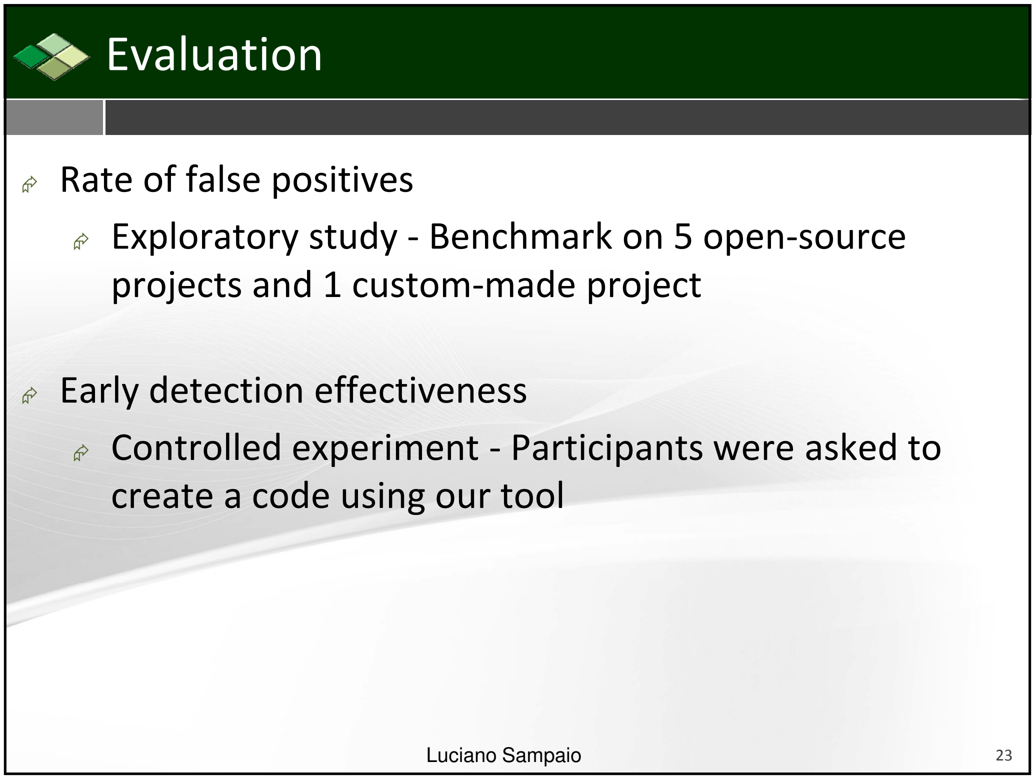
- Containers
- InnerClasses



```
@Override
protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
        throws ServletException, IOException {
  String a = request.getParameter("a");
  String b = request.getParameter("b");

  String[] x = { a, b, "c" };

  response.getWriter().print(x);
  response.getWriter().print(x[0]);
  response.getWriter().print(x[1]);
  response.getWriter().print(x[2]);
  response.getWriter().print(x[50]);
}
```

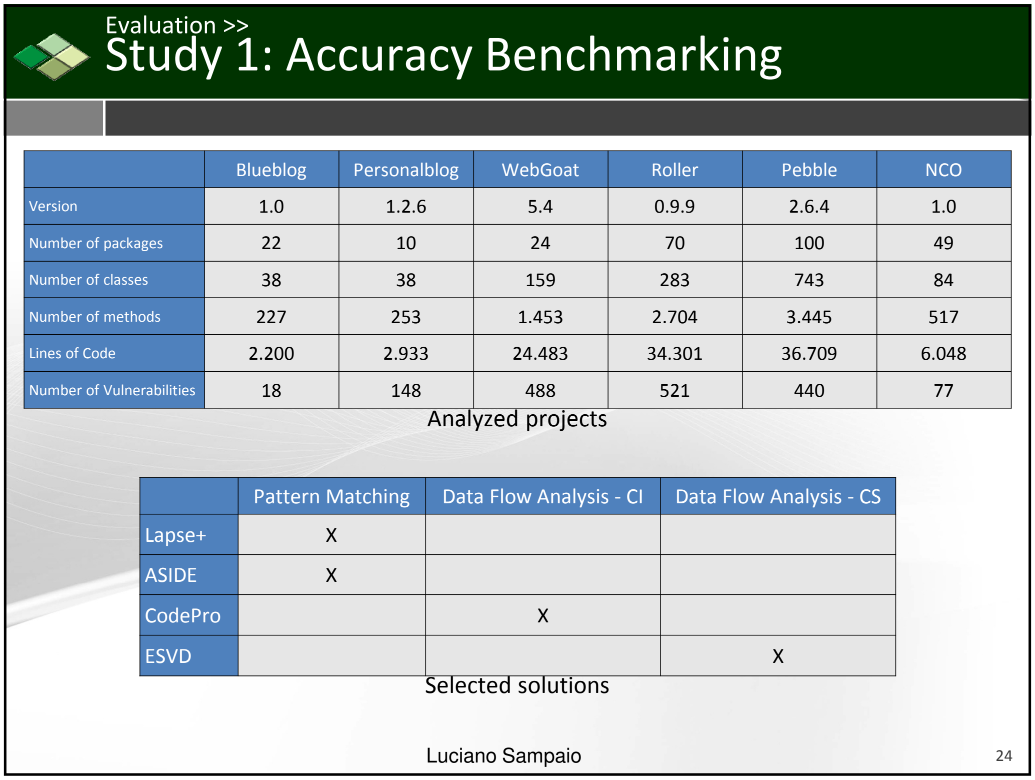
ESVD

Dillig, I., Dillig, T. and Aiken, A. (2011). Precise reasoning for programs using containers. ACM SIGPLAN Notices



Evaluation

- Rate of false positives
  - Exploratory study - Benchmark on 5 open-source projects and 1 custom-made project
- Early detection effectiveness
  - Controlled experiment - Participants were asked to create a code using our tool



Evaluation >>

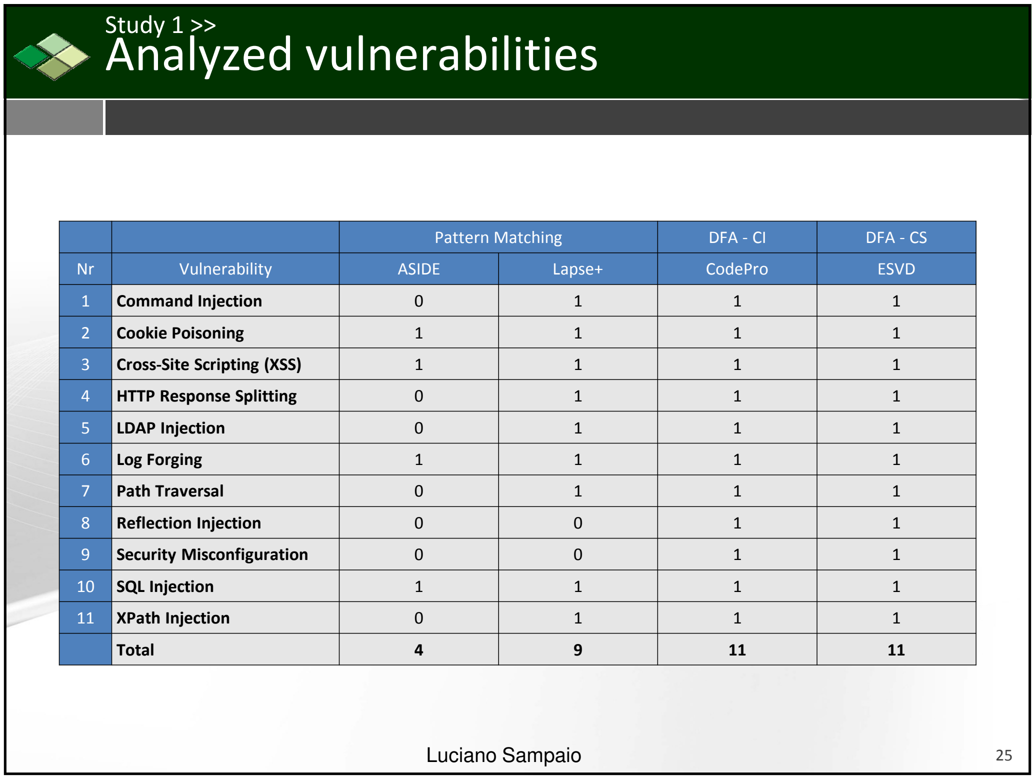
Study 1: Accuracy Benchmarking

	Blueblog	Personalblog	WebGoat	Roller	Pebble	NCO
Version	1.0	1.2.6	5.4	0.9.9	2.6.4	1.0
Number of packages	22	10	24	70	100	49
Number of classes	38	38	159	283	743	84
Number of methods	227	253	1.453	2.704	3.445	517
Lines of Code	2.200	2.933	24.483	34.301	36.709	6.048
Number of Vulnerabilities	18	148	488	521	440	77

Analyzed projects

	Pattern Matching	Data Flow Analysis - CI	Data Flow Analysis - CS
Lapse+	X		
ASIDE	X		
CodePro		X	
ESVD			X

Selected solutions



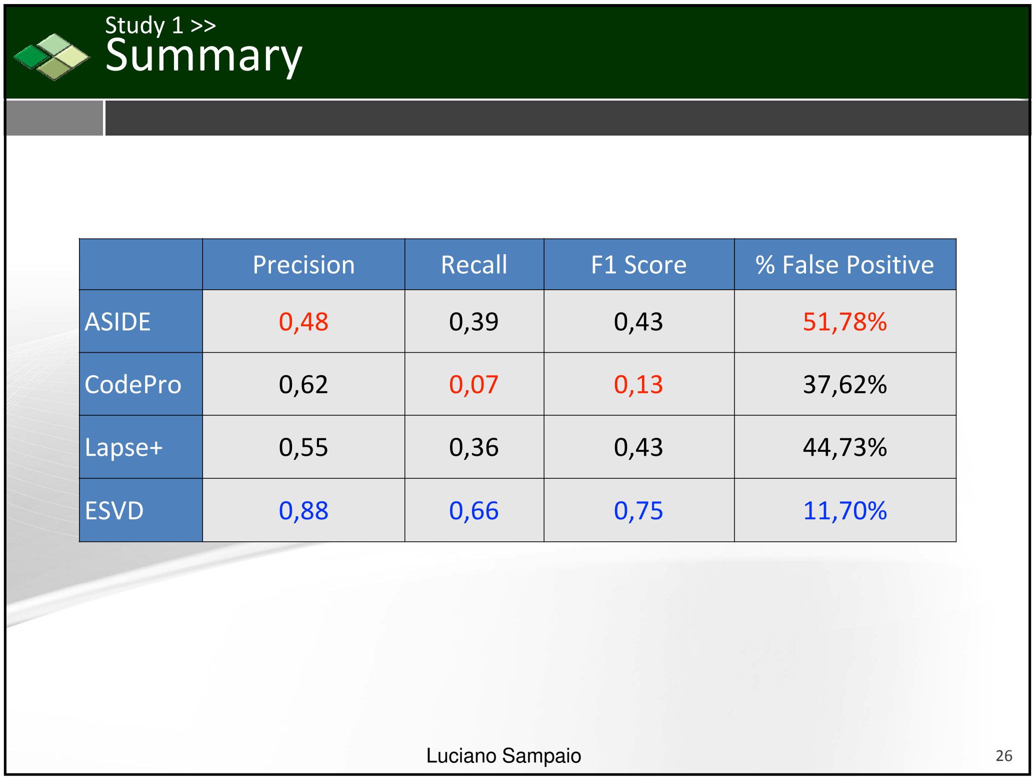
Study 1 >>

Analyzed vulnerabilities

		Pattern Matching		DFA - CI	DFA - CS
Nr	Vulnerability	ASIDE	Lapse+	CodePro	ESVD
1	**Command Injection**	0	1	1	1
2	**Cookie Poisoning**	1	1	1	1
3	**Cross-Site Scripting (XSS)**	1	1	1	1
4	**HTTP Response Splitting**	0	1	1	1
5	**LDAP Injection**	0	1	1	1
6	**Log Forging**	1	1	1	1
7	**Path Traversal**	0	1	1	1
8	**Reflection Injection**	0	0	1	1
9	**Security Misconfiguration**	0	0	1	1
10	**SQL Injection**	1	1	1	1
11	**XPath Injection**	0	1	1	1
	Total	**4**	**9**	**11**	**11**

Luciano Sampaio

25



Study 1 >>

Summary

	Precision	Recall	F1 Score	% False Positive
ASIDE	0,48	0,39	0,43	51,78%
CodePro	0,62	0,07	0,13	37,62%
Lapse+	0,55	0,36	0,43	44,73%
ESVD	0,88	0,66	0,75	11,70%

Luciano Sampaio

26

Study 1 >>

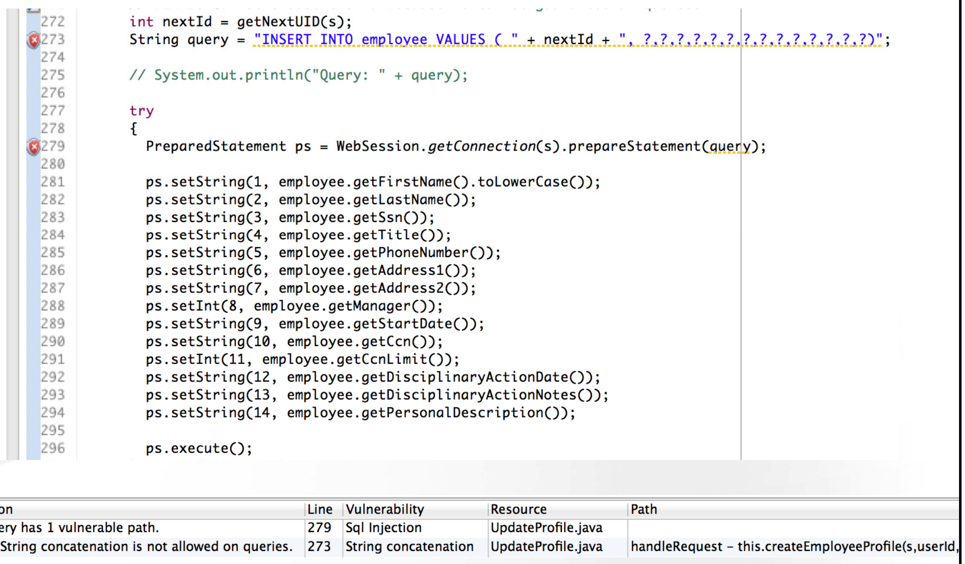
Summary of False Positives

	Blueblog	Personalblog	WebGoat	Roller	Pebble	NCO
ASIDE	74%	13%	51%	70%	50%	29%
Lapse+	20%	25%	50%	22%	29%	64%
CodePro	59%	17%	32%	59%	54%	61%
ESVD	0%	3%	21%	1%	5%	61%

Results of false positives per analyzed project

Study 1 >>

WebGoat - False Positive

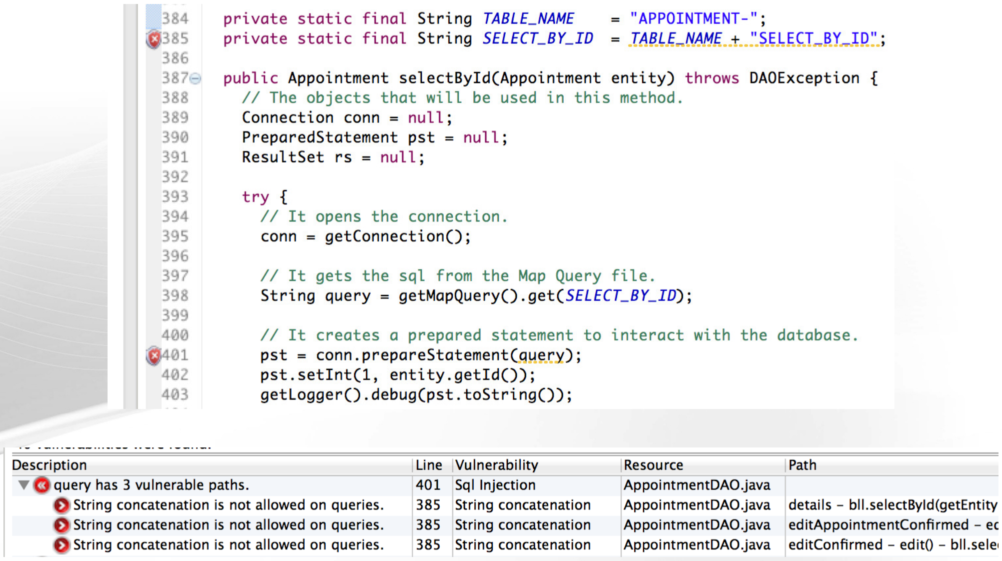


```
272      int nextId = getNextUID(s);
273      String query = "INSERT INTO employee VALUES ( " + nextId + ", ?,?,?,?,?,?,?,?,?,?,?,?,?,?)";
274
275      // System.out.println("Query: " + query);
276
277      try
278      {
279        PreparedStatement ps = WebSession.getConnection(s).prepareStatement(query);
280
281        ps.setString(1, employee.getFirstName().toLowerCase());
282        ps.setString(2, employee.getLastName());
283        ps.setString(3, employee.getSsn());
284        ps.setString(4, employee.getTitle());
285        ps.setString(5, employee.getPhoneNumber());
286        ps.setString(6, employee.getAddress1());
287        ps.setString(7, employee.getAddress2());
288        ps.setInt(8, employee.getManager());
289        ps.setString(9, employee.getStartDate());
290        ps.setString(10, employee.getCcn());
291        ps.setInt(11, employee.getCcnLimit());
292        ps.setString(12, employee.getDisciplinaryActionDate());
293        ps.setString(13, employee.getDisciplinaryActionNotes());
294        ps.setString(14, employee.getPersonalDescription());
295
296        ps.execute();
```

Description	Line	Vulnerability	Resource	Path
▼ query has 1 vulnerable path.	279	Sql Injection	UpdateProfile.java	
String concatenation is not allowed on queries.	273	String concatenation	UpdateProfile.java	handleRequest – this.createEmployeeProfile(s,userId,

Study 1 >>

NCO - False Positive

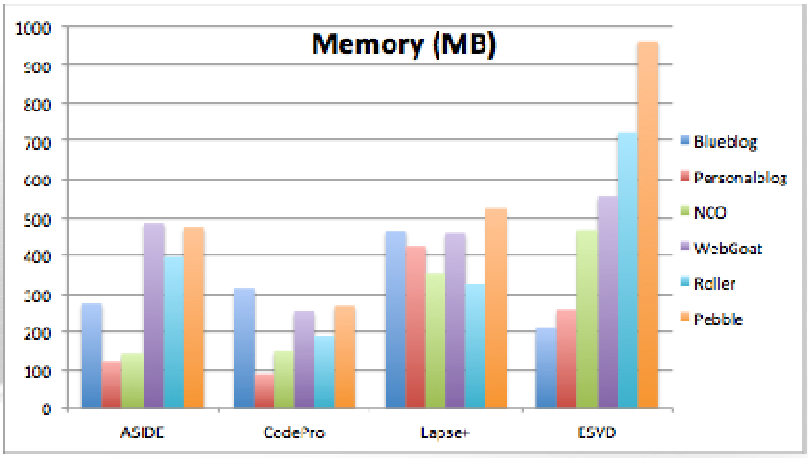


```
384  private static final String TABLE_NAME    = "APPOINTMENT-";
385  private static final String SELECT_BY_ID  = TABLE_NAME + "SELECT_BY_ID";
386
387  public Appointment selectById(Appointment entity) throws DAOException {
388    // The objects that will be used in this method.
389    Connection conn = null;
390    PreparedStatement pst = null;
391    ResultSet rs = null;
392
393    try {
394      // It opens the connection.
395      conn = getConnection();
396
397      // It gets the sql from the Map Query file.
398      String query = getMapQuery().get(SELECT_BY_ID);
399
400      // It creates a prepared statement to interact with the database.
401      pst = conn.prepareStatement(query);
402      pst.setInt(1, entity.getId());
403      getLogger().debug(pst.toString());
```

Description	Line	Vulnerability	Resource	Path
query has 3 vulnerable paths.	401	Sql Injection	AppointmentDAO.java	
String concatenation is not allowed on queries.	385	String concatenation	AppointmentDAO.java	details – bll.selectById(getEntity
String concatenation is not allowed on queries.	385	String concatenation	AppointmentDAO.java	editAppointmentConfirmed – ec
String concatenation is not allowed on queries.	385	String concatenation	AppointmentDAO.java	editConfirmed – edit() – bll.sele

Study 1 >>

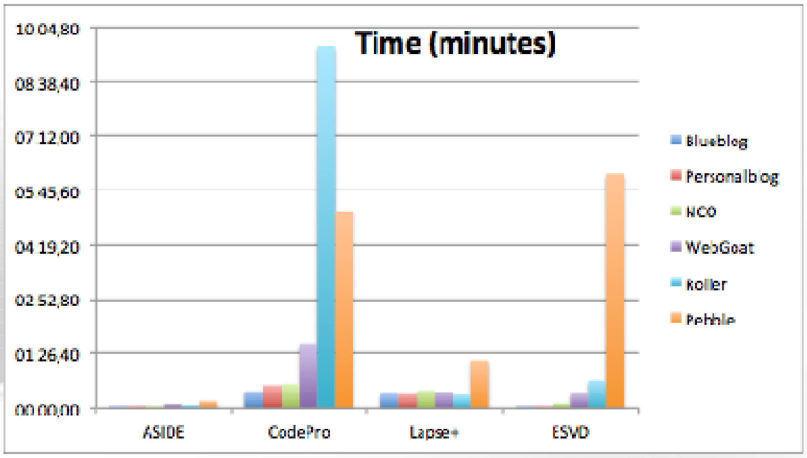
Memory



The applications are ordered by size

Study 1 >>

Time (Minutes)

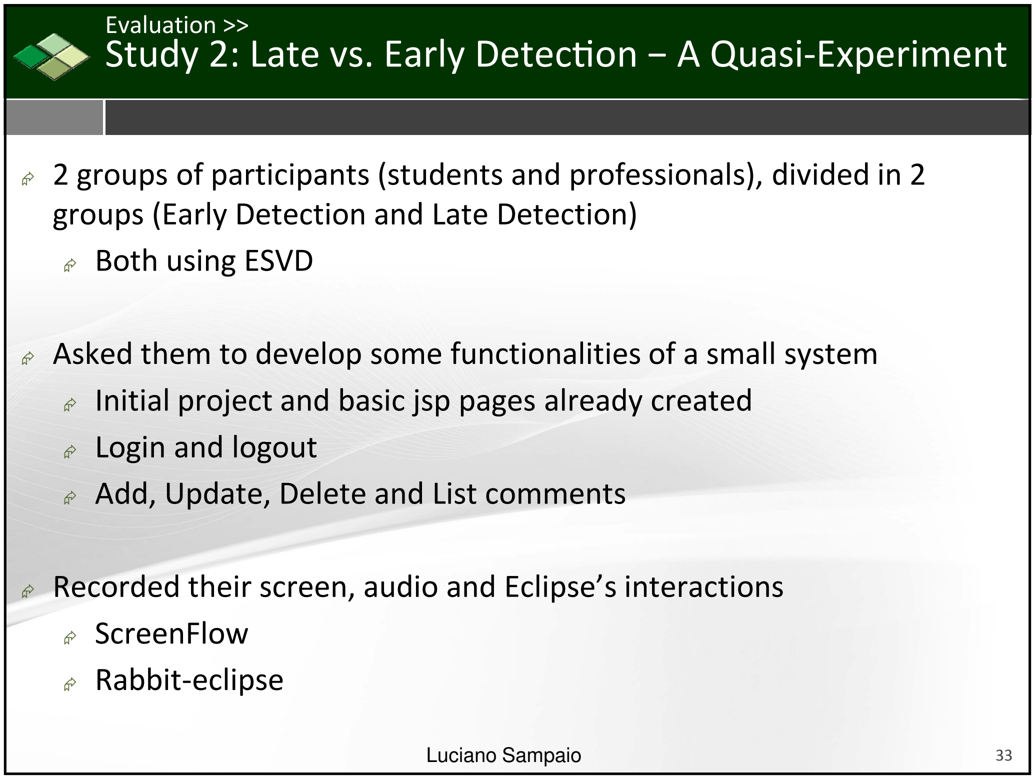


The applications are ordered by size

Study 1 >>

Findings

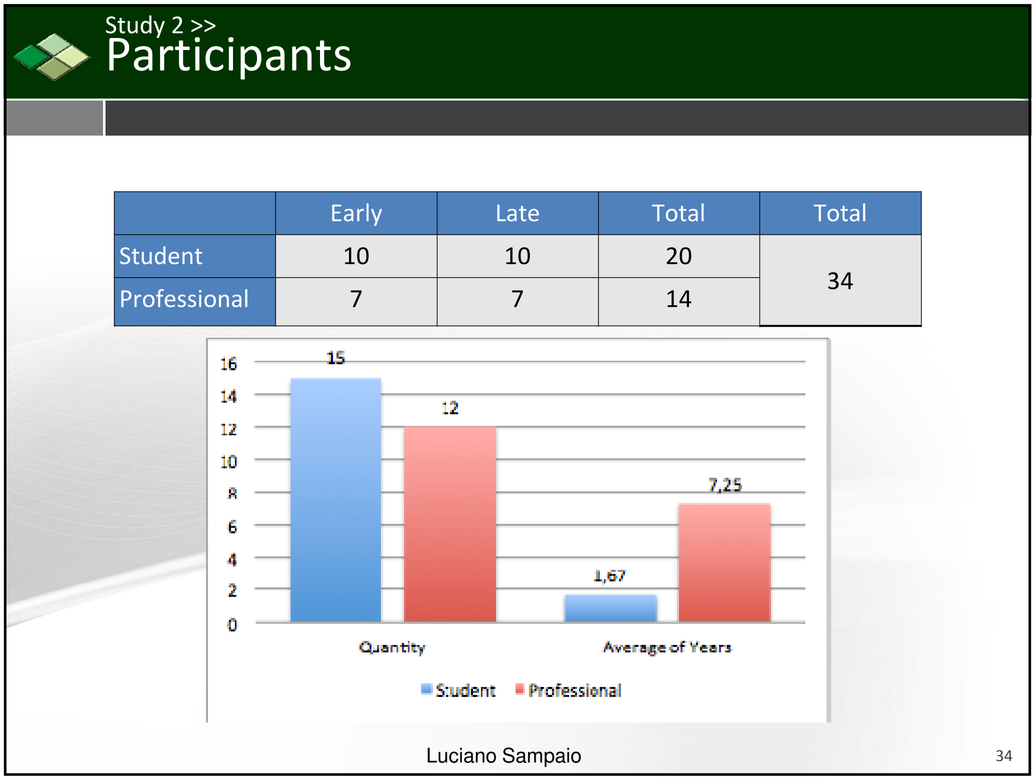
- We achieved 11,70% of rate of false positives
  - The best pattern-matching result was 44,73%
- There is a trade-off, better results mean more time and memory usage
  - This can be problem for large projects when using DFA-CS
- RQ1 - Can DFA-CS decrease the rate of false positives when compared to other techniques ? YES!



Evaluation >>

Study 2: Late vs. Early Detection – A Quasi-Experiment

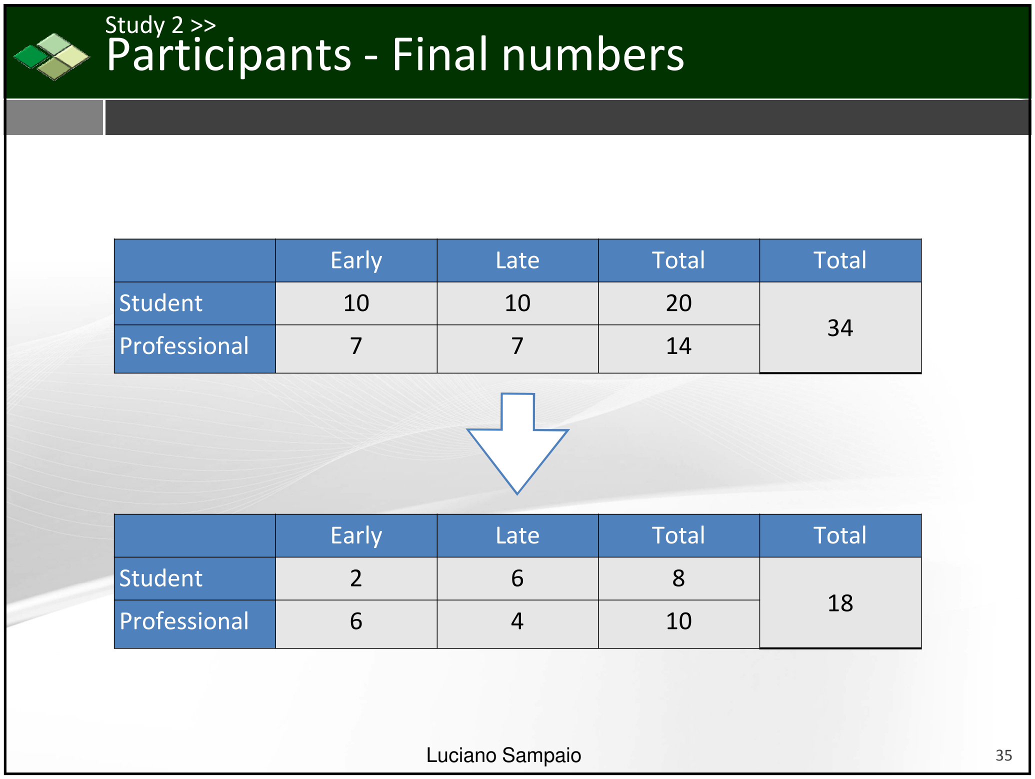
- 2 groups of participants (students and professionals), divided in 2 groups (Early Detection and Late Detection)
  - Both using ESVD
- Asked them to develop some functionalities of a small system
  - Initial project and basic jsp pages already created
  - Login and logout
  - Add, Update, Delete and List comments
- Recorded their screen, audio and Eclipse's interactions
  - ScreenFlow
  - Rabbit-eclipse



Study 2 >>

Participants

	Early	Late	Total	Total
Student	10	10	20	34
Professional	7	7	14	



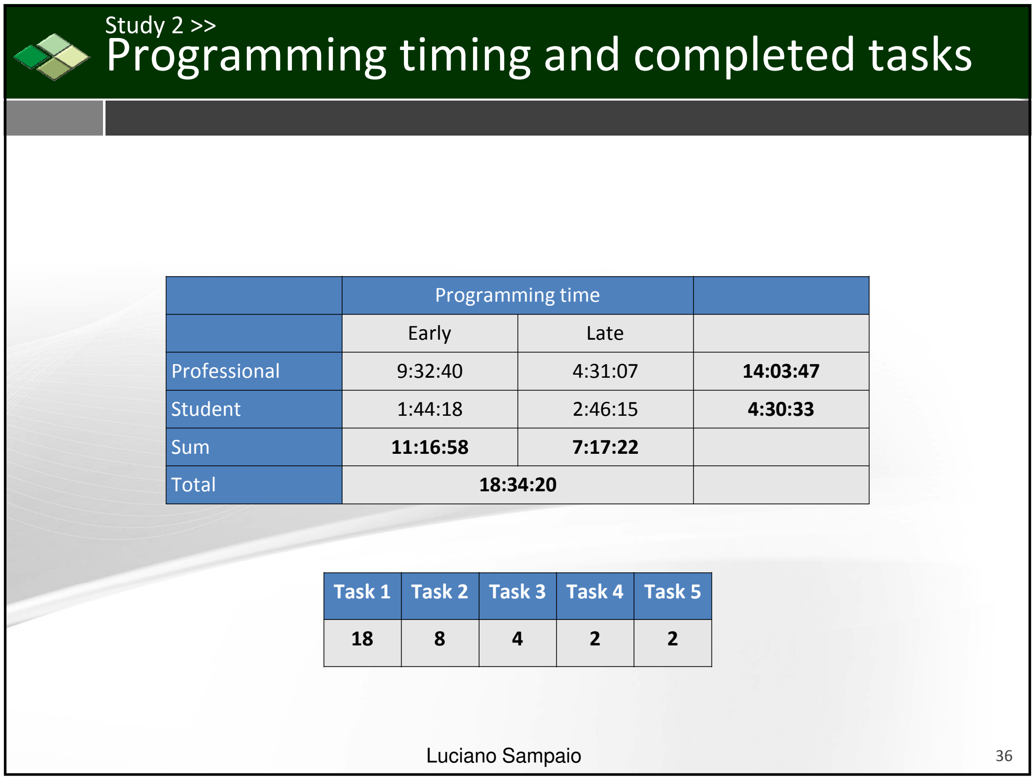
Study 2 >>

Participants - Final numbers

	Early	Late	Total	Total
Student	10	10	20	34
Professional	7	7	14	

	Early	Late	Total	Total
Student	2	6	8	18
Professional	6	4	10	

Luciano Sampaio 35



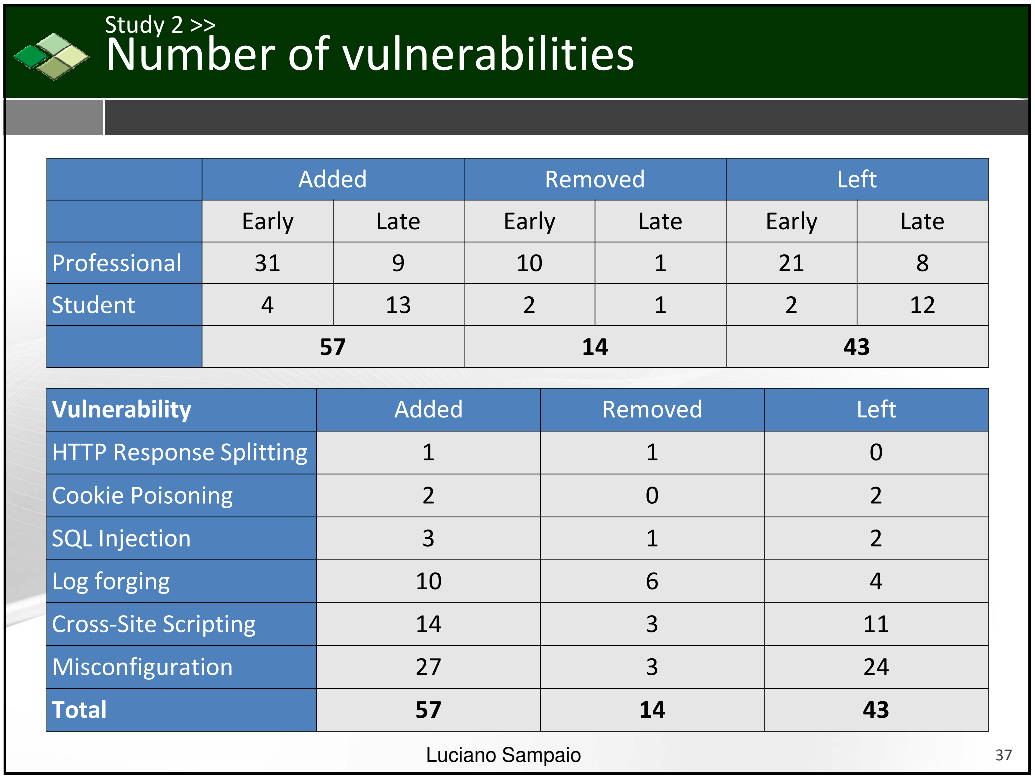
Study 2 >>

Programming timing and completed tasks

	Programming time		
	Early	Late	
Professional	9:32:40	4:31:07	**14:03:47**
Student	1:44:18	2:46:15	**4:30:33**
Sum	**11:16:58**	**7:17:22**	
Total	**18:34:20**		

Task 1	Task 2	Task 3	Task 4	Task 5
18	**8**	**4**	**2**	**2**

Luciano Sampaio 36

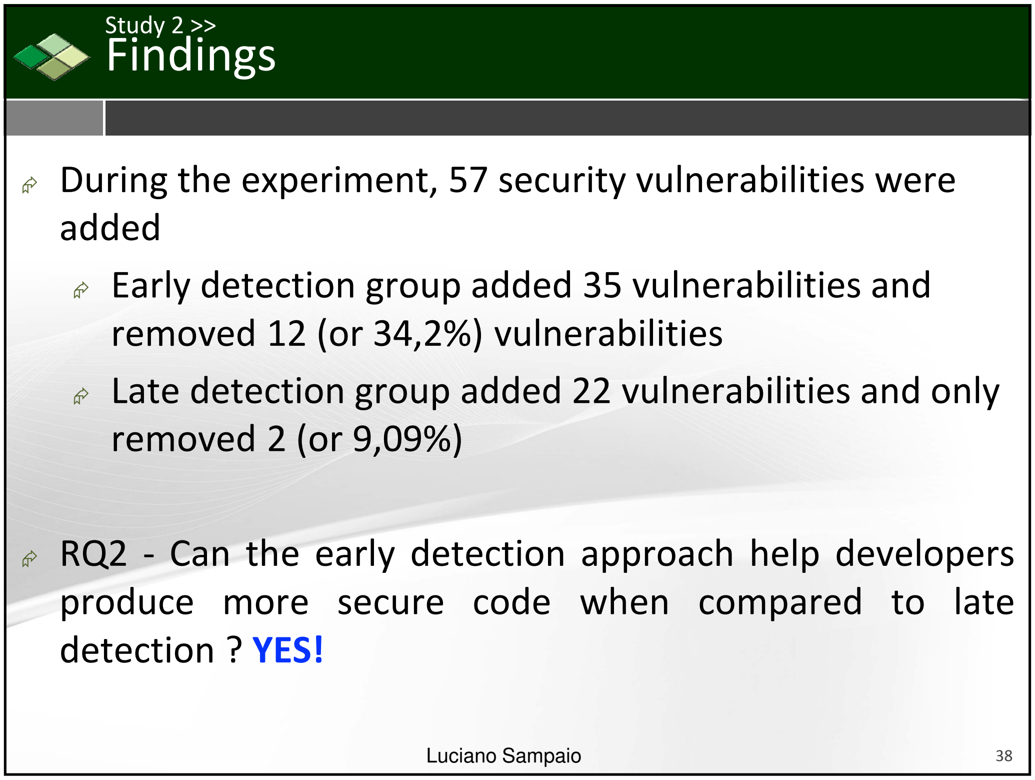


Study 2 >>

Number of vulnerabilities

	Added		Removed		Left	
	Early	Late	Early	Late	Early	Late
Professional	31	9	10	1	21	8
Student	4	13	2	1	2	12
	57		**14**		**43**	

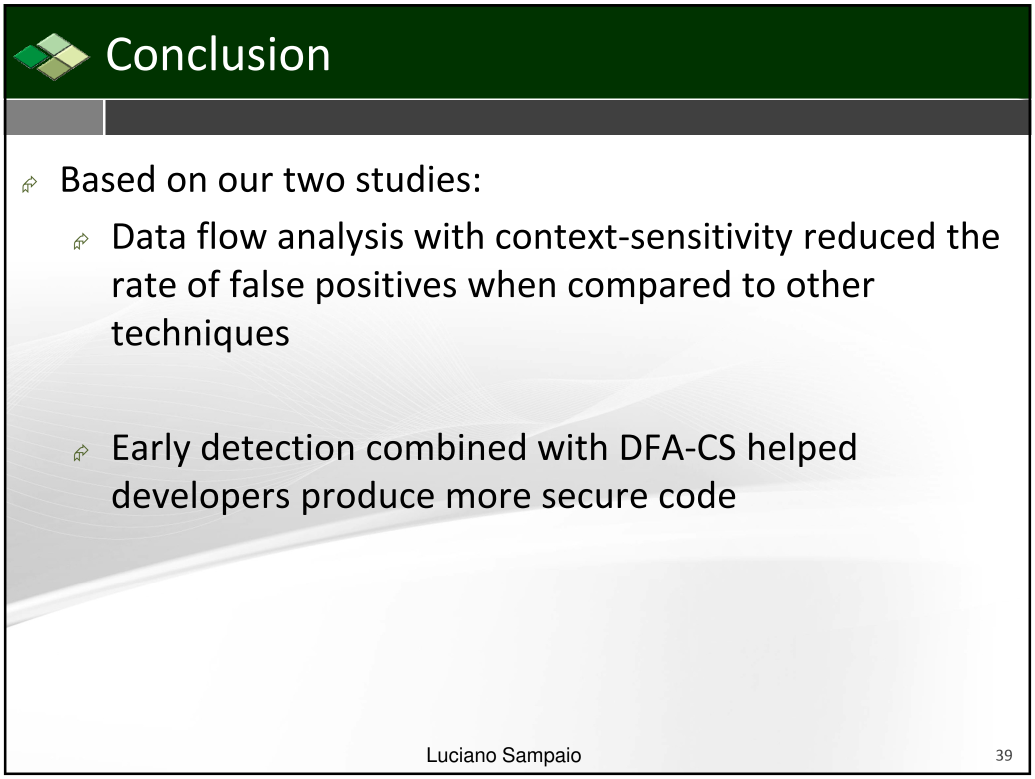
Vulnerability	Added	Removed	Left
HTTP Response Splitting	1	1	0
Cookie Poisoning	2	0	2
SQL Injection	3	1	2
Log forging	10	6	4
Cross-Site Scripting	14	3	11
Misconfiguration	27	3	24
Total	**57**	**14**	**43**



Study 2 >>

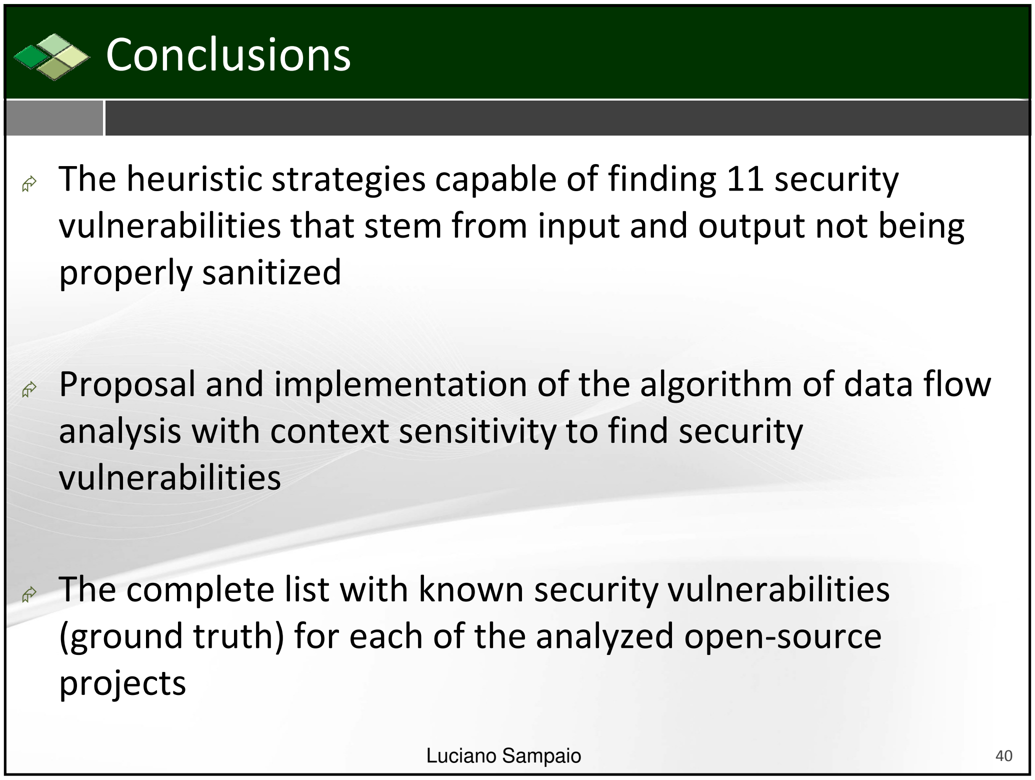
Findings

- During the experiment, 57 security vulnerabilities were added
  - Early detection group added 35 vulnerabilities and removed 12 (or 34,2%) vulnerabilities
  - Late detection group added 22 vulnerabilities and only removed 2 (or 9,09%)
- RQ2 - Can the early detection approach help developers produce more secure code when compared to late detection ? **YES!**



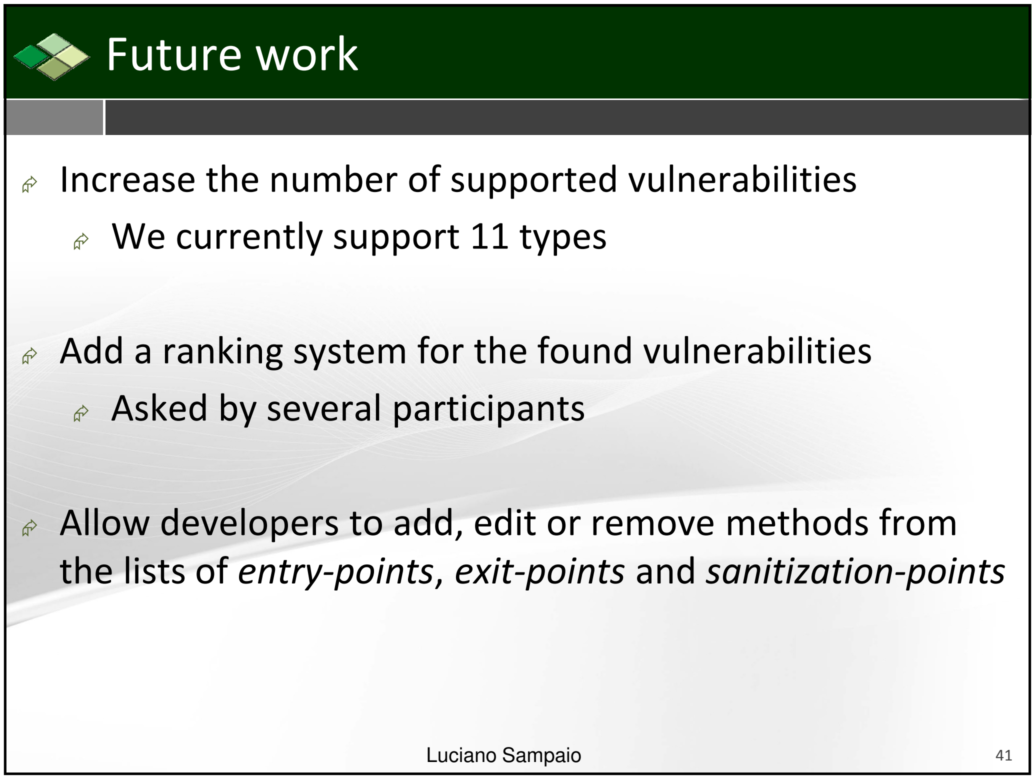
Conclusion

- Based on our two studies:
  - Data flow analysis with context-sensitivity reduced the rate of false positives when compared to other techniques
  - Early detection combined with DFA-CS helped developers produce more secure code



Conclusions

- The heuristic strategies capable of finding 11 security vulnerabilities that stem from input and output not being properly sanitized
- Proposal and implementation of the algorithm of data flow analysis with context sensitivity to find security vulnerabilities
- The complete list with known security vulnerabilities (ground truth) for each of the analyzed open-source projects



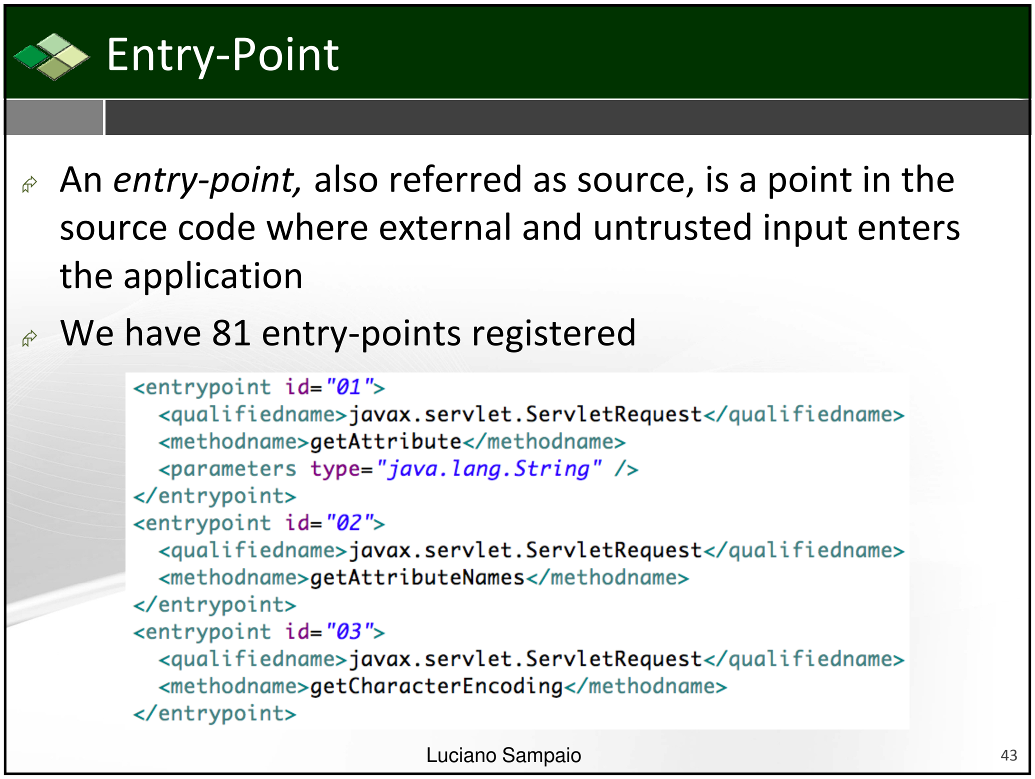
Future work

- Increase the number of supported vulnerabilities
  - We currently support 11 types
- Add a ranking system for the found vulnerabilities
  - Asked by several participants
- Allow developers to add, edit or remove methods from the lists of *entry-points*, *exit-points* and *sanitization-points*

Additional slides

LES | DI |PUC-Rio - Brazil

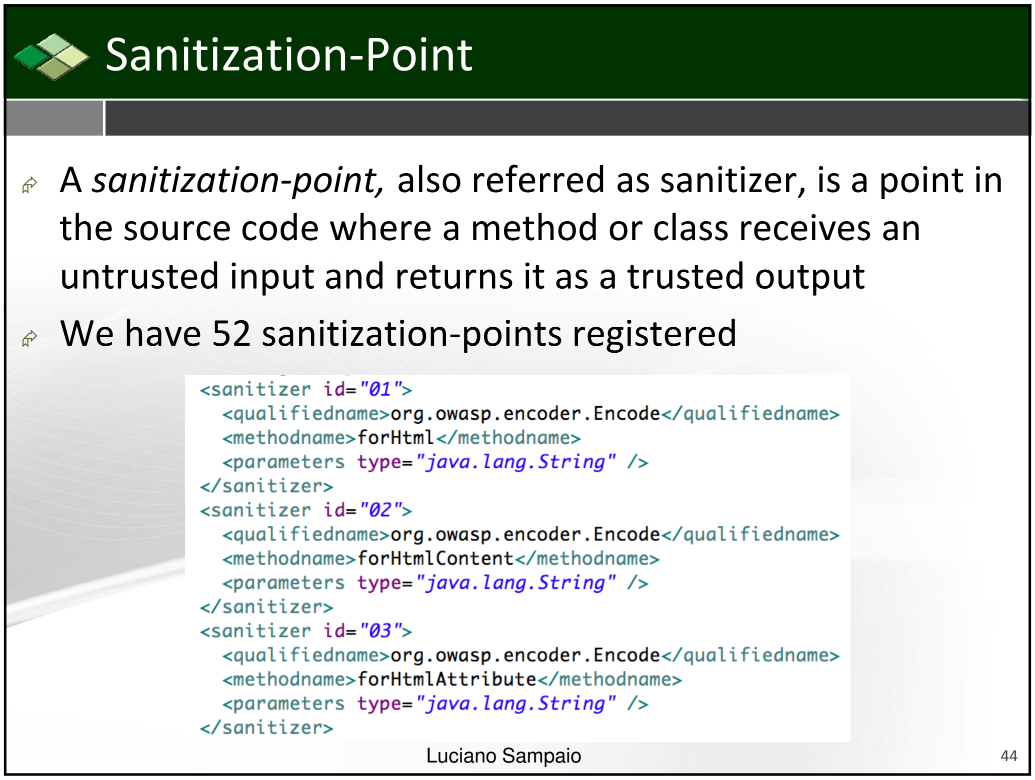
OPUS Research Group



Entry-Point

- An *entry-point,* also referred as source, is a point in the source code where external and untrusted input enters the application
- We have 81 entry-points registered

```
<entrypoint id="01">
  <qualifiedname>javax.servlet.ServletRequest</qualifiedname>
  <methodname>getAttribute</methodname>
  <parameters type="java.lang.String" />
</entrypoint>
<entrypoint id="02">
  <qualifiedname>javax.servlet.ServletRequest</qualifiedname>
  <methodname>getAttributeNames</methodname>
</entrypoint>
<entrypoint id="03">
  <qualifiedname>javax.servlet.ServletRequest</qualifiedname>
  <methodname>getCharacterEncoding</methodname>
</entrypoint>
```



Sanitization-Point

- A *sanitization-point,* also referred as sanitizer, is a point in the source code where a method or class receives an untrusted input and returns it as a trusted output
- We have 52 sanitization-points registered

```
<sanitizer id="01">
  <qualifiedname>org.owasp.encoder.Encode</qualifiedname>
  <methodname>forHtml</methodname>
  <parameters type="java.lang.String" />
</sanitizer>
<sanitizer id="02">
  <qualifiedname>org.owasp.encoder.Encode</qualifiedname>
  <methodname>forHtmlContent</methodname>
  <parameters type="java.lang.String" />
</sanitizer>
<sanitizer id="03">
  <qualifiedname>org.owasp.encoder.Encode</qualifiedname>
  <methodname>forHtmlAttribute</methodname>
  <parameters type="java.lang.String" />
</sanitizer>
```

Exit-Points >>

Accepted Rules

- 0 - Anything
- 1 - Sanitized
- 2 - Null
- 4 - Literal
- 8 - Concatenation (used for SQL Injection)
- …

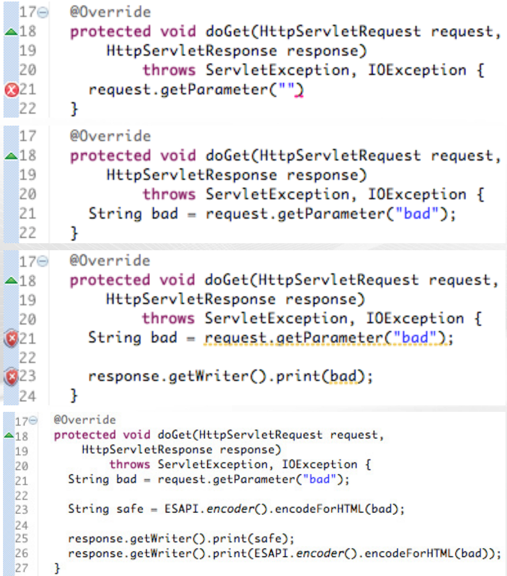
Pattern Matching vs Data Flow Analysis

Pattern Matching

Data Flow Analysis

Data Flow Analysis >>

Early Detection



```
@Override
protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
        throws ServletException, IOException {
  request.getParameter("")
}
```

```
@Override
protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
        throws ServletException, IOException {
  String bad = request.getParameter("bad");
}
```

```
@Override
protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
        throws ServletException, IOException {
  String bad = request.getParameter("bad");

  response.getWriter().print(bad);
}
```

```
@Override
protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
        throws ServletException, IOException {
  String bad = request.getParameter("bad");

  String safe = ESAPI.encoder().encodeForHTML(bad);

  response.getWriter().print(safe);
  response.getWriter().print(ESAPI.encoder().encodeForHTML(bad));
}
```

ESVD

Study 2 >>

Searching for help

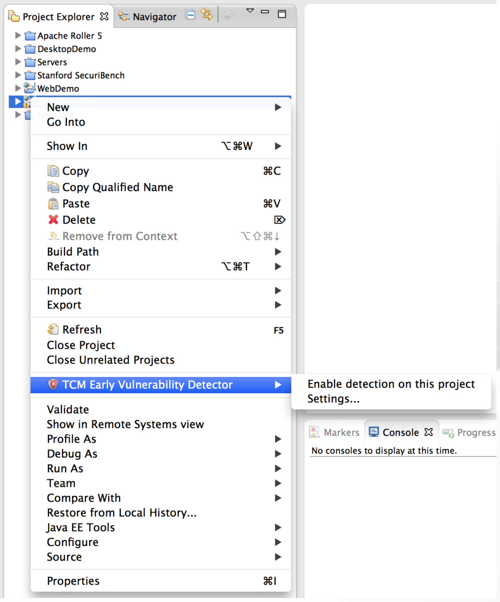
Link	Nr Times
1	27
2	8
3	5
4	6
6	4
7	1
10	1
	52

- You can learn from HelloWold, but should never use its source code.

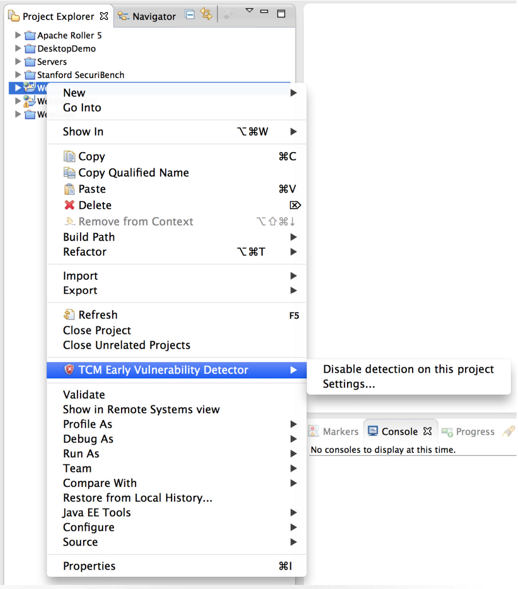
The plug-in

- ESVD - Early Security Vulnerability Detector - 0.3.9;
- Download at: (FREE)
  - https://marketplace.eclipse.org/content/early-security-vulnerability-detector-esvd/
- A project containing several security vulnerabilities:
  - http://www.inf.puc-rio.br/~lsampaio/plugin/early_vulnerability_detector/latest/WebDemo.zip
- How to use ESVD: (Portuguese)
  - https://www.youtube.com/watch?v=pNr38gMWvHQ
- More info at: http://thecodemaster.net/

The plug-in >> Menu



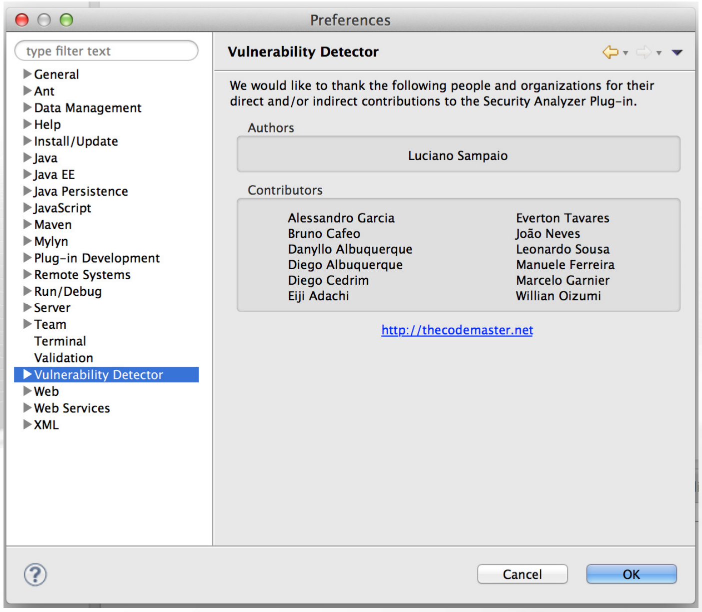
No consoles to display at this time.



No consoles to display at this time.

The plug-in >>

Preferences Page



Vulnerability Detector

We would like to thank the following people and organizations for their direct and/or indirect contributions to the Security Analyzer Plug-in.

Authors

Luciano Sampaio

Contributors

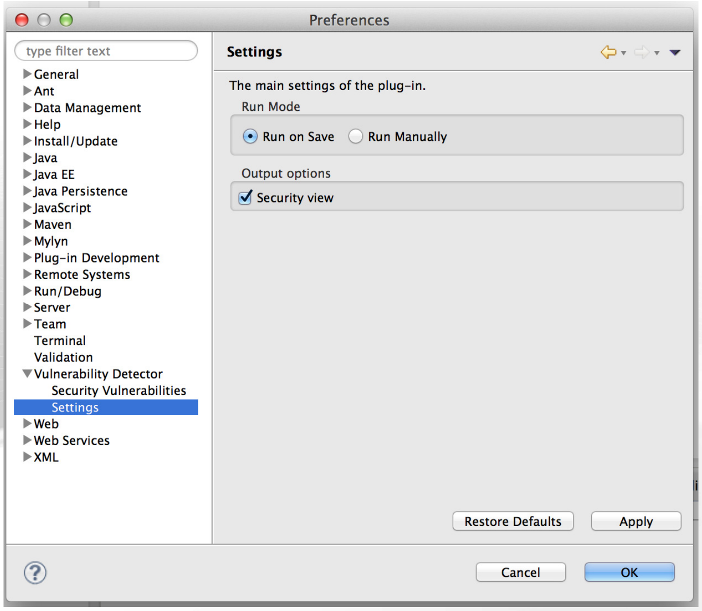
Alessandro Garcia	Everton Tavares
Bruno Cafeo	João Neves
Danyllo Albuquerque	Leonardo Sousa
Diego Albuquerque	Manuele Ferreira
Diego Cedrim	Marcelo Garnier
Eiji Adachi	Willian Oizumi

http://thecodemaster.net

Luciano Sampaio

The plug-in >>

Preferences Page



Settings

The main settings of the plug-in.

Run Mode

● Run on Save ○ Run Manually

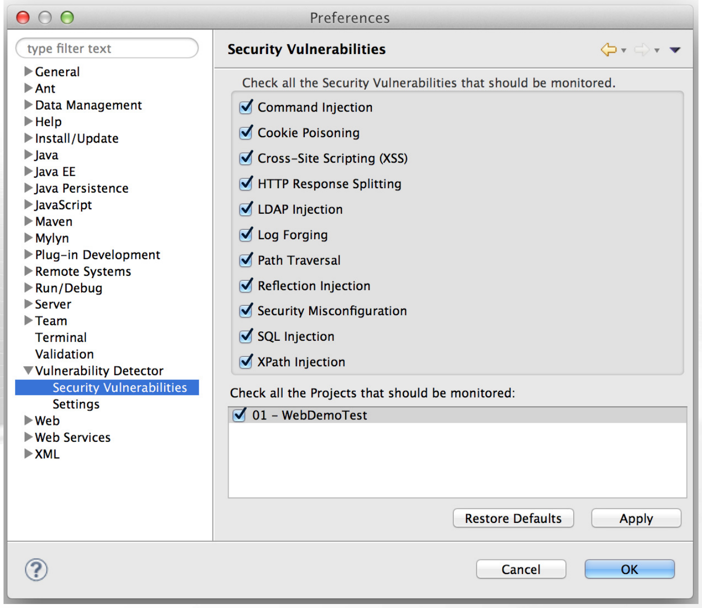
Output options

☑ Security view

Luciano Sampaio

The plug-in >>

Preferences Page



Security Vulnerabilities

Check all the Security Vulnerabilities that should be monitored.

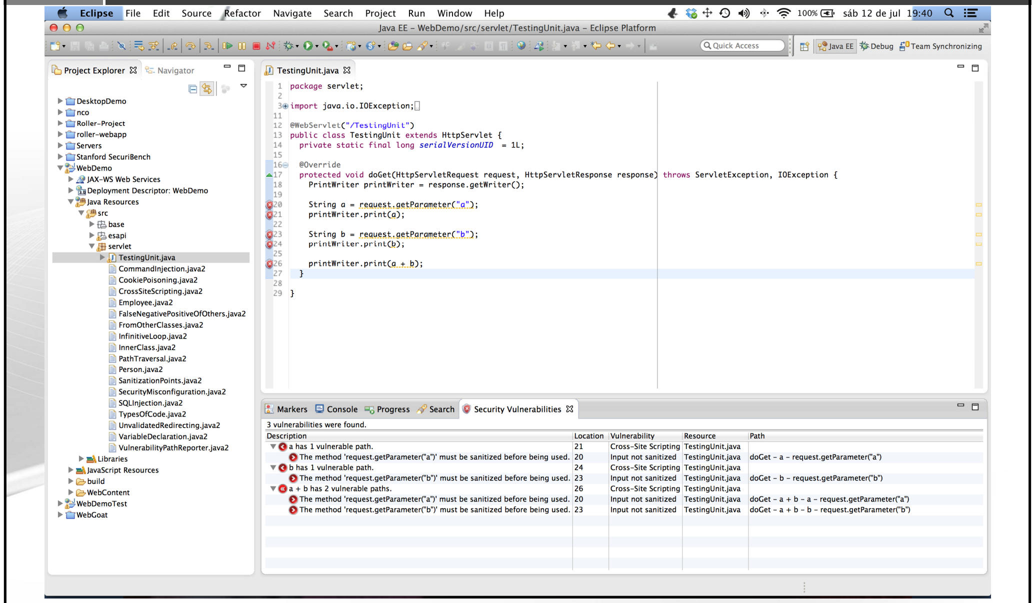
- ☑ Command Injection
- ☑ Cookie Poisoning
- ☑ Cross-Site Scripting (XSS)
- ☑ HTTP Response Splitting
- ☑ LDAP Injection
- ☑ Log Forging
- ☑ Path Traversal
- ☑ Reflection Injection
- ☑ Security Misconfiguration
- ☑ SQL Injection
- ☑ XPath Injection

Check all the Projects that should be monitored:

- ☑ 01 – WebDemoTest

The plug-in >>

User Interface



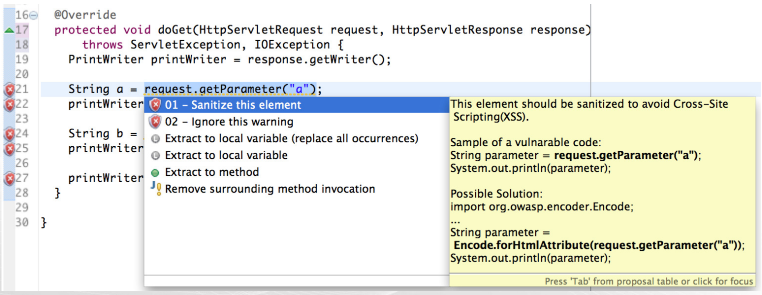
```java
package servlet;

import java.io.IOException;

@WebServlet("/TestingUnit")
public class TestingUnit extends HttpServlet {
  private static final long serialVersionUID  = 1L;

  @Override
  protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    PrintWriter printWriter = response.getWriter();

    String a = request.getParameter("a");
    printWriter.print(a);

    String b = request.getParameter("b");
    printWriter.print(b);

    printWriter.print(a + b);
  }

}
```

3 vulnerabilities were found.

Description	Location	Vulnerability	Resource	Path
a has 1 vulnerable path.	21	Cross-Site Scripting	TestingUnit.java	
The method 'request.getParameter("a")' must be sanitized before being used.	20	Input not sanitized	TestingUnit.java	doGet – a – request.getParameter("a")
b has 1 vulnerable path.	24	Cross-Site Scripting	TestingUnit.java	
The method 'request.getParameter("b")' must be sanitized before being used.	23	Input not sanitized	TestingUnit.java	doGet – b – request.getParameter("b")
a + b has 2 vulnerable paths.	26	Cross-Site Scripting	TestingUnit.java	
The method 'request.getParameter("a")' must be sanitized before being used.	20	Input not sanitized	TestingUnit.java	doGet – a + b – a – request.getParameter("a")
The method 'request.getParameter("b")' must be sanitized before being used.	23	Input not sanitized	TestingUnit.java	doGet – a + b – b – request.getParameter("b")

The plug-in >>

Provide possible solutions



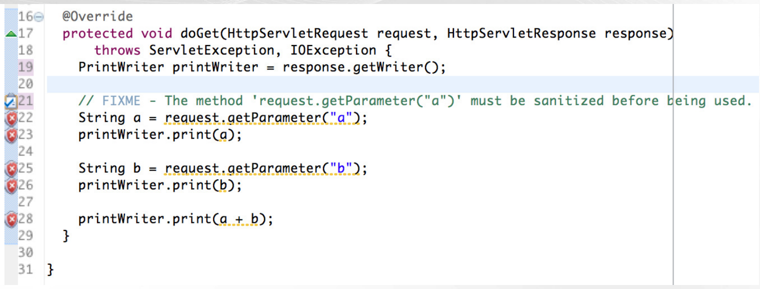
```
@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
  PrintWriter printWriter = response.getWriter();

  String a = request.getParameter("a");
  printWriter

  String b =
  printWriter

  printWriter
}
}
```

01 - Sanitize this element
02 - Ignore this warning
Extract to local variable (replace all occurrences)
Extract to local variable
Extract to method
Remove surrounding method invocation
Press 'Tab' from proposal table or click for focus

This element should be sanitized to avoid Cross-Site Scripting(XSS).

Sample of a vulnarable code:
String parameter = **request.getParameter("a");**
System.out.println(parameter);

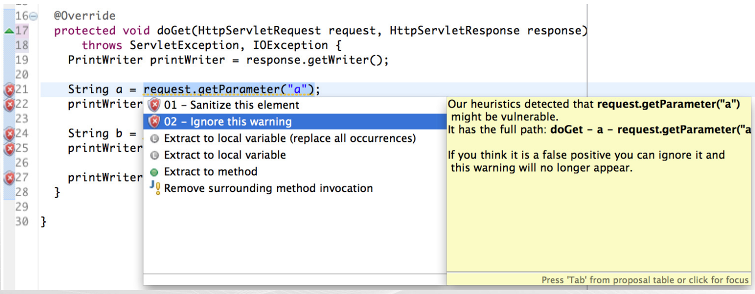
Possible Solution:
import org.owasp.encoder.Encode;
...
String parameter =
Encode.forHtmlAttribute(request.getParameter("a"));
System.out.println(parameter);



```
@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
  PrintWriter printWriter = response.getWriter();

  // FIXME - The method 'request.getParameter("a")' must be sanitized before being used.
  String a = request.getParameter("a");
  printWriter.print(a);

  String b = request.getParameter("b");
  printWriter.print(b);

  printWriter.print(a + b);
}
}
```

The plug-in >>

Provide possible solutions

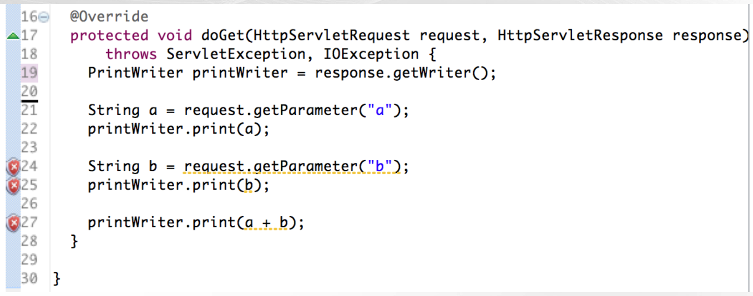


```
@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
  PrintWriter printWriter = response.getWriter();

  String a = request.getParameter("a");
  printWriter

  String b =
  printWriter

  printWriter
}
}
```

01 - Sanitize this element
02 - Ignore this warning
Extract to local variable (replace all occurrences)
Extract to local variable
Extract to method
Remove surrounding method invocation
Press 'Tab' from proposal table or click for focus

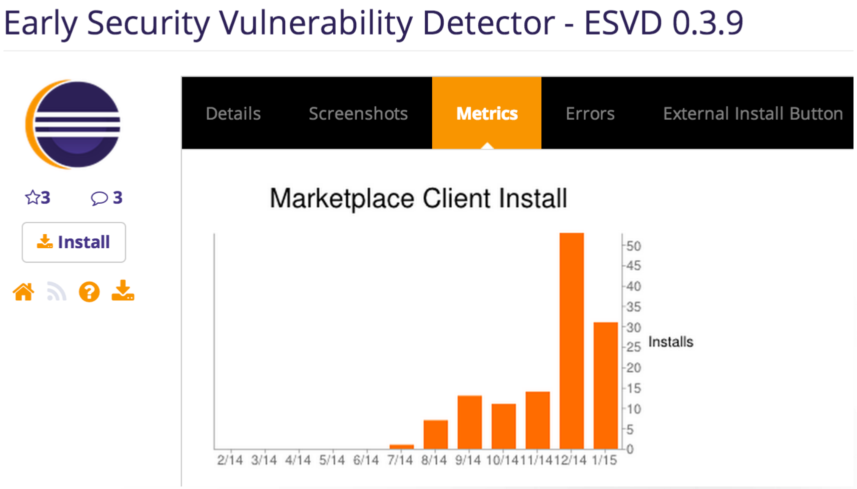
Our heuristics detected that **request.getParameter("a")** might be vulnerable.
It has the full path: **doGet – a – request.getParameter("a**

If you think it is a false positive you can ignore it and this warning will no longer appear.



```
@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
  PrintWriter printWriter = response.getWriter();

  String a = request.getParameter("a");
  printWriter.print(a);

  String b = request.getParameter("b");
  printWriter.print(b);

  printWriter.print(a + b);
}

}
```

The plug-in



Early Security Vulnerability Detector - ESVD 0.3.9