



Western Norway
University of
Applied Sciences

DAT152 – Advanced Web Applications

Security Tools - Static and Dynamic Testing



Goal

- Be aware of what static application security testing (SAST) tools are for web app testing
- Be aware of what runtime tools are for securing web applications
- Understand the metrics for evaluating the performance of analysis tools for security audit
- Understand the limitations of analysis tools

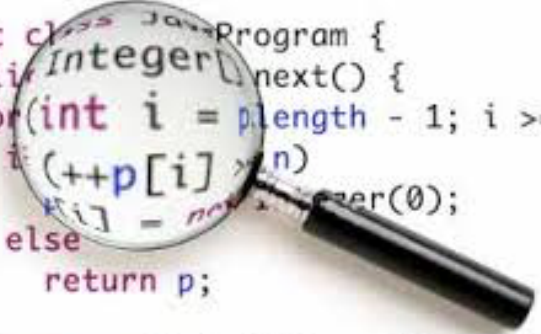
Tools for security testing of web apps

- Many possibilities: Commercial and open source
- Commercial
 - e.g. Checkmarx, HPFortify, Burp, Appscan, Coverity, etc.
- We will use open-source tools for our purpose
- Static Application Security Testing Tools (examples)
 - FindSecBugs (SpotBugs)
 - SonarQube
 - ASIDE
 - ESIDE
 - ESVD
 - etc.
- Runtime Tools for Security Audit
 - OWASP ZAP (Penetration testing tool)
 - SQLMap
 - Wfuzz
 - etc

What is Static Analysis?

- Passive scanning of application code without executing it
- *a source code security analyzer*
 - examines source code to
 - detect and report weaknesses that can lead to security vulnerabilities

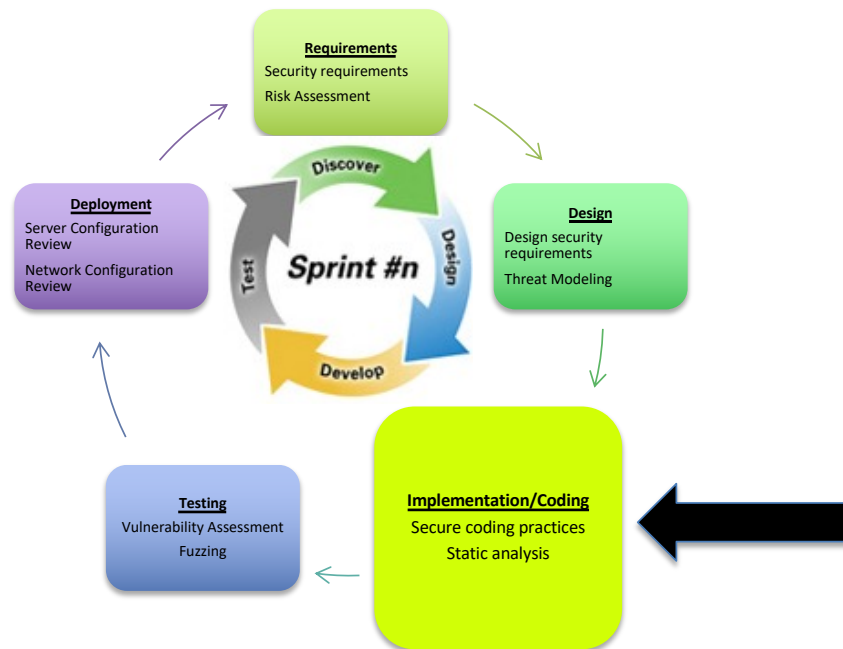
- Results can be in a report form
- May be integrated into IDEs
- May be integrated into CI/CD environments



```
public class JavaProgram {  
    public Integer[] next() {  
        for(int i = p.length - 1; i >= 0;  
            i: (++p[i] > n)  
            p[i] = nextInteger(0);  
        else  
            return p;  
        }  
        throw new NoSuchElementException();  
    }
```

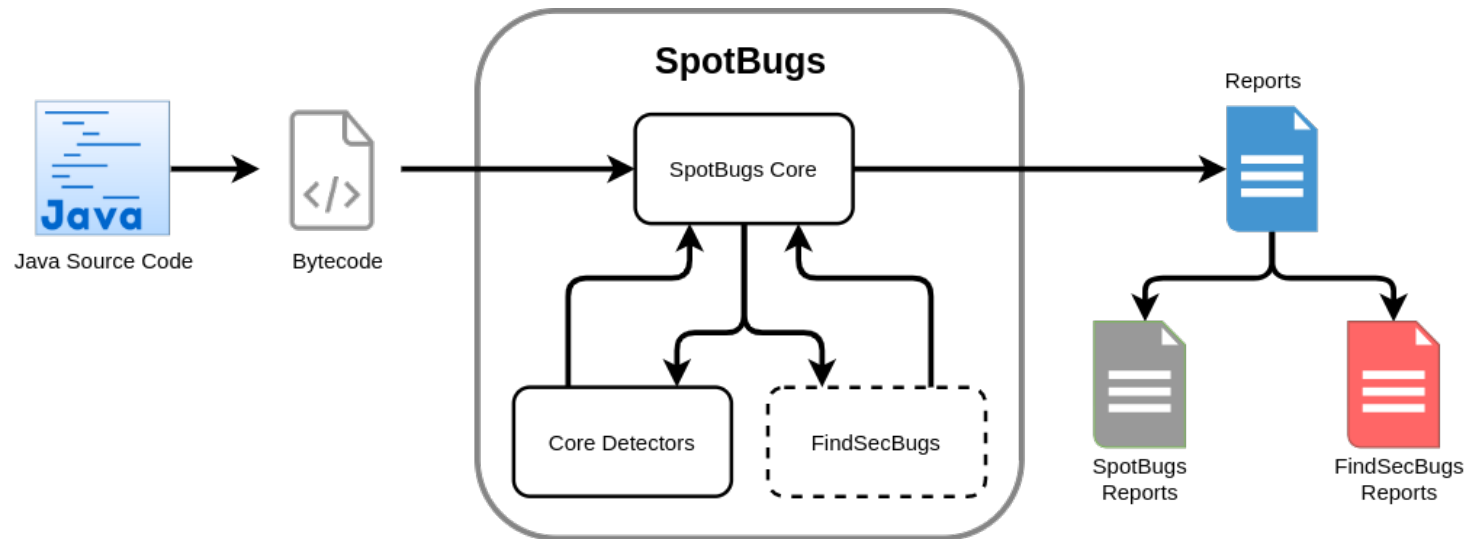
Why Static Code Analysis for security audits?

- Can catch security defects (early) at the implementation stage of the SDLC
- Significant aid for code review (Starting point)



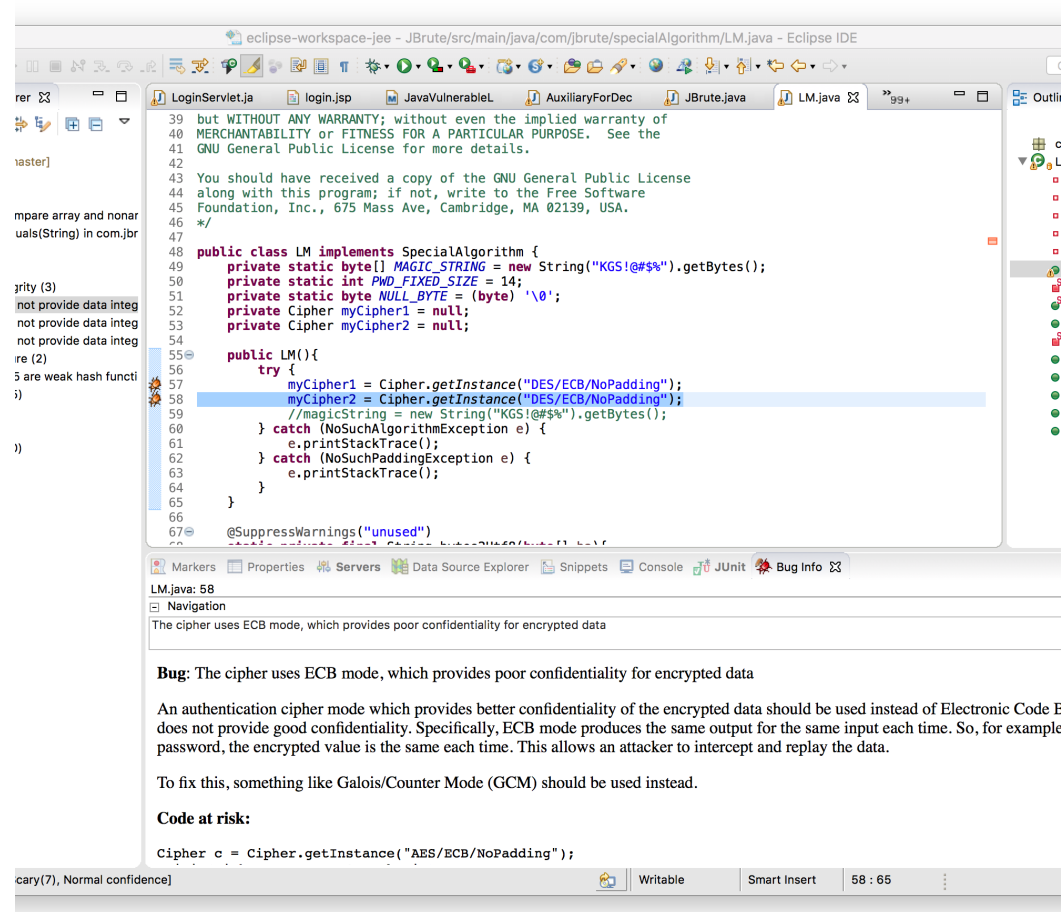
Static Application Security Tools (SAST)

- ✓ FindSecBugs
- ✓ SpotBugs
- ✓ CheckMarx
- ✓ Fortify
- ✓ ...



Static Application Security Tools (SAST)

- FindSecBugs
 - IDE plugin
 - Requires SpotBug
 - Add FindSecBugs plugins



Code vulnerabilities (Good vs. Bad)

- Benchmark - NIST testsuite
- XSS - example

```
/* goodG2B() - uses goodsource and badsink */
private void goodG2B(HttpServletRequest request, HttpServletResponse response)
    throws Throwable
{
    String data;

    /* FIX: Use a hardcoded string */
    data = "foo";

    if (data != null)
    {
        /* POTENTIAL FLAW: Display of data in web page
        * after using replaceAll() to remove script
        * tags, which will still allow XSS with strings
        * like <scr<script>ipt> (CWE 182: Collapse
        * of Data into Unsafe Value)
        */
        response.getWriter().println("<br>bad(): data = "
            + data.replaceAll("<script>", ""));
    }
}
```

```
/* uses badsource and badsink */
public void bad(HttpServletRequest request, HttpServletResponse response)
    throws Throwable
{
    String data;

    /* POTENTIAL FLAW: Read data from a
    * querystring using getParameter
    */
    data = request.getParameter("name");

    if (data != null)
    {
        /* POTENTIAL FLAW: Display of data in web page
        * after using replaceAll() to remove script
        * tags, which will still allow XSS with strings
        * like <scr<script>ipt> (CWE 182: Collapse
        * of Data into Unsafe Value)
        */
        response.getWriter().println("<br>bad(): data = "
            + data.replaceAll("<script>", ""));
    }
}
```


Code vulnerabilities (Good vs. Bad)

- Benchmark - NIST testsuite
- SQL - example

```
dbConnection = IO.getDBConnection();
sqlStatement = dbConnection.createStatement();
/* POTENTIAL FLAW: Read data using an outbound tcp connection
*/
data = readerBuffered.readLine();
/* POTENTIAL FLAW: data concatenated into SQL statement used
in executeQuery(), which could result in SQL Injection */
resultSet = sqlStatement.executeQuery("select * from users
where name='"+data+"'");
/* Use ResultSet in some way */
IO.writeLine(resultSet.getRow());
```

BadSource-BadSink

```
dbConnection = IO.getDBConnection();
sqlStatement = dbConnection.createStatement();
/* FIX: Use a hardcoded string */
data = "foo";
/* POTENTIAL FLAW: data concatenated into SQL statement
used in executeQuery(), which could result in SQL
Injection */
resultSet = sqlStatement.executeQuery("select * from
users where name='"+data+"'");
/* Use ResultSet in some way */
IO.writeLine(resultSet.getRow());
```

GoodSource-BadSink

Code vulnerabilities (Good vs. Bad)

- Benchmark - NIST testsuite
- SQL - example

```
dbConnection = IO.getDBConnection();
sqlStatement = dbConnection.createStatement();
/* POTENTIAL FLAW: Read data using an outbound tcp connection */
data = readerBuffered.readLine();
/* POTENTIAL FLAW: data concatenated into SQL statement used
in executeQuery(), which could result in SQL Injection */
resultSet = sqlStatement.executeQuery("select * from users
where name='"+data+"'");
/* Use ResultSet in some way */
IO.writeLine(resultSet.getRow());
```

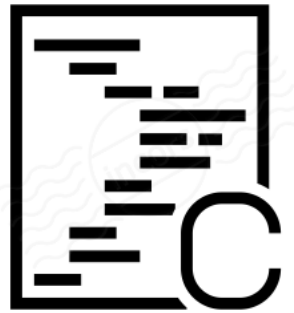
BadSource-BadSink

```
/* POTENTIAL FLAW: Read data using an outbound tcp
connection */
data = readerBuffered.readLine();
/* FIX: Use prepared statement and executeQuery (properly) */
dbConnection = IO.getDBConnection();
sqlStatement = dbConnection.prepareStatement("select * from
users where name=?");
sqlStatement.setString(1, data);
resultSet = sqlStatement.executeQuery();

/* Use ResultSet in some way */
IO.writeLine(resultSet.getRow());
```

BadSource-GoodSink

Code representations



Source code

```
// Bytecode stream: 03 3b 84 00 01 1a 05  
48 3b a7 ff f9  
// Disassembly:  
  
iconst_0      // 03  
istore_0      // 3b  
iinc 0, 1     // 84 00 01  
iload_0       // 1a  
iconst_2      // 05  
imul         // 68  
istore_0      // 3b  
goto -7       // a7 ff f9
```

Bytecode



Security bugs

```
BINARY  
  
00110001 00110001 00110000 00110000 00110000 00110000 001  
00110000 00110001 00110000 00110001 00110000 00110000 001  
00110001 00110001 00110001 00110001 00110000 00110000 001  
00110001 00110001 00110001 00110000 00110000 00110001 001  
00110001 00110000 00110000 00110001 00110000 00110000 001  
00110000 00110001 00110001 00110001 00110000 00110000 001  
00110001 00110001 00110001 00110001 00110000 00110001 001  
00110001 00110001 00110000 00110001 00110001 00110000 001  
00110000 00110000 00110001 00110001 00110001 00110001 001  
00110001 00110000 00110001 00110001 00110001 00110000 001
```

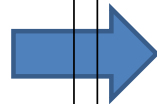
Binaries

```

public class XmlDecodeUtil {

    public static Object handleXml(InputStream in) {
        XMLDecoder d = new XMLDecoder(in);
        try {
            //Deserialization happen here
            Object result = d.readObject();
            return result;
        }
        finally {
            d.close();
        }
    }
}

```



```

public class spotbugs.XmlDecodeUtil {
    public spotbugs.XmlDecodeUtil();
    Code:
        0: aload_0
        1: invokespecial #8          // Method java/lang/Object."<init>":()V
        4: return

    public static java.lang.Object handleXml(java.io.InputStream);
    Code:
        0: new           #16          // class java/beans/XMLDecoder
        3: dup
        4: aload_0
        5: invokespecial #18          // Method java/beans/XMLDecoder."<init>":(Ljava/io/InputStream;)V
        8: astore_1
        9: aload_1
       10: invokevirtual #21          // Method java/beans/XMLDecoder.readObject:()Ljava/lang/Object;
       13: astore_2
       14: aload_2
       15: astore        4
       17: aload_1
       18: invokevirtual #25          // Method java/beans/XMLDecoder.close:()V
       21: aload         4
       23: areturn
       24: astore_3
       25: aload_1
       26: invokevirtual #25          // Method java/beans/XMLDecoder.close:()V
       29: aload_3
       30: athrow
    Exception table:
        from    to  target type
         9      17    24    any
}

```

javap -p -c -constants XmlDecodeutil.class

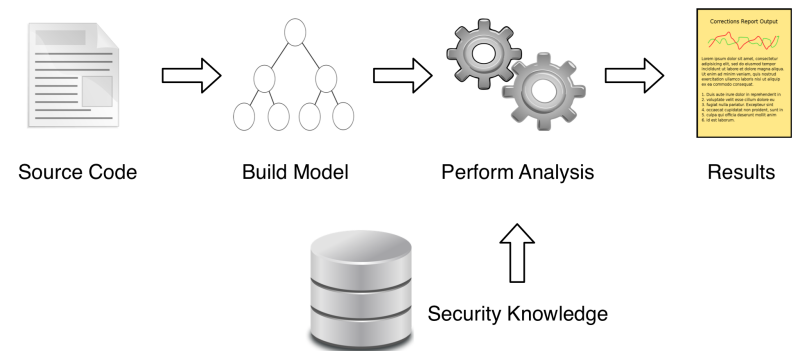
Bytecode instructions

Techniques for static code analysis

- Pattern matching
- Control flow analysis
- Data flow analysis
- Taint analysis (Security)
 - Source (entry points)
 - Sink (Exit points)
 - Sanitization points (Filters)

Techniques for static code analysis (security)

- Model construction for source code analysis
 - Lexical Analysis
 - Split source code into tokens to identify language constructs correctly. Remove unimportant tokens (e.g., whitespaces, comments, etc)
 - Semantic Analysis
 - Check the representation of each token for meaning (types, declarations, etc)
 - Control Flow Analysis
 - Possible paths that the program can take are determined and combined to several control flow graphs that represent all possible data flow paths
- Security knowledge



Pattern Matching

- Report security issues if these patterns are found in the code
 - `USE_OF_INSECURE_RANDOM_FUNCTION`

Bug: This random generator (`java.util.Random`) is predictable

The use of a predictable random value can lead to vulnerabilities when used in certain security critical contexts. For example, when the value is used as:

X a CSRF token: a predictable token can lead to a CSRF attack as an attacker will know the value of the token

X a password reset token (sent by email): a predictable password token can lead to an account takeover, since an attacker will guess the URL of the "change password" form

X any other secret value

Bad Random Function - Example

```
public class BadRandomFunction {  
    public void generateRandomNumber() {  
        Random r = new Random();  
        r.nextInt();  
    }  
}
```

ByteCode

```
if(clzNameInvoked.equals(BAD_RANDOM_FUNCTION)) {  
    System.out.println("BAD RANDOM FUNCTION BUG Detected "  
        + "\n@class: "+jclzName+"\n@Method: "+methodName  
        + "\n@LineNumber: "+lineNumber);  
}
```

CodeAnalyzer

Report Bug

```
private static final String  
BAD_RANDOM_FUNCTION = "java/util/Random";
```


Control Flow

Code can have different control flows

Control flow: If-Statements

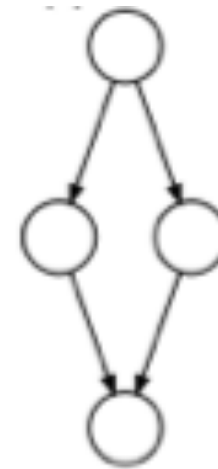
Control flow: While-Statements

Control flow: For-Statements

Control flow: Switch-Statements

Control flow: Exceptions

...



if-then-else



a while loop

Static analyzers need to cover the branches during code analysis –
Control Flow Graphs

Data Flow

Different patterns by which data flows in a program

- ✓ Change in assignments of variables during the execution of a program
- ✓ Intra procedure (within method)
- ✓ Inter procedure (Between methods)
 - ✓ Within a class
 - ✓ Between classes
- ✓ Data flow uses the control flow graphs

```
public static Object handleXml() throws FileNotFoundException {  
    String xmlfile = "Test.xml"; // hardcoded SAFE  
    XMLDecoder d = new XMLDecoder(new BufferedInputStream(new FileInputStream(xmlfile)));  
    try {  
        Object result = d.readObject(); //Deserialization happen here  
        return result;  
    }  
    finally {  
        d.close();  
    }  
}
```

Intra procedure (within method)

Data Flow – Inter (class – class)

Class 1

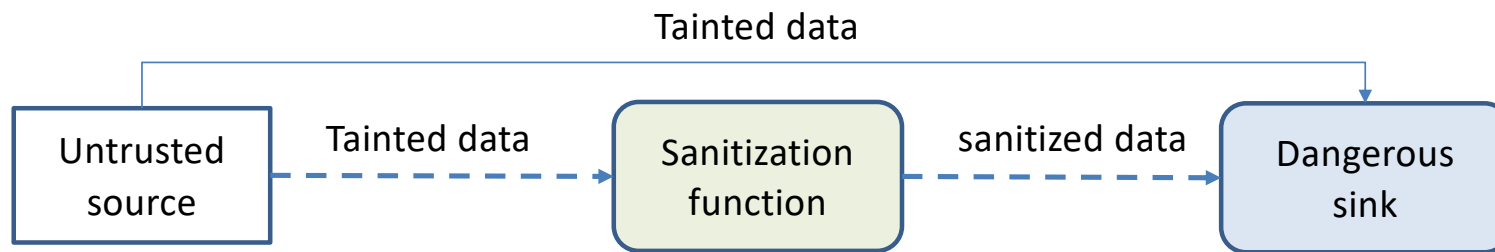
```
public class InterClassDataflow {  
    public String getSysProperty() {  
        String data = System.getProperty("config");  
        return data;  
    }  
}
```

Class 2

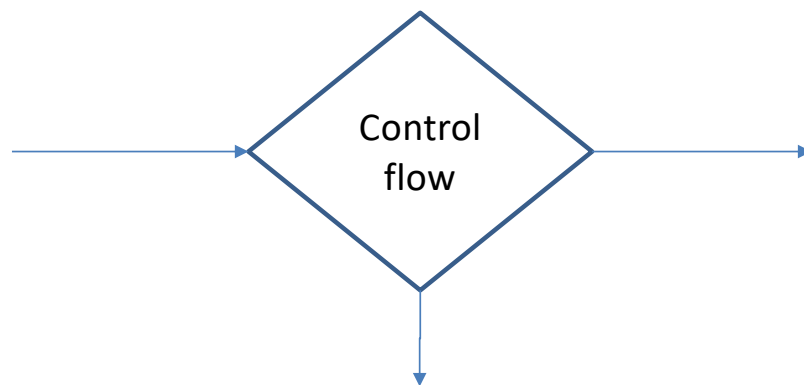
```
public class InterClassDataflow2 {  
    private InterClassDataflow icdf;  
    public InterClassDataflow2(InterClassDataflow icdf) {  
        this.icdf = icdf;  
    }  
    public void useCalleeData() {  
        String prop = icdf.getSysProperty();  
        System.out.println("Tainted: " + prop);  
    }  
}
```

Wrap data in class object to be passed to another class: Composition relation in UML

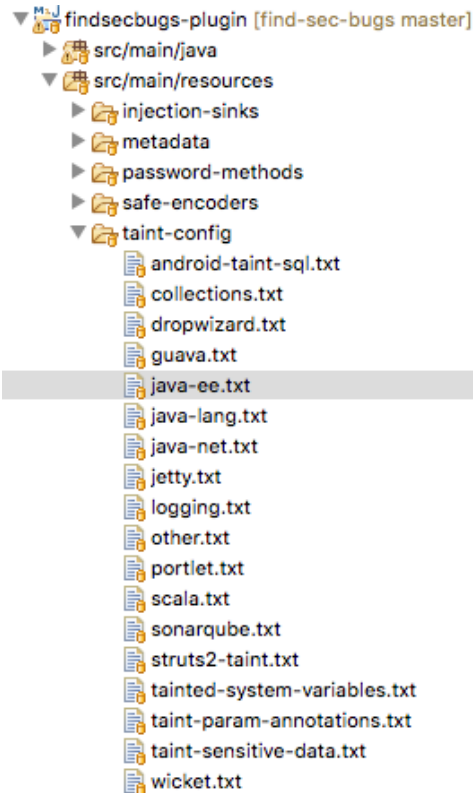
Data Flow - Taint Analysis



If a tainted data reaches a dangerous sink, a security issue may occur



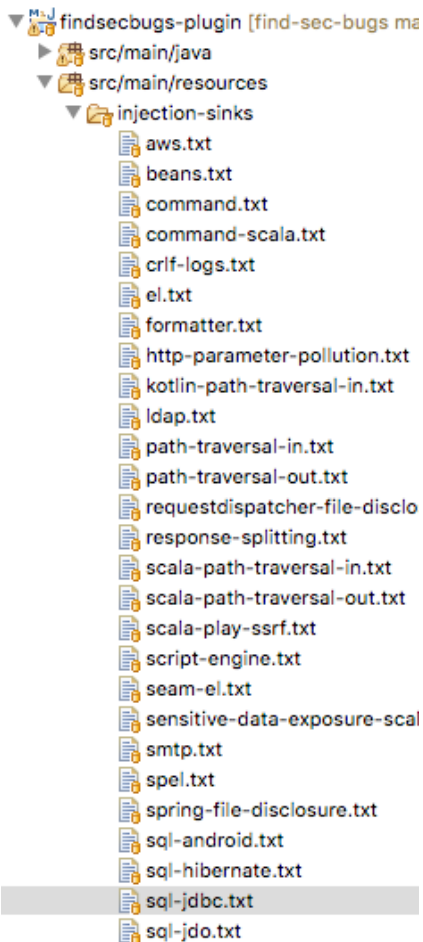
Taint Analysis – Sources (tainted)



```
javax/servlet/ServletRequest.getAttributeNames()Ljava/util/Enumeration;:SAFE  
javax/servlet/ServletRequest.getCharacterEncoding()Ljava/lang/String;:SAFE  
javax/servlet/ServletRequest.getContentType()Ljava/lang/String;:TAINTED  
javax/servlet/ServletRequest.getLocalAddr()Ljava/lang/String;:TAINTED  
javax/servlet/ServletRequest.getLocalName()Ljava/lang/String;:TAINTED  
javax/servlet/ServletRequest.getParameter(Ljava/lang/String;)Ljava/lang/String;:TAINTED  
javax/servlet/ServletRequest.getParameterMap()Ljava/util/Map;:TAINTED
```

A source is tagged tainted if its value can be manipulated by an external actor (process or human)

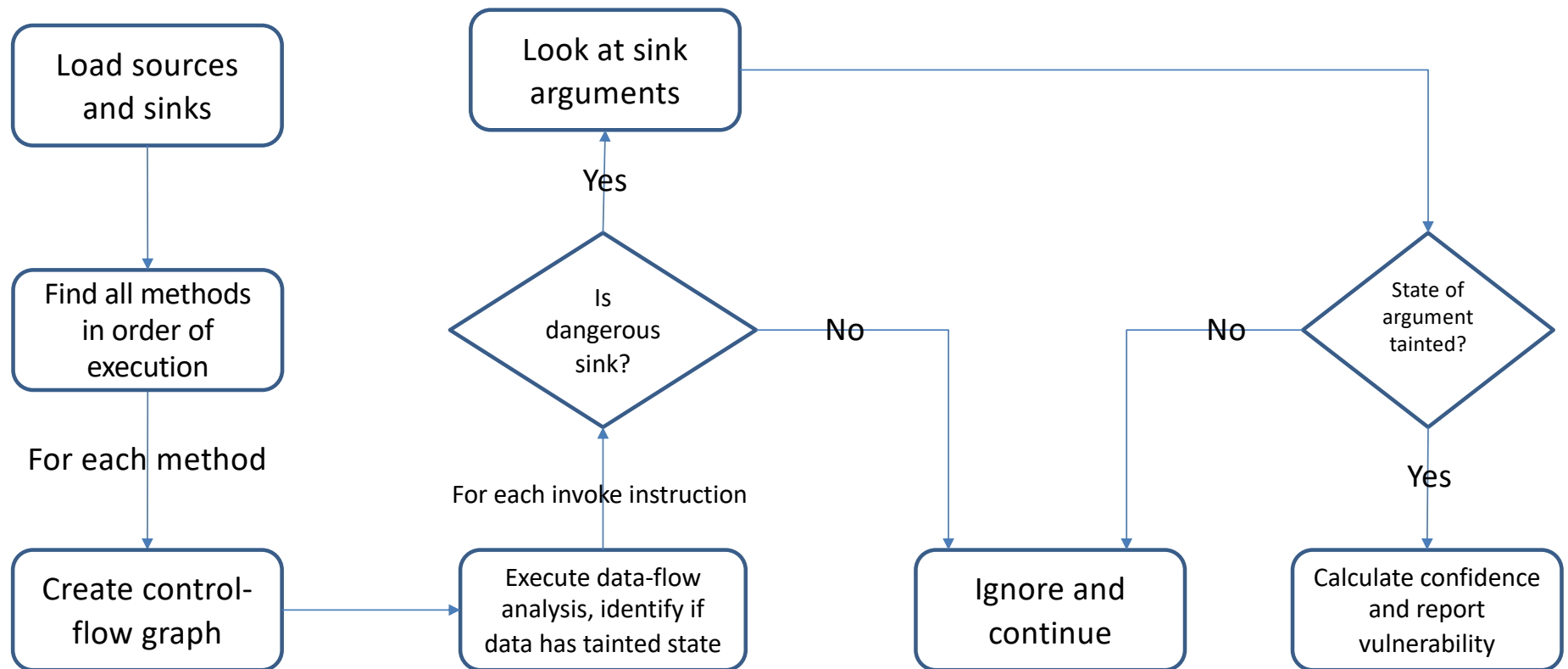
Taint Analysis – Sink (Dangerous)



```
java/sql/Statement.executeQuery(Ljava/lang/String;)Ljava/sql/ResultSet;:0  
java/sql/Statement.execute(Ljava/lang/String;)Z:0  
java/sql/Statement.execute(Ljava/lang/String;I)Z:1  
java/sql/Statement.execute(Ljava/lang/String;[I)Z:1  
java/sql/Statement.execute(Ljava/lang/String;[Ljava/lang/String;)Z:1  
java/sql/Statement.executeUpdate(Ljava/lang/String;)I:0  
java/sql/Statement.executeUpdate(Ljava/lang/String;I)I:1  
java/sql/Statement.executeUpdate(Ljava/lang/String;[I)I:1
```

A sink is of security interest if its operation can read or write - CRUD

Taint analysis (e.g., FindSecBugs)



SQL Injection Analysis - SpotBugs

- Detector Name:
 - FindSqlInjection
- Vulnerability Type:
 - SQL_NONCONSTANT_STRING_PASSED_TO_EXECUTE
 - SQL_PREPARED_STATEMENT_GENERATED_FROM_NONCONSTANT_STRING
- Limitations:
 - Reports on all use of string concatenation in conjunction with a sink (exit point)
 - High recall for all string concatenation cases but low precision
 - Does not use taint analysis (dangerous sources)
 - False positives when non-constant string is re-assigned a 'safe' value

SQL Injection Analysis - SpotBugs

✗ SpotBug detector can't discriminate a SQL injection – False positive

```
/* POTENTIAL FLAW: data concatenated into SQL statement used in execute(), which could result in SQL Injection */
/* FIX: Use a hardcoded string */
data = "foo";
Boolean result = sqlStatement.execute("insert into users (status) values ('updated') where name='"+data+"'");
```

✓ SpotBug detects a SQL injection based on vulnerable sink

```
String data = ""; /* Initialize data */

InputStreamReader readerInputStream = new InputStreamReader(System.in, "UTF-8");
BufferedReader readerBuffered = new BufferedReader(readerInputStream);

/* POTENTIAL FLAW: Read data from the console using readLine */
data = readerBuffered.readLine();

/* POTENTIAL FLAW: data from untrusted source, which could result in SQL Injection */
Boolean result = sqlStatement.execute(data);
```

Problem: Limited data flow analysis

Object and Context sensitivity

✓ Detector is both object and context insensitive

```
34 String name = request.getParameter("name");
35 UserNames un1 = new UserNames(name); // vulnerable
36 UserNames un2 = new UserNames("Anonymous"); // not vulnerable
37
38 PrintWriter pw = response.getWriter();
39 pw.println(un1.getUser());
40 pw.println(un2.getUser());
```

Example of difficult cases for FindSecBugs

- Data flow cases
 - Passing value between methods using a class field
 - Wrapping data into other data structures (e.g., Java Container or serializing and deserializing)
 - Re-assigning non-constants
 - Challenging to identify custom sanitisation functions
 - ...
- New vulnerable sources have to be updated for taint analysis
 - e.g., Log4J JNDI Lookup classes
 - ...

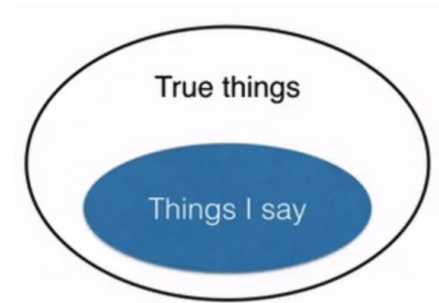
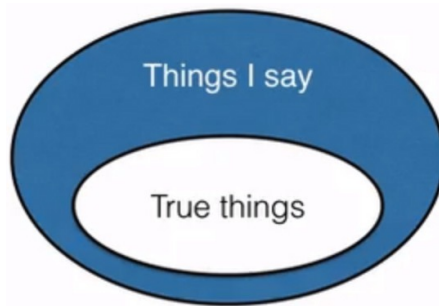
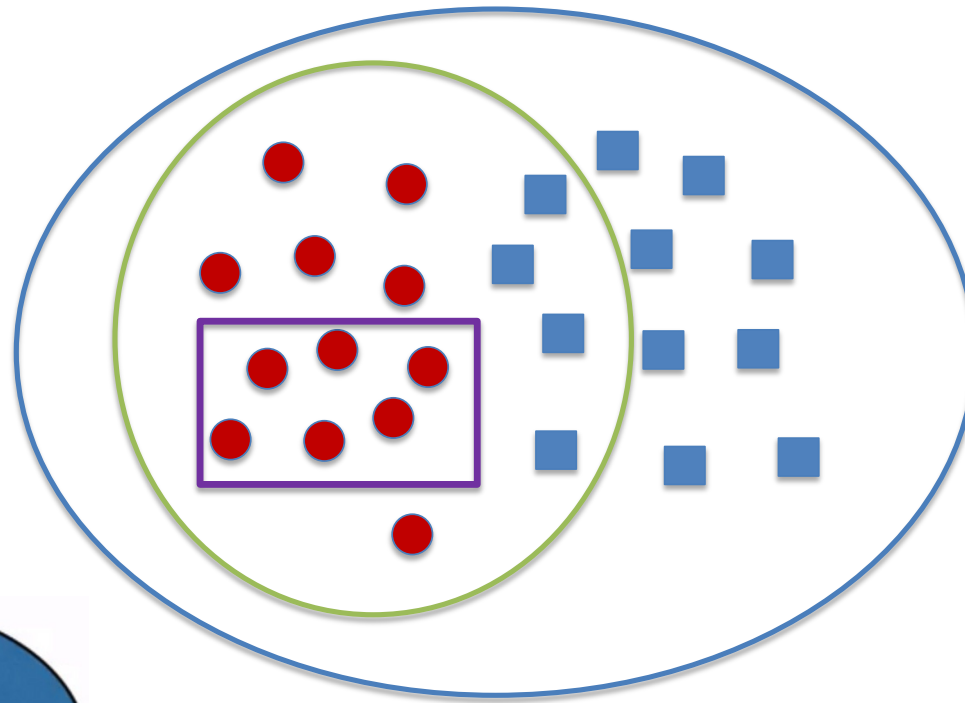
Performance: How good are SAST tools?

- We use certain metrics to measure how good a SAST tool performs
 - Soundness
 - Completeness
- True Positive
- True Negative
- False Positive
- False Negative

Soundness and Completeness

- Non-security bug
- Security bug

Total files = ■ + ●

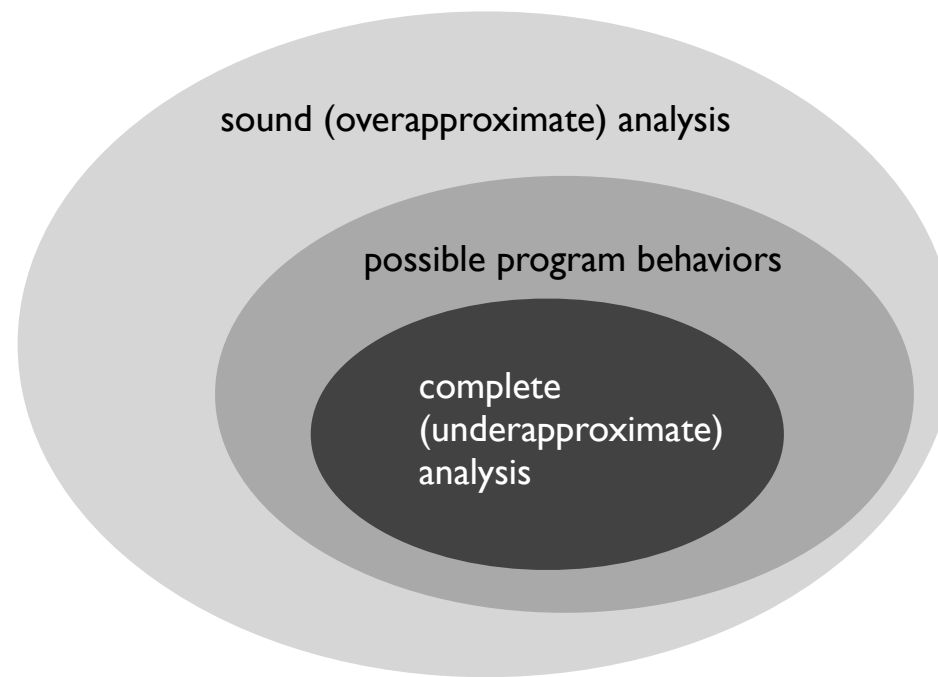


Soundness and Completeness

- **Soundness:** All reported bugs are real. **Sound** if it never overapproximates the set of bugs in a given program
 - If the analysis reports bugs that are not real bugs, then the analysis is **unsound**.
 - We refer to this situation as **false positives**
- **Completeness:** All bugs are reported. **Complete** if it never underapproximates the set of bugs in a given program.
 - If the analysis misses real bugs, then it is **incomplete**.
 - We refer to these misses as **false negatives**

Practical Static Analysis

- Practical static analysis tools' behaviour make compromises between soundness and completeness

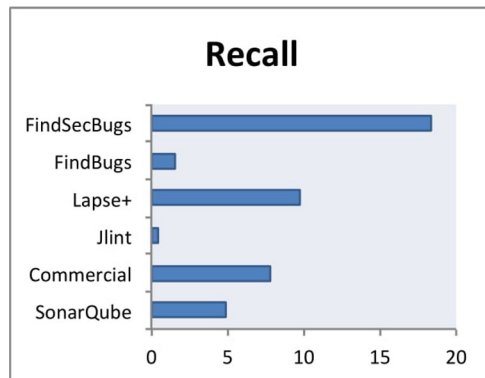


Test Suite – Performance of Tools

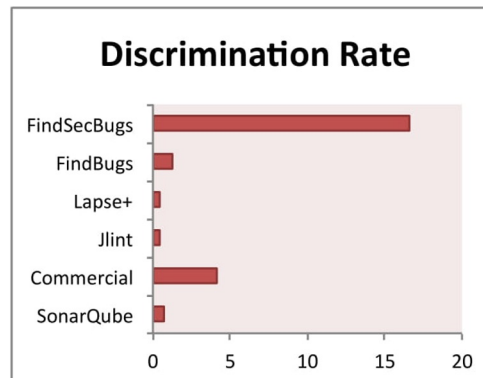
- Static analysis tools - overall performance (see paper on Canvas)

. Number of identified weaknesses by tools from a total of 26120 flaws

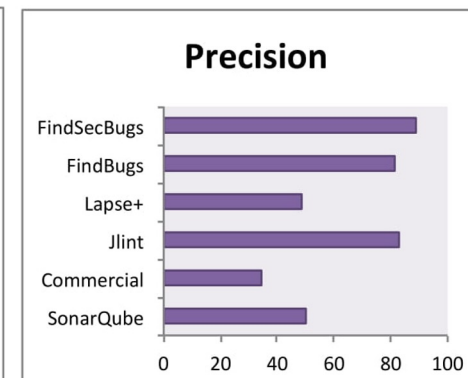
Tool	TP	FP	#Discrimination	Incidental flaws
SonarQube	1292	1275	200	237845
Commercial	2038	3834	1085	360212
FindSecBugs	4811	604	4338	41637
Lapse+	2550	2736	108	18950
JLint	125	26	104	586
FindBugs	426	98	341	22245



$$\text{Recall} = \text{TP} / (\text{TP} + \text{FN})$$



$$\text{Discrimination rate} = \frac{\text{\#discriminations}}{\text{TP} + \text{FN}}$$



$$\text{Precision} = \text{TP} / (\text{TP} + \text{FP})$$

Limitations of SAST Tools

- False Positives
- False Negatives
- Model Construction
 - Can miss certain dependencies (e.g. use of reflection)
- May fail to detect implemented filter/control/validator ☹️
- May not test whether your control is strong enough
- May not find issues in the operational environment
- Only cover half of security defects
- Can not check many design issues
- Provides little insight into the exploitability of the weakness/vulnerability itself
- Can not present an accurate risk picture

Demo with FindSecBug

Runtime/Dynamic Analysis Testing

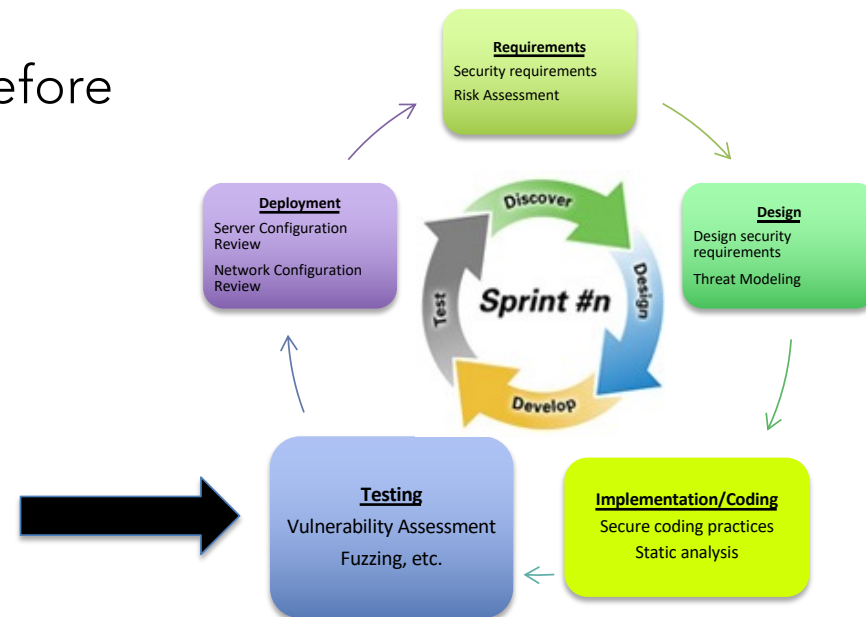
Here, we focus more on **system testing** not individual components

It means that the system must be running, and we want to find vulnerable paths where the system can be exploited

- Penetration testing (e.g., ZAP, Burp Suite)
- Brute forcing directories (DirBuster, Wfuzz)
- Fuzzing (manipulated payloads)
- Sqlmap (<https://github.com/sqlmapproject/sqlmap>)
- etc

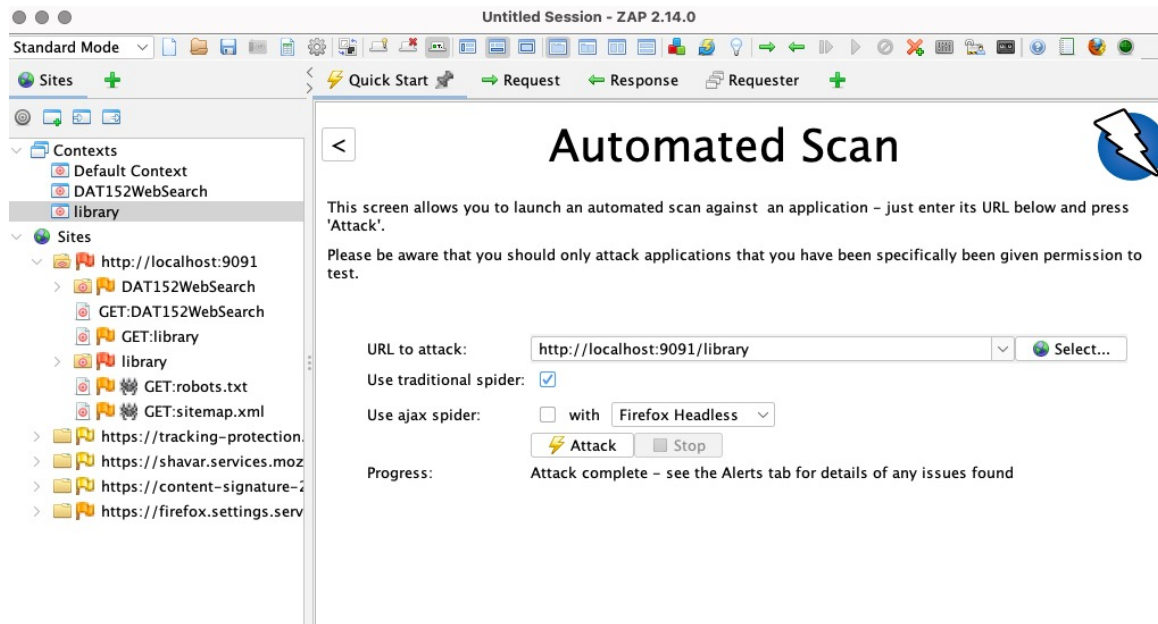
Why Runtime Security Testing?

- Can catch security defects that can be exploited during operation
- Can provide true picture of the security posture of an application
- Significant aid for security testing before applications become operational



Example – Library Web Application

Web app is running at: `http://localhost:9091/library`



Endpoints – Spidering

A spider crawls on all of the pages starting from the specified URL.

- Collects all different endpoints and the HTTP method

Method	URI
GET	http://localhost:9091/library/
GET	http://localhost:9091/library/do/viewbooks
GET	http://localhost:9091/library/do/addbookform
GET	http://localhost:9091/library/do/addauthorform
POST	http://localhost:9091/library/do/addauthor
POST	http://localhost:9091/library/do/addbook
GET	http://localhost:9091/library/do/viewbook?authorid=1&isbn=4f110065-ed9-4cba-91de-e9...
GET	http://localhost:9091/library/do/updatebookform?authorid=1&isbn=4f110065-ed9-4cba-9...
GET	http://localhost:9091/library/do/deletebookconfirm?authorid=1&isbn=4f110065-ed9-4cba...
POST	http://localhost:9091/library/do/updatebook
GET	http://localhost:9091/library/do/deletebook?isbn=4f110065-ed9-4cba-91de-e932e8ae8560
GET	http://localhost:9091/library/do/updatebookform?isbn=4f110065-ed9-4cba-91de-e932e8...

Fuzzed Payloads

Endpoints are tested for different fuzzed payload based on the type of vulnerability being tested.

SQL injection

```
' or "  
-- or #  
' OR '1  
' OR 1 -- -  
" OR "" = "  
" OR 1 = 1 -- -  
' OR '' = '  
'=  
'LIKE'  
'=0--+  
OR 1=1  
' OR 'x'='x  
' AND id IS NULL; --  
.....UNION SELECT '2  
%00  
/*...*/
```

XSS

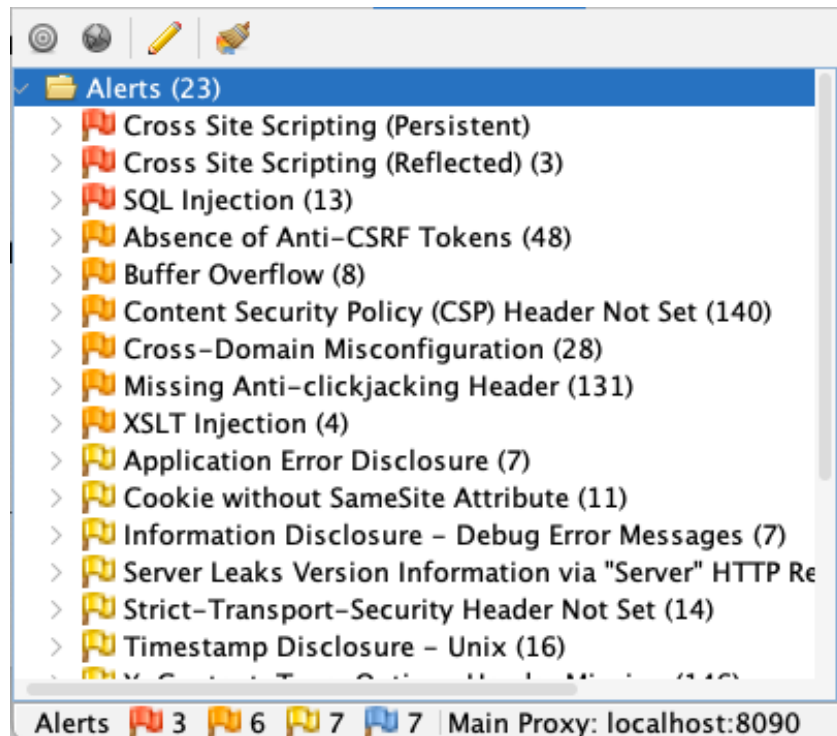
```
"onclick=prompt(8)>"@x.y  
"onclick=prompt(8)><svg/onload=prompt(8)>"@x.y  
<image/src/onerror=prompt(8)>  
<img/src/onerror=prompt(8)>  
<image src/onerror=prompt(8)>  
<img src/onerror=prompt(8)>  
<image src =q onerror=prompt(8)>  
<img src =q onerror=prompt(8)>  
</scrip</script>t><img src =q onerror=prompt(8)>  
<script\x20type="text/javascript">javascript:alert(1);</script>
```

<https://github.com/payloadbox/sql-injection-payload-list>

<https://github.com/payloadbox/xss-payload-list>

Example with ZAP

- OWASP ZAP

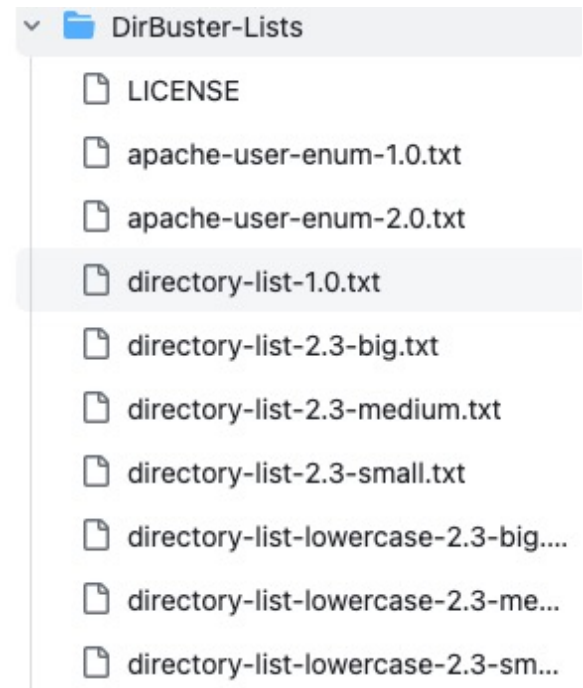


Forced dir browsing

- Bruteforcing directory
 - Wfuzz
 - Is it possible to bypass authorization for pages that require authorization

Target: http://dev1[]i/FUZZ
Total requests: 3036

ID	Response	Lines	Word	Chars	Request
00471:	C=405	139 L	1031 W	26482 Ch	"card"
00706:	C=301	7 L	12 W	184 Ch	"css"
00943:	C=200	846 L	1773 W	35706 Ch	"employees"
01053:	C=301	7 L	12 W	184 Ch	"fonts"
01260:	C=200	866 L	1817 W	36952 Ch	"help"
01272:	C=200	846 L	1773 W	35698 Ch	"history"
01278:	C=302	11 L	22 W	352 Ch	"home"
01304:	C=301	7 L	12 W	184 Ch	"images"
01441:	C=301	7 L	12 W	184 Ch	"js"
01720:	C=302	11 L	22 W	396 Ch	"messages"
02099:	C=403	37 L	65 W	764 Ch	"princess"
02118:	C=302	11 L	22 W	396 Ch	"profile"
02370:	C=200	1941 L	4540 W	89572 Ch	"search"
02382:	C=200	846 L	1787 W	35770 Ch	"security"
02666:	C=500	149 L	1142 W	29309 Ch	"test"
02792:	C=405	139 L	1031 W	26482 Ch	"upload"



Short Demo with ZAP