Finding Security Vulnerabilities in Java Applications with Static Analysis

Benjamin Livshits and Monica S. Lam

Stanford University

SecurityFocus.com Vulnerabilities…

1. PHPList Admin Page SQL Injection Vulnerability
2. Fetchmail POP3 Client Buffer Overflow Vulnerability
3. Zlib Compression Library Buffer Overflow Vulnerability
4. NetPBM PSToPNM Arbitrary Code Execution Vulnerability
5. OpenLDAP TLS Plaintext Password Vulnerability
6. Perl RMTree Local Race Condition Vulnerability
7. Perl Local Race Condition Privilege Escalation Vulnerability
8. Vim ModeLines Further Variant Arbitrary Command Execution Vulnerability
9. Zlib Compression Library Decompression Buffer Overflow Vulnerability
10. Jabber Studio JabberD Multiple Remote Buffer Overflow Vulnerabilities
11. Netquery Multiple Remote Vulnerabilities
12. Multiple Vendor Telnet Client LINEMODE Sub-Options Remote Buffer Overflow Vulnerability
13. Apache mod_ssl SSLCipherSuite Restriction Bypass Vulnerability
14. Multiple Vendor Telnet Client Env_opt_add Heap-Based Buffer Overflow Vulnerability
15. MySQL Eventum Multiple Cross-Site Scripting Vulnerabilities
16. MySQL Eventum Multiple SQL Injection Vulnerabilities
17. AderSoftware CFBB Index.CFM Cross-Site Scripting Vulnerability
18. Cisco IOS IPv6 Processing Arbitrary Code Execution Vulnerability
19. ChurchInfo Multiple SQL Injection Vulnerabilities
20. PHPFreeNews Multiple Cross Site Scripting Vulnerabilities
21. Nullsoft Winamp Malformed ID3v2 Tag Buffer Overflow Vulnerability
22. PHPFreeNews Admin Login SQL Injection Vulnerability
23. Apple Mac OS X Font Book Font Collection Buffer Overflow Vulnerability
24. OpenBook Admin.PHP SQL Injection Vulnerability
25. PowerDNS LDAP Backend Query Escape Failure Vulnerability
26. PowerDNS Recursive Query Denial of Service Vulnerability
27. ProFTPD Shutdown Message Format String Vulnerability
28. ProFTPD SQLShowInfo SQL Output Format String Vulnerability
29. Info-ZIP UnZip Privilege Escalation Vulnerability
30. Trend Micro OfficeScan POP3 Module Shared Section Insecure Permissions Vulnerability

August 1st 2005

Buffer Overrun in zlib (August 1st, 2005)

info | discussion | exploit | solution | references

Zlib Compression Library Buffer Overflow Vulnerability

Zlib is susceptible to a buffer overflow vulnerability. This issue is due to a failure of the application to properly validate input data prior to utilizing it in a memory copy operation.

In certain circumstances, malformed input data during decompression may result in a memory buffer being overflowed. This may result in denial of service conditions, or possibly remote code executing in the context of applications that utilize the affected library.

SecurityFocus.com Vulnerabilities…

1. **PHPList Admin Page SQL Injection Vulnerability**
2. **Fetchmail POP3 Client Buffer Overflow Vulnerability**
3. **Zlib Compression Library Buffer Overflow Vulnerability**
4. **NetPBM PSToPNM Arbitrary Code Execution Vulnerability**
5. OpenLDAP TLS Plaintext Password Vulnerability
6. Perl RMTree Local Race Condition Vulnerability
7. Perl Local Race Condition Privilege Escalation Vulnerability
8. **Vim ModeLines Further Variant Arbitrary Command Execution Vulnerability**
9. **Zlib Compression Library Decompression Buffer Overflow Vulnerability**
10. **Jabber Studio JabberD Multiple Remote Buffer Overflow Vulnerabilities**
11. **Netquery Multiple Remote Vulnerabilities**
12. **Multiple Vendor Telnet Client LINEMODE Sub-Options Remote Buffer Overflow Vulnerability**
13. Apache mod_ssl SSLCipherSuite Restriction Bypass Vulnerability
14. **Multiple Vendor Telnet Client Env_opt_add Heap-Based Buffer Overflow Vulnerability**
15. **MySQL Eventum Multiple Cross-Site Scripting Vulnerabilities**
16. **MySQL Eventum Multiple SQL Injection Vulnerabilities**
17. **AderSoftware CFBB Index.CFM Cross-Site Scripting Vulnerability**
18. Cisco IOS IPv6 Processing Arbitrary Code Execution Vulnerability
19. **ChurchInfo Multiple SQL Injection Vulnerabilities**
20. **PHPFreeNews Multiple Cross Site Scripting Vulnerabilities**
21. **Nullsoft Winamp Malformed ID3v2 Tag Buffer Overflow Vulnerability**
22. **PHPFreeNews Admin Login SQL Injection Vulnerability**
23. **Apple Mac OS X Font Book Font Collection Buffer Overflow Vulnerability**
24. **OpenBook Admin.PHP SQL Injection Vulnerability**
25. PowerDNS LDAP Backend Query Escape Failure Vulnerability
26. PowerDNS Recursive Query Denial of Service Vulnerability
27. **ProFTPD Shutdown Message Format String Vulnerability**
28. **ProFTPD SQLShowInfo SQL Output Format String Vulnerability**
29. Info-ZIP UnZip Privilege Escalation Vulnerability
30. Trend Micro OfficeScan POP3 Module Shared Section Insecure Permissions Vulnerability

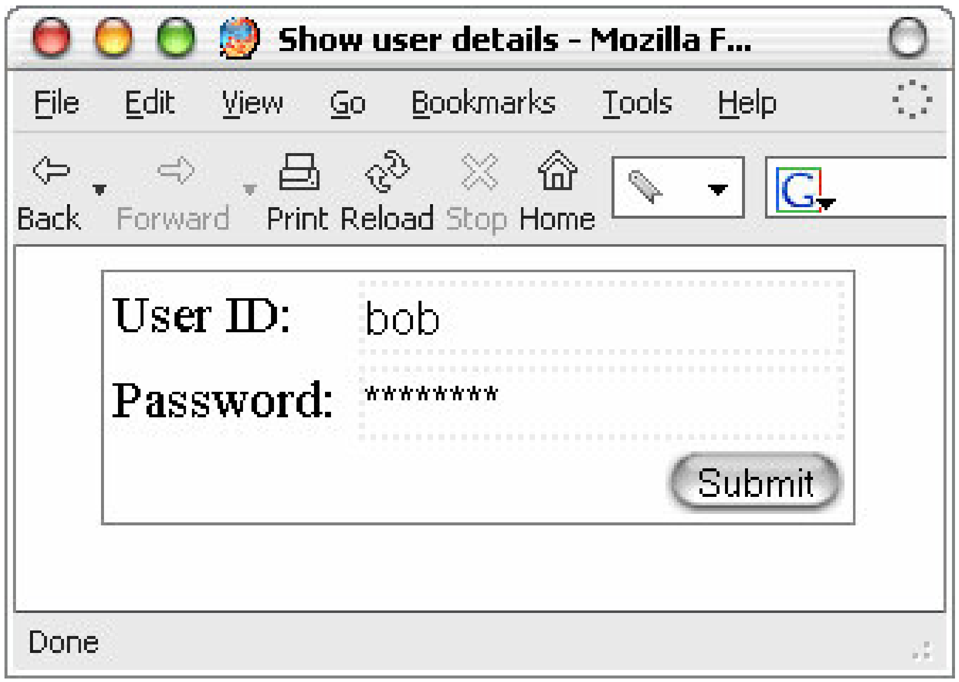
August 1st 2005

22/30=73% of vulnerabilities are due to input validation

Input Validation in Web Apps

- Lack of input validation:
  - #1 source of security errors
- Buffer overruns
  - One of the most notorious
  - Occurs in C/C++ programs
  - Common in server-side daemons
- Web applications are a common attack target
  - Easily accessible to attackers, especially on public sites
  - Java – common development language
  - Many large apps written in Java
    - Modern language – no buffer overruns
    - But can still have input validation vulnerabilities

Simple Web App



- A Web form that allows the user to look up account details
- Underneath – a Java Web application serving the requests

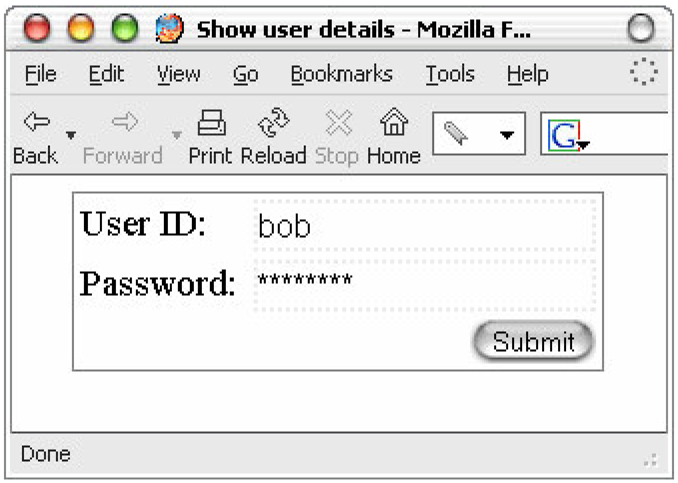
SQL Injection Example

- Happy-go-lucky SQL statement:

```
String query = "SELECT Username, UserID, Password
                FROM Users WHERE
                username =" + user + " AND
                password =" + password;
```

- Leads to **SQL injection**
  - One of the most common Web application vulnerabilities caused by lack of input validation
- But how?
  - Typical way to construct a SQL query using string concatenation
  - Looks benign on the surface
  - But let's play with it a bit more…

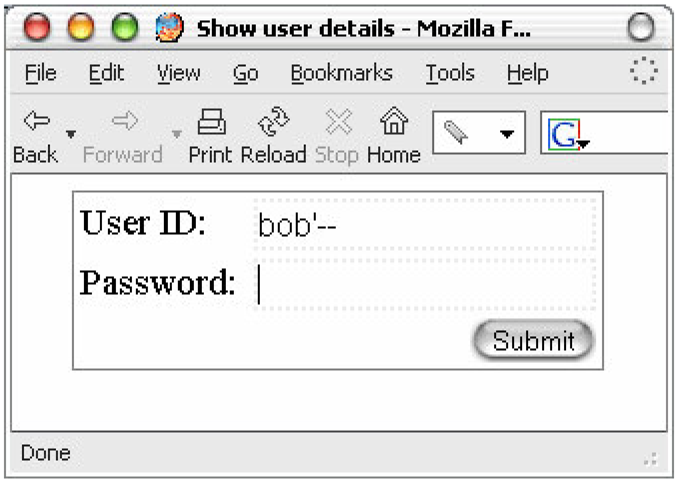
Injecting Malicious Data (1)



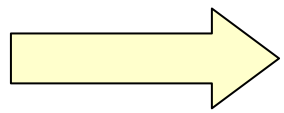
Press “Submit”

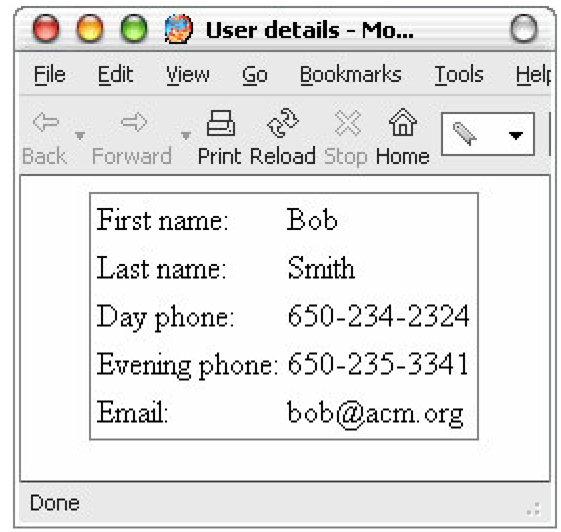

```
query = “SELECT Username,
    UserID, Password
    FROM Users WHERE
        Username = 'bob'
        AND Password = ‘********‘”
```

Injecting Malicious Data (2)



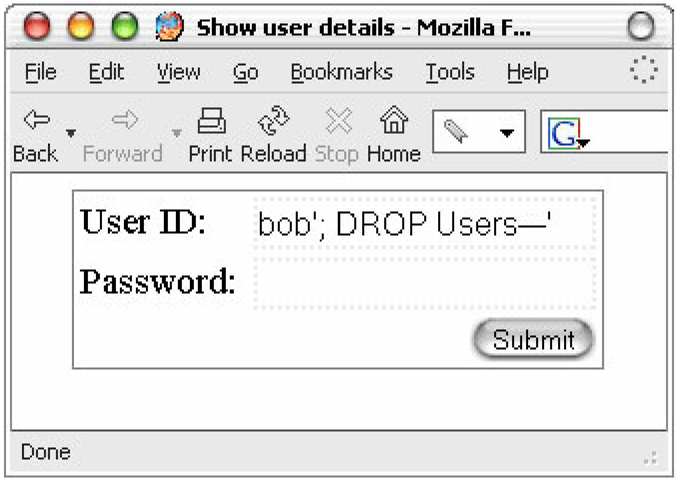
Press “Submit”






```
query = “SELECT Username,
      UserID, Password
      FROM Users WHERE
           Username = 'bob’--
           ’ AND Password = “”
```

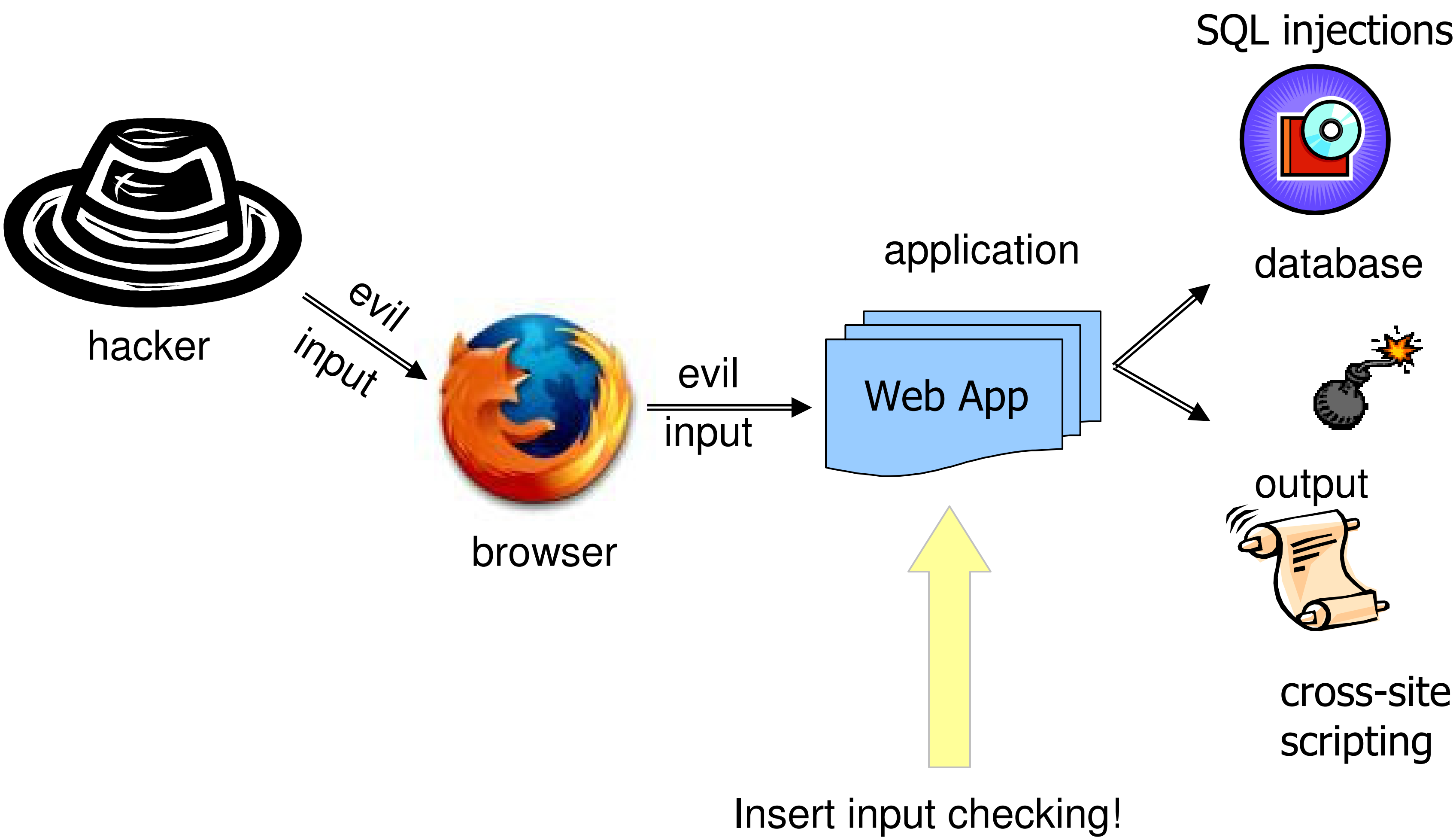
Injecting Malicious Data (3)



Press “Submit”

query = “SELECT Username,
UserID, Password
FROM Users WHERE
Username = 'bob’; DROP Users--
’ AND Password = ‘‘”

Heart of the Issue: Tainted Input Data



Attacks Techniques

1. Inject (taint sources)

- Parameter manipulation
- Hidden field manipulation
- Header manipulation
- Cookie poisoning

2. Exploit (taint sinks)

- SQL injections
- Cross-site scripting
- HTTP request splitting
- Path traversal
- Command injection

1. Header manipulation + **2**. HTTP splitting = vulnerability

- See the paper for more information on these

Related Work: Runtime Techniques

- ~~Client-side validation~~
  - Done using JavaScript in the browser
  - Can be easily circumvented!
- ~~Runtime techniques (application firewalls)~~
  - Input filters – very difficult to make complete
  - Don't work for many types of vulnerabilities

Related Work: Static Techniques

- Manual code reviews
  - Effective – find errors before they manifest
  - Very labor-intensive and time-consuming

Automate code review process with static analysis

- Automatic techniques
  - Metal by Dawson Engler's group at Stanford
  - PreFix used within Microsoft
- Unsound!
  - May miss potential vulnerabilities
  - Can never **guarantee full security**

Develop a sound analysis

Summary of Contributions

Unification:
Formalize existing vulnerabilities within a unified framework

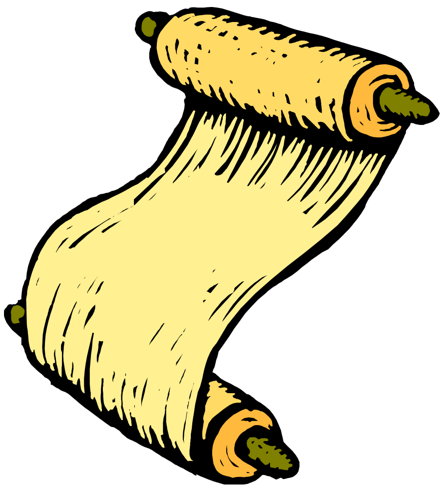
Extensibility:
Users can specify their own new vulnerabilities

Soundness:
Guaranteed to find all vulnerabilities captured by the specification

Precision:
Introduce static analysis improvements to further reduce false positives

Results:
Finds many bugs, few false positives

Why Pointer Analysis?



- Imagine manually auditing an application
- Two statements somewhere in the program

```
// get Web form parameter
String param = request.getParameter(…);
```

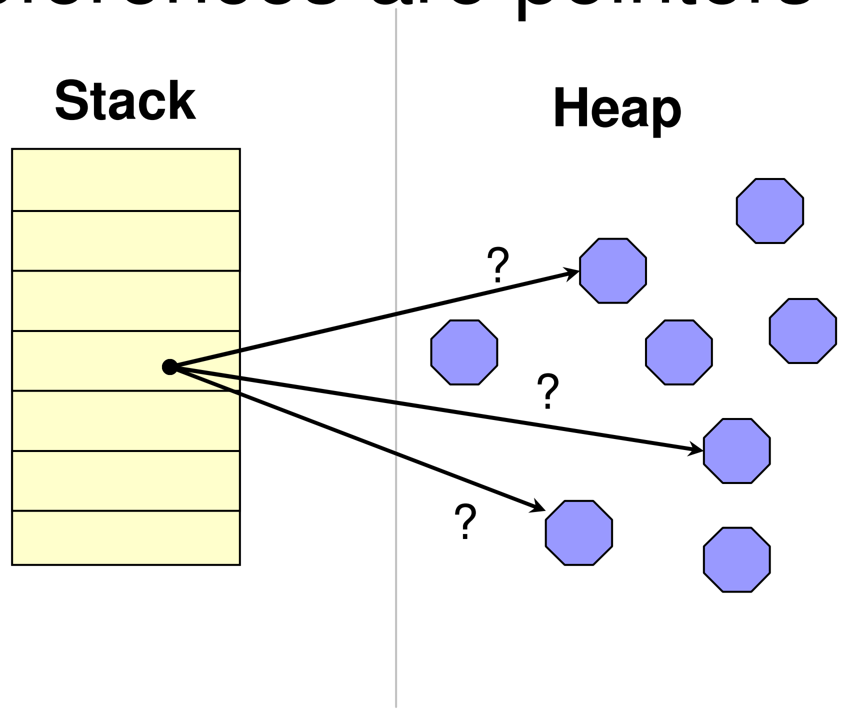
Can these variables refer to the same object?

Question answered by pointer analysis

```
// execute query
con.executeQuery(query);
```

Pointers in Java?

- Yes, remember the `NullPointerException` ?
- Java references are pointers in disguise



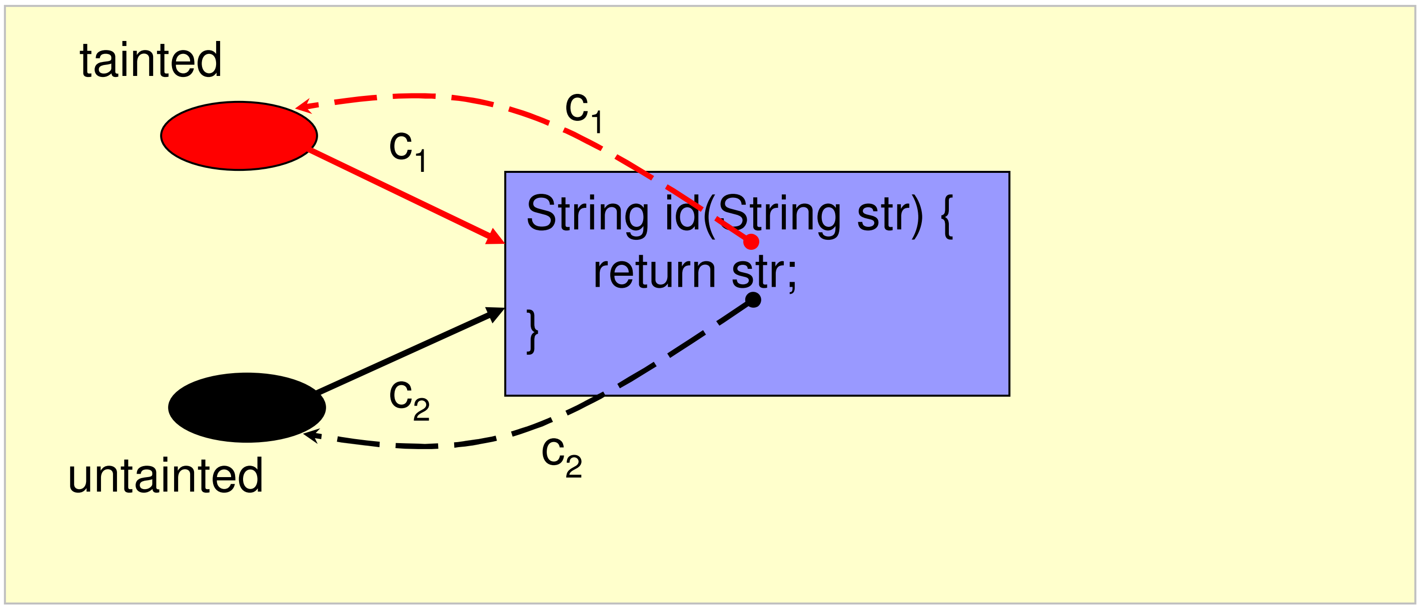
What Does Pointer Analysis Do for Us?

- Statically, the same object can be passed around in the program:
  - **Passed in** as parameters
  - **Returned** from functions
  - **Deposited** to and **retrieved** from data structures
  - All along it is referred to by different **variables**
- Pointer analysis "summarizes" these operations:
  - Doesn't matter what variables refer to it
  - We can follow the object throughout the program

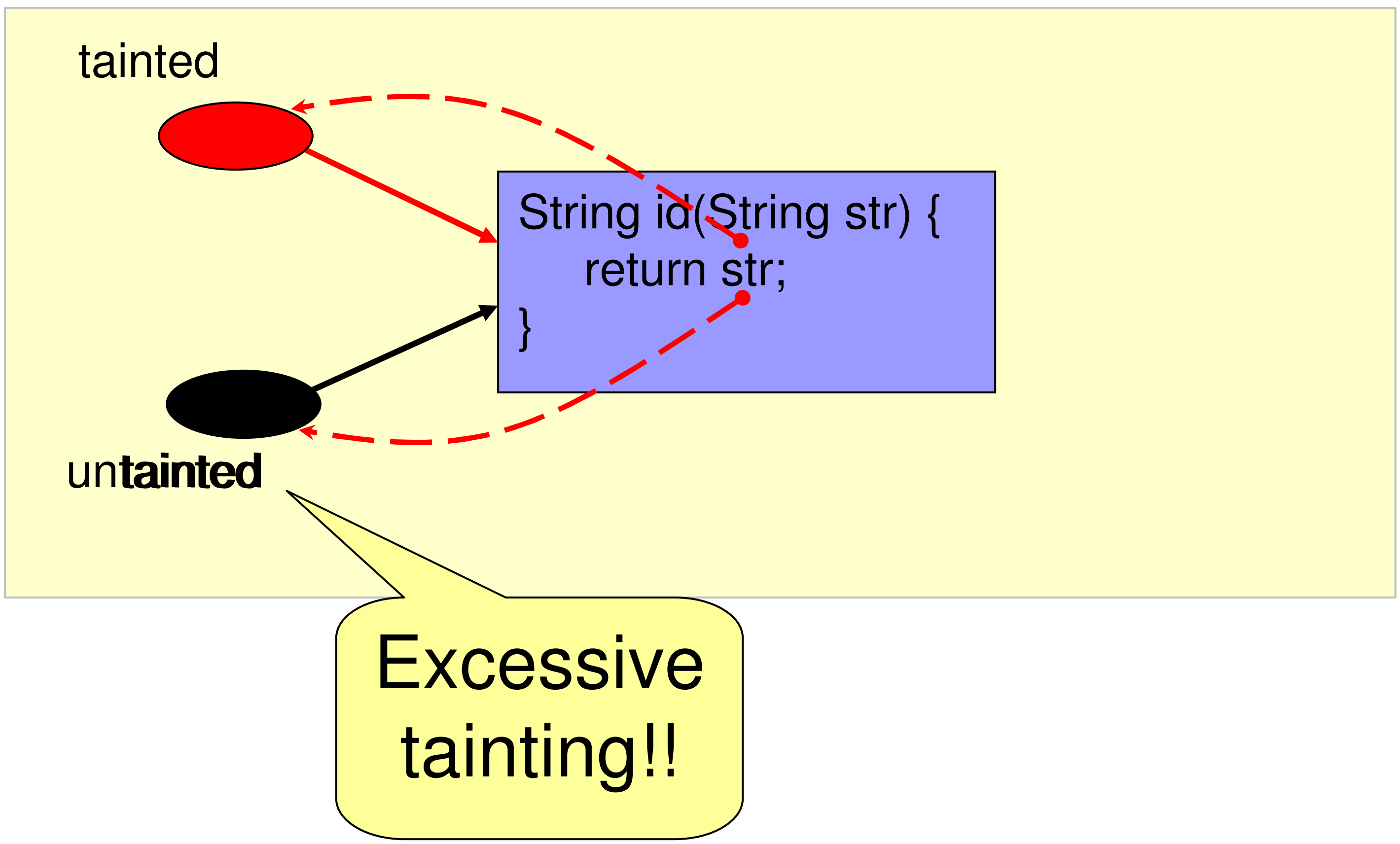
Pointer Analysis Background

- Question:
  - Determine what **objects** a given **variable** may refer to
  - A classic compiler problem for over 20 years
- Our goal is to have a sound approach
  - If there is a vulnerability at runtime, it **will** be detected statically
  - **No** false negatives
- Until recently, sound analysis implied lack of precision
  - We want to have both **soundness** and **precision**
- Context-sensitive inclusion-based analysis by Whaley and Lam [PLDI'04]
  - Recent breakthrough in pointer analysis technology
  - An analysis that is both scalable and precise
  - Context sensitivity greatly contributes to the precision

Importance of Context Sensitivity (1)



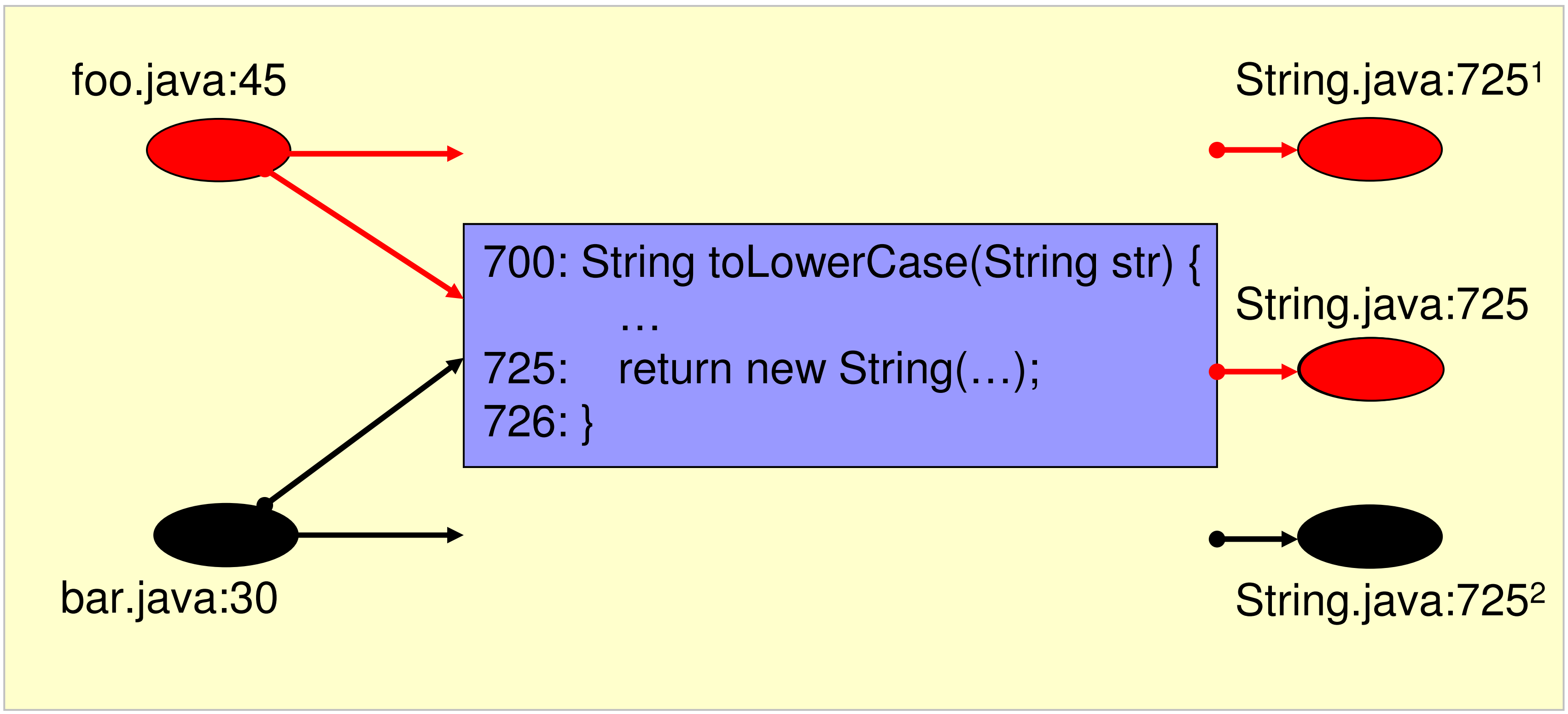
Importance of Context Sensitivity (2)



Pointer Analysis Object Naming

- Need to do *some* approximation
  - **Unbounded** number of dynamic objects
  - **Finite** number of static entities for analysis
- Allocation-site object naming
  - Dynamic objects are represented by the line of code that allocates them
  - Can be imprecise – two dynamic objects allocated at the same site have the same static representation

Imprecision with Default Object Naming



Improved Object Naming

- We introduced an enhanced object naming
  - Containers – HashMap, Vector, LinkedList, etc.
  - Factory functions
- Very effective at increasing precision
  - Avoids false positives in all apps but one
  - All false positives caused by a single factory method
  - Improving naming further gets rid of **all** false positives

Specifying Vulnerabilities

- Many kinds of input validation vulnerabilities
  - Lots of ways to inject data and perform exploits
  - New ones are emerging
- Give the power to the user:
  - **Allow the user** to specify vulnerabilities
  - Use a query language PQL [OOPSLA'05]
- User is responsible for specifying
  - Sources – cookies, parameters, URL strings, etc.
  - Sinks – SQL injection, HTTP splitting, etc.

SQL Injections in PQL

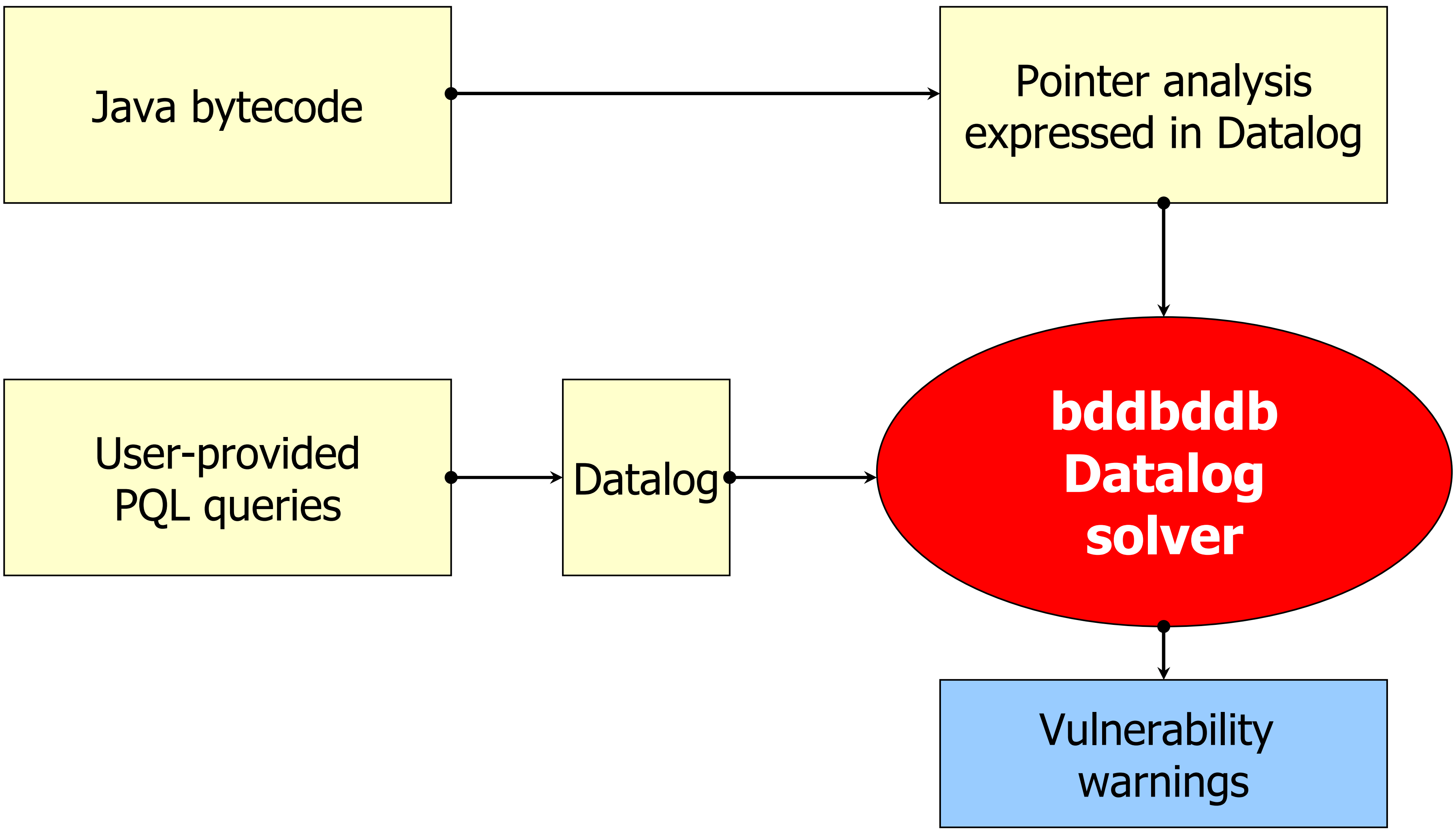
- Simple example
  - SQL injections caused by parameter manipulation
  - Looks like a code snippet
- Automatically translated into static analysis
- Real queries are longer and more involved
- Please refer to the paper

```
query simpleSQLInjection
  returns
      object String param, derived;
  uses
      object HttpServletRequest  req;
      object Connection          con;
      object StringBuffer        temp;
  matches {
      param   = req.getParameter(_);

      temp.append(param);
      derived = temp.toString();

      con.executeQuery(derived);
  }
```

System Overview



Benchmarks for Our Experiments

- Benchmark suite: Stanford SecuriBench
  - We made them publicly available:
    - Google for Stanford SecuriBench
  - Suite of nine large open-source Java benchmark applications
  - Reused the same J2EE PQL query for all
- Widely used programs
  - Most are blogging/bulletin board applications
  - Installed at a variety of Web sites
  - Thousands of users combined

Classification of Errors

Sinks / Sources	SQL injection	HTTP splitting	Cross-site scripting	Path traversal	Total
Header manipulation	0	6	4	0	**10**
Parameter manipulation	6	5	0	2	**13**
Cookie poisoning	1	0	0	0	**1**
Non-Web inputs	2	0	0	3	**5**
Total	**9**	**11**	**4**	**5**	**29**

Classification of Errors

Sources \ Sinks	SQL injection	HTTP splitting	Cross-site scripting	Path traversal	Total
Header manipulation	0	6	4	0	10
Parameter manipulation	6	5	0	2	13
Cookie Poisoning	1	0	0	0	1
Non-Web inputs	2	0	0	3	5
Total	9	11	4	5	29

Classification of Errors

Sources \ Sinks	SQL injection	HTTP splitting	Cross-site scripting	Path traversal	Total
Header manipulation	0	6	4	0	**10**
Parameter manipulation	6	5	0	2	**13**
Cookie poisoning	1	0	0	0	**1**
Non-Web inputs	2	0	0	3	**5**
Total	**9**	**11**	**4**	**5**	**29**

- Total of 29 vulnerabilities found
- We're are sound: all analysis versions report them
- Refer to the paper for more details

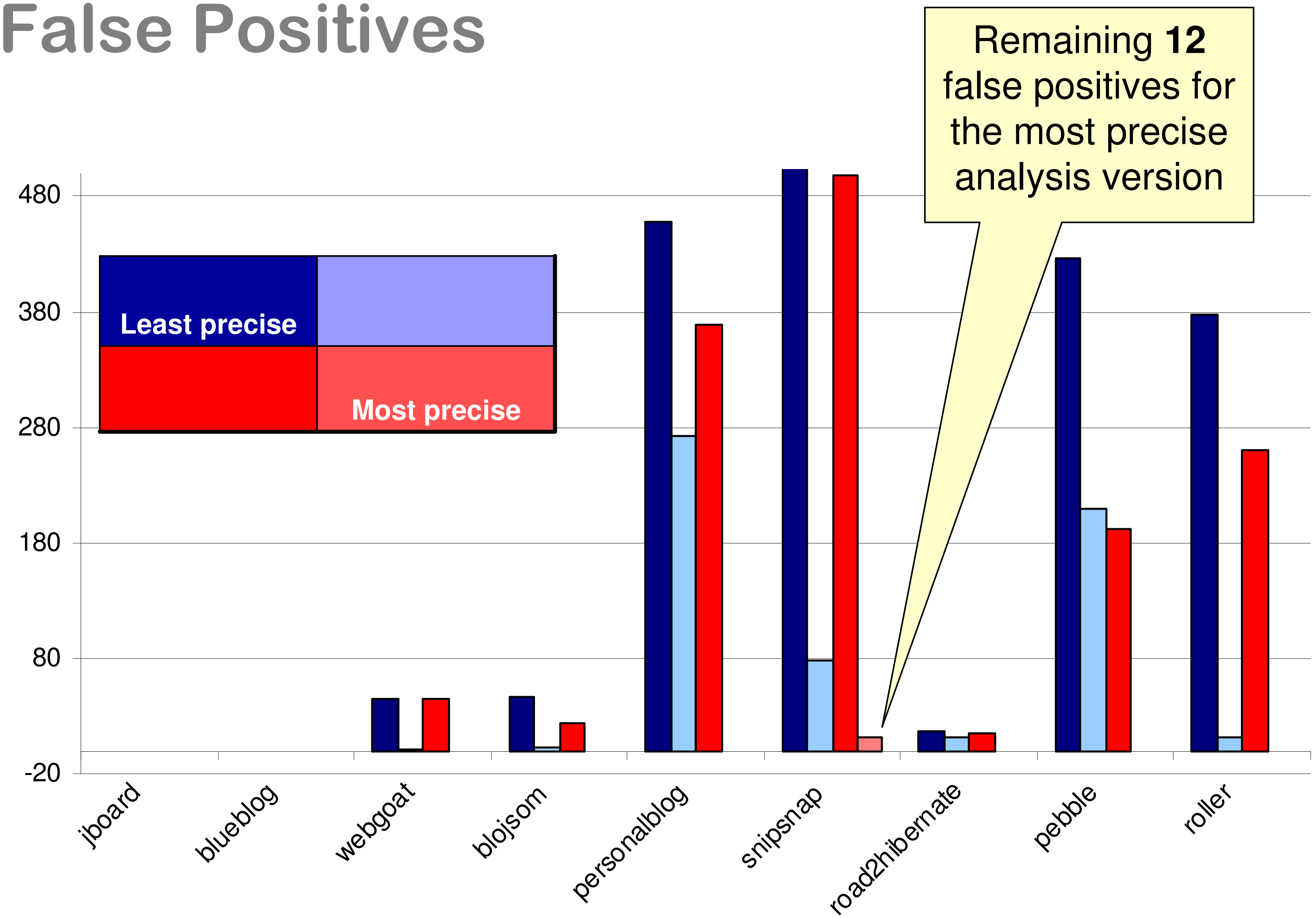
Validating the Vulnerabilities

- Reported issues back to program maintainers
  - Most of them responded
  - Most reported vulnerabilities confirmed as exploitable
- More that a dozen code fixes
- Often difficult to convince that a statically detected vulnerability is exploitable
  - Had to convince some people by writing exploits
  - Library maintainers blamed application writers for the vulnerabilities

Analysis Version Compared

	Default object naming	Improved object naming
Context-insensitive	**Least precise**	
Context-sensitive		**Most precise**

False Positives



Conclusions

A static technique based on a CS pointer analysis

for finding input validation vulnerabilities

in Web-based Java applications

- Results:
  - Found 29 security violations
  - Most reported vulnerabilities confirmed by maintainers
  - Only 12 false positives with most precise analysis version

Project Status

- For more details, we have a TR
  - http://suif.stanford.edu/~livshits/tr/webappsec_tr.pdf

- Stanford SecuriBench recently released
  - http://suif.stanford.edu/~livshits/securibench

- SecuriFly: preventing vulnerabilities on the fly
  - Runtime prevention of vulnerabilities in Web apps
  - See Martin, Livshits, and Lam [OOPSLA'05]