Mitigating Program Security Vulnerabilities: Approaches and Challenges

H. Shahriar and M. Zulkernine
School of Computing
Queen’s University, Kingston, Canada
http://cs.queensu.ca/~shahriar

For ACM SAC 2011 Attendees Only

Background

- Programs are accessible to both legitimate users and hackers
- Ideally, a program should conform to expected behaviors
  - Provide desired outputs based on supplied inputs
- In practice, a program's behavior is unexpected
  - Program crashes or leaks sensitive information

Unexpected program behavior

- A very long URL in the address bar of Microsoft Internet Explorer (6.0) (Source: CVE 2006-3869)
  - Cause
    - Internet Explorer fails due to buffer overflow
  - Impact
    - With a specially crafted request, an attacker can execute arbitrary code

Source: Common Vulnerabilities and Exposures (CVE) http://cve.mitre.org

Information deletion

“The existence of an SQL injection attack on the RIAA’s site came to light via social network news site reports, upon which hackers turned the site into a blank slate, among other things.”

“The RIAA has restored RIAA.org, although whether it's any more secure than before remains open to question” – January 2008

Source: http://www.theregister.co.uk/2008/01/21/riaa_hacktivism/
Information modification

**XSS flaw makes PM say: “I want to suck your blood”**

By Lam Tung, ITNews Australia | 2007/10/9 16:52:02

Topics: cross-site scripting, PM, news, travel, security, XSS

The web sites of Australia’s two major political parties contain cross-site scripting (XSS) flaws, which could be exploited to fraudulently acquire political donations, say security experts.

A short line of script developed by a security enthusiast, Black, caused the Liberal Party’s Web site to read “John Howard says: I want to suck your blood.” While another script caused a window to pop up on the Labor Party’s Web site, urging viewers to “Vote Blue.”

Col Bonnita, security expert at Sunbelt Brekning, said although the vulnerabilities on each party’s Web sites have been exploited for comedic purposes, it would be possible to use the script to fraudulently target people for political donations.

“Web sites of Australia’s two major political parties contain cross-site scripting (XSS) flaws” – October 2007


More list of incidents

- Common Vulnerabilities and Exposures (CVE) http://cve.mitre.org
- Open Source Vulnerability Database (OSVDB), http://osvdb.org
  - Attack method
  - Outcome
  - Location
  - ...

What is wrong in these programs?

- These programs are developed and tested by the brightest engineers in the industry
- Some interesting aspects might be missing
  - Program security vulnerability mitigations in pre and post deployment phases
  - Ensure that programs cannot be exploited with malicious inputs

Tutorial objectives

- Provide an overview of common program security vulnerability exploitations (Part I)
- Introduce mitigation techniques
  - Program security testing (Part II)
  - Static analysis (Part III)
  - Monitoring (Part IV)
  - Hybrid analysis (Part V)
Part I: Program security vulnerability

- Vulnerabilities are flaws in programs that allow attackers to expose or alter sensitive information (Dowd et al. 2007)
- Exploitation of vulnerabilities are attacks
- Common examples of vulnerabilities
  - Buffer Overflow (BOF)
  - Format String Bug (FSB)
  - SQL Injection (SQLI)
  - Cross Site Scripting (XSS)

Buffer overflow (BOF)

- Specific flaws in implementations that allow attackers to overflow data buffers
- Consequence of BOF attacks
  - Program state corruption
  - Program crash due to overwriting of sensitive neighboring variables such as return addresses

Buffer overflow example

```c
1. void foo (int a) {
2.     int var1;
3.     char buf [8];
4.     strcpy (buf, "AAAAAAAAAAAAAA");
      ... return;
    }
```

Sensitive neighboring variables

```
buf [0] ... var1 sfp ret a
<= [0x41 ... 0x41][0x41][0x41][0x41][0x41][0x41][0x41]
top of stack
```

Stack layout of foo function (adapted from [2])

sfp: Stack frame pointer. ret: return address of foo function

Format string bug (FSB)

- Invocation of format functions with non-validated user supplied inputs which contain format strings [3]
  - E.g., format functions of ANSI C standard library
- Consequences
  - Arbitrary reading and writing to format function stacks
Format string bug example

```c
printf(“hello %d %s”, i, str) [GOOD]  printf(“hello %d %s”) [BAD]
```

Stack of the `printf` function call (adapted from [3])

SQL Injection (SQLI) vulnerabilities

- SQL queries generated with user supplied inputs that are not properly validated

- Consequences of SQLI vulnerabilities
  - Authentication bypassing
  - Leakage of private information

SQL Injection (SQLI) vulnerabilities

- SQL queries generated with user supplied inputs that are not properly validated

- Consequences of SQLI vulnerabilities
  - Authentication bypassing
  - Leakage of private information

Legitimate access to an account (step 1)

```
http://localhost880/bookstore/Login.jsp
```

Snapshot of Online BookStore [6]

© Shahriar and Zulkernine, 2011 2011, ACM SAC-15

Legitimate access to an account (step 2)

```
http://localhost880/bookstore/ShoppingCart.jsp
```

© Shahriar and Zulkernine, 2011 2011, ACM SAC-16
Access to an account through an SQLI attack (step 1)

Example of JSP code with SQLI vulnerability*

```java
1. String LoginAction (HttpServletRequest request, ...) throws IOException {
2.    String sLogin = getParam (request, "Login");
3.    String sPassword = getParam (request, "Password");
4.    java.sql.ResultSet rs = null;
5.    String qry = "select member_id, member_level from members where ";
6.    qry = qry + "member_login =" + sLogin + " and member_password =" + sPassword + ";
7.    java.sql.ResultSet rs = stat.executeQuery (qry);
8.    if (rs.next ()) { // Login and password passed
9.        session.setAttribute ("UserID", rs.getString(1));
10.   }
```

Tautology attack (arbitrary condition or true = true)

Some common SQLI attacks (Halfond et al. 2006)

- Tautologies
  - ' or 1=1 --
  - 'greg' like '%gr%'

- UNION queries
  - ' union select 1,1 --

- Piggybacked queries
  - '; show tables; --
Cross Site Scripting (XSS) vulnerabilities

- The generation of HTML contents with invalidated inputs
- Inputs are interpreted by browsers as valid HTML tags
  - The intended behavior of generated web pages alters
  - Visible symptoms (pop-up window)
  - Invisible symptoms (reading form fields, cookies)

XSS attack types

- Stored: Injected code is stored by server programs and used to generate response pages later
- Reflected: Injected code is directly sent back to browsers by server programs
- DOM-based: Injected code is processed by the client-side JavaScript without sanitization

Posting a benign comment (step 1)

Viewing a benign comment (step 2)
Posting a malicious comment (step 1)

Viewing a malicious comment (step 2)

Viewing a malicious comment (step 3)

A reflected XSS attack example

(a) Website offers personalized views based on username
(b) The web page displays a greeting message
(c) Replace the value of username with JavaScript code
Part I: Summary

- Vulnerabilities exist in programs due to lack or weak input validation and logic errors
- Vulnerabilities result in observable abnormal behaviors
  - BOF, FSB, SQLI
- Some vulnerabilities result in unobservable behaviors
  - XSS

Part II: Program security testing

- Classification features of security testing approaches
- Test case generation techniques

References


Program security testing

- Why program security testing is ignored?
  - Program testing cost up to 50% of a project (Sommerville 2001)
  - Time to market pressure
  - Lack of in depth knowledge among related professionals on how malicious inputs alter expected program behaviors

- Program security testing is a misunderstood task (McGraw et al. 2004)
  - Is it port scanning and packet analysis?
  - Is it examining firewall policies?
Security testing steps

- **Test requirement**
  - What aspect of program we want to test?
    - Special classes of bugs that may cause security breaches
  - How these bugs are introduced in programs?
    - Implementation language features
    - Library APIs (ANSI C library)
    - Inputs (network protocol data unit)

- **Test coverage**
  - Does a test suite reveal specific vulnerabilities?
  - Example coverage of program security testing
    - A test suite should reveal all BOF and SQLI vulnerabilities

---

Security testing steps (cont.)

- **Test cases are generated from program and environment**
  - E.g., Source code, library APIs
  - How to define the oracle for each test case?
    - Program states and response messages are used to identify the presence or absence of attacks

- **Test cases are run against implementations**
  - The presence of vulnerabilities are confirmed by comparing program states or outputs with known malicious states or outputs under attacks

---

Classification of security testing approaches*

- **Test case generation method**
- **Source of test case**
- **Test case input type**

---

Classification feature: Test case generation method

- **Example techniques**
  - Fault injection
  - Attack signature
  - Mutation analysis
  - Search-based

- **We will discuss more on these techniques later**

---

*Shahriar and Zulkernine 2011, ACM CSUR*
Classification feature: Source of test case

- Artifacts that are used for generating test cases
  - Program artifacts
    - Vulnerable APIs (ANSI C library functions)
    - Runtime states (registers or variable values)
  - Inputs
    - Valid files, protocol data units (PDUs)
  - Environments
    - File systems, networks, and processors

Classification feature: Test case input type

- It varies based on data types and vulnerabilities
  - Data received by programs
    - Exploiting BOF involve generating strings of particular lengths
    - Exposing FSB requires strings containing format specifiers
  - Vulnerabilities
    - A stored XSS requires two URLs (storing a malicious script and downloading the script)

Test case generation

- Test case generation method
  - Fault injection
  - Attack signature
  - Mutation analysis
  - Search-based

Test case generation: Fault injection

- One of the most widely used test case generation techniques
  - Suitable for black box-based testing
  - Widely used to discover BOF and FSB vulnerabilities
- The objective
  - Corrupt input data and variables
  - Supply to executable programs
  - Observe unexpected responses to conform vulnerabilities
- Fault injection can be divided into three types based on the target of corruption
  - Input
  - Environment
  - Program state
Test case generation: Fault injection (cont.)

- Fault injection in Input data
  - Malform valid data structure into an invalid one
  - E.g., lexical structure modification (replace a non printable character with a printable character)
  - E.g., semantic structure modification (replace the date format "mmddyyyy" to "yyyyddmm")

- Fault injection in environment
  - Modify the attributes of program inputs at different execution stages
  - E.g., during program execution, delete a file or toggle file permissions

- Fault injection in program state
  - Test whether program code can handle malformed states or not
  - E.g., data variables that control program execution flow

Test case generation: Attack signature

- Widely used in security testing of web-based programs

- Replace some parts of normal inputs with attack signatures
  - Signatures are developed from the experience and vulnerability reports
  - SQLI: http://www.xyz.com/login.php?name=' or 1=1 &pwd=
  - XSS: http://www.xyz.com/login.php?name='><script>...</script>&pwd=

- Two ways to traverse web pages
  - Request and response
  - Use case:

Test case generation: Attack signature (cont.)

- Request and response-based method
  - Requests web pages, input fields are filled with attack inputs, and submitted to websites
  - Vulnerability presence is checked by confirming known response messages under attacks
  - Widely used to discover SQLI and XSS vulnerabilities

- Request and response-based method fails to reach inner logics of programs where vulnerabilities might be present

- Use case-based method can alleviate the limitation
  - Relies on interactive user inputs to perform functionalities
  - Generates a collection of user inputs (or URLs)
  - The inputs are replayed back and malicious inputs are injected at specific points of URLs
Test case generation (cont.)

- Test case generation method
  - Fault injection
  - Attack signature
  - Mutation analysis
  - Search-based

Test case generation: Mutation analysis

- Testing of test cases
  - Assessment of test suite adequacy to reveal faults

- Modifies the source code of programs to create mutants

- A mutant is killed if at least one test case generates different output between an implementation and a mutant

- If no test case can kill a mutant, it is said to be equivalent of the original implementation

- Compute the adequacy with mutation score (MS)
  - \( \frac{\text{Total # of mutants killed}}{\text{Total # of non-equivalent mutants}} \)

Test case generation: Mutation analysis (cont.)

- Example of mutation analysis
  - Relational operator replacement (ROR)

<table>
<thead>
<tr>
<th>Original program</th>
<th>Mutant</th>
</tr>
</thead>
</table>
  | 1. int foo (int a, int b){
  | 2. if (a > b) |
  | 3. return (a – b);
  | 4. else |
  | 5. return (a + b); |
  |} | 1. int foo (int a, int b){
  | 2. if (a ≥ b) ///ΔROR |
  | 3. return (a – b);
  | 4. else |
  | 5. return (a + b); |
  |}

  - Before killing the mutant: \( T \{(3, 2)\} \)
  - After killing the mutant: \( T \{(3, 2), (4, 4)\} \)

Test case generation: Mutation analysis (cont.)

- Mutation-based security testing
  - Define mutation operators to inject vulnerabilities
  - Identify mutant killing criteria to generate test cases

- The mutation operators are related to attack type coverage

- The killing criteria specify how to compare outputs between a program and a mutant
  - The end output between an program and a mutant (strong)
  - The intermediate program states between a program and a mutant (weak)
  - At any point between the mutated code and the end of a program (firm)
### Test case generation: Mutation analysis (cont.)

<table>
<thead>
<tr>
<th>Category</th>
<th>Operator</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutating library</td>
<td>S2UCP</td>
<td>Replace <code>strncpy</code> with <code>strcpy</code>.</td>
</tr>
<tr>
<td>function calls</td>
<td>S2UCT</td>
<td>Replace <code>strncat</code> with <code>strcat</code>.</td>
</tr>
<tr>
<td></td>
<td>S2UGT</td>
<td>Replace <code>fgets</code> with <code>gets</code>.</td>
</tr>
<tr>
<td></td>
<td>S2USN</td>
<td>Replace <code>snprintf</code> with <code>sprintf</code>.</td>
</tr>
<tr>
<td></td>
<td>S2UVS</td>
<td>Replace <code>vsprintf</code> with <code>vsnprintf</code>.</td>
</tr>
<tr>
<td>Mutating buffer</td>
<td>RSSBO</td>
<td>Replace buffer size with destination</td>
</tr>
<tr>
<td>size arguments</td>
<td></td>
<td>buffer size plus one.</td>
</tr>
<tr>
<td>Mutating format Strings</td>
<td>RFSNS</td>
<td>Replace <code>&quot;%ns&quot;</code> format string with</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&quot;%sa&quot;</code>.</td>
</tr>
<tr>
<td></td>
<td>RFSBO</td>
<td>Replace <code>&quot;%ns&quot;</code> format string with</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&quot;%nsa&quot;</code>, where, <code>m</code> = size of the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>destination buffer plus one.</td>
</tr>
<tr>
<td></td>
<td>RFSBD</td>
<td>Replace <code>&quot;%ns&quot;</code> format string with</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&quot;%nsa&quot;</code>, where, <code>m</code> is the size of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the destination buffer plus <code>Δ</code>.</td>
</tr>
<tr>
<td></td>
<td>RFSIFS</td>
<td>Replace <code>&quot;%sa&quot;</code> format string with</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&quot;%msa&quot;</code>, where <code>n</code> is the size of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the destination buffer.</td>
</tr>
<tr>
<td>Mutating buffer</td>
<td>MBSBO</td>
<td>Increase buffer size by one byte.</td>
</tr>
<tr>
<td>variable sizes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Removing statements</td>
<td>RMNLS</td>
<td>Remove null character assignment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>statement.</td>
</tr>
</tbody>
</table>

### Test case generation: Mutation analysis (cont.)

**Killing criteria**

- \( CE_P ≠ ES_P \) (strong)
- \( CE_P \leq N & ES_P \) (weak)

- \( P \): The original implementation.
- \( M \): The mutant version.
- \( ES_P \): The exit status of \( P \).
- \( E_S_P \): The exit status of \( M \).
- \( N \): Buffer size.

<table>
<thead>
<tr>
<th>String length (src)</th>
<th>Original program (P)</th>
<th>Mutated program (M)</th>
<th>Output (P)</th>
<th>Output (M)</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>char dest [32];</code></td>
<td><code>char dest [32];</code></td>
<td>No crash</td>
<td>No crash</td>
<td>Live</td>
<td></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>...</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>sscanf (src, &quot;%32s&quot;, dest);</code></td>
<td><code>sscanf (src, &quot;%40s&quot;, dest);</code></td>
<td>[ARFSBD]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>256</code></td>
<td><code>As above</code></td>
<td>No Crash</td>
<td>Crash</td>
<td>Killed</td>
<td></td>
</tr>
</tbody>
</table>

### Test case generation: Mutation analysis (cont.)

**An example of mutation analysis for BOF vulnerability**

- Replace `strncpy` with `strcpy`.  
  \( C_1 \) or \( C_2 \)
- Replace `strncat` with `strcat`. 
  \( C_1 \) or \( C_2 \)
- Replace `fgets` with `gets`.  
  \( C_2 \)
- Replace `snprintf` with `sprintf`. 
  \( C_2 \)
- Replace `vsprintf` with `vsnprintf`. 
  \( C_2 \)
- Replace `"%ns"` format string with `"%sa"`. 
  \( C_1 \) or \( C_2 \)
- Replace `"%nsa"` format string with `"%nsa"`, where, `m` = size of the destination buffer plus one. 
  \( C_2 \)
- Replace `"%ns"` format string with `"%nsa"`, where, `m` is the size of the destination buffer plus `Δ`. 
  \( C_2 \)
- Replace `"%sa"` format string with `"%msa"`, where `n` is the size of the destination buffer. 
  \( C_1 \) or \( C_2 \)
- Increase buffer size by one byte. 
  \( C_1 \)
- Remove null character assignment statement. 
  \( C_1 \) or \( C_2 \)
### Test case generation: Mutation analysis (cont.)

**Relationship between BOF attack type and mutation operator**

<table>
<thead>
<tr>
<th>Attack</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overwriting return addresses</td>
<td>All except RSSBO and RFSBO</td>
</tr>
<tr>
<td>Overwriting stack and heap</td>
<td>All the proposed operators except RMNLS</td>
</tr>
<tr>
<td>One byte overflow</td>
<td>RSSBO and RFSBO</td>
</tr>
<tr>
<td>Arbitrary reading of stack</td>
<td>RMNLS</td>
</tr>
</tbody>
</table>

### Test case generation (cont.)

- **Test case generation method**
  - Fault injection
  - Attack signature
  - Mutation analysis
  - Search-based

### Test case generation: Search-based technique

- **Program test input space might be huge**
  - Finding good test cases becomes time consuming
  - E.g., discovering BOF requires generating a large sized strings

- **Search algorithms can be applied to generate test cases**
  - A random input is chosen as the initial solution
  - The solution is evolved unless an objective function (fitness function) value remains unchanged
  - Fitness functions are designed in ways so that good test cases obtain higher values than that of bad test cases

- **Genetic algorithm**
  1. Generate initial population
  2. Assess fitness of each population
  3. Select a subset of population to perform crossover and mutation
  4. Repeat steps 2 and 3 unless maximum iteration is reached or fitness value of population remains unchanged

- **Fitness function**
  - Assess generated test cases based on a numeric score
  - Guides the choosing of test cases to reveal vulnerabilities

- **Mutation operator**
  - Modification of some random elements in the population to allow evolution of solution
Test case generation: Search-based technique (cont.)

Example code

```c
char * foo (int len, char *str){
    char buf[32];
    if (len>0){
        strncpy(buf, str, len); //no checking of buf length
        return buf;
    }else{
        return NULL;
    }
}
```

- Cross over of 10 and 25

<table>
<thead>
<tr>
<th>10 = [001010]</th>
<th>25 = [011001]</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1: 001001 -&gt; 101001 (41)</td>
<td></td>
</tr>
<tr>
<td>M2: 010011 (19)</td>
<td></td>
</tr>
</tbody>
</table>

- Fitness function: F = max(32–len, 0)
- Initial population (len): [5, 10, 25]
  - Let us assume that a member is represented with 6 bits, max. number is 63
  - F(5) = 27, F(10) = 22, F(25) = 7
- Choose chromosomes with lower fitness values, {25, 10, 5}
  - Perform one point cross over on 10 and 25
  - Randomly replace 0 with 1 in the first gene: {41, 19}
  - New population {41, 25, 19, 10, 5}
  - (len=41) is a test case that reveals BOF

Part II: Summary

- Security testing is analogous to software testing, except some changes in defining requirements, coverage, and oracle
- Fault injection is the most widely used technique
  - Provide no information related to vulnerability and code coverage
- Attack signature-based techniques are mainly applied to web-based programs

References


Part III: Static analysis

- Classification of static analysis-based approaches
- Inference techniques
Static analysis: Overview

- Identify potential vulnerabilities by analyzing source code
- Static analysis techniques do not execute program code
  - Ignore related issues such as environment variables and test cases for reaching specific program points

Classification of static analysis approaches*

- Inference type
- Analysis sensitivity
- Completeness
- Soundness

Classification feature: Inference type

- Four common types
  - Tainted data flow-based
  - Constraint-based
  - Annotation-based
  - String pattern matching-based

Classification feature: Analysis sensitivity

- Static analysis approach might
  - Generate false positive warnings to be examined manually
  - Suffer from false negative (i.e., vulnerabilities are unreported)

- Approaches use pre-computed information
  - Reduce both false positive and negative rates
  - E.g., control flow, context, point-to, value range

*Shahriar and Zulkernine 2011, ACM CSUR
Classification feature: Analysis sensitivity (cont.)

- **Control flow**: An analysis applies inference technique based on statement execution sequence
  - Derived from the control flow graph (CFG) of a program unit
- **Context**: Multiple call sites of a function with arguments are analyzed separately

<table>
<thead>
<tr>
<th>Example code</th>
<th>Context insensitive</th>
<th>Context sensitive</th>
</tr>
</thead>
</table>
| void foo (int len, char *str){
  char buf[32];
  if (len>0){
    strcpy(buf, str); //call site1
  }else{
    strcpy(buf, "hello"); //call site2
  }
} | Call site 1 (BOF) Call site 2 (BOF) | Call site 1 (BOF) Call site 2 (No BOF) |

**Point-to**: compute a set of memory objects that a pointer data type might access in program code
- **Flow sensitive**: a program’s control flow is taken into account to compute point-to graph
- **Flow insensitive**: statements can be analyzed in any order while computing point-to graph

**Value range**: computes lower and upper bound of variables
- The information is applied to check vulnerable function calls

<table>
<thead>
<tr>
<th>Code</th>
<th>size (min)</th>
<th>size (max)</th>
</tr>
</thead>
</table>
| void foo (int len, char *str){
  int size;
  char dest [8];
  if (len >0){
    size = 16;
  }
  memcpy (dest, src, size); |
| -∞ | 16 |

**Data flow**: a dataflow graph to represent data dependencies
- Node represents data defined and used in a basic block
- Edge represents control flow among blocks
- Gen[n]: Block n assigns a tainted value to a variable
- Kill[n]: Block n sanitizes or checks a tainted variable
- IN[n] = U OUT[p], p is a predecessor node of n

<table>
<thead>
<tr>
<th>Code</th>
<th>Block</th>
<th>In</th>
<th>Gen</th>
<th>Kill</th>
<th>Out</th>
</tr>
</thead>
</table>
| void foo (char *str){
  int size;
  char dest [8];
  if (str != NULL){ //b1
    size = 16;
    memcpy (dest, src, size);
    printf (dest);
  }else{ //b2
    ...
  }
} | b0(0-2) | str, size, dest | str | size, dest | str, size, dest |

© Shahriar and Zulkernine, 2011 2011, ACM SAC-65
Classification feature: Soundness

- An approach is sound if it generates no false negative warnings
  - Most approaches are not sound

- Unsupported language features such as specific data type
  - E.g., no BOF detection for buffers that are members of union data structures (Wagner et al. 2000)

- Lack of analysis sensitivity introduces false negative
  - E.g., lack of point to analysis (Weber et al. 2001)
  - Limitation of analysis sensitivity (e.g., flow insensitive points to analysis)
    - Some BOF vulnerabilities remain undetected

Classification feature: Soundness (cont.)

- No warning is provided if an analysis is not certain that a vulnerability can be exploited (result interpretation)
  - Reduce false warnings, but suffers from soundness

- Underlying assumptions to limit the scope of vulnerabilities
  - Vulncheck assumes that most BOF vulnerabilities are present in library function calls accepting unsanitized arguments

Classification feature: Completeness

- An approach is said to be complete, if it generates no false positive warnings

- Analysis sensitivity: Inference approaches based on insensitive analysis result in false positive warnings
  - All unsafe library function calls can be warned as BOF vulnerabilities
  - However, unsafe function calls present in infeasible paths cannot be exploited

- Assumption on program code: Approaches assume that programmers write specific patterns of code
  - Excluded code pattern results in false positive warnings
    - E.g., inference based on known loop structures (Evans et al. 2002)
      - `for (i=0, isize; ++) buf[i] = ...`
      - `for(; ;){buf[i] = x; if (...) break;}`

Inference technique

- Tainted data flow-based
- Constraint-based
- Annotation-based
- String pattern matching-based
Inference technique: Tainted data flow

- Input variables are marked as tainted and their propagations are tracked.
- Warnings are generated, if values derived from tainted inputs are used in sensitive operations.
- Some common techniques:
  - Static data type
  - Implicit

Static data type is extended to include tainted or untainted information (Shankar et al. 2001)
- `int main (int argc, char *argv[])`
- `int main (int argc, tainted char *argv[])`

Check if a labeled source or any value derived from a labeled source participates in sensitive operations
- Writing to buffers (BOF)
- Writing to files/consoles (FSB)
- Generating query statements (SQLI)
- Writing HTML contents (XSS)

Compiler type checking techniques can be leveraged to identify expected type deviation for warning generation.

Type checking is guided by a qualifier lattice that represents subtyping relationships among variables.

A subtyping relationship is analogous to the relationship between a class and a subclass:
- If Q is a subclass of P (Q < P), then an object of class Q can be used whenever an object of class P is expected, but not the vice versa.

Example subtyping for FSB: `untainted char < tainted char`
- A variable labeled as "untainted char" results in a warning, if it is assigned with a "tainted char" variable.

Implicit tainting:
- Suitable for languages that require no static type declarations of variables (e.g., PHP)
- Program variables are not explicitly labeled as tainted or untainted
- Tainting is guided based on the nature of vulnerabilities
- E.g., tainted data derived from HTTP request parameters are vulnerable to XSS (Jovanovic et al. 2006)
- Tainted information flow is performed based on a data flow graph
- Identify locations where tainted data reach to sensitive sinks
  - If a variable is sanitized, then it is not considered as tainted.
Inference technique (cont.)

- Tainted data flow-based
- Constraint-based
- Annotation-based
- String pattern matching-based

Constraint-based

- Generate safety constraints from program code
  - E.g., for BOF, len(s) > alloc(s), for all string variables s.
  - Constraints are generated and propagated while traversing a program
  - If a solution can be found for a set of constraints, a warning is raised
  - Constraint solvers compute input values that violate constraints

Inference technique: Constraint-based (cont.)

- Integer range analysis (Wagner et al. 2000)
  - A constraint is expressed by a pair of integer ranges
  - E.g., buffer allocation size and current size of an allocated buffer
  - gets(s) \rightarrow length of s can be in the range [1, \infty]
  - char s[10] \rightarrow allocation size of s is [10, 10]

  Obtained constraints are solved for each buffer declared and accessed in a program

  We obtain a range of allocation of a buffer [a, b]
  - allocation size of a buffer can vary from a bytes to b byte
  - We obtain current size of a buffer [c, d]
  - current size of a buffer can vary from c bytes to d bytes

  Case 1: b > c, non vulnerable buffer is present
  Case 2: b <= c, vulnerable buffer is present
Inference technique: Annotation

- Program code is added with pre and post conditions
  - Annotations are specified at function declaration and statement levels
- Check if post conditions are hold based on pre conditions and program statements
  - E.g., buffers can be used safely in functions

CBT vs. annotation-based technique

- In CBT, safety conditions are generated automatically
- In annotation, the burden of writing conditions is on programmers

Inference technique: Annotation (cont.)

- Language extension: Specify annotations inside tagged comments in implementation (Evans et al. 2002)
  - E.g., foo (/@nonnull/ char *str)
  - Any value passed to the parameter must not be NULL
  - E.g., strcpy(s1, s2) /*@requires maxSet(s1)>= maxRead(s2) @*/
  - The highest lvalue accessed by s1 must be equal or greater than that of s2

Inference technique (cont.)

- Tainted data flow-based
- Constraint-based
- Annotation-based
- String pattern matching-based

Inference technique: String pattern matching

- Program code is tokenized and scanned to identify pattern of strings that represent vulnerable function calls and arguments
  - E.g., format function calls that include non constant format strings and format strings as the last argument

- Executable files are analyzed to detect vulnerabilities (Tevis et al. 2006)
  - Instead of tokens, an approach analyze symbol tables
  - Check for the presence of vulnerable library functions (e.g., strcpy)
Part III: Summary

- Introducing analysis sensitivity results in better detection of vulnerabilities
- Each inference mechanism is valuable
  - Annotation verifies that certain vulnerabilities are not present
  - Constraint-based inference can prioritize vulnerabilities
  - Tainted data flow can detect multiple vulnerabilities

References


Part IV: Monitoring

- Classification of program monitors
- Monitoring aspects

A generic program monitor

A program without monitor
Program takes input, processes it, and generates output

A program with monitor
A monitor adds an extra layer between program and execution environment
Classification features of monitors*  

- Monitoring aspect
- Program state utilization

Classification feature: Monitoring aspect

- Program runtime properties that are checked
  - Program operation
  - Code execution flow and origin
  - Code structure
  - Value integrity
  - Unwanted value
  - Invariant

Classification feature: Monitoring aspect (cont.)

- Program operation: Memory access
  - Strict bound
  - Flexible bound
  - Function call
  - Output generation

- Memory access with strict bound
  - Do not allow read, write, and free operations on invalid memory locations
  - E.g., Tagging of memory bytes for readability and writability status (BOF, Hastings et al. 1992)
  - E.g., Tracking the base and size of all memory objects at runtime (BOF, Jones et al. 1997, Ruwase et al. 2004)

- Function call
  - Examine argument counts and argument lists of functions
  - E.g., match the number of arguments and types with format specifiers in format strings (FSB, Cowan et al. 2001)

- Output generation
  - Program output points are monitored for presence of tainted inputs or values derived from tainted inputs
  - E.g., input data contains format string (FSB, Newsome et al. 2005)

*Shahriar and Zulkernine 2011, Journal of System and Software
Classification feature: Monitoring aspect (cont.)

- Program runtime properties
  - Program operation
  - Code execution flow and origin
  - Code structure
  - Value integrity
  - Unwanted value
  - Invariant

Classification feature: Monitoring aspect (cont.)

- Code execution flow and origin: Check allowable and unallowable control flow and program code location
  - E.g., information flow tracking
  - Data sources (e.g., file) are marked as tainted
  - Tainted data sources should not compute jump locations
  - Detect exploits that change control flows (BOF, FSB, SQLI)

Classification feature: Monitoring aspect (cont.)

- Monitor if code conforms to desired syntactic structure
  - Interpreter executes implemented code and throws exception for injected code (SQLI, Boyd et al. 2005)
    - Add random numbers in legitimate SQL keywords
      - SELECT becomes SELECT123
    - Attackers fail to correctly add random numbers for keywords
      - SELECT
    - De-randomization is performed before being parsed by query processors
      - SELECT123 becomes SELECT
      - SELECT is not recognized by a parser and generates warning
Monitor sensitive program variables or environment values which are modified during attacks

- Sensitive location without modification
- Sensitive location with modification
- Injected value

Sensitive memory location without modification
- Values stored in sensitive memory locations of programs are checked for corruptions
- E.g., the integrity of a function’s return address can be checked to detect a BOF attack

Sensitive memory location with modification
- To avoid guessing the content of memory locations, sensitive values are stored in encrypted form
- E.g., return addresses are XORed with random keys

Injected value: Check the integrity of injected values
- E.g., canary value is injected before a return address (Cowan et al. 1998)
Classification feature: Monitoring aspect (cont.)

- Data is examined for the presence of unwanted values
  - Input value
  - Output value
- Input value: Inputs are checked for black and white listed characters
  - E.g., SQL meta characters, HTML characters, format specifiers (black listed)
  - E.g., JavaScript code implemented in a program (white listed)
- Output value: Check if output reveals sensitive information
  - Opening a connection from browser that includes cookie (XSS, Krida et al. 2005)

Classification feature: Monitoring aspect (cont.)

- Program runtime properties
  - Program operation
  - Code execution flow and origin
  - Code structure
  - Value integrity
  - Unwanted value
  - Invariant

Classification feature: Monitoring aspect (cont.)

- Monitor the violation of desired properties
  - Requires executing a program with a set of normal (non attack) inputs (i.e., profiling)
- A monitor identifies any deviation from the learned invariants (or profiles) during an actual program run
  - E.g., legitimate API invocation sequence (BOF, Han et al. 2007)
Classification feature: Program state utilization

- A program state needs to be compared with a known program state under attack
- Known states are derived through information extraction, addition, or modification of program states
  - Program state utilization
- Information extraction
  - Program code can be analyzed to extract useful information
  - E.g., a list of known JavaScript code (Johns et al. 2007)

Information addition
- Information is added in current program state, which is retrieved and compared with a future program state
  - E.g., the return address of a function is stored in a safe location
  - E.g., a canary is added before a return address (Cowan et al. 1998)

Information modification
- Modify program (e.g., code, memory) to alter program behaviors during attacks
  - E.g., code randomization (Boyd et al. 2005)
  - E.g., reorganization of program variables such as place all buffers before function pointers (Etoh et al. 2005)

Part IV: Summary

- Each monitoring aspect is useful for specific vulnerabilities
- Program operation and value integrity-based monitoring aspects have mainly addressed BOF attacks
- Unwanted value and code structure-based monitoring have addressed SQLi and XSS attacks
- Some approaches monitor fine-grained level
  - Results in high overhead

References

Part V: Hybrid analysis

- Static analysis suffers from false positive warnings
  - Programmers spend significant time for examining warnings

- The combination of static and dynamic analysis (hybrid analysis, Ernst et al. 2003)
  - Static analysis identifies program locations that need to be examined
  - Exploitations of vulnerabilities are confirmed by executing programs with test cases and examining runtime states (dynamic analysis)
    - E.g., runtime value stored in a sensitive program location

Hybrid analysis vs. monitoring

- Similarity
  - Both techniques examine runtime program states
    - States vary such as data structure, memory locations, etc.

- Dissimilarity
  - Dynamic analysis is an active analysis
    - Apply before the release of a program
    - Used in a wide range of applications (profiling, debugging)
  - Monitoring is a passive analysis technique
    - Apply after the release of a program
    - Do not stop a program execution unless there is a deviation of expected program behaviors
    - Used in (attack/intrusion detection)

Classification features of hybrid analysis*

- Static inference
- Static information
- Dynamic checking

Classification feature: Static inference

- Three common static inference techniques
  - Tainted data flow: Discussed in Part III (static analysis)
  - String pattern matching: Discussed in Part III (static analysis)
  - Untainted data flow: The extraction of legitimate information that is valid in sensitive program operations
    - Instruction sets that can define (or assign) the value of a variable (Castro et al. 2006)

- Tainted vs. untainted data flow
  - A untainted data flow-based technique tracks valid data
  - A tainted data flow-based technique tracks suspected data

*Shahriar and Zulkernine 2011, ACM Computing Surveys
Classification feature: Static information

- Indicate the information gathered in a static analysis
  - Depend on vulnerability types and analysis techniques
  - E.g., unsafe library function calls having pointer arguments (BOF, Aggarwal et al. 2006)
  - E.g., trusted strings (SQLI, Halfond et al. 2006)

Classification feature: Dynamic checking

- Program states that are checked to confirm exploitations in a dynamic analysis

Examples of program state checking
- Program operation: Memory allocation, release, and access (e.g., BOF, Castro et al. 2006)
- Code structure: Validate known structure of program code during runtime (e.g., SQLI, Wei et al. 2006)
- Unwanted value: Check the presence of unwanted values in program outputs (e.g., XSS, Lucca et al. 2004)

Part V: Summary

- The precision of static analysis is important and needs to be carefully considered before applying in a hybrid approach
- Most of the works apply string pattern matching and tainted data flow analysis in the static analysis phase
- Program operation and code integrity are widely used in the dynamic analysis phase

References

Conclusions

- Program security vulnerabilities exist in real world
- Each mitigation technique is important and offers unique advantages
  - The classification feature indicates that techniques can address a subset of vulnerabilities