Secure Software Development: Theory and Practice

Suman Jana
MW 2:40-3:55pm
415 Schapiro [SCEP]

*Some slides are borrowed from Dan Boneh and John Mitchell*
Software Security is a major problem!
Why writing secure code is hard?
Software bugs cost US economy $59.5 billion annually (NIST)
Not all bugs are equal!

Benign functional bugs vs. Security bugs

Why are security bugs more dangerous than other bugs?
Why security bugs are more dangerous?

- Security bugs allow attackers to cause serious damages: take over machines remotely, steal secrets, etc.
How do we deal with security bugs?

- Automatically find and fix bugs
- Monitor a system at runtime to detect and prevent exploits of bugs

- Accept that programs will have bugs and design the system to minimize damages
  - Example: Sandboxes, privilege separation
Theory of bug finding
Finding bugs with Program analyzers

![Diagram with Code, Spec/Invariants, Program Analyzer, and a table]

- **Code**
- **Spec/Invariants**
- **Program Analyzer**

**Descriptions of different classes of bugs**

<table>
<thead>
<tr>
<th>Report</th>
<th>Type</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mem leak</td>
<td>324</td>
</tr>
<tr>
<td>2</td>
<td>buffer oflow</td>
<td>4,353,245</td>
</tr>
<tr>
<td>3</td>
<td>sql injection</td>
<td>23,212</td>
</tr>
<tr>
<td>4</td>
<td>stack oflow</td>
<td>86,923</td>
</tr>
<tr>
<td>5</td>
<td>dang ptr</td>
<td>8,491</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10,502</td>
<td>info leak</td>
<td>10,921</td>
</tr>
</tbody>
</table>
Automated bug detection: main challenges

```c
int main (int x, int y)
{
    if (2*y!=x)
        return -1;
    if (x>y+10)
        return -1;
    ....
    ... /* buggy code*/
}
```

1. Too many paths (may be infinite)
2. How will program analyzer find inputs that will reach different parts of code to be tested?
Automated bug detection: two options

• Static analysis
  – Inspect code or run automated method to find errors or gain confidence about their absence
  – Try to aggregate the program behavior over a large number of paths without enumerating them explicitly

• Dynamic analysis
  – Run code, possibly under instrumented conditions, to see if there are likely problems
  – Enumerate paths but avoid redundant ones
Static vs dynamic analysis

• Static
  – Can consider all possible inputs
  – Find bugs and vulnerabilities
  – Can prove absence of bugs, in some cases

• Dynamic
  – Need to choose sample test input
  – Can find bugs and vulnerabilities
  – Cannot prove their absence
Soundness & Completeness

<table>
<thead>
<tr>
<th>Property</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Soundness</td>
<td>“Sound for reporting correctness”</td>
</tr>
<tr>
<td></td>
<td>Analysis says no bugs $\rightarrow$ No bugs</td>
</tr>
<tr>
<td></td>
<td>or equivalently</td>
</tr>
<tr>
<td></td>
<td>There is a bug $\rightarrow$ Analysis finds a bug</td>
</tr>
<tr>
<td>Completeness</td>
<td>“Complete for reporting correctness”</td>
</tr>
<tr>
<td></td>
<td>No bugs $\rightarrow$ Analysis says no bugs</td>
</tr>
</tbody>
</table>

Recall: $A \rightarrow B$ is equivalent to $(\neg B) \rightarrow (\neg A)$
## Soundness & Completeness

<table>
<thead>
<tr>
<th></th>
<th>Complete</th>
<th>Incomplete</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sound</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complete</td>
<td>Reports all errors</td>
<td>Reports all errors</td>
</tr>
<tr>
<td></td>
<td>Reports no false alarms</td>
<td>May report false alarms</td>
</tr>
<tr>
<td>Undecidable</td>
<td><strong>Undecidable</strong></td>
<td><strong>Decidable</strong></td>
</tr>
<tr>
<td>Unsound</td>
<td>May not report all errors</td>
<td>May not report all errors</td>
</tr>
<tr>
<td></td>
<td>Reports no false alarms</td>
<td>May report false alarms</td>
</tr>
<tr>
<td>Decidable</td>
<td><strong>Decidable</strong></td>
<td><strong>Decidable</strong></td>
</tr>
</tbody>
</table>
When to find bugs?

Cost of bug finding

Credit: Andy Chou, Coverity
Practice of bug finding
Popular classes of security bugs

Memory corruption attacks
Memory corruption attacks

- **Attacker’s goal:**
  - Take over target machine (e.g., web server)
    - Execute arbitrary code on target by hijacking application control flow leveraging memory corruption

- **Examples.**
  - Buffer overflow attacks
  - Integer overflow attacks
  - Format string vulnerabilities
What is needed

• Understanding C functions, the stack, and the heap.
• Know how system calls are made
• The exec() system call

• Attacker needs to know which CPU and OS used on the target machine:
  – Our examples are for x86 running Linux or Windows
  – Details vary slightly between CPUs and OSs:
    • Little endian vs. big endian (x86 vs. Motorola)
    • Stack Frame structure (Unix vs. Windows)
Stack Frame

- arguments
- return address
- stack frame pointer
- exception handlers
- local variables

SP → Stack Growth

Stack Frame Diagram
Linux process memory layout

- **%esp**: Points to the top of the user stack.
- **brk**: Points to the beginning of the run-time heap.
- **Loaded from exec**: Represents memory loaded from the executable.
- **Unused**: Empty memory at the bottom.
- **Shared libraries**: Memory containing shared libraries.
- **Run-time heap**: Memory used for dynamic allocation.
- **User stack**: Memory allocated for function call stacks.
What are buffer overflows?

Suppose a web server contains a function:

```c
void func(char *str) {
    char buf[128];
    strcpy(buf, str);
    do-something(buf);
}
```

When `func()` is called stack looks like:

```
SP
  | return address
  | stack frame pointer
  | char buf[128]
  | argument: str
```

What happens if `str` is larger than 128?
Basic stack exploit

Suppose \*str is such that after strcpy stack looks like:

Program P: exec("/bin/sh")

When func() exits, the user gets shell!
Note: attack code P runs in stack.
The NOP slide

Problem: how does attacker determine ret-address?

Solution: NOP slide
- Guess approximate stack state when `func()` is called
- Insert many NOPs before program P:
  - `nop`, `xor eax, eax`, `inc ax`
How to avoid buffer overflows?

- Rewrite software in a type safe language (Java, Rust)
  - Difficult for existing (legacy) code...

- Use safer functions like `strncpy` instead of `strcpy`
  - Developer may make mistakes
  - Confusing semantics for terminating NULL characters

- Automatically find them
  - Static analysis tools: Coverity, CodeSoner...
  - Dynamic analysis tools: AFL, libfuzzer...

More details about detection techniques later in the semester
Structure of the class

1. Control & data flow analysis
2. Symbolic Execution
3. Fuzzing

Program analysis Fundamentals

- Memory corruption attacks
- Web Attacks: XSS, SQL injection, and CSRF
- Semantic/logic bugs
- Side channel leaks
- DOS attack vectors

Build tools for detecting classes of bugs

Different classes of security bugs
Logistics

Class webpage
http://sumanj.info/secure_sw_devel.html

TAs: Eugene Ang and Plaban Mohanty)

Reading
No text book, slides, and one/two papers per class

Grading :
Quizzes/programming assignments - 35%
Midterm - 30%
Group Project (3-4 students) - 30%
Class participation - 5%
Summary

In this class you will learn about:

1. Different classes of security bugs and their implications
2. State-of-the art of bug finding techniques
3. Using and customizing existing bug finding tools