CS161 Midterm 1 Review

Midterm 1: March 4, 18:30-20:00
Same room as lecture
Security Analysis and Threat Model

• Basic security properties
  – CIA

• Threat model
  A. We want perfect security
  B. Security is about risk analysis and economics

Answer is B.
Software Vulnerabilities

• Buffer overflow vulnerabilities and attacks
• Integer overflow vulnerabilities and attacks
• Format string vulnerabilities and attacks
• Arc injection/return-to-libc/ROP vulnerabilities and attacks
• General control hijacking attacks
• Data hijacking attacks
General Control Hijacking

Overwrite Step:
Find some way to **modify** a Control Flow Pointer to point to your shellcode, library entry point, or other code of interest.

Activate Step:
Find some way to **activate** that modified Control Flow Pointer.
### Instances of Control Hijacking

<table>
<thead>
<tr>
<th>Location in Memory</th>
<th>Control Flow Pointer</th>
<th>How to activate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Return Address</td>
<td>Return from function</td>
</tr>
<tr>
<td>Stack</td>
<td>Frame Pointer</td>
<td>Return from function</td>
</tr>
<tr>
<td>Stack</td>
<td>Function Pointers as local variables</td>
<td>Reference and call function pointer</td>
</tr>
<tr>
<td>Stack</td>
<td>Exception Handler</td>
<td>Trigger Exception</td>
</tr>
<tr>
<td>Heap</td>
<td>Function pointer in heap (i.e. method of an object)</td>
<td>Reference and call function pointer</td>
</tr>
<tr>
<td>Anywhere</td>
<td>setjmp and longjmp program state buffer</td>
<td>Call longjmp</td>
</tr>
</tbody>
</table>

**Diagram:**
- Stack Frame
  - Return Address
  - Frame Pointer
  - Function Pointers as local variables
  - Exception Handler
  - Function pointer in heap (i.e. method of an object)

- Exception Handlers
- Local fn_ptrs
- buf
- T ptr
- FP1: FP2: FP3:
- vtable
- method #1 method #2 method #3

- Setjmp
- Longjmp
- Saved pointer
- ...
Data Hijacking

Modifying data in a way not intended  
Example: Authentication variable

Exploited Situation:
User types in a password which is long enough to overflow buffer and into the authentication_variable. The user is now unintentionally authenticated.
Stack and Format Strings

• Function behavior is controlled by the format string
• Retrieves parameters from stack as requested: “%”
• Example:

    printf("Number %d has no address, number %d has: %08x\n", i, a, &a);

    | A  | Address of the format string |
    |----|------------------------------|
    | i  | Value of Variable i         |
    | a  | Value of variable a         |
    | &a | Address of variable a       |

    stack top
    ...
    <&a>
    <a>
    <i>
    A
    ...
    stack bottom
SW Vuln. Defenses

- Non-execute (NX)
- Stack canaries
- ASLR
- Bounds check
- Which defenses are effective against what attacks?
Effectiveness and Limitations

- Defense against buffer overflow attacks

<table>
<thead>
<tr>
<th>Defenses/Mitigations</th>
<th>Code Injection</th>
<th>Arc Injection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Non-Execute (NX)* ASLR StackGuard(Canaries)</td>
<td>ASLR StackGuard(Canaries)</td>
</tr>
<tr>
<td>Heap</td>
<td>Non-Execute (NX)* ASLR</td>
<td>ASLR</td>
</tr>
<tr>
<td>Exception Handlers</td>
<td>Non-Execute (NX)* ASLR</td>
<td>ASLR</td>
</tr>
</tbody>
</table>

* When Applicable
Fuzzing

• Random fuzzing
• Mutation-based fuzzing
• Generation-based fuzzing
• Code coverage
  – line, branch and path coverage
• Example problem: given a program, calculate how many inputs can achieve a full line/branch/path coverage (e.g., Discussion 5)
Coverage Metrics

Lines
Coverage Metrics

Lines
Coverage Metrics

Lines

Branche
s
Coverage Metrics

Lines

Branches
Coverage Metrics

Lines

Branches

Paths
Coverage Metrics

Lines

Branches

Paths
Coverage Metrics

Lines

Branches

Paths
Quiz on Line Coverage

How many lines are in this code?

- 1
- 2
- 3
- 4

How many test cases (pairs of values for (a,b)) are needed to achieve 100% line coverage?

- 1
- 2
- 3
- 4

```c
if (a > 2)
a = 2;
if (b > 2)
b = 2;
```
Quiz on Branch Coverage

How many branches are in this code?

- 1
- 2
- 3
- 4

How many test cases (pairs of values for (a,b) are needed to achieve 100% branch coverage?

- 1
- 2
- 3
- 4

```java
if (a > 2)
a = 2;
if (b > 2)
b = 2;
```
Quiz on Path Coverage

How many paths are in this code?

0 1
0 2
0 3
0 4

How many test cases (pairs of values for (a,b) are needed to achieve 100% path coverage?

0 1
0 2
0 3
0 4

```java
if (a > 2)
    a = 2;
if (b > 2)
    b = 2;
```
Completeness of Coverage Metrics

Which of the following coverage results guarantee the bug will be found?

1. 100% line coverage
2. 100% branch coverage
3. 100% path coverage
4. None of the above

```c
my_copy(char* dst, char* src){
    if (dst && src)
        strcpy(dst, src);
}
```
Properties of Coverage Metrics

- A numeric measure of an analysis
- An objective basis for comparing different analyses
- A way to evaluate if no progress is made (no coverage metrics are increasing)

*Important*: Metrics are not sufficient conditions for completeness. 100% coverage does not mean all sources of vulnerabilities have been evaluated.
Symbolic Execution

• Path predicates
• Security vulnerabilities as assertion violations
• How to use symbolic execution to find bugs
• Constraint-based automatic test case generation
• Challenges for symbolic execution
Assertion Violation as Satisfiability

In the appropriate theory, the formula

\[
\begin{align*}
\text{input} &< \text{UINT_MAX} - 2 \\
\&\& \text{len} &== \text{input} + 3 \\
\&\& \neg (\text{len} < 10) \\
\&\& \neg (\text{len} \% 2 == 0) \\
\&\& \neg (\text{len} < \text{UINT_MAX} - 1)
\end{align*}
\]

is satisfied by the assignment

<table>
<thead>
<tr>
<th>input</th>
<th>UINT_MAX - 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>len</td>
<td>UINT_MAX</td>
</tr>
</tbody>
</table>

C++ Code:

```c
if (input < UINT_MAX - 2) {
    unsigned len, s;
    char* buf;
    len = input + 3;
    if (len < 10)
        printf("true")
    else
        printf("false")
}

s = len;
if (len % 2 == 0)
    printf("true")
else
    printf("false")

s = len;
assert(len < UINT_MAX - 1);
true

s = len + 2;
false

buf = malloc(s);
false
read(fd, buf, len);
err
```
Suppose we want to know if there is a feasible path to the location ERR in this program.

Suppose we generate one path predicate for each path through this program.

How many path predicates are generated?
Suppose we want to know if there is a feasible path to the location ERR in this program.

Suppose we generate one path predicate for each path through this program.

How many path predicates are generated?

$2^n$
Suppose we want to know if there is a feasible path to the location ERR in this program.

Suppose we generate one path predicate for each path through this program.

How many path predicates are generated?

\[ 2^n \]

Number of predicates can be exponential in the number of branches.
Topics Covered in Midterm 2

• Static analysis
• Program Verification
• Security principles and architectures
• Malware
• Other topics after midterm 2