Language Based Security

Lecture 19
Lecture Outline

• Beyond compilers
  – Looking at other issues in programming language design and tools

• C
  – Arrays
  – Exploiting buffer overruns
  – Detecting buffer overruns
Platitudes

• Language design has influence on
  - Safety
  - Efficiency
  - Security

Recall: Platitude: A flat, dull, or trite remark, especially one uttered as if it were fresh or profound
C Design Principles

• Small language
• Maximum efficiency
• Safety less important

• Designed for the world in 1972
  - Weak machines
  - Trusted networks
Arrays in C

#define char buffer[100];

Declares and allocates an array of 100 chars

0 1 2 99

100 * sizeof(char)
C Array Operations

char buf1[100], buf2[100];

Write:
    buf1[0] = 'a';

Read:
    return buf2[0];
What's Wrong with this Picture?

```c
int i = 0;
for (i = 0; buf1[i] != '\0'; i++) {
    buf2[i] = buf1[i];
}
buf2[i] = '\0'
```
Indexing Out of Bounds

The following are all legal C and may generate no run-time errors

```c
char buffer[100];

buffer[-1] = 'a';
buffer[100] = 'a';
buffer[100000] = 'a';
buffer[1000000] = 'a';
```
Why?

- Why does C allow out of bounds array references?
  
  - Proving at compile-time that all array references are in bounds is very difficult (impossible in C)
  
  - Checking at run-time that all array references are in bounds is expensive
Code Generation for Arrays

buf[i] = 1; /* buf1 has type int[] */

r1 = load &buf1;
r2 = load i;
r3 = r2 * 4;
r4 = r1 + r3;
store r4, 1

(note this last is NOT a MIPS instruction)
Discussion

• 5 instructions worst case
• Often &buf1 and i already in registers
  - Saves 2 instructions
• Many machines have indirect load/stores
  - store r1[r3], 1
  - Saves 1 instruction
• Best case 2 instructions
  - Offset calculation and memory operation
Code Generation for Arrays with Bounds Checks

buf[i] = 1; /* buf1 has type int[] */

r1 = load &buf1;
r2 = load i;
r3 = r2 * 4;
if r3 < 0 then error;
r5 = load limit of buf1;
if r3 >= r5 then error;
r4 = r1 + r3;
store r4, 1
Discussion

• Lower bounds check can often be removed
  - Easy to prove statically that index is positive

• Upper bounds check hard to remove
  - Leaves a conditional in instruction stream

• In C, array limits not stored with array
  - Knowing the array limit for a given reference is non-trivial
C vs. Java

• C array reference typical case
  - Offset calculation
  - Memory operation (load or store)

• Java array reference typical case
  - Offset calculation
  - Memory operation (load or store)
  - Array bounds check
  - Type compatibility check (for stores)
Buffer Overruns

• A buffer overrun writes past the end of an array

• Buffer usually refers to a C array of char
  - But can be any array

• So who’s afraid of a buffer overrun?
  - Can damage data structures
  - Cause a core dump
  - What else?
Stack Smashing

Buffer overruns can alter the control flow of your program!

```c
char buffer[100]; /* stack allocated array */
```

0 1 2 99

100 * sizeof(char)
void foo(char buf1[]) {
    char buf2[100];
    int i = 0;
    for (i = 0; buf1[i] != '\0'; i++) {
        "buff2[i] = buf1[i];"
    }
    buf2[i] = '\0';
}
An Interesting Idea

```c
char buf[104] = { ' ',..., ' ', magic 4 chars }
foo(buf); (**)
```

Foo entry

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>99 return address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>(**))</td>
</tr>
</tbody>
</table>

100 * sizeof(char)

Foo exit

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>99 return address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>magic 4 chars</td>
</tr>
</tbody>
</table>

100 * sizeof(char)
Discussion

• So we can make foo jump wherever we like.

• How is this possible?

• Unanticipated interaction of two features:
  - Unchecked array operations
  - Stack-allocated arrays
    • Knowledge of frame layout allows prediction of where array and return address are stored
  - Note the “magic cast” from chars to an int address
The Rest of the Story

• We can make foo jump anywhere.
• But where is a useful place to jump?

• Idea: Put our own code in the buffer and jump there!
The Plan

char buf[104] = { 104 magic chars }
foo(buf);

"exec /bin/sh"

100 * sizeof(char)

99 return address

Foo exit

0 1 2
Details

• “exec /bin/sh”
  - Easy to write in assembly code
  - Make all jumps relative

• Be careful not to have null bytes in the code (why?)
More Details

• Overwrite return address with start of buffer
  - Harder
  - Need to guess where buffer in called routine starts (trail & error)
  - Pad front of buffer with NOPs
    • Guess need not be exact; just land somewhere in NOPs
And More Details

• Overwrite return address
  - Don’t need to know exactly where return address is
  - Just pad end of buffer with multiple copies of new return address X

char buf[104] =
   "NOPS ... exec /bin/sh XXXXXXXXXXXXXXX"
foo(buf)
The State of C Programming

• Buffer overruns are common
  - Programmers must do their own bounds checking
  - Easy to forget or be off-by-one or more
  - Programs still appear to work correctly

• In C wrt to buffer overruns
  - Easy to do the wrong thing
  - Hard to do the right thing
The State of Hacking

- Buffer overruns are the attack of choice (sort of)
  - 40-50% of new vulnerabilities are buffer overrun exploits (though this figure varies)

- Highly automated toolkits available to exploit known buffer overruns
  - Google search for “buffer overruns” yields tens of thousands of hits!
The Sad Reality

• Even well-known buffer overruns are still widely exploited
  - Hard to get people to upgrade millions of vulnerable machines
    • And upgrading can sometimes create new vulnerabilities!

• We assume that there are many more unknown buffer overrun vulnerabilities
  - At least unknown to the good guys
Static Analysis to Detect Buffer Overruns

• Detecting buffer overruns before distributing code would be better

• Idea: Build a tool similar to a type checker to detect buffer overruns

• Alex Aiken with David Wagner, Jeff Foster, and Eric Brewer
  – “A First Step Toward Automated Detection of Buffer Overrun Vulnerabilities”, NDSS 2000
Focus on Strings

• Most important buffer overrun exploits are through string buffers
  - Reading an untrusted string from the network, keyboard, etc.

• Focus the tool only on arrays of characters
Idea 1: Strings as an Abstract Data Type

• A problem: Pointer operations & array dereferences are very difficult to analyze statically
  - Where does *a point?
  - What does buf[j] refer to?

• Idea: Model effect of string library functions directly
  - Hard code effect of strcpy, strcat, etc.
Idea 2: The Abstraction

• Model buffers as pairs of integer ranges
  – Size       allocated size of the buffer in bytes
  – Length     number of bytes actually in use

• Use integer ranges \([x,y] = \{ x, x+1, \ldots, y-1, y \}\)
  – Size & length cannot be computed exactly
The Strategy

• For each program expression, write constraints capturing the alloc and len of its string subexpressions

• Solve the constraints for the entire program

• Check for each string variable s
  \[ \text{len}(s) < \]