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
C Design Principles

• Small language
• Maximum efficiency
• Safety less important

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

char buffer[100];

Declares and allocates an array of 100 chars

100 *sizeof(char)
C Array Operations

```c
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';
```
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
buf1[i] = 1;    /* buf1 has type int[] */

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

• 5 instructions worst case
• Often &buf1 and i already in registers
  - Saves 2 instructions
• Many machines have indirect loads/stores
  - store r1[r3], 1
  - Saves 1 instruction
• Best case 2 instructions
  - Offset calculation and memory operation
buf1[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 */
```

![Diagram showing stack allocation and buffer overflow]

100 * sizeof(char)
An Overrun Vulnerability

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

char buf[104] = { ' ',...,',' , magic 4 chars }

foo(buf); (**)

Foo entry

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>99</th>
<th>return address</th>
</tr>
</thead>
</table>

100 *sizeof(char)

Foo exit

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>99</th>
<th>return address</th>
</tr>
</thead>
</table>

100 *sizeof(char)

magic 4 chars
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 char’s to an 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);
Details

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

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

• Overwrite return address with start of buffer
  - Harder
  - Need to guess where buffer in called routine starts (trial & 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

```c
char buf[104] = "NOPS ... /bin/exec sh XXXXXXXXXXXX"
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
  - Program still appears 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 widely known since the 1980’s
  - Remain a popular attack today

• Highly automated toolkits available to exploit known buffer overruns
  - Search for “buffer overruns” yields > 100,000 hits
The Sad Reality

• Even well-known buffer overruns are still exploited
  - Hard to get people to upgrade millions of vulnerable machines

• We assume that there are many more unknown buffer overrun vulnerabilities
  - At least unknown to the good guys
How Do We Prevent Buffer Overruns?

• Many proposed techniques!
  - A research Rorschach test

• A brief survey
  - Language design
  - Static analysis
  - Dynamic analysis
Language Design

• Enforce data abstractions!

• How?
  - Type safety
    • The guarantee that if e: T, then e evaluates to a value of type T
    • No unsafe casts
  - Memory safety
    • Array bounds checking
    • No computation on pointers
    • Automatic memory management
Tools for Static Memory Safety

• Bug finding tools
  - Detect common patterns of buffer overruns
  - Use heuristics
    • Focus on scenarios likely to be real overruns, rather than obscure scenarios that might not be
    • Avoid false positives

• Verification
  - Formally prove memory safety
  - Can require deep understanding of the program’s semantics
Dynamic Memory Safety

• Many proposals

• Sandboxing
  - Confine all memory references in the program to its own data space
  - Guarantees damage is limited to the program itself

• Code and data randomization
  - Give everyone a slightly different binary and data layout
  - Variation minimizes chances an attack can work on all copies of a program
Summary

• Programming language knowledge useful beyond compilers

• Useful for programmers
  – Understand what you are doing!

• Useful for tools other than compilers
  – Big research direction