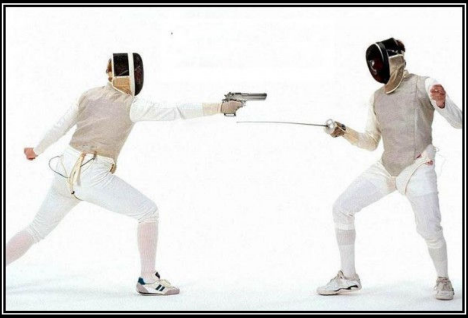


Language Security

Or: bringing a knife to a gun fight

#1

One-Slide Summary

- A language's design principles and features have a strong influence on the security of programs written in that language.
- C's legacy of null-terminated, stack-allocated and non-sized buffers leads directly to the most common sort of security vulnerability: the buffer overrun.
- What can be done?

#2

Lecture Outline

- Beyond interpreters
  - Looking at other issues in programming language design and tools
- C
  - Arrays
  - Exploiting buffer overruns
  - Detecting buffer overruns

#3

Duck-billed Platitudes

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

C Design Principles

- Small language
- Maximum efficiency
- Safety less important

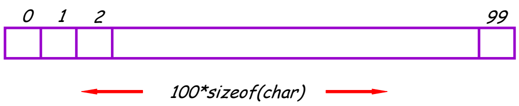
- Designed for the world in 1972
  - Weak machines
  - Trusted networks

#5

Arrays in C

char buffer[100];

Declares and allocates an array of 100 chars



#6

C Array Operations

```
char buf1[100], buf2[100];
```

Write:

```
buf1[0] = 'a';
```

Read:

```
return buf2[0];
```

#7

What's Wrong with this Picture?

```
/* strcpy buf1 into buf2 */
int i;
for (i = 0; buf1[i] != '\0'; i++) {
    buf2[i] = buf1[i];
}
buf2[i] = '\0';
```

#8

Indexing Out of Bounds

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

```
char buffer[100];

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

#9

Why Ask Why?

- Why does C allow out of bounds array references?
  - Proving at compile-time that all array references are in bounds is very difficult (*why?*)
  - Checking at run-time that all array references are in bounds is expensive (*who does this?*)

#10

Code Generation for Arrays

- The C code:

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

- The assembly code:

Regular C

```
r1 = &buf1;
r2 = load i;
r3 = r2 * 4;



r4 = r1 + r3
store r4, 1
```

C with bounds checks

```
r1 = &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
```

Costly!

Finding the array limits is non-trivial

#11

C vs. Java

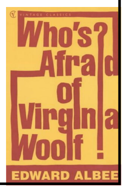
- Typical work for a C array reference
  - Offset calculation
  - Memory operation (load or store)

- Typical work for a Java array reference
  - Offset calculation
  - Memory operation (load or store)
  - Array bounds check
  - Type compatibility check (for stores) (*why?*)

#12

Buffer Overruns

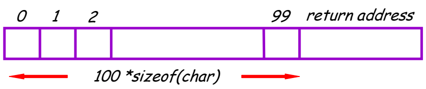
- A buffer overrun writes past the end of an array
- **Buffer** usually refers to a C array of char
  - But can be any array
- So who's afraid of a buffer overrun?
  - Cause a core dump
  - Can damage data structures
  - What else?



Stack Smashing

Buffer overruns can alter the control flow of your program!

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

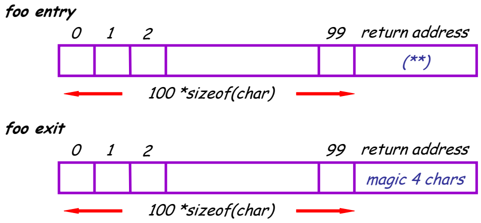


An Overrun Vulnerability

```
void foo(char in[]) {
  char buffer[100];
  int i = 0;
  for(i = 0; in[i] != '\0'; i++)
     buffer[i] = in[i];
  buffer[i] = '\0';
}
```

An Interesting Idea

char in[104] = { 0,…,0, *magic 4 chars* }
foo(in);   (**)



#16

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 to an address

#17

The Rest of the Story

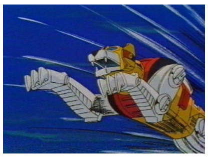
- Say that foo is part of a network server and the in originates in a received message
  - Some *remote* user can make foo jump anywhere!

- But where is a "useful" place to jump?
  - Idea: Jump to some code that gives you control of the host system (e.g. code that spawns a shell)
- But where to put such code?
  - Idea: Put the code in the same buffer and jump there!

#18

Useful Jumps

- Where to jump?



- We want to take control of the program
- How about to a system call?

#19

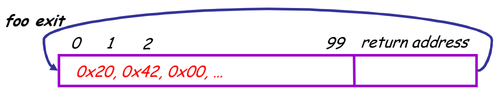
The Plan

- Force a jump to the following code:
- In C: exec("/bin/sh");
- In x86 assembly:
  - movl $LC0, (%esp)
  - call _exec
  - LC0: .ascii "/bin/sh\0"
- In machine code: 0x20, 0x42, 0x00, …

#20

The Plan

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

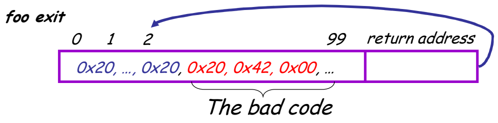


- The last 4 bytes in "in" must equal the start of buffer
  - That position might depend on many factors !

#21

Guess the Location of the Injected Code

- Trial and error: gives you a ballpark
- Then pad the **injected code** with NOP
  - e.g. add $0, $1, 0x2020
    - stores result in $0 which is hardwired to 0 anyway
    - Encoded as 0x20202020

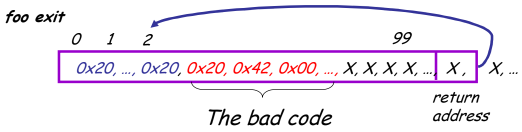


- Works even with an approximate address of buffer !

#22

More Problems

- We do not know exactly where the return address is
  - Depends on how the compiler chose to allocate variables in the stack frame
- Solution: pad the buffer at the end with many copies of the "magic return address X"



#23

Even More Problems

- The most common way to copy the bad code in a stack buffer is using string functions: strcpy, strcat, etc.
- This means that buf cannot contain 0x00 bytes
  - *Why?*
- Solution:
  - Rewrite the code carefully
  - Instead of "addiu $4,$0,0x0015 (code 0x20400015)
  - Use "addiu $4,$0,0x1126; subiu $4, $4, 0x1111"

#24

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 w.r.t. to buffer overruns
  - Easy to do the wrong thing
  - Hard to do the right thing

#25

The State of Hacking

- Buffer overruns are the attack of choice
  - 40-50% of new vulnerabilities are buffer overrun exploits
  - Many recent attacks of this flavor: Code Red, Nimda, MS-SQL server, yada yada
  - “Buffer overflows have been the most common form of security vulnerability for the past ten years …” [OGI DARPA 2000]
- Highly automated toolkits are available to exploit known buffer overruns
  - Look up “script kiddie”

#26

The Sad Reality

- Even well-known buffer overruns are still widely 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

#27

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
- This is a popular research area; we'll present one idea at random [Wagner, Aiken, …]
  - You'll see more in later lectures

#28

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

#29

Idea 1: Strings as an Abstract Data Type

- A problem: Pointer operations and array dereferences are very difficult to analyze statically
  - Where does *ptr point?
  - What does buf[j] refer to?
- Idea: Model effect of string library functions directly
  - Hard code effect of strcpy, strcat, etc.

#30

Idea 2: The Abstraction

- Model buffers as pairs of integer ranges
  - *Alloc* min allocated size of the buffer in bytes
  - *Length* max number of bytes actually in use
- Use integer ranges
  - [x,y] = { x, x+1, …, y-1, y }
  - Alloc and length cannot be computed exactly

#31

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

  len(s) $\leq$ alloc(s)

#32

The Constraints

char s[n]; n $\leq$ alloc(s) (or n=alloc(s))

strcpy(dst,src) len(src) $\leq$ len(dst)

p = strdup(s) len(s) $\leq$ len(p) & alloc(s) $\leq$ alloc(p)

p[n] = '\0' min(len(p),n+1)) $\leq$ len(p)

#33

Constraint Solving

- Solving the constraints is akin to solving dataflow equations (e.g., constant propagation)
- Build a graph
  - Nodes are len(s), alloc(s)
  - Edges are constraints len(s) $\leq$ len(t)
- Propagate information forward through the graph
  - Special handling of loops in the graph

#34

Results

- This technique found new buffer overruns in sendmail
  - Which is like shooting fish in a barrel …
- Found new exploitable overruns in Linux nettools package
- Both widely used, previously hand-audited packages

#35

Limitations

- Tool produces many **false positives** (*why?*)
  - 1 out of 10 warnings is a real bug
- Tool has false negatives (*why?*)
  - Unsound: may miss some overruns
- But still productive to use

#36

Summary

- Programming language knowledge is useful beyond interpreters
- Useful for programmers
  - Understand what you are doing!
- Handy for tools other than compilers
  - Big research direction

#37

Homework

- PA5 Due Friday April 27 (3 days)
- Final Examination
  - Block 4
  - Thursday May 10
  - 1400-1700
  - MEC 214

#38