

# Memory Safety



“If our software is buggy, what  
does that say about its security?”

—*Robert H. Morris*

# It's All About the Software

- Most penetrations are due to buggy software
- Crypto won't help there: bad software defeats good crypto
- Design matters, too

# Security Isn't About Cryptography

- Cryptography is necessary
- But it's possible for cryptographic software to have bugs, too
- In one recent study, 80% of mobile apps had problems with their cryptography



# Three Separate Aspects

- Enforcing security
- Avoiding bugs
- Proper components and proper composition

# Avoiding Bugs

- Many simple bugs can be exploited to cause security problems
- The C language is a large part of the problem
- One big issue: *memory safety*

# Memory Safety

For true memory safety, we need several properties:

- 1 No pointer can ever be dereferenced outside its legal bounds
- 2 Type safety: pointers can only point to objects of the appropriate type, within valid bounds: `int*` pointers only refer to integers, `char*` to characters, etc.
- 3 Only pointers to functions can be invoked as functions (or otherwise executed)
- 4 Etc.

C/C++ do not qualify

# C and C++ are not Memory-Safe

```
int v[5], x;  
int p = v;  
x = *(p+5);
```

v can only legally point to  
v[0] . . . v[4], but C doesn't enforce  
this

```
int q, *qp = &q;  
char cp  
cp = *(char *)qp;
```

qp is supposed to point only to  
integers, but a language feature lets  
it point to characters

```
union {  
    int a;  
    float b;  
    char c[5];  
} u, *unp;  
unp = &u;
```

We can write unp.a, unp.b, and  
unp.c—unp can point to anything

# You Can Even Set Function Pointers to Arrays

```
int (*fp)(int);  
char buf[1024];  
fp = (void *)buf;  
(*fp)(1024);
```

Note: I fed all of these examples to gcc, and it didn't even give a warning...

# Serious Consequences

- C's lack of memory safety and type safety have led to serious consequences
- The lack of array-bounds checking is notorious
- The lack of safe memory allocation and freeing has also been very problematic

# Subscript Out-of-Bounds: Buffer Overflows

- Once responsible for about half of all security vulnerabilities
- Fundamental problems:
  - Character strings in C are actually arrays of chars
  - There is no array bounds checking done in C
- Attacker's goal: overflow the array in a controlled fashion

# Stack Frame

High memory address

0x98	0x76	0x54	0x54	} Return Addr
r	l	d	\0	} Buffer
o		w	o	
h	e	l	l	
(Other local variables)				

Low memory address

When a function is called, the return address is stored on the stack. Lower in memory, all variables local to that function are stored.



# Buffer Overflow

High memory address

l	o	w	\0	} Return Addr
v	e	r	f	
a	n		o	} Local variables
	i	s		
T	h	i	s	} Buffer
(Other local variables)				

Low memory address

If the array bounds are exceeded, the return address can be overwritten.

# Buffer Overflow Attack

High memory address

0x23	0x45	0x67	0x89	} Return Addr
0x0D	0x0E	0x0F	0x10	
0x09	0x0A	0x0B	0x0C	} Buffer
0x05	0x06	0x07	0x08	
0x01	0x02	0x03	0x04	
(Other local variables)				

Low memory address

Put code in the early part of the buffer, then change the return address to point to it. When the function exits, the injected code is executed.

# How Can Such Things Happen?

- C has lots of built-in functions that don't check array bounds
- Programmers frequently don't check, either
- The attacker supplies too-long input

# Sample Problematic Code

```
char line[512];
```

```
...
```

```
gets(line);
```

That's from the 4.3BSD fingerd command, exploited by the first Internet Worm in 1988...

# Bad versus Good

<code>gets()</code>	<code>fgets()</code>
<code>strcpy()</code>	<code>strncpy()</code>
<code>strcat()</code>	<code>strncat()</code>
<code>sprintf()</code>	<code>snprintf()</code>

- Java checks array bounds
- C# checks array bounds
- Go checks array bounds
- Python checks array bounds
- More or less everything *but* C and C++ check. . .

# Indirect Buffer Overflows

```
void f(char *s)
{
    sprintf(s, "....");
}
```

```
void g()
{
    char buf[128];

    f(buf);
}
```

Function f doesn't even know the size of the array!

- Compiler trick—available for gcc and Microsoft compilers
- Generate a random “canary” value at the start of each program execution
- Insert that value between the return address and the rest of the stack frame
- Check if it’s intact before returning
- Any stack-smash attack will have to overwrite the canary to get to the return address
- Remember: the canary’s value is different for each run of the program



# Buffer Overflow Attack

High memory address

0xAB	0xCD	0xEF	0xFF	} Return Address
0x23	0x45	0x67	0x89	
0x0D	0x0E	0x0F	0x10	} Canary
0x09	0x0A	0x0B	0x0C	
0x05	0x06	0x07	0x08	} Buffer
0x01	0x02	0x03	0x04	
(Other local variables)				

If the random canary is overwritten, the program will abort.  
Now standard in C/C++ compilers

# Heap Overflow Attacks

- You can't easily put canaries in the heap area
- Return addresses are on the stack, not the heap—does this make buffer overflows in heap variables safe?

# Heap Overflow Attacks

- You can't easily put canaries in the heap area
- Return addresses are on the stack, not the heap—does this make buffer overflows in heap variables safe?
- Nope
- The heap often contains pointers to functions—especially true for C++, with virtual functions
- Use a buffer overflow to inject code and then change such a pointer to point to it
- When the virtual function is called. . .

*What would we think of a sailing enthusiast who wears his life-jacket when training on dry land but takes it off as soon as he goes to sea? Fortunately, with a secure language, the security is equally tight for production and for debugging.*

*Hints on Programming Language Design, C.A.R. Hoare, 1973*

# Many Defenses

- Many defenses have been tried
- Always, the attackers have found a new variant
- It is unlikely that we can *ever* prevent memory attacks on C/C++

# ASLR: Address Space Layout Randomization

- Put stack at different random location each time program is executed
- Put heap at different random location as well
- Defeats attempts to address known locations
- But—makes debugging harder

# Non-Executable Data Areas

- Modern computer architectures have permission bits for memory pages: can only execute code if the “execute” bit is set
- Defense: on pages with the “write” bit set, don’t set “execute”
- The stack is writable, so code injected by the attacker won’t be executable
- Called “DEP” (Data Execution Prevention) or “ $W \oplus X$ ”

# Checking Code

- Look for suspect calls
- Use static checkers
- Use a better compiler that can insert bounds-checking (but that's very hard if you want binary compatibility)



# Stack versus Heap or BSS Storage

- Easiest to exploit if the buffer is on the stack
- Exploits for heap- or BSS-resident buffers are also possible, though they're harder
- Heap and BSS attacks not preventable with canaries (but there are analogous techniques to protect `malloc()`-allocated storage)
- Some operating systems can make such memory pages non-executable, which is a big help—but that breaks some applications

# Issues for the Attacker

- Finding vulnerable programs
- NUL bytes
- Uncertainty about addresses

# Finding Vulnerable Programs

- Use nm and grep to spot use of dangerous routines
- Probe via very-long inputs
- Look at source or disassembled/decompiled code

- C strings can't have embedded 0 bytes
- Some instructions do have 0 bytes, perhaps as part of an operand
- Solution: use different instruction sequence


# Address Uncertainty

- Pad the evil instructions with NOPs
- This is called a *landing zone* or a *NOP sled*
- Set the return address to anywhere in the landing zone

# Buffer Overflow: Summary

- You *must* check buffer lengths
- Where you can, use the safer library functions
- Write your own safe string library (there's no commonly-available standard)
- Use C++ and class `String`
- Use Java
- Use *anything* but raw C!

# History of Buffer Overflows

- Long-recognized as a security issue
  - First very visible exploit: Robert T. Morris' Internet Worm, November 1988.
  - Popularized by Aleph One in November 1996; serious threat since then
-  The attack was theoretically difficult, but there are canned exploit kits available

# Hoare's Turing Award Lecture, 1980

The first principle was security: . . . A consequence of this principle is that every occurrence of every subscript of every subscripted variable was on every occasion checked at run time against both the upper and the lower declared bounds of the array. . . . I note with fear and horror that even in 1980, language designers and users have not learned this lesson. In any respectable branch of engineering, failure to observe such elementary precautions would have long been against the law.



# Can We Afford Array-Bounds Checking?

- Of course—spend the Moore's Law benefit on something besides better video games
- Compiler optimizations often make the expense a lot less than you'd think
- It's hard to do in C, though, because of array vs. pointer semantics
- Things like `*p++ = *q++` are hard to check efficiently
- A bounds-checking C compiler has been written, but it's largely unused

# Use-After-Free Attacks

- C has lousy memory management:

```
p = malloc((size_t) 1024);
```

```
...
```

```
free((void *)p);
```

- But nothing stops the programmer from incorrectly using p after the call to free()
- Yes, this can cause problems

# Launching the Attack

- Find a code path that causes a new `malloc()` that will reuse that same area
- Have it fill the area with attacker-controlled data
- Cause the program to use that dangling pointer
- It sounds difficult, but it's been used a lot in practice

# Reported Bugs in Chromium, 2011–2013

Severity	Use-after-free	Stack overflow	Heap overflow	Others
Critical	13	0	0	0
High	582	12	107	11
Medium	80	5	98	12
Low	5	0	3	1
Total	680	17	208	24

- Best, of course, is to use a language that has automatic garbage collection
- Good programming habits can help:

```
free((void *)p);  
p = NULL;
```

- Memory leak detectors can also help sometimes

# Return-Oriented Programming

- The previous attacks require the the attacker to actually inject code
- Defenses such as  $W \oplus X$  mean that injected code isn't executable
- Attacker countermove: return-oriented programming (ROP)

# Principles of ROP

- There are lots of segments of bytes in, e.g., the C library that (especially on the Intel x86 architecture) form useful instruction sequences and end in a RET (pop the stack and return) instruction
- Such a sequence is called a *gadget*
- Find a set of gadgets that, strung together, do something “useful”
- Via a buffer overflow or some such, push the addresses of your string of gadgets onto the stack
- When the function returns, it will execute the first gadget; it will return to the second, etc.
- No new code is needed!

# The Role of Specifications

- Contrast this:  
*“File names may be up to 1024 bytes long”*  
with  
*“File names may be up to 1024 bytes long; longer file names must be rejected”*
- The second form alerts the programmer to the real requirement
- Just as important, the second form alerts the *tester* to the requirement
- Testing is done against requirements!



# Format String Errors

- Suppose `str` is input to the program
- Wrong:  

```
printf(str);
```
- Right:  

```
printf("%s", str);
```
- Format strings can be dangerous...
- Note: other functions (i.e., `syslog`) also take format strings

# The %n Problem

- Rather complex; I won't try to explain the details here
- Fundamental issue: %n writes to a variable the number of bytes printed thus far
- The statement

```
printf("Hello\n%n", &cnt)
```

stores a 6 in integer variable cnt
- This can be used to overwrite memory locations
- Use tricks involving other references to (non-existent!) other arguments to let you write to someplace “useful”

# Another Minor Issue

- Minor problem: metacharacters can confuse log files
- Here's an embedded newline in a username

 12:34:56 Permission denied: user  
12:34:xx Watch this crash!


# The Underlying Issues

- Problem 1: C has strange semantics
- The *only* defense is to know the language thoroughly
- You also have to know possible exploits
- There are integer overflow attacks, too
- Problem 2: programs don't always validate their inputs

# Input Validation

- Trust *nothing* supplied by the user
- Must define inputs before they can be checked
- “A program whose behavior has not been specified cannot be buggy, only surprising.”
- Example: is a newline a valid character in a username?

- Rigorously check all inputs against the specification
- Before that, of course, you need a spec
- (Specs can be buggy, too)
- Alternatively, use an earlier filter or check against a known-good list


- Example: `fgets()` stops at a newline; you can't find any embedded
-  But watch for unterminated buffer—what if the input line is too long?
- Note that `argv` has no such guarantee
- Email: check recipient name against valid user list—no funny characters there

# Being Careful Near the Shell

- If user input is being passed to the shell, be especially careful
- Watch for `popen()` and `system()`
- Dangerous characters include:  
`'~#$^&(){}[];' "<>?|\`
- That's most of the special characters!
- You're always much better off with a “good” list than a “bad” list
- Example: on some Unix systems, `^` was treated the same as `|`. Why?  
Because on some models of Teletype—the *ancient* hard-copy terminal!—`^` printed as `↑`, which looked similar to `^`



# Knowing the Semantics

- Sometimes check that there are no / characters in a program name
  - Why? To ensure that the reference is to a given directory
  - Do you need to check \ as well?
-  Will the program ever run on Windows? Note that URLs on Windows use /, but the file system uses \

# Summary

- Trust nothing
- Specify acceptable inputs
- Check everything
- Understand the semantics of anything you invoke
- Try to use a better language than C

# Questions?



(Double-crested cormorant, Morningside Park, September 6, 2020)