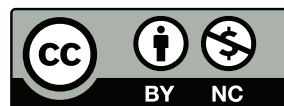# Secure Programming I

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

—*Robert H. Morris*

The Heart of the Problem

- For the typical programmer, crypto is the easy part

- There are standard solutions — and standard libraries — that can solve most such problems for you

- But every program is different

- How you write the programs, and how you combine them, are the most critical part of total system security

It's All About the Software

- Most penetrations are due to buggy software

- Crypto won't help there: bad software trumps good crypto

- Design matters, too

Three Separate Aspects

- Avoiding bugs

- Enforcing security

- 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)

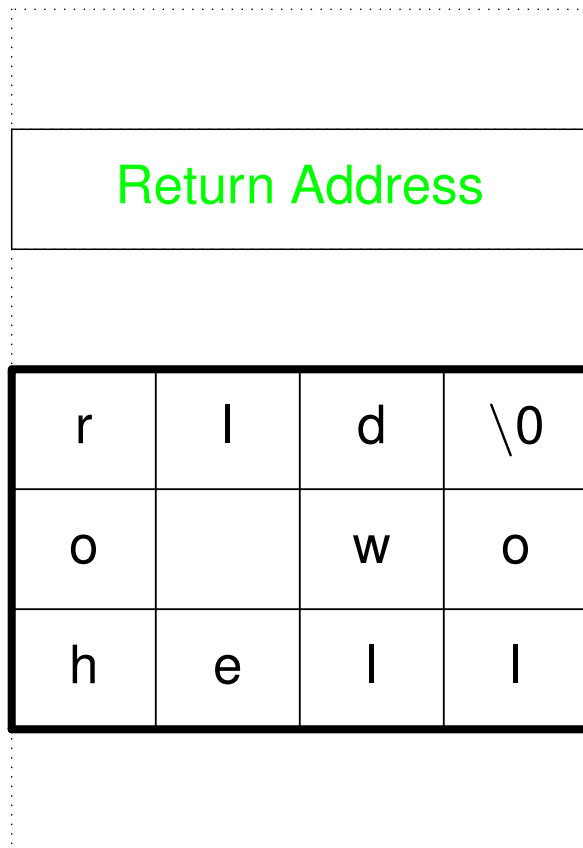- Some of the trouble is compounded by poor specifications

Buffer Overflows

- Responsible for about half of all security vulnerabilities

- Fundamental problems:

  - Character strings in C are actually arrays of `char`s

  - There is no array bounds checking done in C

- Attacker's goal: overflow the array in a controlled fashion

Stack Frame

Return Address	

r	l	d	\0
o		w	o
h	e	l	l

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

Buffer Overflow
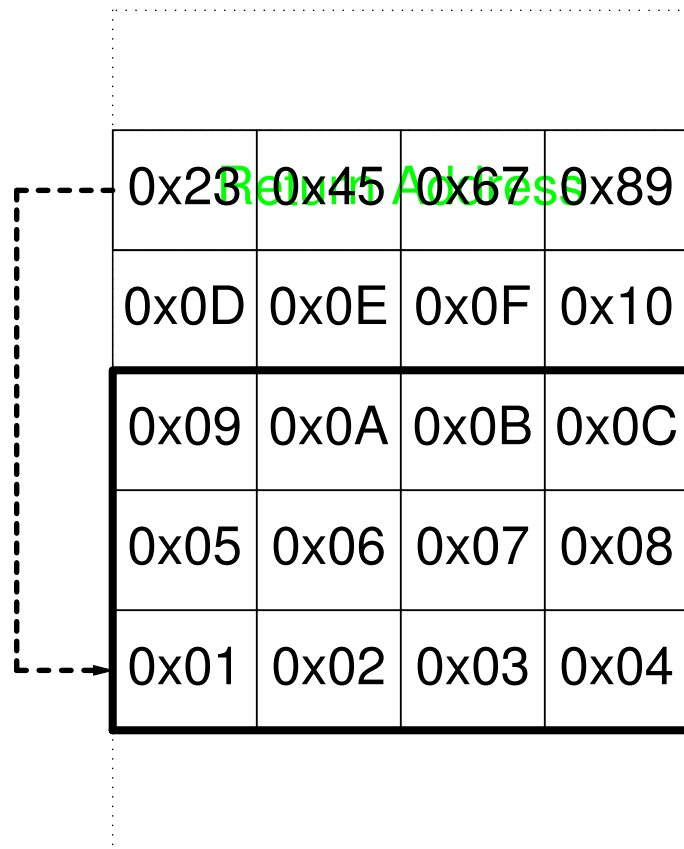
l	o	w	\0
v	e	r	f
a	n		o
	i	s	
T	h	i	s

(Return Address)

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

Buffer Overflow Attack

0x23	0x45	0x67	0x89
0x0D	0x0E	0x0F	0x10
0x09	0x0A	0x0B	0x0C
0x05	0x06	0x07	0x08
0x01	0x02	0x03	0x04

Return 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 in 1988...

Bad versus Good

```
gets()      fgets()
strcpy()    strncpy()
strcat()    strncat()
sprintf()   snprintf()
```

Java vs. C

"Daddy, what's an `IndexOutOfBoundsException`?"

"It's why I'm teaching you Java instead of C."

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!

Canaries

- Compiler trick — available for gcc and Microsoft compilers

- Insert a random "canary" 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

- Use a different random value each time the program is executed

Overwriting a Canary

l Return Address	o w	\0	
v	e Canary		f
a	n		o
	i	s	
T	h	i	s

If the random canary is overwritten, the program will abort

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

Checking Code

- Look for suspect calls

- Use static checkers

- Use language feature like Perl's "taint" mode

"Taint" Mode

- Optional feature in Perl

- Variables whose value comes from an untrusted source are marked as "tainted"

- Variables whose value comes (in part) from a tainted variable are also tainted

- Tainted values cannot be passed to sensitive routines

- Taints are removed via regular expression match; it is presumed that the expression will sanitize the data, and remove anything dangerous

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 disassembed/decompiled code

NUL Bytes

- 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*

- 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 is 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

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

What's the Problem?

- 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!

- Bigger problem: %n

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"

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?

Defenses

- 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

Filtering

- 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 `/etc/passwd` — 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, ˆ is treated the same as |. Why? Because on some models of Teletype, ˆ 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