CMSC 414 — Computer and Network Security

Buffer Overflows

Dr. Michael Marsh

August 30, 2017

Trust and Trustworthiness

You read:

- *Reflections on Trusting Trust* (Ken Thompson), 1984
- *Smashing the Stack for Fun and Profit* ("Aleph One"), 1996

These both highlight:

- We place trust in people/things
- This trust might not be deserved
- This can lead to Bad Things

Relevance

Buffer overflow attacks have been with us for 3 decades

- **Morris worm (1988)**: first self-propagating malware, overflow in finger daemon
- **CodeRed (2001)**: overflow in MS-IIS server
- **SQL Slammer (2003)**: overflow in MS-SQL server
- **Conficker work (2008)**: overflow in Windows RPC
- **Stuxnet (2009)**: overflows in Windows print spooler, LNK shortcut display, task scheduler, and Conficker's target vulnerability
- **Flame (2010)**: Stuxnet's print spooler and LNK targets

More Relevance

Some vulnerabilities are patched before (that we know) they are exploited:

- **X11 Server**: bug introduced in 1991, fixed in 2014
- **GHOST**: vulnerability in glibc's `gethostbyname()`, introduced in 2000, fixed in 2013
- **syslogd on OS X and IOS**:

```
global.lockdown_session_fds = reallocf(
        global.lockdown_session_fds,
        global.lockdown_session_count + 1 * sizeof(int));
```

  Did you notice the missing parentheses?
- $\sim$ 10% of *all* vulnerabilities for the past decade

Even More Relevance

From MITRE's 2011 CWE/SANS Top 25 Most Dangerous Software Errors

Rank	Score	ID	Name
1	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
2	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
3	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
4	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
5	76.9	CWE-306	Missing Authentication for Critical Function
6	76.8	CWE-862	Missing Authorization
7	75.0	CWE-798	Use of Hard-coded Credentials
8	75.0	CWE-311	Missing Encryption of Sensitive Data
9	74.0	CWE-434	Unrestricted Upload of File with Dangerous Type
10	73.8	CWE-807	Reliance on Untrusted Inputs in a Security Decision
11	73.1	CWE-250	Execution with Unnecessary Privileges
12	70.1	CWE-352	Cross-Site Request Forgery (CSRF)

Question 1

Which of the following is *not* on the stack?

A. Dynamically allocated memory

B. Function return address

C. Function arguments

D. Local variables

What Can Buffer Overflows Do?

We know we can use a buffer overflow to gain control of `%eip`, but we can also...

- Overwrite local variables (which live on the stack)
- Extract local variables (without having to modify the stack)
- Do both of the above for function arguments
- Crash a program (which is often bad enough)

Group Exercise 1

Let's win the lottery!

```
tools/get-assignment lottery
```

Write `your_fcn()` so that the program **always** prints "You win!" as its last output. **Do not modify `main()`!**

At your tables, see how many ways you can come up with to win in 10 minutes.

Problems with Assumptions

All programmers make assumptions. There's no way not to.

If these assumptions are wrong $\Rightarrow$ vulnerabilities!

Assumptions *might have* been correct once, but operating environments can change.

What assumptions did our lottery program make?

Lottery Assumptions

1. `your_fcn()` would choose an integer, and do nothing else.
2. Calling `seed()` properly would guarantee reasonable randomness.
3. `your_fcn()` would return and pass control to the following instruction.
4. **The author of `your_fcn()` is trustworthy.**

If this last isn't true, then none of the others will be.

Question 2

Consider the following code:

```
void foo(int i) {
    char s[5];
    if (X)
        sprintf(s, "%d", i);
}
```

With which of the following can we safely replace *X*?

A. `i < 10000`

B. `i*i < 100000`

C. `i > 100`

D. `abs(i) < 10000`

Finding Overflows in Source Code

There are a number of things we can look for, some are easier to find overflows in than others (this list is not exhaustive):

- `strcpy()`
- `gets()`, especially in a loop
- `memcpy()`
- `sprintf()`
- array writes in loops

Finding Overflows *Without* Source Code

Some observable effects that might indicate that a buffer is overflowing:

- Funny characters in output
- Segmentation faults
- Illegal instruction faults
- Repeated actions that *should* have the same result, but don't

Question 3

Often, finding an exploitable overflow is a multi-step process. Once we've seen external evidence of an overflow, such as a garbled message, which of the following steps would we do next?

A. Search the source code for a string we've seen in the output

B. Run the program in the debugger

C. Search the source code for a path from user input to an overflowable buffer

D. Search the source code for potentially exploitable functions, like `strcpy()`

User Input is Dirty, and Should Never be Trusted

Buffer overflows happen because programs read in user-controlled data

Most users aren't malicious, some are

Even those who aren't hit the wrong key sometimes

- Check input lengths
- Check for unexpected characters
- Check ranges for things like numbers
- Don't assume anything!

Be careful when handling inputs

`printf(str)` $\Leftarrow$ BAD!

`printf("%s",str)` $\Leftarrow$ GOOD!

Group Exercise 2

Let's exploit a format string vulnerability!

The `lottery` repo also has a file `format.c`.

At your tables, take 5 minutes and see if you can extract the "secret" arguments to the function `vuln()`. *Hint: You should get a secret message!*

Preventing Buffer Overflow Vulnerabilities

Programmers are going to write bad programs.

How do we keep systems safe in spite of this?

In the case of buffer overflows, there are some techniques. We've disabled some of these for our exercises.

Stack Canaries

Add special values at certain places on the stack.

If these values change $\Rightarrow$ something is wrong!

If these are predictable, still possible to defeat, but harder.

We disabled this, along with bounds checking, with `-fno-stack-protector`.

Non-Executable Stacks

Our shellcode has to be somewhere. Usually, this is a buffer on the stack.

`%eip` should never point to a stack location, but that's exactly what we're doing.

If we mark the stack as *non-executable*, we can defeat this type of attack.

We disabled this with `-z execstack`.

Address Randomization

Disassembling in gdb gives us instruction addresses.

Inspecting variables and memory locations in gdb gives us more information.

If this information is stable, we can construct an exploit that will always work.

If this information changes, our work is much harder, because we have to try many times before we're likely to succeed.

Address-Space Layout Randomization (ASLR) randomizes where things are in memory every time you run.

We disabled this with

```
sudo sysctl -w kernel.randomize_va_space=0
```

Question 4

If ASLR is enabled, how might we improve our chances of a successful exploit?

A. Filling the stack with many random addresses, instead of repeating just one

B. Repeating our shellcode multiple times

C. Adding more NOP instructions before our shellcode

D. Disabling ASLR

Return-Oriented Programming

If we can't execute instructions on the stack, what can we do?

1. "Groom" the stack
   - Set registers to useful values
   - This includes arguments as well as `%esp`
2. Set `%eip` to a "gadget"
   - Existing library code, with predictable (or computable) address
   - *Does not* need to be normal function entrypoint
   - One or more instructions using register values we've groomed
   - Ends with a `ret` to take us to the next gadget
3. Repeat

We aren't going to go into this further, but there are many references on the Internet.

Question 5

What's one thing that makes ROP particularly challenging?

A. Determining what instructions you want to run

B. Overwriting a vulnerable buffer

C. Finding gadgets that do something useful

D. Removing NULLs from the exploit

Using the Shell with Malicious Intent

`your_fcn()` is obviously a toy example. We'd never let users hand us the code to execute. (Actually, sometimes we do!)

Most programs these days are *dynamically* linked. Important functions are pulled in at load time from other files. How these files are found is based on searching a path. We can control this path with an environment variable called `LD_LIBRARY_PATH`.

There is also an environment variable called `LD_PRELOAD` that lets us load a dynamic library of our choosing before anything else.

For extra fun, most people have "`.`" in their executable search path, often as the first entry. Nothing stops you from creating a malicious program named "`ls`", and a sloppy sysadmin might be tricked into running it...

Group Exercise 3

To finish out today's class, discuss with your group how you would find ROP gadgets in a library. Write some simple code (do one simple thing, like add two numbers), look at the instructions that are generated, and see if you can find that as a gadget in a standard library.

You might want to use `disassem/r` in **gdb** and the **xxd** command.