Memory Safety and Buffer Overflows

(with material from Mike Hicks, Dave Levin and Michelle Mazurek)
Today’s agenda

- Why care about buffer overflows?
- Memory layout refresher
- Overflows and how they work
What is a buffer overflow?

- A low-level bug, typically in C/C++
  - Significant security implications!
- If accidentally triggered, causes a crash
- If maliciously triggered, can be much worse
  - Steal private info
  - Corrupt important info
  - Run arbitrary code
Why study them?

• Buffer overflows are still **relevant** today
  • C and C++ are still popular
  • Buffer overflows still occur with regularity

• They have a **long history**
  • Many different approaches developed to defend against them, and bugs like them

• They share **common features** with other bugs we will study
  • In **how the attack works**
  • In **how to defend against it**
C and C++ still very popular

<table>
<thead>
<tr>
<th>Language Rank</th>
<th>Types</th>
<th>Spectrum Ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Python</td>
<td>🌐💻📱</td>
<td>100.0</td>
</tr>
<tr>
<td>2. C++</td>
<td>🎮💻📱</td>
<td>98.3</td>
</tr>
<tr>
<td>3. C</td>
<td>🎮💻📱</td>
<td>98.3</td>
</tr>
<tr>
<td>4. Java</td>
<td>🎮📱</td>
<td>97.2</td>
</tr>
<tr>
<td>5. C#</td>
<td>🎮💻📱</td>
<td>92.7</td>
</tr>
<tr>
<td>6. R</td>
<td>🎮📱</td>
<td>82.8</td>
</tr>
<tr>
<td>7. PHP</td>
<td>🎮📱</td>
<td>82.7</td>
</tr>
<tr>
<td>8. JavaScript</td>
<td>🎮📱</td>
<td>82.6</td>
</tr>
<tr>
<td>9. Go</td>
<td>🎮💻</td>
<td>76.4</td>
</tr>
<tr>
<td>10. Assembly</td>
<td>🎮📱</td>
<td>74.2</td>
</tr>
<tr>
<td>11. Matlab</td>
<td>🎮💻</td>
<td>72.8</td>
</tr>
<tr>
<td>12. Scala</td>
<td>🎮📱</td>
<td>72.1</td>
</tr>
<tr>
<td>13. Ruby</td>
<td>🎮💻</td>
<td>71.3</td>
</tr>
<tr>
<td>14. HTML</td>
<td>🎮💻</td>
<td>70.5</td>
</tr>
</tbody>
</table>

Critical systems in C/C++

- Most **OS kernels** and utilities
  - fingerd, X windows server, shell

- Many **high-performance servers**
  - Microsoft IIS, Apache httpd, nginx
  - Microsoft SQL server, MySQL, redis, memcached

A successful attack on these systems is particularly dangerous!

- Many **embedded systems**
  - Mars rover, industrial control systems, automobiles, healthcare devices, IoT
Trends

Relative Vulnerability Type Totals By Year

The vulnerabilities in the NVD are assigned a CWE based on a slice of the total CWE Dictionary. The visualization below shows a stacked bar graph of the total number of vulnerabilities assigned a CWE for each year. It is possible (although not common) that a vulnerability has multiple CWEs assigned.

https://nvd.nist.gov/vuln/visualizations/cwe-over-time
History of Buffer Overflows

- Morris Worm (1988)
  - First internet worm
  - Spread across Unix Machines
- Code Red (2001)
  - Vulnerability in Microsoft Internet Information Services (for hosting web applications)
  - DDoS attack on White House’s servers
- SQL Slammer (2003)
  - Vulnerability in Microsoft SQL Server 2000.
  - Worm spread across more than 250,000 computers and caused a massive internet outage
Recent Examples

WhatsApp Buffer Overflow Vulnerability Reportedly Exploited In The Wild

A new WhatsApp vulnerability has attracted the attention of the press and security professionals around the world. Zimperium zLabs will be creating a detailed blog soon, but we wanted to provide our readers with preliminary information now.

Boeing 787 On-Board Network Vulnerable to Remote Hacking, Researcher Says

Boeing disputes IOActive findings ahead of security firm’s Black Hat USA presentation.

BLACK HAT USA – Las Vegas – IOActive industrial cybersecurity expert Ruben Santamarta last fall discovered an Internet-exposed Boeing Co. server housing firmware specifications for the aviation manufacturer’s 787 and 737 airplane networks.

Intrigued, Santamarta dug into the firmware for the 787. Boeing’s highly networked plane. He meticulously reverse-engineered the binary code and analyzed configuration files – uncovering multiple security vulnerabilities that could allow an attacker to remotely gain access to the sensitive avionics network of the aircraft, also known as the crew information systems network.

"It turns out the firmware I was analyzing is part of the aircraft that is segregating between different networks," he told Dark Reading prior to publicly disclosing his findings here today. The firmware belongs to a core network component in the 787’s network and was riddled with buffer overflow, memory corruption, stack overflows, and denial-of-service flaws that he says could be exploited by a hacker to remotely reach the aircraft’s sensitive crew information systems network module.

Zero-day vulnerability announced by McAfee at Defcon

At DEFCON, McAfee has announced the discovery of a zero-day vulnerability in a commonly used Delta industrial control system.

The vulnerability found in the Delta enteliBUS Manager could allow malicious actors complete control of the operating...
What we’ll do

- Understand how these attacks work, and how to defend against them

- These require knowledge about:
  - The compiler
  - The OS
  - The architecture

Analyzing security requires a whole-systems view
Note about terminology

- We will use **buffer overflow** to mean *any access of a buffer outside of its allotted bounds*
  - An over-read, or an over-write
  - During *iteration* (“running off the end”) or by *direct access*
  - Could be to addresses that *precede* or *follow* the buffer
Memory layout
Memory Layout Refresher

• How is program data laid out in memory?

• What does the stack look like?

• What effect does calling (and returning from) a function have on memory?

• We are focusing on the Linux process model
  • Similar to other operating systems
All programs stored in memory

The process’s view of memory is that it owns all of it

In reality, these are virtual addresses; the OS/CPU map them to physical addresses
Program instructions are in memory

```
0xffffffff
0x00000000
...
0x4c2 sub $0x224,%esp
0x4c1 push %ecx
0x4bf mov %esp,%ebp
0x4be push %ebp
...```

- 4G
- 0xffffffff
- 0x00000000
- Text
Location of data areas

Set when process starts

Runtime

Known at compile time

0x00000000

0xffffffff

Text

Init’d data

Uninit’d data

Heap

Stack

cmdline & env

int f() {
    int x;
    ...
}

int x;

static int x;

static const int y=10;

malloc(sizeof(long));
Memory allocation

Stack and heap grow in opposite directions

Compiler emits instructions to adjust the size of the stack at run-time

managed in-process by \texttt{malloc}

Focusing on the stack for now
Stack and function calls

• What happens when we call a function?
  • What data needs to be stored?
  • Where does it go?

• What happens when we return from a function?
  • What data needs to be restored?
  • Where does it come from?
Basic stack layout

```c
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    ...
}
```

Happens during caller

Local variables pushed in the same order as they appear in the code

Arguments pushed in reverse order of code

The local variable allocation is ultimately up to the compiler: Variables could be allocated in any order, or not allocated at all and stored only in registers, depending on the optimization level used.
Accessing variables

```c
void func(char *arg1, int arg2, int arg3) {
    ...
        loc2++;
    ...
}
```

Q: Where is (this) `loc2`?
A: `-8(%ebp)`

Frame pointer
- Can’t know absolute address at compile time
- `loc2` is always 8B before `???'s

But can know the **relative** address

Stack frame for `func`
- `0xbfffff323 %ebp`
- `0xffffffff`
Returning from functions

Q: How do we restore previous %ebp?

```c
int main()
{
    ...
    func("Hey", 10, -3);
    ...
}
```

Push current %ebp before locals
Set %ebp to current %esp
Set %ebp to (%ebp) at return
Returning from functions

```c
int main()
{
    ...
    func("Hey", 10, -3);
    ...
}  

Q: How do we resume here?
```

Stack frame for func

Previous $ebp$
Instructions in memory

need to save this address: 0x4a7

%eip
Returning from functions

Q: How do we resume here?

Set %eip to 4(%ebp) at return

Push next %eip before call
Stack and functions: Summary

Calling function:
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you
3. **Jump** to the function’s address

Called function:
4. **Push the old frame pointer** onto the stack: %ebp
5. **Set frame pointer** to where the end of the stack is right now: %ebp = %esp
6. **Push local variables** onto the stack

Returning from function:
7. **Reset the previous stack frame**: %esp = %ebp, pop %ebp
8. **Jump back** to return address: pop %eip
Buffer overflows
Buffer overflows from 10,000 ft

- **Buffer =**
  - Contiguous memory associated with a variable or field
  - Common in C
    - All strings are (NUL-terminated) arrays of char’s

- **Overflow =**
  - Put more into the buffer than it can hold

- Where does the overflowing data go?
  - Well, now that you are experts in memory layouts…
Benign outcome

void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}

Upon return, sets %ebp to 0x0021654d

M e ! \0

Auth M e !

SEGFAULT (0x00216551)
Security-relevant outcome

```c
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ...
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

Code still runs; user now ‘authenticated’

```
M e ! \0
```

```
\x4d 65 21 00 %ebp %eip &arg1
```

buffer authenticated
Could it be worse?

```c
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
```

`strcpy` will let you write as much as you want (til a `\0`).

What could you write to memory to wreak havoc?
Aside: User-supplied strings

- These examples provide their own strings
- In reality strings come from users in myriad ways
  - Text input, packets, environment variables, file input...
- Validating assumptions about user input is critical!
  - We will discuss it later, and throughout the course
Code Injection
Code Injection: Main idea

void func(char *arg1)
{
    char buffer[4];
    sprintf(buffer, arg1);
    ...
}

(1) Load my own code into memory
(2) Somehow get %eip to point to it
Challenge 1
Loading code into memory

• It must be the machine code instructions (i.e., already compiled and ready to run)

• We have to be careful in how we construct it:
  • It can’t contain any all-zero bytes
    - Otherwise, sprintf / gets / scanf / … will stop copying
    - How to write assembly to never contain a full zero byte?
  • It can’t use the loader (we’re injecting)
    - How to find addresses we need?
What code to run?

• One goal: general-purpose shell
  • Command-line prompt that gives attacker general access to the system

• The code to launch a shell is called shellcode

• Other stuff you could do?
Shellcode

```c
#include <stdio.h>
int main( ) {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

Assembly:

```
xorl %eax, %eax
pushl %eax
pushl $0x68732f2f
pushl $0x6e69622f
movl %esp,%ebx
pushl %eax
...
```

Machine code (Part of)

```
"\x31\xc0"
"\x50"
"\x68" "/sh"
"\x68" "/bin"
"\x89\xe3"
"\x50"
...
```

 xor to avoid zero byte

(filename) argv (envp)
Challenge 2

Getting injected code to run

- We have code somewhere in memory
  - We don’t know precisely where
- We need to move %eip to point at it
Stack and functions: Summary

Calling function:
1. Push arguments onto the stack (in reverse)
2. Push the return address, i.e., the address of the instruction you want run after control returns to you
3. Jump to the function’s address

Called function:
4. Push the old frame pointer onto the stack: %ebp
5. Set frame pointer to where the end of the stack is right now: %ebp = %esp
6. Push local variables onto the stack

Returning from function:
7. Reset the previous stack frame: %esp = %ebp, pop %ebp
8. Jump back to return address: pop %eip
Hijacking the saved `%eip`

But how do we know the address?
Hijacking the saved %eip

What if we are wrong?

This is most likely data, so the CPU will panic (Invalid Instruction)
Challenge 3
Finding the return address

- If we don’t have access to the code, we don’t know how far the buffer is from the saved `%ebp`

- One approach: try a lot of different values!
  - Worst case scenario: it’s a 32 (or 64) bit memory space, which means $2^{32}$ ($2^{64}$) possible answers

- Without address randomization (discussed later):
  - Stack always starts from the same fixed address
  - Stack will grow, but usually it doesn’t grow very deeply (unless the code is heavily recursive)
Improving our chances: \textit{nop} sleds

\textit{nop} is a single-byte no-op instruction (just moves to the next instruction)

Now we improve our chances of guessing by a factor of \#nops
Putting it all together

Fill in the space between the target buffer and the `%eip` to overwrite

%eip

padding

good guess

Text ... 0xbdf nop nop nop... \x0f \x3c \x2f ...

buffer

nop sled

malicious code
gdb tutorial
Your new best friends

- **i f**: Show info about the current frame (prev. frame, locals/args, %ebp/%eip)
- **i r**: Show info about registers (%eip, %ebp, %esp, etc.)
- **x/<n> <addr>**: Examine <n> bytes of memory starting at address <addr>
- **b <function> s**: Set a breakpoint at <function> step through execution (into calls)