Memory Safety and Buffer Overflows
(with material from Mike Hicks and Dave Levin)
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. C</td>
<td>![Type Icon]</td>
<td>100.0</td>
</tr>
<tr>
<td>2. Java</td>
<td>![Type Icon]</td>
<td>98.1</td>
</tr>
<tr>
<td>3. Python</td>
<td>![Type Icon]</td>
<td>98.0</td>
</tr>
<tr>
<td>4. C++</td>
<td>![Type Icon]</td>
<td>95.9</td>
</tr>
<tr>
<td>5. R</td>
<td>![Type Icon]</td>
<td>87.9</td>
</tr>
<tr>
<td>6. C#</td>
<td>![Type Icon]</td>
<td>86.7</td>
</tr>
<tr>
<td>7. PHP</td>
<td>![Type Icon]</td>
<td>82.8</td>
</tr>
<tr>
<td>8. JavaScript</td>
<td>![Type Icon]</td>
<td>82.2</td>
</tr>
<tr>
<td>9. Ruby</td>
<td>![Type Icon]</td>
<td>74.5</td>
</tr>
<tr>
<td>10. Go</td>
<td>![Type Icon]</td>
<td>71.9</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

• Many **embedded systems**
  • Mars rover, industrial control systems, automobiles, healthcare devices

A successful attack on these systems is particularly dangerous!
History of buffer overflows

The harm has been substantial

1988

2001  2002  2003

- **Morris worm**
  - Propagated across machines (too aggressively, thanks to a bug)
  - One way it propagated was a buffer overflow attack against a vulnerable version of `fingerd` on VAXes
    - Sent a special string to the finger daemon, which caused it to execute code that created a new worm copy
    - Didn’t check OS: caused Suns running BSD to crash
  - End result: $10-100M in damages, probation, community service

*Morris now a professor at MIT*
History of buffer overflows

The harm has been substantial

1988  2001  2003

- **CodeRed**
  - Exploited an overflow in the MS-IIS server
  - 300,000 machines infected in 14 hours
History of buffer overflows

The harm has been substantial

- **SQL Slammer**
  - Exploited an overflow in the MS-SQL server
  - 75,000 machines infected in 10 minutes
23-Year-Old X11 Server Security Vulnerability Discovered

Posted by Unknown Lamer on Wednesday, January 08, 2014 @10:11...

An anonymous reader writes

"The recent report of X11/X.Org security in bad shape rings more truth today. The X.Org Foundation announced today that they've found a X11 security issue that dates back to 1991. The issue is a possible stack buffer overflow that could lead to privilege escalation to root and affects all versions of the X Server back to X11R5. After the vulnerability being in the code-base for 23 years, it was finally uncovered via the automated cppcheck static analysis utility."

There's a scanf used when loading BDF fonts that can overflow using a carefully crafted font. Watch out for those obsolete early-90s bitmap fonts."
Trends

Total occurrences of CWE 119 (Buffer Error)


http://cwe.mitre.org/top25/

This is a brief listing of the Top 25 items, using the general ranking.

NOTE: 16 other weaknesses were considered for inclusion in the Top 25, but their general scores were not high enough. They are listed in a separate "On the Cusp" page.

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

- Other terms you may hear (more specific)
  - *Underflow, over-read, out-of-bounds access*, etc.
  - Some use *buffer overflow* only for writing off the end
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
```

```
0x4bf mov %esp,%ebp
0x4b8 push %ebp
0x4c1 push %ecx
0x4c2 sub $0x224,%esp
```

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

```
...```

```
```

```
```

```
```
Location of data areas

- **Set when process starts**
- **Runtime**
- **Known at compile time**

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

malloc(sizeof(long));

static int x;

static const int y=10;
```
Memory allocation

Stack and heap grow in opposite directions

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

apportioned by the OS; managed in-process by `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

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

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

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

But can know the relative address
• `loc2` is always 8B before `???s`
Returning from functions

Q: How do we restore previous %ebp?

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

Q: How do we resume here?

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

Stack frame for `func` previous `%ebp`

0xffffffff

%ebp

%ebp

loc2 loc1 %ebp ??? arg1 arg2 arg3 caller's data
Instructions in memory

need to save this address: 0x4a7

%eip

Text

0x00000000

0xffffffff

0x5bf mov %esp,%ebp
0x5be push %ebp
...

0x49b mov $0x804...,(%esp)
0x4a2 call <func>
0x4a7 mov $0x0,%eax
0x4a8 mov $0xa,0x4(%esp)
0x493 movl $0xa,0x4(%esp)
...
Returning from functions

int main()
{
    ...\n    func("Hey", 10, -3);
    ...
}

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`

- **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

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

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

Upon return, sets %ebp to 0x0021654d

```
Me! \0
```

<table>
<thead>
<tr>
<th>Aut</th>
<th>4d 65 21 00</th>
<th>%eip</th>
<th>&amp;arg1</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer</td>
<td>SEGFAULT (0x00216551) (during subsequent access)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Benign outcome

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

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

Upon return, sets `%ebp` to 0x0021654d

```
ME! \0
```

<table>
<thead>
<tr>
<th>Auth</th>
<th>4d 65 21 00</th>
<th>%eip</th>
<th>&amp;arg1</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SEGFAULT (0x00216551)
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ... }
}

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

```c
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

---

**Diagram**

- `%eip` pointing to `buffer` with values: `00 00 00 00`
This is nontrivial

- Pulling off this attack requires getting a few things really right (and some things sorta right)

- Think about what is tricky about the attack
  - The key to defending it will be to make the hard parts really hard
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);
}
```

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

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

(Machine code)

(filename) argv envp

(Part of) your input
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)