Memory Safety and Buffer Overflows
(with some material from Mike Hicks and Dave Levin)
Administrative, etc.

- More undergrad research: Prof. Tudor Dumitras
  - http://www.umiacs.umd.edu/~tdumitra/summer-internship.html

- CS Undergrad forum:
Today’s agenda

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

“The software performs operations on a memory buffer, but it can read from or write to a memory location that is outside of the intended boundary of the buffer.” (NIST/CWE)
What is a buffer overflow?
What is a buffer overflow?
What is a buffer overflow?

- A **low-level** bug, typically in **C/C++**
  - Significant security implications!
What is a buffer overflow?

• A **low-level** bug, typically in **C/C++**
  • Significant security implications!

• If accidentally triggered, causes a crash
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?
Why study them?

- Buffer overflows are still relevant today
  - C and C++ are still popular
  - Buffer overflows still occur with regularity
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
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>99.7</td>
</tr>
<tr>
<td>3. Java</td>
<td>🌐💻</td>
<td>99.5</td>
</tr>
<tr>
<td>4. C++</td>
<td>📱💻</td>
<td>97.1</td>
</tr>
<tr>
<td>5. C#</td>
<td>🌐💻</td>
<td>87.7</td>
</tr>
<tr>
<td>6. R</td>
<td>🌐💻</td>
<td>87.7</td>
</tr>
<tr>
<td>7. JavaScript</td>
<td>🌐📱</td>
<td>85.6</td>
</tr>
<tr>
<td>8. PHP</td>
<td>🌐💻</td>
<td>81.2</td>
</tr>
<tr>
<td>9. Go</td>
<td>🌐💻</td>
<td>75.1</td>
</tr>
<tr>
<td>10. Swift</td>
<td>📱💻</td>
<td>73.7</td>
</tr>
</tbody>
</table>

Critical systems in C/C++
Critical systems in C/C++

• Most **OS kernels** and utilities
  • X windows server, shell
Critical systems in C/C++

• Most **OS kernels** and utilities
  • X windows server, shell

• Many **high-performance servers**
  • Microsoft IIS, Apache httpd, nginx
  • Microsoft SQL server, MySQL, redis, memcached
Critical systems in C/C++

- Most **OS kernels** and utilities
  - 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
Critical systems in C/C++

- Most **OS kernels** and utilities
  - 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

- **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
History of buffer overflows

The harm has been substantial

• 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

1988

2001

2003

• **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 @10:11am from the stack-smashing-for-fun-and-profit dept.

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.
23-Year-Old X11 Server Security Vulnerability Discovered

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)

https://nvd.nist.gov/vuln/search
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
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
Note about terminology

- We will use **buffer overflow** to mean *any access of a buffer outside of its allotted bounds*
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
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*
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
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)
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.
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/C process model
  • Similar to other operating systems
All programs stored in memory
All programs stored in memory
All programs stored in memory

The process’s view of memory is that it owns all of it
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
Program **instructions** are in memory

```
0xffffffff
0x00000000
```

```
0x4bf mov %esp,%ebp
0x4b e push %ebp
0x4c1 push %ecx
0x4c2 sub $0x224,%esp
...
```

...
Location of data areas

```
4G  0xffffffff
```

```
0  0x0000000000
```

Text
Location of data areas

4G — 0xffffffff

Init'd data

static const int y=10;

0xffffffff

Text

0x00000000
Location of data areas

4G -> 0xffffffff

4G

- Uninit’ed data
- Init’ed data
- Text

0x00000000

static int x;
static const int y = 10;
Location of data areas

Known at compile time

4G

0xffffffff

Uninit’d data

Init’d data

Text

static int x;
static const int y=10;
Location of data areas

4G

0xffffffff

cmdline & env

Uninit’d data

Init’d data

Text

static int x;

static const int y=10;

Known at compile time

0

0x00000000

0xffffffff

21-5
Location of data areas

- **Set when process starts**: cmdline & env
- **Known at compile time**: Uninit'd data, Init'd data, Text

Static int x;
Static const int y = 10;
Location of data areas

- **Set when process starts**
  - cmdline & env
  - Stack

- **Known at compile time**
  - Uninit’d data
  - Init’d data
  - Text

Code example:

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

static int x;
static const int y=10;
```
Location of data areas

Set when process starts

Known at compile time

4G

cmdline & env

Stack

Heap

Uninit'd data

Init'd data

Text

0xffffffff

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

malloc(sizeof(long));

static int x;

static const int y=10;
Location of data areas

Set when process starts

Runtime

Known at compile time

- Text
- Init'd data
- Uninit'd data
- Heap
- Stack
- cmdline & env

- int f() {
  int x;
  ... 
}
- malloc(sizeof(long));
- static int x;
- static const int y = 10;
Memory allocation

Stack and heap grow in opposite directions

0x00000000 0xffffffff

Heap      Stack
Memory allocation

Stack and heap grow in opposite directions

0x00000000 0xffffffff

Heap Stack

Stack pointer
Memory allocation

Stack and heap grow in opposite directions

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

Stack pointer
Memory allocation

Stack and heap grow in opposite directions

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

```
0x00000000

Heap

Stack

0xffffffff

Stack pointer
```

- push 1
- push 2
- push 3
Memory allocation

Stack and heap grow in opposite directions

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

Stack pointer

push 1
push 2
push 3
Memory allocation

Stack and heap grow in opposite directions

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

Stack pointer

push 1
push 2
push 3
Memory allocation

Stack and heap grow in opposite directions

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

Stack pointer

push 1
push 2
push 3
Memory allocation

Stack and heap grow in opposite directions

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

```
push 1
push 2
push 3
```

<table>
<thead>
<tr>
<th>Heap</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

Stack pointer

```
0x00000000 0xffffffff
```
Memory allocation

Stack and heap grow in opposite directions

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

```
0x00000000

Stack

0xffffffff

Heap
```

22-9
Memory allocation

Stack and heap grow in opposite directions

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

```
0x00000000  0xffffffff
```

```
| Heap | 2 | 1 | Stack |
```

Stack pointer

push 1
push 2
push 3
Memory allocation

Stack and heap grow in opposite directions

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

Stack pointer

push 1
push 2
push 3
Memory allocation

Stack and heap grow in opposite directions

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

Stack pointer

push 1
push 2
push 3
return
Memory allocation

Stack and heap grow in opposite directions

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

```
0x00000000  0xffffffff
```

```
<table>
<thead>
<tr>
<th>Heap</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>Stack</th>
</tr>
</thead>
</table>
```

Stack pointer

- push 1
- push 2
- push 3
- return
Memory allocation

Stack and heap grow in opposite directions

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

Heap

Stack

apportioned by the OS; managed in-process by malloc

Stack pointer

push 1
push 2
push 3
return
Memory allocation

Stack and heap grow in opposite directions

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

Heap apportioned by the OS; managed in-process by malloc

Stack pointer

Focusing on the stack for now
Stack and function calls
Stack and function calls

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

0xffffffff

caller’s data
Basic stack layout

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

Arguments pushed in reverse order of code

0xffffffff

caller's data
Basic stack layout

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

Arguments pushed in reverse order of code

0xffffffff

<table>
<thead>
<tr>
<th>arg1</th>
<th>arg2</th>
<th>arg3</th>
<th>caller's data</th>
</tr>
</thead>
</table>

24-3
Basic stack layout

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

Arguments pushed in reverse order of code

Happens during caller

<table>
<thead>
<tr>
<th>arg1</th>
<th>arg2</th>
<th>arg3</th>
<th>caller's data</th>
</tr>
</thead>
</table>

0xffffffff
Basic stack layout

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

Arguments pushed in reverse order of code

Happens during caller
Basic stack layout

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

Happens during caller 24-6

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

Arguments pushed in reverse order of code

Arguments pushed in reverse order of code
Basic stack layout

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

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

Arguments pushed in reverse order of code

0xffffffff

<table>
<thead>
<tr>
<th>...</th>
<th>loc2</th>
<th>loc1</th>
<th>???</th>
<th>???</th>
<th>arg1</th>
<th>arg2</th>
<th>arg3</th>
<th>caller’s data</th>
</tr>
</thead>
</table>

Happens during callee

Happens during caller
Basic stack layout

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

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++;
    ...
}
```

![Accessing variables table](image)

25-1
Accessing variables

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

Q: Where is (this) loc2?

0xffffffff

<table>
<thead>
<tr>
<th>...</th>
<th>loc2</th>
<th>loc1</th>
<th>???</th>
<th>???</th>
<th>arg1</th>
<th>arg2</th>
<th>arg3</th>
<th>caller's data</th>
</tr>
</thead>
</table>

---
Accessing variables

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

Q: Where is (this) `loc2`?

```
... loc2 loc1 ??? ??? arg1 arg2 arg3 caller's data
```

0xbffffff323

0xffffffff
Accessing variables

Q: Where is (this) `loc2`?

Can’t know absolute address at compile time
Accessing variables

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

Q: Where is (this) loc2?

Can’t know absolute address at compile time

But can know the relative address
• loc2 is always 8B before ???s
Accessing variables

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

Q: Where is (this) `loc2`?

![Table showing variable locations]

But can know the `relative` address
- `loc2` is always 8B before `??`'s
Accessing variables

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

Q: Where is (this) loc2?

But can know the relative address

- **loc2** is always 8B before ???'s

Stack frame for `func`

<table>
<thead>
<tr>
<th>loc2</th>
<th>loc1</th>
<th>???</th>
<th>???</th>
<th>arg1</th>
<th>arg2</th>
<th>arg3</th>
<th>caller's data</th>
</tr>
</thead>
</table>

0xffffffff
Accessing variables

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

Q: Where is (this) loc2?

But can know the relative address

• loc2 is always 8B before ???s
Accessing variables

void func(char *arg1, int arg2, int arg3) {
    ...
    loc2++;  // Q: Where is (this) loc2?
    ...
}

A: -8(%ebp)

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

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

Stack frame for func
Returning from functions

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

Stack frame for `func`
Returning from functions

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

Stack frame for `func`:

- `%ebp`
- Previous `%ebp`
- Caller's data
- `0xffffffff`
- `... loc2 loc1 ??? ??? arg1 arg2 arg3`
Returning from functions

Q: How do we restore previous %ebp?

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

Stack frame for func

previous %ebp
Returning from functions

Q: How do we restore previous %ebp?

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

Stack frame for func

previous %ebp

 caller's data

0xffffffff

??? arg1 arg2 arg3

26-5
Returning from functions

Q: How do we restore previous %ebp?

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

Stack frame for func previous %ebp

ESP

Previous %ebp
Returning from functions

Q: How do we restore previous %ebp?

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

Push current %ebp before locals
Returning from functions

Q: How do we restore previous %ebp?

Push current %ebp before locals
Set %ebp to current %esp
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

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

...  
0x4a7 mov $0x0, %eax  
0x4a2 call <func>  
0x49b movl $0x4..(%esp)  
0x493 movl $0xa, 0x4(%esp)  
...
Instructions in memory

4G - 0xffffffff

Text

0 0x00000000

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

...
Instructions in memory

4G  0xffffffff

0  0x00000000

Text

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

%eip
Instructions in memory

0xffffffff

0xffffffff

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

%eip
Instructions in memory

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

... 0x4a7 mov $0x0,%eax
0x4a2 call <func>
0x49b movl $0x804..(%esp)
0x493 movl $0xa,0x4(%esp)
...
Instructions in memory

need to save this address: 0x4a7

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

0x4a7 mov $0x0,%eax
0x4a2 call <func>
0x49b movl $0x804..,(%esp)
0x493 movl $0xa,0x4(%esp)
...
Instructions in memory

need to save this address:
0x4a7

Text

4G

%eip

0xffffffff

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

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

%eip

0x00000000
Returning from functions

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

Q: How do we resume here?
Returning from functions

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

Q: How do we resume here?

Push next %eip before call
Returning from functions

Q: How do we resume here?

Push next `%eip` before call
Returning from functions

```c
int main()
{
    ...
    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
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
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
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 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
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
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

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

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

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

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

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

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

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

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

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

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

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

```plaintext
00 00 00 00 00  %ebp  %eip  &arg1
buffer
```
Benign outcome

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

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
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 4d 65 21 00 %eip &arg1

buffer
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

M e ! \0

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

buffer SEGFAULT (0x00216551)
Security-relevant outcome

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

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ... 
}
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ... }
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ... 
}
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);
    ... }
```
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ... }
}

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

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

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

00 00 00 00 00 00 00 00 %ebp %eip &arg1

buffer authenticated
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);
    ... 
}
```

<table>
<thead>
<tr>
<th>Auth</th>
<th>00 00 00 00</th>
<th>ebp</th>
<th>eip</th>
<th>&amp;arg1</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer</td>
<td>authenticated</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ... }
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ... 
}
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ...
}

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

`buffer`

`00 00 00 00 %ebp %eip &mystr`
Could it be worse?

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

All ours!

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

```plaintext
buffer

&mystr
&ebp &esp
```

35-3
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?

All ours!
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
Aside: User-supplied strings

• These examples provide their own strings
Aside: User-supplied strings

- These examples provide their own strings
- In reality strings come from *users* in myriad ways
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…
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!
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);
    ...
}
```

![Diagram of code injection](image)
Code Injection: Main idea

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

(1) Load my own code into memory
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
**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
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
Challenge 1

Loading code into memory

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

- One goal: general-purpose shell
  - Command-line prompt that gives attacker general access to the system
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**
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);
}
```
Shellcode

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

41-3
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 %esp,%ebx
pushl %eax...
```
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
```
"\x31\xc0"
"\x50"
"\x68""/sh"
"\x68""/bin"
"\x89\xe3"
"\x50"
...```

(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
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

```
%eip
```

```
buffer
```
### 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

<table>
<thead>
<tr>
<th>Text</th>
<th>...</th>
<th>00 00 00 00</th>
<th><code>%ebp</code></th>
<th><code>%eip</code></th>
<th><code>&amp;arg1</code></th>
<th>...</th>
<th><code>\x0f \x3c \x2f ...</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

%eip
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
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`

![Diagram of memory layout showing `%eip` and `%ebp` addresses within a buffer and an instruction at 0xbfff]
Hijacking the saved %eip
Hijacking the saved %eip

Diagram showing the memory layout with %ebp and %eip markers and a buffer with data. The diagram highlights the hijacking of the saved %eip value.
Hijacking the saved `%eip`

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

What if we are wrong?
Hijacking the saved \%eip

What if we are wrong?
Hijacking the saved `%eip`

What if we are wrong?
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
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
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
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: \textcolor{red}{nop} \textcolor{red}{sleds}

\textcolor{red}{nop} is a single-byte no-op instruction (just moves to the next instruction)
Improving our chances: **nop sleds**

**nop** is a single-byte no-op instruction (just moves to the next instruction)
Improving our chances: **nop sleds**

**nop** is a single-byte no-op instruction (just moves to the next instruction)

Jumping *anywhere* here will work

---

0xbff

---

0xbff
Improving our chances: \textcolor{red}{nop} sleds

\textcolor{red}{nop} is a single-byte no-op instruction (just moves to the next instruction)
Improving our chances: **nop sleds**

**nop** is a single-byte no-op instruction (just moves to the next instruction)
Improving our chances: **nop sleds**

**nop** is a single-byte no-op instruction (just moves to the next instruction)
Improving our chances: **nop sleds**

**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
Putting it all together

%eip

Buffer

Malicious code
Putting it all together

%eip

Text ... 00 00 00 00 %ebp %eip nop nop nop ... \x0f \x3c \x2f ...

buffer

nop sled

malicious code
Putting it all together

%eip

Text ... 00 00 00 00 %ebp 0xbdf nop nop nop ... \x0f \x3c \x2f ...

buffer

good guess

nop sled

malicious code
Putting it all together

%eip padding good guess

Text \ldots \x0f \x3c \x2f \ldots

buffer nop sled malicious code
Putting it all together

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

%eip

Text ...

padding

good guess

0xbdf

nop nop nop ...

\x0f \x3c \x2f ...

buffer

nop sled

malicious code
Putting it all together

Fill in the space between the target buffer and the `%eip` to overwrite.
Putting it all together

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

Padding

Good guess

%d

nop sled

Malicious code

Text

...
gdb tutorial
Your new best friends

```
if
```

Show **info** about the current **frame**
(prev. frame, locals/args, %ebp/%eip)

```
ir
```

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)