Last time

We kicked off our 1st section: Software Security

By launching Buffer overflows and other memory safety vulnerabilities

• Buffer overflow fundamentals
This time

We will continue

**Buffer overflows**

and other memory safety vulnerabilities

By looking at

**Overflow Atks/Defs**

- Finish stack layout review
- Attack!
- Everything you’ve always wanted to know about `gdb` but were too afraid to ask
- Overflow defenses
Stack layout when calling functions

```c
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int loc2;
    int loc3;
}
```

0x000000000000 - 0xffffffff

caller’s data
Stack layout when calling functions

```c
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
}
```

Arguments pushed in reverse order of code
Stack layout when calling functions

```c
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int loc2;
    int loc3;
}
```

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

Arguments pushed in reverse order of code
Stack layout when calling functions

```c
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int loc2;
    int loc3;
}
```

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

Arguments pushed in reverse order of code
Stack layout when calling functions

```c
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int loc2;
    int loc3;
}
```

Two 4B values between the arguments and the local variables

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

Arguments pushed in reverse order of code
Stack layout when calling functions

```c
void func(char *arg1, int arg2, int arg3) {
    char loc1[4]
    int loc2;
    int loc3;
}
```

Two 4B values between the arguments and the local variables

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

Arguments pushed in reverse order of code

Let's explore (and attack and defend) them
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    ...
    loc2++;
    ...
}
void func(char *arg1, int arg2, int arg3) {
    char loc1[4]
    int loc2;
    int loc3;
    ...
    loc2++;  \textbf{Q: Where is (this) loc2?}
    ...
}

\begin{tabular}{c c c c c c c c}
... & loc2 & loc1 & ??? & ??? & arg1 & arg2 & arg3 & caller\'s data \\
\end{tabular}

\begin{tabular}{c c c c c c c c c c c c}
0x00000000 & & & & & & & & & & 0xffffffff
\end{tabular}
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int loc2;
    int loc3;
    ...
    loc2++;
    ...
}

Q: Where is (this) loc2?

Accessing variables
void func(char *arg1, int arg2, int arg3) {
    char loc1[4]
    int loc2;
    int loc3;
    ...
    loc2++;
    ...
}

Q: Where is (this) loc2?

Undecidable at compile time
void func(char *arg1, int arg2, int arg3) {
    char loc1[4]
    int loc2;
    int loc3;
    ...
    loc2++;
    ...
}

Q: Where is (this) loc2?

Undecidable at compile time

- I don't know where loc2 is,
Accessing variables

```c
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int loc2;
    int loc3;
    ...
    loc2++;    // Q: Where is (this) loc2?
    ...
}
```

Undecidable at compile time

- I don’t know where loc2 is,
- and I don’t know how many args
void func(char *arg1, int arg2, int arg3) {
    char loc1[4]
    int loc2;
    int loc3;
    ...
    loc2++;
    ...
}

Q: Where is (this) loc2?

Undecidable at compile time
- I don’t know where loc2 is,
- and I don’t know how many args
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int loc2;
    int loc3;
    ...
    loc2++;
    ...
}

Q: Where is (this) loc2?

Undecidable at compile time

- I don’t know where loc2 is,
- and I don’t know how many args
- but loc2 is always 8B before “???”s
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int loc2;
    int loc3;
    ...
    loc2++;
    ...
}

- I don’t know where loc2 is,
- and I don’t know how many args
- but loc2 is always 8B before “???”s
Accessing variables

```c
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int loc2;
    int loc3;
    ...
    loc2++;
    ...
}
```

Q: Where is (this) loc2?

- I don't know where loc2 is,
- and I don't know how many args
- *but* loc2 is *always* 8B before “???”'s
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int loc2;
    int loc3;
    ...
    loc2++;
    ...
}

Q: Where is (this) loc2?

- I don’t know where loc2 is,
- and I don’t know how many args
- but loc2 is always 8B before “???”s
```c
void func(char *arg1, int arg2, int arg3) {
    char loc1[4]
    int loc2;
    int loc3;
    ...
    loc2++;
    ...
}
```

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

- I don’t know where loc2 is,
- and I don’t know how many args
- but loc2 is always 8B before “???”'s
Notation

%ebp A memory address

(%ebp) The value at memory address %ebp (like dereferencing a pointer)
Notation

%ebp  A memory address

( %ebp ) The value at memory address %ebp
(like dereferencing a pointer)
Notation

$0xbfff03b8$  \texttt{%ebp}  A memory address

($\texttt{%ebp}$) The value at memory address $\texttt{%ebp}$ (like dereferencing a pointer)
Notation

0xbffff03b8 %ebp A memory address

(%ebp) The value at memory address %ebp (like dereferencing a pointer)
Notation

0xbfff03b8  %ebp  A memory address

0xbfff0720  (%ebp)  The value at memory address %ebp (like dereferencing a pointer)
Notation

0xbffff03b8 \%ebp A memory address

0xbffff0720 (\%ebp) The value at memory address \%ebp (like dereferencing a pointer)

`pushl \%ebp`
Notation

0xbffff03b8 %ebp A memory address

0xbffff0720 (%ebp) The value at memory address %ebp (like dereferencing a pointer)

pushl %ebp
Notation

0xbfff03b8 %ebp A memory address

0xbfff0720 (%ebp) The value at memory address %ebp (like dereferencing a pointer)

pushl %ebp
Notation

0xbfff03b8 %ebp A memory address

0xbfff0720 (%ebp) The value at memory address %ebp (like dereferencing a pointer)

pushl %ebp
Notation

0xbfff03b8 \texttt{%ebp} A memory address

0xbfff0720 (\texttt{%ebp}) The value at memory address \texttt{%ebp} (like dereferencing a pointer)

\texttt{pushl \texttt{%ebp}}
Notation

0xbfff03b8  %ebp  A memory address

0xbfff0720  (%ebp)  The value at memory address %ebp (like dereferencing a pointer)

pushl %ebp
movl %esp %ebp  /* %ebp = %esp */
Notation

%ebp  A memory address

(%ebp)  The value at memory address %ebp (like dereferencing a pointer)

pushl %ebp
movl %esp %ebp  /* %ebp = %esp */
### Notation

- `%ebp` is a memory address.
- The value at memory address `%ebp` (like dereferencing a pointer).

```assembly
pushl %ebp
movl %esp %ebp /* %ebp = %esp */
```

---

**Diagram:**
- `%ebp` and `%esp` points to different memory addresses: `%ebp` at `0xbfff03b8` and `%esp` at `0xbfff0200`.
- The addresses `0xbfff03b8` and `0xbfff0720` are highlighted.

---

**Values:**
- `0xbfff03b8`: `%ebp`
- `0xbfff0720`: `( %ebp)`
- `0xbfff0200`: `%esp`
- `0xbfff0000`: `%ebp` (base pointer)
- `0xffffffff`: `%ebp` (maximum value)
- `0x00000000`: `%ebp` (minimum value)
Notation

%ebp  A memory address

( %ebp )  The value at memory address %ebp
(like dereferencing a pointer)

pushl %ebp
movl %esp %ebp  /* %ebp = %esp */
Notation

%ebp  A memory address

(%ebp)  The value at memory address %ebp (like dereferencing a pointer)

pushl %ebp
movl %esp %ebp  /* %ebp = %esp */
Notation

%ebp  A memory address

(%ebp) The value at memory address %ebp (like dereferencing a pointer)

pushl %ebp
movl %esp %ebp /* %ebp = %esp */
movl (%ebp) %ebp /* %ebp = (%ebp) */

0xbfff0200 %esp

0xbfff03b8

0xbfff0200 0xbfff03b8 0xbfff0720

0x00000000 0xffffffff

0x00000000 0xbfff0720
Notation

\%ebp  A memory address

(\%ebp)  The value at memory address\%ebp
   (like dereferencing a pointer)

\textbf{pushl} \%ebp
\textbf{movl} \%esp \%ebp  /* \%ebp = \%esp */
\textbf{movl} (\%ebp) \%ebp  /* \%ebp = (\%ebp) */
Returning from functions

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

Stack frame for this call to `func`
Returning from functions

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

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

Stack frame for this call to `func`
Returning from functions

```c
int main()
{
    ...
    func(“Hey”, 10, -3);
    … Q: How do we restore %ebp?
}
```

Q: How do we restore %ebp?

Stack frame for this call to `func`
Returning from functions

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

Q: How do we restore %ebp?
```

Stack frame for this call to `func`
Returning from functions

```c
int main()
{
    ...
    func("Hey", 10, -3);
    ... Q: How do we restore %ebp?
}
```

Stack frame for this call to `func`

Q: How do we restore %ebp?
Returning from functions

```c
int main()
{
    ...
    func("Hey", 10, -3);
    ...
    Q: How do we restore %ebp?
}
```

1. Push %ebp before locals

**Stack frame for this call to func**

- %ebp
- ???
- arg1
- arg2
- arg3
- caller's data
Returning from functions

```
int main()
{
    ...
    func("Hey", 10, -3);
    ...
Q: How do we restore %ebp?
}
```

1. Push %ebp before locals
2. Set %ebp to current %esp
Returning from functions

```
int main()
{
    ...
    func("Hey", 10, -3);
    ...  Q: How do we restore %ebp?
}
```

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

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

Stack frame for this call to `func`

%ebp

 caller's data

arg3
arg2
arg1
???
loc2
loc1
%ebp

0x0000000000
0xffffffff
Returning from functions

int main()
{
    ...
    func("Hey", 10, -3);
    ...
    \textbf{Q: How do we resume here?}
}

\textbf{Stack frame for this call to \textit{func}}
The instructions themselves are in memory

4G

0xffffffff

0x00000000

0x0000000000

Text

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

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

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

```
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
The instructions themselves are in memory

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

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

%eip

---

```
0x00000000
```

---

```
Text
```

---

```
4G
```

---

```
0
```
The instructions themselves are in memory

The diagram shows a memory map with the following instructions:

- `0xffffffff`: This is likely the instruction pointer (`eip`), indicating the current instruction to be executed.
- `0x5bf mov %esp,%ebp`
- `0x5be push %ebp`
- `0x4a7 mov $0x804..,(%esp)`
- `0x4a2 call <func>`
- `0x49b movl $0xa,0x4(%esp)`
- `0x493 movl $0x0,%eax`

These instructions are likely part of a function call or an instruction to manipulate the stack pointer (`%esp`), with `%ebp` being used for temporary storage.
The instructions themselves are in memory

4G

0xffffffff

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

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

0x0000000000
Returning from functions

```c
int main()
{
    ...
    func(“Hey”, 10, -3);
    ...
    Q: How do we resume here?
}
```

Q: How do we resume here?
Returning from functions

```c
int main()
{
    ...
    func("Hey", 10, -3);
    ...
    Q: How do we resume here?
}
```

Stack frame for this call to `func`

Push next `%eip` before call
int main()
{
    ...
    func("Hey", 10, -3);
    ...

Q: How do we resume here?

Stack frame for this call to func

Push next %eip before call
int main()
{
    ...
    func("Hey", 10, -3);
    ...  \textbf{Q: How do we resume here?}
}

\textbf{Set \%eip to 4(\%ebp) at return}

\textbf{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: %eip+something
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: `%eip+something`
3. **Jump to the function’s address**

**Called function:**
4. **Push the old frame pointer** onto the stack: `%ebp`
5. **Set frame pointer** `%ebp` to where the end of the stack is right now: `%esp`
6. **Push local variables** onto the stack; access them as offsets from `%ebp`
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: %eip+something
3. Jump to the function’s address

Called function:
4. Push the old frame pointer onto the stack: %ebp
5. Set frame pointer %ebp to where the end of the stack is right now: %esp
6. Push local variables onto the stack; access them as offsets from %ebp

Returning function:
7. Reset the previous stack frame: %ebp = (%ebp) /* copy it off first */
8. Jump back to return address: %eip = 4(%ebp) /* use the copy */
Buffer overflows
Buffer overflows from 10,000 ft

• **Buffer =**
  • Contiguous set of a given data type
  • Common in C
    - All strings are buffers of `char`'s

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

• **Where does the extra data go?**

• **Well now that you’re experts in memory layouts…**
A buffer overflow example

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

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

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

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

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

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

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

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

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

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

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

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

<table>
<thead>
<tr>
<th>00 00 00 00</th>
<th>%ebp</th>
<th>%eip</th>
<th>&amp;arg1</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A buffer overflow example

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

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

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

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

```
%ebp 4d 65 21 00
%eip 00 00 00 00
```

buffer

```
A  u  t  h
M  e  !  \0
```

Auth 4d 65 21 00 %eip &arg1
A buffer overflow example

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

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

Upon return, sets %ebp to 0x0021654d

\text{M e ! \0}

\begin{tabular}{|l|l|l|l|}
\hline
Auth & 4d 65 21 00 & %eip & &arg1 \\
\hline
buffer
\end{tabular}
A buffer overflow example

```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)
A buffer overflow example

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

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

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

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

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

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

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

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

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

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

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

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

```c
00 00 00 00 %ebp %eip &arg1
```

authenticated
A buffer overflow example

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

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

```assembly
00 00 00 00 | 00 00 00 00 | %ebp | %eip | &arg1
buffer       authenticated
```
A buffer overflow example

```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>
A buffer overflow example

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

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

```bash
M e ! \0
Auth 4d 65 21 00 %ebp %eip &arg1
buffer authenticated
```
A buffer overflow example

```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’
void vulnerable()
{
    char buf[80];
    gets(buf);
}
void vulnerable()
{
    char buf[80];
    gets(buf);
}

void still_vulnerable()
{
    char *buf = malloc(80);
    gets(buf);
}
void safe()
{
    char buf[80];
    fgets(buf, 64, stdin);
}
```c
void safe()
{
    char buf[80];
    fgets(buf, 64, stdin);
}

void safer()
{
    char buf[80];
    fgets(buf, sizeof(buf), stdin);
}
```
IE's Role in the Google-China War

By Richard Adhikari
TechNewsWorld
01/15/10 12:25 PM PT

The hack attack on Google that set off the company's ongoing standoff with China appears to have come through a zero-day flaw in Microsoft's Internet Explorer browser. Microsoft has released a security advisory, and researchers are hard at work studying the exploit. The attack appears to consist of several files, each a different piece of malware.

Computer security companies are scurrying to cope with the fallout from the Internet Explorer (IE) flaw that led to cyberattacks on Google and its corporate and individual customers.

The zero-day attack that exploited IE is part of a lethal cocktail of malware that is keeping researchers very busy.

"We're discovering things on an up-to-the-minute basis, and we've seen about a dozen files dropped on infected PCs so far," Dmitri Alperovitch, vice president of research at McAfee Labs, told TechNewsWorld.

The attacks on Google, which appeared to originate in China, have sparked a feud between the Internet giant and the nation's government over censorship, and it could result in Google pulling away from its business dealings in the country.

Pointing to the Flaw

The vulnerability in IE is an invalid pointer reference, Microsoft said in security advisory 979352, which it issued on Thursday. Under certain conditions, the invalid pointer can be accessed after an object is deleted, the advisory states. In specially crafted attacks, like the ones launched against Google and its customers, IE can allow remote execution of code when the flaw is exploited.
User-supplied strings

• In these examples, we were providing our own strings

• But they come from users in myriad ways
  • Text input
  • Network packets
  • Environment variables
  • File input…
What’s the worst that could happen?

```c
void func(char *arg1) {
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
```
What’s the worst that could happen?

```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’s the worst that could happen?

```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`)

buffer
What’s the worst that could happen?

```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!
First, a recap (args)

```c
#include <stdio.h>

void func(char *arg1, int arg2, int arg3)
{
    printf("arg1 is at %p\n", &arg1);
    printf("arg2 is at %p\n", &arg2);
    printf("arg3 is at %p\n", &arg3);
}

int main()
{
    func("Hello", 10, -3);
    return 0;
}
```
First, a recap (args)

```c
#include <stdio.h>

void func(char *arg1, int arg2, int arg3)
{
    printf(“arg1 is at %p\n”, &arg1);
    printf(“arg2 is at %p\n”, &arg2);
    printf(“arg3 is at %p\n”, &arg3);
}

int main()
{
    func(“Hello”, 10, -3);
    return 0;
}
```

What will happen?

```c
&arg1 < &arg2 < &arg3?    &arg1 > &arg2 > &arg3?
```
First, a recap (locals)

```c
#include <stdio.h>

void func()
{
    char loc1[4];
    int  loc2;
    int  loc3;
    printf("loc1 is at %p\n", &loc1);
    printf("loc2 is at %p\n", &loc2);
    printf("loc3 is at %p\n", &loc3);
}

int main()
{
    func();
    return 0;
}
```
First, a recap (locals)

```c
#include <stdio.h>

void func()
{
    char loc1[4];
    int loc2;
    int loc3;
    printf("loc1 is at %p\n", &loc1);
    printf("loc2 is at %p\n", &loc2);
    printf("loc3 is at %p\n", &loc3);
}

int main()
{
    func();
    return 0;
}
```

What will happen?

&loc1 < &loc2 < &loc3?    &loc1 > &loc2 > &loc3?
Stack and functions: Summary

%eip

code

%ebp

caller's data
Stack and functions: Summary

**Calling function:**

1. **Push arguments** of the function you’re calling 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: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**
Stack and functions: Summary

**Calling function:**
1. **Push arguments** of the function you’re calling 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: the current `%eip + (some amount)
3. **Jump to the address of the function you are calling**
Stack and functions: Summary

**Calling function:**
1. **Push arguments** of the function you’re calling 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: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**
Calling function:

1. **Push arguments** of the function you’re calling 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: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**
**Stack and functions: Summary**

**Calling function:**
1. **Push arguments** of the function you’re calling 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: the current `%eip + (some amount)`
3. **Jump to the address of the function you are calling**
Stack and functions: Summary

**Calling function:**
1. **Push arguments** of the function you’re calling 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: the current `%eip` + (some amount)
3. **Jump to the address of the function you are calling**
Stack and functions: Summary

**Calling function:**
1. **Push arguments** of the function you’re calling 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: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**

**Called function:**
4. **Push the old frame pointer** onto the stack: %ebp
5. **Set frame pointer** %ebp to where the end of the stack is right now: %esp
6. **Push local variables** onto the stack; access them as offsets from %ebp
Stack and functions: Summary

**Calling function:**
1. **Push arguments** of the function you’re calling 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: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**

**Called function:**
4. **Push the old frame pointer** onto the stack: %ebp
5. **Set frame pointer** %ebp to where the end of the stack is right now: %esp
6. **Push local variables** onto the stack; access them as offsets from %ebp
Stack and functions: Summary

**Calling function:**
1. **Push arguments** of the function you’re calling 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: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**

**Called function:**
4. **Push the old frame pointer** onto the stack: %ebp
5. **Set frame pointer** %ebp to where the end of the stack is right now: %esp
6. **Push local variables** onto the stack; access them as offsets from %ebp
Stack and functions: Summary

**Calling function:**
1. **Push arguments** of the function you’re calling 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: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**

**Called function:**
4. **Push the old frame pointer** onto the stack: %ebp
5. **Set frame pointer** %ebp to where the end of the stack is right now: %esp
6. **Push local variables** onto the stack; access them as offsets from %ebp
Stack and functions: Summary

**Calling function:**
1. **Push arguments** of the function you’re calling 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: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**

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

![Diagram showing stack and function calling](image)
Stack and functions: Summary

**Calling function:**
1. **Push arguments** of the function you’re calling 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: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**

**Called function:**
4. **Push the old frame pointer** onto the stack: %ebp
5. **Set frame pointer** %ebp to where the end of the stack is right now: %esp
6. **Push local variables** onto the stack; access them as offsets from %ebp
Stack and functions: Summary

Calling function:
1. **Push arguments** of the function you’re calling 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: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**

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

Returning function:
7. **Reset the previous stack frame**: %ebp = (%ebp)
8. **Jump back to return address**: %eip = 4(%ebp)
Stack and functions: Summary

**Calling function:**
1. **Push arguments** of the function you’re calling 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: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**

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

**Returning function:**
7. **Reset the previous stack frame**: %ebp = (%ebp)
8. **Jump back to return address**: %eip = 4(%ebp)
Stack and functions: Summary

**Calling function:**
1. **Push arguments** of the function you’re calling 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: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**

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

**Returning function:**
7. **Reset the previous stack frame**: %ebp = (%ebp)
8. **Jump back to return address**: %eip = 4(%ebp)
gdb: your new best friend

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

i r
Show info about registers (%ebp, %eip, %esp, etc.)

x/<n> <addr>
Examine <n> bytes of memory starting at address <addr>

b <function>
Set a breakpoint at <function>
s step through execution (into calls)
Buffer overflow

char loc1[4];
Buffer overflow

```c
char loc1[4];
gets(loc1);
strcpy(loc1, <user input>);
memcpy(loc1, <user input>);
```

etc.
Buffer overflow

code

loc2

char loc1[4];

Input writes from low to high addresses

gets(loc1);
strcpy(loc1, <user input>);
memcp(loc1, <user input>);

etc.
Buffer overflow

```c
char loc1[4];
gets(loc1);
strcpy(loc1, <user input>);
memcpy(loc1, <user input>);
```

Input writes from low to high addresses
Buffer overflow

Can over-write other data ("AuthMe!")

code | loc2 | loc1 | %ebp | %eip+... | arg1 | arg2 | caller's data

Input writes from low to high addresses

gets(loc1);
strcpy(loc1, <user input>);
memcpy(loc1, <user input>);
etc.
Buffer overflow

Can over-write other data (“AuthMe!”)

Can over-write the program’s control flow (%eip)

code  loc2  loc1  %ebp  %eip+...  arg1  arg2  caller’s data

char loc1[4];

gets(loc1);
strcpy(loc1, <user input>);
memcpy(loc1, <user input>);
etc.

Input writes from low to high addresses
Code injection
High-level idea

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

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

buffer
High-level idea

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

(1) Load our own code into memory
High-level idea

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

(1) Load our own code into memory
(2) Somehow get %eip to point to it
High-level idea

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

1. Load our own code into memory
2. Somehow get %eip to point to it
High-level idea

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

(1) Load our own code into memory
(2) Somehow get %eip to point to it
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 could you write assembly to never contain a full zero byte?
  • It can’t make use of the loader (we’re injecting)
  • It can’t use the stack (we’re going to smash it)
What kind of code would we want to run?

• Goal: **full-purpose shell**
  • The code to launch a shell is called “**shell code**”
  • It is nontrivial to do it in a way that works as injected code
    - No zeroes, can’t use the stack, no loader dependence
• There are many out there
  - And competitions to see who can write the smallest

• Goal: **privilege escalation**
  • Ideally, they go from guest (or non-user) to root
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 %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
movl %esp,%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"
...```
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"/\x68"/\xe3\xe5\x50"
```

(Part of) your input
Privilege escalation

• More on Unix permissions later, but for now…

• Recall that each file has:
  • Permissions: read / write / execute
  • For each of: owner / group / everyone else

• Permissions are defined over userid’s and groupid's
  • Every user has a userid
  • root’s userid is 0

• Consider a service like passwd
  • Owned by root (and needs to do root-y things)
  • But you want any user to be able to execute it
Real vs. Effective userid

- (Real) Userid = the user who ran the process
- Effective userid = what is used to determine what permissions/access the process has
- Consider passwd: root owns it, but users can run it
  - getuid() will return who ran it (real userid)
  - seteuid(0) to set the effective userid to root
    - It’s allowed to because root is the owner
- What is the potential attack?
Real vs. Effective userid

• (Real) Userid = the user who ran the process

• Effective userid = what is used to determine what permissions/access the process has

• Consider passwd: root owns it, but users can run it
  • getuid() will return who ran it (real userid)
  • seteuid(0) to set the effective userid to root
    - It’s allowed to because root is the owner

• What is the potential attack?

  If you can get a root-owned process to run setuid(0)/seteuid(0), then you get root permissions
Challenge 2

Getting our injected code to run

- All we can do is write to memory from buffer onward
  - With this alone we want to get it to jump to our code
  - We have to use whatever code is already running

Thoughts?

```plaintext
... 00 00 00 00 %ebp %eip &arg1 ...

buffer
```
Challenge 2

Getting our injected code to run

- All we can do is write to memory from buffer onward
  - With this alone we want to get it to jump to our code
  - We have to use whatever code is already running

Thoughts?
Challenge 2

Getting our injected code to run

- All we can do is write to memory from buffer onward
  - With this alone we want to get it to jump to our code
  - We have to use whatever code is already running

Thoughts?
Challenge 2

Getting our injected code to run

- All we can do is write to memory from buffer onward
  - With this alone we want to get it to jump to our code
  - We have to use whatever code is already running

Thoughts?
Challenge 2

Getting our injected code to run

- All we can do is write to memory from buffer onward
  - With this alone we want to get it to jump to our code
  - We have to use whatever code is already running

Thoughts?
Challenge 2

Getting our injected code to run

- *All we can do is write to memory from buffer onward*
  - With this alone we want to get it to jump to our code
  - We have to use whatever code is already running

Thoughts?
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: %eip+something
3. **Jump to the function’s address**

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

**Returning function:**
7. **Reset the previous stack frame**: %ebp = (%ebp)
8. **Jump back to return address**: %eip = 4(%ebp)
Hijacking the saved `%eip`
Hijacking the saved %eip
Hijacking the saved `%eip`

![Diagram showing the hijacking of the saved `%eip`]

- `%ebp` points to the buffer
- `%eip` points to the instruction `0xbff`
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?

```
Text ... 00 00 00 00 %ebp 0xbdf &arg1 ... \x0f \x3c \x2f ... 0xbff
```

buffer
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: just try a lot of different values!
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: just 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: just 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

• But without address randomization:
  • The stack always starts from the same, fixed address
  • The stack will grow, but usually it doesn’t grow very deeply (unless the code is heavily recursive)
Improving our chances: \texttt{nop} sleds

\texttt{nop} is a single-byte instruction (just moves to the next instruction)
Improving our chances: **nop sleds**

**nop** is a single-byte instruction (just moves to the next instruction)
Improving our chances: **nop** sleds

**nop** is a single-byte instruction (just moves to the next instruction)

Jumping *anywhere* here will work

0xbff

0xbdf nop nop nop ...

0x0f \x3c \x2f ...

buffer
Improving our chances: \textcolor{red}{nop} sleds

\textcolor{red}{nop} is a single-byte instruction (just moves to the next instruction)

Jumping \textit{anywhere} here will work

\texttt{\%eip}  \texttt{\%ebp}  \texttt{buffer}

\begin{verbatim}
... 00 00 00 00 \%ebp 0xbdf \texttt{nop} \texttt{nop} \texttt{nop} ...
\textbackslash x0f \textbackslash x3c \textbackslash x2f ...
\end{verbatim}
Improving our chances: **nop sleds**

**nop** is a single-byte instruction (just moves to the next instruction)

Now we improve our chances of guessing by a factor of **#nops**

Jumping *anywhere* here will work
Putting it all together
Putting it all together

```
buffer

%eip

Padding

Text ...

%eip &arg1 ...
```
Putting it all together

But it has to be *something*; we have to start writing wherever the input to `gets/etc.` begins.
Putting it all together

But it has to be *something*; we have to start writing wherever the input to `gets/etc.` begins.

But it has to be *something*; we have to start writing wherever the input to `gets/etc.` begins.
Putting it all together

But it has to be something; we have to start writing wherever the input to `gets/etc.` begins.

But it has to be something; we have to start writing wherever the input to `gets/etc.` begins.
Putting it all together

But it has to be something; we have to start writing wherever the input to `gets/etc.` begins.

%eip

buffer

Text ... padding good guess

0xbdf nop nop nop ...

\x0f \x3c \x2f ...

nop sled malicious code
But it has to be *something*; we have to start writing wherever the input to `gets/etc.` begins.

```
Text ... padding good guess %eip

buffer

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

code

```
nop sled malicious code
```
Putting it all together

But it has to be *something*; we have to start writing wherever the input to `gets/etc.` begins.

![Diagram showing buffer, padding, good guess, and malicious code](image)

- **Text**: ...
- **padding**: 0xbdf nop nop nop ...
- **buffer**: \x0f \x3c \x2f ...
- **%eip**: nob sled malicious code
Project one has been posted

Due 2 weeks from today (11:59pm Thursday Feb 18)
Defenses
Recall our challenges

How can we make these even more difficult?

- Putting code into the memory (no zeroes)

- Getting %eip to point to our code (dist buff to stored eip)

- Finding the return address (guess the raw addr)
Detecting overflows with **canaries**

%eip

buffer

```
Text ... 00 00 00 00 %ebp %eip &arg1 ...
```
Detecting overflows with **canaries**

Text ... 00 00 00 00 %ebp %eip &arg1 ...

buffer
Detecting overflows with **canaries**

![Diagram of a buffer with a canary](image)

- **%eip**
- **Text**
- **00 00 00 00**
- **02 8d e2 10**
- **%ebp**
- **%eip**
- **&arg1**
- **...**

**buffer**

**canary**
Detecting overflows with canaries
Detecting overflows with canaries

%eip

Text ... buffer canary

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

Detecting overflows with **canaries**

Not the expected value: abort
Detecting overflows with canaries

Not the expected value: abort

What value should the canary have?
Canary values

From StackGuard [Wagle & Cowan]

1. Terminator canaries (CR, LF, NULL, -1)
   - Leverages the fact that scanf etc. don’t allow these

2. Random canaries
   - Write a new random value @ each process start
   - Save the real value somewhere in memory
   - Must write-protect the stored value

3. Random XOR canaries
   - Same as random canaries
   - But store canary XOR some control info, instead
Recall our challenges

How can we make these even more difficult?

• Putting code into the memory (no zeroes)
  • Option: Make this detectable with canaries

• Getting `%eip` to point to our code (dist buff to stored `eip`)

• Finding the return address (guess the raw addr)

More next time…