CLASSIC MEMORY ATKS & DEFS

CMSC 414
JAN 30 2018
TODAY'S RESOURCES

TODAY'S RESOURCES

Smashing The Stack For Fun And Profit

Aleph One

aleph@anycap.org

"Smashing the stack" [1] programming is. On many implementations it is possible to corrupt the execution stack by writing past the end of an array declared into a variable. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address. This can produce some of the most malicious data-dependent bugs known to mankind. Variants include trash the stack, scramble the stack, or pile the stack, the term range the stack is not used, as this is never done intentionally. See more: we also have bugs, timing, or core, memory leaks, procedure leakage, over read, screw.

Introduction

Over the last few months there has been a large increase in buffer overflow vulnerabilities being both discovered and exploited. Examples of these are y2k, sbove, smush,뽐, Linux/FreeBSD mount. Xe, library, etc. This paper attempts to explain what buffer overflows are, and how they exploit Unix. Basic knowledge of assembly is required. An understanding of virtual memory concepts and experience with ifconfig are very helpful but not necessary. We also assume we are working with an Intel x86 CPU, and that the operating system is Linux. Assume basic definitions before we begin. A buffer is simply a contiguous block of computer memory that holds multiple instances of the same data type. C programmers normally associate with the word buffers arrays. Most commonly, character arrays. Arrays, like all variables in C, can be declared either static or dynamic. Static variables are allocated at build time on the data segment. Dynamic variables are allocated at runtime on the stack segment. To overflow is to flow, or fill over the top, brim, or bounds. We will concern ourselves only with the overflow of dynamic buffers, otherwise known as stack bound buffer overflows.

Process Memory Organization

To understand what stack buffers are we must first understand how a process is organized in memory. Processes are divided into three regions: Text, Data, and Stack. We will concentrate on the stack region, but first a small overview of the other regions is in order. The text region is fixed by the program and includes code (instructions) and read-only data. This region contains the text section of the executable file. This region

GDB: The GNU Project Debugger

GDB Documentation

Printed Manuals

The GNU Press has printed versions of most manuals, including Debugging with GDB available.

Online Documentation

GDB User Guide

Version 3.1

The documentation is contributed by the Free Software Foundation.

GDB Project

GNU Debugger

You can get a copy of the GNU C Library Reference Manual from the Free Software Foundation.

Getting Help

You can get help using:

Learning your Program

GDB: The GNU Project Debugger

Debugging the GNU C Library

Debugging the GNU C Library

Automatic Display

When GDB is used to debug a program, it can display the contents of memory, register contents, and other information.

Breakpoints and Watchpoints

Breakpoints allow you to pause execution of a program at specific locations. Watchpoints allow you to watch for changes to specific memory locations.

Execution Control

You can use the following commands to control the execution of a program:

Commands

The following commands are available in GDB:

- set breakpoints
- set watchpoints
- set execution control
- set display

The GNU C Library Reference Manual

You can get a copy of the GNU C Library Reference Manual from the Free Software Foundation.

You can get a copy of the GNU C Library Reference Manual from the Free Software Foundation.
• 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 ARE STORED IN MEMORY

4G

0
ALL PROGRAMS ARE STORED IN MEMORY
ALL PROGRAMS ARE STORED IN MEMORY

The process’s view of memory is that it owns all of it
ALL PROGRAMS ARE 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.
THE INSTRUCTIONS THEMSELVES ARE STORED IN MEMORY
THE INSTRUCTIONS THEMSELVES ARE STORED IN MEMORY

... 0x4c2 sub $0x224,%esp
0x4c1 push %ecx
0x4bf mov %esp,%ebp
0x4be push %ebp
...
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED

4G

0xffffffff

Text

0

0x0000000000

4G

0xffffffff

Text

0

0x0000000000
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED

4G

0

0xffffffff

0x00000000

Init'd data

Text

static const int y=10;
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED

- Uninit’d data: static int x;
- Init’d data: static const int y=10;
- Text
DATA’S LOCATION DEPENDS ON HOW IT’S CREATED

Known at compile time

4G

0

Uninit’d data
Init’d data
Text

static int x;
static const int y=10;

0xffffffff
0x00000000
Data’s location depends on how it’s created.

Known at compile time:

- Text
- Init’d data
- Uninit’d data
- cmdline & env

```
static int x;
static const int y=10;
```

```
0xffffffff
0x00000000
```
DATA’S LOCATION DEPENDS ON HOW IT’S CREATED

Set when process starts

Known at compile time

static int x;
static const int y=10;
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED

Set when process starts

Known at compile time

4G

cmdline & env

Stack

Uninit'd data

Init'd data

Text

0xffffffff

0x00000000

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

static int x;

static const int y=10;

0x00000000
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED

Set when process starts

Known at compile time

0xffffffff

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

malloc(sizeof(long));

static int x;

static const int y=10;

0x0000000000
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED

- **Set when process starts**
- **Dynamically sized at runtime**
- **Known at compile time**

```
0xffffffff
```

```
int f() {
    int x;
    ...
    malloc(sizeof(long));
    static int x;
    static const int y=10;
```

```
0x00000000
```
DATA’S LOCATION DEPENDS ON HOW IT’S CREATED

Set when process starts

Dynamically sized at runtime

Known at compile time

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;

0x00000000
WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions
WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

0x0000000000

0xffffffff

Heap  Stack
WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime
WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

```
  push 1
  push 2
  push 3
```
WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

Stack and heap grow in opposite directions

Stack pointer

push 1
push 2
push 3
WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

Stack and heap diagram:

- Stack pointer
- Stack (starts at 0x00000000)
- Heap (ends at 0xffffffff)
WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

Stack pointer

push 1
push 2
push 3
WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

0x000000000 0xffffffff

Heap 1 Stack

Stack pointer

push 1
push 2
push 3
WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

0x000000000000 0xffffffff

Heap                Stack

2 1

Stack pointer

push 1
push 2
push 3
WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

Stack pointer

push 1
push 2
push 3
WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

```
push 1
push 2
push 3
```
WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime
WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

```
push 1
push 2
push 3
return
```
WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

apportioned by the OS; managed in-process by malloc

Stack pointer

push 1
push 2
push 3
return
WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

0x00000000

Heap

3 2 1

Stack

apportioned by the OS; managed in-process by malloc

Stack pointer

push 1
push 2
push 3
return

Focusing on the stack for now
STACK LAYOUT WHEN CALLING FUNCTION

```c
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int loc2;
    int loc3;
    ...
}
```
STACK LAYOUT WHEN CALLING FUNCTION

```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 FUNCTION

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

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

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

Arguments pushed in reverse order of code
STACK LAYOUT WHEN CALLING FUNCTION

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 FUNCTION

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

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

0x000000000000  0xffffffff
STACK LAYOUT WHEN CALLING FUNCTION

```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 FUNCTION

```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 FUNCTION

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

Arguments pushed in reverse order of code

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

Caller’s data
STACK LAYOUT WHEN CALLING FUNCTION

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

Two 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

0x0000000000 - 0xffffffff
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int loc2;
    int loc3;
    ...
}

0x0000000000 0xffffffff

caller’s data
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int loc2;
    int loc3;
    ...
}

STACK FRAMES
```c
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int loc2;
    int loc3;
    ...
}
```
The part of the stack corresponding to this particular invocation of this particular function
void main() { countUp(3); }

void countUp(int n) {
    if(n > 1) {
        countUp(n-1);
        printf("%d\n", n);
    }
}
```c
void main() { countUp(3); }

void countUp(int n) {
    if(n > 1)
        countUp(n-1);
    printf("%d\n", n);
}
```
void main() { countUp(3); }

void countUp(int n) {
    if(n > 1)
        countUp(n-1);
    printf("%d\n", n);
}

Stack pointer
```c
void main() { countUp(3); }

void countUp(int n) {
    if(n > 1)
        countUp(n-1);
    printf("%d\n", n);
}
```

Stack frames:

- `countUp(2)`
- `countUp(3)`
- `main()`
void main() {
    countUp(3);
}

void countUp(int n) {
    if (n > 1) {
        countUp(n - 1);
    }
    printf("%d\n", n);
}
void main() { countUp(3); }

void countUp(int n) {
    if(n > 1) {
        countUp(n-1);
        printf("%d\n", n);
    }
}
void main() { countUp(3); }

void countUp(int n) {
    if(n > 1)
        countUp(n-1);
    printf("%d\n", n);
}
void main() { countUp(3); }

void countUp(int n) {
    if(n > 1)
        countUp(n-1);
    printf("%d\n", n);
}
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++;
}

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

Q: Where is (this) loc2?

0x0000000000 0xffffffff 0xffffffff

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

0xbfffff323
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?

- I don’t know where loc2 is,
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
**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

Variable args?

caller’s data

arg1 | arg2 | arg3

- ???
- ???
- loc1
- loc2
- ???
- ...
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
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
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 “???”

%ebp  A memory address

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

( %ebp )  The value at memory address %ebp (like dereferencing a pointer)
0xbffff03b8  %ebp  A memory address

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

0xbfff03b8  \%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)
%ebp A memory address

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
0xbfff03b8  %ebp  A memory address

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

pushl %ebp
%ebp

A memory address

(0xbfff0720)

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

pushl %ebp
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

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**

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

0xbfff03b8-0xbfff0200  %ebp  A memory address
0xbfff0720 (%ebp)  The value at memory address %ebp (like dereferencing a pointer)

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

```assembly
pushl %ebp
movl %esp %ebp /* %ebp = %esp */
movl (%ebp) %ebp /* %ebp = (%ebp) */
```
%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) */
RETURNING FROM FUNCTIONS

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

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

Stack frame for this call to `func`
int main() {
    ...
    func("Hey", 10, -3);
    ...
}
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

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

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

Stack frame for this call to func

0x0000000000  %esp

0xffffffff  %ebp

 caller's data

???  arg1  arg2  arg3

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

1. Push %ebp before locals
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**

```c
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

```cpp
int main()
{
    ...
    func("Hey", 10, -3);
    ...
}
```
int main()
{
    ...
    func("Hey", 10, -3);
    ...
    Q: How do we resume here?
}
INSTRUCTIONS THEMSELVES ARE IN MEMORY

```
0xffffffff
```

```
0x00000000
```

```
4G
```

```
0xffffffff
```

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

4G

0x000000000000

0xffffffff

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

```
0xffffffff
0x00000000
0x49b movl $0x804..,(%esp)
0x493 movl $0xa,0x4(%esp)
0x4a2 call <func>
0x4a7 mov $0x0,%eax
...
```

%eip

---

4G

0x0

Text

---

0x000000000000
INSTRUCTIONS THEMSELVES ARE IN MEMORY

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

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
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

Text

4G
int main()
{
    ...
    func(“Hey”, 10, -3);
    ...
    Q: How do we resume here?
}

Stack frame for this call to \texttt{func}


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

Q: How do we resume here?

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
RETURNING FROM FUNCTIONS

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

Set %eip to 4(%ebp) at return
Push next %eip before call
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

```
leave:  mov %esp %ebp
pop %ebp
ret:    pop %eip
```
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

```
leave:  mov %esp %ebp
        pop %ebp
ret:    pop %eip
```

Current stack frame

Caller’s stack frame

-aged frame pointer
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

```
leave:   mov %esp %ebp
         pop %ebp
ret:     pop %eip
```
RETURNING FROM A FUNCTION

In C

```c
return;
```

In compiled assembly

```
leave:  mov %esp %ebp
pop %ebp
ret:    pop %eip
```

Current stack frame

- `%ebp`
- `%eip`
- `arg1`

Caller’s stack frame

- `%esp`
- `%eip`

Old frame pointer

Caller’s code

- `text`
- `loc2`
- `loc1`
- `%ebp`
- `%eip`

Stack frame variables:

- `%ebp` (Old frame pointer)
- `%esp`
- `%eip`
RETURNING FROM A FUNCTION

In C

```c
return;
```

In compiled assembly

```
leave: mov %esp %ebp
pop %ebp
ret: pop %eip
```

Current stack frame

```
text ... loc2  loc1  %ebp  %eip  arg1
```

Caller’s stack frame

```
%esp
%ebp
%eip
%eip
```
RETURNING FROM A FUNCTION

In C

```c
return;
```

In compiled assembly

```assembly
leave:  mov %esp %ebp  
       pop %ebp  
ret:    pop %eip  
```

Current stack frame

```
[  text  ...  loc2  loc1  %ebp  %eip  arg1   ]

%eip
%ebp %esp
```

Caller’s stack frame
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

```
leave:    mov %esp %ebp
          pop %ebp
ret:      pop %eip
```

Current stack frame

```
text ... loc2  loc1  %ebp  %eip  arg1
```

Caller’s stack frame

```
%eip
%esp
%ebp
```
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

```
leave:  mov %esp %ebp
        pop %ebp
ret:    pop %eip
```

Current stack frame

| text | loc2 | loc1 | ebp | eip | arg1 |

Caller’s stack frame

| esp | ebp |

%eip

%esp

%ebp
RETURNING FROM A FUNCTION

In C

```c
return;
```

In compiled assembly

```assembly
leave:  mov %esp %ebp
        pop %ebp
ret:   pop %eip
```

Current stack frame

Caller’s stack frame

The next instruction is to “remove” the arguments off the stack
RETURNING FROM A FUNCTION

In C

```c
return;
```

In compiled assembly

```
leave:  mov %esp %ebp
        pop %ebp
ret:    pop %eip
```

Current stack frame

![Current stack frame diagram](image1)

Caller’s stack frame

![Caller's stack frame diagram](image2)

The next instruction is to “remove” the arguments off the stack

And now we’re back where we started
STACK & FUNCTIONS: SUMMARY
STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

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: e.g., %eip + 2
3. **Jump to the function’s address**
STACK & FUNCTIONS: SUMMARY

Calling function (before calling):
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: e.g., %eip + 2
3. **Jump to the function’s address**

Called function (when called):
4. **Push the old frame pointer** onto the stack: push %ebp
5. **Set frame pointer** %ebp to where the end of the stack is right now: %ebp=%esp
6. **Push local variables** onto the stack; access them as offsets from %ebp
**STACK & FUNCTIONS: SUMMARY**

**Calling function (before calling):**
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: e.g., %eip + 2
3. **Jump to the function’s address**

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

**Called function (when returning):**
7. **Reset the previous stack frame**: %esp = %ebp; pop %ebp
8. **Jump back to return address**: pop %eip
STACK & FUNCTIONS: SUMMARY

**Calling function (before calling):**

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: e.g., %eip + 2
3. **Jump to the function’s address**

**Called function (when called):**

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

**Called function (when returning):**

7. **Reset the previous stack frame**: %esp = %ebp; pop %ebp
8. **Jump back to return address**: pop %eip

**Calling function (after return):**

9. **Remove the arguments** off of the stack: %esp = %esp + number of bytes of args
BUFFER OVERFLOW ATTACKS
BUFFER OVERFLOWS: HIGH LEVEL

• **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…
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);
    ...
}
```
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>Auth</th>
<th>%ebp</th>
<th>%eip</th>
<th>&amp;arg1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

buffer
**A BUFFER OVERFLOW EXAMPLE**

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

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

Here is a buffer overflow example. The function `func` takes a character string `arg1` as an argument and stores it in a buffer of size 4. If `arg1` is longer than 4 characters, it will cause a buffer overflow, potentially leading to a security vulnerability.

In the `main` function, `mystr` is initialized with the string "AuthMe!", and then `func` is called with `mystr` as the argument. This may overwrite the value of `%ebp` (the base pointer) and `%eip` (the Instruction Pointer) if the buffer is not properly sized to avoid overflow.

Here is a representation of the buffer and the memory addresses involved:

```
buffer
```

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

In this context, `Auth` is the string "AuthMe!", representing the content of the buffer after the `strcpy` operation.
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

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

    M e ! \0

    Auth 4d 65 21 00 %eip &arg1

    SEGFAULT (0x00216551)

buffer
void func(char *argv1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, argv1);
    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);
    ...
}
```

%eip &arg1
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);
    ...
}
```

```
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);
    ... 
}
```

The image shows an example of a buffer overflow. The function `func` receives a pointer to a string and copies it into a buffer without proper bounds checking. If `authenticated` is true, the program hangs indefinitely. The main function calls `func` with a string that overwrites the `eip` and `ebp` registers, causing a buffer overflow. The registers and variables are shown in the image with their values.

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

- **buffer**: The buffer where the string is copied.
- **authenticated**: The variable that determines if the program hangs.
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);
    ...
}
```

```
Auth 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);
    ...
}
```

M e ! \0

Auth 4d 65 21 00 %ebp %eip &arg1

buffer authenticated
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);
    ...
}

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

buffer
What's the worst that can 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 CAN 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’)

All ours!
WHAT'S THE WORST THAT CAN 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`)

What could you write to memory to wreak havoc?
#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;
}
#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?

&arg1 < &arg2 < &arg3?  &arg1 > &arg2 > &arg3?
#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 & FUNCTIONS: SUMMARY

Calling function (before calling):

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: e.g., %eip + 2
3. **Jump to the function’s address**

Called function (when called):

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

Called function (when returning):

7. **Reset the previous stack frame**: %esp = %ebp; pop %ebp
8. **Jump back to return address**: pop %eip
STACK & FUNCTIONS: SUMMARY

Calling function (before calling):
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: e.g., %eip + 2
3. Jump to the function’s address

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

Called function (when returning):
7. Reset the previous stack frame: %esp = %ebp; pop %ebp
8. Jump back to return address: pop %eip
STACK & FUNCTIONS: SUMMARY

Calling function (before calling):
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: e.g., %eip + 2
3. Jump to the function’s address

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

Called function (when returning):
7. Reset the previous stack frame: %esp = %ebp; pop %ebp
8. Jump back to return address: pop %eip
STACK & FUNCTIONS: SUMMARY

Calling function (before calling):
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: e.g., %eip + 2
3. Jump to the function’s address

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

Called function (when returning):
7. **Reset the previous stack frame**: %esp = %ebp; pop %ebp
8. **Jump back to return address**: pop %eip
STACK & FUNCTIONS: SUMMARY

Calling function (before calling):
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: e.g., %eip + 2
3. Jump to the function’s address

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

Called function (when returning):
7. Reset the previous stack frame: %esp = %ebp; pop %ebp
8. Jump back to return address: pop %eip
STACK & FUNCTIONS: SUMMARY

Calling function (before calling):
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: e.g., %eip + 2
3. Jump to the function’s address

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

Called function (when returning):
7. Reset the previous stack frame: %esp = %ebp; pop %ebp
8. Jump back to return address: pop %eip

%eip

%ebp

caller’s data

0x0

%eip+

arg1

arg2

code
STACK & FUNCTIONS: SUMMARY

Calling function (before calling):
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: e.g., %eip + 2
3. Jump to the function’s address

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

Called function (when returning):
7. Reset the previous stack frame: %esp = %ebp; pop %ebp
8. Jump back to return address: pop %eip
STACK & FUNCTIONS: SUMMARY

Calling function (before calling):
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: e.g., %eip + 2
3. **Jump to the function’s address**

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

Called function (when returning):
7. **Reset the previous stack frame**: %esp = %ebp; pop %ebp
8. **Jump back to return address**: pop %eip
STACK & FUNCTIONS: SUMMARY

Calling function (before calling):
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: e.g., %eip + 2
3. Jump to the function’s address

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

Called function (when returning):
7. Reset the previous stack frame: %esp = %ebp; pop %ebp
8. Jump back to return address: pop %eip
STACK & FUNCTIONS: SUMMARY

Calling function (before calling):
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: e.g., %eip + 2
3. Jump to the function’s address

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

Called function (when returning):
7. Reset the previous stack frame: %esp = %ebp; pop %ebp
8. Jump back to return address: pop %eip
STACK & FUNCTIONS: SUMMARY

Calling function (before calling):
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: e.g., `%eip + 2`
3. **Jump to the function’s address**

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

Called function (when returning):
7. **Reset the previous stack frame**: `%esp = %ebp; pop %ebp`
8. **Jump back to return address**: pop `%eip`
STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

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: e.g., %eip + 2
3. **Jump to the function’s address**

Called function (when called):

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

Called function (when returning):

7. **Reset the previous stack frame**: %esp = %ebp; pop %ebp
8. **Jump back to return address**: pop %eip
STACK & FUNCTIONS: SUMMARY

Calling function (before calling):
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: e.g., `%eip + 2`
3. **Jump to the function’s address**

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

Called function (when returning):
7. **Reset the previous stack frame**: `%esp = %ebp; pop %ebp`
8. **Jump back to return address**: pop `%eip`
GDB: YOUR NEW BEST FRIEND

**run <input>**
Run the program with input as the command-line arguments

**print <var>**
(Or just “p <var>”)
Print the value of variable var
(Can also do some operations: p &x)

**b <function>**
Set a breakpoint at function

**s**
**c**
Step through execution (into calls)
Continue execution (no more stepping)
GDB: YOUR NEW BEST FRIEND

- info frame
  (or just “i f”)
  Show info about the current frame
  (prev. frame, locals/args, %ebp/%eip)

- info reg
  (or just “i r”)
  Show info about registers
  (%ebp, %eip, %esp, etc.)

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

char loc1[4];

caller’s data
BUFFER OVERFLOW

```c
char loc1[4];

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

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

Input writes from low to high addresses
buffer overflow

char loc1[4];

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

Input writes from low to high addresses
BUFFER OVERFLOW

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

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

Input writes from low to high addresses

```
getc(loc1);
strcpy(loc1, <user input>);
memcpy(loc1, <user input>);
```
BUFFER OVERFLOW

Can over-write other data ("AuthMe!")
Can over-write the program's control flow (%eip)

```c
char loc1[4];

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

Input writes from low to high addresses
CODE INJECTION
void func(char *arg1)
{
    char buffer[4];
    sprintf(buffer, arg1);
    ...
}

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

buffer
(1) Load our own code into memory
(1) Load our own code into memory
(2) Somehow get %eip to point to it
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
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
• 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
• 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
#include <stdio.h>
int main( ) {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}

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

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

"\x31\xc0"
"\x50"
"\x68="/bin"
"\x68="/sh"
"\x89\xe3"
"\x50"
...
#include <stdio.h>
int main( ) {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
• 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

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

buffer

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?
**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 & FUNCTIONS: SUMMARY

Calling function (before calling):
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: e.g., %eip + 2
3. Jump to the function’s address

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

Called function (when returning):
7. Reset the previous stack frame: %esp = %ebp; pop %ebp
8. Jump back to return address: pop %eip

Calling function (after return):
9. Remove the arguments off of the stack: %esp = %esp + number of bytes of args
HIJACKING THE SAVED %EIP
HIJACKING THE SAVED %EIP

%eip
%ebp

buffer

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

0xbff
HIJACKING THE SAVED %EIP

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

buffer
HIJACKING THE SAVED %EIP

But how do we know the address?
HIJACKING THE SAVED %EIP

What if we are wrong?

%eip
%ebp

buffer

0xbff

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

HIJACKING THE SAVED %EIP

What if we are wrong?
HIJACKING THE SAVED %EIP

What if we are wrong?

buffer

0xbff

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

```
buffer |
---

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

%eip    %ebp
```

0xbfff
nop is a single-byte instruction
(just moves to the next instruction)
nop is a single-byte instruction
(just moves to the next instruction)

Jumping anywhere here will work

Now we improve our chances of guessing by a factor of \#nops
BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER

![Diagram of buffer overflow]

- `%eip` pointing to the buffer
- Buffer contents: `text ... 00 00 00 00 %ebp %eip &arg1 ...`
BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER

![Diagram of buffer overflow]

- `text`
- ... padding ...
- `%eip`
- `&arg1`
- ...
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.
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.
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.