TODAY'S PAPERS

Title: Smashing The Stack For Fun And Profit

Author: Aleph One

Volume Seven, Issue Forty-Nine
File 14 of 16

bmeqg.org, r0b, and underground.org
bring you

Smashing The Stack For Fun And Profit

by Aleph One
aleph@underground.org

"smash the stack" (C programming) n. On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. One 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 insidious data-dependent bugs known to mankind. Variables include trash the stack, scramble the stack, mangle the stack: the term using the attack is not used, as this is never done intentionally. See spam; see also alias bug, handshaker on core, memory leak, precedence leackage, overtrust screw.

Introduction

Over the last few months there has been a large increase of buffer overflow vulnerabilities being both discovered and exploited. Examples of those are sylow, splinter, sundial, 8.7.3, linux/firewall ports, xt library, etc. This paper attempts to explain what buffer overflows are, and how they exploit work.

Basic knowledge of assembly is required. An understanding of virtual memory concepts and experience with x86 is very helpful but not necessary. We also assume we are working with an Intel x86 CPU, and that the operating system is Linux.

Some 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 buffer arrays. Most commonly, character arrays. Arrays, like all variables in C, can be declared either static or dynamic. Static variables are allocated at load time on the data segment. Dynamic variables are allocated at run time on the stack. So overflow is to flow, or fill over the top, craps, or bosses. We will indicate ourselves only with the overflow of dynamic buffers, otherwise known as stack-based buffer overflows.

Process Memory Organisation

Abstract

This paper presents a systematic solution to the persistent problem of buffer overflow attacks. Buffer overflow attacks gained notoriety in 1988 as part of the Morris Worm incident on the Internet [23]. Despite the fact that fixing individual buffer overflow vulnerabilities is fully simple, buffer overflow attacks continue to thrive, as reported in the SANS Network Security Digest.

Buffer overflow problems appear to be the most common software vulnerabilities reported in May, with degradations-of-service attacks a distant second. Many of the buffer overflow problems are probably the result of careless programming, and could have been found and corrected by the vendors before releasing the software, if the vendors had performed elementary testing or code reviews along the way [42].

The basic problem is that, while individual buffer overflow vulnerabilities are simple to patch, the vulnerabilities are pervasive. Thousands of lines of legacy code are still running as privileged daemons (BADCode) that contain numerous software errors. New programs are being developed with care, but are still often developed using unsafe languages such as C, whose simple errors can leave serious vulnerabilities.

The continued success of these attacks is also due to the "pushy" nature by which we protect against such attacks. The cycle of a buffer overflow attack is simple: A real victim runs without the vulnerability at slightly priv-
• 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
ALL PROGRAMS ARE STORED IN MEMORY

4G 0xffffffff
0 0x000000000
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

4G

0xffffffff

Text

0x0000000000

0xffffffff
THE INSTRUCTIONS THEMSELVES ARE STORED IN MEMORY

4G

0xffffffff

0x4bf  mov  %esp,%ebp
0x4b

0xffffffff

0x00000000

...  

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

static const int y = 10;

Init’d data

Text

0

0xffffffff

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

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

Uninit'd data

Init'd data

Text

static int x;

static const int y=10;

Known at compile time

0xffffffff

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

Known at compile time

cmdline & env

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

0xffffffff
0x00000000

Set when process starts

Known at compile time

4G

cmdline & env

Uninit'd data

Init'd data

Text

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

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:
- cmdline & env
- Stack
- Heap
- Uninit’d data
- Init’d data
- Text

Known at compile time:
- int f() {
  int x;
  ...
}
- malloc(sizeof(long));
- static int x;
- static const int y=10;

0xffffffff

0x00000000
Data's location depends on how it's created.

Set when process starts

Runtime

Known at compile time

4G

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
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

0x00000000

Heap

Stack

0xffffffff

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
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

Heap

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

0x00000000  0xffffffff

Heap      3  2  1  Stack

Stack pointer

code
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 3 2 1 Stack

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
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

Heap

apportioned by the OS; managed in-process by malloc

Stack

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

Heap

Stack

0x00000000 0xffffffff

apportioned by the OS; managed in-process by malloc

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;
    ...
}
```

caller’s data
STACK LAYOUT WHEN CALLING FUNCTION

```
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;
    ...
}
```

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

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;
    ...
}
```

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;
    loc2++;
}

0xffffffff  0x00000000

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

0xffffffff
0x00000000

caller’s data
arg3
arg2
arg1
???
???
loc2
loc1
loc2

Variable args?

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++;
}

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?

Stack frame for this call to func

- 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 “???”s
%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)
%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)
**NOTATION**

0xbfff03b8 \( \%ebp \) A memory address

0xbfff0720 \( (\%ebp) \) The value at memory address \( \%ebp \) (like dereferencing a pointer)
0xbffff03b8 \%ebp  A memory address

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

\textbf{pushl} \%ebp
%ebp  A memory address

(%%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)

\[ \text{pushl \%ebp} \]
%ebp A memory address

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

```
pushl %ebp
```
%ebp  A memory address

(%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
movl %esp %ebp  /* %ebp = %esp */
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 */
```
%ebp  A memory address

(%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

<table>
<thead>
<tr>
<th>Address</th>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbfff03b8</td>
<td>%ebp</td>
<td>A memory address</td>
</tr>
<tr>
<td>0xbfff0720</td>
<td>(%ebp)</td>
<td>The value at memory address %ebp (like dereferencing a pointer)</td>
</tr>
</tbody>
</table>

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

%ebp

Stack frame for this call to func

caller’s data
int main()
{
    ...
    func("Hey", 10, -3);
    ...
}

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

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

Stack frame for this call to func

%ebp

%ebp
RETURNING FROM FUNCTIONS

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

Stack frame for this call to func

%ebp

caller’s data
arg3
arg2
arg1
???
int main()
{
    ...
    func("Hey", 10, -3);
    ...
    Q: How do we restore %ebp?
RETURNING FROM FUNCTIONS

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

1. Push %ebp before locals
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

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

Stack frame for this call to func
int main()
{
    ...
    func("Hey", 10, -3);
    ...
    Q: How do we resume here?
}
INSTRUCTIONS THEMSELVES ARE IN MEMORY

```
4G

0xffffffff

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

Text

```
0x00000000
```
INSTRUCTIONS THEMSELVES ARE IN MEMORY

```
... 0x4a7 mov $0x0,%eax 0x4a2 call <func>
0x49b movl $0x804..,(%esp) 0x493 movl $0xa,0x4(%esp)
0x400 eip
```

- 4G
- 0x00000000
- 0xffffffff
INSTRUCTIONS THEMSELVES ARE IN MEMORY

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

%eip
INSTRUCTIONS THEMSELVES ARE IN MEMORY

![Diagram showing memory layout with instructions in the text segment.]

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

 `%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
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)
...
```
int main()
{
    ...
    func("Hey", 10, -3);
    ...
    \textbf{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**

- `%ebp` for this call to `func`
- Caller’s data

**Diagram:**
- Loc2
- Loc1
- `%ebp`
- `???`
- `arg1`
- `arg2`
- `arg3`
- `0xffffffff`
- `0x00000000`

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

Push next %eip before call
RETURNING FROM FUNCTIONS

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

Stack frame for this call to `func`

Set `%eip` to `4(%ebp)` at return

Push next `%eip` before call
STACK & 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 & 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
STACK & 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) /* copy it off first */
8. Jump back to return address: %eip = 4(%ebp) /* use the copy */
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);
    ...
}
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

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

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

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

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

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
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>Auth</td>
<td>%ebp</td>
<td>%eip</td>
<td>&amp;arg1</td>
</tr>
<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);
    ...
}
```

```
AuthMe! \0
```

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

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

buffer
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

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

```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

declare a function

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

declare the main function

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

%ebp %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);
    ...
}
```

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);
    ...

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

Code still runs; user now ‘authenticated’

```
M e ! \0
```

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

buffer 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.
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 can happen?

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

```plaintext
00 00 00 00 %ebp %eip &myst
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);
    ...
}
```

All ours!

`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’)

What could you write to memory to wreak havoc?

All ours!
#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;
}
#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

%eip

0x0

%ebp

0x0

caller’s data

code
STACK & 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 & 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 & 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 & 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**
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 & 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 & 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 & 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 & 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 & 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 & 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 & 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 of stack and function calls with labels for `%eip`, `%ebp`, and stack variables]
STACK & 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 & 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 & 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>
step through execution (into calls)
BUFFER OVERFLOW

```c
char loc1[4];
```

```
code  loc2  loc1  %ebp  %eip+...  arg1  arg2  caller's data
```
BUFFER OVERFLOW

char loc1[4];

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

etc.
BUFFER OVERFLOW

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

Input writes from low to high addresses

data

arg2

arg1

%ebp

loc1

loc2

%eip+…

code

etc.
BUFFER OVERFLOW

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

Input writes from low to high addresses
Can over-write other data ("AuthMe!")

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

can over-write other data ("AuthMe!")
**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

```
gets(loc1);
strcpy(loc1, <user input>);
mempcpy(loc1, <user input>);
```
CODE INJECTION
void func(char *arg1)
{
    char buffer[4];
    sprintf(buffer, arg1);
    ...
}

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

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

(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
(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
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
• Goal: **privilege escalation**
  • Ideally, they go from guest (or non-user) to root

• There are many out there
  - And competitions to see who can write the smallest
#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
movl %esp,%eax
...```

SHELLCODE
#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
...
```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"
...
```c
#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""/sh"
"\x68""/bin"
"\x89\xe3"
"\x50"
...
```
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?
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:
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 %ebb

Returning function:
7. Reset the previous stack frame: %ebp = (%ebp)
8. Jump back to return address: %eip = 4(%ebp)
HIJACKING THE SAVED %EIP

%eip

%ebp

text ... 00 00 00 00 %ebp %eip &arg1 ... \x0f \x3c \x2f ...
HIJACKING THE SAVED %EIP
HIJACKING THE SAVED %EIP

buffer

0x0f \x3c \x2f ...
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?

%eip
%ebp

buffer

0xbff

0xbdf

%ebp 0xbdf &arg1 ... \x0f \x3c \x2f ...
HIJACKING THE SAVED %EIP

What if we are wrong?

buffer

0xbf

\x0f \x3c \x2f ...

0xbdf

&arg1

%ebp

%eip

0x00 0x00 0x00 0x00

0xbff
HIJACKING THE SAVED %EIP

What if we are wrong?

This is most likely data, so the CPU will panic (Invalid Instruction)
CHALLENGE 3: FINDING THE RETURN ADDRESS
• If we don’t have access to the code, we don’t know how far the buffer is from the saved %ebp
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 \(2^{32}\) (or \(2^{64}\)) 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
IMPROVING OUR CHANCES: NOP SLEDS

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

Jumping anywhere here will work
IMPROVING OUR CHANCES: NOP SLEDS

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

%eip

```
buffer
```

```
text ... 00 00 00 00 %ebp %eip &arg1 ...
```
BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER

Diagram showing a buffer with text, padding, %eip, &arg1, and their positions.
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.

```
buffer
padding
0xbdf
nop
nop
good
guess
%eip
\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.

```
buffer

padding

good guess

%eip
```

```

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

nop sled malicious code
```
BUFFER OVERFLOW
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 address)
DETECTING OVERFLOWS WITH CANARIES

%eip

buffer
DETECTING OVERFLOWS WITH CANARIES

%eip

buffer
DETECTING OVERFLOWS WITH CANARIES

%eip

```
text ... 00 00 00 00 02 8d e2 10  %ebp  %eip  &arg1  ...
```

buffer  canary
DETECTING OVERFLOWS WITH CANARIES

%eip

buffer  canary

\text{\ldots} 0xbdf \text{\ldots} 0\times0f 0\times3c 0\times2f \ldots
DETECTING OVERFLOWS WITH CANARIES

%eip

text ...

buffer canalry

0xbdf nop nop nop ...

\x0f \x3c \x2f ...

DETECTING OVERFLOWS WITH CANARIES

Not the expected value: abort

%eip

buffer       canary

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

Not the expected value: abort
DETECTING OVERFLOWS WITH CANARIES

Not the expected value: abort

What value should the canary have?
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 address)
RETURN TO LIBC

%eip

padding

good guess

libc

buffer

nop sled malicious code

text...

0xbdf nop nop nop ...

\x0f \x3c \x2f ...

 malicious code
RETURN TO LIBC

%eip  padding  good guess

libc  text  ...  buffer  0xbdf  nop  nop  nop  ...

nop sled
RETURN TO LIBC

%eip

padding
good
guess

0xbdf
&argv1

...  

buffer

text

libc
RETURN TO LIBC

%eip

padding

text

%eip

&arg1

...
RETURN TO LIBC

%eip

padding

text
libc
buffer

%eip  &arg1

libc

exec()  printf()  "/bin/sh"

libc
RETURN TO LIBC

%eip

padding

known location

0x17f &arg1 ...

... text ...

libc

buffer

... exec() printf() ... 

libc

"/bin/sh" ...
RETURN TO LIBC

%eip

padding

known location

libc

buffer

0x17f 0x20d 1 ...

... exec() printf() ... "/bin/sh" ...

libc
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)
  Non-executable stack doesn’t work so well

• Finding the return address (guess the raw address)
Address Space Layout Randomization

- **Text**: Randomize where exactly these regions start

---

Set when process starts

- **Runtime**:
  - **known at compile time**
  - **4G**
    - **cmdline & env**
    - **Stack**
    - **Heap**
    - **Uninit'd data**
    - **Init'd data**
    - **Text**

---

- **0xf0000000**
- **0x00000000**

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

malloc(sizeof(long));

static int x;

static const int y=10;
```
ADDRESS SPACE LAYOUT RANDOMIZATION

Shortcomings of ASLR

- Introduces return-to-libc atk
- Probes for location of usleep
- On 32-bit architectures, only 16 bits of entropy
- fork() keeps same offsets
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)
  Non-executable stack doesn’t work so well

- Finding the return address (guess the raw address)
  Address Space Layout Randomization (ASLR)

Best defense: Good programming practices
BUFFER OVERFLOW PREVALENCE

Significant percent of all vulnerabilities

Data from the National Vulnerability Database
void safe()
{
    char buf[80];
    fgets(buf, 80, stdin);
}

void safer()
{
    char buf[80];
    fgets(buf, sizeof(buf), stdin);
}
void safe()
{
    char buf[80];
    fgets(buf, 80, stdin);
}

void safer()
{
    char buf[80];
    fgets(buf, sizeof(buf), stdin);
}

void vulnerable()
{
    char buf[80];
    if (fgets(buf, sizeof(buf), stdin) == NULL)
        return;
    printf(buf);
}
```c
void safe()
{
    char buf[80];
    fgets(buf, 80, stdin);
}

void safer()
{
    char buf[80];
    fgets(buf, sizeof(buf), stdin);
}

void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```
FORMAT STRING VULNERABILITIES
int i = 10;
printf("%d %p\n", i, &i);
int i = 10;
printf("%d %p\n", i, &i);
int i = 10;
printf("%d %p\n", i, &i);
int i = 10;
printf("%d %p\n", i, &i);
```c
int i = 10;
printf("%d %p\n", i, &i);
```
PRINTF FORMAT STRINGS

```c
int i = 10;
printf("%d %p\n", i, &i);
```

- printf takes variable number of arguments
- printf pays no mind to where the stack frame "ends"
- It presumes that you called it with (at least) as many arguments as specified in the format string
PRINTF FORMAT STRINGS

int i = 10;
printf("%d %p\n", i, &i);

printf’s stack frame
• printf takes variable number of arguments
• printf pays no mind to where the stack frame “ends”
• It presumes that you called it with (at least) as many arguments as specified in the format string
```c
#include <stdio.h>

int main() {
    int i = 10;
    printf("%d \%p\n", i, &i);
    return 0;
}
```

###PRINTF FORMAT STRINGS

- printf takes variable number of arguments
- printf pays no mind to where the stack frame "ends"
- It presumes that you called it with (at least) as many arguments as specified in the format string
PRINTF FORMAT STRINGS

```c
int i = 10;
printf("%d %p\n", i, &i);
```

- printf takes variable number of arguments
- printf pays no mind to where the stack frame "ends"
- It presumes that you called it with (at least) as many arguments as specified in the format string
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}

"%d %x"
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}

"%d %x"

caller's stack frame

0x000000000000 0xffffffff

... %ebp %eip &fmt
caller's stack frame
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin) == NULL)
        return;
    printf(buf);
}

"%d %x"

caller’s stack frame
```c
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

```
"%d %x"
```

```
caller's stack frame
```

```
0x00000000 0xffffffff
```

```
... %ebp %eip &fmt
```

```
caller's stack frame
```
FORMAT STRING VULNERABILITIES
• `printf("100% dml");`
FORMAT STRING VULNERABILITIES

- `printf("100% dml");`
  - Prints stack entry 4 bytes above saved `%eip`
FORMAT STRING VULNERABILITIES

- `printf("100% dml");`
  - Prints stack entry 4 bytes above saved %eip

- `printf("%s");`
FORMAT STRING VULNERABILITIES

- `printf("100% dml");`
  - Prints stack entry 4 bytes above saved %eip

- `printf("%s");`
  - Prints bytes *pointed to* by that stack entry
• `printf("100% dml");`
  • Prints stack entry 4 bytes above saved %eip

• `printf("%s");`
  • Prints bytes *pointed to* by that stack entry

• `printf("%d %d %d %d ...");`
• `printf("100% dml");`
  • Prints stack entry 4 bytes above saved %eip

• `printf("%s");`
  • Prints bytes pointed to by that stack entry

• `printf("%d %d %d %d ...");`
  • Prints a series of stack entries as integers
FORMAT STRING VULNERABILITIES

- `printf("100% dml");`
  - Prints stack entry 4 bytes above saved %eip

- `printf("%s");`
  - Prints bytes *pointed to* by that stack entry

- `printf("%d %d %d %d ...");`
  - Prints a series of stack entries as integers

- `printf("%08x %08x %08x %08x ...");`
FORMAT STRING VULNERABILITIES

• `printf("100% dml");`
  • Prints stack entry 4 bytes above saved `%eip`

• `printf("%s");`
  • Prints bytes pointed to by that stack entry

• `printf("%d %d %d %d ...")`;
  • Prints a series of stack entries as integers

• `printf("%08x %08x %08x %08x ...");`
  • Same, but nicely formatted hex
FORMAT STRING VULNERABILITIES

- `printf("100% dml");`
  - Prints stack entry 4 bytes above saved %eip

- `printf("%s");`
  - Prints bytes pointed to by that stack entry

- `printf("%d %d %d %d ...");`
  - Prints a series of stack entries as integers

- `printf("%08x %08x %08x %08x ...");`
  - Same, but nicely formatted hex

- `printf("100% no way!");`
FORMAT STRING VULNERABILITIES

- `printf("100% dml");`
  - Prints stack entry 4 bytes above saved `%eip`

- `printf("%s");`
  - Prints bytes *pointed to* by that stack entry

- `printf("%d %d %d %d ...");`
  - Prints a series of stack entries as integers

- `printf("%08x %08x %08x %08x ...");`
  - Same, but nicely formatted hex

- `printf("100% no way!");`
  - **WRITES** the number 3 to address pointed to by stack entry
% of vulnerabilities that involve format string bugs

#define BUF_SIZE 16
char buf[BUF_SIZE];
void vulnerable()
{
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if(len > BUF_SIZE) {
        printf("Too large\n");
        return;
    }
    memcpy(buf, p, len);
}
WHAT'S WRONG WITH THIS CODE?

```c
#define BUF_SIZE 16
char buf[BUF_SIZE];
void vulnerable()
{
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if(len > BUF_SIZE) {
        printf("Too large\n");
        return;
    } 
    memcpy(buf, p, len);
}

void *memcpy(void *dest, const void *src, size_t n);
```
WHAT'S WRONG WITH THIS CODE?

```c
#define BUF_SIZE 16
char buf[BUF_SIZE];
void vulnerable()
{
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if(len > BUF_SIZE) {
        printf("Too large\n");
        return;
    }
    memcpy(buf, p, len);
}
```

void *memcpy(void *dest, const void *src, size_t n);
typedef unsigned int size_t;

WHAT'S WRONG WITH THIS CODE?

```c
#define BUF_SIZE 16
char buf[BUF_SIZE];
void vulnerable()
{
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if(len > BUF_SIZE) {
        printf("Too large\n");
        return;
    }
    memcpy(buf, p, len);
}
```

void *memcpy(void *dest, const void *src, size_t n);
typedef unsigned int size_t;

**Negative**
WHAT'S WRONG WITH THIS CODE?

#define BUF_SIZE 16
char buf[BUF_SIZE];
void vulnerable()
{
    int len = read_int_from_network();
    char *p = read_string_from_network();

    if(len > BUF_SIZE) {
        printf("Too large\n");
        return;
    }

    memcpy(buf, p, len);
}

void *memcpy(void *dest, const void *src, size_t n);
typedef unsigned int size_t;
#define BUF_SIZE 16
char buf[BUF_SIZE];
void vulnerable()
{
  Negative
  int len = read_int_from_network();
  char *p = read_string_from_network();
  Ok if(len > BUF_SIZE) {
    printf(“Too large\n”);
    return;
  }
  memcpy(buf, p, len);
}

void *memcpy(void *dest, const void *src, size_t n);
typedef unsigned int size_t;
INTEGER OVERFLOW
VULNERABILITIES
void vulnerable()
{
    size_t len;
    char *buf;

    len = read_int_from_network();
    buf = malloc(len + 5);
    read(fd, buf, len);
    ...
}
WHAT'S WRONG WITH THIS CODE?

```c
void vulnerable()
{
    size_t len;
    char *buf;

    len = read_int_from_network();
    buf = malloc(len + 5);
    read(fd, buf, len);
    ...
}
```

HUGE
void vulnerable()
{
    size_t len;
    char *buf;
    
    len = read_int_from_network();
    buf = malloc(len + 5);   // **HUGE**
    read(fd, buf, len);
    ...
}
void vulnerable()
{
    size_t len;
    char *buf;

    /* HUGE */
    len = read_int_from_network();
    buf = malloc(len + 5);
    read(fd, buf, len);
    ...
}

Takeaway: You have to know the semantics of your programming language to avoid these errors
INTEGER OVERFLOW PREVALENCE

% of vulnerabilities that involve integer overflows