Last time

We continued our 1st section: Software Security

By launching Buffer overflows and other memory safety vulnerabilities

- Buffer overflow fundamentals
This time

We will finish up

Buffer overflows

and other memory safety vulnerabilities

By looking at

Overflow Defenses

• Finish overflow attacks & other vulnerabilities

• Overflow defenses

• Everything you’ve always wanted to know about `gdb` but were too afraid to ask
First, a recap (args)

```c
#include <stdio.h>

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

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

```c
#include <stdio.h>

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

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

What will happen?

&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;
}
```c
#include <stdio.h>

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

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

What will happen?

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

- `%ebp` caller's data
- `%eip` code
Stack and functions: Summary

**Calling function:**

1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: %eip+something
3. **Jump to the function’s address**
Stack and functions: Summary

**Calling function:**

1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: %eip+something
3. **Jump to the function’s address**
Stack and functions: Summary

**Calling function:**

1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: `%eip+something`
3. **Jump to the function’s address**
Stack and functions: Summary

**Calling function:**

1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: %eip+something
3. **Jump to the function’s address**
Stack and functions: Summary

**Calling function:**

1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: `%eip+something`
3. **Jump to the function’s address**
Stack and functions: Summary

**Calling function:**
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: %eip+something
3. **Jump to the function’s address**
Stack and functions: Summary

**Calling function:**
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: %eip+something
3. **Jump to the function’s address**

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

**Calling function:**
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: %eip+something
3. **Jump to the function’s address**

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

**Calling function:**
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: `%eip+something`
3. **Jump to the function’s address**

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

**Calling function:**
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: `%eip+something`
3. **Jump to the function’s address**

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

**Calling function:**
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: %eip+something
3. **Jump to the function’s address**

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

![Diagram](image-url)
Stack and functions: Summary

**Calling function:**
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: `%eip+something`
3. **Jump to the function’s address**

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

**Calling function:**
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: %eip+something
3. **Jump to the function’s address**

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

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

**Calling function:**
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: `%eip+something`
3. **Jump to the function’s address**

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

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

**Calling function:**
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: %eip+something
3. **Jump to the function’s address**

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

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

**if**

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

**ir**

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

**x/<n> <addr>**

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

**b <function>**

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

char loc1[4];
Buffer overflow

call loc1[4];
gets(loc1);
strcpy(loc1, <user input>);
memcpy(loc1, <user input>);
etc.
Buffer overflow

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

Input writes from low to high addresses
Buffer overflow

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

Input writes from low to high addresses
Buffer overflow

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

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

Input writes from low to high addresses
Buffer overflow

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

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

Input writes from low to high addresses

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

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

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

buffer
High-level idea

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

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

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

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

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

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

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

(1) Load my own code into memory

(2) Somehow get `%eip` to point to it
This is nontrivial

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

- Think about what is tricky about the attack
  - The key to defending it will be to make the hard parts really hard
Challenge 1

**Loading code into memory**

- It must be the machine code instructions (i.e., already compiled and ready to run)

- We have to be careful in how we construct it:
  - It can’t contain any all-zero bytes
    - Otherwise, sprintf / gets / scanf / … will stop copying
    - How could you write assembly to never contain a full zero byte?
  - It can’t make use of the loader (we’re injecting)
  - It can’t use the stack (we’re going to smash it)
What kind of code would we want to run?

• **Goal:** full-purpose shell
  • The code to launch a shell is called “shell code”
  • It is nontrivial to write 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
Shellcode

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

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

Assembly

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

Shellcode

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

Assembly

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

Shellcode

#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"
...
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 %eax
pushl $0x68732f2f
pushl $0x6e69622f
movl %esp,%ebx
pushl %eax
...```

**Machine code**

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

(Part of) your input
Privilege escalation

• Permissions later, but for now…

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

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

- Userid = the user who ran the process
- Effective userid = what is used to determine what access the process has

- Consider passwd: root owns it, but users can run it
  - getuid() will return you (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?
Effective userid

- Userid = the user who ran the process
- Effective userid = what is used to determine what access the process has

- Consider passwd: root owns it, but users can run it
  - `getuid()` will return you (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
Challenge 2

Getting our injected code to run

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

Thoughts?
Challenge 2

Getting our injected code to run

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

Thoughts?
Challenge 2

Getting our injected code to run

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

Thoughts?
Challenge 2

Getting our injected code to run

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

Thoughts?
Stack and functions: Summary

Calling function:
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: %eip+something
3. **Jump to the function’s address**

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

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

![Diagram showing memory regions and registers](image-url)
Hijacking the saved `%eip`
Hijacking the saved `%eip`

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

What if we are wrong?
Hijacking the saved %eip

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

What if we are wrong?
Hijacking the saved %eip

What if we are wrong?

This is most likely data, so the CPU will panic (Invalid Instruction)
Challenge 3

Finding the return address
Challenge 3

Finding the return address

• If we don’t have access to the code, we don’t know how far the buffer is from the saved \%ebp
Challenge 3

Finding the return address

• If we don’t have access to the code, we don’t know how far the buffer is from the saved %ebp

• One approach: just try a lot of different values!
Challenge 3

Finding the return address

- If we don’t have access to the code, we don’t know how far the buffer is from the saved `%ebp`

- One approach: just try a lot of different values!

- Worst case scenario: it’s a 32 (or 64) bit memory space, which means $2^{32}$ ($2^{64}$) possible answers
Challenge 3

Finding the return address

• If we don’t have access to the code, we don’t know how far the buffer is from the saved %ebp

• One approach: just try a lot of different values!

• Worst case scenario: it’s a 32 (or 64) bit memory space, which means $2^{32}$ ($2^{64}$) possible answers

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

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

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

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

buffer
Improving our chances: **nop sleds**

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

Now we improve our chances of guessing by a factor of **#nops**
Putting it all together
Putting it all together

%eip

buffer

padding

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

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

![Diagram of buffer with text and padding]

- `%eip`
- `padding`
- `buffer`
Putting it all together

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

Buffer

%eip

padding

good guess

Text ... 0xbdf &arg1 ...

...
Putting it all together

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

%eip  

```
0 0 0 0
buffer
```

...  

`good guess`

```
...nop...  
```

`sled`

```
0xbdf
```

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

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

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

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

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

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

```
nop sled malicious code
```
Project one will be posted by tomorrow morning

Due 2 weeks later (11:59pm Wednesday Feb 18)
Defenses
Recall our challenges

How can we make these even more difficult?

- Putting code into the memory (no zeroes)

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

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

```
%eip

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

buffer
Detecting overflows with canaries

%eip

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

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

Not the expected value: abort
Detecting overflows with **canaries**

Not the expected value: abort

What value should the canary have?
Canary values

From StackGuard [Wagle & Cowan]

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

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

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

**How can we make these even more difficult?**

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

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

- Finding the return address (guess the raw addr)
Return-to-libc

%eip

buffer

padding

good
guess

Text

0xbdf

nop
nop

good

guess

malicious code

nop

sled
Return-to-libc

%eip

Text

padding

good
guess

0xbdf

nop nop nop ...

buffer

nop sled

libc
Return-to-libc

%eip

padding

good guess

Text

... 0xbdf &arg1 ...

libc

buffer
Return-to-libc

%eip

padding

Text

libc

%eip &arg1 ...

buffer
Return-to-libc

%eip

padding

Text ... %eip &arg1 ...

libc

buffer

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

libc
Return-to-libc

%eip

padding

known location

Text ... 0x17f &arg1 ...

libc

buffer

exec() printf() ... "/bin/sh" ...
Return-to-libc

%eip

padding

known location

Text ... 0x17f 0x20d 1 ...

libc

buffer

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 addr)
Address Space Layout Randomization (ASLR)

• Basic idea: change the layout of the stack

• Slow to adopt
  • Linux in 2005
  • Vista in 2007 (off by default for compatibility with older software)
  • OS X in 2007 (for system libraries), 2011 for all apps
  • iOS 4.3 (2011)
  • Android 4.0
  • FreeBSD: no

How would you overcome this as an attacker?
Overflow defenses summary

• 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 addr)
  • Address Space Layout Randomization

• Good coding practices
Next time

Continuing with Software Security

Writing & testing for Secure Code

Required reading:
“StackGuard: Simple Stack Smash Protection for GCC”

Optional reading:
“Basic Integer Overflows”
“Exploiting Format String Vulnerabilities”