CSAW 2023: ”double zer0 dilemma” Write Up

Will Rosenberg

September 2023

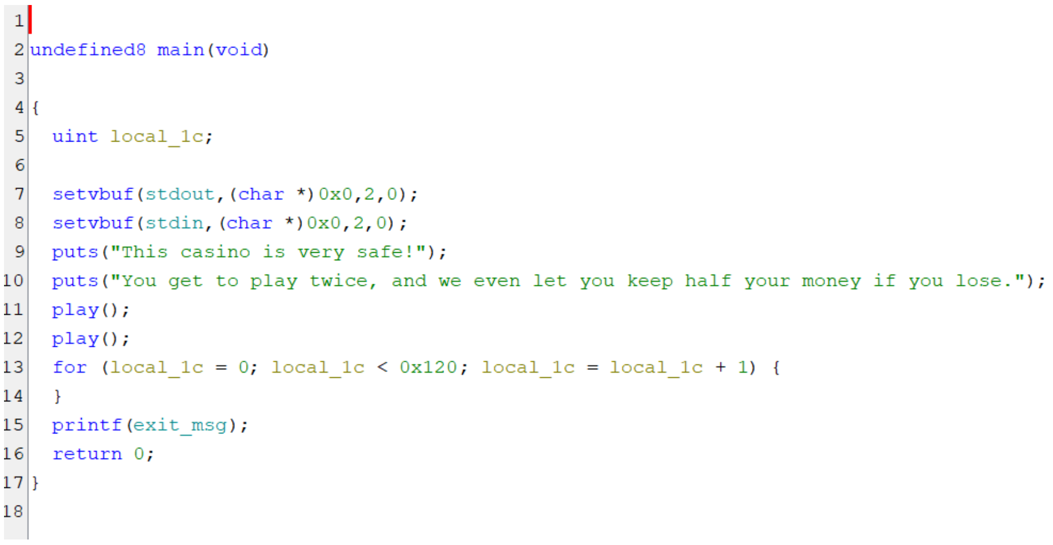
1 Introduction

This is a write up for the ”double zer0 dilemma” challenge from CSAW ’23. I hope to encapsulate my thought process throughout the challenge, as well as, the problems I encountered when creating my exploit, and how I solved these issues. This write-up will assume the reader has little to no PWN knowledge but some systems knowledge.

2 Identifying a Vulnerability

The first step of any PWN is REVERSE ENGINEERING! Given the binary file that comes with the challenge, I opened Ghidra and analyzed the binary.

I followed the logic of the program, starting at main(), and quickly found a vulnerability in the play() function, seen below.



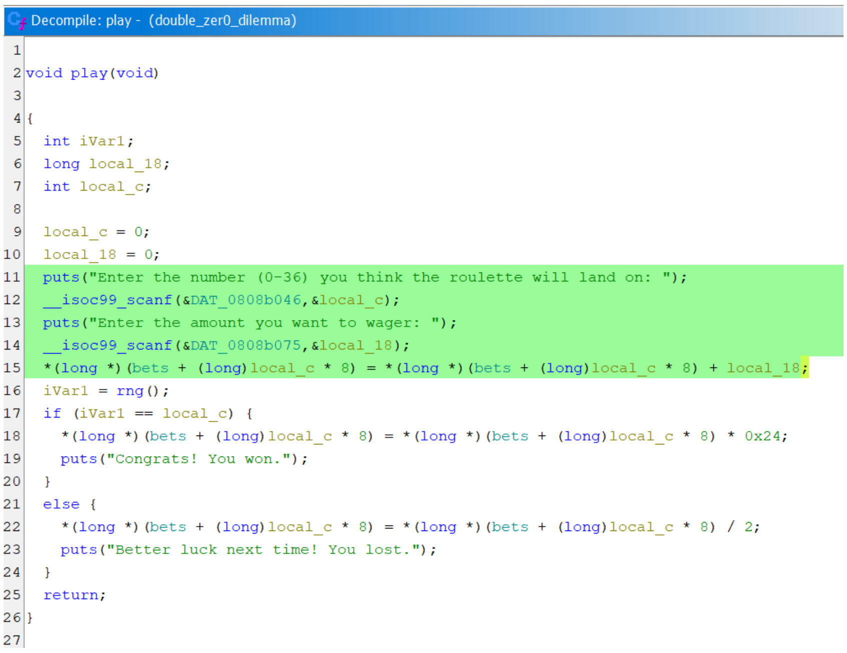
```

undefined8 main(void)

{
  uint local_1c;

  setvbuf(stdout,(char *)0x0,2,0);
  setvbuf(stdin,(char *)0x0,2,0);
  puts("This casino is very safe!");
  puts("You get to play twice, and we even let you keep half your money if you lose.");
  play();
  play();
  for (local_1c = 0; local_1c < 0x120; local_1c = local_1c + 1) {
  }
  printf(exit_msg);
  return 0;
}

```



```

void play(void)

{
  int iVar1;
  long local_18;
  int local_c;

  local_c = 0;
  local_18 = 0;
  puts("Enter the number (0-36) you think the roulette will land on: ");
  __isoc99_scanf(&DAT_0808b046,&local_c);
  puts("Enter the amount you want to wager: ");
  __isoc99_scanf(&DAT_0808b075,&local_18);
  *(long *)(bets + (long)local_c * 8) = *(long *)(bets + (long)local_c * 8) + local_18;
  iVar1 = rng();
  if (iVar1 == local_c) {
    *(long *)(bets + (long)local_c * 8) = *(long *)(bets + (long)local_c * 8) * 0x24;
    puts("Congrats! You won.");
  }
  else {
    *(long *)(bets + (long)local_c * 8) = *(long *)(bets + (long)local_c * 8) / 2;
    puts("Better luck next time! You lost.");
  }
  return;
}

```

The function takes in two values: index and wager. The index is used to index into "bets", which is an array of long integers. Wager is used to set the value of this long integer determined by index. However, the index is never bounds checked, so I can set it to any value (as long as it can be represented by 4 bytes as indicated by the "%d" in scanf). The index is also sign extended to 8 bytes, so negative indices are allowed. As well, the wager value is never checked, so I can set it to any 8 bytes value (the scanf for wager used "%ld"). Therefore, I came to the conclusion that play() effectively allows me to set *almost* any 8 byte word in the program to any value I want.

It should be noted that setting these values is not as simple as just putting an offset and value as inputs. As seen in the play() function, the index is multiplied by 8, meaning the inputted value represents the word offset from "bets." As well, the value is not only added to the value already in the word, but it is also divided by 2 (the if will never be True because we will input indices outside the range of [0, 36]). I will come back to this discussion as it relates to calculating the final inputs.

3 Picking an Exploitation

Now that a vulnerability has been identified, I must find a proper exploit to achieve my goal. Since the flag is never read into memory and this PWN challenge does not give a "win" function, I must take over the control flow of the program and achieve shellcode execution.

The typical methods for this include: executing shellcode written on the stack, return oriented programming, or redirecting libc global offset table (GOT) entries.

To determine which methods are viable, I must check the different protections on this program. Running:

```
readelf -a double_zer0_dilemma | grep GNU_STACK -A 1
```

I found that the stack did not have execution permissions, so the first exploit option is not viable.

The exploit allows me to set any two addresses, so the canary protection will not cause any issues either way. Therefore, the next protection to check is ASLR (address randomization) and PIE (position independent executable). Both of these are off (ASLR is turned off in the DockerFile provided), so I can know the addresses of the .data and .text sections, as well as know the dynamically loaded libc addresses. This would in theory allow me to use ROP or GOT exploits, but given that I only have two words to write to, ROP does not make sense. Furthermore, the stack addresses are likely out of range of my 4 byte index.

We have then come to the conclusion that we must overwrite GOT values to redirect control flow.

4 Executing the Exploit

In theory, I can set the GOT entry to any pointer to code, allowing me to run direct assembly shellcode or piece together gadgets. However, I decided to overwrite the GOT entry with the system() libc call because it makes for a simple exploit. Since system requires a string, I must control this string and set it to something that prints the flag. To do this, I must also edit the parameter passed to the redirected libc function. Therefore, the libc function that I redirect must take a global variable as an input.

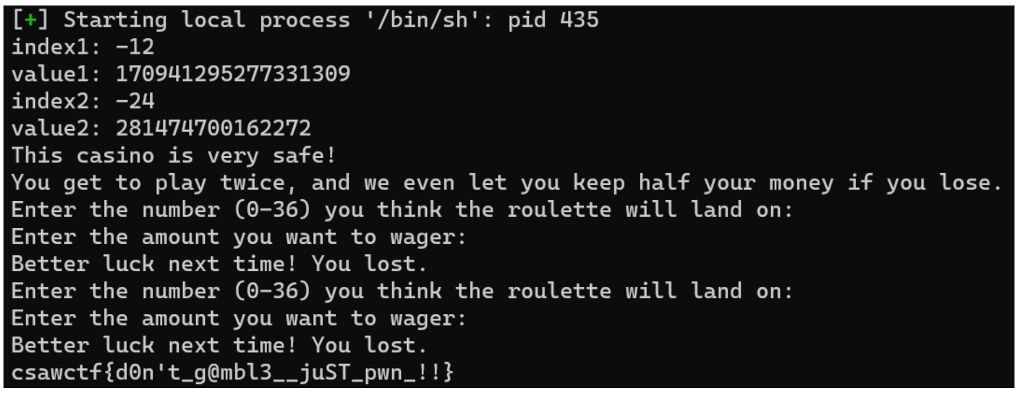
With these conditions in mind, I chose printf(exit_msg) inside of main. To perform my exploit, I need to overwrite exit_msg with some shell command (I chose "cat *ag;" because I must stay within 8 bytes) and overwrite printf's GOT with system().

I need to determine the GOT address of printf, the address of exit_msg, the address of bets, exit_msg's initial value, printf GOT's initial value, and the address of system().

The address of printf's GOT, exit_msg, and bets can all be found using objdump because we have a PIE. The initial value of exit_msg is "Your tot", as given via decompilation or disassembly. printf GOT's initial value would be dependent on the loaded address of printf, but we are redirecting the first and only call of printf in the program. This means the value will be equal to printf@PLT, which can also be found via disassembly. To find the address of system(), I must run the program and see where it is loaded. Fortunately ASLR is off, so the value will always be the same for each instance of the program. Using GDB, I found the value. All the values can be seen in the "attack.py" file provided along with the write up.

After writing my script and entering these values, my exploit worked on my computer but not on the server. After a little bit of testing, I remembered that libc may not be loaded in the same spot on different systems (even if ASLR is off on both). Therefore, I needed to know the address of system() on the server. There is likely a better way to find this value, but I used the control of the exit_msg to create a format string vulnerability in printf. I then analyzed the stack locally using GDB to find the stack offset of a libc pointer (__libc_start_main + some_offset). I then ran the format string vulnerability using the input: %15p to print this pointer's address. I then calculated the offset between system() and the local __libc_start_main address and added this offset to the leaked address, giving the true address of system() on the server.

I then ran my exploit, passing ”cat *ag;” to system, which printed the flag file!



```
[+] Starting local process '/bin/sh': pid 435
index1: -12
value1: 170941295277331309
index2: -24
value2: 281474700162272
This casino is very safe!
You get to play twice, and we even let you keep half your money if you lose.
Enter the number (0-36) you think the roulette will land on:
Enter the amount you want to wager:
Better luck next time! You lost.
Enter the number (0-36) you think the roulette will land on:
Enter the amount you want to wager:
Better luck next time! You lost.
csawctf{d0n't_g@mbl3__juST_pwn_!!}
```

5 Notes on Other Methods

After looking at some other exploit scripts, I think its important to quickly mention a few other methods/ideas. I saw people replace the srand() or time() libc calls with a pointer back to play(). This allowed them to have more than two addresses to overwrite, and they avoided the value being divided by two.