tank! bonus

by mowteam / B34R5HELL

Tags: pwn

Rating:

See the link for solution scripts and a LaTeX formatted write-up

Square CTF 2023: "tank" and "tank! bonus"

Introduction

This write-up will go through the solution for "tank!", which then gets built upon by introducing libc ROP to solve "tank! bonus."

The "tank!" challenge gives a binary with a description discussing const variables, which becomes useful later.

Identifying the Vulnerability

I ran the program and a few preliminary shell commands (nm, strings, objdump) to get an idea of the program, and then ran it through Ghidra. After parsing the decompiled code for a while, I understood the framework of the program: the player inputs a power (effectively velocity) and an angle (in degrees), both limited by a max power and max angle const stack variable, respectively. These values are then used to calculate the landing spot of the projectile, using classic kinematics. The game checks if the projectile hit an enemy ("E" elements in the game board), and if so, the hit counter increments, otherwise the miss counter increments, causing the player to lose one of five lives. The symbol of the ammo also gets placed in the game board where the projectile landed. If the user gets three hits in a row, they get to choose special ammo or regular ammo for their next shot.

This is where the vulnerability comes into play. After doing some calculations, I found that the shots could land anywhere from index 0 to 111. Since the game board array is of length 112, everything is within bounds. However, the special ammo shot not only checks the position it landed but also the positions to the left and right, as well as placing an ammo symbol at these locations. Therefore, we can overflow the buffer by one byte on either side. Looking at what is on either side of the array, we conveniently find that the max angle variable is the four bytes to the left of the game buffer and the max power is the four bytes to the right of the game buffer. We can now change these max values allowing for potentially even greater overflows of the game buffer!

Planning the Attack

Now that we have identified a vulnerability, we need to figure out how to actually exploit it.

The first step is to figure out what happens when we overflow into the max power and angle variables. Since the special ammo shot has symbol "-" (0x2d or 45), we will replace the LSB of max power and the MSB of max angle with 0x2d. This will make the max power 45 and the max angle a value larger than 360, allowing us to use the entire unit circle and go backwards! Once we change these values, we can go even further out of bounds, allowing us to change max power to 0x2d2d2d2d, if we would like. This is important because we can now edit any byte on the stack (the heap and dynamic libraries are probably still out of range).

Great! We can easily crash the program, but we still can allow makes bytes 0x2d or 0x5f ("_"). Next, we look towards the ammo symbol itself. When selecting ammo type for the special ammo, we pick a 1-based index, which is currently bounded to be within the ammo array of the length 2. However, if we look at how this bounds checking takes place, it also uses a value stored on the stack to check the upper bound of 2, which is rather convenient given that a compiler would never do this. So, the question arises, can we edit this value using our overflow? The answer is yes because we can change any byte on the stack. The value of 2 is stored at rbp - 0x1a0. If we "shoot" a byte of value 0x2d into this position, we can now

use an index up to 45. We could also shoot multiple bytes to get a larger index, but we will see later that there is no need to do so.

Now that we can pick any byte forward of the ammo array, which is positioned at rbp - 0x12, we need to find the desired bytes on the stack or put them there ourselves. For "tank!" we just need to call the win function (i_dont_want_to_finangle_a_shell_out_of_this_please_just_give_me_one()). It's address is 0x4013de. The return address of game_loop() back into main is 0x401af7, so we need to edit the lower two bytes of the return address.

Note: One could try to edit the return address of a diffferent function so that only the LSB has to be changed, but in this case, place_enemy() is the only function sharing the second LSB, so this is not possible. We must therefore find 0x13 and 0xde (or some other instruction of the win function).

This is where I got stuck and wasted a lot of time. After not finding how to place my own bytes on the stack (they had to be infront of rbp - 0x12), I thought I could use a cheap technique to grab the points for "tank!" and ignore "tank! bonus." My plan was to use the randomness of the canary to get the bytes I needed. This meant brute forcing until I found the necessary byte inside the canary. Fortunately, I found 0xe3 (a replacement for 0xde) on the stack locally, so I thought I only needed to find 0x13 in the canary on the server. However, after writing a script to search for 0x13 in the canary, I realized that 0xe3 was only on the stack locally. I believe this is because ASLR is on for the server, which means I naively and incorrectly turned ASLR off when testing locally. This meant the last two hours were a complete waste of time. I could still try to find both bytes in the canary, as it is feasible but would just take a while. However, I thought i would go to bed and approach the problem in the morning.

Once I got a fresh pair of eyes, I almost immediaitely saw how to place my own bytes on the stack. When a player makes a move, they must enter "pew!" to shoot. However, the users input is read into an array of size 8 using fgets (only 7 bytes are read as a null byte is placed at the end). This value is then compared, using strcmp, with a string literal "pew!\n".

Therefore, a user must have the first 6 bytes of the array occupied with "pew!\n\x00" in order to progress the process, but they can use the last byte available to them to store a desired byte. Note: the array is cleared the first time the user input is read each game loop, but this does not affect us because we perform the ammo selection and byte shooting within the same loop.

Creating a Payload and Exploiting

I will not go into too much detail here, as I have laid out the groundwork for how to perform such an attack and my scripts can be found in the repo.

Now that we can edit any byte on the stack with any value we want, we can really make any exploit our heart desires. We just need to make sure to give ourselves unlimited lives at some point and also give ourselves unlimited special ammo because it becomes annoying to have to constantly get three hits in a row. Also, my program began breaking because sometimes I would get accidental hits, causing the program to prompt me for special ammo when I was not expecting it. Therefore, its better to just turn it on.

Once I had overwritten the return address and lost the game to exit the game_loop() function, the program segfaulted. This is because the stack is no longer aligned and system() expects the stack to be aligned. Therefore, instead of jumping to the beginning of the win function, I jump just after the push rbp instruction. We now have a successful exploit.

Tank! bonus

This challenge was an extension of the "tank!" challenge. It was the same program with the same stack offsets and exploits with just the win function removed. Therefore, we now use libc ROP to perform an attack. This is no different than most ROP attacks. However, we need to leak a libc address to know where everything is. This can be found in my script, but the basic idea is to select an ammo type at the desired byte location and print it to the first position of the game board (this is the same technique I used to search the canary). We can then view its value and repeat this to piece together the value of a known libc address. I chose the __libc_start main address stored above main's stack frame. I then calculated the desired offsets locally as these should be the same. I used the find functionality in gdb to find the location of "/bin/sh" and 0xc35f (pop rdi) inside of libc. I also had to fix stack alignment by inserting the address to a ret instruction. Note: As with all ROP libc attacks, make sure to LD_PRELOAD the given libc shared object when running locally.

Original writeup (https://github.com/mowteam/CTFWriteUps/tree/master/SquareCTF2023/tank).

Comments