

Guilherme Alt Chagas Merklein

PicoCTF 2024 - Reverse Engineering

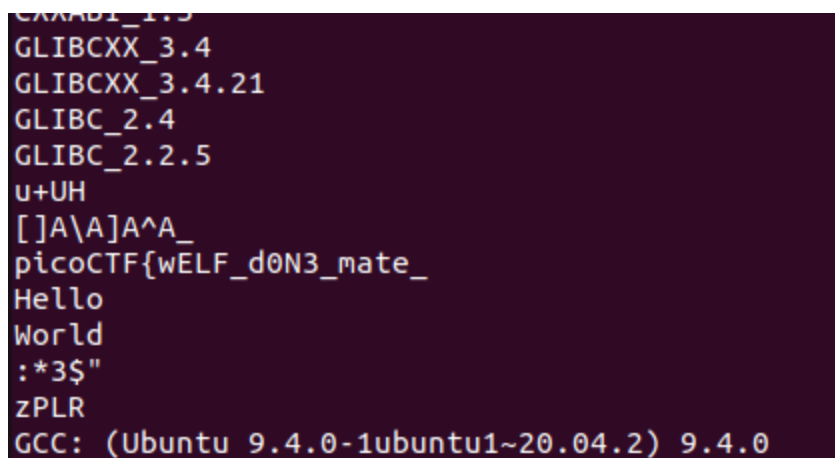
flag: picoCTF{wELF_d0N3_mate_97750d5f}

27/03/2024

FactCheck

In this challenge, you are presented with an executable file named “bin”. Since it was a 64-bit ELF executable, I had to run it on my Ubuntu VM. When running the program, nothing is printed on the shell, and the process just terminates without me having any idea of what happened.

Next thing I tried was running the “strings” command, to look for any clues, and just like that, I got the flag (not really). Image 1 shows some strings outputted by the command, and one of them looks like an incomplete flag. There was also a “Hello” and a “World”. I tried to submit the incomplete flag alone or by adding the hello world and closing the brackets, but yeah, that was clearly not it.



```
CXXABI_1.3  
GLIBCXX_3.4  
GLIBCXX_3.4.21  
GLIBC_2.4  
GLIBC_2.2.5  
u+UH  
[]A\A]A^A_  
picoCTF{wELF_d0N3_mate_  
Hello  
World  
:*3$"  
zPLR  
GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0
```

Image 1: Half-filled flag.

It was time to get serious, so I opened up Ghidra on my Windows machine, which is an open-source tool for analyzing binary files. It also served as a disassembler and decompiler, so I could look at the “source code” (not really) of a compiled binary.

Note: I switched to IDA Free on the later challenges because I personally felt it was more friendly than Ghidra, even if the free version does not come with a decompiler. I'll still use Ghidra sometimes when I want to see decompiled code.

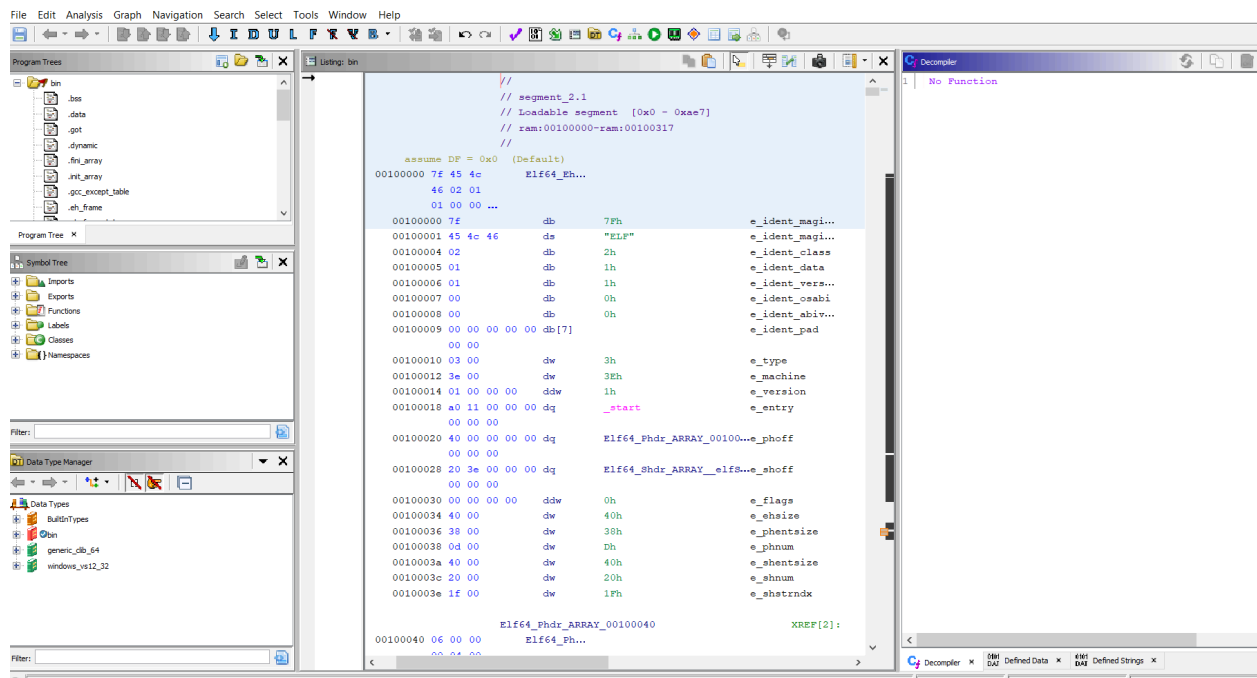


Image 2: Ghidra GUI.

I loaded the file on Ghidra and let it analyze the file. Upon finishing, I navigated to the “Symbol Tree” view and selected the main function in the functions folder, and then I focused my attention on the decompiled view (Image 3). There were some local variable declarations and a lot of calls with this `std::allocator<char>`. Honestly I had no idea what was happening at first.

```

4  undefined8 main(void)
5
6  {
7      char cVar1;
8      char *pcVar2;
9      long in_FS_OFFSET;
10     allocator<char> local_249;
11     basic_string<char,std::char_traits<char>,std::allocator<char>> local_248 [32];
12     basic_string local_228 [32];
13     basic_string<char,std::char_traits<char>,std::allocator<char>> local_208 [32];
14     basic_string local_1e8 [32];
15     basic_string local_1c8 [32];
16     basic_string local_1a8 [32];
17     basic_string local_188 [32];
18     basic_string local_168 [32];
19     basic_string<char,std::char_traits<char>,std::allocator<char>> local_148 [32];
20     basic_string local_128 [32];
21     basic_string<char,std::char_traits<char>,std::allocator<char>> local_108 [32];
22     basic_string<char,std::char_traits<char>,std::allocator<char>> local_e8 [32];
23     basic_string local_c8 [32];
24     basic_string<char,std::char_traits<char>,std::allocator<char>> local_a8 [32];
25     basic_string local_88 [32];
26     basic_string local_68 [32];
27     basic_string<char,std::char_traits<char>,std::allocator<char>> local_48 [40];
28     long local_20;
29
30     local_20 = *(long *) (in_FS_OFFSET + 0x28);
31     std::allocator<char>::allocator();
32     /* try { // try from 001012cf to 001012d3 has its CatchHandler @ 00101975 */
33     std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>> ::basic_string
34         ((char *)local_248,(allocator *)"picoCTF{wELF_d0N3_mate_" );
35     std::allocator<char>::~~allocator(&local_249);
36     std::allocator<char>::allocator();
37     /* try { // try from 0010130a to 0010130e has its CatchHandler @ 00101996 */
38     std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>> ::basic_string
39         ((char *)local_228,(allocator *)&DAT_0010201d);

```

Image 3: Decompiled view for main function (beginning).

I noticed there was an allocator using the incomplete flag that I saw using the strings command, so I clicked on it to see where it was stored in the executable file, and there were actually more characters next to it, on different memory addresses (image 4).

00102005	70 69 63	s_picoCTF{wELF_d0N3_mate__00102005	XREF[1]:	main:001012c5 (*)
	6f 43 54	ds "picoCTF{wELF_d0N3_mate_"		
	46 7b 77 ...			
		DAT_0010201d	XREF[3]:	main:00101300 (*), main:0010133b (*), main:00101427 (*)
0010201d	35	?? 35h 5		
0010201e	00	?? 00h		
		DAT_0010201f	XREF[2]:	main:00101376 (*), main:00101634 (*)
0010201f	37	?? 37h 7		
00102020	00	?? 00h		
		DAT_00102021	XREF[1]:	main:001013b1 (*)
00102021	33	?? 33h 3		
00102022	00	?? 00h		
		DAT_00102023	XREF[1]:	main:001013ec (*)
00102023	30	?? 30h 0		
00102024	00	?? 00h		
		DAT_00102025	XREF[1]:	main:00101462 (*)
00102025	61	?? 61h a		
00102026	00	?? 00h		
		DAT_00102027	XREF[1]:	main:0010149d (*)
00102027	65	?? 65h e		
00102028	00	?? 00h		
		DAT_00102029	XREF[1]:	main:001014d8 (*)
00102029	66	?? 66h f		

Image 4: incomplete flag and other values stored in memory. At the end of the list, there were also the strings “Hello” and “World”.

After seeing that, I just tried to concatenate all these characters to the incomplete flag and submit it (closing the brackets, of course), but it failed. Looking back at image 3, my next guess was that the incomplete flag was being dynamically loaded to memory, and the program would automatically fill it once it started running, so I would have to debug it and read the flag in memory.

I actually spent quite a time trying to understand the static code before opening a debugger, but then a specific line on the decompiled code caught my attention (took some time to actually find it), which was using a “}” character (image 5). I thought this might be the moment when the program finishes writing the flag to the memory, since they always end with “}”. Image 6 shows the assembly for this specific line of code.

```
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>> ::operator+=
(local_248,local_128);
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>> ::operator+=
(local_248,'}');
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string
(local_48);
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string
```

Image 5: Program line using “}” (in the middle). Ignore the highlighted line.

```

c0 fd ff ff
00101853 be 7d 00      MOV     ESI,0x7d
00 00
00101858 48 89 c7      MOV     RDI,RAX
0010185b e8 a0 f8      CALL    <EXTERNAL>::std::__cxx11::basic_string<char,st ...
ff ff
```

Image 6: Assembly view. 0x7d is the ascii value for the ‘}’ character.

Since I knew beforehand that the RSI and RDI registers are commonly used for string manipulation, my hypothesis was that this CALL instruction would call a function that would place ‘}’ in some memory location, and that memory location would probably have to be passed somehow before calling the function. So I had to check the RDI register before this call. Basically, it’s debug time.

First I calculated beforehand the address of my breakpoint. Since the CALL instruction is at 0x0010185B and my main function starts at 0x00101289, I had to add a breakpoint to the address “main + 0x5D2”. The use of relative addressing here is important since the base address of the main function will change at runtime. But the offset will remain the same.

Again, I hopped on to my linux VM and started *GDB*, a famous tool for debugging programs. I went to my directory with the bin file and typed “gdb bin”. Which took me to the debugging program. Next thing was to set a breakpoint, so I typed “break *main + 0x5D2”, which would set a breakpoint at the call instruction I mentioned before.

Then, I typed “run”, and the program ran until my breakpoint was found. At this moment, I typed “x/10i \$pc”, So I could see the 10 next instructions by checking the program counter register value (ChatGPT helped me with some commands). At last, I typed “x/s \$rdi” to see the contents of the memory address pointed by the RDI register, and it didn’t work, but then I tried to print it before the instruction “RDI ← RAX”, and it actually worked (image 7).

```
(gdb) x/10i $pc
=> 0x55555555853 <main+1482>: mov     $0x7d,%esi
0x55555555858 <main+1487>: mov     %rax,%rdi
0x5555555585b <main+1490>: call   0x55555555100 <_ZNSt7__cxx112basic_stringIcSt11cha
0x55555555860 <main+1495>: mov     $0x0,%ebx
0x55555555865 <main+1500>: lea     -0x40(%rbp),%rax
0x55555555869 <main+1504>: mov     %rax,%rdi
0x5555555586c <main+1507>: call   0x555555550f0 <_ZNSt7__cxx112basic_stringIcSt11cha
0x55555555871 <main+1512>: lea     -0x60(%rbp),%rax
0x55555555875 <main+1516>: mov     %rax,%rdi
0x55555555878 <main+1519>: call   0x555555550f0 <_ZNSt7__cxx112basic_stringIcSt11cha
(gdb) x/s $rdi
0x55555556aed0: "picoCTF{wELF_d0N3_mate_97750d5f"
(gdb) █
```

Image 7: flag.