Hurry Up - Wait!

https://play.picoctf.org/practice/challenge/165?category=3&page=1

A binary is given (`svchost.exe`) with no description. Looking at this file in my favorite hex editor quickly noticed it is not a PE file but rather an ELF binary. Let's validate this with file program.

```
file svchost
svchost: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0,
BuildID[sha1]=4c4687ba4c5b7fea4c9f13aaa29269a3ca164b09, stripped
```

Ok, looks like we have an x64 ELF binary so let's get more details about it using `readelf` program:

```
readelf -h svchost
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x1c10
  Start of program headers:          64 (bytes into file)
  Start of section headers:          16688 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         27
  Section header string table index: 26
```

We now know the entry point of the program is at offset `0x1c10` so let's also look at the section headers and see the `.text` start memory offset.

```
readelf -S svchost | grep -E -B4 -A4 "\.text"
  [12] .plt              PROGBITS         0000000000001a00  00001a00
       0000000000000200  0000000000000010  AX       0     0     16
  [13] .plt.got          PROGBITS         0000000000001c00  00001c00
       0000000000000008  0000000000000008  AX       0     0     8
  [14] .text             PROGBITS         0000000000001c10  00001c10
       0000000000000e92  0000000000000000  AX       0     0     16
  [15] .fini             PROGBITS         0000000000002aa4  00002aa4
       0000000000000009  0000000000000000  AX       0     0     4
  [16] .rodata           PROGBITS         0000000000002ab0  00002ab0
```

the entry point shown in the header indeed coincide with the start offset of the `.text` section, nothing out the ordinary but let's see if I can find anything interesting from the `strings` program output. For now, let see any strings five or more chars

```
strings -n 5 svchost
...
pu^$
=J4NZ
In 'callee_echo'
In 'send_secret_1'
In 'send_secret_2'In 'send_secret_3'0
123456789abcdefghijklmnopqrstuvwxyzCTF_{}
;*3$"
GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
...
```

I have remove from the above some output that are not of interest and at lease one line that could potentially takes us to the right path (`CTF{}`). Let's load the file in IDA Pro and see where that string is being referenced.

```
.rodata:00005615DD27BAC0  00000014  C  GNAT Version: 7.5.0
.rodata:00005615DD27BAD8  0000000A  C  _ada_main
.rodata:00005615DD27BB7F  00000005  C  pu^$
.rodata:00005615DD27BBDF  00000005  C  =J4NZ
.rodata:00005615DD27BC58  00000010  C  In 'callee_echo'
.rodata:00005615DD27BC70  00000013  C  In 'send_secret_1'
.rodata:00005615DD27BC90  00000026  C  In 'send_secret_2'In 'send_secret_3'0
.rodata:00005615DD27BCC0  00000029  C  123456789abcdefghijklmnopqrstuvwxyzCTF_{}
.eh_frame:00005615DD27BF... 00000006  C  ;*3$\"
```

In IDA, we see the section where that string is located `.rodata` so let's double click on it and find any functions that could be referencing it and load that code block.

```
.rodata:00005615DD27BCBF db    0
.rodata:00005615DD27BCC0 ; const _ptr unk_5615DD27BCC0
.rodata:00005615DD27BCC0 unk_5615DD27BCC0 db  31h ; 1          ; DATA XREF: sub_5615DD27B136+9↑o
.rodata:00005615DD27BCC1 ; const _ptr unk_5615DD27BCC1
.rodata:00005615DD27BCC1 unk_5615DD27BCC1 db  32h ; 2          ; DATA XREF: sub_5615DD27B16A+9↑o
.rodata:00005615DD27BCC2 ; const _ptr unk_5615DD27BCC2
.rodata:00005615DD27BCC2 unk_5615DD27BCC2 db  33h ; 3          ; DATA XREF: sub_5615DD27B19E+9↑o
.rodata:00005615DD27BCC3 ; const _ptr unk_5615DD27BCC3
.rodata:00005615DD27BCC3 unk_5615DD27BCC3 db  34h ; 4          ; DATA XREF: sub_5615DD27B1D2+9↑o
.rodata:00005615DD27BCC4 ; const _ptr unk_5615DD27BCC4
.rodata:00005615DD27BCC4 unk_5615DD27BCC4 db  35h ; 5          ; DATA XREF: sub_5615DD27B206+9↑o
.rodata:00005615DD27BCC5 ; const _ptr unk_5615DD27BCC5
.rodata:00005615DD27BCC5 unk_5615DD27BCC5 db  36h ; 6          ; DATA XREF: sub_5615DD27B23A+9↑o
.rodata:00005615DD27BCC6 ; const _ptr unk_5615DD27BCC6
.rodata:00005615DD27BCC6 unk_5615DD27BCC6 db  37h ; 7          ; DATA XREF: sub_5615DD27B26E+9↑o
.rodata:00005615DD27BCC7 ; const _ptr unk_5615DD27BCC7
```

The function above, indeed makes a LEA call to that memory address starting at char `1` and later a system IO call to print that char to the screen, so at some point char `1` is being displayed.

```
.text:00005615DD27B136
.text:00005615DD27B136 sub_5615DD27B136 proc near
.text:00005615DD27B136 ; __unwind { // 5615DD279000
.text:00005615DD27B136 push    rbp
.text:00005615DD27B137 mov     rbp, rsp
.text:00005615DD27B13A push    rbx
.text:00005615DD27B13B sub     rsp, 8
.text:00005615DD27B13F lea     rax, unk_5615DD27BCC0
.text:00005615DD27B146 lea     rdx, unk_5615DD27BCB8
.text:00005615DD27B14D mov     rcx, rax
.text:00005615DD27B150 mov     rbx, rdx
.text:00005615DD27B153 mov     rax, rdx
.text:00005615DD27B156 mov     rdi, rcx        ; this
.text:00005615DD27B159 mov     rsi, rax
.text:00005615DD27B15C call    _ada__text_io__put__4
.text:00005615DD27B161 nop
.text:00005615DD27B162 add     rsp, 8
.text:00005615DD27B166 pop     rbx
.text:00005615DD27B167 pop     rbp
.text:00005615DD27B168 retn
.text:00005615DD27B168 ; } // starts at 5615DD27B136
.text:00005615DD27B168 sub_5615DD27B136 endp
.text:00005615DD27B168
```

Ok, let's continue and see the block of code calling this function.

```
.text:00005615DD27B98A sub_5615DD27B98A proc near
.text:00005615DD27B98A ; __unwind { // 5615DD279000
.text:00005615DD27B98A push    rbp
.text:00005615DD27B98B mov     rbp, rsp
.text:00005615DD27B98E mov     rdi, 38D7EA4C68000h
.text:00005615DD27B998 call    _ada__calendar__delays__delay_for
.text:00005615DD27B99D call    sub_5615DD27B616
.text:00005615DD27B9A2 call    sub_5615DD27B4AA
.text:00005615DD27B9A7 call    sub_5615DD27B372
.text:00005615DD27B9AC call    sub_5615DD27B5E2
.text:00005615DD27B9B1 call    sub_5615DD27B852
.text:00005615DD27B9B6 call    sub_5615DD27B886
.text:00005615DD27B9BB call    sub_5615DD27B8BA
.text:00005615DD27B9C0 call    sub_5615DD27B922
.text:00005615DD27B9C5 call    sub_5615DD27B3A6
.text:00005615DD27B9CA call    sub_5615DD27B136
.text:00005615DD27B9CF call    sub_5615DD27B206
.text:00005615DD27B9D4 call    sub_5615DD27B30A
.text:00005615DD27B9D9 call    sub_5615DD27B206
.text:00005615DD27B9DE call    sub_5615DD27B57A
.text:00005615DD27B9E3 call    sub_5615DD27B8EE
.text:00005615DD27B9E8 call    sub_5615DD27B40E
.text:00005615DD27B9ED call    sub_5615DD27B6E6
.text:00005615DD27B9F2 call    sub_5615DD27B782
.text:00005615DD27B9F7 call    sub_5615DD27B8EE
.text:00005615DD27B9FC call    sub_5615DD27B206
.text:00005615DD27BA01 call    sub_5615DD27B372
.text:00005615DD27BA06 call    sub_5615DD27B136
.text:00005615DD27BA0B call    sub_5615DD27B3A6
.text:00005615DD27BA10 call    sub_5615DD27B136
.text:00005615DD27BA15 call    sub_5615DD27B30A
.text:00005615DD27BA1A call    sub_5615DD27B3DA
.text:00005615DD27BA1F call    sub_5615DD27B956
.text:00005615DD27BA24 nop
.text:00005615DD27BA25 pop     rbp
.text:00005615DD27BA26 retn
.text:00005615DD27BA26 ; } // starts at 5615DD27B98A
```

Hmmm, there are three references to that function. Could this mean that char 1 is printed 3 times to output? Well, let's put a break point at the function prolog and run this program in IDA with its Linux remote debugger capability and see what we get. Also, I see that before all these functions call, there is what looks like an execution time delay which is using a very long integer. I was not sure what a call to the `Ada.Calendar.Delays` function did but after a quick search, that's exactly what it does: `Delay_For (D : Duration): Delay until an interval of`

`length (at least) D seconds has passed, or the task is aborted to at least the current ATC nesting level.` . That means that we need to make sure we edit the RDI register before the function call to a small value. Let make it 0x000000000000000F

```
.text:00005615DD27B98A
.text:00005615DD27B98A
.text:00005615DD27B98A ; Attributes: bp-based frame
.text:00005615DD27B98A
.text:00005615DD27B98A sub_5615DD27B98A proc near
.text:00005615DD27B98A ; __unwind { // 5615DD279000
.text:00005615DD27B98A push    rbp
.text:00005615DD27B98B mov     rbp, rsp
.text:00005615DD27B98E mov     rdi, 38D7EA4C68000h
.text:00005615DD27B998 call    _ada__calendar__delays__delay_for
.text:00005615DD27B99D call    sub_5615DD27B616
.text:00005615DD27B9A2 call    sub_5615DD27B4AA
.text:00005615DD27B9A7 call    sub_5615DD27
.text:00005615DD27B9AC call    sub_5615DD27
.text:00005615DD27B9B1 call    sub_5615DD27
.text:00005615DD27B9B6 call    sub_5615DD27
.text:00005615DD27B9BB call    sub_5615DD27
.text:00005615DD27B9C0 call    sub_5615DD27
.text:00005615DD27B9C5 call    sub_5615DD27
.text:00005615DD27B9CA call    sub_5615DD27
.text:00005615DD27B9CF call    sub_5615DD27
.text:00005615DD27B9D4 call    sub_5615DD27
.text:00005615DD27B9D9 call    sub_5615DD27
.text:00005615DD27B9DE call    sub_5615DD27
.text:00005615DD27B9E3 call    sub_5615DD27
.text:00005615DD27B9E8 call    sub_5615DD27
.text:00005615DD27B9ED call    sub_5615DD27
.text:00005615DD27B9F2 call    sub_5615DD27
.text:00005615DD27B9F7 call    sub_5615DD27
.text:00005615DD27B9FC call    sub_5615DD27
.text:00005615DD27BA01 call    sub_5615DD27
.text:00005615DD27BA06 call    sub_5615DD27
.text:00005615DD27BA0B call    sub_5615DD27
.text:00005615DD27BA10 call    sub_5615DD27
.text:00005615DD27BA15 call    sub_5615DD27
.text:00005615DD27BA1A call    sub_5615DD27
.text:00005615DD27BA1F call    sub_5615DD27
.text:00005615DD27BA24 nop
.text:00005615DD27BA25 pop     rbp
.text:00005615DD27BA26 retn
.text:00005615DD27BA26 ; } // starts at 5615DD27B98A
```

```
; =============== S U B R O U T I N E =======================================

; Attributes: bp-based frame

sub_5615DD27B616 proc near              ; CODE XREF: sub_5615DD27B98A+13↓p
; __unwind { // 5615DD279000
                push    rbp
                mov     rbp, rsp
                push    rbx
                sub     rsp, 8
                lea     rax, unk_5615DD27BCD8
                lea     rdx, unk_5615DD27BCB8
                mov     rcx, rax
                mov     rbx, rdx
                mov     rax, rdx
                mov     rdi, rcx        ; this
                mov     rsi, rax
                call    _ada__text_io__put__4
                nop
                add     rsp, 8
                pop     rbx
                pop     rbp
                retn
; } // starts at 5615DD27B616
```

Doing that, now we get to make the first function calls if the block, what character would this print and once we reach to `sub_5615DD27B136`, would it print `1` ? Let's see...

```
.text:00005615DD27B98A push    rbp
.text:00005615DD27B98B mov     rbp, rsp
.text:00005615DD27B98E mov     rdi, 38D7EA4C68000h
.text:00005615DD27B998 call    _ada__calendar__delays__delay_for
.text:00005615DD27B99D call    sub_5615DD27B616
.text:00005615DD27B9A2 call    sub_5615DD27B4AA
.text:00005615DD27B9A7 call    sub_5615DD27B372
.text:00005615DD27B9AC call    sub_5615DD27B5E2
.text:00005615DD27B9B1 call    sub_5615DD27B852
.text:00005615DD27B9B6 call    sub_5615DD27B886
.text:00005615DD27B9BB call    sub_5615DD27B8BA
.text:00005615DD27B9C0 call    sub_5615DD27B922
.text:00005615DD27B9C5 call    sub_5615DD27B3A6
.text:00005615DD27B9CA call    sub_5615DD27B136
.text:00005615DD27B9CF call    sub_5615DD27B206
.text:00005615DD27B9D4 call    sub_5615DD27B30A
.text:00005615DD27B9D9 call    sub_5615DD27B206
.text:00005615DD27B9DE call    sub_5615DD27B57A
.text:00005615DD27B9E3 call    sub_5615DD27B8EE
.text:00005615DD27B9E8 call    sub_5615DD27B40E
.text:00005615DD27B9ED call    sub_5615DD27B6E6
.text:00005615DD27B9F2 call    sub_5615DD27B782
.text:00005615DD27B9F7 call    sub_5615DD27B8EE
.text:00005615DD27B9FC call    sub_5615DD27B206
.text:00005615DD27BA01 call    sub_5615DD27B372
.text:00005615DD27BA06 call    sub_5615DD27B136
.text:00005615DD27BA0B call    sub_5615DD27B3A6
```

```
./linux_server64
IDA Linux 64-bit remote debug server(ST) v8.3.28. Hex-Rays (c) 2004-2023
2023-12-15 23:26:39 Listening on 0.0.0.0:23946...
2023-12-15 23:27:40 [1] Accepting connection from x.x.x.x...

Looking for GNU DWARF file at "/usr/lib/debug/.build-
id/4c/4687ba4c5b7fea4c9f13aaa29269a3ca164b09.debug"... no.
picoCTF{d1
```

Well, I think the next step is to let it run until all this function calls complete and then get our flag output and close the challenge

```
./linux_server64
IDA Linux 64-bit remote debug server(ST) v8.3.28. Hex-Rays (c) 2004-2023
2023-12-15 23:26:39 Listening on 0.0.0.0:23946...
2023-12-15 23:28:46 [4] Accepting connection from x.x.x.x...
Looking for GNU DWARF file at "/usr/lib/debug/.build-
id/4c/4687ba4c5b7fea4c9f13aaa29269a3ca164b09.debug"... no.
```

```
picoCTF{d15a5m_ftw_5c1d1ae}
```