Crackmes: destructeur's Sh4ll1

Website: https://crackmes.one/

Author of the crackme: destructeur

Language: C/C++

Level: 1

Index

Introduction

This is a quite easy challenge, so it will take very little time. It does not require advanced skills, just a basic knowledge of assembly. The tip is really what the crackme is all about: noise in the stack.

Tools

We will only use radare2 on this walkthrough. I am using version 2.8.0, but any other version should be fine too. If it is your first time with radare, do not panic, you do not need any kind of skill to follow this writing. Radare is quite a complex tool for beginners, but it is so cool huh. If you do not feel comfortable using it from the command line, you could use its web interface (instructions at their GitHub page). For more information on radare, head to their website and grab their book! (See last section)

Walkthrough

The first step is always to know what we are facing. We know, as stated on the challenge, that it is made to be run on Unix/linux. However, what if we had just the binary with no info at all? Lets check it with a linux utility called *file*.

```
remnux@remnux:/home/remnux/challenge$ ls
crackMe1.bin
remnux@remnux:/home/remnux/challenge$ file crackMe1.bin
crackMe1.bin: ELF 64-bit LSB  shared object, x86-64, version 1 (SYSV), dynamica
lly linked (uses shared libs), for GNU/Linux 2.6.32, BuildID[sha1]=9a480eb4b171
1f12768f5cee5915d3024a8815cc, not stripped
remnux@remnux:/home/remnux/challenge$
```

Now we know it is an ELF 64-bit file (Wow, genius!). We can also get the strings from the command line with another linux utility: *strings*. The output is bigger than intended here, so I will grep it to show the three interesting strings we should pay attention to.

```
remnux@remnux:/home/remnux/challenge$ strings crackMe1.bin | grep assword
Password:
Good password
Bad password
```

We grep “assword” (ok, sorry). So, there we have those strings, seems that they will ask us for a password. We can confirm that by running the crackme once. If it was a malware sample, you should take a snapshot called “Before detonation” first, so you can revert its execution. But these is not a malware program, so lets run it.

```
remnux@remnux:/home/remnux/challenge$ ./crackMe1.bin
Password: my_pass
Bad password
```

Nice, we have collected enough information now, lets get into radare!

We could have collected its strings from radare, but it is important to know different tools. First of all, open the binary and analyze it.

```
remnux@remnux:/home/remnux/challenge$ r2 crackMe1.bin
 -- 
[0x000008a0]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Constructing a function name for fcn.* and sym.func.* functions (aan)
[x] Type matching analysis for all functions (afta)
[x] Use -AA or aaaa to perform additional experimental analysis.
[0x000008a0]>
```

Lets see the strings radare has localized.

```
[0x000008a0]> fs strings
[0x000008a0]> f
0x00000b85 11 str.Password:
0x00000b90 14 str.Good_password
0x00000b9e 13 str.Bad_password
0x000008a0 1 str.
[0x000008a0]>
```

The first command, *fs strings*, selects a flagspace (strings in this particular case). So now we can issue *f*, which will list the flags. Remember that we have switched to strings' flagspace, so we are only getting flags from that space. As we obtained before, there we have our three interesting strings.

Ok, so now we could go to wherever the binary is using the string *Password*, since that is the string it shows us when we have to enter the password for the challenge.

We can get references to it with *axt*.

```
[0x000008a0]> axt str.
str.Password:       str.Good_password
str.Bad_password
[0x000008a0]> axt str.Password:
sym.systemo 0xa0a [DATA] lea rsi, str.Password:
[0x000008a0]>
```

Our string is being used at 0xa0a, which is in a function called *system*. As you can notice, system seems to be a flag of a flagspace called *sym*. Nice, that is the flagspace for functions. Lets head there.

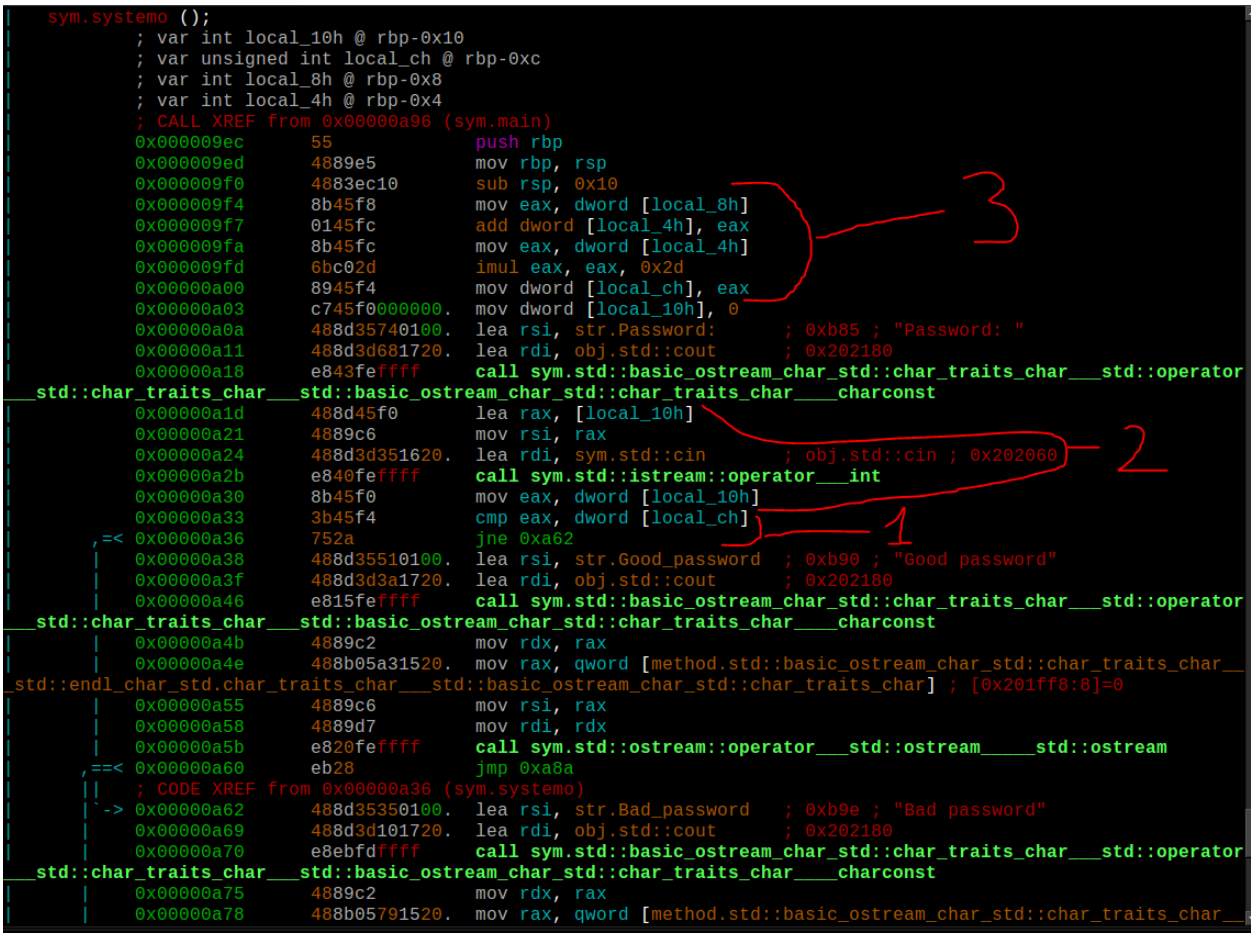
We head there with *s* and disassemble 20 bytes with *pd*.

```
[0x000008a0]> s sym.systemo
[0x000009ec]> pd 20
/ (fcn) sym.systemo 161
|   sym.systemo ();
|           ; var int local_10h @ rbp-0x10
|           ; var unsigned int local_ch @ rbp-0xc
|           ; var int local_8h @ rbp-0x8
|           ; var int local_4h @ rbp-0x4
|           ; CALL XREF from 0x00000a96 (sym.main)
|           0x000009ec      55              push rbp
|           0x000009ed      4889e5          mov rbp, rsp
|           0x000009f0      4883ec10        sub rsp, 0x10
|           0x000009f4      8b45f8          mov eax, dword [local_8h]
|           0x000009f7      0145fc          add dword [local_4h], eax
|           0x000009fa      8b45fc          mov eax, dword [local_4h]
|           0x000009fd      6bc02d          imul eax, eax, 0x2d
|           0x00000a00      8945f4          mov dword [local_ch], eax
|           0x00000a03      c745f0000000.   mov dword [local_10h], 0
|           0x00000a0a      488d35740100.   lea rsi, str.Password:      ; 0xb85
; "Password: "
|           0x00000a11      488d3d681720.   lea rdi, obj.std::cout      ; 0x2021
80
|           0x00000a18      e843feffff     call sym.std::basic_ostream_char_std
::char_traits_char___std::operator___std::char_traits_char___std::basic_ostream
_char_std::char_traits_char____charconst
```

We see it is a function called by the *main* function (see the line which says “CALL XREF …”). The function has 4 local variables, but no arguments at all. The interesting part here is that it is using its local variables without initializing them! That smells tricky, but it is soon, we still do not know what the purpose of those variables is. Lets just take note of that, we will be coming back here later. What we need to notice know is the point where it decides if our password is good or not, so we are gonna head to the reference for the string *Good password*.

```
[0x000009ec]> axt str.Good_password
sym.systemo 0xa38 [DATA] lea rsi, str.Good_password
[0x000009ec]>
```

It is referenced on *systemo*, the same function we were examining later. Lets grab a screenshot of a big disassembly of the function and get more information.

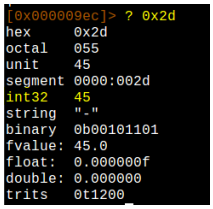


Lets break it into pieces:

1. Here it is a comparison between **eax** and the local variable **local_c**. In order to get to where we want to go (*Good password*), we need them to be equal, so it does not take the jump. Ok, then, what is **eax** and what is **local_c**?
2. At this point, we see there is a call to a well-known function in C++: *cin*. This function is used to get input from the user (note above that it calls *cout* when printing "*Password:"*). Before calling to *cin*, it moves **local_10** to **rsi**, which seems to be a parameter into the function (see *cout* above, it takes rsi as a parameter). Then, it moves **local_10** to **eax**, which is used to compare against **local_c**. So, at this point, we can assume that **local_10** is the string we write when we are asked for a password. Moreover, we know **local_10** must be equal to **local_c**, from where we can say that **local_c** is the password we need to get.
3. This is the last time **local_c**'s value is modified. From these lines, we know the following:

   local_c = (local_8 + local_4) * 0x2D

   Note: we can use radare to evaluate math expressions and convert data. Try "? 0x2d".



What we need to know is those values, as these local variables are not initialized inside the function. Here is the trick of the challenge. These values may have been initialized somewhere else, although that would not be standard. Remember the tip: *noise in the stack*.

Note: if you do not feel comfortable on how the stack works, refer to Appendix A: Stack.

Lets see where does main call this function.

```
[0x000009ec]> axt
sym.main 0xa96 [CALL] call sym.systemo
[0x000009ec]> s sym.main
[0x00000a8d]> pdf
|           ;-- main:
/ (fcn) sym.main 21
|   sym.main ();
|           ; DATA XREF from 0x000008bd (entry0)
|           0x00000a8d      55              push rbp
|           0x00000a8e      4889e5          mov rbp, rsp
|           0x00000a91      e83affffff      call sym.systemv
|           0x00000a96      e851ffffff      call sym.systemo
|           0x00000a9b      b800000000      mov eax, 0
|           0x00000aa0      5d              pop rbp
\           0x00000aa1      c3              ret
[0x00000a8d]>
```

Notice it is calling another function just before calling the one we have been disassembling. Lets get into *sym.systemv*.

```
[0x00000a8d]> s sym.systemv
[0x000009d0]> pdf
/ (fcn) sym.systemv 28
|   sym.systemv ();
|           ; var int local_ch @ rbp-0xc
|           ; var int local_8h @ rbp-0x8
|           ; var int local_4h @ rbp-0x4
|           ; CALL XREF from 0x00000a91 (sym.main)
|           0x000009d0      55              push rbp
|           0x000009d1      4889e5          mov rbp, rsp
|           0x000009d4      c745fc050000.   mov dword [local_4h], 5
|           0x000009db      c745f8070000.   mov dword [local_8h], 7
|           0x000009e2      c745f4f50100.   mov dword [local_ch], 0x1f5
|           0x000009e9      90              nop
|           0x000009ea      5d              pop rbp
\           0x000009eb      c3              ret
[0x000009d0]>
```

If you are comfortable enough with reversing and the stack, you know the trick now, but lets explain it for those who are beginners. This function has three local variables. Local variables are stored on the stack, which grow towards the lowest addresses. This function can start storing local variables and using them from *rbp* towards lower positions. It initializes *local_4*, *local_8* and *local_c*, but does not use them

(weird). The point is, when a function leaves, it does not remove the values from the stack, it just restores stack pointers (*rbp* and *rsp*). Everything outside of the scope of the current stack is considered garbage. When a function pushes values onto the stack, it considers them garbage and just overwrites its values. Why do them act like this? The answer is: efficiency. It would take time and cpu cycles to remove those values from the stack. So, what is happening here is that this function is initializing these values on the stack and the next one is using them. We now know from these function that:

- Local_4 = 5
- Local_8 = 7
- Local_c = 0x1F5

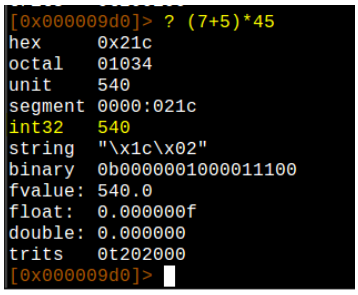
These variables are stored at *rbp-0x4*, *rbp-0x8* and *rbp-0xc*, respectively. If we go back to the first function we analyzed, these are the same addresses of the local variables from that function (note that *rbp* does not change between each call). Then we substitute the values in our equation (note that *local_c*'s value is not used) and get:

local_c = (7 + 5) * 0x2D

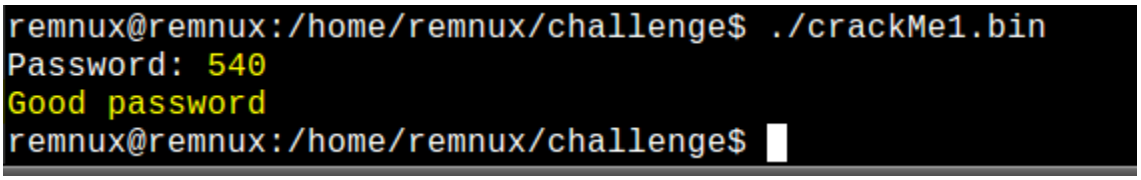
We said 0x2D was 45, so:

local_c = (7 + 5) * 45
local_c = 540

It is not a complex operation, but we can do it with radare as follows.



If we are not mistaken, that is the correct password.



Cheers!

Appendix A: Stack

The stack is LIFO structure, which means *Last In, First Out*. It grows upside-down, towards lower addresses. *Rbp* and *Rsp* (*ebp* and *esp* on x86-32 architectures) registers point to the base of the stack and its current position, respectively. They indicate the stack's window the function owns. Everything from *rbp* to *rsp* are local variables. When it pushes a new value onto the stack, it is pushed on address indicated by *rsp* and *rsp* is incremented to point to the next address on the stack. When the function returns, it makes *rsp* point to the base (*rbp*) and restores *rbp* to its original value before the function's call (which is stored at *rbp*+0). Note that it does not remove or clean the stack, it just restores *rbp* and *rsp*. Everything out of the stack is considered garbage.

For more information, please refer to higher quality resources, such as some of the following:

- https://securedorg.github.io/RE101/section1.2/
- https://hshrzd.wordpress.com/how-to-start/ (points to interesting resources)
- https://www.begin.re/x86-overview

References

- Radare: https://rada.re/r/index.html
- Radare Book: https://radare.gitbooks.io/radare2book/content/
- ELF: https://en.wikipedia.org/wiki/Executable_and_Linkable_Format