Crackmes: destructeur's Sh4ll1

Website: <https://crackmes.one/>

Author of the crackme: destructeur

Language: C/C++

Level: 1

Index

[Introduction 3](#_Toc519850538)

[Tools 3](#_Toc519850539)

[Walkthrough 3](#_Toc519850540)

[Appendix A: Stack 9](#_Toc519850541)

[References 10](#_Toc519850542)

# Introduction

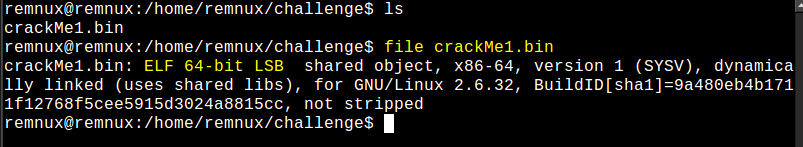
This is a quite easy challenge, so it will take very little time. It does not require advanced skills, just a basic knowledge of assembly. The tip is really what the crackme is all about: noise in the stack.

# Tools

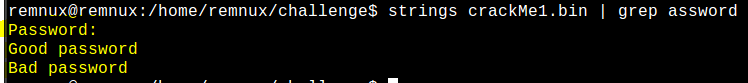
We will only use radare2 on this walkthrough. I am using version 2.8.0, but any other version should be fine too. If it is your first time with radare, do not panic, you do not need any kind of skill to follow this writing. Radare is quite a complex tool for beginners, but it is so cool huh. If you do 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*.



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.

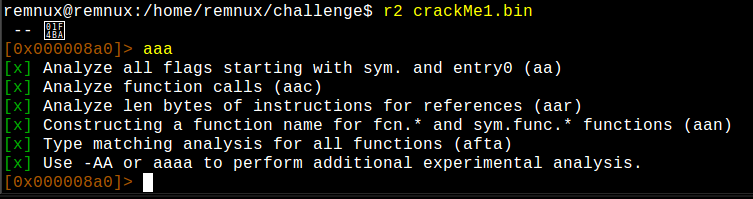


We grep “assword” (ok, sorry). So, there we have those strings, seems that they will ask us for a password. We can confirm that by running the crackme once. If it was a malware sample, you should take a snapshot called “Before detonation” first, so you can revert its execution. But these is not a malware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

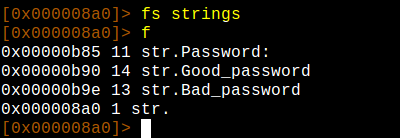


Nice, we have collected enough information now, lets get into radare!

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.



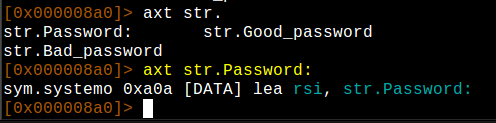
Lets see the strings radare has localized.



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.

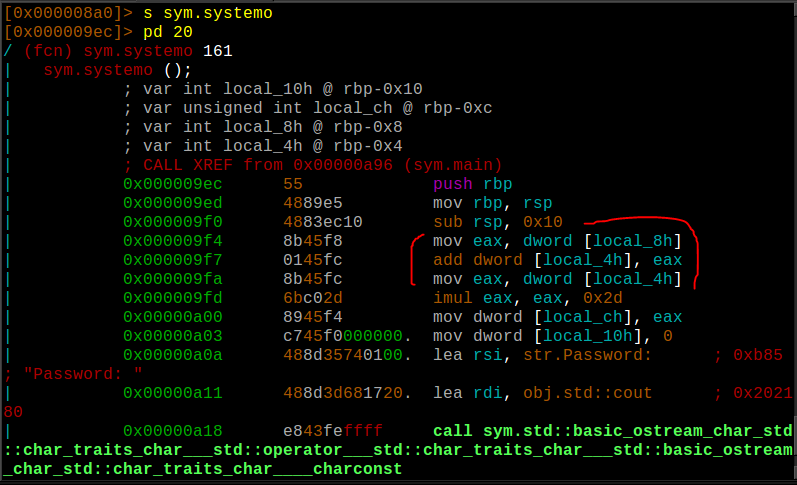
Ok, so now we could go to wherever the binary is using the string *Password*, since that is the string it shows us when we have to enter the password for the challenge.

We can get references to it with *axt*.

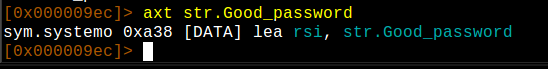


Our string is being used at 0xa0a, which is in a function called *system*. As you can notice, system seems to be a flag of a flagspace called *sym*. Nice, that is the flagspace for functions. Lets head there.

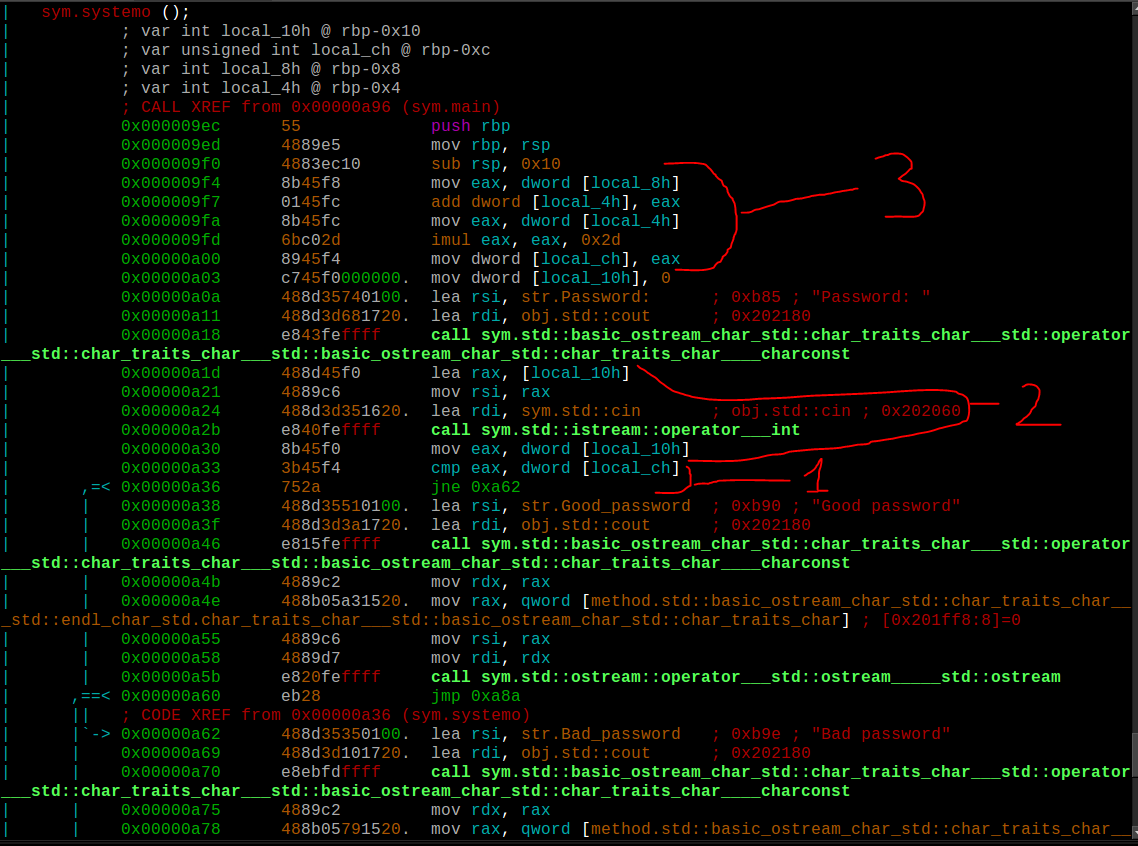
We head there with *s* and disassemble 20 bytes with *pd*.



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*.



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.

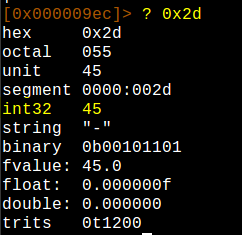


Lets break it into pieces:

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**

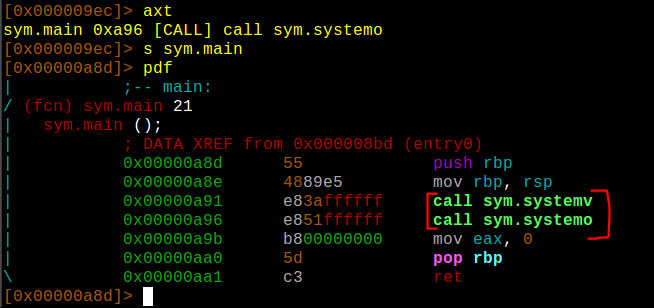
Note: we can use radare to evaluate math expressions and convert data. Try “? 0x2d”.



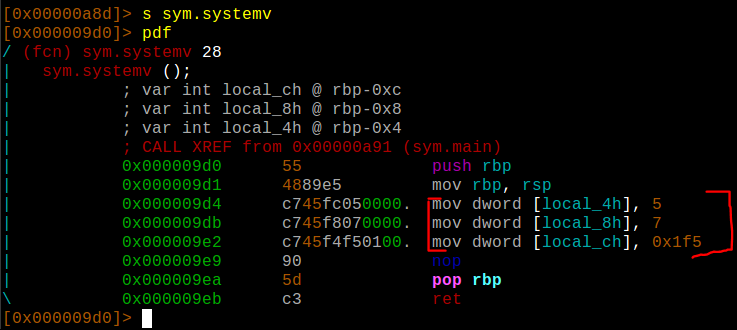
What we need to know is those values, as these local variables are not initialized inside the function. Here is the trick of the challenge. These values may have been initialized somewhere else, although that would not be standard. Remember the tip: noise in the stack.

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.



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



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:

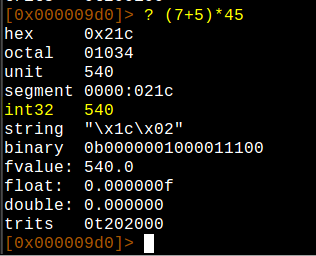
**local\_c = (7 + 5) \* 0x2D**

We said 0x2D was 45, so:

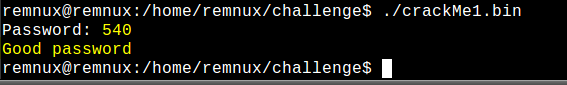
**local\_c = (7 + 5) \* 45**

**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>