Reverse Engineering Challenges solved step-by-step to demonstrate the many uses of reverse engineering for the Graduate Programs at the Polytechnic University of Puerto Rico

Joel Maldonado
Computer Sciences Program
Dr. Jeffrey Duffany
Computer Sciences Department
Polytechnic University of Puerto Rico

Abstract *— The field of reverse engineering has seen many different applications such as analysis of computer viruses and malware such as trojans, worms, viruses, ransomware, and so on. Other uses involve analyzing legacy code to possibly recreate in a more modern program and can even be used to test the quality of software. These are just some of the uses of reverse engineering, which should be discussed and be more well known throughout people who practice coding. There are many different approaches to reverse engineering since one can use tools such as IDA, CFF Explorer, Ghidra, Hopper, GDB, and many others in order to examine the programs. Being able to properly understand how to use these tools will help in the proper understanding of what the code is doing and how it is behaving which will be better demonstrated by solving reverse engineering challenges and explaining the methodology behind how they were solved.*

Key Terms- *GDB, reverse engineering*

INTRODUCTION

A technological area which should be better known about because of the many uses it provides is reverse engineering. Reverse engineering is an area that has seen many different implementations and approaches which has helped tackle different problems such as working with legacy code [1], analyzing malware to determine how it functions and what it targets [2], and has also been used to improve existing software. Despite the wide range of implementations that reverse engineering offers it is seldom taught as a core class in many programs which is why bringing more awareness to it and being able to demonstrate to other computer scientist and anyone the possibilities and uses it brings should be done so.

GDB which is also known as the GNU debugger, can be used in programming languages such as C, C++, Go, among others. [3] With GDB you can disassemble specific parts of code one wants to analyze into assembly language, and this can be used to interpret the code and reverse engineer it. Assembly language is a low-level programming language which was created with the purpose to be able to directly communicate with a computer's hardware and is also readable by humans. Assembly language is also used when converting high-level programming languages such as Python, Ruby, C#, Java, etc. Assembly language is barely written directly since humans instead use high-level programming languages which are then converted to assembly language which is then used in machine language which is basically in binary or 1s and 0s. [4] With GDB basic concepts of reverse engineering can be demonstrated and step by step processes can be shown. There are also other tools that are able to provide better insight and are more aware of techniques that are used to try to thwart reverse engineering attempts, these tools include IDA, Ghidra, etc.

Reverse engineering can be considered as another aspect of cybersecurity since it can be used to analyze malware and other such programs in order to determine what vulnerabilities are being exploited and be able to counter these, because of this reverse engineering challenges can often be found in cyber security competitions and challenges can also be found online in order to help individuals learn and practice. These challenges vary and can involve different type of Operating Systems and not all challenges can be solved using the exact same process since the difficulty varies and not all challenges are alike. [2]

With all of the previously mentioned in mind, showing how to solve reverse engineering challenges from the crackmes.one webpage in a step-by-step manner will hopefully be able to demonstrate the importance of reverse engineering, the applications it can be used for and show that it isn't as frightening as one may think when they hear about reverse engineering. The crackmes.one page is a website that was made to continue on the spirit of a previous website known as cracmes.de which no longer exists. The website is a place for any reverse engineer to upload their challenges in order to help others practice and learn reverse engineering skills, techniques and to improve what they know. The website also allows you to submit your solution and offers a variety of challenges which have varying difficulty and ways to be solved. In this work these challenges will be solved by simply using GDB.

BACKGROUND

The reason for selecting this as the focus of research for the design project is because of many varied reasons. Mainly because even though I have had some experience with reverse engineering, I feel I have not seen as much of it as I should and feel that it is a fascinating topic which merits more discussion between the topics of computer science. The first experience I had with reverse engineering was basically a challenge for the computer architecture class I took which involved a project known as the "binary bomb" which I found to be interesting and fully enjoyed. It consisted of many distinct phases in which you had to understand what was occurring with the assembly language and keep track of the registers in order to be able to get the necessary input in order to pass to the next phase of the challenge.

I also took a reverse engineering elective where we started with the basics, but then higher leveled challenges and tools were used. Both of these experiences have given me a perspective of the importance that reverse engineering has and how not many students get to experience and view these. This and the fact that I wanted to go back and work on reverse engineering again is what prompted this step-by-step solution to reverse engineering challenges for the design project since I believe more computer science students should try to get involved and experience this area so that they too can get a perspective on it and understand it's importance.

EQUIPMENT AND MATERIALS

Software Components

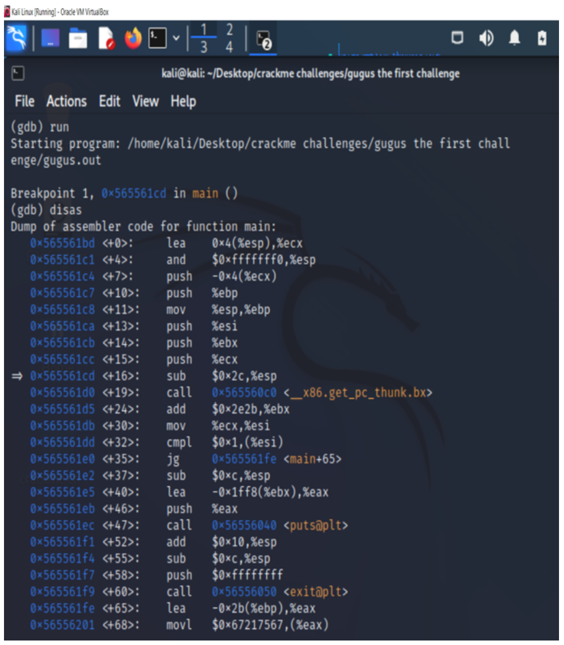
The materials that are mentioned in this project are open tools that are open source and free to use that have been created by different organizations and individuals focusing on cybersecurity. The software involved includes:

- **VirtualBox:** VirtualBox is known as a powerful x86 and AMD64/Intel 64 virtualization software that allows the running of virtual machines on your machine without the need for the installation of the other OS on your physical machine. It offers many features, high performance and is also an open-source software. Some of the guest operating systems that it includes are Windows (NT 4.0, 2000, XP, Server 2003, Vista, Windows 7, Windows 8, Windows 10), DOS/Windows 3.x, Linux (2.4,2.6,3.x and 4.x), Solaris and Open Solaris, OS/2, and OpendBSD. [5]
- **Kali Linux:** is an open-source, Debian-based Linux distribution which was created in order to be used for security related task such as Penetration Testing, Security Research, Computer Forensics and Reverse Engineering. [6]
- **GDB:** which is a known as a portable debugger that is able to run on many Unix-like systems and is able to be used on many different programming languages such as Ada, C, C++, Fortran, Go, and many others. The GNU Project debugger was made in order to allow users to be able to determine what is occurring inside another program while it is executing. [3]

METHODOLOGY

The approach that was used in order to pick challenges and solve them in a step-by-step manner in order to be able to demonstrate how one could go about doing reverse engineering challenges and gain a better understanding of the process behind it involved considering tackling basic challenges in order to be able to demonstrate reverse engineering concepts to readers who may have not been exposed to assembly language or GDB before. In doing so assembly language concepts are explained in detail to provide an understanding of what exactly is occurring in the challenge that is being solved and to explain any possibly unknown concepts that the reader may stumble upon. Since reverse engineering challenges are not all identical and can vary on their process an explanation on everything that was done is provided as well in hope of being able to highlight how reverse engineering can vary. Six challenges were chosen to demonstrate the varying approaches that can be taken to solve a challenge and to highlight the difference between challenges.

In order to solve problems that were chosen from the crackmes.one site, they were first analyzed by using tools or commands such as the file command in order to get more details about what type of executable file was being worked on and then proceeding to use GDB in order to take a much more in-depth analysis of the executable file. GDB has many different commands that have different uses in helping better understand the code such as disassemble (disas) which allows the user to disassemble a specific function or a function fragment thus allowing a closer look at the assembly code to that specific area, breakpoint which allows the user to specify a breakpoint which will make the execution of the program halt once reached, next instruction (ni) which allows the user to go to the following instruction in the assembly code therefore giving the user a chance to verify changes that occurred, the examined instruction which allows the examination of the provided memory but also allows one to place a flag in order to indicate how the values should be represented before it is displayed, info functions in order to see debugging symbols that can be accessed in order to help with the analysis of the program and many others such commands which are used in order to tackle the challenges. Figure 1 shows the disassembly command being used in one of the challenges in order to display the main function of the challenge program that is being executed at the moment by using GDB.

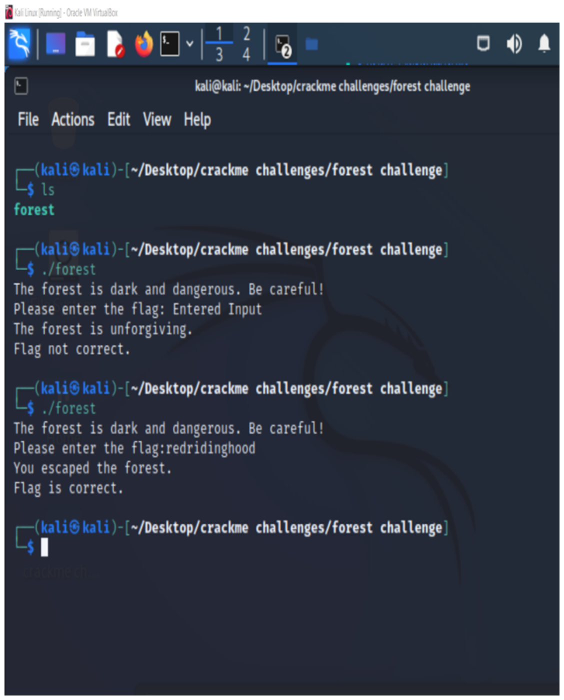

Kali Linux [Running] - Oracle VM VirtualBox
kali@kali: ~/Desktop/crackme challenges/gugus the first challenge
File Actions Edit View Help

```
(gdb) run
Starting program: /home/kali/Desktop/crackme challenges/gugus the first chall
enge/gugus.out

Breakpoint 1, 0×565561cd in main ()
(gdb) disas
Dump of assembler code for function main:
   0×565561bd <+0>:     lea    0×4(%esp),%ecx
   0×565561c1 <+4>:     and    $0×fffffff0,%esp
   0×565561c4 <+7>:     push   -0×4(%ecx)
   0×565561c7 <+10>:    push   %ebp
   0×565561c8 <+11>:    mov    %esp,%ebp
   0×565561ca <+13>:    push   %esi
   0×565561cb <+14>:    push   %ebx
   0×565561cc <+15>:    push   %ecx
⇒ 0×565561cd <+16>:    sub    $0×2c,%esp
   0×565561d0 <+19>:    call   0×565560c0 <__x86.get_pc_thunk.bx>
   0×565561d5 <+24>:    add    $0×2e2b,%ebx
   0×565561db <+30>:    mov    %ecx,%esi
   0×565561dd <+32>:    cmpl   $0×1,(%esi)
   0×565561e0 <+35>:    jg     0×565561fe <main+65>
   0×565561e2 <+37>:    sub    $0×c,%esp
   0×565561e5 <+40>:    lea    -0×1ff8(%ebx),%eax
   0×565561eb <+46>:    push   %eax
   0×565561ec <+47>:    call   0×56556040 <puts@plt>
   0×565561f1 <+52>:    add    $0×10,%esp
   0×565561f4 <+55>:    sub    $0×c,%esp
   0×565561f7 <+58>:    push   $0×ffffffff
   0×565561f9 <+60>:    call   0×56556050 <exit@plt>
   0×565561fe <+65>:    lea    -0×2b(%ebp),%eax
   0×56556201 <+68>:    movl   $0×67217567,(%eax)
```

Figure 1
Disassembly of main function in GDB

In figure 1 where the arrow starts is where the main function starts, so one just has to keep track what is going on in the assembly code, the changes occurring in the registers and values, and proceed instruction by instruction to then analyze and understand what is occurring to then solve the challenge. As figure 2 shows, the main objective of most challenges is to find what sort of input it is expecting, if you write the correct input the challenge will be solved, but if it is incorrect, it will usually display that you got it wrong.


kali@kali: ~/Desktop/crackme challenges/forest challenge
File Actions Edit View Help

```
┌──(kali㉿kali)-[~/Desktop/crackme challenges/forest challenge]
└─$ ls
forest

┌──(kali㉿kali)-[~/Desktop/crackme challenges/forest challenge]
└─$ ./forest
The forest is dark and dangerous. Be careful!
Please enter the flag: Entered Input
The forest is unforgiving.
Flag not correct.

┌──(kali㉿kali)-[~/Desktop/crackme challenges/forest challenge]
└─$ ./forest
The forest is dark and dangerous. Be careful!
Please enter the flag:redridinghood
You escaped the forest.
Flag is correct.

┌──(kali㉿kali)-[~/Desktop/crackme challenges/forest challenge]
└─$ █
```

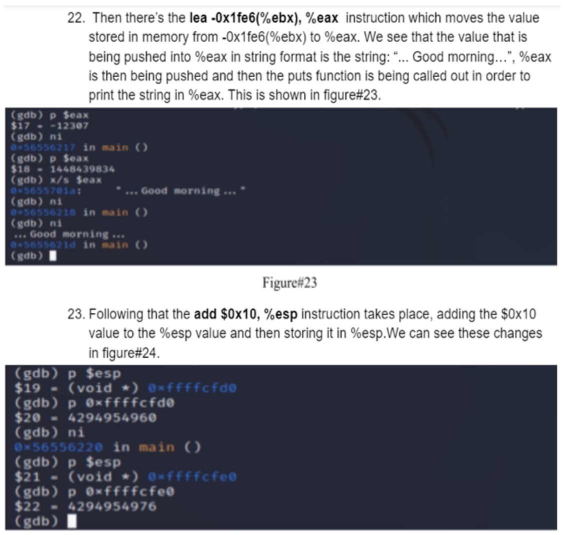
Figure 2
Example of challenge prompt and indication of right and wrong answer

Figure 2 shows the executable file of the challenge being run, in which it then prints out the first line and part of the second, but then waits for the user to input the flag. Keeping this in mind we develop a write up step-by step guide to explain what is going on in each step, verify register and other areas in memory in order to explain reasoning, deductions and show the problem-solving process involved when tackling these challenges that lead to the solutions.

This writeup is organized by first indicating which problem is being solved and indicating where others can download the challenge and is then proceeded by steps in which assembly and reverse engineering concepts are explained as they are seen. All assembly code that is being explained is accompanied by a figure in order to better illustrate what is being explained and also is shown to demonstrate how values in memory were accessed in order to reach the given problem solution.

Therefore, when tackling the challenges, personal notes were being taken at the same time along with screenshots with the findings being discovered in order to later on be able to organize these in a proper document for the step-by-step process for interested individuals to follow. Not only that but even the most basic of commands or instructions are discussed allowing individuals of any level of experience to be able to follow the guide. As previously mentioned, the guide was developed with the concept of being able to allow individuals who are following the guide to understand how the challenges are being solved even when they don't have any experience with these instructions or tools and therefore what is occurring should be described in a basic, clear, and concise way to avoid confusion.

An example of the format can be seen in figure 3, which displays how memory changes occur until it has the string "…Good morning…" loaded into the register, which is then pushed in order to print out when the code is ran. This is one of the many steps and different assembly instructions that are discussed and presented in this write up step-by-step document. It goes down to the basics and tries to cover all the possible questions that the reader may have when it comes to how certain calculations occurred or when did certain areas of memory change, which is why these are also demonstrated and explained in order to not leave any opening for misinterpretations and have all the details of the process clear. After problems are solved, it is also shown how to run the challenges and view that the correct answer was reached, therefore proving that the challenge was successfully solved. With this interested individual that want to explore reverse engineering will have the chance to go through the entire process of solving the problem as well and have the confirmation of the answer in the end.



22. Then there's the **lea -0x1fe6(%ebx), %eax** instruction which moves the value stored in memory from -0x1fe6(%ebx) to %eax. We see that the value that is being pushed into %eax in string format is the string: "... Good morning...", %eax is then being pushed and then the puts function is being called out in order to print the string in %eax. This is shown in figure#23.

```
(gdb) p $eax
$17 = -12307
(gdb) ni
0x56556217 in main ()
(gdb) p $eax
$18 = 1448439834
(gdb) x/s $eax
0x5655701a:     "... Good morning ..."
(gdb) ni
0x56556218 in main ()
(gdb) ni
... Good morning ...
0x5655621d in main ()
(gdb) 
```

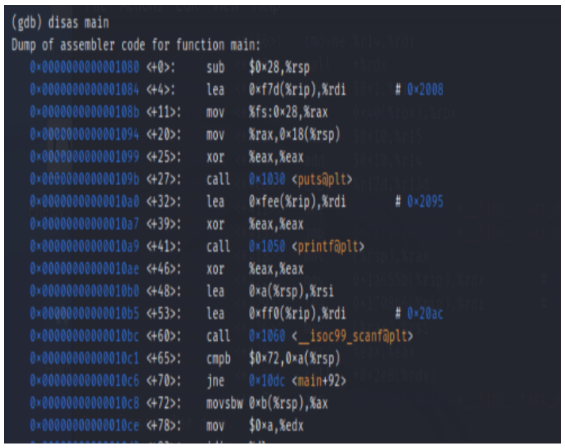
Figure#23

23. Following that the **add $0x10, %esp** instruction takes place, adding the $0x10 value to the %esp value and then storing it in %esp.We can see these changes in figure#24.

```
(gdb) p $esp
$19 = (void *) 0xffffcfd0
(gdb) p 0xffffcfd0
$20 = 4294954960
(gdb) ni
0x56556220 in main ()
(gdb) p $esp
$21 = (void *) 0xffffcfe0
(gdb) p 0xffffcfe0
$22 = 4294954976
(gdb) 
```

Figure 3
Example of write up document demonstration and explanation

Now that this was discussed we can briefly explain how each of the six challenges were solved. The first challenge, MKesenheimer's Forest, states that it has multiple hidden strings but only some are needed in order to get the correct answer. [7] We can see how it's normally run in Figure#2. One of the possible answers is also shown but how the solution was reached was by doing the following. We run the executable file in gdb and use the info functions command which will show that a main function is accessible. A breakpoint is set for the main function and the run command is done in order to start the program which will then stop at the breakpoint in the main function. We can then use the disas command in order to look at the assembly code to analyze it and get a better grasp of what is occurring, this can be seen in figure 4 which shows part of the assembly that makes up the main function. As you follow through the assembly it can be seen that each individual character of the string that was given as an input will be compared to specific values as the assembly code goes on. For example, the first comparison being made to the first character in the string is 0x72, which is hexadecimal for r. Many of the comparisons are in hexadecimal like the previously stated example and if the comparison fails the program will print the wrong solution string seen in figure 2. There are some comparisons however where the answer can vary, such is the case with the second character's decimal representation, which will take any value whose division will give a remainder of one which is then subtracted by 1. Examples that work are 'e', which is 101 in decimal, o, which is 111 in decimal, and other such values. The following value is d which is discovered by following the assembly instructions. The reason for this being that d is the only value that when is square rooted and then multiplied by 5 as we saw will then equal to 50 which is the comparison which is being made. The rest of the characters in the string can be verified by direct comparisons being made to the hex values found in the assembly and the characters of the input provided. Therefore, one of the possible solutions for this challenge is the string 'redridinghood'.



```
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000001080 <+0>:     sub    $0x28,%rsp
   0x0000000000001084 <+4>:     lea    0xf7d(%rip),%rdi        # 0x2008
   0x000000000000108b <+11>:    mov    %fs:0x28,%rax
   0x0000000000001094 <+20>:    mov    %rax,0x18(%rsp)
   0x0000000000001099 <+25>:    xor    %eax,%eax
   0x000000000000109b <+27>:    call   0x1030 <puts@plt>
   0x00000000000010a0 <+32>:    lea    0xfee(%rip),%rdi        # 0x2095
   0x00000000000010a7 <+39>:    xor    %eax,%eax
   0x00000000000010a9 <+41>:    call   0x1050 <printf@plt>
   0x00000000000010ae <+46>:    xor    %eax,%eax
   0x00000000000010b0 <+48>:    lea    0xa(%rsp),%rsi
   0x00000000000010b5 <+53>:    lea    0xff0(%rip),%rdi        # 0x20ac
   0x00000000000010bc <+60>:    call   0x1060 <__isoc99_scanf@plt>
   0x00000000000010c1 <+65>:    cmpb   $0x72,0xa(%rsp)
   0x00000000000010c6 <+70>:    jne    0x10dc <main+92>
   0x00000000000010c8 <+72>:    movsbw 0xb(%rsp),%ax
   0x00000000000010ce <+78>:    mov    $0xa,%edx
```

Figure 4
Part of the disassembly of the main function for challenge # 1 example

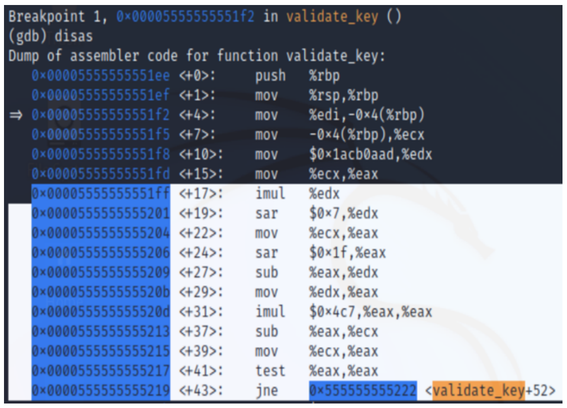
The second challenge, gugus the first challenge, we use gdb to solve the problem. [8] The info functions command is used to verify what functions are present in the program. Breakpoint are set in the main function; the run command is used to start the program in GDB and it stops in the breakpoint which was previously set up. The disas command is used to disassemble the main function assembly to analyze it. After looking at the disassembly and playing with a string input one can notice that the strcmp function, which compares two strings, is being called and is taking the given input and is also taking the string 'gu!gu?s' to compare the values. By following the assembly instructions and verifying certain areas in memory one can determine that the string is the needed input for the challenge to be solved.

The third challenge, license checker 0x02, follows the same approach as the previous two problems. [9] Using GDB we verify the functions that are in the program and set up breakpoints to the functions that we want to verify. The breakpoint is set up in the main function and the disas command is used to disassemble the main function and be able to examine the assembly. By examining what is being loaded into memory we can see that the string “NomanProdhan” is being moved into the %rdx register while the string which was input is in %rax. The string comparison is then called in order to compare both of those strings. After that it will load the license information later on in the assembly that is needed which as well is "KS-LICENSE-KEY-2021-REV-2" which is also compared to the value that was placed as an input for our license plate. Both strings used together is the answer for this challenge.

The fourth challenge, Super Easy, also has the basic approach that has been followed up till now by using GDB. [10] First the info functions command and then the breakpoints are set. This program has two interesting functions which are main and the check_pass function, so a look is taken at both. The main function is used to send along the input to the check_pass function. In check_pass it verifies to see if there is a second value to the command line arguments provided when the code was run. The strlen and atoi function is then used in the input that was made. The following will then compare the value of the strlen that was obtained from the input and verify if it at least has a length of three, if not the program will fail. The program then proceeds to send the value that was input to the is_prime function in order to verify if it is a prime value. Afterwards by looking at the rest of the assembly you can conclude that for the password to work it must be any number that is at least 3 digits long and is a prime number for it to be an accepted password.

The fifth challenge, SilentWRaith's lockcode, was solved by first verifying the functions the program has. [11] The program has a main function and a val function which will both be checked. In the main function the program verifies how many command line arguments were passed, if not enough an error message will be displayed. The program loads a string into %rax which then is passed to the strlen function to finally have both the string from the program and its length be sent to the val function. The val function has a for loop which will add the character value currently being looked at to a variable until all the characters of the string have been passed. The val function is then left and afterwards the input that was made is passed to the val function as well along with its length. It does the same and does the for loop and addition of the values. Once it leaves the val function this time however the sum of the characters is then passed to the res function along with the integer 2977. In the res function one is added to 2978 and then that value is subtracted by the sum of the characters of the string which was obtained previously. Afterwards it will do a comparison in order to verify if the value after the subtraction was 1, if it is not it will proceed to jump and print out that the password is wrong, but if it is equal 1 it will print out the congratulation message. With this you can determine that the value of the sum of the characters of the strings must be equal to 2977 for the password to be accepted. An example of a string which is accepted would be the string ‘HelloThereGeneralKenobiSkywaHH’.

Lastly the sixth challenge, ezman’s easy keyg3nme, was solved in a similar fashion. [12] The program is placed on GDB and is checked with the info function command to verify the functions. By doing so the main and validate_key function is discovered. Breakpoints are set for both functions and the program are run in GDB. In the main function it asks the user for an input and stores the value in a register to then later on pass it to the validate_key function. The assembly instructions to look out for here are highlighted in figure 5. We pay close attention to that last imul specifically because of the implications it has which are: if %eax is 0, that multiplication will result in a value of 0 which is then being subtracted by a value and then being used in a test instruction in order to determine whether the jump should occur or not. The point of the arithmetic being done beforehand is another way of checking if the value is equal to the value $04c7, which is 1223 in decimal, then if it is it will move 1 into %eax and return said value, else it will return a zero and an incorrect password message.



```
Breakpoint 1, 0x00005555555551f2 in validate_key ()
(gdb) disas
Dump of assembler code for function validate_key:
   0x00005555555551ee <+0>:     push   %rbp
   0x00005555555551ef <+1>:     mov    %rsp,%rbp
⇒  0x00005555555551f2 <+4>:     mov    %edi,-0x4(%rbp)
   0x00005555555551f5 <+7>:     mov    -0x4(%rbp),%ecx
   0x00005555555551f8 <+10>:    mov    $0x1acb0aad,%edx
   0x00005555555551fd <+15>:    mov    %ecx,%eax
   0x00005555555551ff <+17>:    imul   %edx
   0x0000555555555201 <+19>:    sar    $0x7,%edx
   0x0000555555555204 <+22>:    mov    %ecx,%eax
   0x0000555555555206 <+24>:    sar    $0x1f,%eax
   0x0000555555555209 <+27>:    sub    %eax,%edx
   0x000055555555520b <+29>:    mov    %edx,%eax
   0x000055555555520d <+31>:    imul   $0x4c7,%eax,%eax
   0x0000555555555213 <+37>:    sub    %eax,%ecx
   0x0000555555555215 <+39>:    mov    %ecx,%eax
   0x0000555555555217 <+41>:    test   %eax,%eax
   0x0000555555555219 <+43>:    jne    0x555555555222 <validate_key+52>
```

Figure 5
Highlighting the math operations being done in the challenge to compare answer

For purposes of the report the explanations are kept to a minimum but are accompanied with a report of each challenge and step by step analysis on how they were solved. In the end the important thing is that the process of reaching the answer is understood instead of jumping straight to the answer will help develop the reverse engineering skills and help with challenging other problems later on.

Results and Discussion

The results from this project are that of a step-by-step documentation on how each challenge was solved and contains the explanation of things such as basics of assembly code, what is occurring as the program executes, and figures to be able to guide the individuals reading the documentation through the challenges while also learning from it as they experience it by hands on application on these challenges.

With this computer science and cybersecurity students will hopefully be able to follow the steps with explanations and understand what is occurring in the provided figures. Feedback and discussion on how the problems were solved is also essential to consider since it is important avoid tunnel vision and to have the documentation be understandable, informative, and practical for those interested in practicing reverse engineering concepts to expand their knowledge and develop their skills when working on such problems. This can only really be solved if they properly understand what is transpiring when the code is running, which is what the tools are for, and be able to figure out the solutions or the programs purpose when it is being executed.

Conclusion

With this step-by-step documentation on how to solve the challenges hopefully other computer science and cybersecurity students gain some basic knowledge of reverse engineering concepts, develop curiosity, and interest in this field and try to do more research on it. As discussed previously, reverse engineering has many applications from analyzing malware in order to counter the vulnerabilities they exploit, to working with legacy code. With more individuals working, practicing and being aware of the uses of reverse engineering more can be discussed and brought to the community it has but can also present unique points of view which can definitely incorporate different types of thinking which could be discussed and could help the area of reverse engineering keep growing and expanding with new tools, concepts and discoveries.

Future Work

There are many different proposals that can be considered for future works such as using specific tools such as IDA or Ghirdra in order to highlight the capabilities of said tool and how they are able to fend off some of the techniques that exists in order to try to stop the reverse engineering process from being achieved. This could be achieved by taking some higher-level challenges or identifying some of these techniques and apply them in order to then try to reverse engineer them. Another potential future work involves taking several reverse engineering tools to solve the same set of challenges to determine the strengths and weaknesses of the tools and to evaluate whether they can detect techniques that are used to throw off reverse engineering attempts.

As mentioned before reverse engineering can also be utilized for malware analysis, which is more pertinent to cybersecurity, to understand how the malware work and what vulnerabilities are being exploited in order to be able to patch and counter the use of these. A future project where some diverse types of malwares are analyzed, handled and properly reverse engineered would be ideal, especially to highlight why it's important for cybersecurity.

Lastly one more future project which could be develop involves legacy code. As previously discussed, legacy code is old code that needs to be change and is difficult to understand. With reverse engineering you could highlight how said code could be examined in order to determine what its purpose is and then create a more up to date code in order to substitute the legacy code with the newer process. It would be interesting to see older programming languages be reverse engineer into newer languages using this technique

REFERENCES

[1] R. Singh, "A review of reverse engineering theories and tools," *International Journal of Engineering Science Invention*, vol. 2, no. 1, Jan. 2013 [Online]. Available: https://idc-online.com/technical_references/pdfs/mechanical_engineering/A%20Review%20of.pdf

[2] S. Megira, A. R. Pangesti, and F. W. Wibowo, "Malware analysis and detection using reverse engineering technique," *IOP Conf. Series: Journal of Physics: Conf. Series: Conference Series*, Mar. 12, 2019 [Online]. Available: https://doi.org/10.1088/1742-6596/1140/1/012042

[3] Sourceware.org, "GDB: The GNU project debugger." Accessed May 23, 2022 [Online]. Available: https://www.sourceware.org/gdb/

[4] Tutorials Point, "Assembly - Introduction." Accessed May 23, 2022 [Online]. Available: https://www.tutorialspoint.com/assembly_programming/assembly_introduction.htm

[5] Virtual Box, "Homepage." Accessed May 23, 2022 [Online]. Available: https://www.virtualbox.org/

[6] g0t mi1k, "What is Kali Linux?", Kali, Mar. 30, 2022 [Online]. Available: https://www.kali.org/docs/introduction/what-is-kali-linux/

[7] MKesenheimer, "Forest," Crackmes, Jul. 17, 2021 [Online]. Available: https://crackmes.one/crackme/60f31f1d33c5d42814fb3381

[8] bueb810, "gugus the first," Crackmes, Jan. 20, 2022 [Online]. Available: https://crackmes.one/crackme/61e9983133c5d413767ca5ac

[9] NomanProdhan, "License Checker 0x02," Crackmes, Dec. 24, 2021 [Online]. Available: https://crackmes.one/crackme/61c62bde33c5d413767ca0a0

[10] eventhorizon02, "super_easy," Crackmes, Aug. 19, 2021 [Online]. Available: https://crackmes.one/crackme/611e9bfb33c5d45db85dc2d7

[11] SilentWraith, "lockcode," Crackmes, Dec. 16. 2020 [Online]. Available: https://crackmes.one/crackme/5fda4fa433c5d41f64dee37b

[12] ezman, "easy keyg3nme," Crackmes, Oct 13, 2019 [Online]. Available: https://crackmes.one/crackme/5da31ebc33c5d46f00e2c661