Part 2 of this project required a lot of trial and error, patience, and Googling in order to find the passphrase for each program. I initially tried to take the same approach as we did in the lab to set a break and examine its functions and registers, but quickly discovered that this was only a helpful approach for the 1st program and that I would need to learn how to more effectively use the gdb debugger.

Program: nll21\_1, passphrase: “gpZNSAOBClRcQdJNyQlBoZh”

The general approach that we took in lab worked well for the first executable. I tackled the problem by opening the executable in gdb and setting a break point in main. I read through the contents of main and while I didn’t notice a strcmp function call like in the lab, I did see an x86 instruction cmpsb. Cmpsb it turns out is a string comparison instruction and seemed like it would be a potential place to check for a passphrase.[[1]](#footnote-1) I set a second break point at the address of the instruction call and examined both of the registers that this instruction took as arguments ($edi and $esi). The x/s command revealed that $edi held the pointer to the string that I initially input, and $esi held the pointer to a string that I had not input. I thought this second string may be the passphrase, so I copied it and re-ran the program from the beginning. Sure enough, it was the correct passphrase and I unlocked the program. I decided to see if my mystrings program was working properly so I tested it with nll21\_1. The program showed the exact string, among many others, and confirmed my suspicion that this passphrase was hardcoded as a string literal in the program.

Program: nll21\_2, passphrase: a palindrome of a length of at least 11.

I initially tried this same approach with the second executable as I did with the first. Just as with nll21\_1 there were no calls to strcmp but I also didn’t see a cmpsb instruction either. This indicated that the comparison probably happened in one of the three function calls (c, p, or s) that I didn’t recognize. I started by examining what function c did by setting 2 break points, one before and one after the function call. At each breakpoint I examined the 6 main registers ($eax, $ebx, $ecx, $edx, $esi, $edi) and discovered the following:

Function c: removes the ‘\n’ at the end of the user entered string. It returns 2 values: $eax is the position of the ‘\n’ in the original string, and $ebx is the modified string with the ‘\n’ removed.

Since function c was not the comparison function, I moved on to function p and set a break point after p to see what it returned. Examination of the registers at this break point revealed that p returned one value in $eax, presumably true (1) or false (0) based on the comparison instruction (cmp $0x1, %eax). Since the return value didn’t provide much insight into what the function did, I set another break point at function p (b p) to look into the function and reran the program from the beginning.

After several tries with different input strings I deduced that function p would only return true if the string was made of a palindrome. It took several variations on the input (e.g. aaaa and abbbbba) to figure this out. Since function p calls function s, I also found out that function s returns the number of characters in the input string in $eax.

Continuing to read through the main function I noticed that after main called function s, it compared if its return value (the string length) with the number 10. If the result of the comparison was less than or equal to 10, it would jump to a function call to puts, which presumably was the print statement for an incorrect string. I reran the program with a string of 11 a’s and it unlocked the program.

Program: nll21\_3, passphrase: a string of 16 characters in length (potentially including repeated ‘\n’ at the end) that must contain 2 of the following characters with duplicates allowed {‘0’, ‘4’, ‘9’, ‘c’, ‘s’}.

The last program took the most time to crack due to it being a stripped executable with no debugging information. Seeing as it didn’t contain a main function to break to, I tried breaking to a few of the other functions that I could recognize (found using the “info functions” command). Without any success and too many functions to try though, I quickly abandoned that approach.

I turned next to the online textbook (particularly the “debugging under gdb” section).[[2]](#footnote-2) This gave me a very helpful introduction to some of the other features of the debugger that I wasn’t quite familiar with. In particular the nexti (ni), info, print, examine (x), disassemble (disas), and object dump (objdump) commands proved to be indispensable in finding a solution.

While I had the tools I still couldn’t think of a good approach and turned to Google to see what online resources were out there for tackling this type of problem. After several sites and blog posts, I finally stumbled across one post on a Stack Overflow discussion that proved the most fruitful. The initial post in the discussion was about debugging stripped applications which was what I was working with. The reply with the highest votes by a Dr. Beco outlined an approach for where and how to choose breakpoints when there’s no debugging information.[[3]](#footnote-3) I used this post as a rough guide for my own program.

I started by using the info command (info files) in order to see the section boundaries. I set my first breakpoint at the end of the initialization (.ini) section and stepped through the text (.text) section instruction-by-instruction using nexti (ni) until a point in the text where I was prompted for input. Next, I opened a second terminal window and did an object dump (objdump nll21\_3) to see the entire contents of the text section and zero in on the instruction pointer.

Looking at this sub-section closely showed that it was looping and counting up to 16. Inside the loop, the program would get a character (from the stack) and store it temporarily in $eax. Next, it checked that character against several values (e.g. cmp $0x39,%eax), and depending on the value of character, it would either increment the main counter (also located in the stack at -0x10(%ebp)), or increment a secondary counter (located in the stack at -0xc(%ebp)).

I figured this had to be the password matching algorithm and carefully stepped through this section to see exactly what it and when it was checking. Using several test strings I deduced that the program was seeing if the string contained 2 characters from the set {‘0’, ‘4’, ‘9’, ‘c’, ‘s’} and was 16 characters long. After the 16th character, the loop stopped, and checked whether 2 characters from the previously mentioned set were in the string. This meant that at minimum a valid string would contain 2 characters from the required set, plus 14 ‘\n’ characters. The program rejects any string with more than 2 of the reserved characters (e.g. “cs0449cs0449cs04” would reject) and will not terminate unless 16 characters are entered (this could be 16 ‘\n’ characters).

In conclusion, this project helped me to appreciate how hard debugging can be and how to more effectively use the tools provided by GNU debugger. It also gave me insight into how the compiler actually translates the C code into assembly, and what some of the more advanced features of the compiler do to optimize code for the machine.

# Works Cited

Beco, D. (2011, April 2). *How to disassemble the main function of a stripped application?* (D. Beco, Editor) Retrieved June 27, 2018, from Stack Overflow: https://stackoverflow.com/questions/5475790/how-to-disassemble-the-main-function-of-a-stripped-application/5520318#5520318

Intel Corporation. (2016, Septmber). *Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z.* Retrieved June 27, 2018, from https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf

Misurda, J. (2017, July 27). *CS 0449: Introduction to Systems Software.* Retrieved June 27, 2018, from CS 0449 (2187): https://people.cs.pitt.edu/~jmisurda/teaching/cs449/cs449\_latest.pdf

1. (Intel Corporation, 2016) [↑](#footnote-ref-1)
2. (Misurda, 2017) [↑](#footnote-ref-2)
3. (Beco, 2011) [↑](#footnote-ref-3)