Jack Anderson
jja54
Jarrett Billingsley
CS 449

Project 2 Writeup

Executable 1:

Password: **wwpAiUiaEXfpYcoLLTYgSaTOzq**

For the first executable I decided to use gdb straight away and called “disas main” to get a dissasembly of the main function. In the dump I noticed that there were calls to fgets, a function called chomp, printf, and puts. Each of these were accompianed by 7 digit hex address. Initially I thought to use the x/s function in gdb that we discussed in lecture on the addresses next to these functions, but only got what seemed like garbage in a specific format. I assume that x/s was attempting to present a string representation of the addresses where those functions are stored. Moving on I noticed a line “mov esi, 0x80b388c” after the chomp function and remembered from class that this address had to be an actual string. Upon x/s-ing the address, I was returned the line “0x80b388c <__dso_handle+4>: "wwpAiUiaEXfpYcoLLTYgSaTOzq"” and realized that this must be the password for executable 1.

Executable 2:

Password: Your current thoth IP address, e.g **150.212.207.177**

Cracking executable 2 proved to be a much harder task to handle. Initially I started the same way as executable 1, disassembling main and finding a few interesting things. The first thing I noticed straight away was that the function “getenv” was called and knew immediately that this had to play a major role in the generation of the password. Looking it up I found that it is generally used to return a string of either the current environment’s path, home, or root. Further analysis of the disassembly showed calls to functions “sc”, “u”, “s”, “c”, “strchr”, and “strncmp”. It was here that I decided to set breakpoints for these various functions and examine them closely. Coming out of u I noticed that we ended up with the string “SSH_CLIENT” in the esi register (I should briefly note that it was originally ssh_client, but the s function called within u capitalized each

letter). Following this was the call to getenv which I assume "SSH_CLIENT" was used as an argument. The function returned what appeared to be an IP address with some extra bits tacked on at the end separated by a space. This was essentially the password being stored in memory, but at first I didn't know that yet. To be more brief, the program continued with at least 2 checks of this IP to make sure that the program was in fact ran on thoth, else a function "error_out" would be called which contained a pretty humorous string telling us to use thoth for the project. Eventually strchr is called with the IP string passed in and the char to search for "_". Looking up the function let me understand that it searches a string for the first occurance of the provided char and returns everything after it (including the provided char). Following a helped function "c" which cut off the newline character at the end of the entered password, this shorter "extra-bits" return was eventually subtracted off the full IP string to get a value of the correct number of chars to compare the IP string and the user entered password with (15) in the function strncmp, which is different than the more standardly understood strcmp.
Here's a link to a pastebin of my commented x86 code in case anyone wanted to explore my thought process on this one: https://pastebin.com/DDFvJg4h

Executable 3:

Password: Any 16 character string, as long as there are exactly 9 brackets within it:

e.g **[][][][][]aaaaaaa**, **((abCf)){[}}}BCa**, **({[456{)[7895]})**

This executable took a few days to work through but was a lot more rewarding and gratifying to understand as I worked through it line by line. My first attempt here was to just run mystrings on it, which came up with a string "@Z(ad7uKxc" which certainly looks like a password, however this did not work and I quickly abandoned digging around in mystrings land for any useable info. This executable happens to be stripped, so you can't simple place breakpoints at function names or disasemble main, etc. Instead, I used "objdump -M intel -d jja54_3" to get a intel-syntax objdump of all the commands run during the program. With this, I managed to find calls to getchar and a definite loop structure. So I decided to copy the x86 instructions from the top of a function (probably main), marked by pushes, a subtraction of the esp and the allocation of two local variables. I pasted this code (probably less than 50 lines) into notepad++ and started to

trace it, commenting on each line to help me understand what was going on. I discovered that the return of getchar was being put into a byte pointer, and began to understand the address calculations going on to store them. To keep things brief, running through the code I found checks against my input and specific hex values (e.g 0x5b, 0x7d, etc.) which correlate to bracket characters in the ASCII table. I then noticed that every time one of these characters was found, a local variable that wasn't the loop counter was incremented. At the end of the loop (which ran for 16 entered chars) it checked to see if this extra local variable was exactly 9, and if so, printed the congratulations string. Therefore, I understood that it was possible to enter any 16-char string as long as there were exactly 9 of the bracket characters "[,], (,), {, }".

Here's a link to a pastebin of my commented x86 code in case anyone wanted to explore my thought process on this one: https://pastebin.com/8svp6HHL