Project 2 Write-up

Mathew Varughese
mav120@pitt.edu

Program 1

Procedure

My first thought was to run the executable through the strings program. When I did this, I noticed there were a lot of strings. Two strings that looked intriguing to me was a string that contained about 30 Qs followed by a j, and the string `btvsTdsWbXxFDvUlHsRfDPE`. The second string was especially interesting, because it was right above near the “`Sorry! Not correct!`", "`Congratulations!`", and "`Unlocked with passphrase %s`" strings. I then ran the program through `gdb` and set a breakpoint on main. One line of assembly I saw was `mov eax,ds:0x80d691c`. I inspected the value `0x80d691c`, using the `x/s` command and noticed that it was `btvsTdsWbXxFDvUlHsRfDPE`. I then decided to try this as the password, and it worked. To ensure that I was not missing anything, I decided to step through the assembly with gdb. Inside the `chomp` function, I noticed some jump instructions and a `repnz scas al`. After googling this instruction, I learned that it scans through the string. After realizing this and looking at some of the assembly afterwards, I was fairly confident there was only one password, and chomp was doing some sort of `strcmp`.

Solution

btvsTdsWbXxFDvUlHsRfDPE

Post Mortem/Notes

- By the large file size and the fact that I was able to break on library functions in gdb, I assume that this executable is likely statically linked with an intact symbol table
- I am still unsure why the string “QQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQj” was found within the executable.

Program 2

Procedure

I decided to run the executable through the strings programs, since it worked so well for the last program. It did not help. Thus, I chose to run `gdb` and `disas`. I became frustrated trying to figure out what some assembly instructions did. However, I noticed a call to `getenv` with some calls to `strcmp`s after that. I set a breakpoint after the call to `getenv` and ran `x/s $eax`. That returned `mav120` which looked very promising. I tested this as the password and it worked! I searched for documentation of the `getenv` function and found that it returns a value of an environment variable, specified as a parameter. I ran a `printenv` command on thoth to see the environment variables. At first, I thought the program took the `HOME` directory path and took the characters off the last backslash. However, then I saw the `LOGNAME` variable was `mav120`. I changed this to `mat`, but the password did not change. I then saw the `USER` variable was also `mav120`. I changed *that* to `mat`, and viola, the password to the executable changed to `mat`.

I was still curious what the "u" function did. I set some breakpoints in that function, and I found that it called another function, "s". I was getting lost in the assembly, but I concluded that it was a loop of some sort that capitalized a string. I ran `x/s $edx` and got back "user". Through each iteration of the loop, the next character in that string became capitalized. In other words, the first-time `u` called `s`, the string became User, then USer the second time, and USER the fourth time.

Solution

The value of the USER environment variable. In other words, getenv("USER").

For me – mav120

Post Mortem/Notes

- I assume that the string "user" is stored somewhere, and something similar to getenv(u("user")) is being called.
- The symbol table is still in it

- I noticed "@plt" after some functions, and some research led me to believe this file is dynamically linked

Program 3

Procedure

This was quite challenging, to say the least. Running the executable through strings gave me almost no info, except for `getchar` and `&#k!, -x|?`. Neither of these worked as the password, and I could not even break on main with `gdb` because the symbol table is stripped. I also was confused why this executable had 4 different input lines at first. I was very lost. But, after doing some more experiments and looking up the documentation of `getchar`, I realized that this password accepts 16 characters. I then ran `objdump -d -Mintel mav120_3 > 3.asm` to try to look at the assembly. I thought to myself, "`Sorry! Not correct!`" should be located somewhere in the assembly, so I searched for `73 6f 72 72 79` ("Sorry" in hex) in the assembly with no success. I then realized this could be because I used the -d flag instead of -D. Still after doing this, I could not find the text. I decided to go another route and look for the main method. I saw `call 8048334 <__libc_start_main@plt>`, and some research led me to believe that it will jump to wherever the main method is loaded in memory after the program is run. I set a breakpoint on `0x8048370` since this was the first value in the .text segment. I then spent the next 5 hours carefully stepping through, looking at registers, setting breakpoints almost everywhere, and using the `x` command.

Little by little, pieces of the program became clear to me. I entered in "aaaaaaaaaaaaaaaa" as the password while debugging, and then when I found 97 (a's ASCII value) in the $eax register, I realized I was in a loop doing some comparisons on the user input. I then tried reasoning through the assembly.

Whenever I figured out what a piece of assembly did, I annotated my asm file produced by `objdump`. What helped me the most was finding the lines that contained `0x8048602` and `0x80485d4`, which were the "`Sorry`…" and "`Congratulations`…" texts respectively. I then looked at the code surrounding these, and I figured out that if `-0x10(%ebp)` is equal to 9, then the password is considered correct. I then tried understanding the code before this, which was a numerous amount of jump instructions. I reasoned the C code that produced this would be few if/else statements. I tried to hand translate the assembly, and using an ASCII code chart, and I got something along the lines of:

```
if(c == >) {
     // > : 3e
     // je 0x8048495
}
if(c > >) {
     // > : 3e
     // jg 0x8048486
}
if(c == &) {
     // & : 26
     // je 0x8048495
}
// etc . . .
if(c != ^) {
     //

}
```

To figure out what happened inside of the if statements, I figured I could look at the instructions at each of the jumps. I wished to try out some passwords first. I knew there was some sort of counter that should equal 9, so I tried “<<<<<<<<<aaaaaaa", and I was astonished that it worked. I then tried different combinations of passwords containing the characters >, &, <, |, ~, and ^ and they also worked.

I realized after compiling some of my own C code and looking at the produced assembly, that the code I hand translated is doing a logical “or”, checking if the current character is one of the following characters: >&<|~^.

Solution

- The password is 16 characters long
- 9 of these characters must be of the following: >&<|~^ (repeats allowed obviously)
- The other 7 must be something other than those characters

For example, all of the following are valid passwords:

- <<<<<<<<<aaaaaaa,
- >>>>>>>>>aaaaaaa
- >&<|~^>>>abcdefg
- btvsTds>>>>>>>>>
- mav120>>>>>>>>>

Post Mortem/Notes

- This was the most painful/challenging/rewarding CS assignment I've ever done
- The symbol table is most definitely stripped
- I assume some dynamic linking or loading is being done
- Running the x command saves the parameters for the next time it is run. For example, if you run "x/s", and then "x", it will remember those parameters