Name: Brian Jon Stout
Date: 27MAY2017
Current Module: x86 Assembly Programming
Project Name: bomb

Project Goals

This goal of this project is to analyze the assembly instructions generated from a pre-existing binary to look for methods to bypass tests in the program so the user can eventually "disarm" the bomb. The program has 8 stages (possible a secret stage?) each requesting input from fgets and passing it to a function whose source code isn't available to the user.

Considerations

- The programmer should be able to "disarm" the bomb using the original binary and no tools, though they're needed to figure out the correct input to advance
- The program may use common security practices to prevent debuggers and other such tools from working on the binary

Stage 1

The first password was revealed with the strings commands. The only phrase that appeared to be a plain text password was Swordfish.

Stage 2

Running strace or ltrace reveals a ptrace check in order to avoid reverse engineering. Doing an object dump reveals that in stage two there's a getenv function call. Strings reveals that there is a string for USER, which is a valid env variable which, when called with getenv, gives the user's ID. On my account the user id is 'bstout.'

Stage 3

Attempting to gdb to the pid for bomb results in a error because uid does not match the uid of the target and is already traced by a different process. We can get around this by setting a break at the ptrace call and stepping till there's a cmp that results in the seg fault. The compare is the result of ptrace stored at *rax*, setting *rax* to the value of 1 gets by the seg fault. ***_start+41*** is the address for the cmp.

We're going to get the logic for each stage by setting a breakpoint using the break *stage_1 command and so on. Looking through we see a strncat call (no risk of code injection) which appends what appears to be bstout (user) stored at a global address. We can determine the value by breaking at ***stage_3+37*** and printing out what gets stored in *rsi*. It appends this string

to the input passed into the stage_3 function from main. While the return address is 0'd out, the start of the pointer for "10" (input for stage 3) is stored in *rbp* and can be printed out for the result "10bstout."

The next interesting instruction is ***repnz scas***, which according to https://stackoverflow.com/questions/26783797/repnz-scas-assembly-instruction-specifics is the equivalent of strlen on our new string. Stepping through the loop it seems to be moving up the address till hits a null byte and finally stepping out of the loop. You could use the address difference to get a character count. At the end of the loop with an 8 letter string *rcx* is set to -10. Not *rcx* command flips the signedness to 9 so it's every letter +1 for the length of the string possibly? The value of *rcx* is stored to *rbx* using lea -1 giving the exact length of the string.

sscanf eventually gets called and *"%u"* is passed in as the format. Your initial input string is passed in as the string, so it just outputs what you did. It being an unsigned int makes it dangerous to int overflows. Inputting a negative number for stage 3 breaks the program and the overflow causes enough issues to pass the *je* check. There's a multiply call so having a sufficiently large input should cause an overflow too.

Stage 4

Running through the instructions for stage 4 shows another sscanf. The format string is *"%u %u %u %u %u"* so the input needs to be 5 numbers.

At the start of the program the instructions store the address to the global pointer "bstout." Some weird stuff is done and it stores the ascii value of the first letter to *ebx* using movzx (I don't really understand this beyond movzx is supposed to convert data between registry sizes.)

After the call to sscanf there's a quick check to make sure sscanf outputted 5 values. The value from b (98) is moved into *eax* and *edx* and it's compared together. The program then goes on to check if that value is less than the first decimal outputted by sscanf. If it is, it'll jump to a section of the program that adds the value from the user's first letter, to the user's first inputted decimal, keeping it stored in *rdx*. The program then jumps back to the same *ecx*, *edx* comparison, seeing if the resulting sum is less than the next number, meaning the first number has to at least be one more than the letter, and each successive number has to be at least double the previous one. 99 198 396 792 1584. This appears to be the case from stepping through the program and looking at the registers when cmps happen.

After passing this test for each of the five inputted numbers, the final value is divided by the number of characters in the user's id (6 in my case.) If there isn't any remainder, the program will set *rax* to 0 and return *rax*. The final number must be twice the amount of the previous number, and not divisible by 6.

Final solution is 99 198 396 792 1585 and works fine, though 200, 400, 800, 1600, 3201 does not so some of the logic isn't what it appears to be. Stepping through the program reveals the 98 number is important to the addition logic. It works all the way up to the final portion and some checks fail for reasons I can't find.

Stage 5

Analyzing the dump for stage_5 reveals another call to scanf, the format string is *"%c %d %c"* meaning it expects a character, a number, and then another character. The cmp immediately after scanf reveals if three things aren't inputted it'll error out as expected.

Judging from the other stages, the program will use my userID, or part of it, and looking ahead shows the same repnz scas loop to find the number of characters in the user id, so I'm going to guess the string is *"b 6 b"* from my limited amount of information.

Stepping through the program does show 'bstout' gets stored into *rdx*. Both *r8* and *esi* get the character 'b' stored into them. An null byte/empty string gets stored into rdi, which somehow modifies rax to contain the address for "bstout." That address is stored into *r9*. The value 98 'b' gets stored in *edx*.

A bunch of weird seemingly pointless checks occur and then the program jumps to ***QWORD PTR [rcx*+0x4014a8]*** which brings us to address ***0x400ed2*** or ***<stage_5+146>***.

Looking ahead at the instructions it appears the pass condition is for the value stored at ***rps+0x4*** to be 0, and values stored in *edx* and *esi* subtract from that in a loop based off the length of the username.

In attempt to follow what's happening in the loop, random but different values will be used to track how the data is used. *"c 11 d"*.

> ***rsp+0x2*** is 'c' or 99
> ***rsp+0x3*** is 'd' or 100
> and ***rsp+0x4*** is 11, and can be shown by using ***x/dh $rsp+0x4***

Running through the loop it's clear that 11 is a much too small number. Instead of trying to figure out this crazy jump mess, the plan is to just use the same values for the letters (which on occasion, randomly seem to add to edx before it subtracts it from ***$rsp+0x4***.) so that the same subtraction happens. At the end of the loop grab ***rsp+0x4***. Using a 2000, the end result was 744, so doing *"a 1256 a"* should theoretically work since no logic is done to ***$rsp+0x4*** to modify what's subtracted from it (that would be evil.) Sure enough *"a 1256 a"* works as an answer.

Stage 6

Just like all the other stages there's a sscanf call to parse user data. The format string translates to *"%d %c %d"*. I Decided to randomly input the values *"97 b 99"*

Looking ahead there seems to be some mystical function call, so we're going to have to step into that. The first and second arguments for this wonky mysterious call is 97 and 99 respectively. *rdx* is 0 (possible third argument,) and *rcx* is 40 (possible fourth argument.)

The first thing noticed about the function call is that it calls itself inside of the function meaning recursion. That's not nice. Stepping into the function reveals that you have to make sure edi and esi are not the same values, or it will return with the first letter of the user id

Testing a bunch of different combinations to see the flow of the recursive functions usually results in a SIGSEGV. After spending what feels like several hours entering *'si'* into gdb, only thing that doesn't result in a segfault is having the first argument second argument of the function call be equivalent.

Looking after the recursive call it looks like the success condition for this stage is to

1. and *eax* with *edx*
2. add *ebp* to *ebx*
3. cmp equivalency for *eax* to *ebx*

On a early return for the recursive function, eax is the first letter of the user id, 'b' in my case. *Edx* is the middle letter you provided to sscanf. Randomly attempting 'd' as the middle letter results in the number 96 when anded with *eax*.

When stepping into ***stage_6+81*** (add *ebx*, *ebp*) it was discovered that *ebx* and *ebp* were the same number, so restarting the program and inputting '48 d 48' as the input successfully passed the stage.

Stage 7:

Looking at the instructions, the only way to return from this stage with a nonzero *rax/eax* registry is to jump before *stage_7+90* and after *stage_7+67* and have 0 data in your *al* registry (8 bit version of *rax/eax*.)

The easiest way to do this is jump at ***stage_7+17*** which tests to see if *al* is 0, which is what we want. Right before this test is ***movzx eax, BYTE PTR [rdi]*** which stuffs the argument we passed to the stage_7 function into eax. The argument is return from fgets. We can get fgets to return a 0 if we don't input anything at all to stdin (by just hitting enter.)

Final Results

1. swordfish
2. bstout
3. -1
4. 99 198 396 792 1585
5. a 1256 a
6. 48 d 48
7. (just hit enter at the start)
8. could not complete

Conclusion

This project was definitely GDB: the game. I got immensely familiar with using GDB to debug assembly programs, and some of that knowledge will definitely carry over to when I actually have a c source file (might be too easy.) It also got me use to seeing some of the assembler quirks such as the xors, tests, and the assembly version of strlen.

I thought the actual assembly functions for each stage were a little odd, and it might have benefited to see common group of assembly instructions often found in real world programs.

At the end of it all people are at the very least familiar with how the stack works, and how values are stored on it.