Name: Jesse Schoenwald-Oberbeck
Date: 27 May 2017
Current Module: Assembly
Project Name: Bomb

Project Goals:
We were presented with a binary called bomb, which runs in stages. Our task is to diffuse the bomb, or to get as close to said goal as possible.

As a note: all references to gdb and registers will be lower case, as this is what I observed in the course materials.

Considerations:
Only a compiled binary and a driver file were supplied. Without source code to look at, installed analysis tools are the only option. Tools used will be strings, objdump, and gdb.

The program is run with zero arguments, taking all input via prompts and the user's environment. When run with GDB, there is a cmp after a call to "ptrace" which causes a jump to a program exit. Interfering with this comparison by setting rax to a different value (I tended to use 5 for no particular reason) prevents this jump from taking place.

Stage 1:

Used "strings" on compiled binary. A few strings stood out, including indication of possible hidden stages, some of the library functions used, and "swordfish," which turned out to be the stage 1 answer/password.

Answer: swordfish

Stage 2:

The stage2 function calls getenv, and USER is present in strings. Entered my username, and progressed.

Answer: jschoenw

Stage 3:

Entered 12345. Moves the user's input string to rbp. Moves username into rsi. Concatenates rsi and rdi, up to size 128. So... 12345jschoenw. repnz decrements rcx. not rcx turns rcx into its 2s complement, 14. rcx -1 is loaded into rbx. rdx is made null in a needlessly weird way. An arbitrary value, 401493 is loaded into esi. sscanf is called, and passed 12345jschoenw, and a format string of "%u". The format string will only take an unsigned integer. rax and rdi now hold

12345. Division of rax by 13 occurs, yeilding 949. Rax is shifted right by 2, yeilding 237. This is multiplied by aaaaaaab. rbx is compared to rdx, followed by setb (set byte if below).

Answer: put in a big number and win. 12345.

Stage 4:

(A lot of trial and error here was not documented, largely due to being too focused on my own failure and frustration. Initially I was assuming every letter in the username was relevent to the answer, and trying to interpret the code as such. Things went much better once I set this idea aside.) Avoided the first jump (jne) by setting eax to 5. This seems to be the number of arguments expected (also the format string is "%u %u %u %u %u"). ebx is set to the first letter of the previously saved off user name string, so "j". the numeric value of j is compared to the first argument, continuing if the argument is greater. A loop is started that checks if the current argument is greater than the sum of all arguments thus far, until the last argument. The last argument, the sum of all previous has to be evenly divisible by the length of the user's name.

Answer: 107 214 428 856 1713

Stage 5:

The next format string is "%c %d %c", so I'll start by assuming they stay in order, and use this to format my input. "j 100 j" is my first guess. according to the arguments passed to sscanf, my guess is correct. rdx is pointed to the username. This is why I guessed j. off to a good start. rcx is set to 9 after repnz via not. one us subtracted, so a jump if equal (je) is skipped. moves username pointer to rax. A single byte of rax is moved to edx. The value at rcx (9) is compared to 0x19, and a jump if above is skipped. A jump occurs based on the number in rcx. The pointer to username is moved to the next character. schoenw is compared to rdi, which is... null. Jump to stage_5110 is executed. Another jump based on rcx occurs. j is added to s. edx is subtracted from a value past the stack pointer. A jump to stage_5198 is done. rax is moved over again. "choenw". Jump back to 110
"QT_ACCESSIBILITY=1"... wat?
The stage loops over each letter in the username, jumping to a location in the function based on said letter, with a manner of one-line arithmetic jump table.. In each letter based section of code, at which point varied maths happen...
I've decided to try skipping the middle portion, as it is a ton to track, and see how varying input affects stage_5207, which I believe to be the final compare. j 500 s ends at -557 j 1000 s ends at -57 j 1057 s ends at 0 This causes it to successfully pass.

Answer: j 1057 s

Stage 6:

The next format string is "%d %c %d" so I'll guess that's the format for this one. So... I got lucky with a guess. My first guess of "100 j 100" was unsuccessful, but my second guess of "53 j 53" worked. What are the odds? Each number is half of the decimal value of "J". As usual, sscanf breaks up the function. The above format string did end up being used. A mysterious function is called, which decrements the first two arguments to zero. The values actually stored on the stack are unchanged. "j" is moved into edx, then anded with eax, which contains the same value. ebp, which contains the number passed, 53, is added to ebx, containing the same number. The result is equal to "j." sete is called. done.

Answer: 53 j 53

Stage 7:

I first attempted it with a number. 42. Then a string "Hello World." It looks as though the user entry isn't actually used for anything. Negative number maybe? No. Perhaps entering a null character (latex won't print slash zero) No. Nothing. Nothing works. Which is to say, supplying the function with an argument that is empty, is the answer. Null. So, RDI is null. The second jump is hit. rdx is equal to rdx. Shocking. It compares a rip relative address to rcx, which is 0. It seems the gibberish at the relative address is equivalent to zero.

Answer: NULL

Potential Pitfalls:
There are many. Following the code into system or function calls that take forever to get out of, misinterpreting a value (char vs int, etc) or focusing too hard on portions of the code that have no effect on outcome, such as the stack cookie/canary portions, or the repeated chunk that serves as strlen.

Conclusion:
The project was seemingly equal parts a test of assembly literacy, and ability to effectively use tools. Mainly gdb. I had a good deal of difficulty interpreting the actions of the code into something more understandable, such as trying to decompile it back to C manually. Stage 4 took me a great deal of time to finally figure out because I was having a hard time following the loop it goes through. Ultimately I ran out of time, and relied on luck to get a couple of them done within the time frame given (stages 6 and 7 were largely luck, for me I think). Over all I would say it was a good learning experience, though very time consuming. I spent more time miserable than triumphant by far.