Name: Stephen Paradis
Date: May 27 2016
Current Module: Intel Assembly using C
Project Name: Bomb

Bomb Program

Stage 1:

For this stage I began with calling strings on the “bombs” executable. From this I found after the plain text “stage[#]:” the plaintext “swordfish” so I ran the program to test against that string and found it to be the correct password. When I looked at it through objdump I found the <stage_1> label and began looking through it. I saw the strcmp call so I realized that strcmp was comparing what I entered against the variable “swordfish”.

My answer: swordfish

Stage 2:

For stage 2 I checked strings again and saw the word USER so I tried that first, when it did not work I went into objdump and found the <stage_2> label. Inside the label I saw a <getenv> call at 0x400eb8. Underneath that I also saw the comment <username> so I tested my username which worked.

My answer: sparadis

Stage 3:

For stage 3 I checked strings which gave me no real value so I went into objdump and found the <stage_3> label. As I ran through it I found a strlen and strncat call. Based on a check Wilding made, I tried ‘8675309’ which worked. Through later testing I found that the lower limit of the number entered could change based on the length of the username, which was 8 characters for me, which led to the lowest number i can enter being 1872.

Later after extensively talking with SSG Torres I found out that the user input is concatenated with the username. Strlen then is is called on the concatenated string, the user input number is then divided by that number. The result of that is divided by 4 and the result of that is multiplied by 0xaaaaaaab. The returning value of that must overflow the data register, then that number is divided by 2. The resulting number from that must be greater than the length of the concatenated string.

My answer: 1872

Stage 4:

For this stage I moved onto the <stage_4> label and went straight into gdb. Through extensive testing Follenbee and I found that the strcmp is comparing against “%u %u %u %u

%u” however it kept failing before getting to the math portion. Through further testing of gdb I found these specific lines that sprung out at me:

- 0x400fb0 - we grabbed the username
- 0x400fba - it grabbed out the first character of my username and moved it into eax
- 0x40100c - compares to make sure that five things were matched to sscanf

After sscanf is matched we start looping through first checking if the first number is greater than the ascii decimal value of the first letter of your username. If is is greater than, the fail check does not get hit and we continue by adding based on these lines:

- 0x401033 - which moves the next input value into eax
- 0x401037 - which adds the decimal value of the first ascii character to the current input value you are on in the loop. So for example 1+<address of input>

So for me I entered 116 310 620 1240 5047, based on the code it would add 115(ascii value for s) to 116 and stored that into the variable at [rbp-0x2c] and then eax moves onto the next argument. If all checks don’t lead to failure then we move onto the final part which either leads to success in the end or the bomb exploding. On the line:

- 0x40104f - the final value of all input is stored into the counter register
- 0x401052-0x40106f - arithmetic operations which if the correct numbers were entered will set eax to 0

The arithmetic operations only really check if the decimal value of the ascii character plus the sum of the 5 input values are divisible by 7 and if it is it returns successful.

My answer: 116 310 620 1240 5047

Stage 5:

For this stage I first went to strings and saw the string “%c %d %c”, from there I went to the <stage_5> label in objdump and began reading through. It was slightly helpful but it did not tell or show me what was happening so I moved into gdb. For the first attempt I put in “s 2080 w” and began watching as gdb stepped through. My assumption that the format had to be in “%c %d %c” format was correct because sscanf returned 3 which means it matched 3 things. I then watched it move my username into rax, and then rdi before calling strlen to get the length of my username. It the jumps to this line:

- 0x40121b - which moves the counter variable into rax
- 0x40121f - this compares this value to the length of my username(found from strlen)

This showed me that this was just looping through and for each letter in the username the counter variable would be incremented. It then jumped again to these lines:

- 0x401103 - moves the username into the data register
- 0x40110a - moves the current number count into rax
- 0x40110e - the it adds the username + the count variable to get the next character in the username

The above lines are what moves from the current character in the username to the next character in the username. It then moves the value of the current character into the accumulator and makes a check to see if it is lower case ascii. From there it will jump to whichever case is found based on this line:

- 0x401125 - It moves into rax the memory address of the case for the switch statement

From here it goes to whichever case statement applies and based on my username it went to these cases:

- Case a:
    - Add username[position] and first %c argument then subtract that from the given %d
- Case b-d:
    - Subtract username[position] from the given %d, does this twice
- Case i:
    - Add username[position] and first %c argument then subtract that from the given %d
- Case o-t:
    - Add username[position] and second %c argument then subtract from the given %d

Finally once the loop has gone through each character in the username it tests the accumulator against itself to see if the given %d is 0, if it is the stage will be defused. One thing to note is that as long as the characters you enter are lower case ascii and the value of the %d matches so that it becomes 0 after the full loop, then multiple different answers are acceptable.

My answer: s 1766 w

Stage 6:

For this stage I moved to the <stage_6> label and began with gdb and objdump to place my break points. With the help of Iracane I found that the format string for stage 6 was “%d %c %d” and sscanf returned that it matched 3 things successfully. If the sscanf was successful it would jump down and move the first %d input into the edx register and the second %d input into the eax register. Then those 2 arguments are moved into edi and esi so that they can be given to the ___ function. Through trial and error I found that since the beginning letter of my is ‘s’ which has an odd value in decimal the ___ function would loop indefinitely. However if I picked an even number such as ‘r’ and 57 on either side which is half of the decimal value of ‘r’ then the ___ function was successful bringing back to stage_6. I found that it would grab the ‘s’ from my username and move it into eax on this line:

- 0x401384 - moves the variable at rbp-0x15(decimal value of ‘s’) into eax
- 0x401388 - moves al into eax effectively checking against the letter below ‘s’ since its ascii decimal value is even.

The main part that stood out in this code though was on these lines:

- 0x40138b - which does a logical and on ‘s’ and ‘r’
- 0x40138e - which moves one of the 2 %d values given into ecx
- 0x401391 - which moves the other %d value into edx
- 0x401394 - adds the 2 values so the value from the and can be compared against
- 0x401396 - compares and if the values are the same and if they are set the al byte

It move the values of the first and second %d into the ecx and edx register and adds them to see if those values add up and equal the decimal ascii value of the entered %c. Then the sum of

the previous addition and the %c entered are compared and if that repair returns 0(meaning the values were the same) the al byte is set and the stage is successfully defused.

My answer: 57 r 57

Stage 7:

For this stage I looked at the <stage_7> label in objdump, however for my first try I collaborated with Iracane who found out that by entering a carriage return stage 7 would successfully be defused. However in the aspect of learning how it did this I ran it in gdb to see what was happening inside the code. So as I walked through gdb I saw rax gets the address of <head> on lines:

- 0x4013b9 - moves the address of <head> into rax
- 0x4013c0 - moves rax into [rbp - 0x10](variable)

Then I saw an unconditional jump that lead to a comparison between 0 and [rbp - 0x10] (address of <head>). If that comparison fails to be equal then it continue down zeroing out rax on line:

- 0x401428 - Moves with zero extending a single byte into eax

The above line sets the al byte to 0. After testing to make sure al is set to 0 the length of the command line argument is moved into rax, in this case 0. It is then compared to see if [rbp-0x8], which was set to 0 at the beginning of the function, and rax are 0. If they are the al byte of the accumulator register is set and the function exits causing stage 7 to be successfully defused.

My answer: <carriage return>

Stage super duper secret:

As I was looking the label <stage_super_duper_secret> in objdump with Follensbee and Iracane we figured out that was the ascii representation of a string we found when looking at bomb with strings which was “Nice try, but such a stage does not exist.”