Proactive Computer Security
Assignment 3

Nikolaj Dybdahl Rathcke (rfq695)

May 12, 2016

In the report, I'll try to explain how I solved the phases. The file `src/assem.asm` contains commented assembly code. I have mainly tried to comment the loops in the code as these are the most important. They are commented in a sort of "low-level" way as it helped me to understand what went on in a higher level, which is what I will try to explain in this report. If there are tricks in the assembly I have tried to point it out in the comments (such as `shr eax, 1` is just `eax`/2). I managed to solve the first 6 phases.

Note: I omitted the small functions I made from the `src` folder, but a copy of the bomb, solution.txt and assem.asm (commented assembly from running `objdump -M intel -d bomb`) is found in the folder.

Phase 1: As the very first thing I ran the command `strings bomb`. Above the output "Phase 1 has been defused", I found what was the actual solution. Namely the string "This is the first flag".

Phase 2: For phase 2, looking at the output for `strings bomb`, I could see it would take six arguments, all decimal numbers. I then started looking at the assembly code for the program (`objdump -M intel -d bomb`). First thing I noticed is that it checks if we have 6 arguments (which you can also see from how many arguments it pushes onto the stack). The approach from here was to just follow the code. It pulls out the first argument and checks if it 9. From here you can see a loop. It initializes something to 1, does some calculations, and then check if it is less than or equal to 5, in which case it goes back up. Trying to write the loop in high level code, the loop looks like this (it does not run):

```
arr = [9]              # List of arguments, we are given 9
for i in 1..5
  eax = i
  edx = arr[eax]       # Get the next argument
  eax = i              # Set it again (bad compiler!)
  eax -= 1             # Subtract one
  eax = arr[eax]       # Get current arg
  eax = eax * i        # Multiply it with counter
  if eax == edx        # If the next arg is equal
    continue           # to previous arg * i, we dont explode
  else
    break
end
```

To which we can sort of guess the actual loop is something like:

```
arr = [9]
for i in 1..5
  arr[i] = arr[i-1]*i
end
```

Printing arr will give us the six arguments: 9, 9, 18, 54, 216, 1080.

Phase 3: Again looking at the assembly code, I could see it took 2 arguments. Same as before, I realized it check if the first argument is greater than 999 (we'd explode otherwise). Again we initialize some kind of a counter to 0. I could see some kind of loop, where we changed the first argument and checked if it was 1 after each iteration. I could tell it took out the last bit (`and` with $0x1$), and used this bit to do one of two things. If it was 1, we added the value in register eax to itself two times and added one ($3n + 1$). If it was 0, we bit shifted to the right once ($n/2$). Obviously, this is what is known as the Collatz conjecture. It came quickly that the point of the second argument was how many iterations of the loops we produced. An easy answer that was thus 1024 and 10 (we bitshift it 10 times before getting 1). The loop would be something like:

```
a = 1024
i = 0
while a != 1
  if (a % 2) == 0
    a /= 2
  else
    a = a*3+1
  end
  i += 1
end
```

This loop is obviously specific to where the first argument to 1024 and we can find i this way.

Phase 4: From the assembly code, I could tell we took 1 argument. We check if this argument is positive and from there we call the function `func4` with this argument. In `func4`, we check that the argument is not less than or equal to 2 (since we go out of the function and return 1 otherwise). It was quite easy to see that the function decreased the argument and called itself with a new argument three times. The argument was decreased every time. I felt this was relatively easy and figured it all called itself recursively (three times), so I wrote a quick function to calculate the number, but I couldn't match the number it checks up against in the assembly code for phase 4 (289329). Turns out my mistake (as I have already said) was that I returned 2 instead of 1 when the argument was ≤ 2. Such small mistakes can apparently take a long time to figure out. Now, the result made sense as it is the n'th Tribonacci number we are calculating. The function might look as simple as this:

```
func a
  | a > 2  = func(a-1) + func(a-2) + func(a-3)
  | a <= 2 = 1
```

where it turns out we were looking for the 22 Tribonacci number (yeah I know, I don't handle negative input, this is just quickly written code).

Phase 5: This phase and the next I solved with a lot help of `gdb`. Admittedly, I had a hard time figuring out what the code did from looking at it. The easy answer "I WILL NOT DEFUSE THE BOMB" did not work, but I could tell we were working with strings (all these byte pointers instead of word pointers). We would also check these bytes, so I figured it was some kind of encryption. This led me to write the entire alphabet as input and then using `gdb` I could see the ASCII values changing. As a true monkey, I figured out what all the letters were changed to and saw it started repeating itself after p. These 16 letters however were sufficient to make the string "I WILL NOT DEFUSE THE BOMB". Turns out my intuition was right and the string "epnehhpjimpcbdlkbpmfbpaiga" would be encrypted to "I WILL NOT DEFUSE THE BOMB". The commented assembly might not make much sense as I mainly used `gdb` to solve this task.

Phase 6: Again, I had a hard time reading the assembly, however, I noticed a few things. We take 6 arguments. Another things I noticed was that there was only one place where we would explode. Before this, there is a small loop, which loops from 0 to 5, where we check the arguments up against the counter. Another thing I noticed was in `func6` that we move the arguments around. Though I'm not entirely sure what is moved where, this led me to insert a break point in the loop. Since we check against a counter going from 0 to 5, I gave as input "0 1 2 3 4 5". From there I could tell that the first thing it pulled in was 2. This gave me the idea that all it did was permute the arguments. Truly so, it turns out that it was the permutation "2 0 1 5 4 3", would be permuted to "0 1 2 3 4 5" as it is pulled in and matched against the counter. Again, the assembly comments are a little lacking, but I have commented on the key observation I have made.