Defusing the Bomb - Writeup

Erik Steringer
April 2, 2015

This paper is a writeup for the Bomb challenge presented by Matt Briggs and Frank Poz on opensecuritytraining.info for their Introduction to Reverse Engineering Software curriculum.

1 Preparation

To begin the disassembly, we need to acquire a few pieces of software.

1. Microsoft Windows, running natively or in a virtual machine.
2. IDA, The Interactive Disassembler. It should be a version with a debugger.
3. The Bomb executable file we’re studying. That’s available at: http://opensecuritytraining.info/IntroductionToReverseEngineering.html along with the password needed to open the .zip with the executable.

2 Disassembly

We are given an executable file, which we unwisely run on our machine to see how it behaves. It opens a prompt and presents the following output:

Welcome to my fiendish little bomb. You have 6 phases with which to blow yourself up. Have a nice day!

Since we do not know what we can do from here to prevent the bomb’s detonation, we need to use IDA to begin disassembly. Loading the executable into IDA reveals that it’s a portable executable (Windows executable files). We kick off the disassembly and use the debugging symbols.

3 _main

There is a function labeled "_main" which shows us where execution begins. We open that function, and make sure it’s in Graph Mode to give us a proper
IDA was kind enough to label the argument-count’s location as ebp+argc. The program splits between two branches. If it doesn’t take the jump to loc_401017, it goes straight to the meat of the program. If it does take the jump, it checks for an argument. If it has two arguments (first is the program name, second turns out to be a filename) it calls a function named ”fopen” which again implies opening a file. If the function fopen returns a value besides zero, it jumps to the meat of the program. If it does return a zero, it prints an error message and calls the exit function.

Inside the meat of the _main function, we can see the strings first printed to the console that are pushed before printf is called. From there, we can see a function labeled as sub_401870 which is called. Opening the function, we can look around at the strings linked by the function which imply it reads either a file input or stdin. We can label sub_401870 as ”ReadInput”. Note that the register eax will point to where the string input can be found.

### 4 Phase One

After reading input, another function labeled as sub_401190 is called. Inside the function, a string pointer is pushed to the stack. The string pushed contains the text:

Public speaking is very easy.

The value labeled ebp+arg_0 is then transferred to the register eax then pushed onto the stack. This means two string pointers are pushed onto the stack before sub_401740 is called. We jump into this subroutine. sub_401740 calls two functions, labeled sub_401700. We should investigate that function while noting that the string pointers passed in are each passed as an argument when calling sub_401700. This means sub_401700 takes a string for input.

Inside sub_401700 there are two variables. The variable ebp+var_8 is a byte pointer (or char pointer as we come to learn). The variable var_4 starts with a zero, but will be incremented as the method progresses and loops. The function is looking for a value of zero in the loop, and moves the value stored in var_4 to eax before exiting. Based on the byte pointer movement and the loop checking for a zero, we deduce that this is a function that looks for the length of a string and returns it. We name sub_401700 to StringLength.

Returning to sub_401740, we see that the lengths of the strings are compared in registers esi and eax. If they don’t match, the value one is moved into eax and the function cuts out. Since the function is looking to compare string lengths and immediately cuts out after mismatched lengths, it becomes clear that sub_401740 is a string comparison function which is named StringCompare.
Again we jump out to sub_401190 and review what we’ve figured out. We have two strings. One was stored in the executable, the other is provided as input. If the strings are the same, the function returns. Otherwise, there is another function, sub_401960 called. Looking inside that function, we see the string:

**BOOM!!!**
The bomb has blown up.

Which makes it very clear that this is the function called when we fail to provide the correct input. I named that function Failure. So if the input we give doesn’t work, the function Failure is called, but otherwise we just return out of sub_401190 which I named PhaseOne. Typing the string:

Public speaking is very easy.

Leads to the response:

Phase 1 defused. How about the next one?

## 5 Phase Two

A function, sub_401990 gets called in between PhaseOne and printing the message about defusing the first phase. Looking inside, we see some strings about a secret phase. We’ll just ignore this for now, and move on to solve the second phase.

Sub_4011B0, which we rename to PhaseTwo, pushes an integer pointer and the pointer to the input string onto the stack. After that, it calls sub_4016B0. Sub_4016B0 immediately stands out with a string that IDA nicely shows us. This string is a print format string with six integers, reading as:

"%d %d %d %d %d %d"

I renamed sub_4016B0 SixIntStuff after this format string. SixIntStuff pushes a pointer to the stack, six times, each of which hold an integer. Then, the location of our input string is also pushed before a function called sscanf is called. The function sscanf, as the name implies, takes the string input and extracts the six integers to store in where we pushed those int locations. If six integers aren’t grabbed, then SixIntStuff will call the Failure function. Once those six integers are stored, we return to the main portion of PhaseTwo. The function checks if the first integer in the input is a one and calls Failure if it’s not. Then we encounter a loop, which uses a variable stored at ebp+4 to track this loop and sees if it’s less than six. If it is, then the loop progresses, increments and starts over. If it’s equal or greater than six, then the function returns as a success.

The body of the loop is key to solving this phase. It is multiplying a value
stored in ebp+(ecx*4)-0x20 by 1. If you debug the program and examine this memory, you’ll see these values are integers which match those we provide for input to solve. If the input doesn’t match those integers, Failure is called. Otherwise, the loop proceeds until PhaseTwo is completed. The proper input...

1 2 6 24 120 720

Will lead to the response...

That’s number 2. Keep going!

Which indicates our success in completing PhaseTwo.

6 Phase Three

We return to main and see that the secret phase is called again. After that, the bomb gets new input from the user and calls sub_401210 which I rename to PhaseThree. PhaseThree greets us with another format string looking for an integer, a character and another integer. Not following that format leads to a failure.

After which, the code branches into a jump table. The first integer input lets you pick which branch you want to tackle, they are all solvable. If your first integer is greater than 7, the Failure function is called. Otherwise, you’ll jump to the branch matching your first integer input. I chose branch 1 at location 40127A.

Branch 1 stores 0x62 into ebp+1 then compares the value 0xD6 to the third integer input we give (214 in base 10). Failure is called if the third integer input is not 214. Otherwise it jumps to the location 401312 and stores ebp+1 into eax (eax will get 0x62). Then it stores our input character into ecx. 0x62 is the character ‘b’ so giving that as input will allow the function to return. The proper input (although there are others) is:

1 b 214

... which then leads to the output...

Halfway there!

... indicating our success in PhaseThree.
7 Phase Four

Sub_401390, renamed to PhaseFour, looks for a single integer for input as evidenced by the format string. If a single integer isn’t provided, the Failure function gets called. Then, the value we gave as input is stashed in the stack before calling sub_401350. Before diving in, notice that PhaseFour does a comparison to 0x37 (which is 55 in base 10) to determine failure or success. Sub_401350 is unique in that it calls sub_401350. I renamed it to Recursion because of this. Recursion checks if the passed arg is greater than one. If it is, it goes into the branch that makes the recursive calls. If it’s less than or equal to one, it goes into the base case that returns 1. The recursive calls take the argument inputs, subtracts one from that, then calls Recursion. Then, it stores the result in esi. After that, Recursion is called again but with two subtracted from the argument. Then it returns the addition of Recursion(arg - 1) and Recursion(arg - 2). This is, if you can tell, a function that returns numbers in the Fibonacci sequence. Here is a table that shows the result of calls to Recursion with different arguments:

<table>
<thead>
<tr>
<th>recursion</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>recursion(1)</td>
<td>1</td>
</tr>
<tr>
<td>recursion(2)</td>
<td>2</td>
</tr>
<tr>
<td>recursion(3)</td>
<td>3</td>
</tr>
<tr>
<td>recursion(4)</td>
<td>5</td>
</tr>
<tr>
<td>recursion(5)</td>
<td>8</td>
</tr>
<tr>
<td>recursion(6)</td>
<td>13</td>
</tr>
<tr>
<td>recursion(7)</td>
<td>21</td>
</tr>
<tr>
<td>recursion(8)</td>
<td>34</td>
</tr>
<tr>
<td>recursion(9)</td>
<td>55</td>
</tr>
</tbody>
</table>

Looking at recursion(9), the value 55 that PhaseFour is looking for is there. Giving the input...

9

... then gives the output...

So you got that one. Try this one.

Which indicates that we completed PhaseFour.

8 Phase Five

Sub_4013E0, renamed PhaseFive, takes a string and checks the string for a length of six. If the string is longer or shorter, Failure gets called. With a proper length, the execution goes into a loop which iterates six times. IDA reveals the string ”giants” but that is not the right input. Instead, the input needs to be a string where each character’s lower four bits map to the letters
starting at byte_1024580.
Using the hex window to explore byte_1024580 reveals the string of characters “isrveawhobpnutfg” which contain the letters needed to spell “giants” to complete the phase. This means we need to map to the index 15 (0xF), 0, 5, 11 (0xB), 13 (0xD) and 1. This means the lower four bits of the binary representations of the characters of our input string need to match those indices. Although there are multiple possible answers, one that I used that works is...

o@ekma

... which gives the output...

Good work! On to the next...

That completes this phase of the challenge. Here is a table to show you why that input works:

<table>
<thead>
<tr>
<th>char</th>
<th>o</th>
<th>@</th>
<th>e</th>
<th>k</th>
<th>m</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>hex</td>
<td>0x6F</td>
<td>0x40</td>
<td>0x65</td>
<td>0x6B</td>
<td>0x6D</td>
<td>0x61</td>
</tr>
<tr>
<td>hex &amp; 0x0F</td>
<td>0xF</td>
<td>0x0</td>
<td>0x5</td>
<td>0xB</td>
<td>0xD</td>
<td>0x1</td>
</tr>
</tbody>
</table>

9 Phase Six
I’m still working on this one. Check back in a little while!