bomb

dj$_{\text{dvorak}}$

April 23, 2020

Contents

The bomb is a program that takes 6 inputs from user prompt which are presumably "secrets". Failure to enter the correct secret will cause the bomb to detonate! (explode). Using reverse engineering and debugging, we can find the secret answers for these six phases. Debugging symbols are provided with the binary.

1 Phase 1

The phase 1 function calls a string compare function to the string "Public speaking is very easy.". This is the expected answer. Pretty straightforward.

2 Phase 2

The second phase function has an internal call to a function `read_six_numbers` which presumably does exactly that. Within `read_six_numbers()`, we

see a call to `scanf("%d %d %d %d %d %d")`, so indeed there are six integers being loaded into an array. Any less and fail. Here is the decomposed function which individually checks the elements for some trait.

```
1
2  void phase_2(undefined4 input)
3
4  {
5    int index;
6    int input_temp [7];
7
8    read_six_numbers(input,input_temp + 1);
9    if (input_temp[1] != 1) {
10     explode_bomb();
11   }
12   index = 1;
13   do {
14     if (input_temp[index + 1] != (index + 1) * input_temp[index]) {
15       explode_bomb();
16     }
17     index = index + 1;
18   } while (index < 6);
19   return;
```

Here, we note that the following formula must be satisfied for array x.

$$x_1 := 1$$

$$x_{i+1} := (i+1) * x_i$$

Of course, this recursive function and initial conditions allows only one input `[1, 2, 6, 24, 120, 720]` ... hey! it's the factorial function!!

3 Phase 3

Like the last phase, phase 3 uses a `scanf()` to ask for three parameters. Any less and BOOM! Now we're looking for two integers with a `char` sandwiched in the middle.

```
void phase_3(char *param_1)

{
  int iVar1;
  char c2;
  uint i1;
  char c;
  int i2;
```

```
iVar1 = sscanf(param_1,"%d %c %d",&i1,&c,&i2);
if (iVar1 < 3) {
  explode_bomb();
}
switch(i1) {
case 0:
  c2 = 'q';
  if (i2 != 0x309) {
    explode_bomb();
  }
  break;
case 1:
  c2 = 'b';
  if (i2 != 0xd6) {
    explode_bomb();
  }
  break;
case 2:
  c2 = 'b';
  if (i2 != 0x2f3) {
    explode_bomb();
  }
  break;
case 3:
  c2 = 'k';
  if (i2 != 0xfb) {
    explode_bomb();
  }
  break;
case 4:
  c2 = 'o';
  if (i2 != 0xa0) {
    explode_bomb();
  }
  break;
case 5:
  c2 = 't';
  if (i2 != 0x1ca) {
    explode_bomb();
```

```
    }
    break;
  case 6:
    c2 = 'v';
    if (i2 != 0x30c) {
      explode_bomb();
    }
    break;
  case 7:
    c2 = 'b';
    if (i2 != 0x20c) {
      explode_bomb();
    }
    break;
  default:
    c2 = 'x';
    explode_bomb();
  }
  if (c2 != c) {
    explode_bomb();
  }
  return;
}
```

We see at the bottom that our input c must be equal to some c2, which is set in the switch statement. The switch takes the first integer, `i1`. So we can just pick a tuple of `(i1, c, i2)` that is valid from this table. I chose `(0, 'q', 777)`.

4 Phase 4

Phase 4 starts out pretty straightforward. It scans for a single integer which must be greater than 1. Then it checks if the output of `func4(i)` is 55 . We need to dissect then `func4()`.

```
void phase_4(char *param_1)

{
  int num_input;
  int i;
```

```
  num_input = sscanf(param_1,"%d",&i);
  if ((num_input != 1) || (i < 1)) {
    explode_bomb();
  }
  num_input = func4(i);
  if (num_input != 0x37) {
    explode_bomb();
  }
  return;
}

int func4(int in)
{
  int a;
  int b;

  if (in < 2) {
    b = 1;
  }
  else {
    a = func4(in + -1);
    b = func4(in + -2);
    b = b + a;
  }
  return b;
}
```

And what do you know, it's a recursive function! Instead of doing it by hand, we can just run the code ourselves and find the answer by plugging in arbitrary values of `in`. Doing so gives us the answer 9.

5 Phase 5

Phase 5 is an interesting puzzle. It resembles a cipher!

```
void phase_5(int in_str)

{
  int l;
```

```
  undefined local_c [6];
  undefined local_6;

  l = string_length(in_str);
  if (l != 6) {
    explode_bomb();
  }
  l = 0;
  do {
    local_c[l] = (&array.123)[(char)(*(byte *)(l + in_str) & 0xf)];
    l = l + 1;
  } while (l < 6);
  local_6 = 0;
  l = strings_not_equal(local_c,"giants");
  if (l != 0) {
    explode_bomb();
  }
  return;
}
```

The gist is you build need to build the string ="giants"= by reverse engineering the cipher. There is an equation that looks up chars in `array.123` which contains the following:

`i s r v e a w h o b p n u t f g`

So the "enigma" here is the use of modulo-16 function. To spell giants we need a ASCII sequence that when mod-16 yields `f 0 5 b d 1`. Actually since we can just add the key as an offset to... let's say 64 or 0x40. So we need the ascii characters `O @ E K M A` without the spaces of course.

6 Phase 6

Okay, phase 6 starts to play with pointers and maybe this struct which makes things a lot dirtier.

The first part is rather straightforward:

```
void phase_6(undefined4 param_1)

{
  astruct *ptr1;
  int b;
```

```
 astruct *ptr2;
 int i;
 astruct *first_node;
 astruct *6ptrs [6];
 int 6nums [6];

 first_node = &node1;
 read_six_numbers(param_1,6nums);
 i = 0;
 do {
   if (5 < 6nums[i] - 1U) {
     explode_bomb();
   }
   b = i + 1;
   if (b < 6) {
     do {
if (6nums[i] == 6nums[b]) {
  explode_bomb();
}
b = b + 1;
     } while (b < 6);
   }
   i = i + 1;
 } while (i < 6);
 i = 0;
 do {
   b = 1;
   ptr2 = first_node;
   if (1 < 6nums[i]) {
     do {
ptr2 = (astruct *)ptr2->nextNode;
b = b + 1;
     } while (b < 6nums[i]);
   }
   6ptrs[i] = ptr2;
   i = i + 1;
 } while (i < 6);
 i = 1;
 ptr2 = 6ptrs[0];
 do {
```

```
    ptr1 = 6ptrs[i];
    *(astruct **)&ptr2->nextNode = ptr1;
    i = i + 1;
    ptr2 = ptr1;
  } while (i < 6);
  ptr1->nextNode = (int *)0x0;
  i = 0;
  do {
    if (6ptrs[0]->key < *6ptrs[0]->nextNode) {
      explode_bomb();
    }
    6ptrs[0] = (astruct *)6ptrs[0]->nextNode;
    i = i + 1;
  } while (i < 5);
  return;
}
```

There are about 4 for loops here. The first one checks the input of 6 integers to make sure that they are greater than 6 and unique.

Then we have a mess. But realizing that we are dealing with structs, we can debug correctly by identifying the structure of the struct. We are given a hint in that node1 points to some hardcoded data and we can inspect it there. It appears we have a struct with two integers and a pointer. The pointer looks like they refer to other structs in a linear way... much like that of a LinkedList!

It becomes clear but the next 3 for loops are doing. They are loading the linked list in the order into my astruct array! Then `ptr1` finds the tail and ends the list. Then, we need to check the each nextNodes in increasing order. Hence the solution 4 2 6 3 1 5.