Defusing a binary bomb with `gdb` - Part 1

12 Nov 2015

This series of posts will show you how we can defuse a binary bomb. So what's a binary bomb?

> *“A "binary bomb" is a Linux executable C program that consists of six "phases." Each phase expects the student to enter a particular string on stdin. If the student enters the expected string, then that phase is "defused." Otherwise the bomb "explodes" by printing "BOOM!!!". The goal for the students is to defuse as many phases as possible.”*

I found this type of bomb in the website for the excellent book “Computer Systems: A Programmer's Perspective”.

The basic tools necessary to defuse such bomb are `gdb` and `objdump`. `gdb` is a debugger which we will use to inspect the program as we run it. `objdump` is a tool to disassemble object files so we can see the actual instructions that the computer is executing.

This series is not intended to be a tutorial about `gdb` specially because it was my first time using it.

Enough of that, let's start having some fun. After extracting the tarball we are left with:

```
$ ls -l
total 36
-rwxr-xr-x 1 carlos carlos 26406 Jun  9 15:41 bomb
-rw-r--r-- 1 carlos carlos  4069 Jun  9 15:41 bomb.c
-rw-rw-r-- 1 carlos carlos    49 Jun  9 15:46 README
```

Looking at `bomb.c` we see a bunch of comments and how everything is setup.

You can pass a file as argument to avoid typing every time the correct input for already defused phases.

Next we need to take a look at the `bomb` executable which is binary data so we won't see anything interesting if we open it using `$EDITOR`. That's why we need `objdump` to disassemble this executable.

```
$ objdump -d bomb > bomb.s
```

If we take a look at the first few lines of this new file we see:

```
bomb:     file format elf64-x86-64


Disassembly of section .init:

0000000000400ac0 <_init>:
  400ac0:   48 83 ec 08             sub    $0x8,%rsp
  400ac4:   e8 f3 01 00 00          callq  400cbc <call_gmon_start>
  400ac9:   48 83 c4 08             add    $0x8,%rsp
  400acd:   c3                      retq
```

That is what an ELF file looks like when disassembled. Let's look at the `main` function then:

```
0000000000400da0 <main>:
  400da0:   53                      push   %rbx
  400da1:   83 ff 01                cmp    $0x1,%edi
  400da4:   75 10                   jne    400db6 <main+0x16>
  400da6:   48 8b 05 9b 29 20 00    mov    0x20299b(%rip),%rax
# 603748 <stdin@@GLIBC_2.2.5>
  400dad:   48 89 05 b4 29 20 00    mov    %rax,0x2029b4(%rip)
# 603768 <infile>
  400db4:   eb 63                   jmp    400e19 <main+0x79>
  400db6:   48 89 f3                mov    %rsi,%rbx
  400db9:   83 ff 02                cmp    $0x2,%edi
  400dbc:   75 3a                   jne    400df8 <main+0x58>
  400dbe:   48 8b 7e 08             mov    0x8(%rsi),%rdi
  400dc2:   be b4 22 40 00          mov    $0x4022b4,%esi
  400dc7:   e8 44 fe ff ff          callq  400c10 <fopen@plt>
  400dcc:   48 89 05 95 29 20 00    mov    %rax,0x202995(%rip)
# 603768 <infile>
  400dd3:   48 85 c0                test   %rax,%rax
  400dd6:   75 41                   jne    400e19 <main+0x79>
  400dd8:   48 8b 4b 08             mov    0x8(%rbx),%rcx
  400ddc:   48 8b 13                mov    (%rbx),%rdx
  400ddf:   be b6 22 40 00          mov    $0x4022b6,%esi
  400de4:   bf 01 00 00 00          mov    $0x1,%edi
  400de9:   e8 12 fe ff ff          callq  400c00 <__printf_chk@plt>
  400dee:   bf 08 00 00 00          mov    $0x8,%edi
  400df3:   e8 28 fe ff ff          callq  400c20 <exit@plt>
  400df8:   48 8b 16                mov    (%rsi),%rdx
  400dfb:   be d3 22 40 00          mov    $0x4022d3,%esi
  400e00:   bf 01 00 00 00          mov    $0x1,%edi
  400e05:   b8 00 00 00 00          mov    $0x0,%eax
  400e0a:   e8 f1 fd ff ff          callq  400c00 <__printf_chk@plt>
  400e0f:   bf 08 00 00 00          mov    $0x8,%edi
  400e14:   e8 07 fe ff ff          callq  400c20 <exit@plt>
  400e19:   e8 84 05 00 00          callq  4013a2 <initialize_bomb>
  400e1e:   bf 38 23 40 00          mov    $0x402338,%edi
  400e23:   e8 e8 fc ff ff          callq  400b10 <puts@plt>
  400e28:   bf 78 23 40 00          mov    $0x402378,%edi
  400e2d:   e8 de fc ff ff          callq  400b10 <puts@plt>
  400e32:   e8 67 06 00 00          callq  40149e <read_line>
  400e37:   48 89 c7                mov    %rax,%rdi
  400e3a:   e8 a1 00 00 00          callq  400ee0 <phase_1>
  400e3f:   e8 80 07 00 00          callq  4015c4 <phase_defused>
```

I didn't paste the entire function since it's big enough and we are not concerned about the other phases yet.

Before we start analyzing the function we need to understand the structure of each line. Let's take the following line as example:

```
  400db6:   48 89 f3                mov    %rsi,%rbx
```

We can break this line into three sections:

- `400db6` : the address of the code we are looking at.
- `48 89 f3` : the encoded instruction.
- `mov %rsi,%rbx` : the decoded instruction.

The first few lines in the `main` function correspond to the C code that checks whether or not we passed a file as argument to the program. Skipping those lines we start to see the fun part:

```
  400e19:   e8 84 05 00 00          callq  4013a2 <initialize_bomb>
```

This line says that the function `initialize_bomb` should be called. The correspoding line in the C file is the following:

```
/* Do all sorts of secret stuff that makes the bomb harder to defuse.
*/
initialize_bomb();
```

So let's jump to the `initialize_bomb` function.

```
00000000004013a2 <initialize_bomb>:
  4013a2:   48 83 ec 08             sub    $0x8,%rsp
  4013a6:   be a0 12 40 00          mov    $0x4012a0,%esi
  4013ab:   bf 02 00 00 00          mov    $0x2,%edi
  4013b0:   e8 db f7 ff ff          callq  400b90 <signal@plt>
  4013b5:   48 83 c4 08             add    $0x8,%rsp
  4013b9:   c3                      retq
```

Inspecting the values don't reveal anything interesting. Let's move on. The next few lines after `initialize_bomb` in the `main` function correspond to the following lines in the C file:

```
printf("Welcome to my fiendish little bomb. You have 6 phases with
\n");
printf("which to blow yourself up. Have a nice day!\n");

/* Hmm...  Six phases must be more secure than one phase! */
input = read_line();             /* Get input                   */
phase_1(input);                  /* Run the phase               */
```

So they print the messages and read the input. Then it's time to defuse the first phase.

```
  400e3a:   e8 a1 00 00 00          callq  400ee0 <phase_1>
```

Again, this calls the function `phase_1` located at `0x400ee0`. Let's see what the first phase looks like:

```
0000000000400ee0 <phase_1>:
  400ee0:   48 83 ec 08             sub    $0x8,%rsp
  400ee4:   be 00 24 40 00          mov    $0x402400,%esi
  400ee9:   e8 4a 04 00 00          callq  401338 <strings_not_equal>
  400eee:   85 c0                   test   %eax,%eax
  400ef0:   74 05                   je     400ef7 <phase_1+0x17>
  400ef2:   e8 43 05 00 00          callq  40143a <explode_bomb>
  400ef7:   48 83 c4 08             add    $0x8,%rsp
  400efb:   c3                      retq
```

Notice on `0x400ee4` that the value `0x402400` is copied to the register `esi`. The `esi` register is usually used as the register for the second argument of a function that will be called later. In our case such function is called right after the `mov` instruction. You might then ask: where is the first argument? The first argument is usually placed in the `edi` register which in this case will be the string we provided as input. If you take a look at the `main` function you will see:

```
  400e32:   e8 67 06 00 00          callq  40149e <read_line>
  400e37:   48 89 c7                mov    %rax,%rdi
  400e3a:   e8 a1 00 00 00          callq  400ee0 <phase_1>
```

The return value (stored in `rax`) of the `read_line` function has been placed in the `rdi` register (`edi` is a 32-bit register and `rdi` is the equivalent 64-bit register) and will be used as the first argument for the function that will be called next which in this case is `phase_1`. And that is exactly what the C code is doing:

```
/* Hmm...  Six phases must be more secure than one phase! */
input = read_line();             /* Get input                   */
phase_1(input);                  /* Run the phase               */
```

Ok, back to the `phase_1` function. We now know what the arguments given to `strings_not_equal` are and after executing such function there is a test to check the result:

```
400ee9: e8 4a 04 00 00          callq  401338 <strings_not_equal>
400eee: 85 c0                   test   %eax,%eax
400ef0: 74 05                   je     400ef7 <phase_1+0x17>
400ef2: e8 43 05 00 00          callq  40143a <explode_bomb>
```

The `test` instruction will perform a bitwise AND operation between its operands and set the appropriate flags on register `eflags`. The `je` instruction is a conditional jump instruction that jumps to the specified location only if the previous comparison set the `ZF` (Zero Flag) to `1` in the `eflags` register.

So the `test` instruction will set `ZF` to `1` only when we have `0` in `eax` which only happens when `strings_not_equal` returns `0`. *(Examining `strings_not_equal` doesn't reveal anything interesting, it's exactly what you expect from a function with such name. It returns `1` if both arguments are not equal and `0` otherwise.)*

If the strings are not equal the conditional jump will not be performed and then the next line will be executed which will explode the bomb. If the strings are equal we jump to `0x400eef7` and return to `main`:

```
  400ef0:   74 05                    je     400ef7 <phase_1+0x17>
  400ef2:   e8 43 05 00 00           callq  40143a <explode_bomb>
  400ef7:   48 83 c4 08              add    $0x8,%rsp
  400efb:   c3                       retq
```

Ok, we now know that the first phase requires us to provide a string that we don't know. How we are going to discover which string is this? We need to start executing the program. But in this case instead of executing like you usually do with other programs we will run it with `gdb`. `gdb` will help us to inspect the values and find out what is this mysterious string.

```
$ gdb bomb
```

The line above starts `gdb` with the `bomb` program attached to it so we can execute the `bomb` and inspect the values, set breakpoints, etc. In this case we have already done most of the work by only examining the assembly code and we know that the mysterious string is located at address `0x402400` (when it was loaded on register `esi` at address `0x400ee4`). To see what is the value of it we can simply ask `gdb` to print the value at the desired address and treat it as sequence of `char`:

```
(gdb) p (char *) 0x402400
$1 = 0x402400 "Border relations with Canada have never been better."
```

And voilà! We have the string we need.

Now executing the program:

```
(gdb) run
Starting program: /home/carlos/Downloads/bomb/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
```

Then entering the full string we will see that phase 1 was defused:

```
Border relations with Canada have never been better.
Phase 1 defused. How about the next one?
```

Defusing a binary bomb with `gdb` - Part 2

19 Nov 2015

> *This post is part of a series where I show how to defuse a binary bomb by reading assembly code and using `gdb`. You might want to read the first part if you haven't yet.*

After defusing the first phase we were challenged to defuse the next one:

```
Border relations with Canada have never been better.
Phase 1 defused. How about the next one?
```

The corresponding assembly code in the `main` function is the following:

```
400e3a: e8 a1 00 00 00          callq  400ee0 <phase_1>
400e3f: e8 80 07 00 00          callq  4015c4 <phase_defused>
400e44: bf a8 23 40 00          mov    $0x4023a8,%edi
400e49: e8 c2 fc ff ff          callq  400b10 <puts@plt>
400e4e: e8 4b 06 00 00          callq  40149e <read_line>
400e53: 48 89 c7                mov    %rax,%rdi
400e56: e8 a1 00 00 00          callq  400efc <phase_2>
400e5b: e8 64 07 00 00          callq  4015c4 <phase_defused>
```

As we can see (at `0x400e53`) it puts our input in the `rdi` register to be used as the first argument to `phase_2` which will be called by the next instruction. Just like you would imagine that the actual C code is doing:

```
printf("Phase 1 defused. How about the next one?\n");

/* The second phase is harder.  No one will ever figure out
 * how to defuse this... */
input = read_line();
phase_2(input);
phase_defused();
```

So what `phase_2` looks like?

```
0000000000400efc <phase_2>:
  400efc:   55                      push   %rbp
  400efd:   53                      push   %rbx
  400efe:   48 83 ec 28             sub    $0x28,%rsp
  400f02:   48 89 e6                mov    %rsp,%rsi
  400f05:   e8 52 05 00 00          callq  40145c <read_six_numbers>
  400f0a:   83 3c 24 01             cmpl   $0x1,(%rsp)
  400f0e:   74 20                   je     400f30 <phase_2+0x34>
  400f10:   e8 25 05 00 00          callq  40143a <explode_bomb>
  400f15:   eb 19                   jmp    400f30 <phase_2+0x34>
  400f17:   8b 43 fc                mov    -0x4(%rbx),%eax
  400f1a:   01 c0                   add    %eax,%eax
  400f1c:   39 03                   cmp    %eax,(%rbx)
  400f1e:   74 05                   je     400f25 <phase_2+0x29>
  400f20:   e8 15 05 00 00          callq  40143a <explode_bomb>
  400f25:   48 83 c3 04             add    $0x4,%rbx
  400f29:   48 39 eb                cmp    %rbp,%rbx
  400f2c:   75 e9                   jne    400f17 <phase_2+0x1b>
  400f2e:   eb 0c                   jmp    400f3c <phase_2+0x40>
  400f30:   48 8d 5c 24 04          lea    0x4(%rsp),%rbx
  400f35:   48 8d 6c 24 18          lea    0x18(%rsp),%rbp
  400f3a:   eb db                   jmp    400f17 <phase_2+0x1b>
  400f3c:   48 83 c4 28             add    $0x28,%rsp
  400f40:   5b                      pop    %rbx
  400f41:   5d                      pop    %rbp
  400f42:   c3                      retq
```

Right off the bat we can see that this phase is expecting us to enter six numbers:

```
400f05: e8 52 05 00 00          callq  40145c <read_six_numbers>
```

That can be confirmed by inspecting `read_six_numbers` function:

```
000000000040145c <read_six_numbers>:
  40145c:   48 83 ec 18             sub    $0x18,%rsp
  401460:   48 89 f2                mov    %rsi,%rdx
  401463:   48 8d 4e 04             lea    0x4(%rsi),%rcx
  401467:   48 8d 46 14             lea    0x14(%rsi),%rax
  40146b:   48 89 44 24 08          mov    %rax,0x8(%rsp)
  401470:   48 8d 46 10             lea    0x10(%rsi),%rax
  401474:   48 89 04 24             mov    %rax,(%rsp)
  401478:   4c 8d 4e 0c             lea    0xc(%rsi),%r9
  40147c:   4c 8d 46 08             lea    0x8(%rsi),%r8
  401480:   be c3 25 40 00          mov    $0x4025c3,%esi
  401485:   b8 00 00 00 00          mov    $0x0,%eax
  40148a:   e8 61 f7 ff ff          callq  400bf0 <__isoc99_sscanf@pl
t>
  40148f:   83 f8 05                cmp    $0x5,%eax
  401492:   7f 05                   jg     401499 <read_six_numbers+0
x3d>
  401494:   e8 a1 ff ff ff          callq  40143a <explode_bomb>
  401499:   48 83 c4 18             add    $0x18,%rsp
  40149d:   c3                      retq
```

At `0x40148a` we see that it calls `sscanf` which has the following purpose:

> ```
> #include <stdio.h>
>
> int scanf(const char *format, ...);
> int fscanf(FILE *stream, const char *format, ...);
> int sscanf(const char *str, const char *format, ...);
> ```
>
> *The `scanf()` family of functions scans input according to `format` as described below. This format may contain conversion specifications; the results from such conversions, if any, are stored in the locations pointed to by the pointer arguments that follow `format`. Each pointer argument must be of a type that is appropriate for the value returned by the corresponding conversion specification.*

Following the same idea we used on phase 1 we can confirm this function does exactly what its name suggests. On `0x401480` something is stored at `esi` to be used as the second argument for `sscanf` which as seen above is the expected format for our input.

```
401480: be c3 25 40 00          mov    $0x4025c3,%esi
```

Then on `gdb` we can print the value just like we did on `phase_1`:

```
(gdb) p (char *) 0x4025c3
$1 = 0x4025c3 "%d %d %d %d %d %d"
```

`read_six_numbers` then checks if we typed at least six numbers, if we did it returns, otherwise the bomb explodes.

Back at `phase_2` function we find that our first number must be `1` (comparison at `0x400f0a`) otherwise the bomb will explode right away:

```
400f05: e8 52 05 00 00          callq  40145c <read_six_numbers>
400f0a: 83 3c 24 01             cmpl   $0x1,(%rsp)
400f0e: 74 20                   je     400f30 <phase_2+0x34>
400f10: e8 25 05 00 00          callq  40143a <explode_bomb>
```

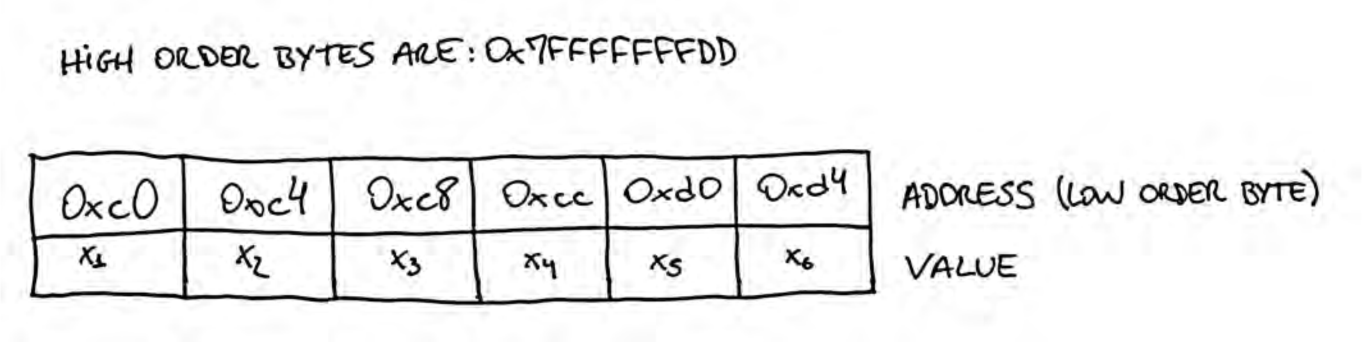
After confirming that our first number was `1` it goes to `0x400f30`:

```
400f30: 48 8d 5c 24 04          lea    0x4(%rsp),%rbx
400f35: 48 8d 6c 24 18          lea    0x18(%rsp),%rbp
400f3a: eb db                   jmp    400f17 <phase_2+0x1b>
```

On `0x400f30` the address of the next number is stored on `rbx` and on `0x400f35` `rbp` gets the address right after the address of the last number parsed by `sscanf` on `read_six_numbers`.

```
(gdb) p $rsp+0x18
$2 = (void *) 0x7fffffffddd8
(gdb) p $rsp
$3 = (void *) 0x7fffffffddc0
```

Considering just the low order byte: `0xd8` - `0xc0` = `0x18`. Which is decimal `24`. Each `int` takes four bytes so the memory structure looks like the image below which explains why `rbp` holds the address after the sixth number:



HIGH ORDER BYTES ARE: 0x7FFFFFFFDD

0xc0	0xc4	0xc8	0xcc	0xd0	0xd4	ADDRESS (LOW ORDER BYTE)
x_1	x_2	x_3	x_4	x_5	x_6	VALUE

Then the execution will continue on `0x400f17`:

```
400f17: 8b 43 fc                mov    -0x4(%rbx),%eax
400f1a: 01 c0                   add    %eax,%eax
400f1c: 39 03                   cmp    %eax,(%rbx)
400f1e: 74 05                   je     400f25 <phase_2+0x29>
400f20: e8 15 05 00 00          callq  40143a <explode_bomb>
```

On `0x400f17` the previous number is copied into `eax` then the next instruction duplicates this value on `eax` which is then compared with our second number. If they are equal the function will continue execution at `0x400f25`, otherwise you know what.

On `0x400f25` the pointer goes to the next number. Next it checks if the pointer passed the last number which means all six numbers were checked. If it didn't it goes back to `0x400f17` to check the next number and if all numbers were already checked it will jump to `0x400f3c` that will then return to `main`.

```
400f25: 48 83 c3 04             add    $0x4,%rbx
400f29: 48 39 eb                cmp    %rbp,%rbx
400f2c: 75 e9                   jne    400f17 <phase_2+0x1b>
400f2e: eb 0c                   jmp    400f3c <phase_2+0x40>
```

Checking numbers, moving pointers forward, jumping back and forth suggests that we are dealing with a loop. Assuming `p` is a pointer to the first number, the loop in `phase_2` might look like the following:

```
for (int *x = p + 1; x != (p + 6); x++) {
    int previous = *(x - 1);
    if (*x != previous * 2)
        explode_bomb();
}
```

Alright, now we have an idea of what the next five numbers should be. They have to be the double of the previous number. If we start at `1` the next is `2`, the next is `4` and so on. This might ring a bell, doesn't it? Our input must be the first six powers of `2`:

- $2^0 = 1$
- $2^1 = 2$
- $2^2 = 4$
- $2^3 = 8$
- $2^4 = 16$
- $2^5 = 32$

After entering the six numbers we see that we defused the second phase:

```
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
```

Defusing a binary bomb with `gdb` - Part 3

03 Dec 2015

> *This post is part of a series where I show how to defuse a binary bomb by reading assembly code and using `gdb`. You might want to read the other parts if you haven't yet.*

Following the usual process, after defusing the second phase we were challenged to defuse the third one:

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Border relations with Canada have never been better.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
```

The corresponding instructions on `main` are the following:

```
400e5b: e8 64 07 00 00          callq  4015c4 <phase_defused>
400e60: bf ed 22 40 00          mov    $0x4022ed,%edi
400e65: e8 a6 fc ff ff          callq  400b10 <puts@plt>
400e6a: e8 2f 06 00 00          callq  40149e <read_line>
400e6f: 48 89 c7                mov    %rax,%rdi
400e72: e8 cc 00 00 00          callq  400f43 <phase_3>
```

The code for `phase_3` is the following:

```
0000000000400f43 <phase_3>:
  400f43:   48 83 ec 18             sub    $0x18,%rsp
  400f47:   48 8d 4c 24 0c          lea    0xc(%rsp),%rcx
  400f4c:   48 8d 54 24 08          lea    0x8(%rsp),%rdx
  400f51:   be cf 25 40 00          mov    $0x4025cf,%esi
  400f56:   b8 00 00 00 00          mov    $0x0,%eax
  400f5b:   e8 90 fc ff ff          callq  400bf0 <__isoc99_sscanf@pl
t>
  400f60:   83 f8 01                cmp    $0x1,%eax
  400f63:   7f 05                   jg     400f6a <phase_3+0x27>
  400f65:   e8 d0 04 00 00          callq  40143a <explode_bomb>
  400f6a:   83 7c 24 08 07          cmpl   $0x7,0x8(%rsp)
  400f6f:   77 3c                   ja     400fad <phase_3+0x6a>
  400f71:   8b 44 24 08             mov    0x8(%rsp),%eax
  400f75:   ff 24 c5 70 24 40 00    jmpq   *0x402470(,%rax,8)
  400f7c:   b8 cf 00 00 00          mov    $0xcf,%eax
  400f81:   eb 3b                   jmp    400fbe <phase_3+0x7b>
  400f83:   b8 c3 02 00 00          mov    $0x2c3,%eax
  400f88:   eb 34                   jmp    400fbe <phase_3+0x7b>
  400f8a:   b8 00 01 00 00          mov    $0x100,%eax
  400f8f:   eb 2d                   jmp    400fbe <phase_3+0x7b>
  400f91:   b8 85 01 00 00          mov    $0x185,%eax
  400f96:   eb 26                   jmp    400fbe <phase_3+0x7b>
  400f98:   b8 ce 00 00 00          mov    $0xce,%eax
  400f9d:   eb 1f                   jmp    400fbe <phase_3+0x7b>
  400f9f:   b8 aa 02 00 00          mov    $0x2aa,%eax
  400fa4:   eb 18                   jmp    400fbe <phase_3+0x7b>
  400fa6:   b8 47 01 00 00          mov    $0x147,%eax
  400fab:   eb 11                   jmp    400fbe <phase_3+0x7b>
  400fad:   e8 88 04 00 00          callq  40143a <explode_bomb>
  400fb2:   b8 00 00 00 00          mov    $0x0,%eax
  400fb7:   eb 05                   jmp    400fbe <phase_3+0x7b>
  400fb9:   b8 37 01 00 00          mov    $0x137,%eax
  400fbe:   3b 44 24 0c             cmp    0xc(%rsp),%eax
  400fc2:   74 05                   je     400fc9 <phase_3+0x86>
  400fc4:   e8 71 04 00 00          callq  40143a <explode_bomb>
  400fc9:   48 83 c4 18             add    $0x18,%rsp
  400fcd:   c3                      retq
```

On `0x400f51` we see that some value is stored on `esi`, `eax` gets initialized and then `sscanf` is called.

If you don't remember from the previous phase, `sscanf` is a function that scans input according to some format that is given as argument to it. Such format must be what's stored on `esi`.

```
(gdb) p (char *) 0x4025cf
$1 = 0x4025cf "%d %d"
```

So we must enter 2 integers. This could also be confirmed by looking at the following instructions which compare if we entered more than one integer to continue executing the third phase otherwise the bomb will explode.

```
  400f60:   83 f8 01                cmp    $0x1,%eax
  400f63:   7f 05                   jg     400f6a <phase_3+0x27>
  400f65:   e8 d0 04 00 00          callq  40143a <explode_bomb>
```

You might be wondering how `sscanf` got its first argument which is the string used as source for scanning the desired values. As explained in previous posts the first argument to functions is usually placed on register `rdi` and any instruction can interact with any register, the same way you would interact with a global variable in a program. If you look at `0x400e6f` in the `main` function the string we enter as input for `phase_3` is copied to `rdi` to then be used as the first argument to `sscanf`:

```
400e6f: 48 89 c7                mov    %rax,%rdi
400e72: e8 cc 00 00 00          callq  400f43 <phase_3>
```

Ok, continuing the execution of `phase_3` the next instruction to be executed is located at `0x400f6a` (assuming we entered two integers, of course). At that location the program compare the first integer we gave as input. It should be less than or equal to `7` otherwise the execution will continue on `0x400fad`.

```
  400f6a:   83 7c 24 08 07          cmpl   $0x7,0x8(%rsp)
  400f6f:   77 3c                   ja     400fad <phase_3+0x6a>
```

And at `0x400fad` we know what's expecting us:

```
  400fad:   e8 88 04 00 00          callq  40143a <explode_bomb>
```

Assuming we entered an integer smaller than or equal to `7` the program continues:

```
  400f71:   8b 44 24 08             mov    0x8(%rsp),%eax
  400f75:   ff 24 c5 70 24 40 00    jmpq   *0x402470(,%rax,8)
```

The first instruction above will copy the first integer to `eax` and then on the second instruction jump to a location based on this integer. Let's pretend we entered `0` as our first integer. In that case the program would jump to the address location stored at `0x402470`. The rule for calculating the address is the following:

```
*(%rax * 8 + 0x402470)
```

That is: multiply the value stored on `rax` by `8`, add it to `0x402470` and then read the value stored at the result location. Inspecting the value on `0x402470` we see that's the address of the next line:

```
(gdb) x 0x402470
0x402470:   0x00400f7c
```

So entering `0` as our first number would execute the following instructions:

```
  400f7c:   b8 cf 00 00 00          mov    $0xcf,%eax
  400f81:   eb 3b                   jmp    400fbe <phase_3+0x7b>
```

This stores `0xcf` (decimal 207) on `eax` and then jumps to `0x400fbe` that does the following:

```
  400fbe:   3b 44 24 0c             cmp    0xc(%rsp),%eax
  400fc2:   74 05                   je     400fc9 <phase_3+0x86>
  400fc4:   e8 71 04 00 00          callq  40143a <explode_bomb>
  400fc9:   48 83 c4 18             add    $0x18,%rsp
  400fcd:   c3                      retq
```

In other words, on `0x400fbe` the program will compare our second number to what was stored on `eax`, in this imaginary case of entering `0` as our first number it would then compare if our second number was `207`. If that's the case it means we defused `phase_3`:

```
0 207
Halfway there!
```

So that's it?! Pretty simple, right? But what about all the other instructions above `400fbe`? What are their purpose?

All these instructions are part of a `switch` statement. That's why on `0x400f75` the address that the program will jump to will be calculated based on what we entered, opposed to what happened before when the location to jump to was hardcoded in the instruction itself[1]. Taking a closer look at the instructions before `0x400fbe` we see that they all follow the same pattern: store some value on `eax` and then jump to `0x400fbe` to compare our second number to this value stored on `eax`. So `phase_3` has more than one answer, let's see all of them:

```
(gdb) x/8g 0x402470
0x402470:   0x0000000000400f7c  0x0000000000400fb9
0x402480:   0x0000000000400f83  0x0000000000400f8a
0x402490:   0x0000000000400f91  0x0000000000400f98
0x4024a0:   0x0000000000400f9f  0x0000000000400fa6
```

The command above tells `gdb` to examine the memory starting at address `0x402470` and display eight blocks (`8` in the command) of eight bytes (`g` in the command, `g` as in giant words). The output then shows two values per line so we can build a table relating the first input number, the address the `switch` jumps to and what should be the second input number:

First input number	Address to jump to	Expected second input number (hex)	Expected second input number (decimal)
0	`0x400f7c`	`0xcf`	207
1	`0x400fb9`	`0x137`	311
2	`0x400f83`	`0x2c3`	707
3	`0x400f8a`	`0x100`	256
4	`0x400f91`	`0x185`	389
5	`0x400f98`	`0xce`	206
6	`0x400f9f`	`0x2aa`	682

7	`0x400fa6`	`0x147`	327

Any of the combinations above will work.

Notes

1. In this case using PC-relative addressing. ↩

Defusing a binary bomb with `gdb` - Part 4

25 Apr 2016

> *This post is part of a series where I show how to defuse a binary bomb by reading assembly code and using `gdb`. You might want to read the first part if you haven't yet.*

We are back on defusing the fourth phase of the binary bomb.

The code for `phase_4` is the following:

```
000000000040100c <phase_4>:
  40100c:   48 83 ec 18             sub    $0x18,%rsp
  401010:   48 8d 4c 24 0c          lea    0xc(%rsp),%rcx
  401015:   48 8d 54 24 08          lea    0x8(%rsp),%rdx
  40101a:   be cf 25 40 00          mov    $0x4025cf,%esi
  40101f:   b8 00 00 00 00          mov    $0x0,%eax
  401024:   e8 c7 fb ff ff          callq  400bf0 <__isoc99_sscanf@pl
t>
  401029:   83 f8 02                cmp    $0x2,%eax
  40102c:   75 07                   jne    401035 <phase_4+0x29>
  40102e:   83 7c 24 08 0e          cmpl   $0xe,0x8(%rsp)
  401033:   76 05                   jbe    40103a <phase_4+0x2e>
  401035:   e8 00 04 00 00          callq  40143a <explode_bomb>
  40103a:   ba 0e 00 00 00          mov    $0xe,%edx
  40103f:   be 00 00 00 00          mov    $0x0,%esi
  401044:   8b 7c 24 08             mov    0x8(%rsp),%edi
  401048:   e8 81 ff ff ff          callq  400fce <func4>
  40104d:   85 c0                   test   %eax,%eax
  40104f:   75 07                   jne    401058 <phase_4+0x4c>
  401051:   83 7c 24 0c 00          cmpl   $0x0,0xc(%rsp)
  401056:   74 05                   je     40105d <phase_4+0x51>
  401058:   e8 dd 03 00 00          callq  40143a <explode_bomb>
  40105d:   48 83 c4 18             add    $0x18,%rsp
  401061:   c3                      retq
```

Exactly as happened on phase 3 we can see that this phase is expecting two integers as input. On `0x40101a` the same format used before is stored on `esi` which is then used by `sscanf` on `0x401024` .

```
(gdb) p (char *) 0x4025cf
$1 = 0x4025cf "%d %d"
```

On `0x401029` we can also confirm that if we enter more than 2 integers the code will jump to `0x401035` that calls `explode_bomb` :

```
  401029:   83 f8 02                cmp    $0x2,%eax
  40102c:   75 07                   jne    401035 <phase_4+0x29>
```

So which exact numbers should we enter? It must be a number that is less than 15:

```
  40102e:   83 7c 24 08 0e          cmpl   $0xe,0x8(%rsp)
  401033:   76 05                   jbe    40103a <phase_4+0x2e>
  401035:   e8 00 04 00 00          callq  40143a <explode_bomb>
```

`cmpl` compares `0xe` which is 14 to the first integer[1] we entered for this phase. Then `jbe` ("jump below or equal") will skip exploding the bomb if the value is less than or equal to 14.

After that we can see that some setup is done before calling a new function, `func4` :

```
  40103a:   ba 0e 00 00 00          mov    $0xe,%edx
  40103f:   be 00 00 00 00          mov    $0x0,%esi
  401044:   8b 7c 24 08             mov    0x8(%rsp),%edi
  401048:   e8 81 ff ff ff          callq  400fce <func4>
```

`edi` is usually used as the register to hold the first argument, `esi` holds the second and `edx` holds the third argument. The first argument is the first number we provided, the second and third are `0` and `14` , respectively.

Let's take a look at `func4` :

```
0000000000400fce <func4>:
  400fce:   48 83 ec 08             sub    $0x8,%rsp
  400fd2:   89 d0                   mov    %edx,%eax
  400fd4:   29 f0                   sub    %esi,%eax
  400fd6:   89 c1                   mov    %eax,%ecx
  400fd8:   c1 e9 1f                shr    $0x1f,%ecx
  400fdb:   01 c8                   add    %ecx,%eax
  400fdd:   d1 f8                   sar    %eax
  400fdf:   8d 0c 30                lea    (%rax,%rsi,1),%ecx
  400fe2:   39 f9                   cmp    %edi,%ecx
  400fe4:   7e 0c                   jle    400ff2 <func4+0x24>
  400fe6:   8d 51 ff                lea    -0x1(%rcx),%edx
  400fe9:   e8 e0 ff ff ff          callq  400fce <func4>
  400fee:   01 c0                   add    %eax,%eax
  400ff0:   eb 15                   jmp    401007 <func4+0x39>
  400ff2:   b8 00 00 00 00          mov    $0x0,%eax
  400ff7:   39 f9                   cmp    %edi,%ecx
  400ff9:   7d 0c                   jge    401007 <func4+0x39>
  400ffb:   8d 71 01                lea    0x1(%rcx),%esi
  400ffe:   e8 cb ff ff ff          callq  400fce <func4>
  401003:   8d 44 00 01             lea    0x1(%rax,%rax,1),%eax
  401007:   48 83 c4 08             add    $0x8,%rsp
  40100b:   c3                      retq
```

Looking at the first few instructions we can try to write what exactly is happening with the arguments that were given to this function. The instructions until `0x400fe2` are actually doing something like the following:

```
int func4(int a, int b, int c) {
    int x = c - b; // 0x400fd2 and 0x400fd4
    int y = x >> 31; // 0x400fd6 and 0x400fd8
    x = x + y; // 0x400fdb
    x = x >> 1; // 0x400fdd
    y = x + b; // 0x400fdf
}
```

Then it compares `y` with `a` (our first input number for this phase) and if `y <= a` it will jump to `0x400ff2`. Otherwise it calls `func4` again but this time `c` will be `y - 1` (set at `0x400fe6`). So on `0x400fe9` we can think of the invocation as:

```
func4(a, b, y - 1);
```

After calling it you can see that `0x400fee` will double the result from that "inner" invocation (`eax` holds the return value from that execution) and then jump to `0x401007` which cleanups the stack frame for this invocation. So the result from this recursive call is actually:

```
return 2 * func4(a, b, y - 1);
```

If `y >= a` the code will continue execution on `0x400ff2`. At that line it sets the possible result as `0` and then compares the same `y` with `a` again. This time if `y >= a` it jumps to `0x401007` and returns `0`, that's why `eax` got the value `0` by the instruction before that. If `y < a` then `func4` will be called again but in this case `b` will be `y + 1` (this assignment happens on `0x400ffb`) and the following invocation happens on `0x400ffe`:

```
func4(a, y + 1, c);
```

After returning from this recursive call the return value is set on `0x401003` using the result from the recursive call so the result is actually:

```
return 2 * func4(a, y + 1, c) + 1;
```

With all this context we can now try to guess what exactly is going on with this function:

```
int func4(int a, int b, int c) {
    int x = c - b;
    int y = x >> 31;
    x = x + y;
    x = x >> 1;
    y = x + b;

    if (y <= a) {
        if (y >= a) {
            return 0;
        } else {
            return 2 * func4(a, y + 1, c) + 1;
        }
    } else {
        return 2 * func4(a, b, y - 1);
    }
}
```

Remember that the initial call to `func4` is `func4(ourFirstInputNumber, 0, 14)` and that `ourFirstInputNumber <= 14`. Let's try to see what happens if we input `1` as our first number:

```
func4(1, 0, 14) {
    int x = 14 - 0;
    int y = 14 >> 31;
    x = 14 + 0;
    x = 14 >> 1;
    y = 7 + 0;
}
```

Then we can see that `func4` will be called by the `else` clause of the first `if`:

```
return 2 * func4(1, 0, 7 - 1);
```

The first recursive call will then be:

```
func4(1, 0, 6) {
    int x = 6 - 0;
    int y = 6 >> 31;
    x = 6 + 0;
    x = 6 >> 1;
    y = 3 + 0;
}
```

Again the same branch will be taken:

```
return 2 * func4(1, 0, 3 - 1);
```

The execution will then be:

```
func4(1, 0, 2) {
    int x = 2 - 0;
    int y = 2 >> 31;
    x = 2 + 0;
    x = 2 >> 1;
    y = 1 + 0;
}
```

This time `y == 1` so both `if` branches will be taken and `0` will be returned from this recursive invocation. Remember that we invoked this function twice so the final result will be:

```
return func4(1, 0, 14);
    return 2 * func4(1, 0, 6);
        return 2 * func4(1, 0, 2);
            return 0;
```

Since the last call to `func4` returned `0` the final result will also be `0` and the execution will continue this time on `phase_4` at `0x40104d`:

```
  401048:   e8 81 ff ff ff          callq  400fce <func4>
  40104d:   85 c0                   test   %eax,%eax
  40104f:   75 07                   jne    401058 <phase_4+0x4c>
  401051:   83 7c 24 0c 00          cmpl   $0x0,0xc(%rsp)
  401056:   74 05                   je     40105d <phase_4+0x51>
  401058:   e8 dd 03 00 00          callq  40143a <explode_bomb>
  40105d:   48 83 c4 18             add    $0x18,%rsp
  401061:   c3                      retq
```

This line and the line below test whether or not the result of that `func4` invocation returned `0`, if it did our first input is correct and execution continues at `0x401051`. This instruction then simply checks if our second input is `0` and if it is the function returns and phase 4 is defused:

```
1 0
So you got that one.  Try this one.
```

We have something interesting here, remember that there are two checks about `y` and `a`? The second check uses the `jge` instruction that checks whether the value is greater than or equal to (kind of obvious if you think about what `ge` might mean in the instruction name). The interesting fact here is that the previous check also tested for equality with `jle`. So, if `y <= a` and then you check whether `y >= a` there's only one case that would satisfy both conditions and that is `y == a`. I'm not sure if the compiler chose `jge` even though the code was written as `if (y == a)` or if the code was actually written as `if (y >= a)`.

Another interesting thing about this phase is how the compiler manages the results of the recursive calls. Since `eax` is a register the results of each recursive invocation will be available to the stack frame that invoked it and then it can simply be returned without saving the result in some other place. Also you see that after calling `func4` again there's nothing managing the local variables `x` and `y` which is why they are also stored in registers instead of being saved inside each stack frame.

An exercise left to the reader is trying to find the other possible solutions for this phase including one input that doesn't even call `func4` recursively. Also try to see which numbers are invalid for this phase and the result that `func4` returns when these numbers are used.

Notes

1. `0x8(%rsp)` and `0xc(%rsp)` store both input numbers as local variables in the `phase_4` stack frame. ↩

Defusing a binary bomb with `gdb` - Part 5

28 Apr 2016

This post is part of a series where I show how to defuse a binary bomb by reading assembly code and using `gdb`. You might want to read the first part if you haven't yet.

After defusing the fourth phase the program continues to the next phase:

```
400ea2: e8 f7 05 00 00          callq  40149e <read_line>
400ea7: 48 89 c7                mov    %rax,%rdi
400eaa: e8 b3 01 00 00          callq  401062 <phase_5>
```

The code for `phase_5` is the following:

```
0000000000401062 <phase_5>:
  401062:   53                      push   %rbx
  401063:   48 83 ec 20             sub    $0x20,%rsp
  401067:   48 89 fb                mov    %rdi,%rbx
  40106a:   64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
  401071:   00 00
  401073:   48 89 44 24 18          mov    %rax,0x18(%rsp)
  401078:   31 c0                   xor    %eax,%eax
  40107a:   e8 9c 02 00 00          callq  40131b <string_length>
  40107f:   83 f8 06                cmp    $0x6,%eax
  401082:   74 4e                   je     4010d2 <phase_5+0x70>
  401084:   e8 b1 03 00 00          callq  40143a <explode_bomb>
  401089:   eb 47                   jmp    4010d2 <phase_5+0x70>
  40108b:   0f b6 0c 03             movzbl (%rbx,%rax,1),%ecx
  40108f:   88 0c 24                mov    %cl,(%rsp)
  401092:   48 8b 14 24             mov    (%rsp),%rdx
  401096:   83 e2 0f                and    $0xf,%edx
  401099:   0f b6 92 b0 24 40 00    movzbl 0x4024b0(%rdx),%edx
  4010a0:   88 54 04 10             mov    %dl,0x10(%rsp,%rax,1)
  4010a4:   48 83 c0 01             add    $0x1,%rax
  4010a8:   48 83 f8 06             cmp    $0x6,%rax
  4010ac:   75 dd                   jne    40108b <phase_5+0x29>
  4010ae:   c6 44 24 16 00          movb   $0x0,0x16(%rsp)
  4010b3:   be 5e 24 40 00          mov    $0x40245e,%esi
  4010b8:   48 8d 7c 24 10          lea    0x10(%rsp),%rdi
  4010bd:   e8 76 02 00 00          callq  401338 <strings_not_equal>
  4010c2:   85 c0                   test   %eax,%eax
  4010c4:   74 13                   je     4010d9 <phase_5+0x77>
  4010c6:   e8 6f 03 00 00          callq  40143a <explode_bomb>
  4010cb:   0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
  4010d0:   eb 07                   jmp    4010d9 <phase_5+0x77>
  4010d2:   b8 00 00 00 00          mov    $0x0,%eax
  4010d7:   eb b2                   jmp    40108b <phase_5+0x29>
  4010d9:   48 8b 44 24 18          mov    0x18(%rsp),%rax
  4010de:   64 48 33 04 25 28 00    xor    %fs:0x28,%rax
  4010e5:   00 00
  4010e7:   74 05                   je     4010ee <phase_5+0x8c>
  4010e9:   e8 42 fa ff ff          callq  400b30 <__stack_chk_fail@p
lt>
  4010ee:   48 83 c4 20             add    $0x20,%rsp
  4010f2:   5b                      pop    %rbx
  4010f3:   c3                      retq
```

The first few lines setup the stack for this function and on `0x40106a` the stack protector[1] is setup.

On `0x40107a` a new function `string_length` is called to check the length of the input we gave to `phase_5`. Remember that the argument for `phase_5` is stored in the `rdi` register and the same value is available when `string_length` is called. The input must be exactly six characters as shown by the comparison on `0x40107f`. If the input length is correct it then jumps to `0x4010d2` that sets `rax` to `0` and then jumps to `0x40108b` to continue executing this phase.

On `0x40108b` the first byte from the input string is copied to `ecx`. Notice that at the start of this function the input string stored on `rdi` was copied to `rbx` at `0x401067`. The instruction at `0x40108b` uses both `rbx` and `rax` but since `rax` has a value of `0` the data address used as source will be solely the address stored on `rbx`. We can confirm it with:

```
(gdb) p (char *) $rbx
$1 = 0x6038c0 <input_strings+320> "input5"
```

And to see each byte as hex:

```
(gdb) x/6xb $rbx
0x6038c0 <input_strings+320>:   0x69    0x6e    0x70    0x75    0x74
0x35
```

Stepping to next instruction we can also confirm that the byte from the first character is stored on `ecx`:

```
(gdb) i r $ecx
ecx            0x69 105
```

`0x69` is `i` in the ASCII table.

Then on the next two instructions this byte is copied to `rdx` and on `0x401096` a bitwise AND is performed against this byte. The value after the bitwise AND is then used as an offset to copy a value from some location to the same `edx` register on `0x401099`. That same 1-byte value is then stored in a local variable on `0x4010a0`. The next instruction increments `rax` which is then checked by the next instruction to see if this process was executed for all characters from our input. If the process wasn't performed for all characters it then jumps again to `0x40108b` to read the next character and repeat the steps above.

At the end of this process a local variable will hold a new string that is created by using our input characters as an offset to a mysterious string.

If the process was executed for all characters it then prepares the arguments for the next function call, starting at `0x4010ae`. This new string we're talking about will be used as an argument to the `strings_not_equal` function that we've seen before in this series. The other string that ours will be compared to is located at `0x40245e` as you can see on `0x4010b3`.

After calling `strings_not_equal` the result will be tested on `0x4010c2` and if they are equal the code will jump to the end of `phase_5` at `0x4010d9` that will check if stack wasn't corrupted and return. If the strings are different you know what's going to happen.

Now that we know the flow for this phase it's time to see which string our input must produce.

```
(gdb) p (char *) 0x40245e
$2 = 0x40245e "flyers"
```

Now all that we need is to look at the intermediary string to see which are the offsets we need to input so that the final string will be `flyers`.

```
(gdb) p (char *) 0x4024b0
$3 = 0x4024b0 <array> "maduiersnfotvbylSo you think you can stop the
bomb with ctrl-c, do you?"
```

Looking at the output above we can guess that the actual intermediary string must be just `maduiersnfotvbyl`. Looking at the position of each character we can then check that the correct offsets must be:

letter	offset
f	0x9
l	0xf
y	0xe
e	0x5
r	0x6
s	0x7

So the answer is actually any sequence of characters which their byte representations end in `9FE567`.

Considering just the printable characters from the ASCII table we can devise the following table:

offset	possible chars
0x9	) 9 I Y i y
0xf	/ ? O _ o DEL
0xe	. > N ^ n ~

0x5	% 5 E U e u
0x6	& 6 F V f v
0x7	' 7 G W g w

Any combination of a single char for each offset will defuse `phase_5`.

The sixth and final phase will be covered in the next post.

Notes

1. https://en.wikipedia.org/wiki/Buffer_overflow_protection ↩

Defusing a binary bomb with `gdb` - Part 6

19 May 2016

This post is part of a series where I show how to defuse a binary bomb by reading assembly code and using `gdb`. You might want to read the first part if you haven't yet.

Here we are at the last phase. This is the most interesting phase so far. The code for `phase_6` is the following:

```
00000000004010f4 <phase_6>:
  4010f4:   41 56                   push   %r14
  4010f6:   41 55                   push   %r13
  4010f8:   41 54                   push   %r12
  4010fa:   55                      push   %rbp
  4010fb:   53                      push   %rbx
  4010fc:   48 83 ec 50             sub    $0x50,%rsp
  401100:   49 89 e5                mov    %rsp,%r13
  401103:   48 89 e6                mov    %rsp,%rsi
  401106:   e8 51 03 00 00          callq  40145c <read_six_numbers>
  40110b:   49 89 e6                mov    %rsp,%r14
  40110e:   41 bc 00 00 00 00       mov    $0x0,%r12d
  401114:   4c 89 ed                mov    %r13,%rbp
  401117:   41 8b 45 00             mov    0x0(%r13),%eax
  40111b:   83 e8 01                sub    $0x1,%eax
  40111e:   83 f8 05                cmp    $0x5,%eax
  401121:   76 05                   jbe    401128 <phase_6+0x34>
  401123:   e8 12 03 00 00          callq  40143a <explode_bomb>
  401128:   41 83 c4 01             add    $0x1,%r12d
  40112c:   41 83 fc 06             cmp    $0x6,%r12d
  401130:   74 21                   je     401153 <phase_6+0x5f>
  401132:   44 89 e3                mov    %r12d,%ebx
  401135:   48 63 c3                movslq %ebx,%rax
  401138:   8b 04 84                mov    (%rsp,%rax,4),%eax
  40113b:   39 45 00                cmp    %eax,0x0(%rbp)
  40113e:   75 05                   jne    401145 <phase_6+0x51>
  401140:   e8 f5 02 00 00          callq  40143a <explode_bomb>
  401145:   83 c3 01                add    $0x1,%ebx
  401148:   83 fb 05                cmp    $0x5,%ebx
  40114b:	7e e8                	jle    401135 <phase_6+0x41>
  40114d:	49 83 c5 04          	add    $0x4,%r13
  401151:	eb c1                	jmp    401114 <phase_6+0x20>
  401153:	48 8d 74 24 18       	lea    0x18(%rsp),%rsi
  401158:	4c 89 f0             	mov    %r14,%rax
  40115b:	b9 07 00 00 00       	mov    $0x7,%ecx
  401160:	89 ca                	mov    %ecx,%edx
  401162:	2b 10                	sub    (%rax),%edx
  401164:	89 10                	mov    %edx,(%rax)
  401166:	48 83 c0 04          	add    $0x4,%rax
  40116a:	48 39 f0             	cmp    %rsi,%rax
  40116d:	75 f1                	jne    401160 <phase_6+0x6c>
  40116f:	be 00 00 00 00       	mov    $0x0,%esi
  401174:	eb 21                	jmp    401197 <phase_6+0xa3>
  401176:	48 8b 52 08          	mov    0x8(%rdx),%rdx
  40117a:	83 c0 01             	add    $0x1,%eax
  40117d:	39 c8                	cmp    %ecx,%eax
  40117f:	75 f5                	jne    401176 <phase_6+0x82>
  401181:	eb 05                	jmp    401188 <phase_6+0x94>
  401183:	ba d0 32 60 00       	mov    $0x6032d0,%edx
  401188:	48 89 54 74 20       	mov    %rdx,0x20(%rsp,%rsi,2)
  40118d:	48 83 c6 04          	add    $0x4,%rsi
  401191:	48 83 fe 18          	cmp    $0x18,%rsi
  401195:	74 14                	je     4011ab <phase_6+0xb7>
  401197:	8b 0c 34             	mov    (%rsp,%rsi,1),%ecx
  40119a:	83 f9 01             	cmp    $0x1,%ecx
  40119d:	7e e4                	jle    401183 <phase_6+0x8f>
  40119f:	b8 01 00 00 00       	mov    $0x1,%eax
  4011a4:	ba d0 32 60 00       	mov    $0x6032d0,%edx
  4011a9:	eb cb                	jmp    401176 <phase_6+0x82>
  4011ab:	48 8b 5c 24 20       	mov    0x20(%rsp),%rbx
  4011b0:	48 8d 44 24 28       	lea    0x28(%rsp),%rax
  4011b5:	48 8d 74 24 50       	lea    0x50(%rsp),%rsi
  4011ba:	48 89 d9             	mov    %rbx,%rcx
  4011bd:	48 8b 10             	mov    (%rax),%rdx
  4011c0:	48 89 51 08          	mov    %rdx,0x8(%rcx)
  4011c4:	48 83 c0 08          	add    $0x8,%rax
  4011c8:	48 39 f0             	cmp    %rsi,%rax
  4011cb:	74 05                	je     4011d2 <phase_6+0xde>
  4011cd:	48 89 d1             	mov    %rdx,%rcx
  4011d0:	eb eb                	jmp    4011bd <phase_6+0xc9>
  4011d2:	48 c7 42 08 00 00 00 	movq   $0x0,0x8(%rdx)
  4011d9:	00
  4011da:	bd 05 00 00 00       	mov    $0x5,%ebp
  4011df:	48 8b 43 08          	mov    0x8(%rbx),%rax
  4011e3:	8b 00                	mov    (%rax),%eax
  4011e5:	39 03                	cmp    %eax,(%rbx)
  4011e7:	7d 05                	jge    4011ee <phase_6+0xfa>
  4011e9:	e8 4c 02 00 00       	callq  40143a <explode_bomb>
  4011ee:	48 8b 5b 08          	mov    0x8(%rbx),%rbx
  4011f2:	83 ed 01             	sub    $0x1,%ebp
  4011f5:   75 e8                   jne    4011df <phase_6+0xeb>
  4011f7:   48 83 c4 50             add    $0x50,%rsp
  4011fb:   5b                      pop    %rbx
  4011fc:   5d                      pop    %rbp
  4011fd:   41 5c                   pop    %r12
  4011ff:   41 5d                   pop    %r13
  401201:   41 5e                   pop    %r14
  401203:   c3                      retq
```

It's longer than the other phases and seems more complicated so we're going to break it in parts to explain what each part is doing.

The first part we can look at is where the function initializes. It starts by saving some registers values because they are going to be used as local variables in this function, then making room for other local variables and then reading the input that will be used to defuse the phase. At `0x401106` we can see that the input for this phase must be six numbers:

```
4010f4: 41 56                   push   %r14
4010f6: 41 55                   push   %r13
4010f8: 41 54                   push   %r12
4010fa: 55                      push   %rbp
4010fb: 53                      push   %rbx
4010fc: 48 83 ec 50             sub    $0x50,%rsp
401100: 49 89 e5                mov    %rsp,%r13
401103: 48 89 e6                mov    %rsp,%rsi
401106: e8 51 03 00 00          callq  40145c <read_six_numbers>
40110b: 49 89 e6                mov    %rsp,%r14
40110e: 41 bc 00 00 00 00       mov    $0x0,%r12d
401114: 4c 89 ed                mov    %r13,%rbp
401117: 41 8b 45 00             mov    0x0(%r13),%eax
40111b: 83 e8 01                sub    $0x1,%eax
40111e: 83 f8 05                cmp    $0x5,%eax
401121: 76 05                   jbe    401128 <phase_6+0x34>
401123: e8 12 03 00 00          callq  40143a <explode_bomb>
```

After reading the six numbers and placing the first one on `rsp` the code copies the address pointing to the first one into `r14` then it sets up other variables and finally on `0x40111e` it checks whether or not the first number we provided is less than or equal to `6`. How it did that?

`rsi` is used to hold the second argument for a function call and prior to calling `read_six_numbers` (at `0x401103`) the address of `rsp` was copied to `rsi` to be used by `read_six_numbers`. That's where our numbers were stored, in an array that starts at the address that is on `rsi`. This same address is also stored on `rsp` and `r13`. We can look at the registers to see which address this is:

```
(gdb) i r rsp rsi r13
rsp            0x7fffffffdd60   0x7fffffffdd60
rsi            0x7fffffffdd60   140737488346464
r13            0x7fffffffdd60   140737488346464
```

After returning from `read_six_numbers` this same address is stored on `r14` at `0x40110b` and on `0x40114` it is stored on `rbp` as well.

Then on `0x401117` the value stored in the address on `r13`, our first number, is copied to `eax` and then the code checks if it is less than or equal to `6`.

So now that we understand how the check was made, let's proceed to the next part:

```
401128: 41 83 c4 01             add    $0x1,%r12d
40112c: 41 83 fc 06             cmp    $0x6,%r12d
401130: 74 21                   je     401153 <phase_6+0x5f>
401132: 44 89 e3                mov    %r12d,%ebx
401135: 48 63 c3                movslq %ebx,%rax
401138: 8b 04 84                mov    (%rsp,%rax,4),%eax
40113b: 39 45 00                cmp    %eax,0x0(%rbp)
40113e: 75 05                   jne    401145 <phase_6+0x51>
401140: e8 f5 02 00 00          callq  40143a <explode_bomb>
```

Remember that on line `0x40110e` the register `r12d` stored the value `0` so the first three lines are just for checking if the code already went through 6 iterations. Let's continue on `0x401132` where the new value of `r12d` (which is `1` for the first iteration) is copied into `ebx` which then is copied to `rax` by sign-extending from double word (4 bytes) to quad word (8 bytes) since `ebx` is a 32-bit register while `rax` is a 64-bit register.

After that the next number we entered is checked against the first one. The second number is copied to `eax` by this instruction:

```
401138: 8b 04 84                mov    (%rsp,%rax,4),%eax
```

What this line actually does is: multiply by 4 the value on `rax` (`1` since it came from `r12d`) and add that value to value stored on `rsp` (the starting address of the array holding our input numbers). For the first iteration the resulting address will be:

```
(gdb) x $rsp+$rax*0x4
0x7fffffffdd64: 0x00000002
```

The value stored at this resulting address will then be copied to `eax`. For the first iteration it means our second input number.

Then our second value (on `eax`) is compared to first one to see if they are not equal and jumps to the next part:

```
401145: 83 c3 01                add    $0x1,%ebx
401148: 83 fb 05                cmp    $0x5,%ebx
40114b: 7e e8                   jle    401135 <phase_6+0x41>
40114d: 49 83 c5 04             add    $0x4,%r13
401151: eb c1                   jmp    401114 <phase_6+0x20>
```

The first three lines will check if we did this check for all six numbers which means we cannot input repeated numbers. After that, on `0x40114d`, `r13` is changed to hold the address of the second input number by adding 4 bytes (`sizeof(int)`) to the address `r13` is currently storing. Then it goes back to `0x401114` to do the same checks against the other numbers we provided.

Now we know some facts about the expected input:

- It must be six numbers;
- They need to be less than or equal to `6`;
- They cannot repeat.

Let's continue to see which other characteristics these numbers must have.

After doing these initial checks the code will jump to `0x401153`:

```
401153: 48 8d 74 24 18          lea    0x18(%rsp),%rsi
401158: 4c 89 f0                mov    %r14,%rax
```

```
  40115b: b9 07 00 00 00          mov    $0x7,%ecx
  401160: 89 ca                   mov    %ecx,%edx
  401162: 2b 10                   sub    (%rax),%edx
  401164: 89 10                   mov    %edx,(%rax)
  401166: 48 83 c0 04             add    $0x4,%rax
  40116a: 48 39 f0                cmp    %rsi,%rax
  40116d: 75 f1                   jne    401160 <phase_6+0x6c>
  40116f: be 00 00 00 00          mov    $0x0,%esi
  401174: eb 21                   jmp    401197 <phase_6+0xa3>
```

The first line of this part simply defines the address that means we iterated over all six numbers. The first number is stored on the address that `rsp` holds and the sixth number will be on `$rsp + 0x14` (start address + offset of 5 `int`).

`r14` also holds the address for the first number so it's going to be copied to `rax` on `0x401158` to be used in this iteration. Then on the next two lines both `ecx` and `edx` store the value `7`. After the setup the actual iteration will start by first subtracting from `edx` the value stored by the address in `rax` (our first number in the first iteration). At `0x401164` the result of this subtraction will overwrite the value on `rax` and then on `0x401166` the code will move to the next `int` we provided, compare on `0x40116d` if we iterated over all six numbers and jump back to `0x401160` if we did not, otherwise get out of the loop and continue execution on `0x401197`.

Let's simulate what happens after this loop is executed. Suppose we entered `1 2 3 4 5 6` as our input numbers for this phase. Then after iterating in this loop our array will have the following new values: `6 5 4 3 2 1`. The loop just changes the all numbers in the array to be `abs(n-7)`.

After exiting the loop we continue on `0x401197` which brings us to the next part in this phase:

```
  401176: 48 8b 52 08             mov    0x8(%rdx),%rdx
  40117a: 83 c0 01                add    $0x1,%eax
  40117d: 39 c8                   cmp    %ecx,%eax
  40117f: 75 f5                   jne    401176 <phase_6+0x82>
  401181: eb 05                   jmp    401188 <phase_6+0x94>
  401183: ba d0 32 60 00          mov    $0x6032d0,%edx
  401188: 48 89 54 74 20          mov    %rdx,0x20(%rsp,%rsi,2)
  40118d: 48 83 c6 04             add    $0x4,%rsi
  401191: 48 83 fe 18             cmp    $0x18,%rsi
  401195: 74 14                   je     4011ab <phase_6+0xb7>
  401197: 8b 0c 34                  mov    (%rsp,%rsi,1),%ecx
  40119a: 83 f9 01                  cmp    $0x1,%ecx
  40119d: 7e e4                     jle    401183 <phase_6+0x8f>
  40119f: b8 01 00 00 00            mov    $0x1,%eax
  4011a4: ba d0 32 60 00            mov    $0x6032d0,%edx
  4011a9: eb cb                     jmp    401176 <phase_6+0x82>
```

Although the first line of this part is at `0x401176`, execution actually starts at `0x401197`. After executing this line `ecx` will hold the first number from our array because the value stored on `rsi` is `0` (from `0x40116f`) and the instruction on `0x401197` means: copy the value stored by the address of `$rsi*0x1 + $rsp` to `ecx`. This operation clearly means it is part of some iteration and we can guess that `rsi` will be updated in the process to go over the other numbers in the array.

```
  401197: 8b 0c 34                  mov    (%rsp,%rsi,1),%ecx
  40119a: 83 f9 01                  cmp    $0x1,%ecx
  40119d: 7e e4                     jle    401183 <phase_6+0x8f>
  40119f: b8 01 00 00 00            mov    $0x1,%eax
  4011a4: ba d0 32 60 00            mov    $0x6032d0,%edx
  4011a9: eb cb                     jmp    401176 <phase_6+0x82>
```

If the current number on `ecx` is less than or equal to `1` the code will jump to `0x401183`, otherwise it will jump to `0x401176`.

In the last part our array became: `6 5 4 3 2 1`. So the code will jump to `0x401176`. Let's see what happens there.

```
  401176: 48 8b 52 08               mov    0x8(%rdx),%rdx
  40117a: 83 c0 01                  add    $0x1,%eax
  40117d: 39 c8                     cmp    %ecx,%eax
  40117f: 75 f5                     jne    401176 <phase_6+0x82>
  401181: eb 05                     jmp    401188 <phase_6+0x94>
```

Notice that before jumping to `0x401176`, the address `0x6032d0` was stored on `edx`. Then the value stored after the first 8 bytes of this address will be copied to `rdx` on `0x401176`. After this operation, `eax` will be incremented and compared with `ecx` to then conditionally jump to another place in this part or again to `0x401176`. The initial value of `eax` in this case is `1` that was set on `0x40119f` before jumping to `0x401176`. `ecx` holds the value `6` so we go back to `0x401176` and copy the value on `$rdx + 0x8` to `rdx`, increment `eax` and check against `ecx`.

The values on `ecx` and `eax` will match only after six iterations: in our example, `ecx` starts with `6` and `eax` with `1`. In this case before jumping to `0x401188`, `rdx` will have whatever value is stored in `$rdx + 0x48` (six times adding `0x8` to the initial address and copying the value in the new address to `rdx`).

Then the code jumps to `0x401188`:

```
401183: ba d0 32 60 00          mov    $0x6032d0,%edx
401188: 48 89 54 74 20          mov    %rdx,0x20(%rsp,%rsi,2)
40118d: 48 83 c6 04             add    $0x4,%rsi
401191: 48 83 fe 18             cmp    $0x18,%rsi
401195: 74 14                   je     4011ab <phase_6+0xb7>
401197: 8b 0c 34                mov    (%rsp,%rsi,1),%ecx
40119a: 83 f9 01                cmp    $0x1,%ecx
40119d: 7e e4                   jle    401183 <phase_6+0x8f>
40119f: b8 01 00 00 00          mov    $0x1,%eax
4011a4: ba d0 32 60 00          mov    $0x6032d0,%edx
4011a9: eb cb                   jmp    401176 <phase_6+0x82>
```

At this line the address that `rdx` is holding will be copied to the address that results from: `$rsi*0x2 + $rsp + 0x20`. `rsi` is the index over the iteration that is going on:

```
(gdb) i r rsi
rsi            0x0  0
```

Next, `rsi` is incremented by `4` (`sizeof(int)`) and compared against `0x18` to see if we iterated over all six numbers. If we did not then on `0x401197`, `ecx` gets the next number from our array and the next iteration begins.

Now that we know what this iteration is all about let's see what exactly is stored by the initial address on `rdx`:

```
(gdb) x 0x6032d0
0x6032d0 <node1>:   0x0000014c
```

Huh, `node1`, interesting name, right? Let's see what this address plus 8 bytes holds:

```
(gdb) x 0x6032d0+0x8
0x6032d8 <node1+8>: 0x006032e0
```

Looking at what is on `0x6032e0`:

```
(gdb) x 0x6032e0
0x6032e0 <node2>:   0x000000a8
```

Aha! That looks like a linked list. To see the address of `node3`:

```
(gdb) x 0x6032e0+0x8
0x6032e8 <node2+8>: 0x006032f0
```

And what `node3` stores:

```
(gdb) x *(0x6032e0+0x8)
0x6032f0 <node3>:   0x0000039c
```

We have a linked list that holds an `int` value, the node identifier and the pointer to the next node, something like the following:

```
struct node {
    int x;
    int i;
    struct node *next;
};
```

So when the code jumps to `0x401188`, `rdx` will have the address of the value stored by `node6` since our array is `6 5 4 3 2 1` and it went through

`0x401176` six times:

```
(gdb) i r rdx
rdx            0x603320 6304544
(gdb) x 0x6032d0+0x48
0x603318 <node5+8>: 0x00603320
(gdb) x $rdx
0x603320 <node6>:   0x000001bb
```

The address `rdx` holds, which stores the value `0x1bb`, will be placed in the first position of a new array starting at `$rsp + 0x20`. Since this code iterates over all six numbers, after executing this part for all numbers this new array will actually store the addresses holding the values that the corresponding node stores, based on our transformed input array.

Explaining: from `0x401176` until `0x401181` the code is looking for the corresponding node for the current iteration. On `0x401188` the address of value `x` that the node holds is then copied to the current iteration index on the new array. Then the next iteration begins.

Let's see what are the values that each node stores and their addresses. First we define a command to print the node values and move to the next node:

```
define plist
 set var $n = $arg0
 while $n
  printf "node%d (%p): value = %#.3x, next=%p\n", *($n+0x4), $n, *$n,
*($n+0x8)
  set var $n = *($n+0x8)
 end
end
```

Printing the values:

```
(gdb) plist 0x6032d0
node1 (0x6032d0): value = 0x14c, next=0x6032e0
node2 (0x6032e0): value = 0x0a8, next=0x6032f0
node3 (0x6032f0): value = 0x39c, next=0x603300
node4 (0x603300): value = 0x2b3, next=0x603310
node5 (0x603310): value = 0x1dd, next=0x603320
node6 (0x603320): value = 0x1bb, next=(nil)
```

After iterating over the six numbers using our input array which was transformed into `6 5 4 3 2 1` the new array (starting at `$rsp + 0x20`) will hold the addresses of `x` in the following order: `node6 node5 node4 node3 node2 node1`.

```
(gdb) x/6gx $rsp+0x20
0x7fffffffdd80: 0x0000000000603320  0x0000000000603310
0x7fffffffdd90: 0x0000000000603300  0x00000000006032f0
0x7fffffffdda0: 0x00000000006032e0  0x00000000006032d0
```

After creating this new array the code continues execution at `0x4011ab` which is the next part:

```
4011ab: 48 8b 5c 24 20          mov    0x20(%rsp),%rbx
4011b0: 48 8d 44 24 28          lea    0x28(%rsp),%rax
4011b5: 48 8d 74 24 50          lea    0x50(%rsp),%rsi
4011ba: 48 89 d9                mov    %rbx,%rcx
4011bd: 48 8b 10                mov    (%rax),%rdx
4011c0: 48 89 51 08             mov    %rdx,0x8(%rcx)
4011c4: 48 83 c0 08             add    $0x8,%rax
4011c8: 48 39 f0                cmp    %rsi,%rax
4011cb: 74 05                   je     4011d2 <phase_6+0xde>
4011cd: 48 89 d1                mov    %rdx,%rcx
4011d0: eb eb                   jmp    4011bd <phase_6+0xc9>
4011d2: 48 c7 42 08 00 00 00    movq   $0x0,0x8(%rdx)
4011d9: 00
4011da: bd 05 00 00 00          mov    $0x5,%ebp
4011df: 48 8b 43 08             mov    0x8(%rbx),%rax
4011e3: 8b 00                   mov    (%rax),%eax
4011e5: 39 03                   cmp    %eax,(%rbx)
4011e7: 7d 05                   jge    4011ee <phase_6+0xfa>
4011e9: e8 4c 02 00 00          callq  40143a <explode_bomb>
4011ee: 48 8b 5b 08             mov    0x8(%rbx),%rbx
4011f2: 83 ed 01                sub    $0x1,%ebp
4011f5: 75 e8                   jne    4011df <phase_6+0xeb>
```

At the first line of this part `rbx` gets the value of the first element of the new array which is the address of `x` for `node6`. In the next line `rax` gets the second element of the new array and in the third line `rsi` gets an address that is just past the last element of this new array, most likely to check later if we iterated over the entire new array.

On `0x4011ba`, `rcx` will hold the value of the first element of the new array, then `rdx` will get the second value and in the next line, `0x4011c0`, this value will be copied to `$rcx + 0x8`. Let's back it up for a minute and see the values in both registers after executing the instruction at `0x4011c0`:

```
(gdb) i r rcx rdx
rcx            0x603320 6304544
rdx            0x603310 6304528
```

`$rcx + 0x8` is the address that holds the pointer to the `next` element of the list and after executing `0x4011c0` it will point to the value that `rdx` is storing:

```
(gdb) x $rcx+0x8
0x603328 <node6+8>: 0x0000000000603310
```

After that, on `0x4011c4` the next value from the new array is placed in `rax` for the next iteration which is checked against `rsi` in the next line and the process repeats. At the end of it the `next` pointer for each `node` will be changed according to the values of the new array. As we've seen above the new array has the following values:

```
(gdb) x/6gx $rsp+0x20
0x7fffffffdd80: 0x0000000000603320  0x0000000000603310
0x7fffffffdd90: 0x0000000000603300  0x00000000006032f0
0x7fffffffdda0: 0x00000000006032e0  0x00000000006032d0
```

These values correspond to `node6 node5 node4 node3 node2 node1` so after the iteration above the pointers will be changed to reflect this order. We can confirm it with our `plist` command after executing the instruction on `0x4011d2`:

```
(gdb) plist 0x603320
node6 (0x603320): value = 0x1bb, next=0x603310
node5 (0x603310): value = 0x1dd, next=0x603300
node4 (0x603300): value = 0x2b3, next=0x6032f0
node3 (0x6032f0): value = 0x39c, next=0x6032e0
node2 (0x6032e0): value = 0x0a8, next=0x6032d0
```

```
node1 (0x6032d0): value = 0x14c, next=(nil)
```

With the initial input of `1 2 3 4 5 6` the code has reversed the list. Note that I now used the address of `node6` as the starting address for `plist` since that is the order from the array located at `$rsp + 0x20`. Also notice that on `0x4011d2` the `next` pointer for the last iteration receives the `NULL` value and in our case it is `node1->next`.

```
4011da: bd 05 00 00 00          mov    $0x5,%ebp
4011df: 48 8b 43 08             mov    0x8(%rbx),%rax
4011e3: 8b 00                   mov    (%rax),%eax
4011e5: 39 03                   cmp    %eax,(%rbx)
4011e7: 7d 05                   jge    4011ee <phase_6+0xfa>
4011e9: e8 4c 02 00 00          callq  40143a <explode_bomb>
4011ee: 48 8b 5b 08             mov    0x8(%rbx),%rbx
4011f2: 83 ed 01                sub    $0x1,%ebp
4011f5: 75 e8                   jne    4011df <phase_6+0xeb>
```

Now that the list was changed the execution continues at `0x4011da` by storing the value `5` in `ebp`. Then the address of value `x` of the next node is stored on `rax` and then in the next line the actual `x` value is stored on `eax` (32-bit register since we're dealing with `int`). On `0x4011e5` the value `x` from the first node (at `rbx`) is compared against the value `x` of the second and if the first value is greater than or equal to the second the code jumps to `0x4011ee` that will update the value of `rbx` to point to the next `x` value and also update the value of `ebp` that is then compared to see if we iterated over the entire list.

So this iteration is checking that the `x` value of the current node is greater than or equal to the next `x` for all nodes in the list. If they are not the bomb explodes as we can see at `0x4011e9`.

The other part of this function starting at `0x4011f7` cleanups the stack frame for `phase_6` and returns.

Now that we saw everything that this function is doing we know that **our input numbers are used to sort the linked list in descending order.**

Don't forget that before being used to reorder the list each input number will be changed to `abs(n-7)`.

The values of each node are:

node #	x (hex)	x (dec)
1	0x14c	332
2	0x0a8	168
3	0x39c	924
4	0x2b3	691
5	0x1dd	477
6	0x1bb	443

The values of `x` for the final list must be in the following order:

```
924 -> 691 -> 477 -> 443 -> 332 -> 168
```

Which means the list must be reordered as:

```
node3 -> node4 -> node5 -> node6 -> node1 -> node2
```

Then the solution for this phase is:

```
4 3 2 1 6 5
Congratulations! You've defused the bomb!
```

Finally! The bomb has been defused!

Or not? There's something odd in the C file. The `main` function ends like this:

```
input = read_line();
phase_6(input);
phase_defused();

/* Wow, they got it!  But isn't something... missing?  Perhaps
 * something they overlooked?  Mua ha ha ha ha! */

return 0;
```

Defusing a binary bomb with `gdb` - Part 7

24 May 2016

This post is part of a series where I show how to defuse a binary bomb by reading assembly code and using `gdb`. You might want to read the first part if you haven't yet.

This is it. We are finally in the last post of the series.

In the last post the bomb was defused but there was something odd in the C file. This was the end of the `main` function:

```
input = read_line();
phase_6(input);
phase_defused();

/* Wow, they got it!  But isn't something... missing?  Perhaps
 * something they overlooked?  Mua ha ha ha ha! */

return 0;
```

And I left a hint in the last post about a call to a `secret_phase` function from `phase_defused`.

```
401630: e8 0d fc ff ff          callq  401242 <secret_phase>
```

`phase_defused` is a function that is called every time a phase is defused so let's take a look at it:

```
00000000004015c4 <phase_defused>:
  4015c4:   48 83 ec 78             sub    $0x78,%rsp
  4015c8:   64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
  4015cf:   00 00
  4015d1:   48 89 44 24 68          mov    %rax,0x68(%rsp)
  4015d6:   31 c0                   xor    %eax,%eax
  4015d8:   83 3d 81 21 20 00 06    cmpl   $0x6,0x202181(%rip)
# 603760 <num_input_strings>
  4015df:   75 5e                   jne    40163f <phase_defused+0x7b
>
  4015e1:   4c 8d 44 24 10          lea    0x10(%rsp),%r8
  4015e6:   48 8d 4c 24 0c          lea    0xc(%rsp),%rcx
  4015eb:   48 8d 54 24 08          lea    0x8(%rsp),%rdx
  4015f0:   be 19 26 40 00          mov    $0x402619,%esi
  4015f5:   bf 70 38 60 00          mov    $0x603870,%edi
  4015fa:   e8 f1 f5 ff ff          callq  400bf0 <__isoc99_sscanf@pl
t>
  4015ff:   83 f8 03                cmp    $0x3,%eax
  401602:   75 31                   jne    401635 <phase_defused+0x71
>
  401604:   be 22 26 40 00          mov    $0x402622,%esi
  401609:   48 8d 7c 24 10          lea    0x10(%rsp),%rdi
  40160e:   e8 25 fd ff ff          callq  401338 <strings_not_equal>
  401613:   85 c0                   test   %eax,%eax
  401615:   75 1e                   jne    401635 <phase_defused+0x71
>
  401617:   bf f8 24 40 00          mov    $0x4024f8,%edi
  40161c:   e8 ef f4 ff ff          callq  400b10 <puts@plt>
  401621:   bf 20 25 40 00          mov    $0x402520,%edi
  401626:   e8 e5 f4 ff ff          callq  400b10 <puts@plt>
  40162b:   b8 00 00 00 00          mov    $0x0,%eax
  401630:   e8 0d fc ff ff          callq  401242 <secret_phase>
  401635:   bf 58 25 40 00          mov    $0x402558,%edi
  40163a:   e8 d1 f4 ff ff          callq  400b10 <puts@plt>
  40163f:   48 8b 44 24 68          mov    0x68(%rsp),%rax
  401644:   64 48 33 04 25 28 00    xor    %fs:0x28,%rax
  40164b:   00 00
  40164d:   74 05                   je     401654 <phase_defused+0x90
>
  40164f:   e8 dc f4 ff ff          callq  400b30 <__stack_chk_fail@p
lt>
  401654:   48 83 c4 78             add    $0x78,%rsp
  401658:   c3                      retq
  401659:   90                      nop
  40165a:   90                      nop
  40165b:   90                      nop
  40165c:   90                      nop
  40165d:   90                      nop
  40165e:   90                      nop
  40165f:   90                      nop
```

The first few lines until `0x4015d8` are for setting the stack protector[1] and then saving `rax` before setting it to `0` on `0x4015d6`. Then on `0x4015d8` it compares if we already went through all the six phases by looking at the number of input strings. This comparison will only be true after defusing the six phases so before the last post the code always jumped to `0x40163f` that restored the value of `rax` and checked the stack protector before returning.

Now that the sixth phase was defused the code will continue on `0x4015e1`. In the next three lines, registers `r8`, `rcx` and `rdx` will store addresses to hold local variables and then on `0x4015f0` and `0x4015f5`, `esi` and `edi` receive two addresses. Let's look at them to see what is in each address:

```
(gdb) p (char *) 0x402619
$1 = 0x402619 "%d %d %s"
(gdb) p (char *) 0x603870
$2 = 0x603870 <input_strings+240> "1 0"
```

How could I guess they were strings? Look at the next line:

```
4015fa: e8 f1 f5 ff ff          callq  400bf0 <__isoc99_sscanf@plt>
```

`sscanf` has the following signature:

```
int sscanf(const char *str, const char *format, ...);
```

Both `edi` and `esi` are used to hold the first and second arguments, respectively. Then the only thing that both registers could store were addresses pointing to strings[2].

The value `1 0` on `edi` looks familiar? It should, because that is the answer for the fourth phase.

`sscanf` is then used to scan the input of the fourth phase again but with a different format now, expecting a string after the second value. You can see that on `0x4015ff` if it doesn't see 3 values it jumps to `0x401635` that will print the following message and return:

```
(gdb) p (char *) $edi
```

```
$3 = 0x402558 "Congratulations! You've defused the bomb!"
```

Otherwise this function will compare if we entered the correct string to activate the secret phase. Looking at the instruction on `0x401604` we can see what is the address of the activation string:

```
(gdb) p (char *) 0x402622
$4 = 0x402622 "DrEvil"
```

If the input for phase 4 includes `DrEvil` at the end the code will then call the secret phase on `0x401630`:

```
401630: e8 0d fc ff ff          callq  401242 <secret_phase>
```

Now we know how to get to the point of calling `secret_phase`. Let's take a look at the code for it:

```
0000000000401242 <secret_phase>:
  401242:   53                      push   %rbx
  401243:   e8 56 02 00 00          callq  40149e <read_line>
  401248:   ba 0a 00 00 00          mov    $0xa,%edx
  40124d:   be 00 00 00 00          mov    $0x0,%esi
  401252:   48 89 c7                mov    %rax,%rdi
  401255:   e8 76 f9 ff ff          callq  400bd0 <strtol@plt>
  40125a:   48 89 c3                mov    %rax,%rbx
  40125d:   8d 40 ff                lea    -0x1(%rax),%eax
  401260:   3d e8 03 00 00          cmp    $0x3e8,%eax
  401265:   76 05                   jbe    40126c <secret_phase+0x2a>
  401267:   e8 ce 01 00 00          callq  40143a <explode_bomb>
  40126c:   89 de                   mov    %ebx,%esi
  40126e:   bf f0 30 60 00          mov    $0x6030f0,%edi
  401273:   e8 8c ff ff ff          callq  401204 <fun7>
  401278:   83 f8 02                cmp    $0x2,%eax
  40127b:   74 05                   je     401282 <secret_phase+0x40>
  40127d:   e8 b8 01 00 00          callq  40143a <explode_bomb>
  401282:   bf 38 24 40 00          mov    $0x402438,%edi
  401287:   e8 84 f8 ff ff          callq  400b10 <puts@plt>
  40128c:   e8 33 03 00 00          callq  4015c4 <phase_defused>
  401291:   5b                      pop    %rbx
  401292:   c3                      retq
  401293:   90                      nop
  401294:   90                      nop
  401295:	90                   	nop
  401296:	90                   	nop
  401297:	90                   	nop
  401298:	90                   	nop
  401299:	90                   	nop
  40129a:	90                   	nop
  40129b:	90                   	nop
  40129c:	90                   	nop
  40129d:	90                   	nop
  40129e:	90                   	nop
  40129f:	90                   	nop
```

After reading the input for this secret phase on `0x401243`, `edx` will store the value `10`, `esi` will store `NULL` and `rdi` will store the input we gave. Then `strtol`[3] will be called on `0x401255` and judging by the values on the registers it means our input will be converted to a number in base 10. After that, on `0x40125a` the already converted number will be stored on `rbx`. On `0x40125d` the value will be modified by subtracting `1` and placed again on `eax`. Then a comparison against `0x3e8` which is `1000` in base 10. If our input is less than `1000` the code will jump to `0x40126c` to continue and if not the bomb will explode.

At `0x40126c` our value is placed on `esi` and `edi` gets an address. Both values will be used in the next function `fun7` that is called on `0x401273`. The result that `fun7` must return is `2` as you can see on `0x401278`.

Before looking at `fun7` let's look at what is in the address that `edi` received and is used as the first argument for `fun7`:

```
(gdb) x 0x6030f0
0x6030f0 <n1>:	0x00000024
```

Hmm, `n1`. This name doesn't give any clues about what exactly this is. Let's look at `fun7` then:

```
0000000000401204 <fun7>:
  401204:	48 83 ec 08          	sub    $0x8,%rsp
  401208:	48 85 ff             	test   %rdi,%rdi
  40120b:	74 2b                	je     401238 <fun7+0x34>
  40120d:	8b 17                	mov    (%rdi),%edx
  40120f:	39 f2                	cmp    %esi,%edx
  401211:   7e 0d                   jle    401220 <fun7+0x1c>
  401213:   48 8b 7f 08             mov    0x8(%rdi),%rdi
  401217:   e8 e8 ff ff ff          callq  401204 <fun7>
  40121c:   01 c0                   add    %eax,%eax
  40121e:   eb 1d                   jmp    40123d <fun7+0x39>
  401220:   b8 00 00 00 00          mov    $0x0,%eax
  401225:   39 f2                   cmp    %esi,%edx
  401227:   74 14                   je     40123d <fun7+0x39>
  401229:   48 8b 7f 10             mov    0x10(%rdi),%rdi
  40122d:   e8 d2 ff ff ff          callq  401204 <fun7>
  401232:   8d 44 00 01             lea    0x1(%rax,%rax,1),%eax
  401236:   eb 05                   jmp    40123d <fun7+0x39>
  401238:   b8 ff ff ff ff          mov    $0xffffffff,%eax
  40123d:   48 83 c4 08             add    $0x8,%rsp
  401241:   c3                      retq
```

It first checks whether `rdi` is `NULL` (`0x0`) and if it is the function will return the value `-1` (`0xffffffff` at `0x401238`). If not, the value stored in the address `rdi` is storing will be placed on `edx` and then compared (on `0x40120f`) with our input number. If the value on `edx` is less than or equal to our number the code goes to `0x401220`, if not it continues on `0x401213`. Let's continue on `0x401213` then we come back to see what happens in the other branch.

On `0x401213` `rdi` will store whatever is 8 bytes after the address already on `rdi` and call `fun7` again.

In the other branch, when the number on `edx` is less than or equal to our number, the code goes to `0x401220` that sets `eax` to `0`, compares what is on `edx` with our number and if they are equal it goes to `0x40123d` that returns. If the numbers are different then the instruction at `0x401229` will change `rdi` to store whatever is 16 bytes after the current address it has and then call `fun7` as the other branch does.

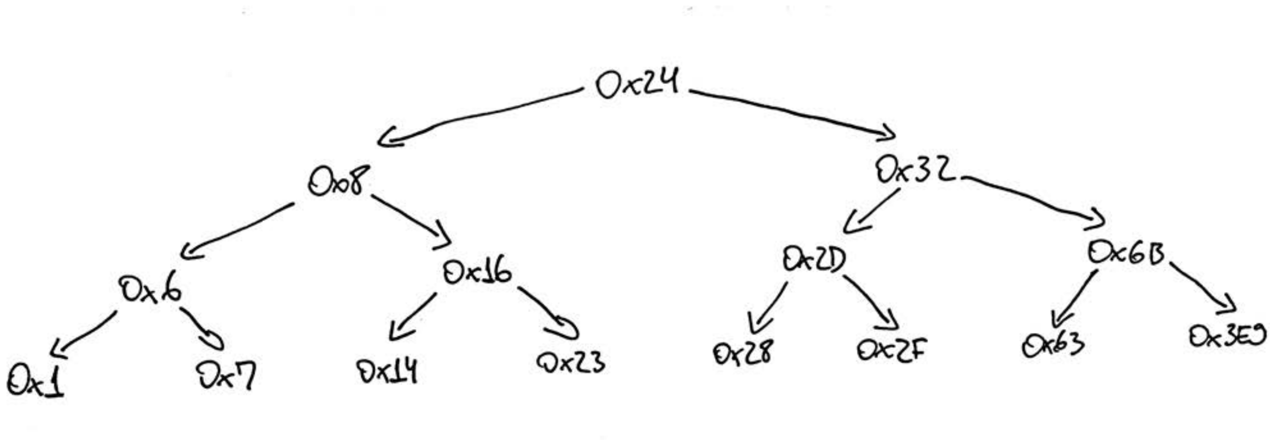
In each branch `rdi` is changed to hold the new address located either 8 or 16 bytes after its current address. The first thing that `rdi` holds is a number, the other two are pointers, so **this might be a binary tree**:

```
struct node {
    int data;
    struct node *left;
    struct node *right;
};
```

The root node, located at the initial address that `fun7` is called with is the following:

```
(gdb) x 0x6030f0
0x6030f0 <n1>:  0x00000024
(gdb) x 0x6030f0+0x8
0x6030f8 <n1+8>:    0x00603110
(gdb) x 0x6030f0+0x10
0x603100 <n1+16>:   0x00603130
```

After following the node pointers we can see that the tree structure is the following:



We know how the tree is structured but we don't know what we need to do to get to the right answer. `secret_phase` calls `fun7` and expects it to return the value `2`:

```
401273: e8 8c ff ff ff          callq  401204 <fun7>
401278: 83 f8 02                cmp    $0x2,%eax
40127b: 74 05                   je     401282 <secret_phase+0x40>
40127d: e8 b8 01 00 00          callq  40143a <explode_bomb>
401282: bf 38 24 40 00          mov    $0x402438,%edi
401287: e8 84 f8 ff ff          callq  400b10 <puts@plt>
40128c: e8 33 03 00 00          callq  4015c4 <phase_defused>
401291: 5b                      pop    %rbx
401292: c3                      retq
```

So let's see what `fun7` returns. We know it is a recursive function. Let's see what is the base case, which are actually two:

```
401208: 48 85 ff                    test   %rdi,%rdi
40120b: 74 2b                       je     401238 <fun7+0x34>
```

And:

```
401225: 39 f2                       cmp    %esi,%edx
401227: 74 14                       je     40123d <fun7+0x39>
```

Both show that the function returns when we reached a `NULL` pointer (first case) or if the number we gave is equal to the one in the current node (second case).

But in the first case the code jumps to `0x401238` that sets the return value to `-1`:

```
401238: b8 ff ff ff ff              mov    $0xffffffff,%eax
40123d: 48 83 c4 08                 add    $0x8,%rsp
401241: c3                          retq
```

The second case jumps straight to `0x40123d` but before comparing the numbers it sets the return value to `0` at `0x401220`:

```
401220: b8 00 00 00 00              mov    $0x0,%eax
401225: 39 f2                       cmp    %esi,%edx
401227: 74 14                       je     40123d <fun7+0x39>
```

So if we provide a number that is not present in the tree the code will reach a `NULL` pointer and return `-1` and if our number is present it will return `0`.

Now we need to look at each recursive call to `fun7` and see what the current call will do with the result from the inner call. Let's see what happens after the inner `fun7` call returns:

```
40120f: 39 f2                       cmp    %esi,%edx
401211: 7e 0d                       jle    401220 <fun7+0x1c>
401213: 48 8b 7f 08                 mov    0x8(%rdi),%rdi
401217: e8 e8 ff ff ff              callq  401204 <fun7>
```

```
40121c: 01 c0                       add    %eax,%eax
40121e: eb 1d                       jmp    40123d <fun7+0x39>
```

This is the case when our number is smaller than the current node value. It simply doubles the return value (on `0x40121c`) and jumps to `0x40123d` to return.

In the other branch, when our number is greater than or equal to the current node value it goes to `0x401220`:

```
401220: b8 00 00 00 00              mov    $0x0,%eax
401225: 39 f2                       cmp    %esi,%edx
401227: 74 14                       je     40123d <fun7+0x39>
401229: 48 8b 7f 10                 mov    0x10(%rdi),%rdi
40122d: e8 d2 ff ff ff              callq  401204 <fun7>
401232: 8d 44 00 01                 lea    0x1(%rax,%rax,1),%eax
401236: eb 05                       jmp    40123d <fun7+0x39>
```

Here if the values are equal the function will return `0` as we saw earlier but if they are different, `fun7` will be called again and the return value from the inner call will be doubled as in the other branch but in this case it will also add `1` to it:

```
401232: 8d 44 00 01                 lea    0x1(%rax,%rax,1),%eax
```

Although `lea` means ***load effective address*** this instruction is often used for arithmetic operations[4] and that's exactly the case here. The instruction above means:

```
eax = 1 * rax + rax + 1
```

Simplifying:

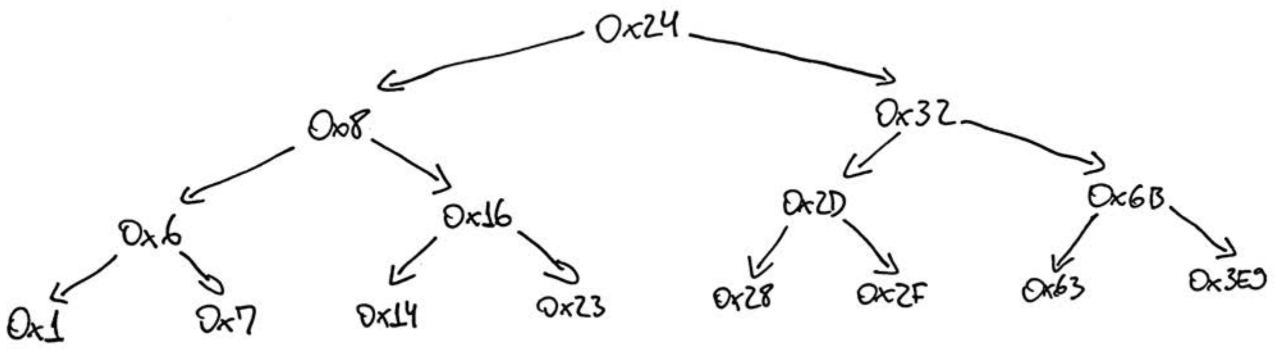
```
eax = 2*rax + 1
```

Which then leads us to the following code for `fun7`:

```
int fun7(node *n, int value) {
    if (n == NULL) {
        return -1;
    }

    if (n->data <= value) {
        if (n->data == value) {
            return 0;
        }
        return 2 * fun7(n->right, value) + 1;
    } else {
        return 2 * fun7(n->left, value);
    }
}
```

Looking again at the tree structure we can try to guess which value will provide the correct answer by visiting the correct nodes.



One option is to visit `0x8` and `0x16`. Replacing the node addresses with the actual `data` they have we would have the following calls to `fun7`:

```
return fun7(0x24, 0x16); // from `secret_phase`
    return 2 * fun7(0x8, 0x16);
        return 2 * fun7(0x16, 0x16) + 1;
            return 0;
```

That will give the right answer:

```
Curses, you've found the secret phase!
But finding it and solving it are quite different...
22
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
```

There is another possible answer for this phase and finding it is left as an exercise to the reader.

Notes

1. Assuming this is a correct program. ↩
2. https://en.wikipedia.org/wiki/Buffer_overflow_protection ↩
3. For more information about `strtol`: http://man7.org/linux/man-pages/man3/strtol.3.html ↩

   > ```
   > long int strtol(const char *nptr, char **endptr, int base);
   > ```
   >
   > *The strtol() function converts the initial part of the string in nptr to a long integer value according to the given base, which must be between 2 and 36 inclusive, or be the special value 0.*

4. If you want to learn more about `lea`, its difference with `mov`, how and why arithmetic operations can be performed with `lea` you can start by reading the following page: https://en.wikibooks.org/wiki/X86_Assembly/Data_Transfer#Load_Effective_Address ↩