Bomb Lab

Antonio Bernardini

January 3, 2025

Contents

Chapter 1

Homework

This is the homework section where you can add the content related to the homework.

1.1 Introduction

The evil *Dr. Evil* planted a slew of "binary bombs" on our department's machines. A binary bomb is a program that consists of a sequence of phases. Each step requires a certain string to be inserted into `stdin`. If the correct string is entered, then the phase is *defused* and the bomb advances to the next phase. Otherwise, the bomb *explodes* by printing `BOOM!!!` and then terminating. The bomb is defused when all phases are defused.

Too many bombs have been planted and we alone cannot handle them. For this reason we give each student a bomb to defuse. Your mission, which you cannot refuse, is to defuse your bomb before the date on which it expires. Good luck, and welcome to the *bomb squad*!

1.2 Step 1: Unload your bomb

To get your bomb, you *must commit to this repository*. Edit the `get_bomb.txt` file by inserting your serial number. An Action will be activated on GitHub which will add an archive called `bomb.zip` to the repository. A version is provided for Linux, Windows and MacOS: you can use whichever you prefer, they are equivalent. If an error occurs while recovering the bomb, a description of the error will be placed in the `ERROR.txt` file.

WARNING

The bomb is for `x86_64` processors, so if you have a Mac with `M1` or `M2`, you need to enter a terminal configured for `x86_64`:

```
    $ env /usr/bin/arch -x86_64 /bin/zsh --login
```

In this environment the bomb has not been tested, therefore its functioning is not guaranteed.

1.3 Step 2: Defuse the bomb (and get bonus points)

Your job for this task is to defuse your bomb. The computer on which execute the bomb must be connected to the Internet, otherwise the professor will not be able to evaluate your solution and award you bonus points. Pay attention because we have been inform yourself that Dr. Evil is truly evil and has inserted some devices into the bomb who notice if the bomb has been tampered with. Despite this caveat, you can really do whatever you want with your bomb. Just remember that this will be evaluated based on what we receive, not on what you tell us you have done on your computer. In other words: at the end of the task, the inputs you entered to defuse the various phases will be verified. You can use various tools to defuse your bomb. Read the section 1.6 for tips and ideas. Each time the bomb explodes 2 points are deducted from the score. Each defused phase gives 10 points. Then there are additional points hidden inside the bomb that you can try to find. The stages become progressively more difficult, although the experience gained in the previous stages should still make the later stages easier. The last stage is still challenging even for the best, so try not to leave it to the last minute. The bomb ignores blank lines in input. If you run the bomb with a command line argument, for example:

```
    $ ./bomb solution.txt
```

it will read the lines from the `solution.txt` file up to `EOF` (end of file) and then read from `stdin`. In a moment of weakness, Dr. Evil introduced this feature, so you won't need to rewrite the solutions to the stages you've already defused every time. To avoid accidentally detonating the bomb, you will have to learn to move through the disassembled code step by step and make extensive use of breakpoints. You will also need to learn to inspect the state of registers and memory. One of the side effects of this homework is that you will become quite good at using the debugger. It's a skill that will come in quite handy when you develop code for this course and throughout your career.

1.4 Logistics

This homework is an individual project. The delivery is electronic and only the ranking and what we receive on our servers are valid for the bonus points. The professor reserves the right not to award bonus points at his complete discretion when correcting your solution. Any clarifications and corrections will be announced on the course noticeboard on Teams.

1.5 Delivery

There is no explicit delivery. The bomb will continue to update your progress as you work on it. Upon expiration, the bomb will simply stop functioning properly. You can track your progress and that of other students by going to the rankings page. This page updates in real time to show the status of all bombs.

1.6 Tips (read carefully!)

There are various ways to defuse your bomb. You can examine the assembly code in detail without ever running it to understand exactly how it behaves. This technique is very useful, but it is not always easy. You can also run the bomb inside a debugger, watching what it does step by step, using the information you gather to defuse it. This is probably the quickest technique. We ask you, however, to avoid adopting a *brute force attack* technique: you could write a program that tests all possibilities to find the right solution for each stage. However, this approach is not good for more than one reason:

- Every time the bomb explodes, you lose two points. The probability of finishing last is 100%.
- You don't know how long the strings to provide as input are, nor do you know which characters they are made up of. Even if you made the (*incorrect*) assumption that the strings are at most 80 characters long and contain only lowercase letters, you would have to try 26^{80} combinations for each step. Even with a modern computer, you wouldn't finish before the delivery date.
- As you interact with the bomb, it contacts our servers to update the ranking in real time. Even if Dr. Evil was foresighted and implemented a form of request throttling, you could easily saturate the server's bandwidth, creating a disruption to the entire class.

There are various tools that are designed to help you understand how a program works and what's wrong when it doesn't work. Below is a list of some tools you may find useful for bomb analysis, with guidance on how to use them. Many of them are available for all operating systems, you just need to find them.

- `gdb` / `lldb` : these are command line debuggers, available on virtually any platform. You can move through the program line by line, examine the memory and registers, look at both the source code and the assembly code (unfortunately Dr. Evil didn't leave us the sources...), set breakpoints, set watchpoints in memory, and even write scripts in Python.
- `objdump -t` : the `objdump` command allows you to extract various information from a compiled binary. Using it with the `-t` flag will print the symbol table. The symbol table contains the names of all functions and all global variables present in the bomb, the names of all functions that the bomb calls, as well as their addresses. You can find out something by looking at the function names!
- `objdump -d` : with this flag the command disassembles the bomb. You can also watch individual features. Reading the assembly code can help you understand how the bomb works. While `objdump -d` will give you a lot of information, it can't tell you the whole story like `gdb` . In fact, many "magical" things happen after a program has been started, because between the *program* and the *process* there is a lot of complex work that the operating system and the standard library do in concert. You'll discover all this in future courses, but the effect is that in some cases a call to `scanf` can become something cryptic like:

  ```
  8048c36:  e8 99 fc ff ff  call   80488d4 <_init+0x1a0>
  ```

- `strings` : this tool shows all the printable strings that are present in the bomb.

Are you looking for a particular instrument? Do you want to know how to best use the ones that have been described? It's time to read the documentation! Never forget about commands like `apropos`, `man` or `info`: they are your friends and can provide you with a lot of information far beyond simple library calls or command line programs. For example, `man ascii` also gives you information about typical encodings used by strings in C. `apropos gdb` will give you more information than you can handle about `gdb`. `info gas` will give you more than you ever wanted to know about the GNU assembler. Furthermore, the web can also be a treasure trove of information. If you find yourself in difficulty, do not hesitate to ask your professor for help, aware however that he too is dealing with bombs to defuse and may not be able to give you useful advice.

Chapter 2

Solution

This is the solution section where the solution to the homework are presented.

2.1 Introduction

To solve this homework we can use the `gdb` tool to disassemble the various phases. Subsequently, for a more complete explanation we convert the assembly code into C code. This allows us to understand in which points there are the various flags to find to move on to the next phase. Enjoy!

2.2 Source Code

The only code we can access is written in C language and is saved inside the `main.c` file which I reproduce below (I removed the comments to make reading more enjoyable):

```c
#include <stdio.h>
#include <stdlib.h>

#include "phases.h"
#include "utils/utils.h"

FILE *infile;

int main(int argc, char **argv) {
    char *input;

    if (argc == 1) {
        infile = stdin;
    } else if(argc == 2) {
        if(!(infile = fopen(argv[1], "r"))) {
            printf("%s: Error: Unable to open file %s\n", argv[0], argv[1]);
            exit(8);
        }
    } else {
        printf("Usage: %s [<input_file>]\n", argv[0]);
        exit(8);
    }

    initialize_bomb();

    printf("Welcome to my little devil bomb. You have six stages that you can use to blow
    yourself up. Have a great day!\n");

    input = read_line();
    phase_1(input);
    phase_defused();
    printf("Phase 1 defused. What do you think about the next one?\n");

    input = read_line();
    phase_2(input);
    phase_defused();
    printf("This was number 2. Keep it up!\n");

    input = read_line();
    phase_3(input);
    phase_defused();
    printf("You're halfway there!\n");

    input = read_line();
    phase_4(input);
    phase_defused();
    printf("Ok, you did it with this one. Now try the next one...\n");

    input = read_line();
    phase_5(input);
    phase_defused();
    printf("Great job! Now on to the next one...\n");

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

    return 0;
}
```

After a quick read of the code it is easy to understand the path to follow. In particular, it is advisable to set, at each start of `gdb`, a breakpoint on the specific phase that we want to solve. The executable to analyze with `gdb` is found inside the `bombs/` folder, and in my case it has the name `311435.linux`.

2.3 Phase 1

As anticipated, we need to use the following command:

```
$ gdb bombs/311435.linux
```

Now, we can set a breakpoint on phase 1 using the following command inside the `gdb` environment:

```
(gdb) b phase_1
```

and finally we execute the code with the command:

```
(gdb) r
```

Now let's disassemble the `phase_1` function:

```
(gdb) disassemble phase_1
```

obtaining the following assembly code:

```
push   %rbp
mov    %rsp,%rbp
sub    $0x20,%rsp
mov    %rdi,-0x8(%rbp)
call   0x401ef0 <get_student_id>
mov    %rax,-0x10(%rbp)
mov    -0x10(%rbp),%rdi
call   0x401610 <sum_digits>
mov    %rax,-0x10(%rbp)
mov    -0x8(%rbp),%rdi
mov    -0x10(%rbp),%rax
mov    %rdi,-0x18(%rbp)
mov    %rax,%rdi
```

```
call   0x405950 <get_quote>
mov    -0x18(%rbp),%rdi
mov    %rax,%rsi
call   0x405b80 <strings_not_equal>
and    $0x1,%al
movzbl %al,%ecx
cmp    $0x0,%ecx
je     0x4016c5 <phase_1+85>
call   0x4015b0 <explode_bomb>
add    $0x20,%rsp
pop    %rbp
ret
```

The C dual of this assembly code is as follows:

```c
void phase_1(const char *input) {
    int student_id = get_student_id();
    int sum = sum_digits(student_id);
    const char *quote = get_quote(sum);
    if (strings_not_equal(input, quote)) {
        explode_bomb();
    }
}
```

This means that the return value of the `get_quote` function is compared with the input we need to insert. Therefore using the command:

```
(gdb) x/s $rdi
```

we can see the string that we need to insert to defuse the bomb. In this case the string is:

```
La semplicita` non precede la complessita`, ma ne consegue.
```

We can then save our string inside the `solution.txt` file. Before starting the next phase, we exit `gdb` with the `q` command.

2.4 Phase 2

To start the next phase we use the following command:

```
$ gdb --args 311435.linux ../solution.txt
```

with which we can specify the `solution.txt` file as a parameter. Now, we can set a breakpoint on phase 2 using the following command inside the `gdb` environment:

```
(gdb) b phase_2
```

and finally we execute the code with the command:

```
(gdb) r
```

Now let's disassemble the `phase_2` function:

```
(gdb) disassemble phase_2
```

obtaining the following assembly code:

```
push   %rbp
mov    %rsp,%rbp
sub    $0x10,%rsp
mov    %rdi,-0x8(%rbp)
call   0x401ef0 <get_student_id>
shr    $0x2,%rax
mov    %rax,-0x10(%rbp)
mov    -0x8(%rbp),%rdi
mov    -0x10(%rbp),%rax
xor    %ecx,%ecx
```

```
mov    %ecx,%edx
mov    $0x2c,%esi
div    %rsi
imul   $0xff,%rdx,%rdx
movabs $0x40b9b0,%rsi
add    %rdx,%rsi
call   0x405b80 <strings_not_equal>
and    $0x1,%al
movzbl %al,%ecx
cmp    $0x0,%ecx
je     0x4017a9 <phase_2+89>
call   0x4015b0 <explode_bomb>
add    $0x10,%rsp
pop    %rbp
ret
```

The C dual of this assembly code is as follows:

```c
void phase_2(const char *input) {
    char *student_id = get_student_id();
    unsigned long value = student_id / 4;

    unsigned long result = value * 0xFF;
    const char *correct_string = (const char *)(0x40b9b0 + result);

    if (strings_not_equal(input, correct_string)) {
        explode_bomb();
    }
}
```

Therefore using the command:

```
(gdb) x/s $rsi
```

we can see the string that we need to insert to defuse the bomb. In this case the string is:

```
Se qualcuno dice: "Voglio un linguaggio di programmazione in cui devo solo dire cio` che
    voglio," dagli un lecca-lecca.
```

We can then save our string inside the `solution.txt` file. Before starting the next phase, we exit `gdb` with the `q` command.

2.5 Phase 3

To start the next phase we use the following command:

```
$ gdb --args 311435.linux ../solution.txt
```

with which we can specify the `solution.txt` file as a parameter. Now, we can set a breakpoint on phase 3 using the following command inside the `gdb` environment:

```
(gdb) b phase_3
```

and finally we execute the code with the command:

```
(gdb) r
```

Now let's disassemble the `phase_3` function:

```
(gdb) disassemble phase_3
```

obtaining the following assembly code:

```
push   %rbp
mov    %rsp,%rbp
sub    $0x30,%rsp
mov    %rdi,-0x8(%rbp)
```

```
movl   $0x8,-0xc(%rbp)
movl   $0xffffffff,-0x10(%rbp)
movq   $0x0,-0x18(%rbp)
mov    -0x8(%rbp),%rdi
movabs $0x4062eb,%rsi
lea    -0xc(%rbp),%rdx
lea    -0x10(%rbp),%rcx
mov    $0x0,%al
call   0x401200 <__isoc99_sscanf@plt>
mov    %eax,-0x1c(%rbp)
cmpl   $0x2,-0x1c(%rbp)
jge    0x401851 <phase_3+81>
call   0x4015b0 <explode_bomb>
mov    -0xc(%rbp),%eax
mov    %eax,%ecx
mov    %rcx,%rdx
sub    $0x7,%rdx
mov    %rcx,-0x28(%rbp)
ja     0x4018fd <phase_3+253>
mov    -0x28(%rbp),%rax
mov    0x406208(,%rax,8),%rcx
jmp    *%rcx
mov    0x40a4e8,%rax
mov    %rax,-0x18(%rbp)
jmp    0x401902 <phase_3+258>
mov    0x40a4f8,%rax
mov    %rax,-0x18(%rbp)
jmp    0x401902 <phase_3+258>
mov    0x40a508,%rax
mov    %rax,-0x18(%rbp)
jmp    0x401902 <phase_3+258>
mov    0x40a518,%rax
mov    %rax,-0x18(%rbp)
jmp    0x401902 <phase_3+258>
mov    0x40a528,%rax
mov    %rax,-0x18(%rbp)
jmp    0x401902 <phase_3+258>
mov    0x40a4f0,%rax
mov    %rax,-0x18(%rbp)
jmp    0x401902 <phase_3+258>
mov    0x40a500,%rax
mov    %rax,-0x18(%rbp)
jmp    0x401902 <phase_3+258>
mov    0x40a510,%rax
mov    %rax,-0x18(%rbp)
jmp    0x401902 <phase_3+258>
call   0x4015b0 <explode_bomb>
mov    -0x18(%rbp),%rax
movslq -0x10(%rbp),%rcx
cmp    %rcx,%rax
je     0x401918 <phase_3+280>
call   0x4015b0 <explode_bomb>
add    $0x30,%rsp
pop    %rbp
ret
```

The C dual of this assembly code is as follows:

```c
void phase_3(const char *input) {
    int index, target_value;
    long resolved_value;

    int read_values = sscanf(input, "%d %d", &index, &target_value);
    if (read_values < 2) {
        explode_bomb();
    }

    if (index < 0 || index > 7) {
        explode_bomb();
    }

    static const long jump_table[] = {
        0x40a4e8, 0x40a4f0, 0x40a4f8, 0x40a500,
        0x40a508, 0x40a510, 0x40a518, 0x40a528
    };

    resolved_value = jump_table[index];

    if (resolved_value != target_value) {
        explode_bomb();
    }
}
```

In this case the input numbers that we need to insert to defuse the bomb is:

```
7 41
```

We can then save our numbers inside the `solution.txt` file. Before starting the next phase, we exit `gdb` with the `q` command.

2.6 Phase 4

To start the next phase we use the following command:

```
$ gdb --args 311435.linux ../solution.txt
```

with which we can specify the `solution.txt` file as a parameter. Now, we can set a breakpoint on phase 4 using the following command inside the `gdb` environment:

```
(gdb) b phase_4
```

and finally we execute the code with the command:

```
(gdb) r
```

Now let's disassemble the `phase_4` function:

```
(gdb) disassemble phase_4
```

obtaining the following assembly code:

```
push   %rbp
mov    %rsp,%rbp
sub    $0x20,%rsp
mov    %rdi,-0x8(%rbp)
movabs $0x40a250,%rax
mov    %rax,-0x10(%rbp)
movq   $0x0,-0x18(%rbp)
mov    -0x8(%rbp),%rax
mov    %rax,-0x20(%rbp)
mov    -0x10(%rbp),%rax
cmpq   $0x0,(%rax)
je     0x4019de <phase_4+110>
mov    -0x10(%rbp),%rax
mov    (%rax),%rax
cmpq   $0x0,0x8(%rax)
je     0x4019ce <phase_4+94>
mov    -0x10(%rbp),%rax
mov    (%rax),%rax
mov    0x8(%rax),%rax
mov    %rax,-0x18(%rbp)
jmp    0x4019de <phase_4+110>
mov    -0x10(%rbp),%rax
mov    (%rax),%rax
mov    %rax,-0x10(%rbp)
jmp    0x40199a <phase_4+42>
jmp    0x4019e3 <phase_4+115>
mov    -0x18(%rbp),%rax
cmpb   $0x0,(%rax)
je     0x401a2c <phase_4+188>
mov    -0x18(%rbp),%rax
mov    %rax,%rcx
add    $0x1,%rcx
mov    %rcx,-0x18(%rbp)
movsbl (%rax),%edx
mov    -0x20(%rbp),%rax
mov    %rax,%rcx
add    $0x1,%rcx
mov    %rcx,-0x20(%rbp)
movsbl (%rax),%esi
cmp    %esi,%edx
je     0x401a27 <phase_4+183>
call   0x4015b0 <explode_bomb>
jmp    0x4019e3 <phase_4+115>
add    $0x20,%rsp
pop    %rbp
ret
```

The C dual of this assembly code is as follows:

```c
void phase_4(const char *input) {
    const char *correct_string = (const char *)0x40a250;
    const char *list_string = NULL;
    const char **node = (const char **)correct_string;

    while (node != NULL) {
        if (*node == NULL) {
            break;
        }
        if ((*node)[1] == '\0') {
            list_string = *node;
            break;
        }
        node = (const char **)(*node)[1];
    }

    if (list_string == NULL) {
        explode_bomb();
    }

    const char *p1 = list_string;
    const char *p2 = input;

    while (*p1 != '\0') {
        if (*p2 == '\0') {
            explode_bomb();
        }
        if (*p1 != *p2) {
            explode_bomb();
        }
        p1++;
        p2++;
    }

    if (*p2 != '\0') {
        explode_bomb();
    }
}
```

By analyzing the addresses of the linked list we arrive at the following string:

```
Poi c'e` il processore a 1 bit che ha due istruzioni: nop e halt.
```

We can then save our string inside the `solution.txt` file. Before starting the next phase, we exit `gdb` with the `q` command.

2.7 Phase 5

To start the next phase we use the following command:

```
$ gdb --args 311435.linux ../solution.txt
```

with which we can specify the `solution.txt` file as a parameter. Now, we can set a breakpoint on phase 5 using the following command inside the `gdb` environment:

```
(gdb) b phase_5
```

and finally we execute the code with the command:

```
(gdb) r
```

Now let's disassemble the `phase_5` function:

```
(gdb) disassemble phase_5
```

obtaining the following assembly code:

```
push   %rbp
mov    %rsp,%rbp
sub    $0x20,%rsp
mov    %rdi,-0x8(%rbp)
mov    -0x8(%rbp),%rdi
call   0x405b30 <string_length>
mov    %rax,-0x10(%rbp)
cmpq   $0x7,-0x10(%rbp)
je     0x401ab9 <phase_5+41>
call   0x4015b0 <explode_bomb>
movl   $0x0,-0x1c(%rbp)
cmpl   $0x7,-0x1c(%rbp)
jge    0x401afb <phase_5+107>
mov    -0x8(%rbp),%rax
movslq -0x1c(%rbp),%rcx
movsbl (%rax,%rcx,1),%edx
and    $0xf,%edx
movslq %edx,%rax
mov    0x40a2b0(,%rax,1),%sil
movslq -0x1c(%rbp),%rax
mov    %sil,-0x18(%rbp,%rax,1)
mov    -0x1c(%rbp),%eax
add    $0x1,%eax
mov    %eax,-0x1c(%rbp)
jmp    0x401ac0 <phase_5+48>
lea    -0x18(%rbp),%rdi
movb   $0x0,-0x11(%rbp)
mov    0x40a568,%rsi
call   0x405b80 <strings_not_equal>
and    $0x1,%al
movzbl %al,%ecx
cmp    $0x0,%ecx
je     0x401b23 <phase_5+147>
call   0x4015b0 <explode_bomb>
add    $0x20,%rsp
pop    %rbp
ret
```

The C dual of this assembly code is as follows:

```c
void phase_5(const char *input) {
    size_t length = string_length(input);
    if (length != 7) {
        explode_bomb();
    }

    char transformed[7];
    for (int i = 0; i < 7; ++i) {
        char c = input[i];
        c = c & 0xF;
        transformed[i] = (char)(0x40a2b0[c]);
    }
    transformed[7] = '\0';

    if (strings_not_equal(transformed, (const char *)0x40a568)) {
        explode_bomb();
    }
}
```

Analyzing the address `0x40a568` there is the string `mielosi`. Since a mask is applied to the input, to obtain `mielosi`, knowing that the alphabet (at the address `0x40a2b0`) is `mersnoilt` we simply have to insert in the file `solution.txt` (and therefore in input) the positions that form this string, namely `0617536`. Before starting the next phase, we exit `gdb` with the `q` command.

2.8 Phase 6

To start the next phase we use the following command:

```
$ gdb --args 311435.linux ../solution.txt
```

with which we can specify the `solution.txt` file as a parameter. Now, we can set a breakpoint on phase 6 using the following command inside the `gdb` environment:

```
(gdb) b phase_6
```

and finally we execute the code with the command:

```
(gdb) r
```

Now let's disassemble the `phase_6` function:

```
(gdb) disassemble phase_6
```

obtaining the following assembly code:

```
push   %rbp
mov    %rsp,%rbp
sub    $0x20,%rsp
mov    %rdi,-0x8(%rbp)
movl   $0xffffffff,-0xc(%rbp)
movl   $0xffffffff,-0x10(%rbp)
movl   $0xb,-0x14(%rbp)
mov    -0x8(%rbp),%rdi
movabs $0x4062e8,%rsi
lea    -0xc(%rbp),%rdx
lea    -0x14(%rbp),%rcx
lea    -0x10(%rbp),%r8
mov    $0x0,%al
call   0x401200 <__isoc99_sscanf@plt>
mov    %eax,-0x1c(%rbp)
cmpl   $0x3,-0x1c(%rbp)
jne    0x401db9 <phase_6+89>
cmpl   $0x0,-0xc(%rbp)
jge    0x401dbe <phase_6+94>
call   0x4015b0 <explode_bomb>
xor    %esi,%esi
mov    -0xc(%rbp),%edi
mov    $0xa,%edx
call   0x401c90 <bs>
mov    %eax,-0x18(%rbp)
mov    -0x18(%rbp),%eax
cmp    -0x14(%rbp),%eax
jne    0x401dfc <phase_6+156>
mov    -0x10(%rbp),%eax
cmp    0x40a540,%eax
jne    0x401dfc <phase_6+156>
mov    -0xc(%rbp),%eax
cmp    0x40a570,%eax
je     0x401e01 <phase_6+161>
```

```
call   0x4015b0 <explode_bomb>
add    $0x20,%rsp
pop    %rbp
ret
```

The C dual of this assembly code is as follows:

```c
void phase_6(const char *input) {
    int num1, num2, num3;

    int parsed = sscanf(input, "%d %d %d", &num1, &num3, &num2);
    if (parsed != 3) {
        explode_bomb();
    }

    if (num1 < 0) {
        explode_bomb();
    }

    int bs_result = bs(num1, 0xA);
    if (bs_result != num3) {
        explode_bomb();
    }

    if (num2 != 0x40a540) {
        explode_bomb();
    }

    if (num1 != 0x40a570) {
        explode_bomb();
    }
}
```

In this case the input numbers that we need to insert to defuse the bomb is:

```
8 0 3
```

We can then save our numbers inside the `solution.txt` file. Before starting the next phase, we exit `gdb` with the `q` command.

2.9 Phase Secret

To access the secret phase, note that in the `phase_defused` function there is a call to the `phase_secret` function:

```
push   %rbp
mov    %rsp,%rbp
sub    $0x10,%rsp
call   0x4056e0 <defused>
call   0x404ad0 <async_notify>
cmpl   $0x6,0x40a544
jne    0x401fd8 <phase_defused+200>
movabs $0x40a5b0,%rax
add    $0x2fd,%rax
mov    %rax,%rdi
mov    $0x20,%esi
call   0x401160 <strrchr@plt>
mov    %rax,-0x8(%rbp)
mov    -0x8(%rbp),%rax
add    $0x1,%rax
mov    %rax,-0x8(%rbp)
mov    %rax,%rdi
movabs $0x4062f1,%rsi
call   0x405b80 <strings_not_equal>
and    $0x1,%al
movzbl %al,%ecx
cmp    $0x0,%ecx
jne    0x401fb2 <phase_defused+162>
```

```
movabs $0x4062fa,%rdi
call   0x4010d0 <puts@plt>
mov    %eax,-0xc(%rbp)
call   0x4024f0 <read_line>
mov    %rax,%rdi
call   0x405d40 <phase_secret>
call   0x401f10 <phase_defused>
movabs $0x406363,%rdi
call   0x4010d0 <puts@plt>
call   0x404a00 <sync_notify>
movabs $0x406398,%rdi
call   0x4010d0 <puts@plt>
movabs $0x4063c4,%rdi
mov    %eax,-0x10(%rbp)
call   0x4010d0 <puts@plt>
add    $0x10,%rsp
pop    %rbp
ret
```

And in particular, this function can be accessed only if, scrolling through the linked list in phase 4, the string `--DrMale` is found. Then you need to change the string in phase 4 to the following:

```
    Poi c'e` il processore a 1 bit che ha due istruzioni: nop e halt. --DrMale
```

So by restarting `gdb`, you will actually be able to access the `secret_phase` function of which I report directly the disassembled:

```
push   %rbp
mov    %rsp,%rbp
sub    $0x20,%rsp
mov    %rdi,-0x8(%rbp)
mov    -0x8(%rbp),%rdi
movabs $0x4062ee,%rsi
lea    -0xc(%rbp),%rdx
mov    $0x0,%al
call   0x401200 <__isoc99_sscanf@plt>
mov    %eax,-0x10(%rbp)
cmpl   $0x1,-0x10(%rbp)
je     0x405d77 <phase_secret+55>
call   0x4015b0 <explode_bomb>
movl   $0x0,-0x14(%rbp)
mov    -0x14(%rbp),%eax
mov    %eax,%ecx
cmp    $0x20,%rcx
jae    0x405ddd <phase_secret+157>
imul   $0x41c64e6d,0x40a564,%eax
add    $0x3039,%eax
and    $0x7fffffff,%eax
mov    %eax,0x40a564
mov    0x40a564,%eax
and    $0x1,%eax
mov    -0xc(%rbp),%ecx
and    $0x1,%ecx
cmp    %ecx,%eax
je     0x405dc6 <phase_secret+134>
call   0x4015b0 <explode_bomb>
mov    -0xc(%rbp),%eax
sar    $0x1,%eax
mov    %eax,-0xc(%rbp)
mov    -0x14(%rbp),%eax
add    $0x1,%eax
mov    %eax,-0x14(%rbp)
jmp    0x405d7e <phase_secret+62>
add    $0x20,%rsp
pop    %rbp
ret
```

The C dual of this assembly code is as follows:

```c
void phase_secret(const char *input) {
    if (input == NULL) {
        explode_bomb();
    }

    int value;
    int iterations = 0;
    int random_state = 4;

    if (sscanf(input, "%d", &value) != 1) {
        explode_bomb();
    }

    while (iterations < 32) {
        random_state = (random_state * 0x41c64e6d + 0x3039) & 0x7fffffff;

        if ((random_state & 1) != (value & 1)) {
            explode_bomb();
        }

        value >>= 1;
        iterations++;
    }

    printf("Phase secret defused successfully!\n");
}
```

To solve the secret phase it is better to create an exploit in C that performs exactly the reverse operations of the secret phase, here is the complete code:

```
#include <stdio.h>
#include <stdlib.h>

void explode_bomb() {
    printf("Bomb exploded!\n");
    exit(1);
}

void phase_secret(const char *input) {
    if (input == NULL) {
        explode_bomb();
    }

    int value;
    int iterations = 0;
    int random_state = 4;

    if (sscanf(input, "%d", &value) != 1) {
        explode_bomb();
    }

    while (iterations < 32) {
        random_state = (random_state * 0x41c64e6d + 0x3039) & 0x7fffffff;

        if ((random_state & 1) != (value & 1)) {
            explode_bomb();
        }

        value >>= 1;
        iterations++;
    }

    printf("Phase secret defused successfully!\n");
}

void exploit() {
    int random_state = 4;
    int target_value = 0;

    for (int i = 0; i < 32; ++i) {
        random_state = (random_state * 0x41c64e6d + 0x3039) & 0x7fffffff;
        int bit = (random_state & 1);
        target_value |= (bit << i);
    }
```

```c

    printf("Exploit phase_secret with: %d\n", target_value);
}

int main(void) {
    char input[100];

    exploit();

    printf("Enter a number: ");
    fgets(input, sizeof(input), stdin);
    phase_secret(input);

    return 0;
}
```

so if we compile and run the exploit we get:

```
$ gcc phase-secret.c
$ ./a.out
Exploit phase_secret with: 1431655765
Enter a number: 1431655765
Phase secret defused successfully!
```

2.10 Conclusion

This homework had such a deadline that by the time I thought about creating this pdf I could no longer test the actual commands on `gdb` and thus actually show the reasoning step by step. So I went off the cuff, writing down what I remembered of each individual phase. Unfortunately there are some things that if not seen in detail you have to trust their actual functioning. I don't know if in the future I will have the opportunity to redo a homework like this, if it were to happen I will remember to document the entire process as soon as possible.