REVERSE ENGINEERING

"The one where we dig deep into the rabbit hole."

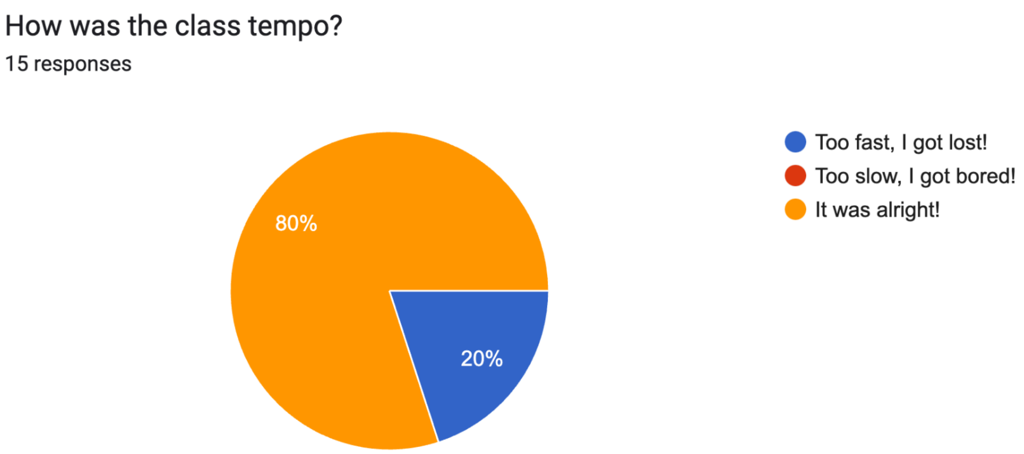
November 21st, 2024

Credits

Content: Muris Sladić, Sebastian Garcia, Maria Rigaki
Martin Řepa, Lukáš Forst., Veronica Valeros, Ondřej Lukáš
Illustrations: Fermin Valeros
Design: Veronica Garcia, Veronica Valeros, Ondřej Lukáš
Music: Sebastian Garcia, Veronica Valeros, Ondřej Lukáš
CTU Video Recording: Jan Sláma, Václav Svoboda, Marcela Charvatová
Audio files, 3D prints, and Stickers: Veronica Valeros

CLASS DOCUMENT	https://bit.ly/BSY2024-9
WEBSITE	https://cybersecurity.bsy.fel.cvut.cz/
CLASS MATRIX	https://matrix.bsy.fel.cvut.cz/
CLASS CTFD **(CTU STUDENTS)**	https://ctfd.bsy.fel.cvut.cz/
CLASS PASSCODE FORM **(ONLINE STUDENTS)**	https://bit.ly/BSY-VerifyClass
FEEDBACK	https://bit.ly/BSYFEEDBACK
LIVESTREAM	https://www.youtube.com/playlist?list=PLQL6z4JeTTQmu09ItEQaqjt6tk0KnRsLh
INTRO SOUND	https://bit.ly/BSY-Intro
VIDEO RECORDINGS PLAYLIST	https://www.youtube.com/playlist?list=PLQL6z4JeTTQk_z3vwSIvn6wIHMeNQFU3d
CLASS AUDIO	https://audio.com/stratosphere

Results from the survey of the last class (14:32)



Responses to feedback:

- The last class required knowledge of programming, and it wasn't taught before:
  - Yes. This is a University class for master's students who have already taken several courses on programming.
  - We will consider how to better support online students who do not have this prerequisite next year, maybe with additional lectures or reading material.
- Architecture issues to follow the examples:
  - Yes. Mac with Arm or other architectures would make this harder. It is very hard to provide support for all. We are trying to overcome this issue, but most students still have non-arm architecture.
- In the ctfd system, the past homework is not visible.
  - Yes. There is no way to lock a challenge only to change its visibility on CTFd. If someone knows how to do this or wants to implement a feature (CTFd is free software), reach out. You can still see the scoreboard to see your progress.

Class outline

- What is Reverse Engineering?
- Preparations
- Reversing a Network Protocol
- Defusing a binary bomb 💣

Before We Start

Online students only!

Because it might take some time, make sure you start your StratoCyberLab now and start **Class 9.**

Once it starts, to follow this class, you need to do the following:

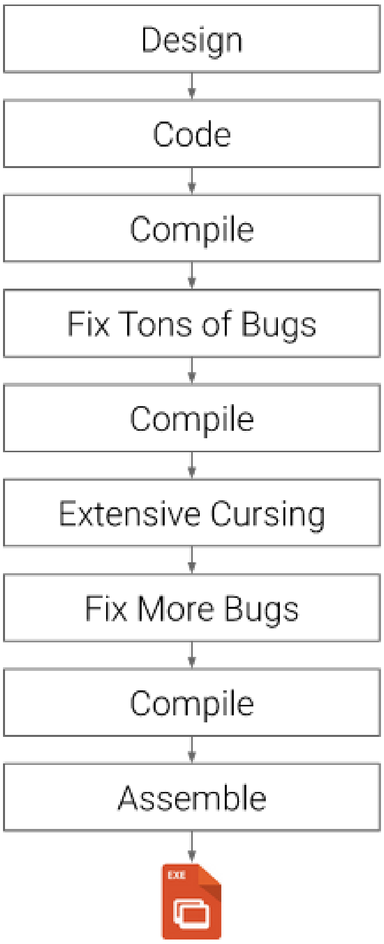
- `ssh root@172.20.0.120`
- `Pass: admin`

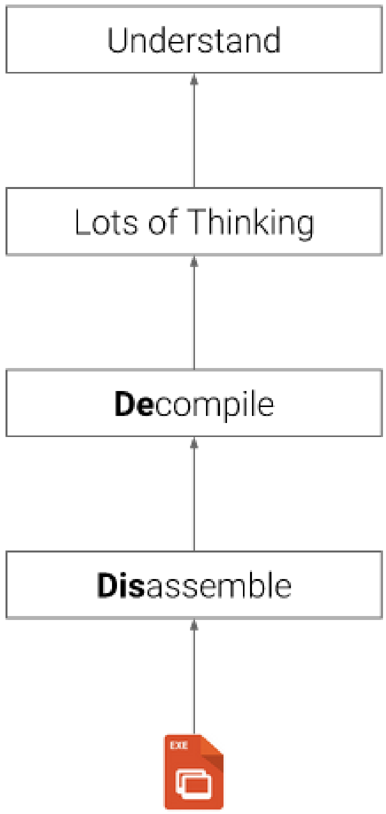
What is Reverse Engineering?

Learn about different aspects of reverse engineering and how we can use it in security.

- A process of analyzing a (black-box) system to understand its design, function, and operation.
- A critical tool for researchers, analysts, and security professionals to protect against cyber threats and to improve the security of systems.

"Forward engineering process"[1] **"Reverse engineering process"**





Why do we need it?

There are many cases where we need reverse engineering in security, in particular:

- To understand how malicious software operates
  - What are its goals and capabilities?
  - How does it spread?
  - What are its vulnerabilities or weaknesses?
- To find fingerprints and identify/track threat actors
- To develop countermeasures and defenses against malware

[1] Images from https://pwn.college/cse365-f2023/reverse-engineering

Case study: WannaCry ransomware (2017)

WannaCry is a malware with ransomware and worm components. As ransomware, it can encrypt the files in a computer system. As a worm, it is capable of spreading and infecting other computers. The key characteristics of WannaCry:

- WannaCry used an NSA-leaked exploit called **EternalBlue** to gain access and infect other computers:
    - 300,000 computers infected across 150 countries in a few hours
    - Damages ranged from hundreds of millions to billions of USD.
- A group of reverse engineers analyzed WannaCry and managed to stop the spreading of the malware by just registering a domain.

Types of analysis: static vs dynamic

- **Static** analysis: analyze a binary file **without** executing it (at rest).
    - Binary dependent (type of compilation, programming language, architecture)
    - Analyzing assembly language, decompiled code, statically allocated variables, imported libraries, etc.
    - Almost impossible with packed malware
    - Safe for the analyst

- **Dynamic** analysis: analyze a program at **runtime**
    - Using tracing tools to trace system and library calls (*strace*, *ltrace*)
    - Debugging (*GDB*, *WinDBG*, etc)
    - Sandboxes to simulate a real environment
    - Malware can detect emulation (anti-debug techniques)
    - Might be dangerous

Toolbox

Many different tools are used in the reversing process: disassemblers, debuggers, decompilers, sandboxes, system forensics tools, network analysis tools, etc.

Each platform has a different set of tools:

- Disassemblers and decompilers: Ghidra, IDA, radare2 and iaito (GUI), binary ninja, Hopper (for Mac), JADX (for Android)
- Useful Linux tools: gdb, objdump, readelf, file, strings, nm, strace, ltrace, and more.
- Windows tools VM: Flare VM (Mandiant) has everything you need.

In this class, we will be using IDA and GDB. IDA is one of the most popular and widely used tools for reversing. It is a disassembler, has a decompiler, debugger, and many other functionalities. Due to the scope of the class we are not able to go over some other tools.

Preparations: x64 Assembly and GDB (14:45)

Goal: Learn and/or remind ourselves of registers and calling conventions in x64. Introduce basic usage and concepts of GDB.

You can find nice x64 cheatsheets here[2], or here[3].

Let's take a look at the registers we will be encountering later.

	64 bit	32 bit	16 bit	8 bit
A (accumulator)	RAX	EAX	AX	AL
B (base, addressing)	RBX	EBX	BX	BL
C (counter, iterations)	RCX	ECX	CX	CL
D (data)	RDX	EDX	DX	DL
	RDI	EDI	DI	DIL
	RSI	ESI	SI	SIL
Numbered (n=8..15)	Rn	RnD	RnW	RnB
Stack pointer	RSP	ESP	SP	SPL
Frame pointer	RBP	EBP	BP	BPL

The calling convention is:

- Registers **rdi, rsi, rdx, rcx, r8, and r9** are used to pass the first six integers or pointer parameters to called functions. For floats **xmm0 - xmm7** registers are used.
- Registers **rbp, rbx, r12 - r15** are **nonvolatile** - must be preserved by callee.
- Rest are **volatile** - function can overwrite them.
- **rax** - return value register

[2] https://gist.github.com/justinian/385c70347db8aca7ba93e87db90fc9a6
[3] https://learn.microsoft.com/en-us/cpp/build/x64-software-conventions?view=msvc-170

Examples:

- **mov rax, rdx** - move data from rdx register to the rax register
- **lea rdi, [rbx+0x10]** - move address rbx+0x10 into the rdi register
- **add rax, rdx** - set rax equal to rax + rdx
- **push rax** - grow stack by 8 bytes, and put contents of the rax on top
- **pop rax** - put the top 8 bytes of the stack in the rax register
- **jmp 0x520310** - jump to address 0x520310 and execute the instruction there

Also, install and test GEF[4] in your containers, which is a set of extensions for the GDB debugger (This is already in your dockers) :

- If you need to install it with the following command:
    - `wget https://gef.blah.cat/sh`
    - `vi sh (to check it is not malicious code from marik0)`
    - `bash sh`
    - `Or, as a one-liner:`
        - `bash -c "$(curl -fsSL https://gef.blah.cat/sh)"`
- Test it works
    - `gdb -nx -ex 'pi print(sys.version)' -ex quit`

```
3.8.10 (default, May 26 2023, 14:05:08)
[GCC 9.4.0]
```

[4] https://hugsy.github.io/gef/

Let's take GDB out for a spin.

We are going to start by creating a simple C code and compiling it.

```
#include <stdio.h>

int return_five(void){

    for(int i = 0; i < 6; i++){

        if(i == 6 - 1) {

             return 5;

        }

    }

}

void main(void){

    int a = 5;

    if(a == return_five())
         printf("Hello World %d\n!", a);
}
```

- You can find this code in your dockers:
    - `cp -r /data/reversing-class/ ./reversing-class`
        - `-r - copy the directory`
    - `cd ~/reversing-class/hello_world`
- Let's use gcc to compile the code
    - `gcc -o hello_world hello_world.c`
        - `-o to set the name of the executable binary`
- We can try to run our code:
    - `./hello_world`
- Good! Now we can open it in GDB
    - `gdb ./hello_world`
- We can put a breakpoint at the main function.
    - `b main`
- Now we run the code

    - `r`
- We can take a look at the disassembled code
    - `disass main`

```
Dump of assembler code for function main:
   0x00005cccc5100173 <+0>:     endbr64
   0x00005cccc5100177 <+4>:     push   rbp
   0x00005cccc5100178 <+5>:     mov    rbp,rsp
=> 0x00005cccc510017b <+8>:     sub    rsp,0x10
   0x00005cccc510017f <+12>:    mov    DWORD PTR [rbp-0x4],0x5
   0x00005cccc5100186 <+19>:    call   0x5cccc5100149 <return_five>
   0x00005cccc510018b <+24>:    cmp    DWORD PTR [rbp-0x4],eax
   0x00005cccc510018e <+27>:    jne    0x5cccc51001a9 <main+54>
   0x00005cccc5100190 <+29>:    mov    eax,DWORD PTR [rbp-0x4]
   0x00005cccc5100193 <+32>:    mov    esi,eax
   0x00005cccc5100195 <+34>:    lea    rax,[rip+0xe68]        # 0x5cccc5101004
   0x00005cccc510019c <+41>:    mov    rdi,rax
   0x00005cccc510019f <+44>:    mov    eax,0x0
   0x00005cccc51001a4 <+49>:    call   0x5cccc5100050 <printf@plt>
   0x00005cccc51001a9 <+54>:    nop
   0x00005cccc51001aa <+55>:    leave
   0x00005cccc51001ab <+56>:    ret
```

- The current instruction where the breakpoint is
    - `sub rsp, 0x10`
    - What does this mean?
- We can execute the next instruction by typing one of these
    - `ni`
    - `nexti`
- There is a call to a *return_five* function. We can put a breakpoint at that address
    - `b return_five`
    - Or we can put a breakpoint at the address:
        - `b *0x<put_the_address_shown_in_gdb>`
        - To see the breakpoints we can use
            - `info b`
        - To remove a breakpoint
            - `delete <breakpoint_number>`
- We can further analyze the assembly. To continue execution to next breakpoint
    - `c`
- Let's return to the main function. We can put a new breakpoint to the instruction after the function call to *return_five*. So we go back in our terminals and look for
    - `cmp DWORD PTR [rbp-0x4], eax`
    - `b *0x<put_the_address_shown_in_gdb>`
    - `c`

- The *jne* jump after the current instruction will not be taken since the value of the **eax** register, and the memory location **rbp-0x4** are the same.
    - But if we want to make that jump for some reason, we can use GDB to modify the disassembly. For example, first, we can check what is the value at address *rbp-0x4*
        - `x/w $rbp-0x4`
            - w - this means dword
    - Then we can change the value of at the address pointed to by **rbp-0x4**
        - `set *<address_from_x/w_command>=0x6`
    - Notice now that the jump is taken

First part recap:

- We reminded ourselves of the registers in x64 architecture and how they are passed to functions.
- We went over some basic commands in assembly.
- We got familiar with some basic use cases in GDB.

And now for something a little bit different...and then we return to assembly and GDB 🙂

Example 1: Reversing a Network Protocol (15:00)

Goal: Learn to reverse engineer a custom command and control protocol.

Download the RAT04.pcap file **on your laptop** from here.

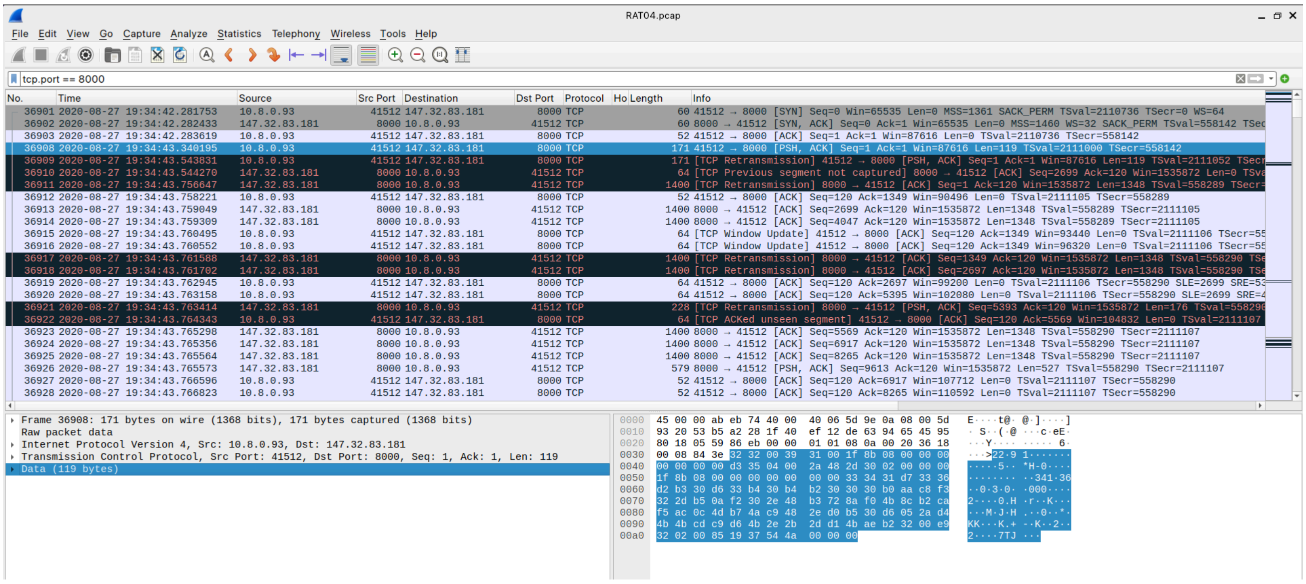
This is a packet capture between a RAT malware (SpyMax) running on an Android phone and a command and control server[5].

*RAT stands for **R**emote **A**ccess **T**rojan. It is a type of malware that controls a system through a remote network. It is typically installed without the victim's knowledge.*

Open the file with Wireshark. We will focus on the traffic between the mobile phone and the server. Use the following filter and look at the first packet that contains data:

[5] https://www.stratosphereips.org/blog/2021/2/26/dissecting-a-rat-analysis-of-the-spymax

- `tcp.port == 8000`



The first packet that contains data is the packet with the PSH byte on.

The data start at the byte with the value “0x32” and have a size of 119 bytes.

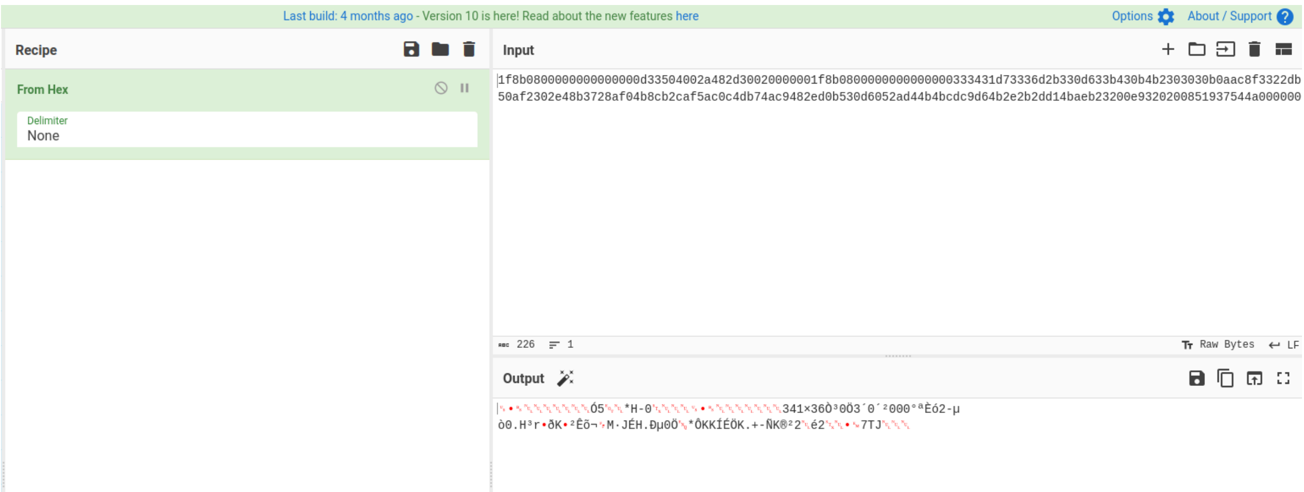
- Are there any interesting patterns in the data?
- What could the bytes at the beginning be? Look at the ASCII representation on the right side.
- Hint: 22 + 91 = 113, but the total number of bytes in the data are 119!
- What is “**1f 8b**”?

```
45 00 00 ab eb 74 40 00  40 06 5d 9e 0a 08 00 5d   E····t@· @·]····]
93 20 53 b5 a2 28 1f 40  ef 12 de 63 94 65 45 95   · S··(·@ ···c·eE·
80 18 05 59 86 eb 00 00  01 01 08 0a 00 20 36 18   ···Y···· ····· 6·
00 08 84 3e 32 32 00 39  31 00 1f 8b 08 00 00 00   ···>22·9 1·······
00 00 00 00 d3 35 04 00  2a 48 2d 30 02 00 00 00   ·····5·· *H-0····
1f 8b 08 00 00 00 00 00  00 00 33 34 31 d7 33 36   ········ ··341·36
d2 b3 30 d6 33 b4 30 b4  b2 30 30 30 b0 aa c8 f3   ··0·3·0· ·000····
32 2d b5 0a f2 30 2e 48  b3 72 8a f0 4b 8c b2 ca   2-···0.H ·r··K···
f5 ac 0c 4d b7 4a c9 48  2e d0 b5 30 d6 05 2a d4   ···M·J·H .··0··*·
4b 4b cd c9 d6 4b 2e 2b  2d d1 4b ae b2 32 00 e9   KK···K.+ -·K··2··
32 02 00 85 19 37 54 4a  00 00 00                  2····7TJ ···
```

Let’s use CyberChef[6] to test some ideas:

[6] https://cyberchef.org/

- Select the data part of the packet.
- Right-click on the part that says *Data (119 bytes)*, select **Copy,** and then "**… as a Hex Stream**"
- Remove the bytes until the "**1f 8b**" and use the "**From Hex**" recipe with delimiter **None**
- What does CyberChef suggest these bytes are?
- Magic bytes: https://www.wikiwand.com/en/List_of_file_signatures



Link to a ready to use Cyberchef in case you can not do it.

- Now that we have a good idea of the data in the first packet, let's check some more packets. Does the pattern continue?
- Use this filter to show only packets that contain data:
  - `(tcp.port == 8000) && (data)`
- Let's go back to Wireshark and see if we can find similar packets.

Automating the extraction (16:10)

We notice that the data of a smaller size seems to follow the pattern we noticed in the first packet. So we can filter by size as well.

- `(((tcp.port == 8000) && (data))) && (data.len <= 300)`

Since we have (potentially) hundreds of thousands of packets that follow a similar format, it is best to automate the process. We will do this in our containers.

In the docker container, we have to go to second example folder :

- `cd ~/reversing-class/first_example`

Online people can download the Pcap file to their class 9 container with the following command:

- `curl "https://drive.usercontent.google.com/download?id=18v47zh32W-yzI59ZY4_TH7nd79wPODhm&export=download&authuser=0&confirm=t" -o RAT04.pcap`

We will use **tshark**[7] to extract the interesting data with the same filter we created in Wireshark:

- `tshark -r RAT04.pcap -Tfields -Y "(tcp.port == 8000) && (data) && (data.len <= 300)" -e data > small_data.log`
    - `-r:` read a file
    - `-Y:` use a display filter
    - `-T fields`: format of text output -> fields
    - `-e data`: field to print -> data

Now that we have all the packets with data smaller than 300 bytes, let's try to automate the decoding process using Python.

For example, we can do it manually for the first packet:

```
Python 3.10.12 (main, Sep 11 2024, 15:47:36)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> with open("small_data.log", "r") as f:
...   log_data = f.readlines()
...
>>> log_data[0]
'3232003931001f8b0800000000000000d33504002a482d30020000001f8b08000000000000000333431d73336d2b330d633b430b4b2303030b0aac8f3322db50af2302e48b3728af04b8cb2caf5ac0c4db74ac9482ed0b530d6052ad44b4bcdc9d64b2e2b2dd14baeb23200e9320200851937544a000000\n'
>>> int(bytes.fromhex(log_data[0].strip())[:2])
22
>>> byte_stream = bytes.fromhex(log_data[0].strip())
>>> part1 = byte_stream[6:6 + 22]
>>> part2 = byte_stream[6 + 22:]
>>> import gzip
>>> gzip.decompress(part1)
b'-1'
```

[7] https://tshark.dev/


```
>>> gzip.decompress(part2)
b'147.32.83.181:8000:xnJ5u:RH3pf:BXNaZ:mIyUg:dhcp-83-181.felk.cvut.cz:00
00:2'
```

Let’s check the full script and then run it:

- `cat decoder_small.py`
- `python3 decoder_small.py | less`
    - If you don't have less, install it: `apt install -y less`

Let's answer some questions:

1. What kind of information is there?

```
b'147.32.83.181:8000:xnJ5u:RH3pf:BXNaZ:mIyUg:dhcp-83-181.felk.cvut.cz:00
00:2'

b'1x0F0xplugens.angel.plugens.infox0F0xmethodx0F0x1GRU802x0F0xinfox0D0xx
nJ5ux0D0xmIyUg'

b'PING dhcp-83-181.felk.cvut.cz (147.32.83.181) 56(84) bytes of data.---
dhcp-83-181.felk.cvut.cz ping statistics ---1 packets transmitted, 0
received, 100% packet loss, time 0ms'

b'1x0F0xplugens.angel.plugens.filesx0F0xmethodx0F0x5XBL990x0F0xfilesx0D0
xget0'

b'1x0F0xplugens.angel.plugens.filesx0F0xmethodx0F0x-1x0F0xcommendx0D0xcp
-R /storage/emulated/0/DCIM/Camera /storage/emulated/0/DCIM'

b'65.9667x0D0x-18.5333x0D0x0.0x0D0x0'
```

2. Did we manage to reverse all the packets? Do we understand everything?
3. Next steps:
    a. Separate traffic, add timestamps, enrich the data we extract from the pcap, etc.
    b. Look at the bigger packets.
    c. Look at the failed packets.

  d. Reverse the android apk file.

  e. Other?

4. Read more in the blog "Dissecting a RAT. Analysis of the SpyMAX"[8][9]

~~~~ 💗 **First Break!** 💗~~~~ (15:30, 10m)

Example 3: Defusing a binary bomb 💣 (15:40)

Goal: Reverse a C-based Linux binary using a disassembler/decompiler and a debugger.

[8] https://www.stratosphereips.org/blog/2021/2/26/dissecting-a-rat-analysis-of-the-spymax

[9] https://dspace.cvut.cz/bitstream/handle/10467/94720/F3-BP-2021-Babayeva-Kamila-Kamila_Bachelor_Thesis__RAT_Execution_and_Analysis.pdf?sequence=-1&isAllowed=y

~~~~

A lab exercise from CMU. From the original web page[10]:

> *A "binary bomb" is a program provided to students as an object code file. When run, it prompts the user to type in **6 different strings**. If any of these is incorrect, the bomb "explodes," printing an error message...*
>
> *Students must "defuse" their own bomb by disassembling and reverse engineering the program to determine what the 6 strings should be.*

Preparation

If you have not installed it already, please download and install IDA Free[11] on your computers.

Download the bomb binary

Download the binary in your computers from here[12].

The binary is also in the class folder in docker. Let's execute it and try some strings:

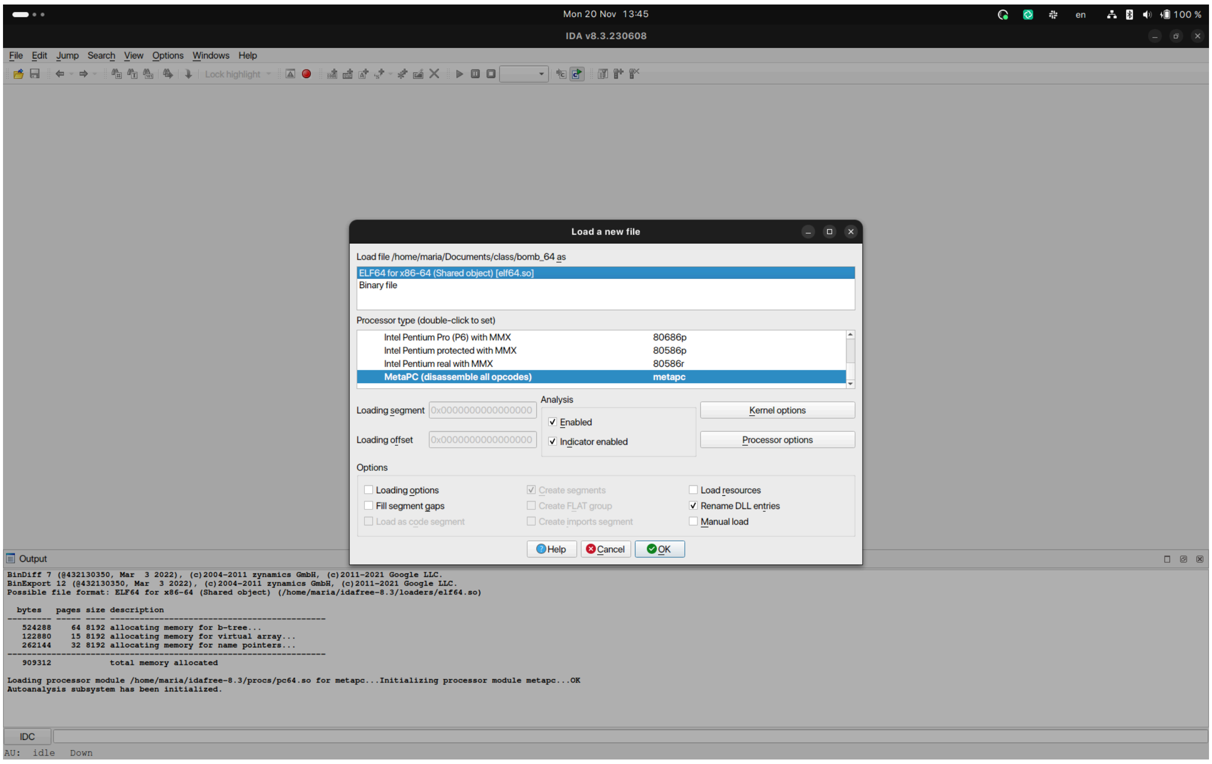
- `cd ~/reversing-class/second_example`
- `chmod +x bomb_64`
- `./bomb_64`
  - Put some data in the input and press Enter
- Run the command "file":
  - `file bomb_64`

```
bomb_64: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=7dd166a66acce52fc6103bbf61a0c32b7e667841, for GNU/Linux
3.2.0, with debug_info, not stripped
```

- Run the command `strings`:
  - `strings bomb_64 | more`

Open IDAFree and **load** the binary file. Press "**OK**" on the first screen and "**Yes**" on the next window.

[10] http://csapp.cs.cmu.edu/3e/labs.html
[11] https://hex-rays.com/ida-free/#download
[12] https://p.ost2.fyi/courses/course-v1:OpenSecurityTraining2+Arch1001_x86-64_Asm+2021_v1/about



You will be greeted with the decompilation view of the **main** function.

1. Press "**space**" to see the disassembled function listing. Pressing "**space**" again will return to the Graph view.
2. If you double-click on a variable or a function name, it will take you to its definition. Pressing "**Esc**" takes you back to the previous function.
3. There are multiple views:
   a. "Imports" shows the imported libraries
   b. "Exports" shows the exported functions.
4. **Shift-F12** opens the "Strings" view
5. **Shift-F7** opens the "Segments" view.
6. You can find all available views in **View** -> **Open subviews**
7. Pressing **F5** while in a disassembly window (IDA View-A) opens the decompilation view.

You can discover more shortcuts in this cheatsheet[13].

---

13 https://elhacker.info/Cursos/%5BTutsNode.com%5D%20-%20Reverse%20Engineering%20IDA%20For%20Beginners/04%20Basic%20File%20Analysis%20and%20IDA%20Usage/008%20IDA%20Cheatsheet-v3.pdf

Note: If F5 does not work due to networking issues (cloud decompilation), close IDA, download the database from here, and save it in the same folder as the binary. Then, open the database file either by double-clicking it (Windows) or by opening it through IDA.

Main function

The list of disassembled functions is on the left panel. If you scroll up and down you can see

- libc functions
- Bomb related functions (phase_1, main. Phase_defused, etc)

Let's first look at the `main` function. The first part checks the input arguments:

- If there is no extra argument, the program will read from the standard input.
- If there is more than 1 argument, it will print the usage
- If there is one argument, it expects it to be a file and it will attempt to read it.

```
if ( argc == 1 )
{
  infile = (FILE *)stdin;
}
else
{
  if ( argc != 2 )
  {
    __printf_chk(1LL, "Usage: %s [<input_file>]\n", *argv);
    exit(8);
  }
  infile = fopen(argv[1], "r");
  if ( !infile )
  {
    __printf_chk(1LL, "%s: Error: Couldn't open %s\n", *argv, argv[1]);
    exit(8);
  }
}
```

In the second part there is an initialization function and some messages that are printed on screen using `puts`. Then the program reads a line (`read_line`) and then goes through the 6 phases.

After each phase there is a `phase_defused` function.

```
initialize_bomb();
puts("Welcome to my fiendish little bomb. You have 6 phases with");
puts("which to blow yourself up. Have a nice day!");
line = (const char *)read_line();
phase_1((__int64)line);
phase_defused(line);
puts("Phase 1 defused. How about the next one?");
v4 = (const char *)read_line();
phase_2((__int64)v4);
phase_defused(v4);
puts("That's number 2.  Keep going!");
v5 = (const char *)read_line();
phase_3((__int64)v5);
phase_defused(v5);
puts("Halfway there!");
v6 = (const char *)read_line();
phase_4((__int64)v6);
phase_defused(v6);
puts("So you got that one.  Try this one.");
v7 = (const char *)read_line();
phase_5((__int64)v7);
phase_defused(v7);
puts("Good work!  On to the next...");
v8 = (const char *)read_line();
phase_6((__int64)v8);
phase_defused(v8);
return 0;
```

initialize_bomb

It defines a signal handler for signal[14] 2 (SIGINT[15]) that defines how to behave when "Ctrl-C" is pressed:

```
_sighandler_t initialize_bomb()

return signal(2, sig_handler);
```

This is the signal_handler

[14] https://www.man7.org/linux/man-pages/man2/signal.2.html
[15] https://www.man7.org/linux/man-pages/man7/signal.7.html

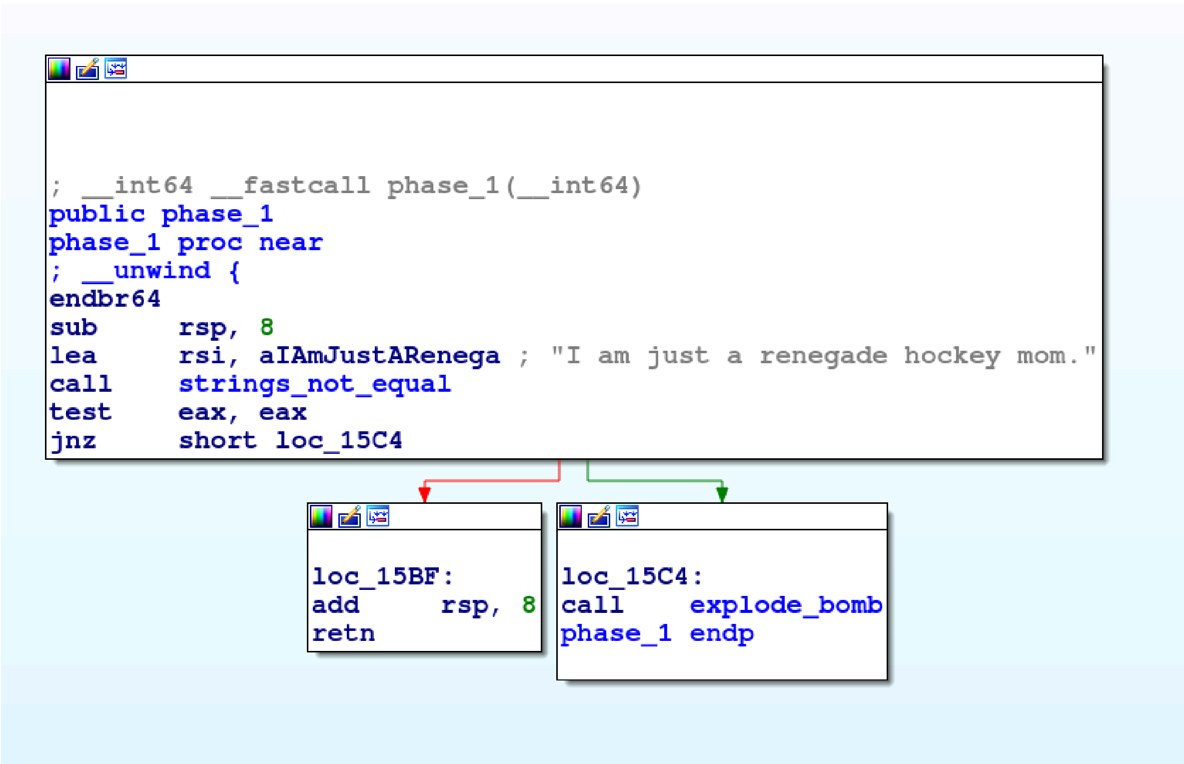

```
void __noreturn sig_handler()
{
  puts("So you think you can stop the bomb with ctrl-c, do you?");
  sleep(3u);
  __printf_chk(1LL, "Well...");
  fflush(_bss_start);
  sleep(1u);
  puts("OK. :-)");
  exit(16);
}
```

We can test this by running the bomb and pressing Ctrl-C instead of giving the input.

```
./bomb_64

Welcome to my fiendish little bomb. You have 6 phases with

which to blow yourself up. Have a nice day!

^CSo you think you can stop the bomb with ctrl-c, do you?

Well...OK. :-)
```

Phase 1

Double-click on the **phase_1** function call and view the listing:

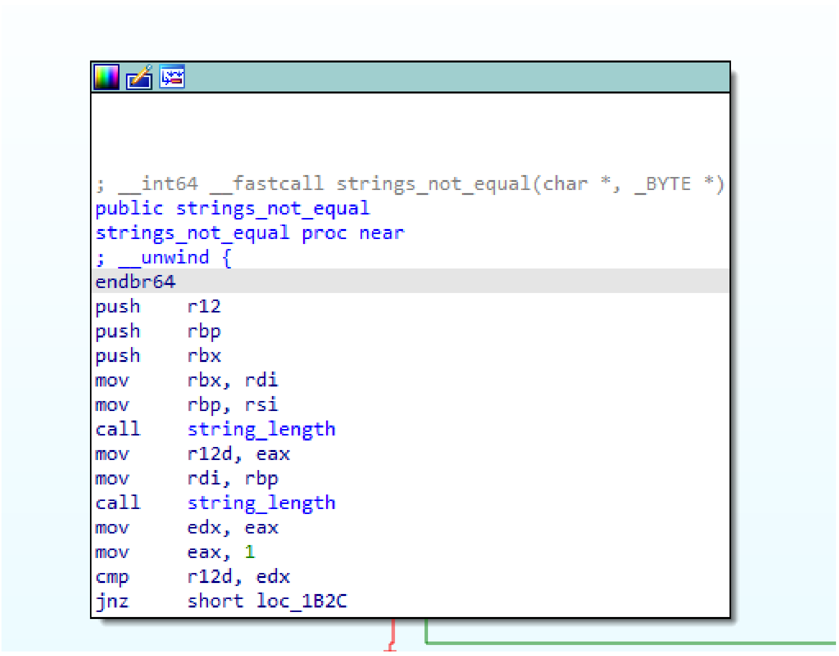


We can just look at the decompiled version or even figure the solution out from the graph but that wouldn't be fun. So let's do this by looking at the disassembly only! Step by step...

Students' faces at this moment

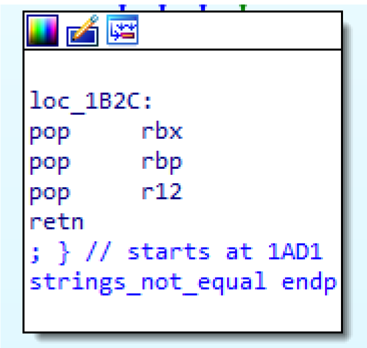
We notice that there is a call to the *strings_not_equal* function. Let's follow it by double-clicking on it.

We see something like this:



What are the first three **push** commands for?

This is what we call **prologue** of a function. At the **epilogue** of a function we restore these values **in reverse** order. We do this with **pop** command.


```
loc_1B2C:
pop     rbx
pop     rbp
pop     r12
retn
; } // starts at 1AD1
strings_not_equal endp
```

Now that we know how to recognize prologue and epilogue of a function we can take a look at what is happening inbetween.

We notice that we are moving value of **rdi** register to **rbx**, and **rsi** to **rbp.**

What is written in **rdi** and **rsi**? Maybe we need to backtrack a bit...

- Let's go back to *phase_1* function by pressing **Esc** on your keyboards.
- Notice that before the function call we put something in the **rsi** register.
    - It is a string: "I am just a renegade hockey mom."
- But what is in **rdi**?
- Let's go even more back, to the main function
    - Press **Esc** again


```
loc_1483:
call    initialize_bomb
lea     rdi, s          ; "Welcome to my fiendish little bomb. You"...
call    _puts
lea     rdi, aWhichToBlowYou ; "which to blow yourself up. Have a nice "...
call    _puts
call    read_line
mov     rdi, rax
call    phase_1
```

You can notice that we move **rax** to **rdi** before we call the function. So what is in **rax**?

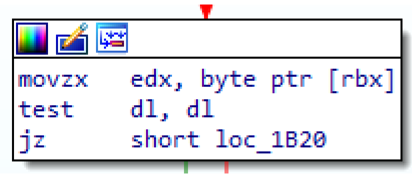
Notice the *read_line* function before **mov.** As we reminded ourselves at the beginning of the class **rax** register is the *return value register*. So what could be a return value of the *read_line* function? Well, most likely our input. So now we know what is in **rdi** and **rsi**.

We can continue with Phase 1.

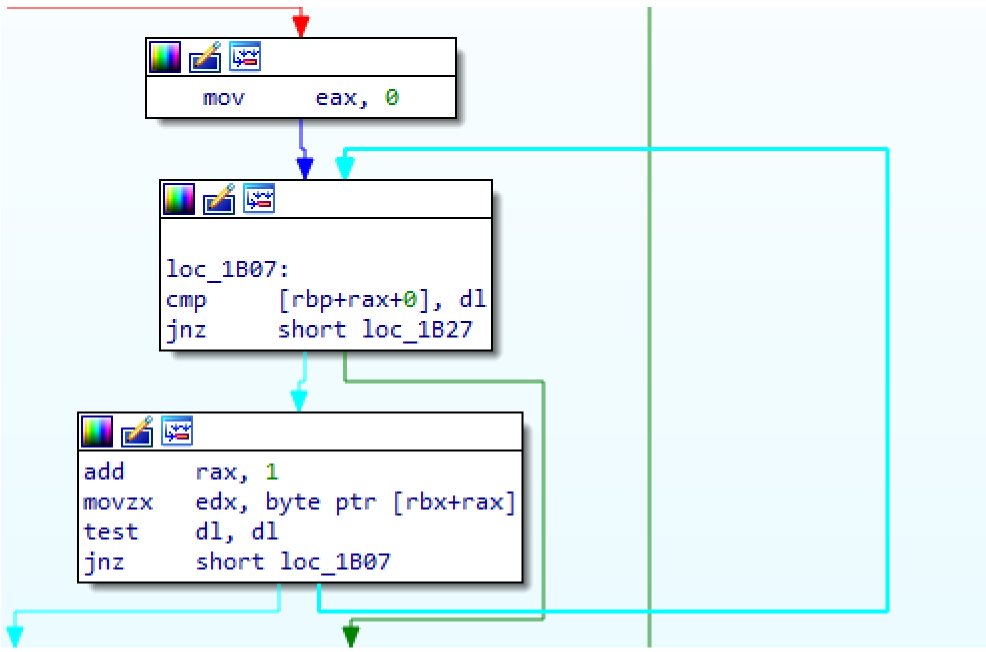
We notice that the next call is to the *string_length* function. Furthermore, this function is called twice. We will not analyze these functions is such great detail (yay), but in a nutshell, they return the length of the string, as the function name suggests.

Once we know the length of both strings, we compare them. The **jnz** command tells us that if the lengths are not equal, we jump to the end of the function and return. Notice that before **cmp** the function writes 1 to **eax.** If the jump happens, 1 is returned from the function.

In the *phase_1* function, there is a **test eax, eax**. If the result is not zero, the bomb explodes. So the strings have to be equal in length.



Then we load the first byte of **rbx** to edx. We then check lower 8-bits of **edx**, **dl,** if the first byte is null. If it is zero is returned. If not, we continue comparing strings.



Notice that there is some loop here. What does it do?

First, it compares the byte of the **edx** register with the byte at **rbp+rax**. If they are not equal, the function jumps, and 1 is returned.

If they are equal **rax** is increased by 1. So we can conclude that **rax**, in this case, holds the value of the counter, and we increment it in each step. This is some kind of while loop or for loop.

If both strings reach the end, and the last comparison of characters, both are **\0,** we return 0 (zero) from the function!

So this function really does compare if two strings are equal. Who would have guessed? 🙂

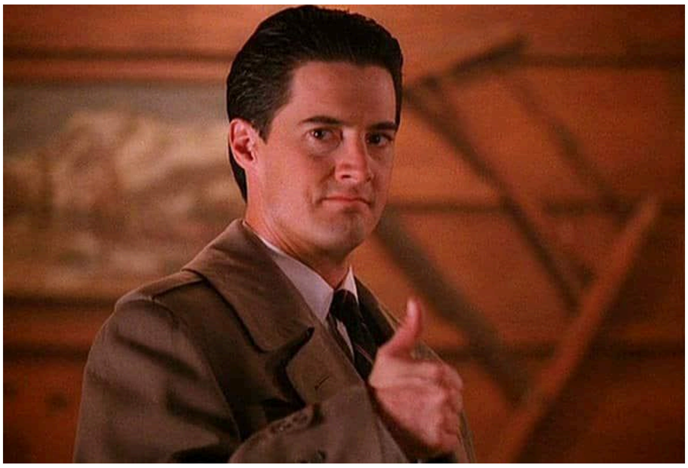
We know that `strings_not_equal` checks if the input is equal to a constant string stored in the memory. Let's try the string:

```
root@bsylabs:~/reversing-class/third_ex$ ./bomb_64

Welcome to my fiendish little bomb. You have 6 phases with

which to blow yourself up. Have a nice day!

I am just a renegade hockey mom.

Phase 1 is defused. How about the next one?
```

Now that we have the correct string we can use the input file to avoid typing.

- `echo "I am just a renegade hockey mom." > bomb64_input.txt`
- `./bomb_64 bomb64_input.txt`

Solution:



Phase 2

Go back to the `main()` function by pressing **Esc** or by selecting the function from the left side window. Double-click on the `phase_2` function and then **F5** to view the decompiled version:

```
unsigned __int64 __fastcall phase_2(__int64 a1)
{
  int *v1; // rbx
  unsigned __int64 result; // rax
  int v3[5]; // [rsp+0h] [rbp-38h] BYREF
  char v4; // [rsp+14h] [rbp-24h] BYREF
  unsigned __int64 v5; // [rsp+18h] [rbp-20h]

  v5 = __readfsqword(0x28u);
  read_six_numbers(a1, v3);
  if ( v3[0] != 1 )
    explode_bomb(a1);
  v1 = v3;
  do
  {
    if ( v1[1] != 2 * *v1 )
      explode_bomb(a1);
    ++v1;
  }
  while ( v1 != (int *)&v4 );
  result = __readfsqword(0x28u) ^ v5;
  if ( result )
    return phase_3(a1);
  return result;
}
```

Phase_2 analysis:

- It calls the function `read_six_numbers`
- `read_six_numbers` uses `sscanf` to read six integers from the input line (a1) and stores them in the a2 array:

```
__int64 __fastcall read_six_numbers(__int64 a1, __int64 a2)
{
  __int64 result; // rax

  result = __isoc99_sscanf(a1, "%d %d %d %d %d %d", a2, a2 + 4, a2 + 8, a2 + 12, a2 + 16, a2 + 20);
  if ( (int)result <= 5 )
    explode_bomb(a1);
  return result;
}
```

- After that, the phase_2 function checks if the first number in the array is equal to 1.
- Then the remaining numbers are expected to be the previous number multiplied by 2 (while loop)

- `echo "                    " >> bomb64_input.txt`

Solution:

Phase 3

The assembly graph has a very interesting structure. It is a `switch/case` construct.

The decompiled listing shows us that `sscanf()` is used to read the next input and that two integers are expected. The switch checks the first number (v11).

At the end, there is also a check that requires v11 to be less than 5, otherwise, the bomb will explode.

```
  v13 = __readfsqword(0x28u);
  if ( (int)__isoc99_sscanf(a1, "%d %d", &v11, &v12) <= 1 )
    explode_bomb();
  switch ( v11 )
  {
    case 0:
      v2 = 628;
      goto LABEL_5;
    case 1:
      v2 = 0;
LABEL_5:
      v3 = v2 - 588;
      goto LABEL_6;
    case 2:
      v3 = 0;
LABEL_6:
      v4 = v3 + 688;
      goto LABEL_7;
    case 3:
      v4 = 0;
LABEL_7:
      v5 = v4 - 126;
      goto LABEL_8;
    case 4:
      v5 = 0;
LABEL_8:
      v6 = v5 + 126;
      goto LABEL_9;
    case 5:
      v6 = 0;
LABEL_9:
      v7 = v6 - 126;
      goto LABEL_10;
    case 6:
      v7 = 0;
LABEL_10:
      v8 = v7 + 126;
      break;
    case 7:
      v8 = 0;
      break;
    default:
      explode_bomb();
  }
  v9 = v8 - 126;
  if ( v11 > 5 || v12 != v9 )
    explode_bomb();
  result = __readfsqword(0x28u) ^ v13;
  if ( result )
    return func4(a1, (__int64)"%d %d", v1);
  return result;
}
```


We can work with the value 4 for the first number to minimize the number of jumps and calculate the required value of the second number.

We can also use GDB to see how we can easily find the value of v12 without any calculations. Let's put the values 4 and 100 to the `bomb64_input.txt` and then run GDB:

- `echo "4 100" >> bomb64_input.txt`
- `gdb ./bomb_64`
- `break phase_3`
- `run < bomb64_input.txt`
- `disassemble`

Set another breakpoint before the check to explode the bomb.

- `b *0x<address_of_instruction>`
- `continue`
- `ni`
    - ni: Run until next instruction. Steps over calls.

Just before the second `cmp` instruction we can check the contents of the `rax` register that has the expected value. We can set the value in the stack to pass the check:

- `x/x $rsp+4`
- `set *<address_of_rsp+4>=0`
- `x/x $rsp+4`
- `c`

Solution:

~~~~ 💗 Second Break! 💗~~~~ (16:40, 10m)

Phase 4 (16:50)

Looking at the `sscanf()` function, the main function of `phase_4()` expects two integers. The first one (v3) needs to be less or equal to 14 (0xE).

Then it calls another function `func4()`, with v3 as a parameter and expects the result to be 10. In the same line, the second integer (v4) is checked as well, and the expected result is also 10.

```
if ( (unsigned int)__isoc99_sscanf(a1, "%d %d", &v3, &v4) != 2 || v3 > 0xE )
  explode_bomb();
v1 = v3;
if ( (unsigned int)func4(v3, 0LL, 14LL) != 10 || v4 != 10 )
  explode_bomb();
```

This is what the decompiled listing for `func4()` looks like:

```
IDA View-A        Pseudocode-A        Hex View-1

__int64 __fastcall func4(__int64 a1, __int64 a2, __int64 a3)
{
  int v3; // ebx

  v3 = a2 + ((int)a3 - (int)a2) / 2;
  if ( v3 > (int)a1 )
  {
    v3 += func4(a1, a2, (unsigned int)(v3 - 1));
  }
  else if ( v3 < (int)a1 )
  {
    v3 += func4(a1, (unsigned int)(v3 + 1), a3);
  }
  return (unsigned int)v3;
}
```

It is a recursive function, and it can be tricky to understand. But we can always re-write it in Python and brute-force which input value returns the value 10:

- `python3 phase4.py`

Solution:

Phase 5

```
  v7 = __readfsqword(0x28u);
  if ( (int)__isoc99_sscanf(a1, "%d %d", &v5, &v6) <= 1 )
    explode_bomb(a1);
  v1 = v5 & 0xF;
  v5 = v1;
  if ( v1 == 0xF )
    goto LABEL_7;
  v2 = 0;
  v3 = 0;
  do
  {
    ++v3;
    v1 = array_3471[v1];
    v2 += v1;
  }
  while ( v1 != 15 );
  v5 = 0xF;
  if ( v3 != 15 || v6 != v2 )
LABEL_7:
    explode_bomb(a1);
```

The level expects two integers (v5 and v6). The first one (v5) needs to be less than 15 (0xF). The main loop replaces the first number with the contents of `array_3471`, which is an array of 16 numbers from 0 to 15. None of the numbers is repeated.

Double-click on the array to see its contents:

```
.rodata:00000000000031C0 ; _DWORD array_3471[16]
.rodata:00000000000031C0 array_3471      dd 0Ah, 2, 0Eh, 7, 8, 0Ch, 0Fh, 0Bh, 0, 4, 1, 0Dh, 3, 9
.rodata:00000000000031C0                                          ; DATA XREF: phase_5+49↑o
.rodata:00000000000031F8                 dd 6, 5
```

The loop uses v1 as an index to the array, it cycles through all the values and stops when the value of v1 becomes 0xF. The check expects that the loop is run 15 times and that the sum of the numbers is equal to the second integer input.

There are multiple ways we can work with this. We can use pen and paper, Python, or GDB. But if we reason about it, we can quickly find out the first number.

Notice that:

a) The loop has to be executed 15 times
b) The last value has to be 0xF.
c) We need to find where to start to end up at 0xF
d) The code goes through all the elements of the array. If it didn't and there was a repetition, it could not have reached 0xF in 15 steps.
e) If we allow it to run more than 15 steps it will repeat the cycle.
f) Doesn't that mean that the start number should be where 0xF points to (5)?

The second number is the sum of all the numbers we go through in the loop. Is it the sum of the whole array (120)? We can use GDB to find out:

We set v5 to 5, and we will set v6 to 120 to start the analysis:

- `echo "5 120" >> bomb64_input.txt`
- `gdb ./bomb_64`
- `break phase_5`
- `run < bomb64_input.txt`
- `ni`
- `disassemble`

Find the address just before the two final cmp instructions and add a breakpoint:

- `b *<address_before_two_cmp>`
- `continue`
- `ni`
- `x/x $rsp+4`

Check the value of the `ecx` register to see what is expected:

- `p $ecx`
- `set *<address_of_rsp+4>=0x73`

Solution:

Phase 6

This one is the most complicated.

After some renaming of variables, the first part of the listing looks like this:

```
v23 = __readfsqword(0x28u);
arr = input_array;
read_six_numbers(a1, (__int64)input_array);
for ( i = 1LL; ; ++i )
{
  if ( (unsigned int)(*arr - 1) > 5 )
    explode_bomb(a1);
  if ( (int)i > 5 )
    break;
  ctr = i;
  do
  {
    if ( *arr == input_array[ctr] )
      explode_bomb(a1);
    ++ctr;
  }
  while ( (int)ctr <= 5 );
  ++arr;
}
```

The input is expected to be 6 integers. What the first loop is doing, is checking that all numbers are less than or equal to 6 and that they are all different from each other.

So we are looking at the permutation of the numbers 1-6.

The key to understanding the rest of the code is to first identify the **node1** struct that we see in the listing.

- Double-click on the **node1** in IDA
- The variable is stored in the **.data** section along with another 5 nodes.
- It is a struct with the following format:

```
struct node {
      int value;
      int id;
      struct node *next;
};
```


This is how the node structs look like in memory, using GDB:

```
gef➤  x/20x 0x0000558821125200
0x558821125200 <node1>: 0x00000212      0x00000001      0x21125210      0x00005588
0x558821125210 <node2>: 0x000001c2      0x00000002      0x21125220      0x00005588
0x558821125220 <node3>: 0x00000215      0x00000003      0x21125230      0x00005588
0x558821125230 <node4>: 0x00000393      0x00000004      0x21125240      0x00005588
0x558821125240 <node5>: 0x000003a7      0x00000005      0x21125110      0x00005588
gef➤  x/4x 0x0000558821125110
0x558821125110 <node6>: 0x00000200      0x00000006      0x00000000      0x00000000
```

IDA allows us to create custom structs and use them to guide the disassembly and decompilation. We will need to go to the **Structures** tab and press Insert to add a new struct.

We will name it “`node_struct`”. In order to add members to the struct we can click on the last line and press the ‘d’ button three times to add an integer (dd). We repeat the process to add the second integer. To add the pointer, we need to press ‘d’ 4 times (dq).

db -> byte

dw -> word

dd - > double word (32 bit)

dq -> quad word (64 bit)

We can rename the variables by pressing “N”.

```
00000000 node_struct      struc ; (sizeof=0x10, mappedto_29)
00000000 value           dd ?
00000004 id               dd ?
00000008 ptr              dq ?
00000010 node_struct      ends
00000010
```

- In the code listing on line 50, right-click on the `v7` variable, select “**Convert to struct***” and then select `node_struct`.
- Do the same with the `v8` variable.

The `do…while` loop at the end is very interesting. It goes through the linked list and checks if the value of the next `node` is higher than the current node. And it explodes if it is:

```
do
{
  if ( v8->value < *(_DWORD *)v8->ptr )
    explode_bomb(a1);
  v8 = (node_struct *)v8->ptr;
  --v14;
}
while ( v14 );
```

What if we just need to order the nodes in descending order of their values?

Solution:

Assignment 7 (4 Points)

1. The assignment has two parts. 2 points each.
2. The first part will test your prowess with the network.
3. The second part will test your skills in reversing binaries.

Class Feedback

By giving us feedback after each class, we can make the next class even better!

bit.ly/BSYFeedback