

Solution:

First, we tried to run the exe file (rop.exe):

```
C:\Users\harel\Downloads\buffer overflow - ex3-20250511T231315Z-1-001\buffer overflow - ex3>rop.exe  
Please provide an hex string
```

We can deduce from the message that the program takes some arguments (more specifically, a hex string).

Now we will dive into the source code (in the file rop.cpp).

First, we can see 3 gadgets:

```
pop eax  
ret  
pop ecx  
ret  
mov [eax], ecx  
ret
```

Of course, to use them, we will need to find where they are located in the memory, but for now, let's find an entry point to take advantage of ROP techniques.

In the main, we call this function:

```
void unhexlify(char* dst, const char* src, size_t length)  
{  
    char hexBuffer[3] = { 0 };  
    for (size_t i = 0; i < length / 2; ++i) {  
        memcpy(hexBuffer, src + 2 * i, 2);  
        sscanf_s(hexBuffer, "%02hhX", dst + i);  
    }  
}
```

The function unhexlify gets three arguments -

1. dst - a pointer to IBuffer
2. src - a pointer to argv[1]
3. length - the length of argv[1]

The function unhexlify writes binary data into IBuffer without checking the destination size.

The input is passed via argv[1], which is a hex-encoded string.

Each 2 hex characters are converted into 1 byte and written into IBuffer.

No length check is performed to ensure that $\text{length} / 2 \leq 10$ (the size of IBuffer).

This allows an attacker to overwrite the return address on the stack, enabling arbitrary code execution through ROP

So we learn that our entry point is through argv[1].

Now we need to build the correct input to cause a buffer overflow using ROP.

Notice that the overall string should be in the form of:

```
<ID in ascii><Padding><pop eax gadget><ID address><pop ecx gadget><&IBuffer>mov  
gadget><&printf>
```

We need to clarify a few things:

1. Why do we need each part of the input
2. Where to find each part of the input

3. Why the parts in this order

Let's start with "why the parts in this order" - first we need to load the string we want to print (and add padding to match the architecture of the system), after that we want to load the address of this string into eax, then load the buffer address to ecx and move our string to the buffer (now the string is inside the buffer in eax) and now we call printf which will print our string to the console.

This also answers the question "why do we need each part of the input?"

We need to clarify one more thing, "where to find each part of the input" - to find all the addresses, we used VS and looked in the disassembly of the exe file, where we found all the addresses of the gadgets:

```
004605A9  pop          eax
004605AA  ret
004605AB  pop          ecx
004605AC  ret
004605AD  mov          dword ptr [eax],ecx
004605AF  ret  ►|
```

To find the other addresses, we enter the unhexlify function and look at the pointers:

Name	Value
► dst	0x0019fedc
► hexBuffer	0x0019fe74
length	480
► src	0x008fec08

and the address of printf:

```
Address: printf(const char * const, ...)

▼ Viewing Options
{
|00460800  push      ebp
```

Now we assign the correct values to each part and we are done :)