**CSED211 Lab 4**

20210643 김현준

**<Lab 4>**

Lab4 시간에서는 Assembly language와 Memory Layout에 대해 다루었다. Gcc를 이용하여 컴파일한 실행 파일을 gdb로 disassemble하여 assembly 코드를 확인할 수가 있다. 어셈블리 코드를 살펴보면 각 함수마다 pushq하면서 Prolog부터 시작하여 return하는 형태로 되어 있고 중간의 movq, leaq 등을 이용하여 rdi, rsi, rdx, rcx, r8, r9 같은 resister들의 값을 수정하고, callq로 다른 함수를 호출한다. 함수가 리턴될 때에는 rax에 리턴값을 저장하여 리턴된다. Loop 같은 경우에는 conditional jump와 non-conditional jump들을 활용하여 구현되고, jmp, jle, je, ja 등이 조건에 따라 jump하여 특정 코드의 주소로 이동하도록 하는 명령어이다.

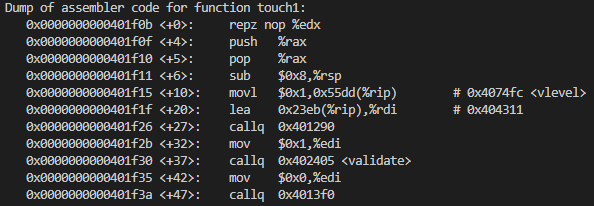
Memory Layout 같은 경우 stack과 heap, uninitialized data bss, initialized data, text로 이루어져 있고, Stack Frame에는 active subroutines들이 저장된다. Stack에는 parameter들과 return address, old frame pointer, local variable들이 들어가게 되고, 이것들을 관리하는 데 rbp(base pointer)와 rsp(stack pointer)가 중요한 역할을 하게 된다.

Lab4에 이어서 Lab5를 통해, Attacklab에서 꼭 필요한 buffer overflow에 대해 배웠다. Memory Layout에서 Stack Frame에서 rsp 위치에 buffer가 있고, 특정 bytes가 buffer에 할당되어 있을 때, 해당 buffer를 넘기는 size의 입력을 넣었을 때에는 다음 함수를 호출하는 부분의 stack을 침범하게 된다. 따라서 해당 부분의 값을 마음대로 수정할 수가 있고, 이를 이용하여 원하는 함수를 호출시키는 code injection attack을 할 수 있다.

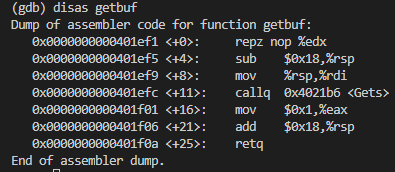
Lab6에서는 Attacklab에서 rtarget을 attack하기 위한 ROP에 대해 배웠다. Code injection attack의 경우 Stack Canary, DEP/NX, ASLR 등의 방어책으로 어려워졌는데, Return Oriented Programming은 gadget과 리턴을 이용하여 원하는 명령어를 사용하여 attack을 할 수 있는 방법이다.

**<Code Injection: Level 1>**

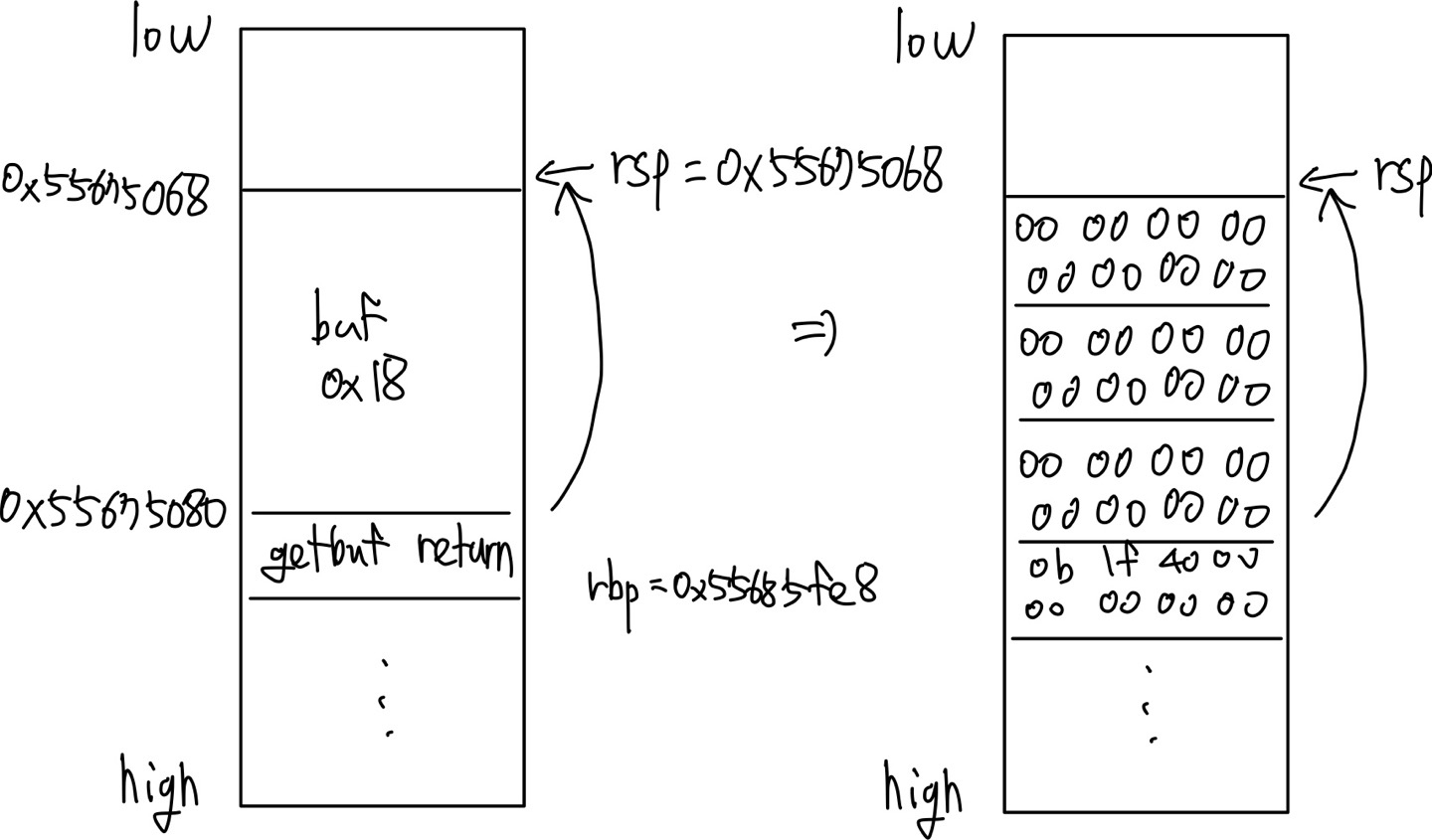
먼저, level1에서는 touch1함수를 호출만 하면 된다. Touch1에는 parameter가 없고 단순히 printf만 하기 때문이다. 따라서 먼저 touch1을 disas해 보았다.



그리고, getbuf를 통해 입력을 받기 때문에, getbuf에서 몇 byte를 입력으로 받으려고 설정해 놓는지를 알아야 했기 때문에, getbuf 함수를 disas해 보았다.

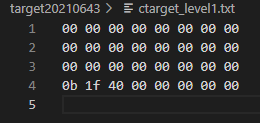


그러자 위와 같이 +4 부분에서 rsp에 0x18을 빼는 것을 확인할 수 있었다. 따라서 입력을 받는 getbuf함수의 buffer 크기가 0x18bytes임을 알 수 있었다. 0x18는 10진수로 24byte이고, 8byte의 배수이다. (8byte씩 할당하므로) 따라서 24byte 만큼을 dummy 값으로 채우고 그 다음 부분에 touch1 함수의 시작 주소를 넣으면, getbuf함수에서 호출하는 gets함수가 입력을 받아서 버퍼부터 저장을 하고, getbuf가 리턴할 때에 원래 리턴해야 할 곳으로 돌아가는 것이 아니라 touch1으로 가게 될 것이다.

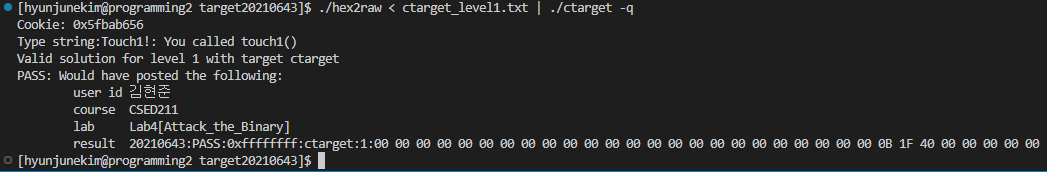


(low가 위, high가 아래로 그렸습니다.)

rbp와 rsp에 주목하여 ctarget을 돌려보면서 값을 확인하였고, Stack frame을 위와 같이 생각해볼 수 있었다. Buffer의 size가 10진수로 24만큼이므로, 이때 rbp=0x55685fe8이고, rsp=0x55675068로 0x18만큼 빼졌고, rsp부터 입력이 들어와 getbuf의 return address 부분에 touch1의 시작 address를 넣을 수 있도록 구현하면 될 것이라고 생각하였다.



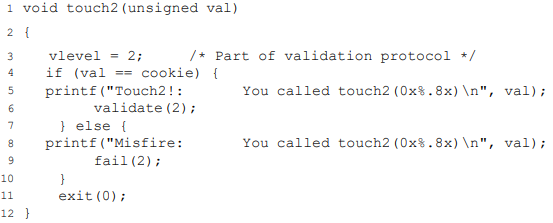
따라서, ctarget\_level1.txt에 {00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0b 1f 40 00 00 00 00 00 }을 입력해주어 실행해보았다.



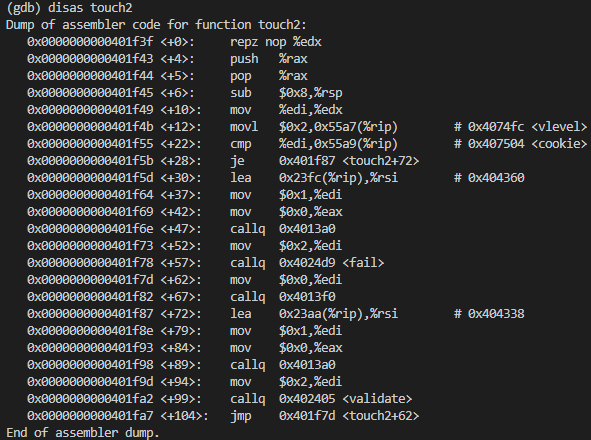
그 결과 PASS가 된 것을 확인할 수 있었다. 계획한 대로 touch1함수를 실행시킬 수 있었다.

**<Code Injection: Level 2>**

Level 2에서는 touch2함수를 실행시켜야 한다.



그런데 touch2 함수는 단순히 호출을 하는 것이 아니라 cookie값을 찾아서 같이 넘겨주어야 한다는 것을 확인할 수 있었다.



Touch2 함수를 disas해보니 위와 같았고, edi에는 C code 상의 val이 들어가게 될 것이고, +22부분의 cmp에서 C code 상의 val과 cookie 비교를 진행할 것으로 생각하였다. 따라서, rdi 레지스터에 cookie 값을 저장하고, touch2 함수를 호출시키면, touch2에서 edi에 cookie 값이 들어가 있을 것이고, val과 비교하여 같을 시, 제대로 작동이 되는 일이 일어날 것이다. 그런데 rdi에 cookie 값을 넣어 주기 위해서는 이를 실행할 명령어들이 필요했다.



먼저 cookie 값을 확인해보니 0x5fbab656이었다. 그리고, touch2의 시작 주소는 0x401f3f이므로, 0x401f3f를 push하고 rdi에 cookie값인 0x5fbab656을 mov해주고 리턴 하는 어셈블리 코드를 level2\_cookie.s라는 파일을 만들어 작성해보았다.

=============

pushq $0x401f3f

movq 0x5fbab656, $rdi

retq

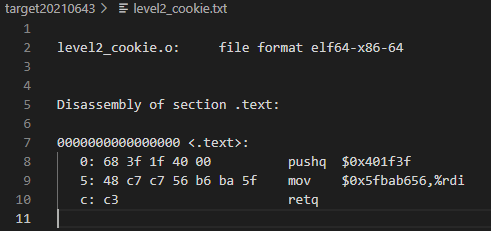
=============



그리고 이를 gcc를 이용하여 level2\_cookie.o 파일을 생성하였다. 그런데 작성한 어셈블리 코드의 machine code를 알기 위해서는 .o 파일을 disassemble해 주어야 했다.



그래서 objdump로 level2\_cookie.txt파일에 disassemble하였다.



================

68 3f 1f 40 00

48 c7 c7 56 b6 ba 5f

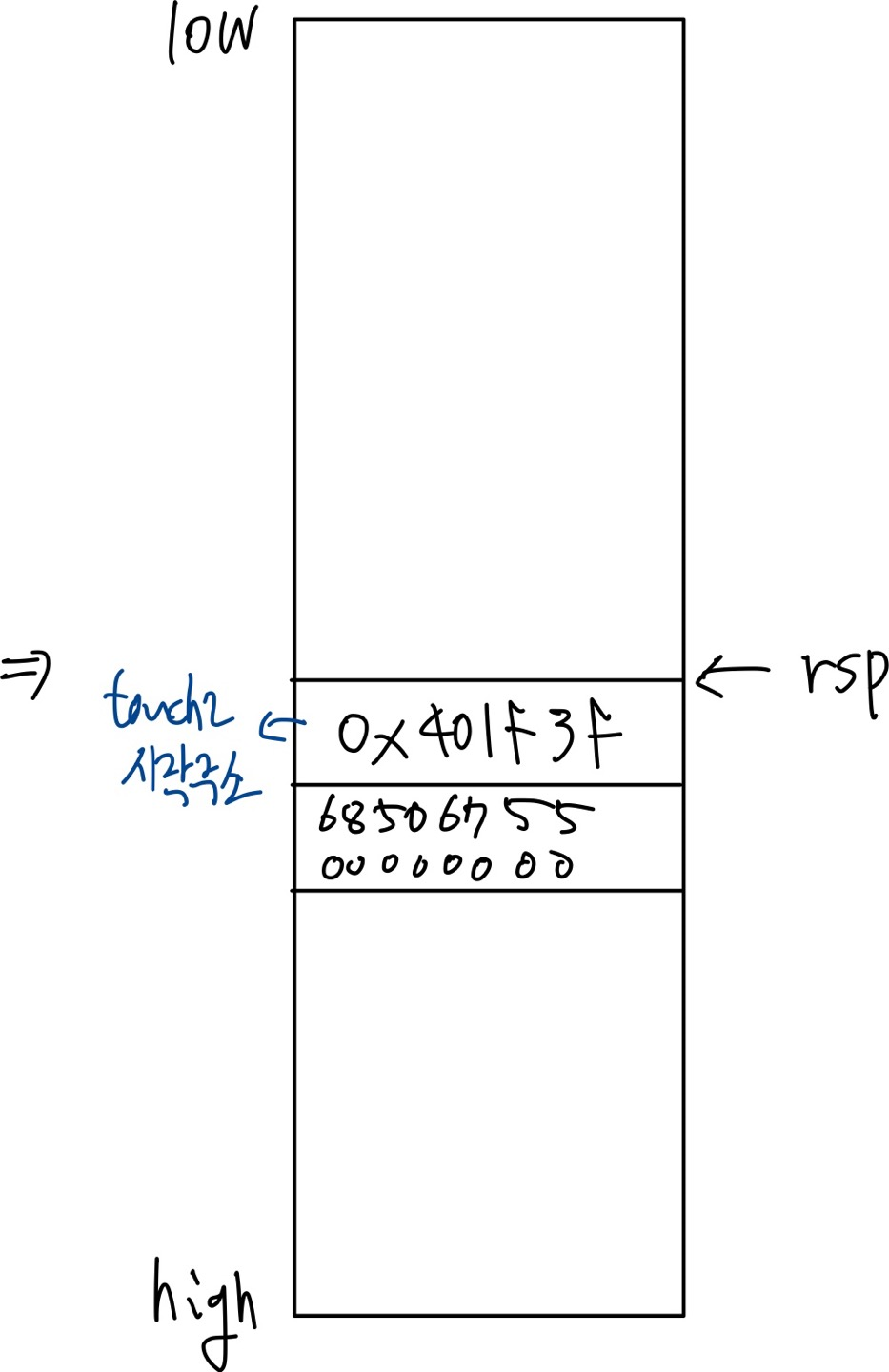
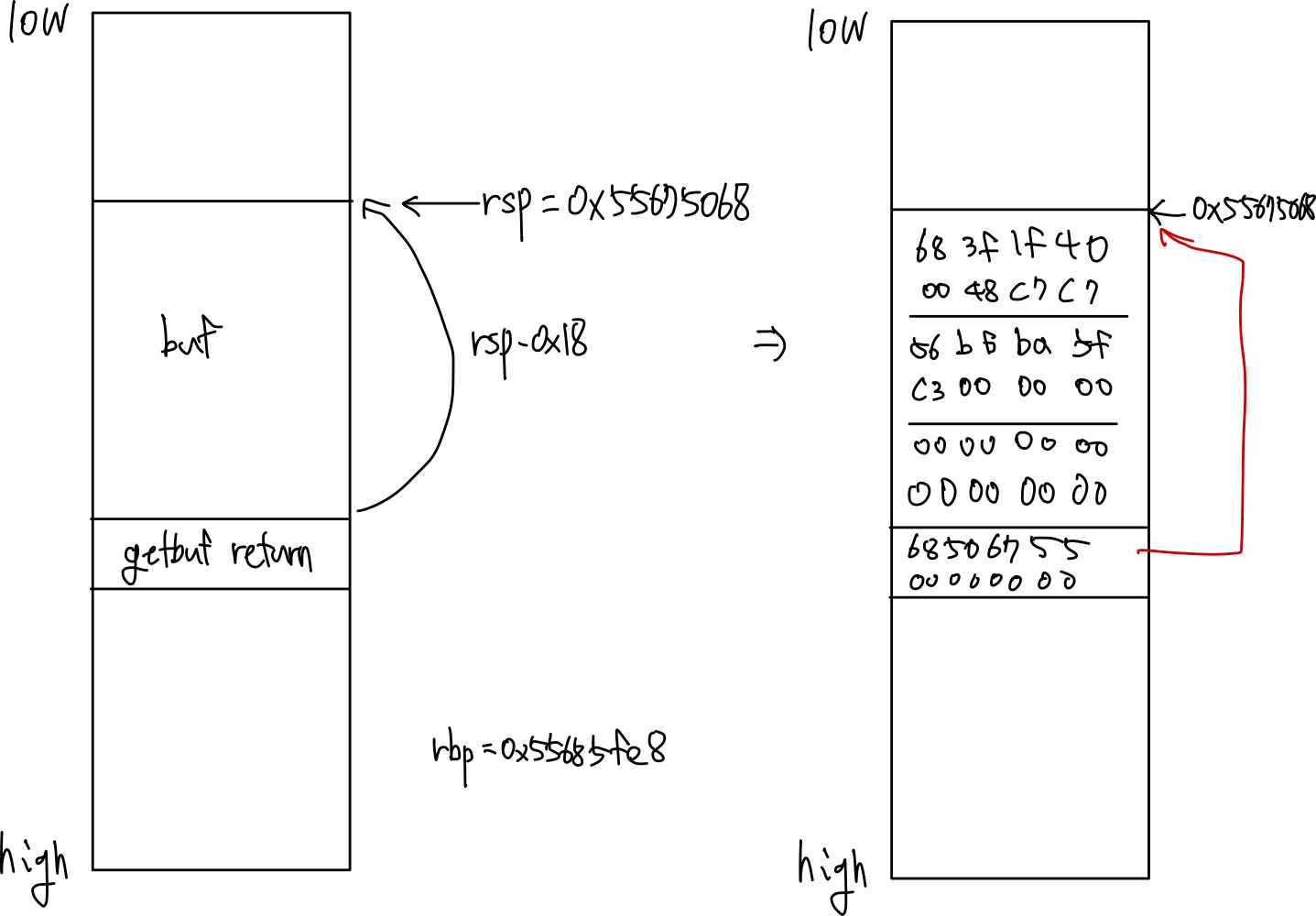
C3

================

그 결과 위와 같은 machine code를 얻을 수 있었다. 따라서 이 machine코드를 입력하여 buffer에 넣어주려고 하였다. 그런데 이 code가 실행되기 위해서는 getbuf가 리턴될 때 이 machine code가 있는 부분으로 이동하여야 한다. 그런데 이 부분은 getbuf가 실행 중일 때 18만큼 할당하고 나서의 rsp값이다. Mov로 rdi로 값을 옮겨 놓으니, rdi에 저장하고 있는 값이기도 하다.

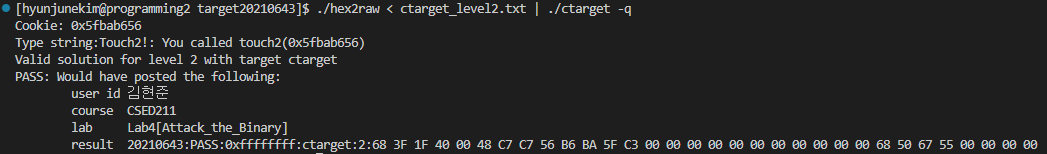


코드를 돌려 이를 확인하니 0x55675068이었다. 따라서, 실행하고자 하는 machine code를 실행시키려면 0x55675068부분으로 리턴하여 코드를 실행해야 하므로, 결론적으로 machine code와 dummy 값으로 0x18의 buffer를 채우고, 리턴 부분에 0x55675068을 넣어주면 될 것으로 생각하였다.



(low가 위, high가 아래로 그렸습니다.)

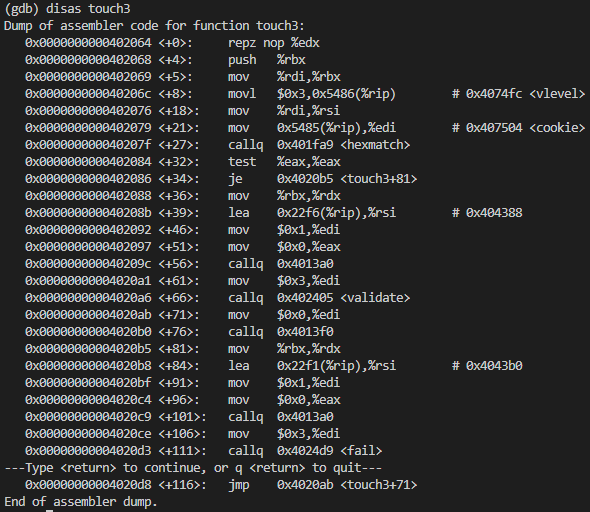
그래서 ctarget\_level2.txt 파일에 {68 3f 1f 40 00 48 c7 c7 56 b6 ba 5f c3 00 00 00 00 00 00 00 00 00 00 00 68 50 67 55 00 00 00 00}을 넣고 실행하였다. 그 결과 touch2의 시작주소가 pushq되면서 rsp에서 0x8이 빼지고, 다음 코드들이 실행되며, 이후 stack에 push된 0x401f3f로 cookie 값이 저장된 rdi를 갖고 넘어가게 될 것으로 생각하였다.



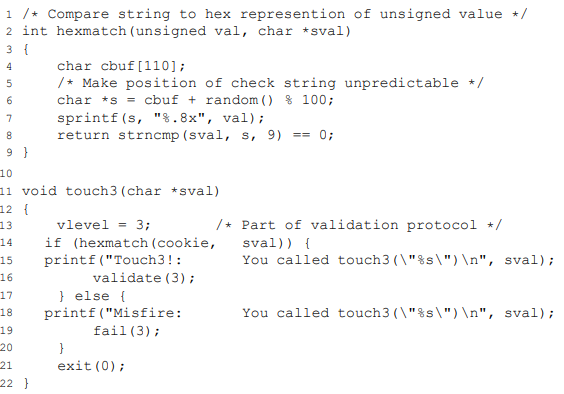
그 결과 PASS가 나오는 것을 확인할 수 있었다. Touch2 함수가 제대로 작동되었으므로, ctarget의 level2가 해결되었다.

**<Code Injection: Level 3>**

Level 3의 경우 touch3 함수를 실행해야 하고, touch3의 경우 hexmatch라는 함수를 부르는 형태로 구성되어 있었다. 형태를 보니 특정 문자열을 전달해주어야 통과할 수 있을 것으로 생각하였다. 다음으로 touch3을 disas해 보았다.



결과는 위와 같고, 구조를 보면 rdi에 특정 문자열의 주소를 저장한 다음, hexmatch함수를 호출하게 됨을 추측할 수 있었다.



C 코드를 보면 위와 같으므로, hexmatch 함수에 문자열을 보내 hex 형태로 cookie와 비교한다는 것으로 추측할 수 있었다. 그런데, 이전에 확인한 cookie의 값은 0x5fbab656인데, 이것을 hex 형태로 바꾸어 주어야 했다. ASCII 코드표를 참고하여 cookie를 hex로 바꾸어 보면, {35 66 62 61 62 36 35 36 00}이었다. 마지막에 00은 null character를 넣어줘야 하기 때문에 추가하였다.

지금까지의 내용을 종합해 보면, rdi에 cookie 문자열의 주소를 넣어서 touch3을 호출하면 되기 때문에, buffer에 먼저 rdi에 cookie 문자열 주소를 넣어주는 machine code를 넣고, 그 다음에 buffer를 dummy 값으로 채운 다음, getbuf의 리턴 자리에 그 machine code의 시작 주소를 넣어서 코드를 돌려준 다음, getbuf return 자리+0x08 위치에 touch3으로 리턴할 수 있도록 touch3의 시작 주소인 0x402064를, 그리고 +0x10 자리에 cookie 문자열을 넣어주면 될 것으로 생각하였다.

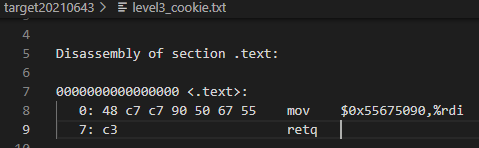
=================

movq $0x55675090, %rdi

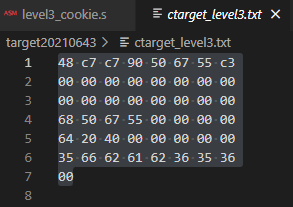
retq

=================

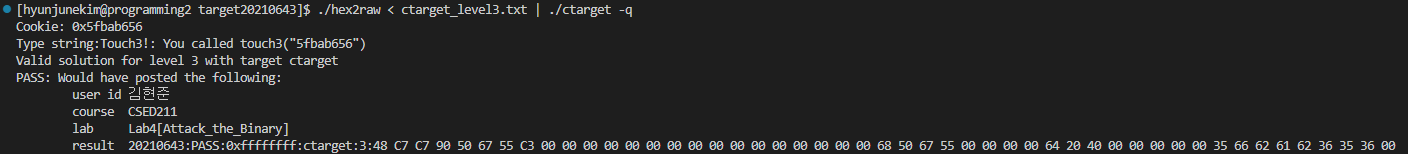
따라서 먼저 위와 같이 machine code를 짜 주었고, level2를 할 때와 마찬가지로 gcc, objdump를 이용하여 level3\_cookie.txt라는 파일에 disassemble된 내용을 넣어주었다.



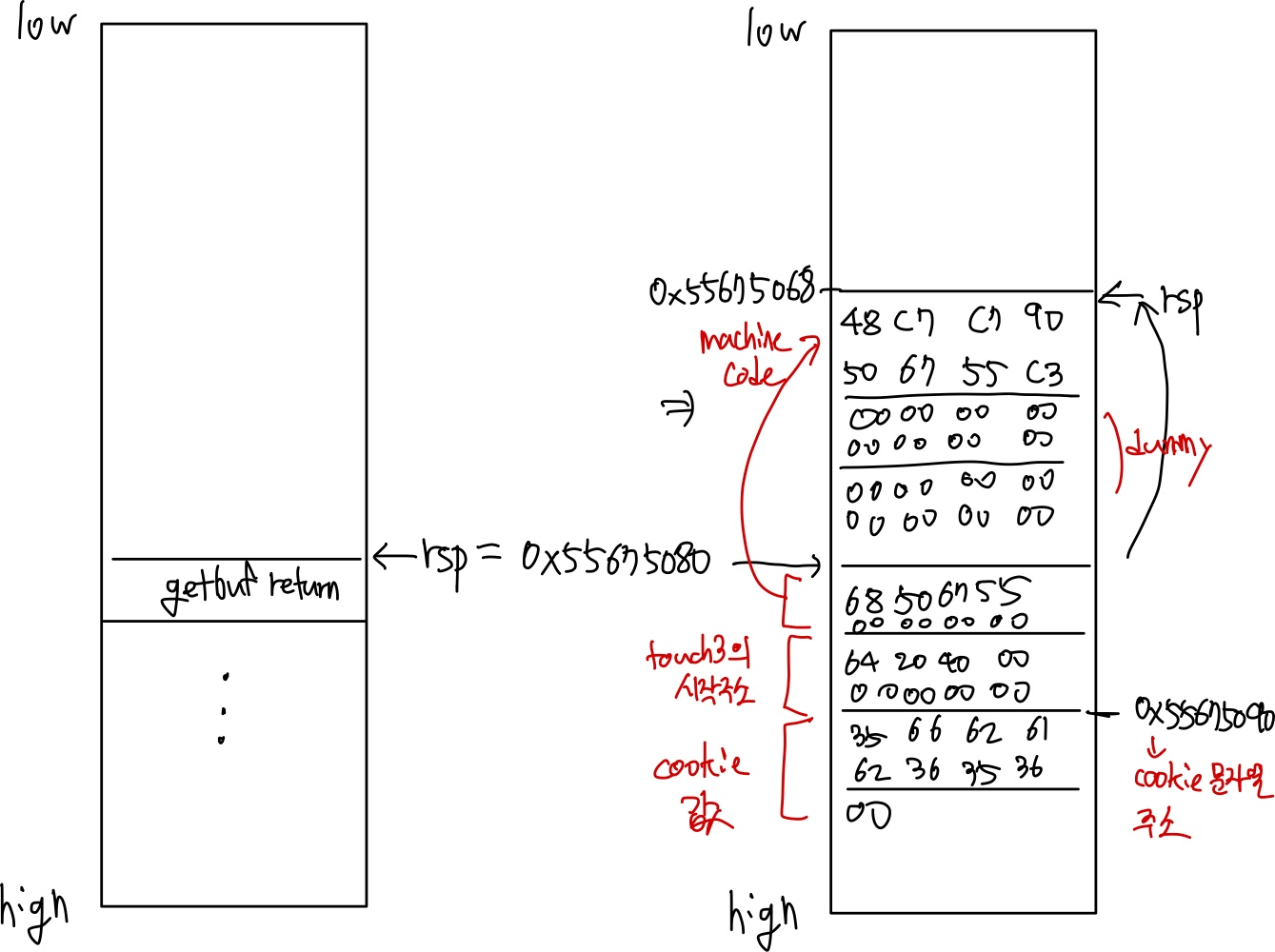
그 결과 {48 c7 c7 90 50 67 55 c3}의 machine code를 얻을 수 있었다. 그리고 cookie 문자열의 주소는 원래 getbuf의 리턴주소가 들어있는 Stack의 주소, 즉 0x55675080에서 +0x10만큼 해준 주소이므로, 0x55675090이 된다. 결론적으로 ctarget\_level3.txt 파일에 {48 c7 c7 90 50 67 55 c3 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 68 50 67 55 00 00 00 00 64 20 40 00 00 00 00 00 35 66 62 61 62 36 35 36 00 }를 넣어 주고, 실행해보았다.



0x5fbab656 => 35 66 62 61 62 36 35 36 00



그 결과 PASS가 나오는 것을 확인할 수 있었다. Ctarget의 level3을 clear하였다.

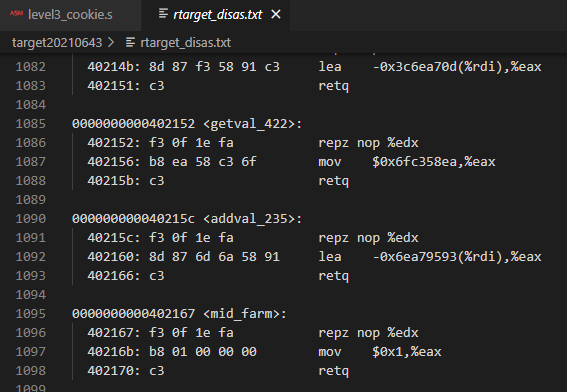


(low가 위, high가 아래로 그렸습니다.)

Stack Frame의 모습은 위와 같이 되었을 것으로 생각하였다.

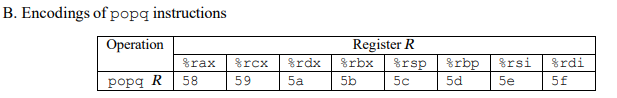
**<Return-Oriented: Level 2>**

Rtarget level 2의 경우 rtarget에 있는 touch2함수를 제대로 실행시키면 된다. ROP attack을 위해 주어진 gadget들이 start\_farm부터 end\_farm까지 들어있다. 따라서 rtarget을 먼저 objdump로 disas하여 어셈블리 및 machine code를 확인하였다.

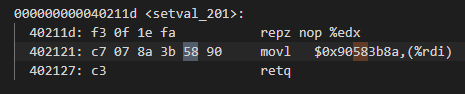


Rtarget\_disas라는 파일을 만들어 objdump -d rtarget > rtarget\_disas로 disas하여 위와 같은 사용 가능한 gadget들을 확인하였다. Touch2 함수를 disas해 보니, ctarget에서의 함수와 일치하는 것으로 보였다.

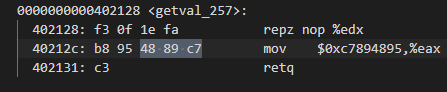
따라서 touch2를 호출하기 위해서는 ctarget때와 비슷하게 cookie 값을 rdi로 넘겨 호출해주어야 한다. 그렇게 하기 위해서 popq명령어를 사용하면 될 것으로 생각하였다. popq는 rsp에 있는 값을 operand에 넣어주고 rsp를 +0x8해주는 명령어이므로, 해당 주소에 cookie값을 bufferoverflow로 넣어주고, rdi로 pop할 수 있는 gadget을 찾으면 될 것으로 생각하였다.



Popq의 machine code가 이러하므로, 5f를 검색해보았는데, 5f 다음 무시가 가능한 90 등을 거쳐 c3으로 리턴되는 것을 찾기 어려웠다. 따라서 rax를 통해서 가는 방법을 시도해보았다.



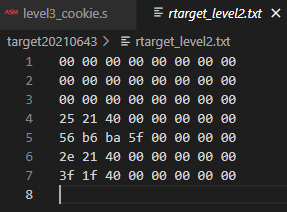
위와 같이 popq %rax에 해당하는 부분을 찾게 되었고, 해당 주소는 0x402121+0x4=0x402125이다.

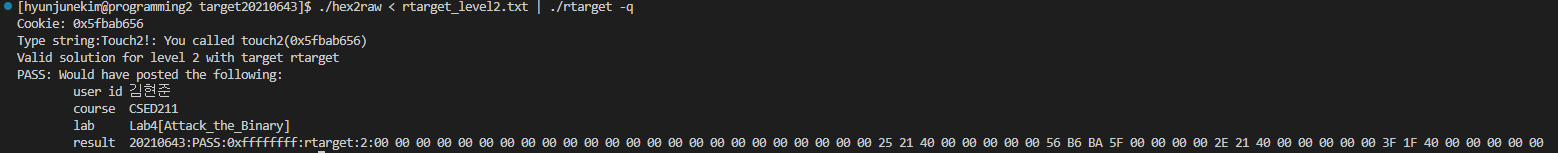


다음으로는 rax를 rdi로 옮겨야 하므로 movq %rax, %rdi에 해당하는 machine code 48 89 c7을 찾았다. 이 코드의 주소는 0x40212c+0x2=0x40212e였다.

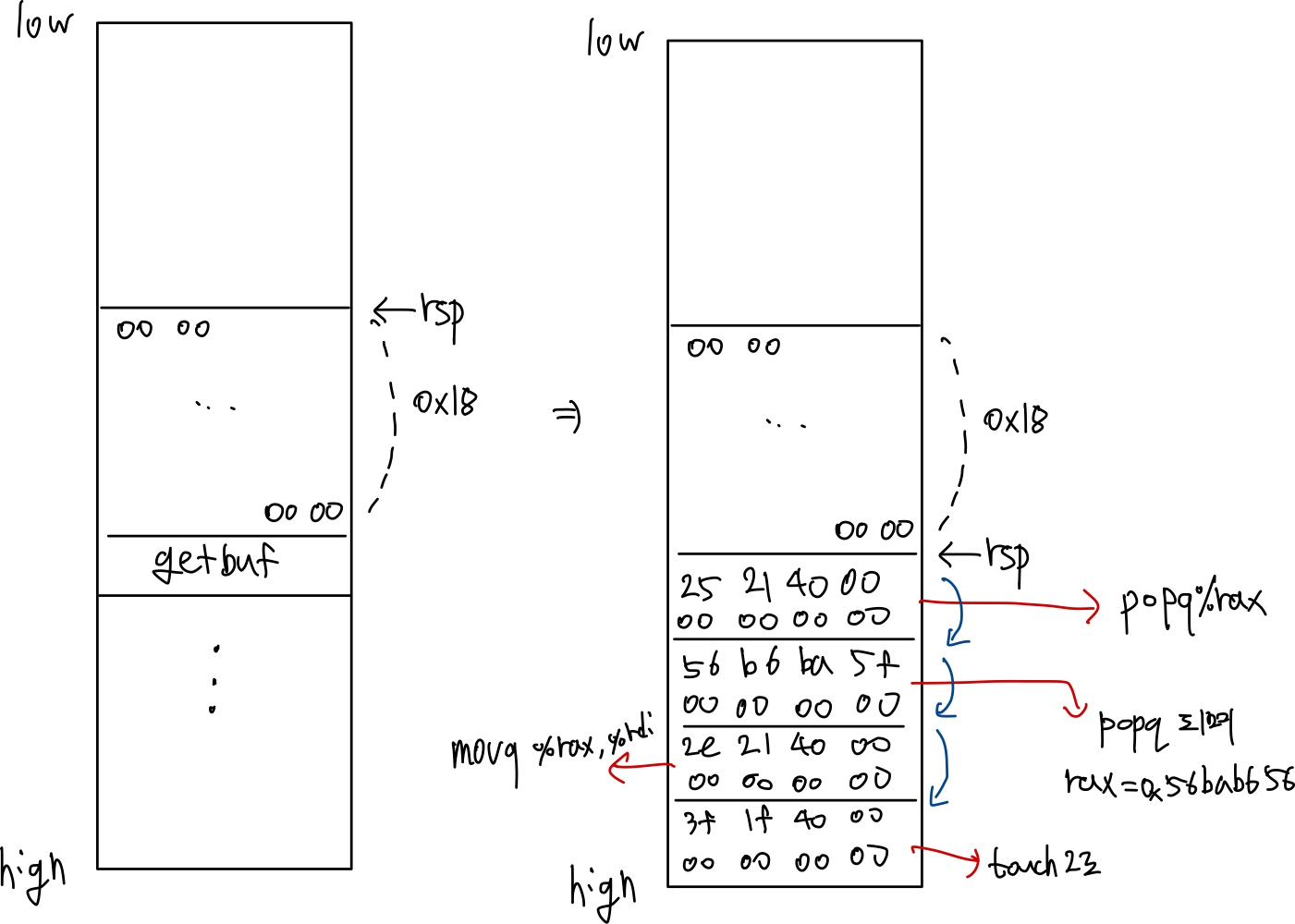


그리고 touch2의 주소는 0x401f3f이다. 따라서 이 정보를 모두 종합해보면, buffer를 모두 dummy로 채운 다음 0x402125, 쿠키 값인 0x5fbab656, 0x40212e, 0x401f3f순으로 넣어 주면 원하는 대로 touch2가 불러질 것으로 생각하였다. 따라서 rtarget\_level2.txt파일에 {00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 25 21 40 00 00 00 00 00 56 b6 ba 5f 00 00 00 00 2e 21 40 00 00 00 00 00 3f 1f 40 00 00 00 00 00 }를 넣어 실행해보았다.





그 결과 PASS가 뜨는 것을 확인할 수 있었다. Rtarget의 level2를 해결하였다.

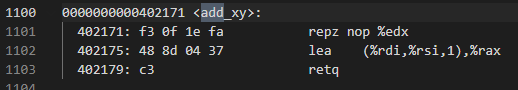


(low가 위, high가 아래로 그렸습니다.)

대략적으로 stack frame을 나타내 보면 위와 같을 것으로 생각하였다.

**<Return-Oriented: Level 3>**

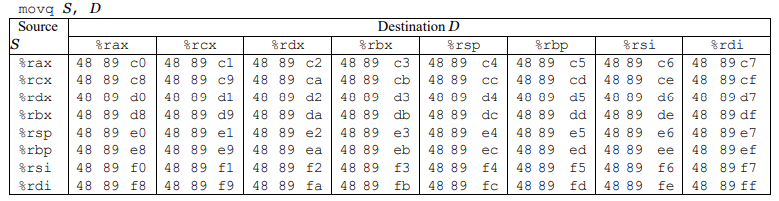
마지막 rtarget의 level3은 ctarget 때와 마찬가지로 문자열의 주소를 rdi에 담아서 touch3으로 전달하도록 구현하면 될 것이다. 그런데, 이를 위해서는 rsp주소에 특정 값을 더하는 기능이 있어야 문자열의 주소를 구할 수 있을 것이다.



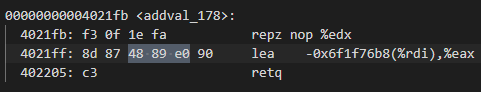
그러한 기능을 할 만한 함수가 있는지 찾아보니 add\_xy함수가 있었고, 이 함수는 rdi가 가리키는 값과 rdi가 가리키는 값을 더해서 rax에 저장하는 기능을 한다. 이 함수를 이용해서 문자열 주소를 rax에 저장할 수 있을 것으로 생각하였다.

Ctarget과 마찬가지로 구현하기 위해서, 먼저 rsp를 rdi에 옮기고 popq를 이용하여 rsi에 특정 값을 넣어준 다음, rdi와 rsi를 갖고 add함수를 호출한다. 그 다음에는 rax에 cookie의 문자열 주소가 저장될 수 있도록 rsi를 조절하여 rax에 cookie문자열을 얻은 다음, 이를 rdi로 옮겨서 touch3의 argument로 들어갈 수 있도록 하는 이러한 과정을 거치면 될 것으로 생각하였다.

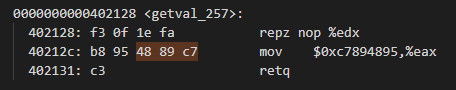
그런데 문제는 mov %rsp, %rdi와 popq %rsi, add\_xy, mov %rax, %rdi, touch3 호출을 바로 진행하는 gadget들이 있다면 좋겠지만 바로 이들을 실행할 수 있는 gadget이 없어서 다른 레지스터로 우회하는 방법들을 찾아야 했다.



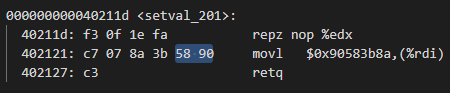
위와 같은 pdf 안의 machine code를 참고하여, rtarget\_disas 파일에서 찾아본 결과 우회하는 루트 하나를 찾아냈다. 그것은 rsp를 rax로 옮기고, rax를 rdi로 옮겨서 결국 rsp를 rdi로 옮기도록 우회하고, popq는 rax로 바로 한 다음, rax(eax)에서 edx, ecx를 거쳐 esi로 값을 넘겨주어 add\_xy를 호출하고, 그 결과인 rax를 rdi로 바로 옮겨 touch3을 호출하는 방법이다. 순서대로 찾은 gadget의 주소들은 다음과 같다.



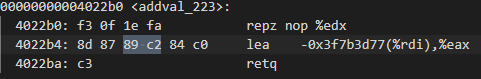
movq %rsp, %rax이고, 90은 무시해도 괜찮다. 주소는 0x402201



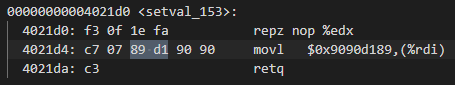
movq %rax, %rdi이고, 주소는 0x40212e



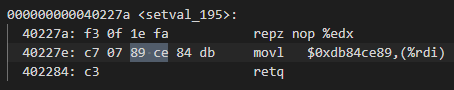
popq %rax이고, 주소는 0x402125이다. Popq 다음에는 offset 값을 넣어주었는데, 실행할 모든 가젯들 다음에 cookie 문자열의 값을 넣어줄 것이기 때문에 0x48값을 offset으로 취하였다. (10진수로 72)



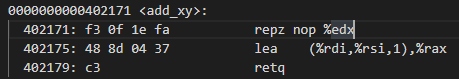
movl %eax, %edx이고, 주소는 0x4022b6



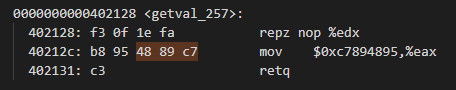
movl %edx, %ecx이고, 주소는 0x4021d6



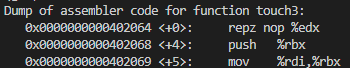
movl %ecx, %esi이고, 주소는 0x402280



add\_xy의 주소는 0x407121

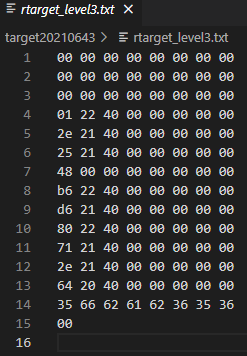


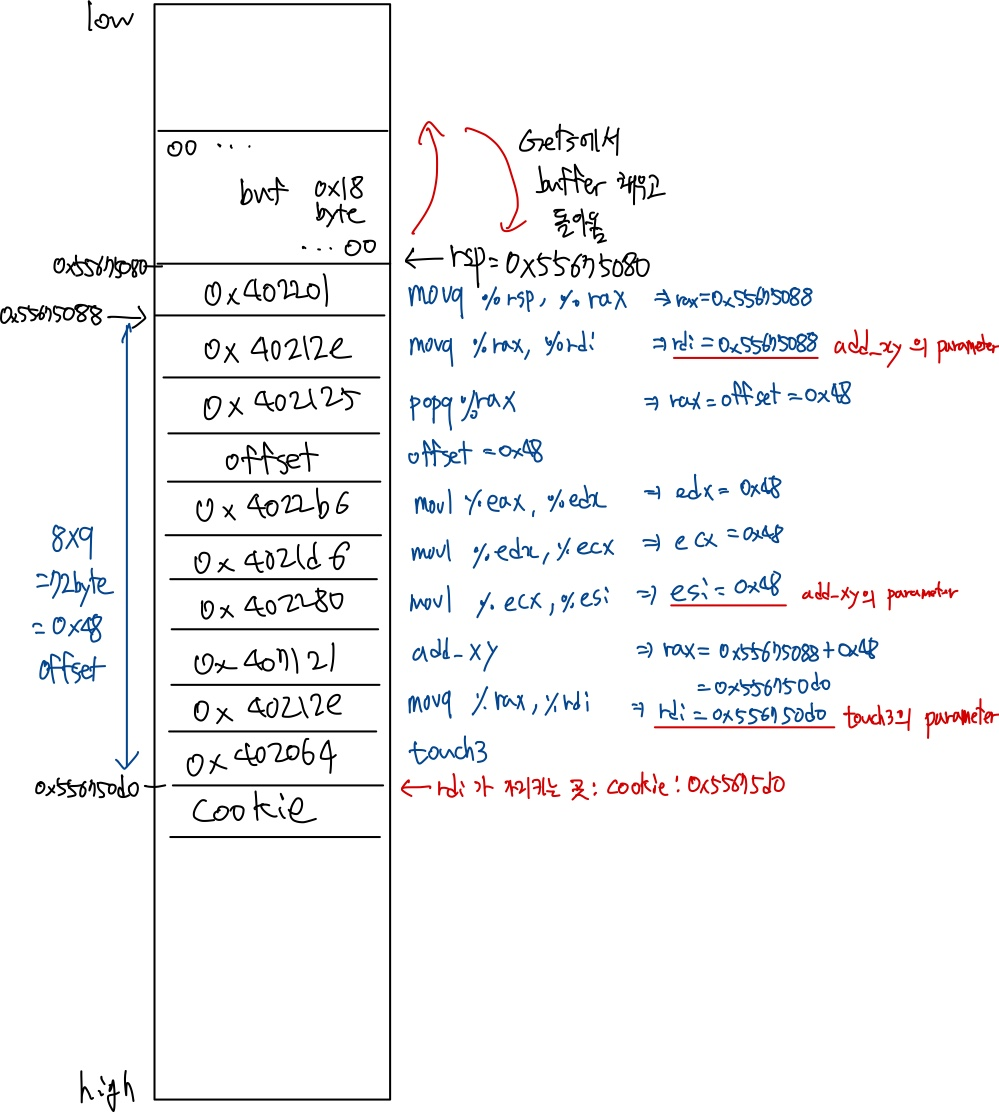
앞에서 한 번 사용한 movq %rax, %rdi이고, 주소는 0x40212e



마지막으로 touch3의 주소는 0x402064이고 이 다음으로는 NULL character를 포함한 cookie의 문자열 내용을 bufferoverflow 시켜준다. Cookie 문자열 값은 {35 66 62 61 62 36 35 36 00 }

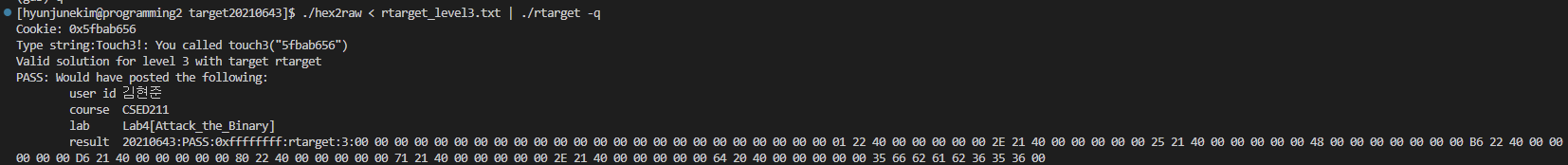
결론적으로, rtarget\_level3.txt 파일에 모든 gadget들의 주소와 offset, cookie를 위에서 쓴 순서대로 넣어주었다. 맨 앞에 버퍼부분은 dummy를 넣어주었다. 그 결과는 {00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 22 40 00 00 00 00 00 2e 21 40 00 00 00 00 00 25 21 40 00 00 00 00 00 48 00 00 00 00 00 00 00 b6 22 40 00 00 00 00 00 d6 21 40 00 00 00 00 00 80 22 40 00 00 00 00 00 71 21 40 00 00 00 00 00 2e 21 40 00 00 00 00 00 64 20 40 00 00 00 00 00 35 66 62 61 62 36 35 36 00 }이다.





(low가 위, high가 아래로 그렸습니다.)

이 때 Stack Frame은 위와 같이 될 것이라고 생각하였다.



실행 결과 PASS가 나왔다. Rtarget의 level 3를 해결하였다. 5개의 phase 모두 해결하였다.