**2022-2 CSED211 Lab04 Report**

학번 : 20210479

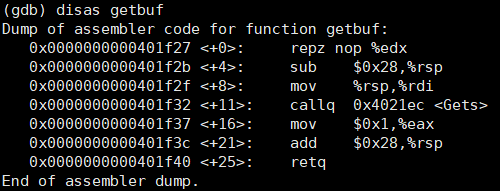
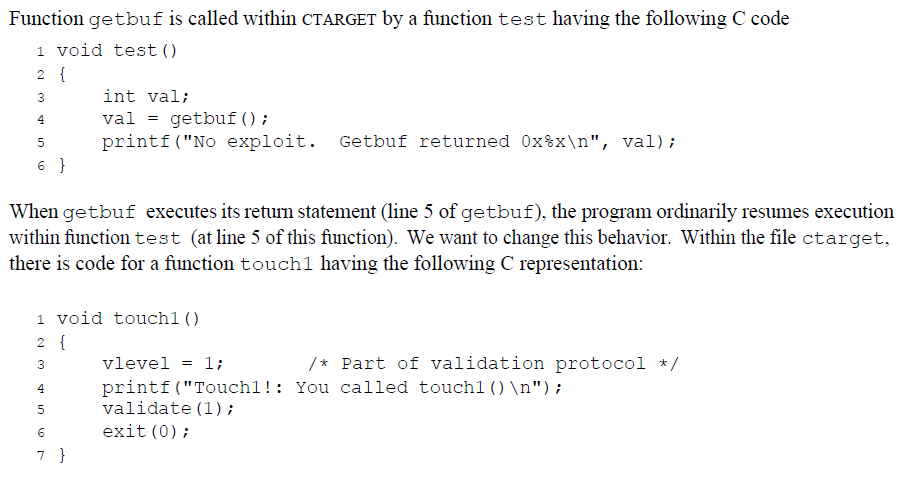
이름 : 이주현

**명예서약 (Honor Code)**

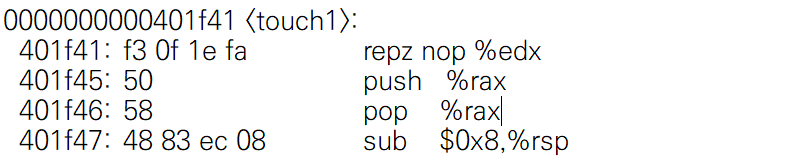
나는 이 프로그래밍 과제를 다른 사람의 부적절한 도움 없이 완수하였습니다.

I completed this programming task without the improper help of others.

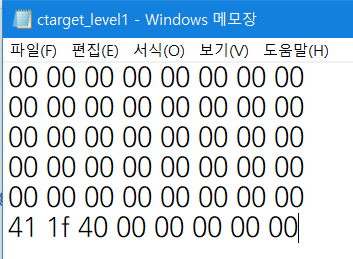
1. **문제별 코드 설명**
   * **Phase\_1**
     + 해결에 앞서 서버에 target20210479.tar를 넣고 압축 해제하여 파일들을 준비했다. ‘README.pdf’를 통해 test와 touch1의 c code를 알게 되었고, test 함수가 getbuf를 부른다는 것을 알게 되었으므로, gdb에서 ‘disas getbuf’를 통해 getbuf의 어셈블리어를 확인해 보았다.



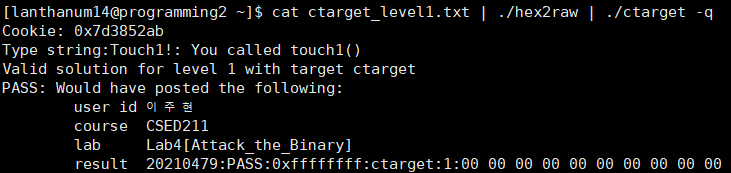
* + - 0x28 = 40(10)이므로, getbuf에서는 stack을 40byte만큼 확보했다는 것을 의미한다. 그리고 그 다음 줄을 보면 바로 %rsp를 %rdi로 옮기고, Gets 함수를 부르는 것을 알 수 있다. Gets는 ‘README.pdf’ 파일을 보면 알 수 있듯, standard library function gets와 비슷한 동작을 하므로, ‘\n’ 또는 end-of-file이 나오기 전까지 string을 읽어오는 함수이다. 즉 다시 말해 40byte보다 작은 string을 입력하면 getbuf의 다음 instruction들이 정상적으로 진행되게 되는 것이다.
    - 이때 우리의 목표는 getbuf에 입력한 string이 touch1의 시작 주소가 되게 하는 것이므로, touch1의 시작 주소를 알아내야 한다는 것을 알 수 있다. 이를 알아내기 위해 xshell 창에 직접 ‘objdump -d ctarget’을 쳤으나, 전체 코드가 뜨지 않고 중간에 잘리는 부분이 있어서 제대로 확인할 수가 없었다. 이에 ‘objdump -d ctarget > ctarget.txt’ 명령어를 사용해 필요한 disassembled version을 txt 파일로 얻었다. 그리고 ‘sz ctarget.txt’ 명령어를 사용해 이것을 다운로드 받아 full disassembled version을 확인할 수 있었다.



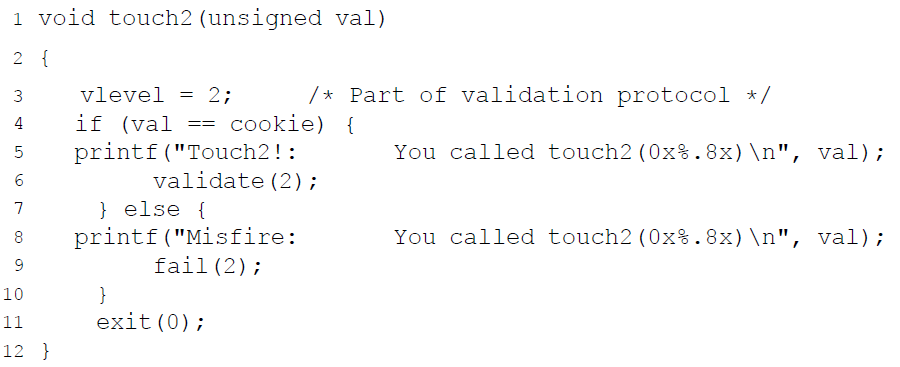
* + - 이를 통해 시작 주소가 401f41임을 알 수 있다. 이때 little endian 방식을 써야 하므로, hex2raw에는 ’41 1f 40’과 같이 넣어줘야 한다는 점을 고려했다. 그리고 현재 총 40byte만큼의 버퍼가 존재하므로, 그것만큼의 dummy를 넣어준 후 touch1의 시작 주소를 넣어 touch1으로 넘어갈 수 있도록 해주었다. (40byte 이내라면 getbuf의 다음 instruction이 진행될 것이므로, overflow가 일어날 수 있도록 dummy 40byte를 넣은 후 touch1의 시작주소를 넣었다.)



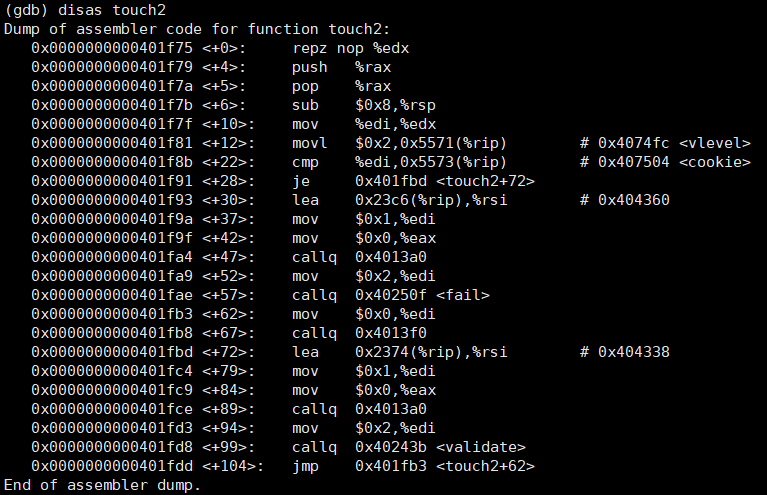
* + - 이것을 xshell에 불러온 후, hex2raw에 넣고 ctarget을 실행한 결과, phase1을 해결했음을 알 수 있었다. 아래는 PASS 문구가 뜬 xshell의 캡쳐 화면이다.



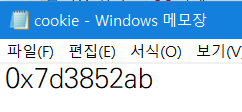
* + **Phase\_2**
    - 문제 해결을 위한 기본 설정(파일 불러오기)은 계속 앞과 동일하다. 그리고 이번에도 ‘README.pdf’를 통해 touch2의 c code를 알 수 있었다. c code를 분석해 보면, val이 어떤 cookie의 값과 동일할 때 정상 작동하는 것을 알 수 있다. 즉, 이번에는 touch2를 호출하는 것뿐만 아니라 val을 적절한 cookie값으로 설정해주어야 하는 것이다. ‘README.pdf’에서 상기시켜준 바를 참고하면, 첫 번째 argument는 %rdi에 저장되므로 이 점을 이용해 injected code를 작성해야 한다는 것을 떠올릴 수 있었다.



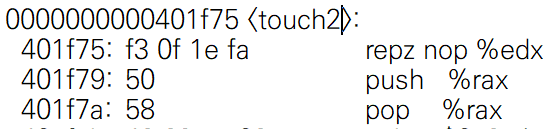
* + - 상술한 값(touch2의 시작 주소 및 적절한 cookie의 값)을 찾기 위해 touch2의 어셈블리 코드를 확인해 보았다.

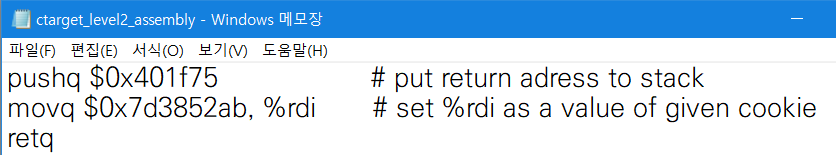


* + - 위의 어셈블리 코드를 해석해 보자면, %edi는 첫 번째 argument, 즉 val일 것임을 알 수 있다. 그리고 <touch2+22>에서 val값과 %rip + 0x5573 간에 cmp 동작이 일어나고 있는데, 이것이 val과 cookie에 대한 비교 동작임을 알 수 있다. 주어진 ‘target20210479.tar’ 파일을 열어 확인한 cookie 값은 아래와 같았다.

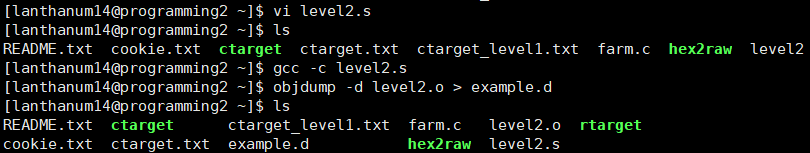


* + - 즉 다시 말해 injected code로 작성할 내용은 %rdi에 해당 cookie 값을 넣고 touch2를 실행하는 것이다. ‘README.pdf’에서 appendix로 따로 설명하는 대로 실행하고자 하는 내용을 담은 어셈블리 코드를 작성했다. 위에서 뽑아내었던 ctarget의 disassembled version을 확인하면, touch2의 시작주소가 0x401f75임을 알 수 있었다. 그리고 cookie는 0x7d3852ab임을 이미 확인하였으므로 이를 넣어 적절한 코드를 작성했다. 작성한 코드는 두 번째 사진과 같다.

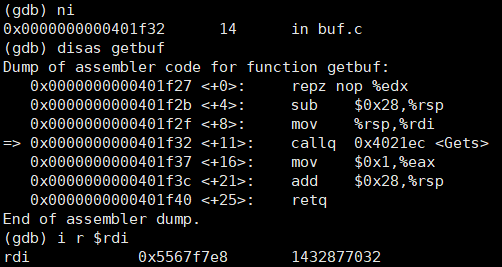




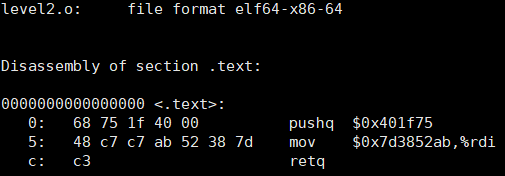
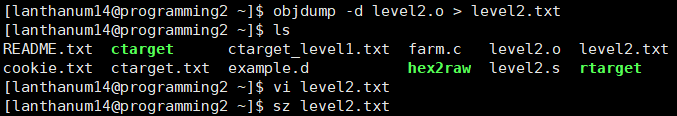
* + - 코드를 러프하게 작성해본 후, ‘vi level2.s’ 명령어를 수행해 이를 .s 파일로 만들어 주었다. 그 후 ‘README.pdf’를 따라 이 파일을 ‘level2.d’ 파일로 바꾸어 주었다.



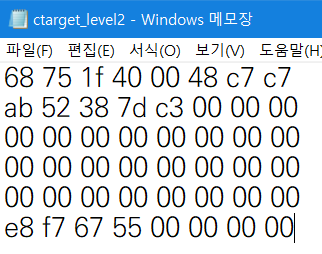
* + - 이제 필요한 코드를 모두 작성해두었으므로 필요한 것은 이 코드를 집어넣을 주소이다. 이때 phase2 또한 getbuf 함수를 이용해 string을 입력받을 것임을 알고 있으므로, getbuf 함수 진행 중 Gets call 직전의 %rdi register의 위치를 알아보았다. 이때 Lab3와는 다르게 ‘run -q’ 명령어를 사용해 주었다. 이를 통해 알아낸 %rdi의 주소는 0x5567f7e8이었다.



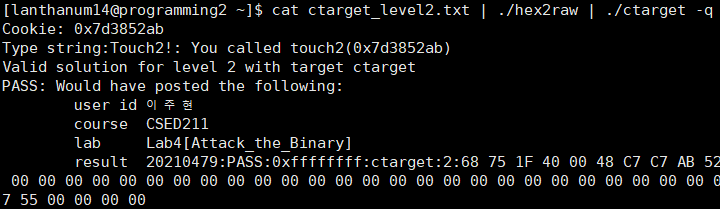
* + - 그리고 이때 ‘level2.s’로 작성한 어셈블리 코드를 기계어로 변환한 것 또한 hex2raw에 넣어줘야 하므로, 아래와 같이 machine code를 읽어내어 주소를 얻어냈다.



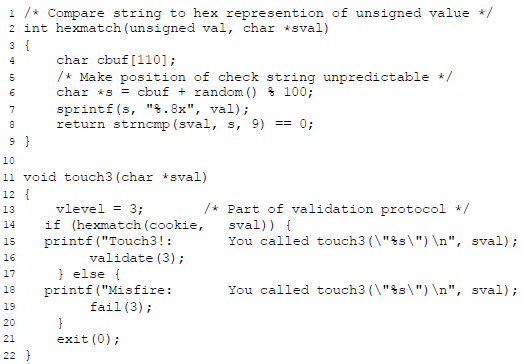
* + - 이를 이용해 phase1과 비슷하게 hex2raw에 넣어줄 파일을 작성해주면 아래와 같다.



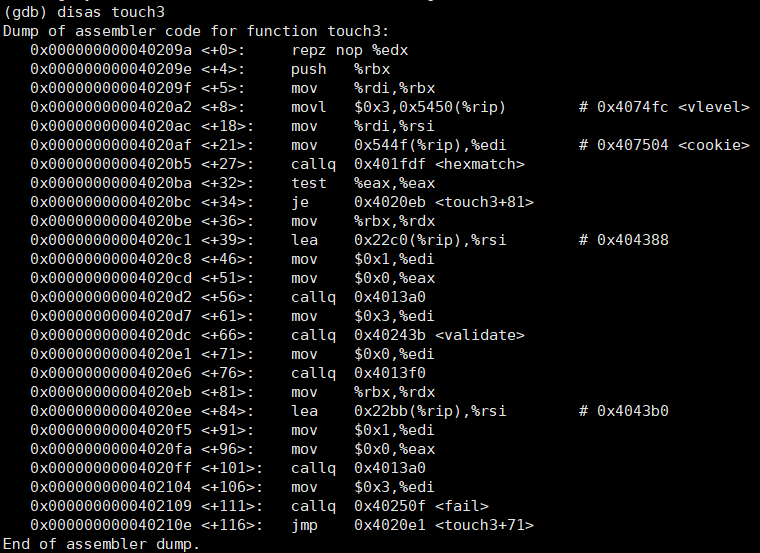
* + - 이것을 xshell에 불러온 후, hex2raw에 넣고 ctarget을 실행한 결과, phase2를 해결했음을 알 수 있었다. 아래는 PASS 문구가 뜬 xshell의 캡쳐 화면이다.



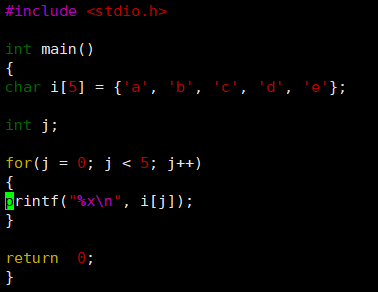
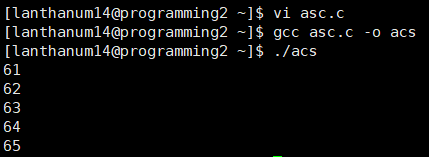
* + **Phase\_3**
    - 이번에는 ‘README.pdf’로부터 touch3과 hexmatch의 c code를 알 수 있었다. 이를 분석해 보면, 적절한 string을 %rdi에 넣어주어야 touch3을 정상적으로 작동시킬 수 있는 것으로 보였다.



* + - 우선 해결을 위해 touch3의 어셈블리 코드를 살펴보았다. 이를 통해 %rdi에 입력한 문자열을 옮긴 후, hexmatch 함수를 호출해 %rdi와 cookie를 비교한다는 것을 알 수 있었다. 즉 phase2를 수행한 것과 비슷한 방법을 거쳐 phase3을 해결할 수 있음을 알 수 있었다.



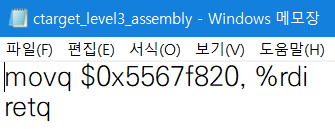
* + - phase2에서도 살펴보았듯, ‘target20210479.tar’ 내에 주어진 cookie의 값은 0x7d3852ab이다. ‘README.pdf’의 appendix A를 참고하여 이를 string 형태로 바꾸어주었다. 이때 ‘man ascii’ 명령어가 동작하지 않아 간단한 c code를 작성해 수동으로 ascii code를 확인했다. 확인한 code와 출력된 값들은 아래와 같다.

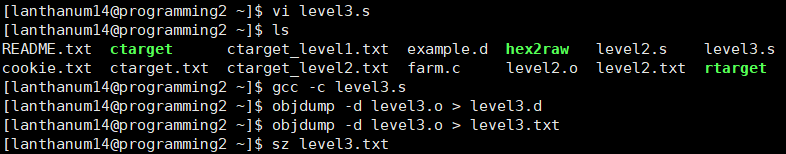
* + - 이렇게 ascii 코드를 확인한 후 알맞게 바꾸어준 string은 ’37 64 33 38 35 32 61 62’이고, string의 맨 뒤에는 ‘00’이 붙는다고 하였으므로 전체 string은 ‘37 64 33 38 35 32 61 62 00’이 된다.
    - 이제 phase3을 해결하기 위해 필요한 것은 cookie가 들어갈 주소와 이 injected code가 들어갈 주소, touch3의 시작 주소, injected code의 machined language이다. 하나씩 구하기 위해 우선 getbuf에서 %rsp의 주소값을 알아보았다. getbuf가 시작되고 40byte를 빼기 전의 %rsp 값을 읽어보면 0x5567f810이다. 함수들의리턴 주소가 2개 저장될 것을 생각하면 %rsp와 %rsp + 0x8에 주소가 저장될 것이므로, 이에 영향을 받지 않고 cookie string을 저장하기 위해서는 %rsp + 0x10이라는 주소가 필요하다. 즉, 문자열을 저장할 주소는 0x5567f820인 것이다.



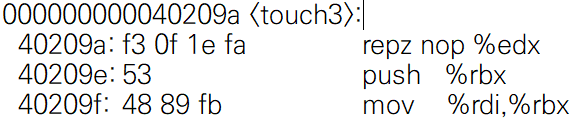
* + - 이를 통해 %rdi 값에 저장할 주소를 알게 되었으므로 다음과 같이 어셈블리 코드를 작성할 수 있었다.



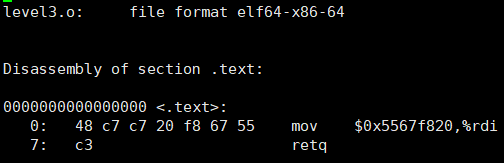
* + - 이전 phase2에서 했듯이, 코드를 러프하게 작성해본 후 ‘vi level3.s’ 명령어를 수행해 이를 .s 파일로 만들어 주었다. 그 후 ‘README.pdf’를 따라 이 파일을 ‘level3.d’ 파일로 바꾸어 주었다. 아래는 해당 동작들을 수행한 다음 화면을 캡쳐한 것이다.



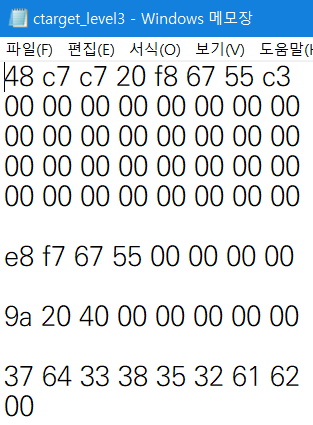
* + - 또한 앞서 구한 ‘ctarget.txt’를 이용해 touch3의 시작 주소가 0x40209a임을 알아냈다.



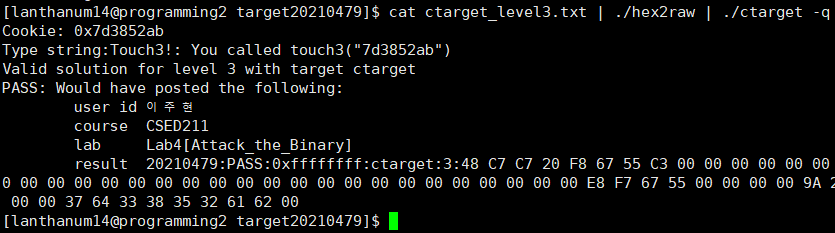
* + - 이 injected code는 이전 phase2와 같이 %rdi에 삽입되어야 하므로, 앞서 phase2에서 getbuf의 %rdi 값을 읽어왔던 것, 즉 0x5567f7e8에 삽입하면 된다.
    - 이제 필요한 것이 모두 준비되었으므로, 마지막으로 작성한 어셈블리 코드의 machine language를 읽어와 준비했다.



* + - 이를 이용해 hex2raw에 넣어줄 파일을 작성해주면 아래와 같다. 위에서 프로그램의 흐름을 통해 알아낸 대로, %rdi에 cookie의 주소를 넣는 것을 buffer에 삽입하고, buffer(40byte)만큼 dummy를 채워준 후, 버퍼의 시작주소(=%rdi, 코드가 삽입될 지점)와 touch3의 시작 주소, cookie 문자열을 순서대로 넣어주었다. 그리고 cookie가 string이라는 점을 감안해 마지막에 ‘00’을 넣어주었다.



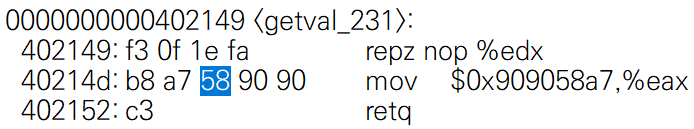
* + - 이것을 xshell에 불러온 후, hex2raw에 넣고 ctarget을 실행한 결과, phase3을 해결했음을 알 수 있었다. 아래는 PASS 문구가 뜬 xshell의 캡쳐 화면이다.



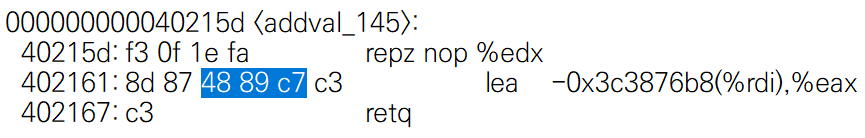
* + **Phase\_4**
    - 이번에는 phase2를 반복하되 rtarget 방식으로 반복해야 했다. 조건은 movq / popq / ret / nop만 사용하고, start\_farm과 mid\_farm에서 필요한 모든 gadget을 찾는 것이었다. 최소 필요 gadget은 2개라는 것까지 ‘README.pdf’를 통해 알 수 있었다.
    - 따라서 gadget들을 확인하기 위해서 ‘objdump -d rtarget > rtarget.txt’ 명령어를 수행해서 disassembled version을 얻었다.



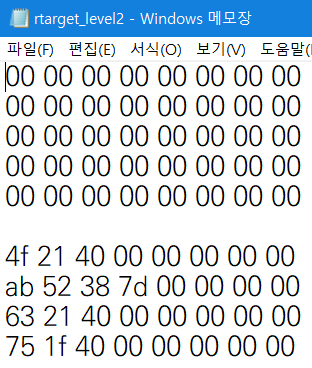
* + - 이제 여기에서 필요한 gadget을 찾기 시작했다. ‘README.pdf’에 있는 gadget 표를 참고해서 필요한 기능을 수행하는 gadget을 찾았다. phase2의 과정을 반복하려면 cookie 값을 %rdi로 옮겨야 한다. 이번 phase에서는 return argument인 %rax를 pop해서 stack에 존재하는 쿠키값을 가져오고, 그것을 %rdi로 mov하는 아이디어를 사용해보았다. pop %rax는 58, mov %rax, %rdi는 48 89 c7이므로 이것이 포함된 gadget을 찾아주었다. 이때 pdf에 적혀 있듯이 0xc3과 0x90도 고려해서 찾도록 했다.
    - 우선 첫번째로 pop %rax와 ret가 연속해서 나오는 gadget을 아래와 같이 찾아주었다. 이것을 통해 stack에 있는 cookie 값을 읽어서 %rax에 저장하는 동작을 수행할 수 있게 될 것이다. 이때 우리가 원하는 동작인 58부터 수행될 수 있도록 40214f를 주소값으로 설정했다.



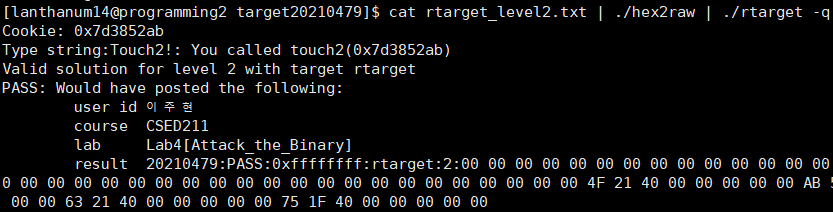
* + - 다음으로 mov %rax, %rdi가 나오는 gadget을 찾아주었다. 이것을 통해 touch2의 argument인 %rdi에 저장하는 동작을 수행할 수 있게 된다. 이것은 처음부터 우리가 원하는 동작인 48로 시작하므로 402163이라는 주소값을 그대로 써도 된다.



* + - 따라서 이 내용을 담아서 hex2raw에 넣어줄 파일을 작성해주면 아래와 같다. 맨 마지막 줄에는 부르고자 하는 함수인 touch2의 주소값을 넣어주었다.

****

* + - 이것을 xshell에 불러온 후, hex2raw에 넣고 ctarget을 실행한 결과, phase4를 해결했음을 알 수 있었다. 아래는 PASS 문구가 뜬 xshell의 캡쳐 화면이다.



* + **Phase\_5**
    - 이번에는 phase3를 반복하되 rtarget 방식으로 해결해야 했다. 즉 이번에도 목적은 %rdi로 cookie 값을 넘기는 것임을 알 수 있었다. 그리고 ‘README.pdf’를 통해 start\_farm부터 end\_farm 내에서 필요한 모든 gadget을 찾아야 하며, phase4에서 허용되었던 동작과 movl을 사용할 수 있다는 것을 알 수 있었다. 또한 8개의 gadget이 있어야 한다는 것을 알았다.
    - 즉 원하는 동작을 수행하기 위해서 gadget들을 알맞게 배열하는 과정이 필요하다는 것이다. 우선 start\_farm부터 end\_farm까지 중에서 사용할 수 있을 만한 것들을 골라내 보면 다음과 같다. (중복되는 동작을 하는 것은 적지 않았다.)

0x40214f: 58 90 90 c3 popq %rax / nop / nop / retq

0x402159: 89 c7 90 c3 movl %eax, %edi / nop / retq

0x40216e: 48 89 c7 c3 movq %rax, %rdi / retq

0x4021e1: 89 ca 84 c9 c3 movl %ecx, %edx / testb %cl, %cl (**nop**) / retq

0x402242: 89 c1 90 c3 movl %eax, %ecx / nop / retq

0x40222f: 89 d6 c3 movl %edx, %esi / retq

0x40225a: 89 e0 c3 movl %esp, %eax / retq

0x402263: 48 89 e0 c3 movq %rsp, %rax / retq

* + - 이것을 가지고 해야 하는 동작은 %rsp를 %rdi에 옮기고, 적절한 offset값을 %rsi에 넣어 그것을 다시 %rdi에 더해 %rdi를 문자열이 있는 곳으로 보내준 다음, 그 더해진 계산값을 다시 %rdi에 넣어주는 것이다. 즉 아래와 같은 방법을 통해 동작을 수행할 수 있다.
    - 중간에 %rdi를 문자열이 있는 곳으로 보내기 위해 offset 48(10)을 더해주는데, 이는 buffer가 40byte이고 cookie는 그보다 0x8 = 8byte 위에 저장되어 있기 때문이다. 그러나 add 동작을 쓸 수 없기 때문에 mov와 pop을 적절히 이용해 %rdi와 offset의 값을 %rsi, %rdi로 보내주어야 한다.
    - 그런 후 %rsi와 %rdi가 함수의 argument임을 이용해 add\_xy 함수를 불러 둘을 더해준다. 이 동작 때문에 이 값이 retrun argument인 %rax에 저장되므로 이것을 다시 %rdi로 보내주는 작업을 한번 더 거쳐야 한다. (현재 %rax에 들어있는 값이 최종값이므로) 그리고 맨 마지막에는 touch3 함수의 시작 주소와 cookie string을 넣어주면 된다..
    - movq %rsp, %rax: 63 22 40 00 00 00 00 00

movq %rax, %rdi: 6e 21 40 00 00 00 00 00

popq %rax: 4f 21 40 00 00 00 00 00

48 00 00 00 00 00 00 00 //offset

movl %eax, %ecx: 42 22 40 00 00 00 00 00

movl %ecx, %edx: e1 21 40 00 00 00 00 00

movl %edx, %esi: 2f 22 40 00 00 00 00 00

a8 21 40 00 00 00 00 00 //add\_xy

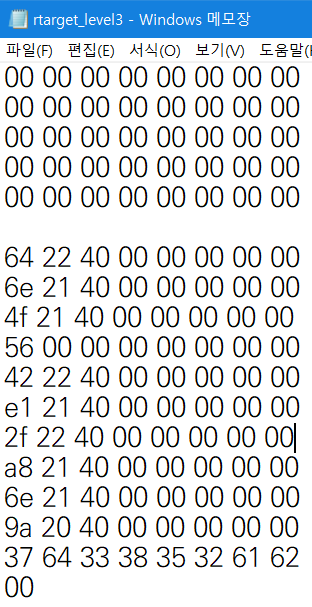
movq %rax, %rdi: 6e 21 40 00 00 00 00 00

9a 20 40 00 00 00 00 00 //touch3

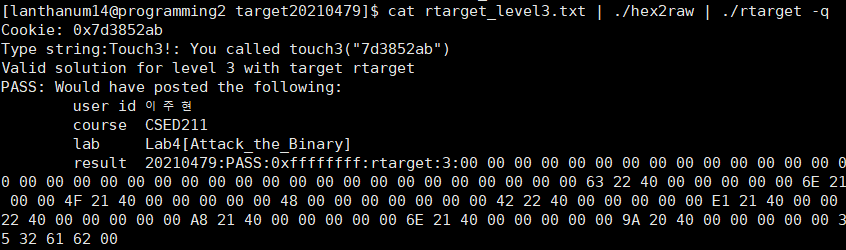
37 64 33 38 35 32 61 62 //cookie string

00

* + - 따라서 이 내용을 담아서 hex2raw에 넣어줄 파일을 작성해주면 아래와 같다. 위의 dummy 40byte는 buffer에 채워주기 위해서 넣은 것이다.



* + - 이것을 xshell에 불러온 후, hex2raw에 넣고 ctarget을 실행한 결과, phase5를 해결했음을 알 수 있었다. 아래는 PASS 문구가 뜬 xshell의 캡쳐 화면이다.



1. **Lab04에서 배운 점**

* Lab04를 해결하면서 code-injection 및 return-oriented-programming 공격에 대해 잘 이해할 수 있었다. 그리고 지난번 Lab03에 이어서 gdb를 이용해 debugging하는 법을 익힐 수 있었고, 이번에는 objdump와 hex2raw를 이용하는 방법도 알게 되었다.