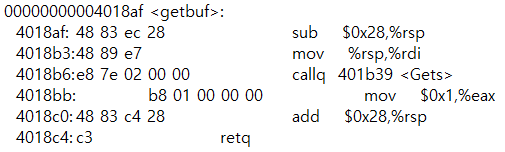
**Attack Lab (시스템 소프트웨어)**

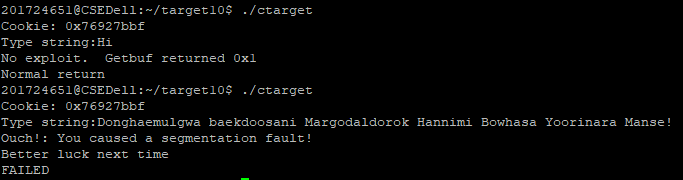
**(Phase 1)**

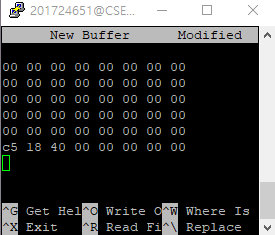
Attack lab은 버퍼 오버플로우를 이용하여 프로그램의 프로세서를 조작하는 방법을 실습해보는 것이다. 우선 target 파일을 다운받고 WinSCP로 해당 파일을 서버에 업로드하고, 압축 해제를 하고,

objdump –d ctarget > ans.txt 명령을 사용하면 disass결과를 txt 파일로 저장되어 메모장에서 확인할 수 있다. 해당 실습에서 target 프로그램은 getbuf를 통해 std input으로 string을 읽어들인다. Gets 함수를 사용하므로 \n 혹은 16진수 0x0a가 입력되면 입력을 멈춘다.



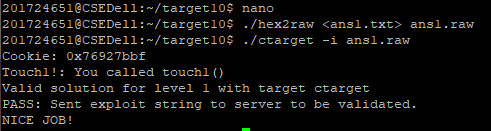
%rsp를 0x28(10진수로 40)만큼 감소시켰다는 것은 작업 스택을 0x28byte만큼 확보했다는 것이고, 다음 줄에서는 %rdi에 %rsp를 옮기고 Gets 함수를 호출한다. Gets는 char 포인터 타입의 인자를 받아 std input을 시행하는 함수이기 때문에 스택 상에서 %rsp에 입력받는 string을 차례대로 저장한다. 그래서 0X28, 즉 40보다 작은 string을 입력하면 프로세스가 정상적으로 넘어간다. 그러나 이 buffer보다 긴 string을 입력하면 Segmentation fault가 발생하고 FAILED과 함께 종료된다.





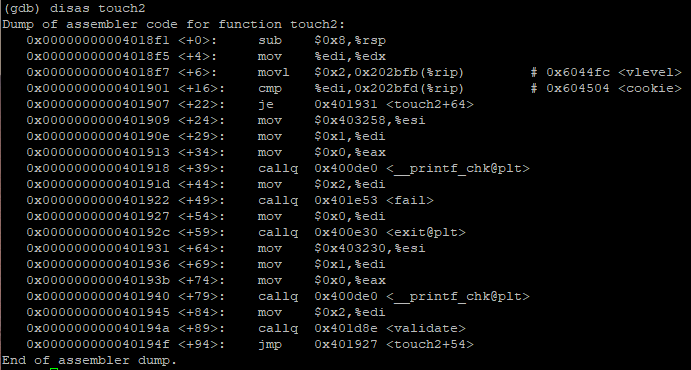
touch1은 보다시피 불러내기만 하면 함수이다. Byte + return address를 덮어씌울 새로운 touch1의 address를 넣으면 되는데, 스택으로 저장되므로 24byte는 아무거나 넣고 4018c5를 c5 18 40식으로 차례대로 nano 명령어를 치고 들어가 입력하여 txt파일로 저장해주면 된다.

그런 후 ./hex2raw <ans1.txt> ans1.raw 명령어로 ans1.raw를 만들어주고 ./ctarget –i ans1.raw.를 입력하면 touch1를 불러내면서 성공한다

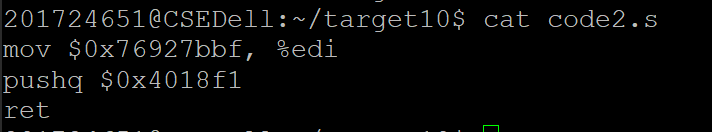


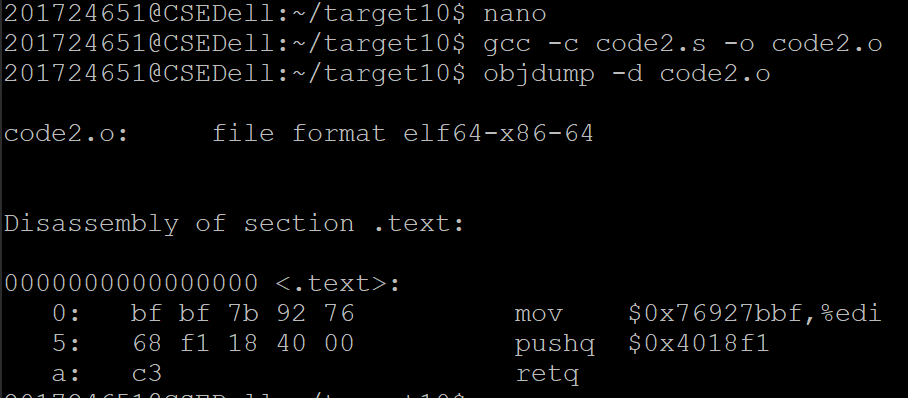
**(Phase 2)**

touch2는 %rdi를 이용하여 하나의 인자를 받는 함수이다.

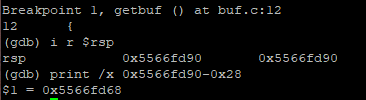
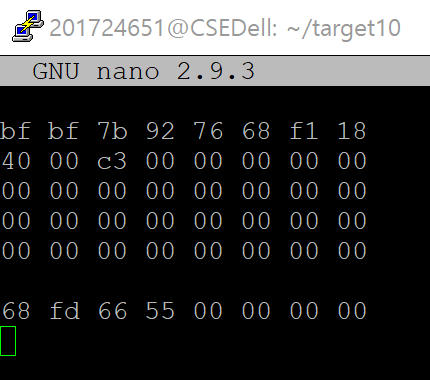


그림과 같이 cookie와 인자가 같아야한다. Cookie는 위에서 언급한 16진수이다. 이때 return address뿐만이 아니라 edi(rdi)에 들어가야할 값까지 수정해야 한다. 그러기 위해 machine code를 작성해주어야 한다. 그래서 %rdi에 쿠키값인 0x76927bbf를 넣도록 mov $0x76927bbf, %edi와 return 주소인 touch2의 주소를 넣어주도록 push1 $0x4018f1, return을 넣어준다. 그리고 이것을 disass해주면 바이트 표현이 나오는데, 이때 이 코드의 리턴 주소는 buffer의 주소여야 한다.

****

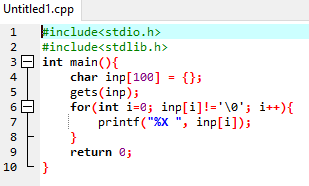


getbuf에 breakpoint를 걸고 %rsp 레지스터 정보를 i r $rsp로 확인해보면 getbuf가 시작할 때 레지스터 정보를 확인할 수 있고, 여기서 buffer 크기를 빼주면 buffer의 주소가 된다. 또 getbuf의 리턴 주소는 touch1때와 동일하게 touch2의 주소여야 한다.





이러면 touch2를 불러오며 성공.

**(Phase 3)**

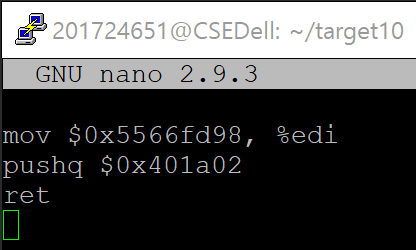
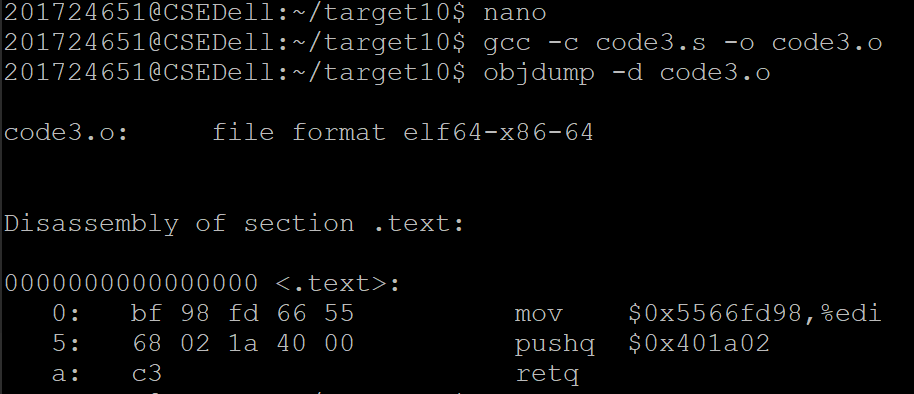
touch3도 touch2와 크게 다른 부분은 없었다.

Touch3함수는 char 포인터 즉 문자열을 입력으로 받아 hexmatch 함수를 호출해서 문자열이 쿠키와 같은지 비교하나. Touch2의 숫자를 비교하는 것이었다면 touch3는 문자열의 비교인 것이다.

나의 쿠키인 0x76927bbf를 아스키 코드를 참조하여 표현하면 37 36 39 32 37 62 62 66 이 된다.

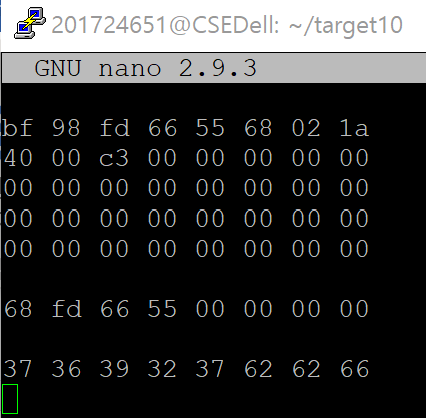
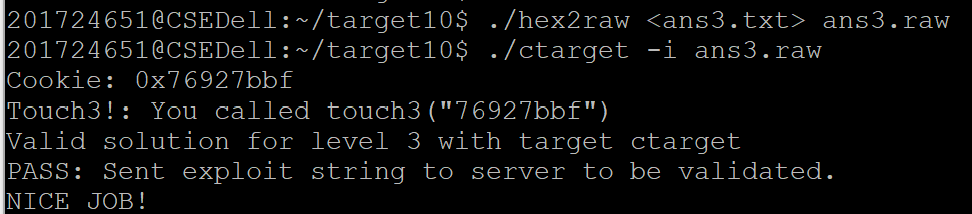
(아스키 코드 계산은 c코드를 이용하였다)

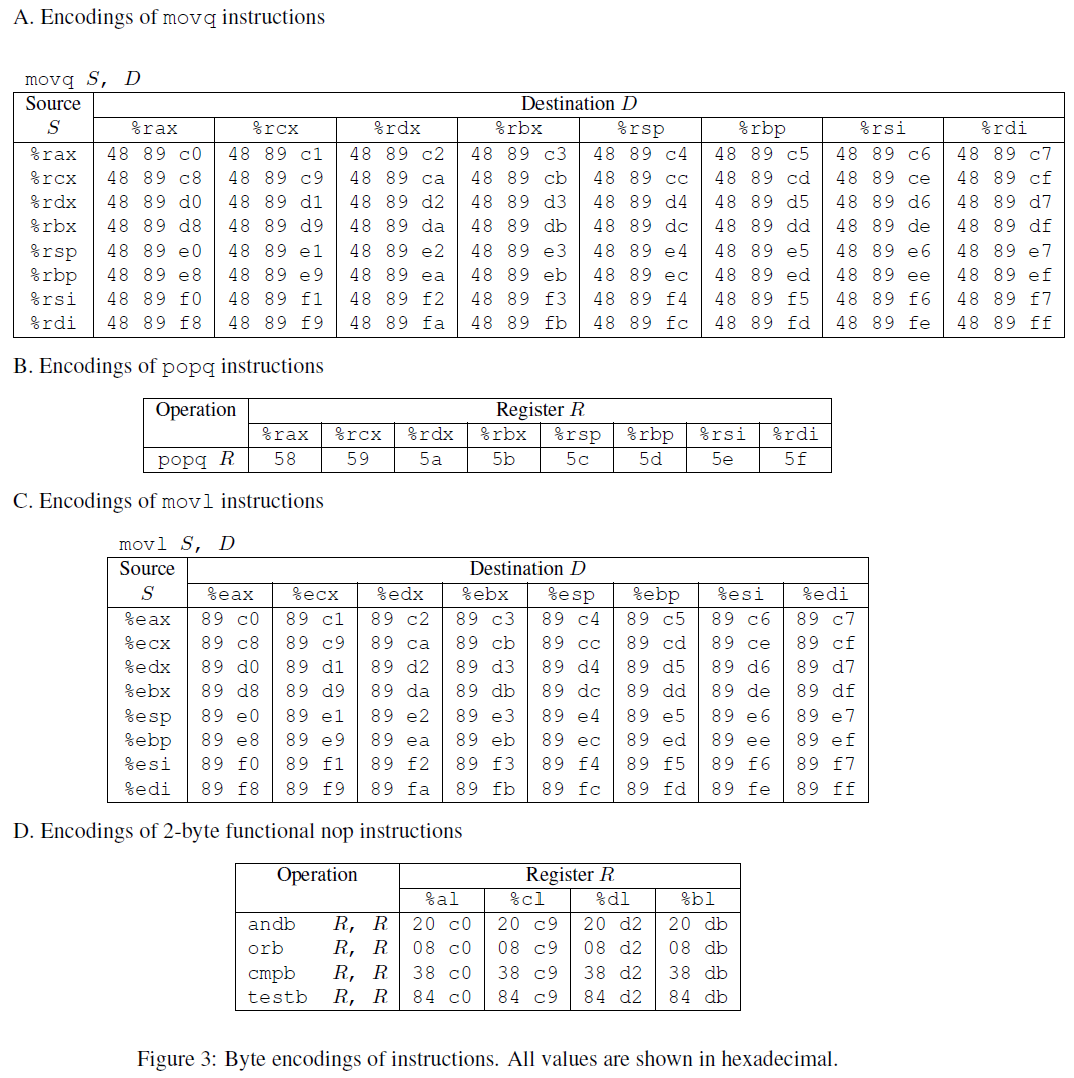
Touch3의 주소를 push 명령어로 넣고, touch3 주소 자리에 이제 쿠키 문자열이 위치하니 쿠키 문자열의 위치는 0x5566fd90 + 0x8 = 0x5566fd98 이다. 그러므로 코드를 작성해보면 아래의 오른쪽 그림과 같이 표현이 가능하고, 그렇게 작성된 code3.s를 code.o로 어셈블하면 아래와 같다.



위의 설명한 사항들을 순서대로 나열하면 code3.o + 패딩(0x28) / 코드 주소 / 쿠키 문자열이다.

Ans3.txt은 다음과 같고 ./hexraw2 <ans3.txt> ans3.raw 명령어와 ./ctarget -I ans3.raw 명령어를 입력하면 정답임을 확인할 수 있다.

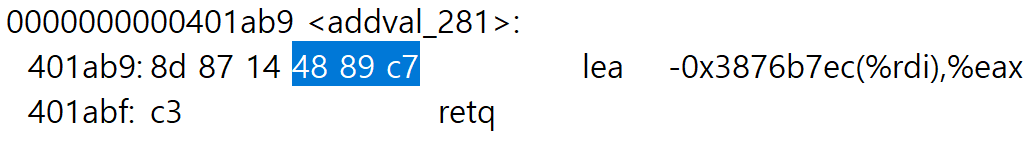


**(Phase 4)**

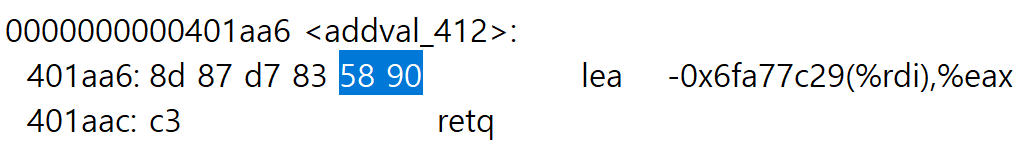
이번 단계부터는 ctarget이 아닌 rtarget을 이용해야 한다. 다른 점이라고 하면 ctarget에서는 해당 주소를 특정해줄 수 있었지만 rtarget에서는 그것이 불가능하다. 즉, rtarget은 실행시킬 때마다 스택의 주소가 변하기 때문에 스택의 주소를 특정할 수가 없다는 것이 ctarget과의 차이점이다.

우선, objdump –d rtarget > rans.txt 명령어를 통해 disass 결과를 txt파일로 저장해 두어 안의 내용들을 확인해보자. 압축 파일 내에 farm.c 파일이 있음을 확인할 수 있겠지만 rans.txt의 내용을 확인하다 보면 <start\_farm>과 <end\_farm> 부분을 볼 수 있는데 그 사이 부분들을 보면 여러가지 함수들이 있는데 거기서 아래의 사진과 같은 내용들을 볼 수 있는데 여기서 파란 블록 부분은 오른쪽 사진, 즉 가젯 목록에서 일치한 부분을 rans.txt 파일 내에서 찾아 그 주소를 확인할 수 있다.

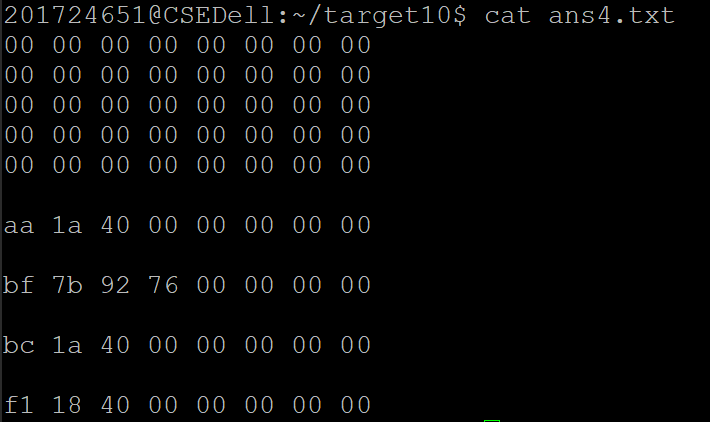
그러면 이제 이 주소들을 이용하여 touch2에서 했던 것과 동일하게 하면 된다.



* ***48 89 c7 c3(mov %rax, %rdi 후 retq) -> 401abc (address)***



* ***58 90 c3 (pop %rax 후 retq) -> 401aaa***



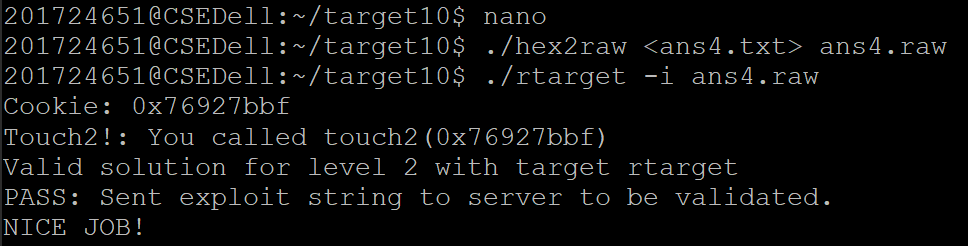
순서는 다음과 같다. 버퍼 크기 만큼의 패딩

-> popq %rax, ret이 있는 주소

-> 쿠키값

-> movq %rax, %rdi가 있는 주소

-> touch2 함수가 있는 주소

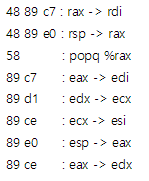


이러면 Phase\_4는 해결이다.

**(Phase 5)**

이번에도 phase\_4와 비슷하게 touch3를 해결하면 된다. 다른 점이라고 하면 이번에는 farm 안에 add\_xy라고 하는 함수를 사용해야한다.



add\_xy 함수를 보면 %rdi와 %rsi의 값을 더해서 %rax에 저장하는 것을 확인할 수 있다.

우선 나는 farm 안에 있는 내가 가능한 모든 가능한 가젯을 골라보았다.

farm 안에서 가젯과 매칭한 경우의 수는 오른쪽의 사진과 같았다.

그래서 이 것들로 위의 add\_xy에 맞게 조합을 해보았다.

결과는 아래의 사진과 같았다.

우선 buffer 사이즈에 대한 padding을 주고,

%rsp -> %rax -> %rdi와

%eax -> %ecx -> %edx -> %esi

처럼 각각의 레지스터를 통해 원하는 값을 전달했다.

offset은 rax를 불러오는 부분 바로 밑에서부터 cookie의 거리를 말한다. 그러므로 9칸인데 9\*8byte = 72

를 16진수로 표현하면 0x48이 된다.

