**Lab4 report**

20210774 김주은

<Lab4 강의 내용 요약>

1) Calling convention

- 함수를 부를 때에는 function에 인자를 넘겨줘야 한다. 그리고 인자는 정해진 순서대로 각 register에 저장되게 되는데, 그 순서는 rdi, rsi, rdx, rcx, r8, r9와 같다. 그리고 나머지 인자들은 stack에 저장된다.

- 리턴값은 rax에 저장된다.

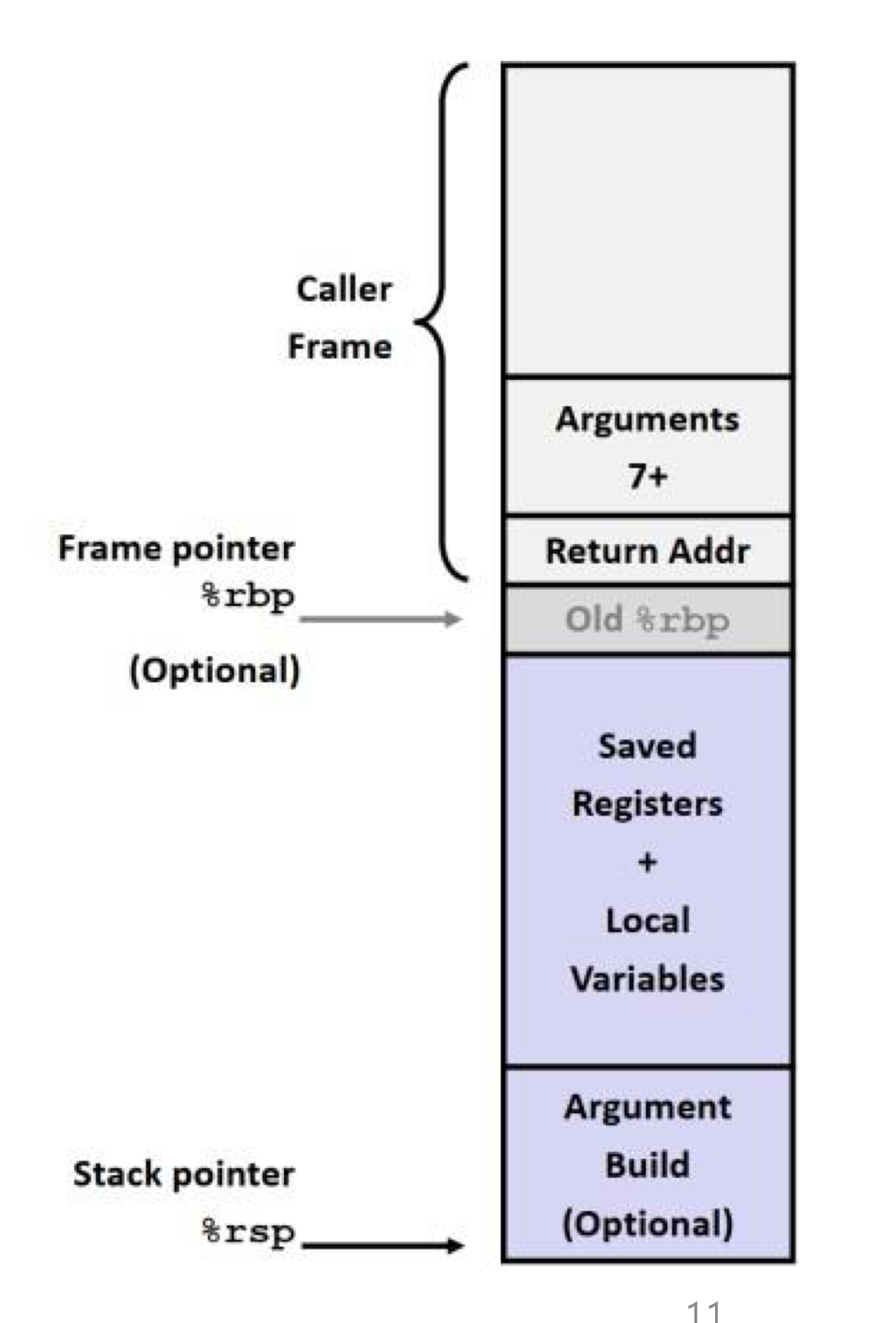
2) program memory layout

- 함수는 stack에 각자 자신만의 frame을 가진다.

- stack은 LIFO 구조

- 지역변수도 stack에 저장된다.

- stack의 구조



- 함수가 처음 시작할 때의 프롤로그가 존재 : push %rbp mov %rsp, %rbp인데, 이는 위의 구조에서 old %rbp값을 stack에 넣고 mov %rsp, %rbp를 통해 %rbp값을 현재 %rsp값으로 업데이트하는 것을 알 수 있다.

<Lab5 강의 내용 요약>

1) array의 경우 T A[L];의 형태로 선언되는데, A[i]의 주소는 A + i\*k 이다.

2) structure : 원소들이 순서대로 align된 구조

3) union : 같은 메모리상에서 같은 공간을 차지하고 다른 해석이 가능한 구조

4) buffer overflow : buf 입력값을 받다가 이 값이 일정 크기를 넘어서면서 return address를 수정할 경우 입력값에 따라 다른 코드를 실행할 수 있음.

<Lab4 풀이 내용>

**(1) ctarget\_level1**

먼저 getbuf 함수는 다음과 같다.

텍스트이(가) 표시된 사진

자동 생성된 설명

그리고 우리가 getbuf를 종료하면서 호출해야 하는 함수는 touch1 함수다.

텍스트이(가) 표시된 사진

자동 생성된 설명

먼저 getbuf를 분석해보면,

텍스트이(가) 표시된 사진

자동 생성된 설명

%rsp에서 0x38 즉, 56만큼 빼는 과정을 통해 총 56만큼의 stack 공간을 할당하는 것을 알 수 있다.

이후 %rsp값을 %rdi에 넣고 gets함수를 호출하는 것으로 보아, %rsp에서부터 buf값을 입력 받는 것을 알 수 있다.

Getbuf 함수가 끝나면서 기존의 return adress로 돌아가지 않고 touch1의 시작주소로 가야하므로, 입력값을 통해 return address를 변경해줘야 한다는 것을 알 수 있었다.

그래서 return address 전까지의 총 56만큼의 byte를 0으로 채워주고 return address 공간에 touch1의 시작주소를 넣어주었다.

Touch1의 시작주소는 disas touch1을 통해 알 수 있었다.

텍스트이(가) 표시된 사진

자동 생성된 설명

시작주소는 401e09임을 알 수 있다.

Answer :



**(2) ctarget\_level2**

Level2에서 getbuf에서 불러야 할 함수는 touch2함수이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

코드를 살펴보면, val == cookie 이어야 touch2!가 출력됨을 알 수 있으므로, Touch2를 호출함과 동시에 인자 값인 val 값이 cookie 값과 똑같아야 한다.

그래서 이번에는 touch2함수를 호출해야 함과 동시에 cookie값과 동일한 입력값을 rdi 레지스터에 넣어 주는 것이 필요하다.

먼저, cookie값을 확인해보면

텍스트이(가) 표시된 사진

자동 생성된 설명

0x192556bb임을 알 수 있다.

그래서 이를 %rdi에 넣는 과정이 필요하기에 이를 assembly 언어로 적어준다.

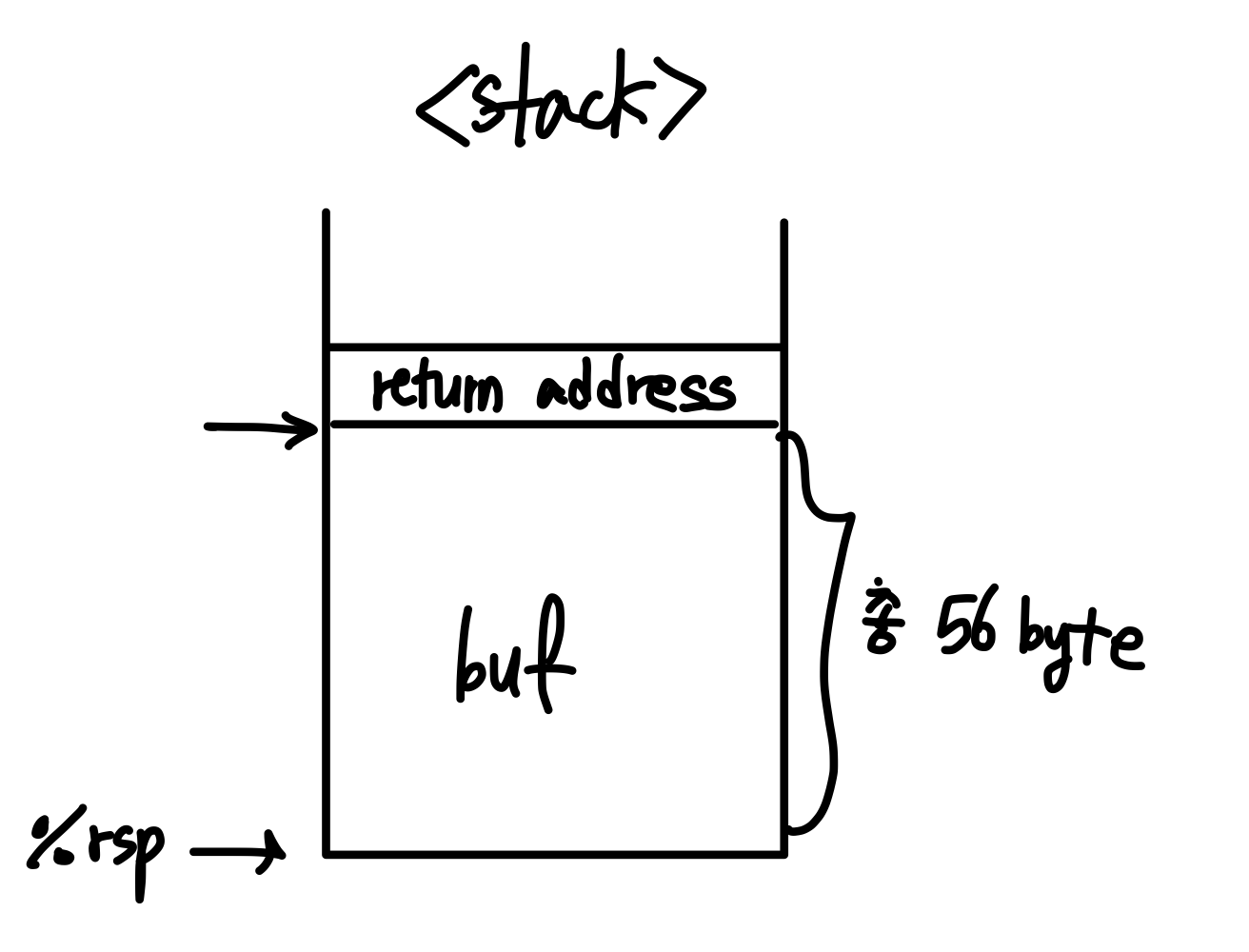
movq $0x192556bb, %rdi

그리고 이를 수행한 후 touch2로 넘어 가야 하므로, touch2의 시작주소를 찾아보면,

텍스트이(가) 표시된 사진

자동 생성된 설명

0x401e3d임을 알 수 있다.



이러한 stack의 구조가 있다고 볼 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

우리가 의도한 바를 진행시키려면 명령어를 buf 입력값에 넣고, return address를 명령어들이 담겨있는 곳의 주소, 즉 현재 %rsp 값으로 설정해야 한다. 그렇게 해야 return address구간에서 명령어가 있는 곳으로 가서 명령어를 수행할 수 있다. 그래서 명령어가 있는 주소를 알기 위해서 breakpoint를 걸어놓고 getbuf를 실행한 후 %rsp를 0x38만큼 뺀 곳의 주소를 %rdi에 있는 값을 통해 알아낸다.

텍스트이(가) 표시된 사진

자동 생성된 설명

알아낸 결과로는 %rsp값이 0x5562b5b8임을 알 수 있다.

이제 수행하는 명령어를 마저 작성하면 먼저 stack의 top에 touch2의 시작주소를 넣어서 return address로 설정하여 retq를 할 시 touch2가 실행하도록 한다. 그리고 rdi 값을 설정해주면 된다. 따라서 코드는 아래와 같다.

텍스트이(가) 표시된 사진

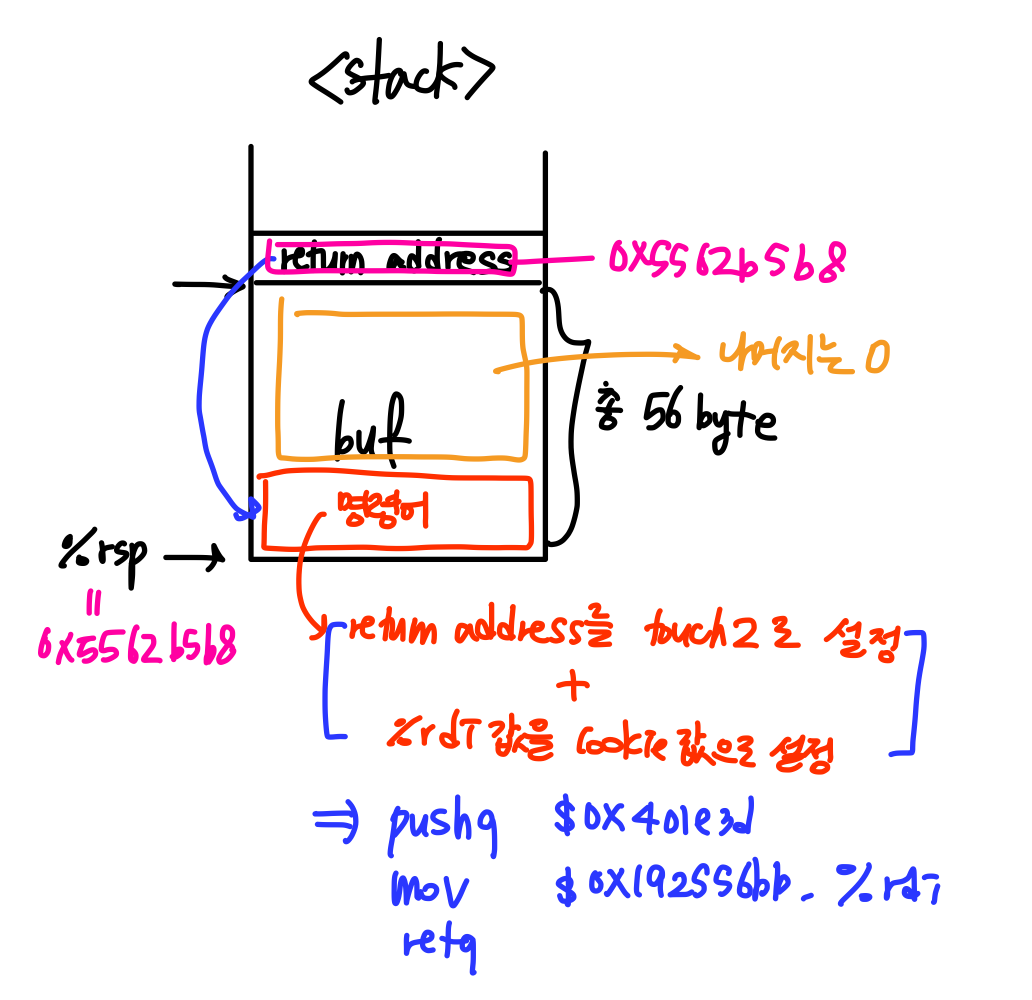
자동 생성된 설명

이를 그대로 넣으면 안되고 기계어로 바꾼다음 16진수의 형태로 넣어야 하므로, 컴파일을 하여 대응되는 16진수의 형태로 나온 값을 확인한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

즉, 위의 68부터 c3까지의 숫자들이 명령어에 해당하는 것이다. 결국 이렇게 구한 명령어와 %rsp 값으로 txt파일을 만든다.



이렇게 수정된 파일을 통해 프로그램을 실행하면 level2를 pass 할 수 있었다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Answer:

텍스트, 전자기기이(가) 표시된 사진

자동 생성된 설명

**(2) ctarget\_level3**

우리가 실행해야 하는 함수는 touch3이다.

텍스트이(가) 표시된 사진

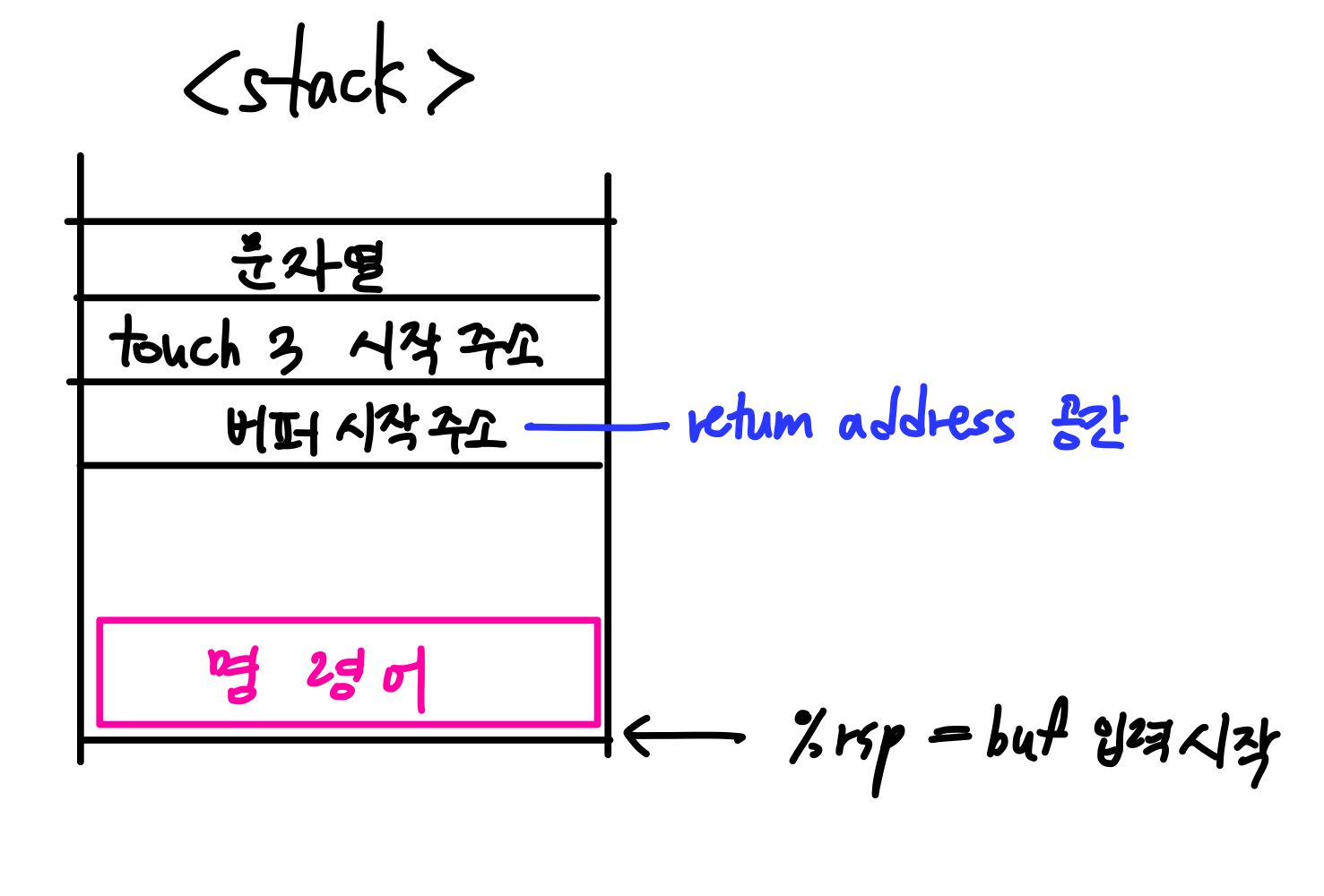
자동 생성된 설명

Cookie와 sval을 hexmatch 함수를 호출하여 비교하는데, sval 주소에 담긴 문자열과 cookie 값이 일치해야 touch3!가 출력되면서 pass 되는 것이다. 즉, sval이라는 주소에 cookie값과 동일한 문자열을 넣고 touch3를 호출해야 한다.

아까 level2의 방법과 비슷하게 명령어를 넣어줘야 한다. Touch3를 호출할 때 인자값을 설정해줘야 하기 때문에 이를 명령어를 통해 넣어주어야 한다. 명령어로는 문자열의 시작주소를 rdi에 넣어주는 것이 필요하다.

즉, movq (문자열의 시작주소), %rdi 이 되어야 한다.

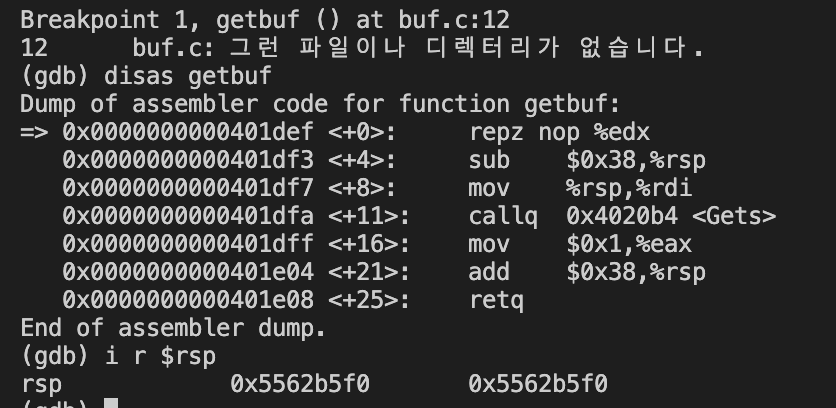
이제 문자열을 어디에 넣어야하 는지가 문제인데, return address에는 buf 입력값의 시작주소, touch3의 시작주소 등이 차곡차곡 쌓여서 들어가야 한다. Buf 입력값의 처음 주소로 들어가 명령어를 통해 rdi값을 설정하고, Touch3가 호출되어야 하기 때문이다. Return address 부분이 pop되면서 사라지므로, 결국 문자열은 touch3의 시작주소 보다 더 위의 stack 부분에 들어가야 한다.



문자열의 경우는 cookie값을 넣어야 하는데, 이의 각각 아스키코드에 해당되는 값을 넣어야하고, null 값도 마지막에 넣어야한다. 이를 아스키코드 표를 참고하여 알아내면

0x31 39 32 35 35 36 62 62 00 이다.

이 상태에서 문자열의 시작주소를 계산할 수 있는데 본래의 return address의 공간에 %rsp값이 가리키고 있었는데 gets함수를 통해 버퍼입력값을 받기전에 0x38만큼 빼므로 그 전의 단계에서 %rsp값을 알아내면 return adresss공간의 주소를 알 수 있을 것이다.



이를 통해 버퍼시작주소를 담고자 하는 return address공간의 시작주소가 0x5562b5f0이며 여기서 16만큼 더한 0X5562b600 이 문자열의 시작주소임을 알 수 있다.

그래서 이를 %rdi 레지스터에 넣으면 되므로 이로써 명령어가 완성된다. 이를 위의 level2와 동일한 방법으로 컴파일 후 기계어로 바꾼 후 16진수를 확인하면 아래와 같다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이제 buf 입력값의 시작주소를 찾아보면

텍스트이(가) 표시된 사진

자동 생성된 설명

0x5562b5b8임을 알 수 있고, 이 주소가 명령어의 시작주소가 됨을 알 수 있다. 즉, 첫번째 return address여야 할 것이다.

이제는 touch3의 시작주소만 알면 된다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Disas touch3를 통해 시작주소가 0x401f62임을 알 수 있다. 지금까지 알아낸 값들을 토대로 txt 파일을 만들 수 있다.

Answer:

텍스트, 전자기기이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

**(4) Rtarget - Level2**

Rtarget의 경우에는 Gadget들을 활용해서 attack을 해야 한다.

즉, return address에서부터 한 줄 씩 입력을 해가는 방식이고, 이는 진행될 때마다 pop이 되어 사라진 다는 것을 생각하고 짰다.

Level2를 pass하기 위해서는 cookie와 동일한 값을 %rdi에 넣은 후에 touch2를 호출하면 된다.

그러나 여기서는 기존에 존재하는 코드들을 잘라서 가져와 실행하는 방식이므로 ctarget의 경우와 조금 다른 방식으로 진행해야 함을 알게 되었다.

일단 buf 입력값으로 cookie와 동일한 값과, touch2 시작주소를 넣어야 함은 당연한 사실이기에 이를 어떻게 실행시킬 지 생각을 해봤다. ctarget의 경우에는 코드를 직접 작성할 때 mov를 통해 쿠키값을 rdi레지스터에 넣었으므로, 이와 비슷한 코드를 찾아봤다. Farm에서 mov명령어와 pop명령어를 사용해 구현할 수 있을 것이라고 생각했다.

그래서 cookie값을 stack에 넣어둔 상태라고 생각하면 pop을 사용하면 stack을 pop시키면서 cookie값을 다른 레지스터로 옮길 수 있고, mov를 통해 cookie값이 들어가있는 레지스터에서 %rdi 레지스터로 값을 옮길 수 있기 때문이다.

그래서 생각해낸 코드는

popq %rax

movq %rax, %rdi 이다.

여기서, pop %rax의 코드가 담긴 주소값을 a, movq %rax, %rdi의 코드가 담긴 주소값을 b라고 한다.

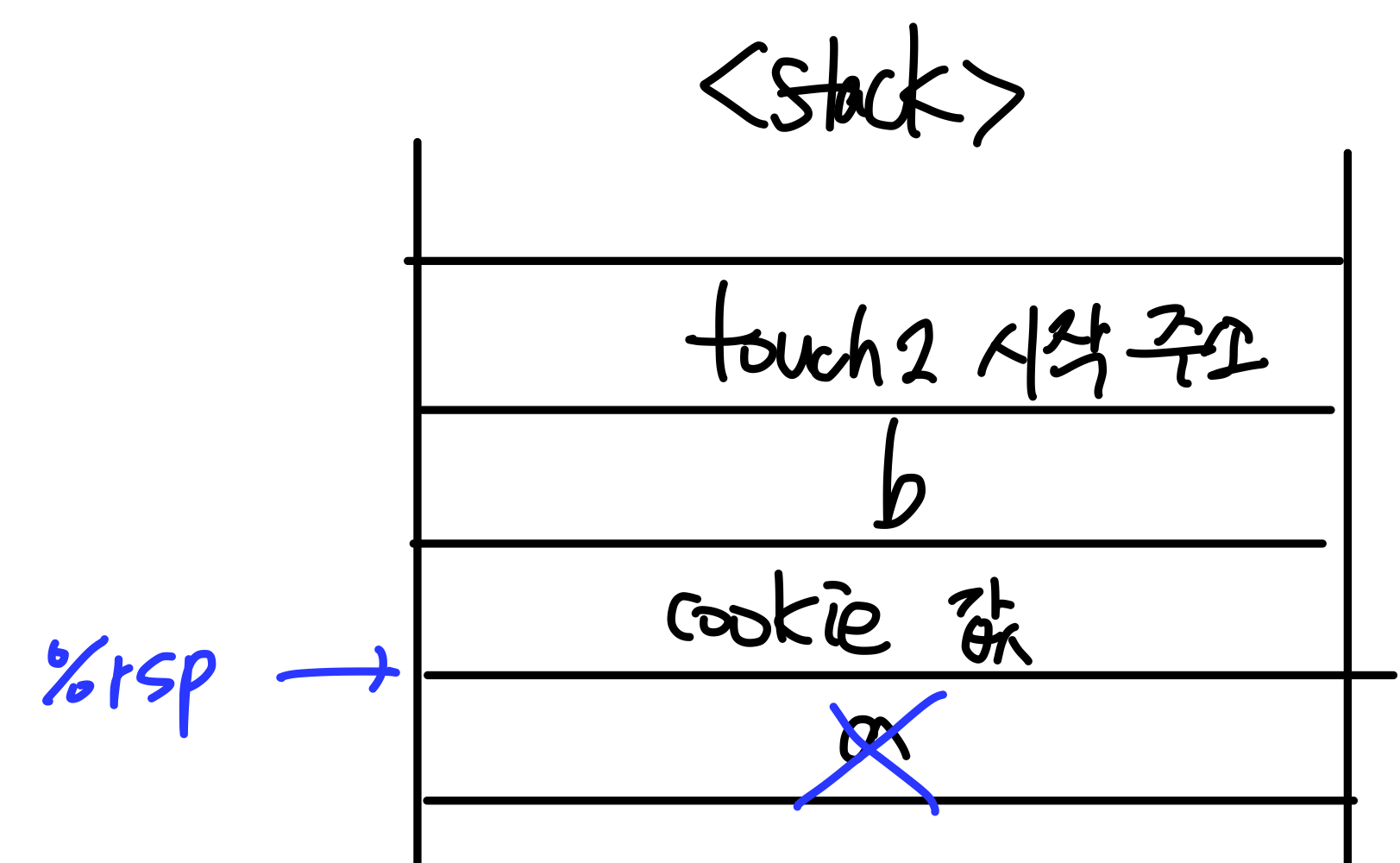
그러면

테이블이(가) 표시된 사진

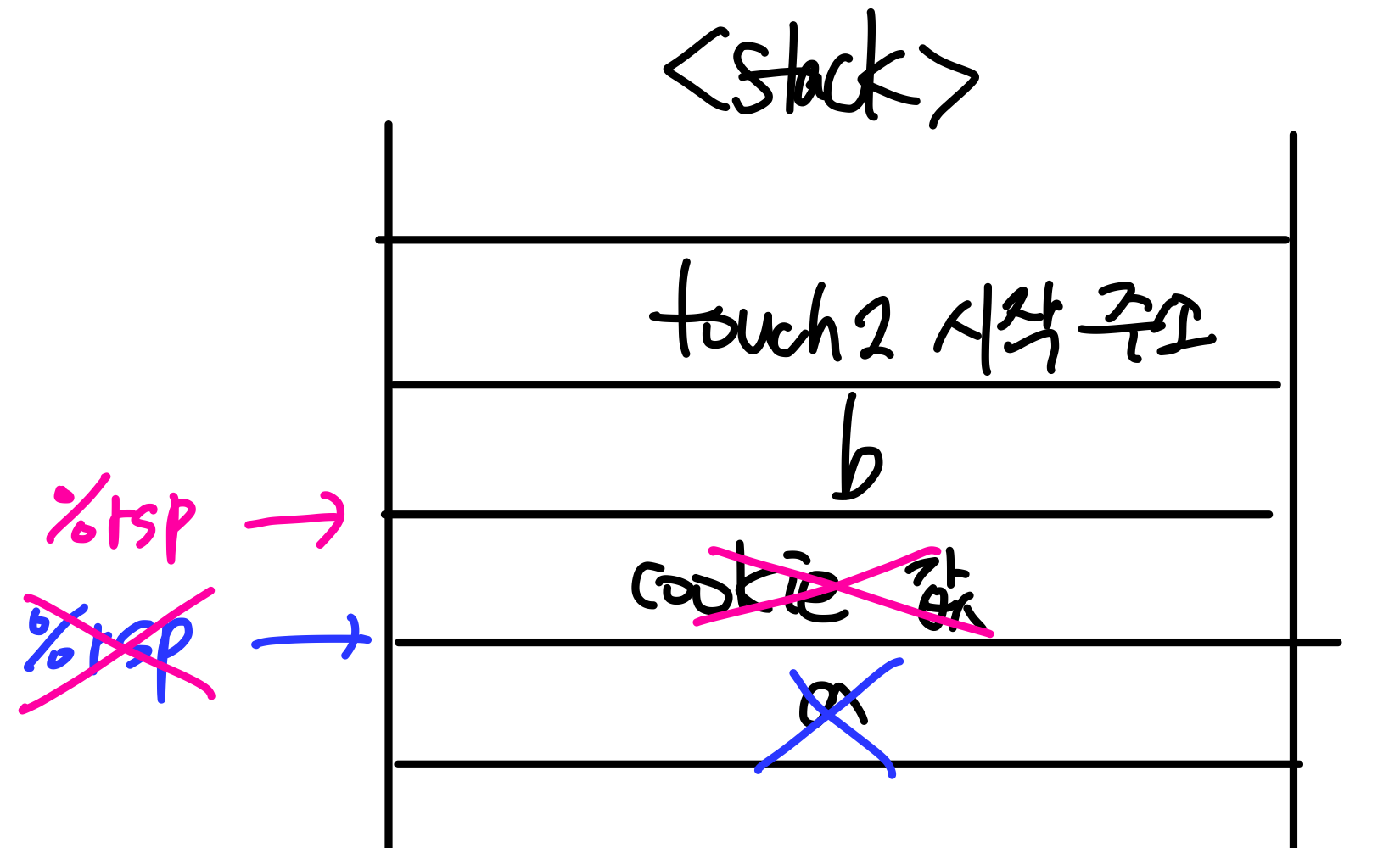
자동 생성된 설명

이런식으로 stack을 설정한다면, 먼저 a구간부터 시작하여 a가 가리키는 곳으로 이동하게 된다.

이동을 하면서 stack의 구조는 아래와 같이 변경되고, %rsp값이 업데이트되면서 cookie값이 저장된 곳을 가리킨다.



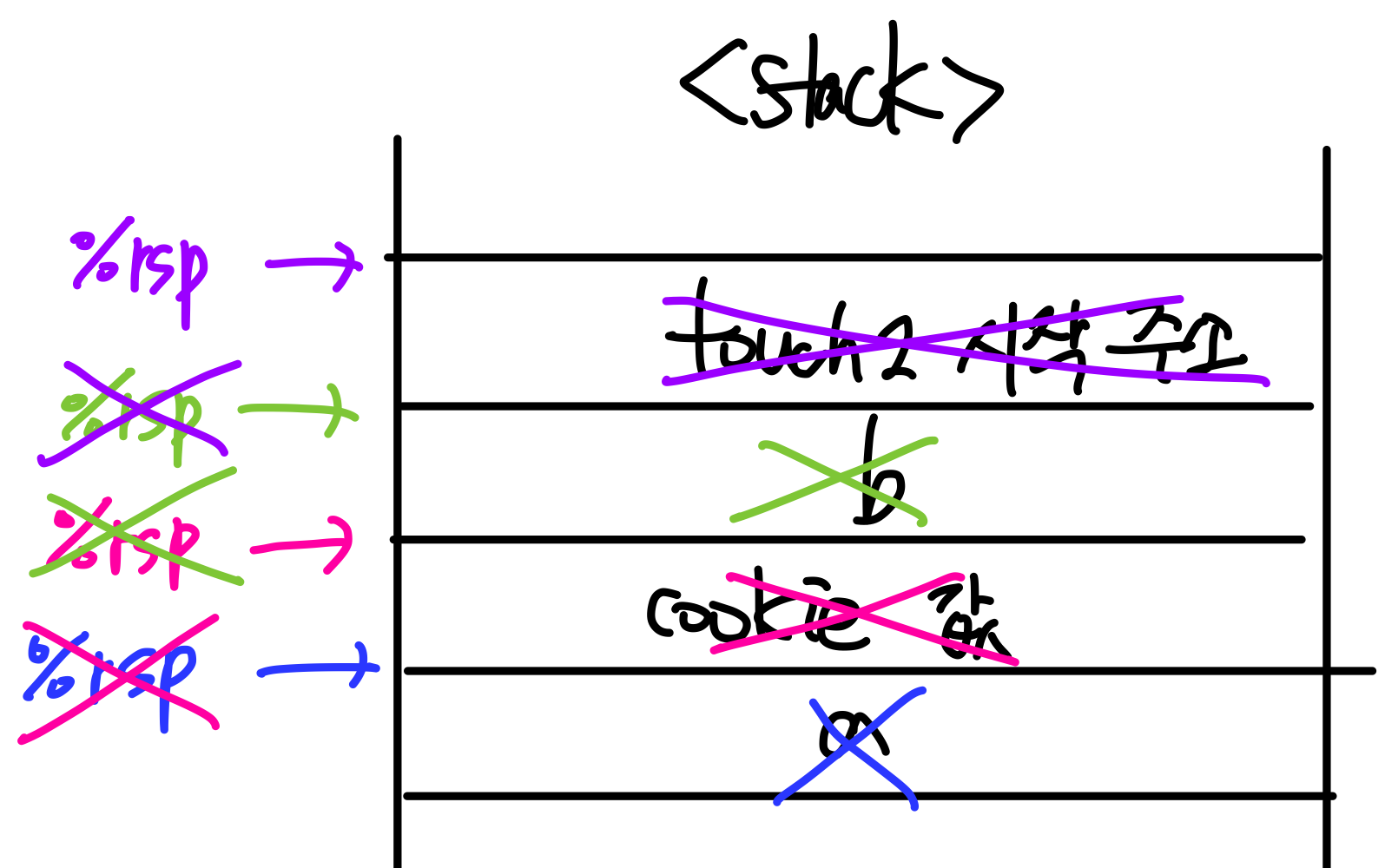
A가 가리키는 곳의 코드인 pop %rax를 실행하여 현재 stack의 top에 있는 값인 cookie 값을 %rax에 저장하고, %rsp값이 업데이트 되며 stack의 구조는 아래와 같이 한 번 더 업데이트 된다.



이제 return하면서 return address인 b가 가리키는 곳으로 이동하게 되고, stack의 구조는 업데이트되며 %rsp값이 바뀐다.



이 상태에서, b가 가리키는 곳에서 movq %rax, %rdi 코드를 실행하며 %rax에 저장되어 있던 cookie 값을 %rdi레지스터에 저장한다.



이후 return을 하면서 return address인 Touch2의 시작주소로 이동하고, 제대로 된 인자가 전달 된 채 touch2 함수가 실행된다.

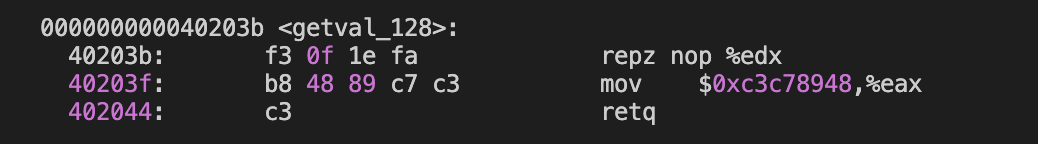
이제 이러한 과정이 올바른 과정임을 확인했으니, 정확한 코드들의 주소인, 즉 a,b 값을 찾아내면 된다. 우리가 가져와야 하는 코드는

먼저 pop %rax 인데 이는 58이므로 58 90 90 90 c3 (여기서 90은 nop) 을 잘라서 가져오면 된다. 즉, 40201f+2 = 402021이 pop %rax의 시작주소로 사용될 수 있다.

텍스트, 장치, 측정기, 게이지이(가) 표시된 사진

자동 생성된 설명

두 번째로는 movq %rax, %rdi 명령어인데 이는 48 89 c7이었으므로, 이를 포함하며 c3로 끝나는 코드를 getval\_128 함수에서 찾을 수 있고, 여기서 시작주소 값은 40203f+1 = 402040임을 알 수 있다.



이를 토대로 최종 txt파일을 만들면

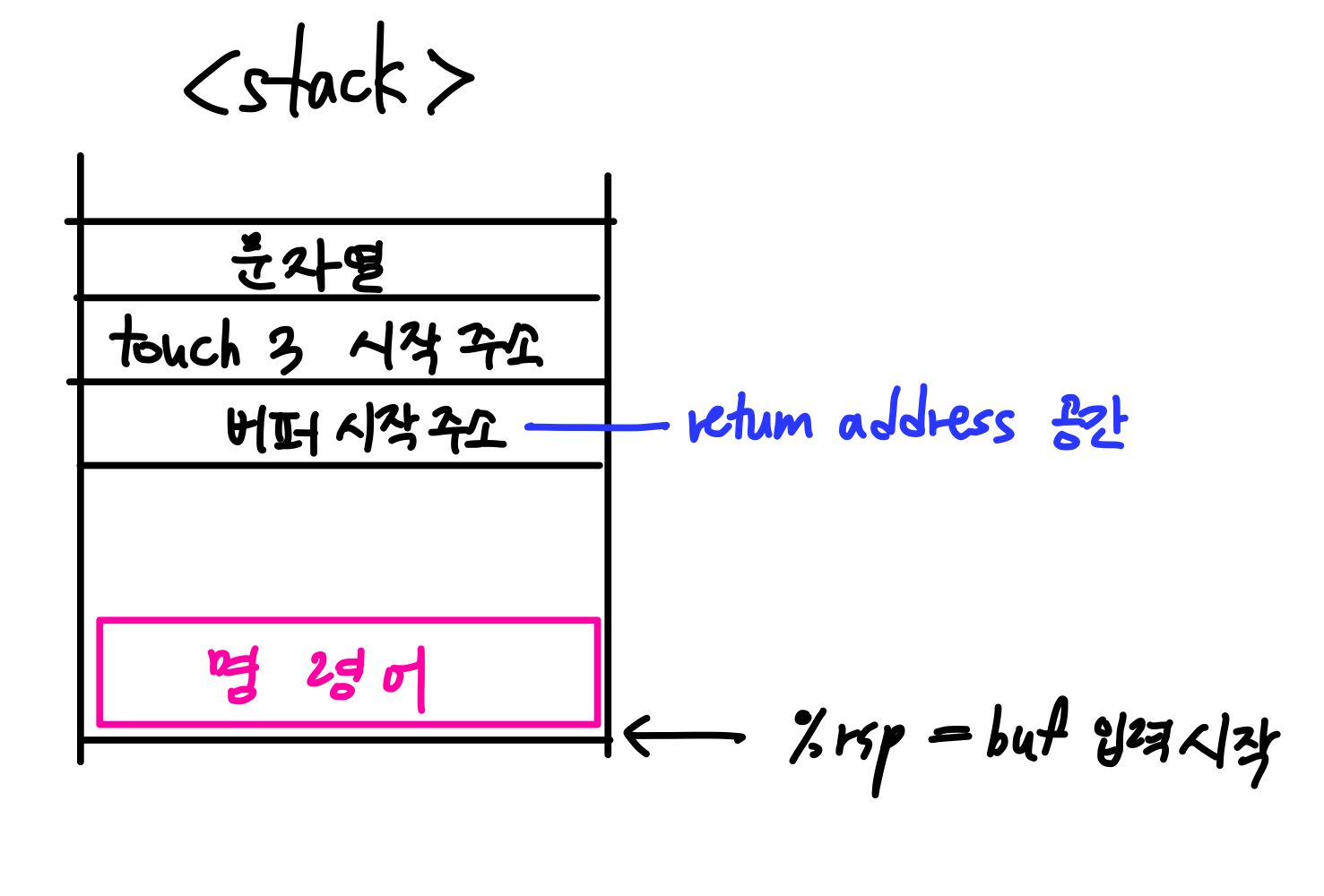
텍스트, 전자기기이(가) 표시된 사진

자동 생성된 설명 이 된다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**(5) Rtarget – level3**



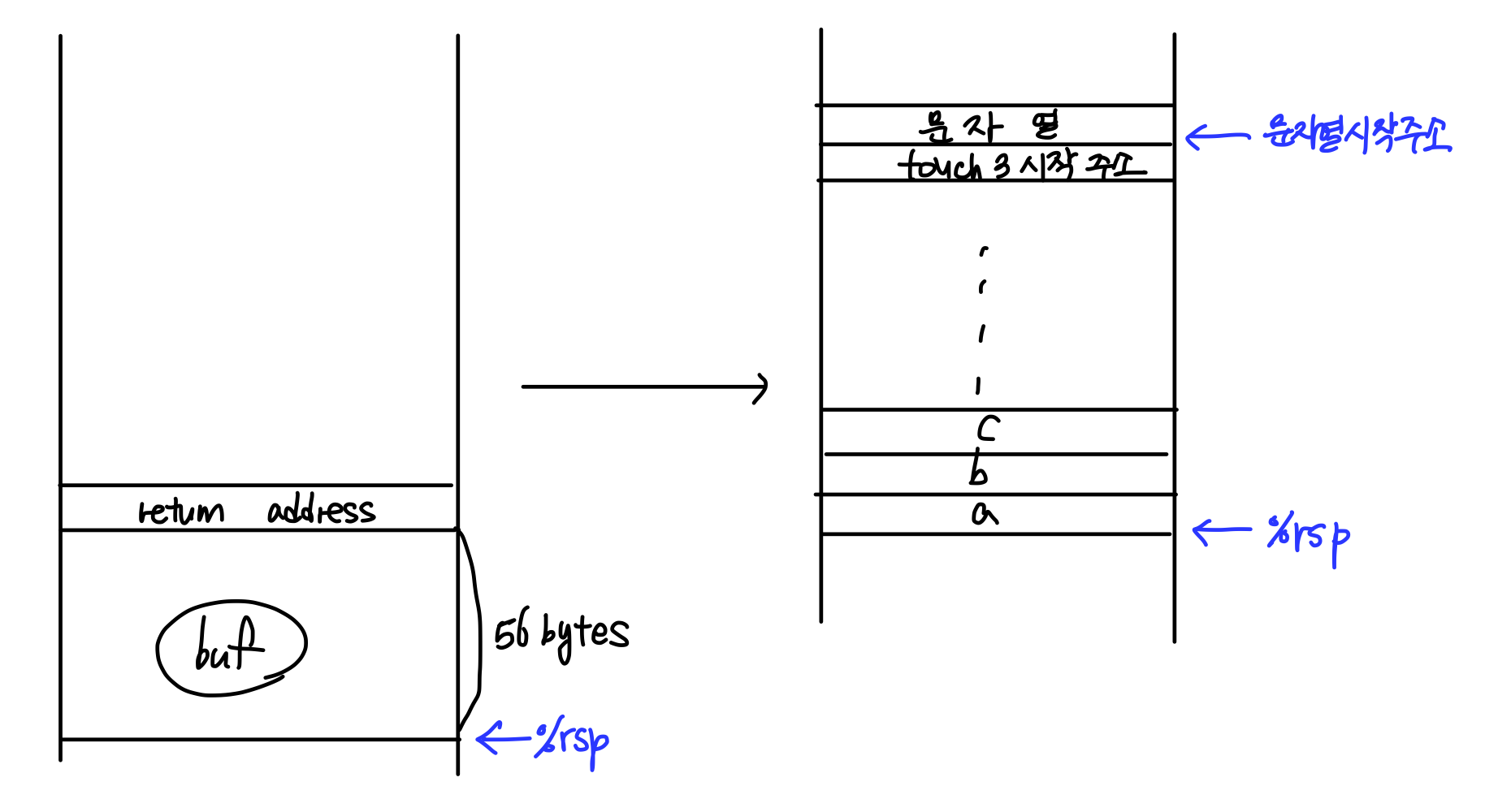
위에서 ctarget에서 풀었던 stack의 구조를 가져와봤다. 여기서 실행해야 하는 것들과 그들의 순서를 다시 상기시켜본다면, 문자열의 시작주소를 인자값으로 설정 -> touch3 호출이다. 마찬가지로 문자열은 touch3의 시작 주소가 담긴 공간보다 더 위에 담겨서 그 전에 삭제되지 않도록 해야 한다.

여기서 이제 문자열의 시작주소를 계산하여 넣으려면 movq가 아닌 다른 주솟값을 계산하여 대입할 수 있는 명령어가 필요함을 깨달았고, 그래서 lea 명령어가 있는 부분을 찾았다. 그래서 이를 사용해서 문자열의 시작주소를 계산하여 대입하는 것이 필요할 것이다.



여기서 보면 %rdi+%rsi를 계산하여 %rax에 대입하는 것을 알 수 있다. 즉, %rdi+%rsi 값을 계산한 것이 문자열의 시작 주소이어야 한다.

어떻게 문자열의 시작주소를 계산하는 지에 대해 고민해보기 위해 stack의 구조를 살펴봤다.



Buf 입력값에 56byte만큼 0으로 채운 후 return address부터 알맞는 값들을 집어넣으면 오른쪽 stack의 형태처럼 쌓일 것이다. 그리고 오른쪽 stack의 구조부터는 a가 가리키는 곳으로 가서 실행함과 동시에 %rsp값이 +8만큼 업데이트되고, stack이 업데이트되며, return하면서 b가 가리키는 곳으로 가고 stack이 업데이트 되는 형식이 계속하여 반복된다.

즉, touch3가 호출되기 전에 %rdi를 세팅하기 위한 코드들의 주소들이 나올 것이고, 그러한 코드들을 다 실행한 후에 touch3를 호출하게 된다. 그러므로 %rsp값에서 그 차이 만큼을 더해준다면 문자열의 시작주소를 계산할 수 있을 것이다. 그래서 %rsp값은 %rdi에 저장을 하고, 차이값 즉 offset 값을 %rsi에 저장하여 add\_xy함수에서 %rax 에 %rdi+%rsi를 할 수 있도록 했다.

즉, 내가 수행해야할 부분을 순서대로 설명한다면

1. %rsp값을 %rdi에 이동
2. Offset값을 %rsi에 이동
3. Lea 명령어를 통해 %rdi+%rsi를 %rax에 저장
4. %rax값을 %rdi에 이동하여 인자값 세팅
5. Touch3 호출
6. Cookie 문자열 값

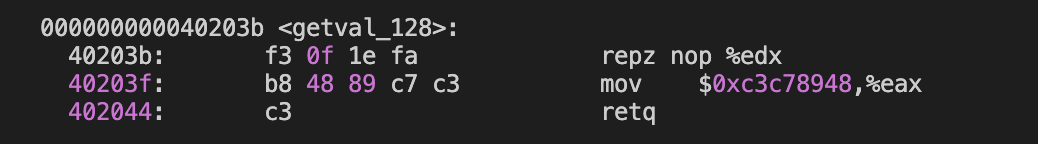
1. %rsp값을 %rdi에 이동

먼저, %rsp값을 %rdi에 저장하기 위해 movq명령어를 찾아보았지만 찾을 수 없었기에 %rsp 에서 %rax에 저장하는 명령어와 %rax에서 %rdi로 값을 이동하는 명령어를 사용하여 이동하는 형식을 채택했다.

movq %rsp, %rax는 pdf에 있는 표에 따르면, 48 89 e0이고, movq %rax, %rdi는 48 89 c7이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

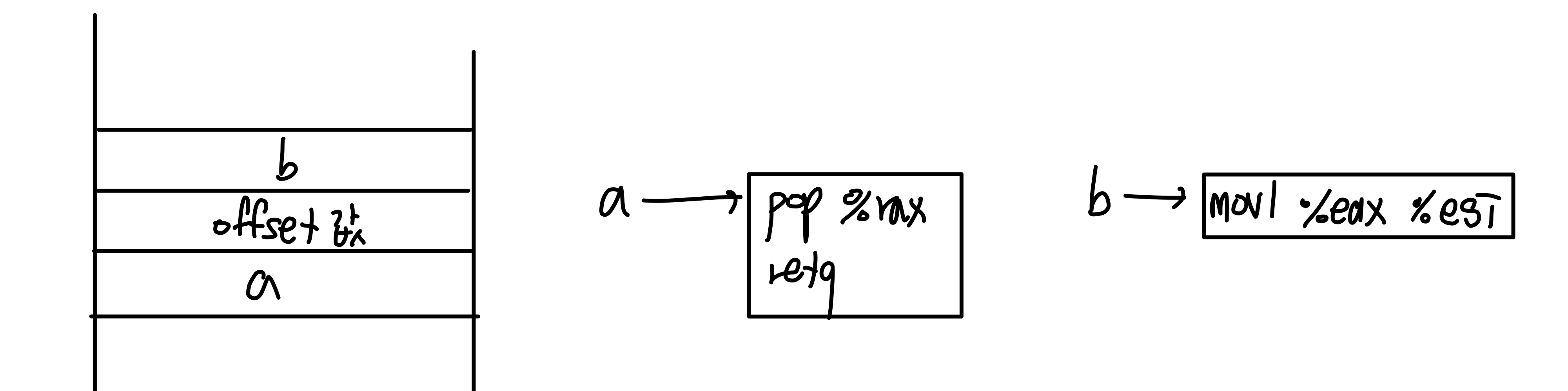


두 함수에서 각 부분을 찾을 수 있었고, 각 시작 주소는 4020f0, 402040임을 알 수 있었다.

2. Offset값을 %rsi에 이동

Offset에 해당하는 값을 입력값으로 같이 넣어줘야 하며 이를 레지스터에 넣어야 한다는 점에서 rtarget의 level2에서 cookie값을 stack에 넣은 상태에서 pop을 해주는 과정과 같은 과정으로 실행할 수 있음을 알 수 있다.

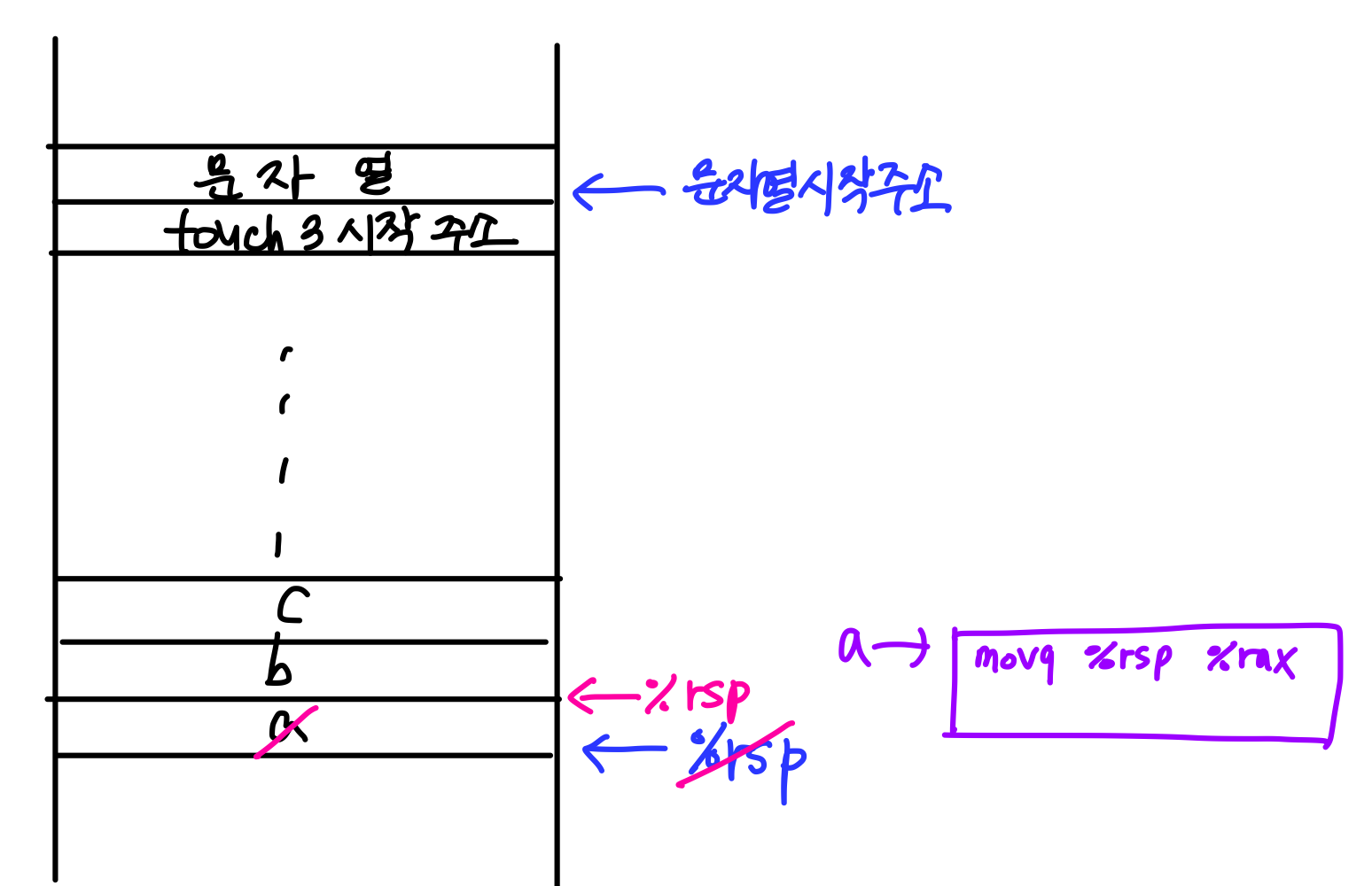
즉,



Stack에 위와 같은 구조로 넣는 다면, offset값을 %rax에 넣은 후 %rax에 있는 값을 %rsi에 넣어서 offset값을 %rsi에 넣을 수 있을 것이다.

Pop %rax 코드의 경우는 level2에서 사용했던 402021주소를 그대로 사용하면 된다.

여기서 offset의 값으로 상수값을 넣어야 하는데, 이는 %rdi에 들어가있는 %rsp값이 정확히 어디를 가리키고 있는지를 알면 쉽게 계산할 수 있다.



가장 먼저 movq %rsp %rax를 실행하고 있는데, 이 코드를 가리키고 있는 주소값 a 가 담긴 곳의 바로 윗 공간, 즉 그 다음 코드를 가리키고 있는 b가 담긴 곳의 시작주소가 %rsp값이고 이 값이 %rdi에 저장되고 있음을 알 수 있다.

즉, b부터 시작하여 문자열이 시작하는 주소까지 몇 번이 쌓이는지 확인한 후 곱하기8을 해주면 offset 일 것이다.

그리고 그 다음 고려해야 하는 부분이 바로 offset이 저장된 %rax에서 %rsi로 저장되어야 하므로 movl %rax, %rsi를 사용하면 된다. 하지만 해당되는 코드가 찾았을 때 없었으므로 다른 레지스터를 거쳐서 %rsi에 저장되도록 하는 방법을 택하였다.

그래서 movl 과 관련된 여러 경우의 수를 나열한 후 최종적으로 %rax에서 %rsi값으로 값을 이동할 수 있는 경우는 %eax->%edx->%ecx->%esi이다. 해당하는 코드들을 찾은 결과는 다음과 같다.

movl %eax, %edx에 해당하는 89 c2 -> 시작 주소 : 4021c4

텍스트이(가) 표시된 사진

자동 생성된 설명

Movl %edx, %ecx에 해당하는 89 d1 -> 시작 주소 : 4020d0

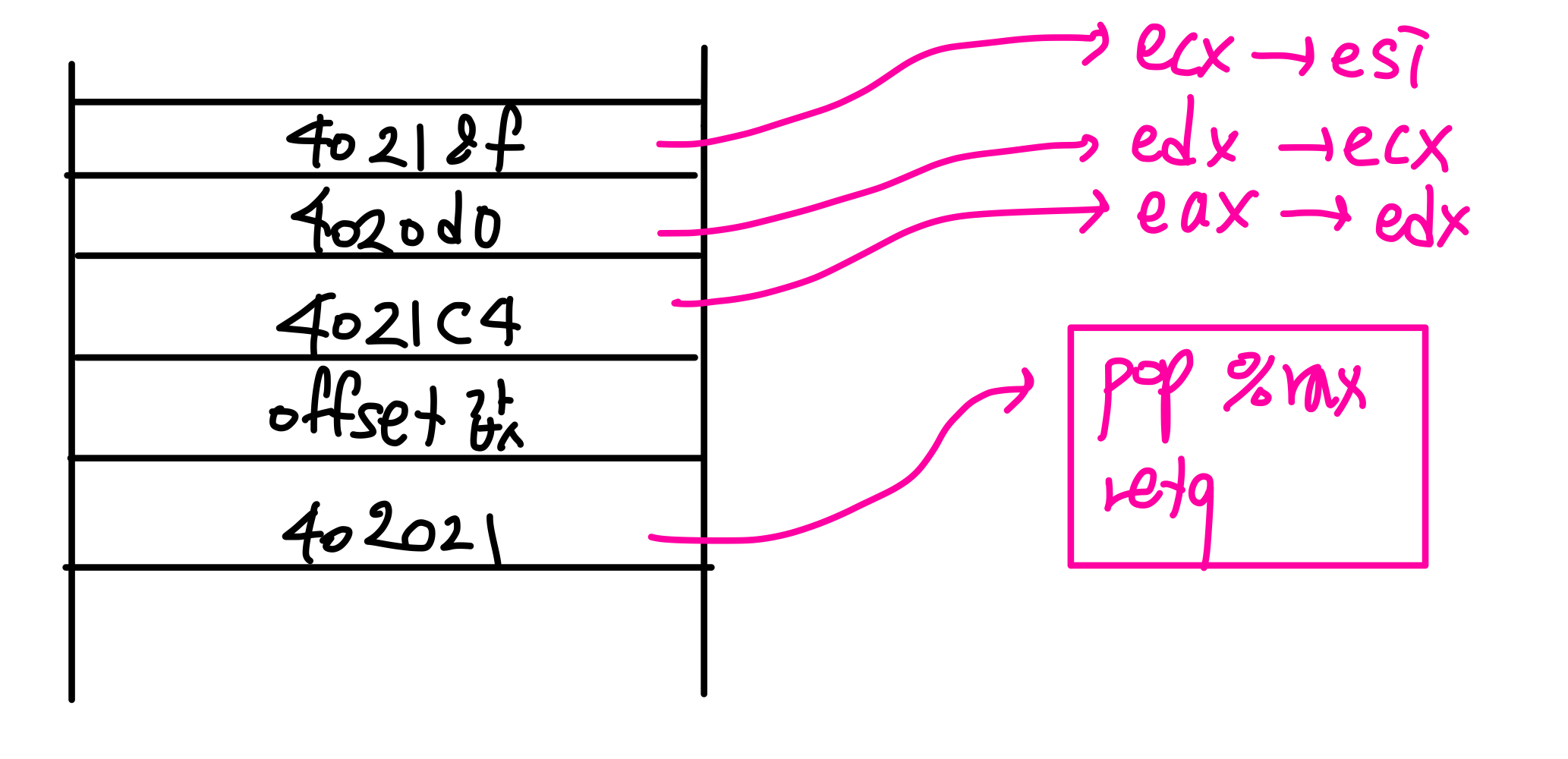
텍스트이(가) 표시된 사진

자동 생성된 설명

Movl %ecx, %esi 에 해당하는 89 ce -> 시작 주소 : 40218f

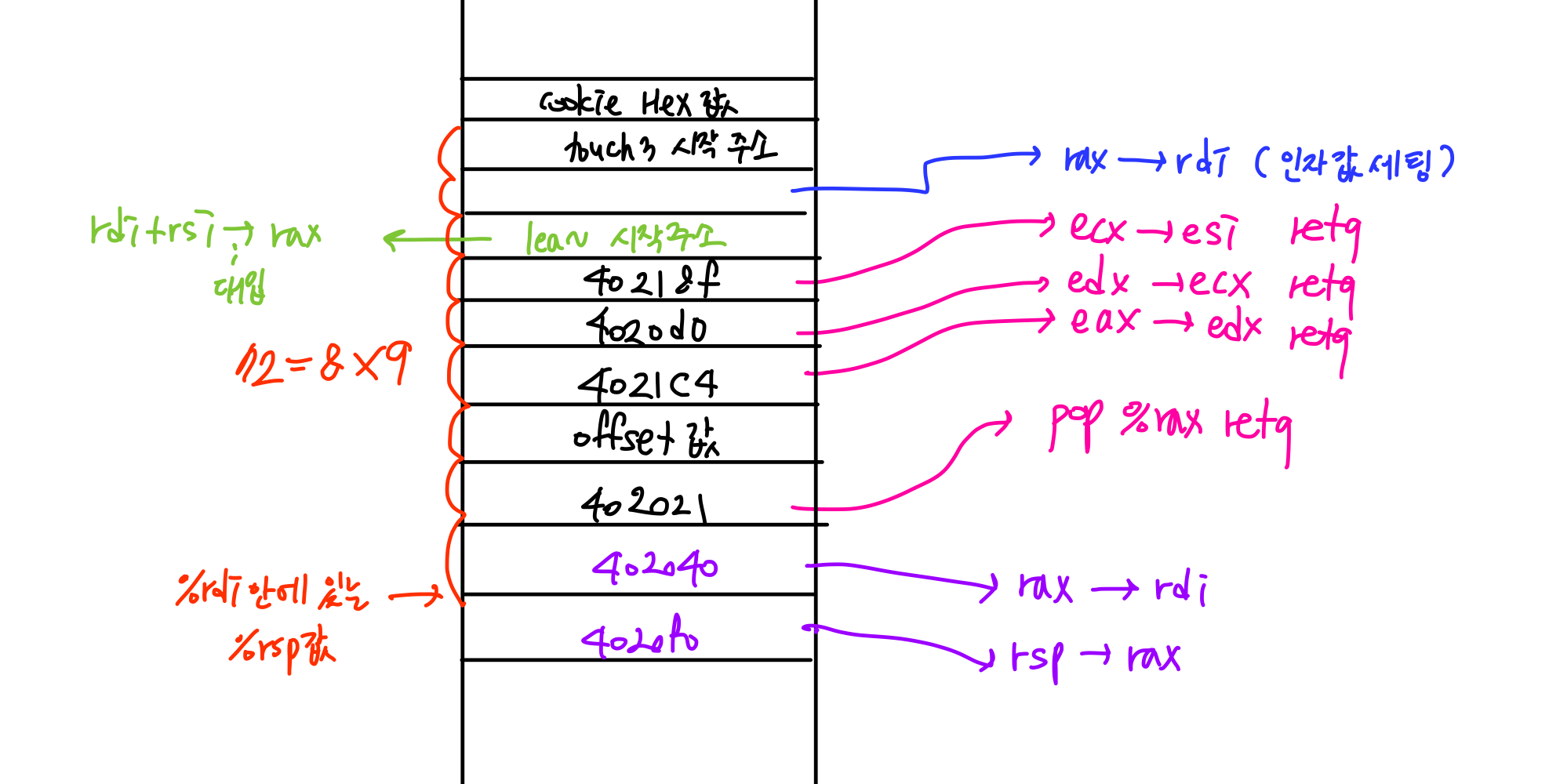
텍스트, 장치, 측정기이(가) 표시된 사진

자동 생성된 설명



Stack구조는 위와 같이 표현될 것이다.

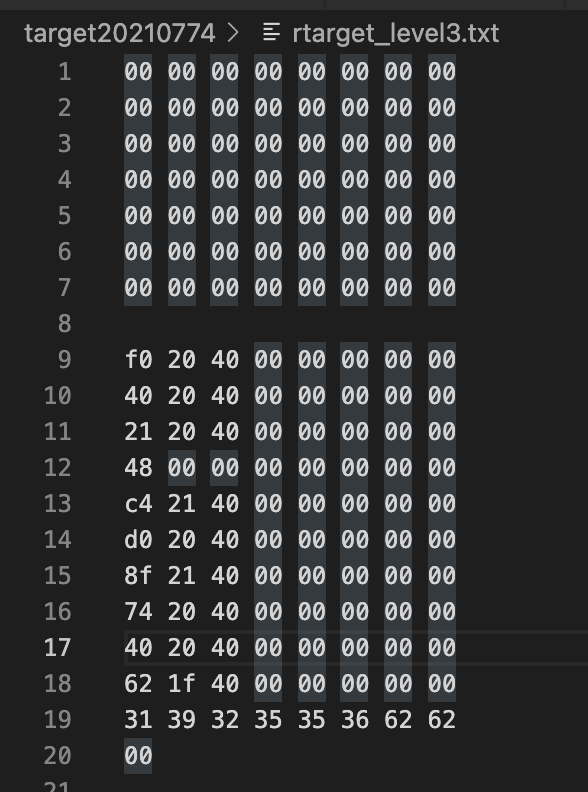
그리고 지금까지 고려된 사항들에 따라 stack구조를 모두 종합해본다면

 이 된다.

즉, rsi 레지스터까지 설정이 끝나면 lea 명령어를 수행하는 곳으로 가서 rdi+rsi 값을 rax에 넣어주고, rax값을 rdi에 넣어주면서 인자값을 세팅하고, touch3를 호출하는 것이다.

이로써 전체적인 코드가 나오고 offset를 구할 수 있는데, 위에서 언급했듯이 402040이 들어가 있는 곳이 %rdi에 대입된 %rsp값이므로 여기서 총 9줄을 건너면 cookie 문자열이 들어가있는 곳으로 갈 수 있다. 그러므로 offset은 8\*9 = 72이고 16진수로 48이 된다.

해당 값들을 모두 정리하여 txt파일에 입력하면 다음과 같고 문제가 해결된다.



텍스트, 타이프라이터, 키보드이(가) 표시된 사진

자동 생성된 설명