2018년 2학기

컴퓨터 SW시스템 개론

CSED211

Lab Report #4

담당교수 : 김종

학번 : 20160074

학과 : 화학공학과

이름 : 고진민

POVIS ID : eric9709

--------------------------------------------------------------------------------------------------------------

<명예서약>

나는 이 프로그래밍 과제를 다른 사람의 부적절한 도움 없이 완수하였습니다.

--------------------------------------------------------------------------------------------------------------

1. **Level별 설명**

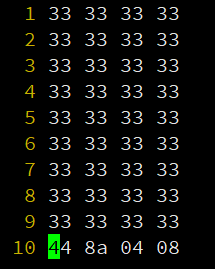
* **Level 0: Candle**

Level 0는 수업 시간에 설명을 간단하게 진행했다. 앞으로의 level에서 사용할 방법을 간단하게 익히는 느낌이었다. Level 0의 목표는 smoke 함수를 호출하는 것이다. Smoke 함수는 호출되면 바로 프로그램을 종료시키기 때문에 호출만 하면 추가 작업은 필요 없다.

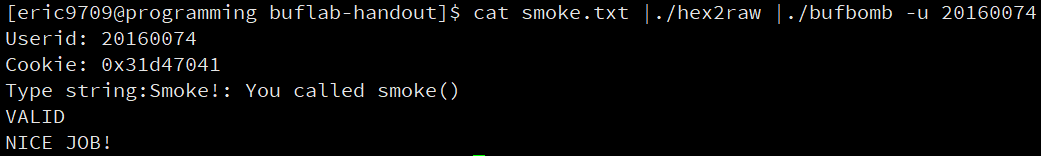
Test 함수에서 getbuf 함수를 호출하고, 이 함수에서는 gets를 이용해 32byte 크기의 array에 값을 입력받는다. 여기서, gets는 입력받는 변수의 크기를 넘어서도 입력을 받을 수 있다. 이 때문에 다은 데이터를 오염시키는 것이 가능하다. Stack의 구조를 생각해보자. Test 함수에서 getbuf 함수를 호출하면 parameter를 push하고 return address를 push한 뒤 getbuf에서 old ebp를 push하고 ebp가 그곳을 가리킨다. 그리고 esp를 갱신한다. 그 다음에는 array의 값이 저장된다. 따라서 esp부터 32byte는 buffer가 저장되어 있고 그 위로 4byte는 old rbp, 그 위로 4byte는 return address가 저장되어 있다. 이 return address 부분을 buffer overflow로 오염시키면 원하는 주소로 return이 가능해진다. Stack의 구조를 표로 나타내면 아래와 같다. 한 칸이 4byte라고 생각하면 된다.

|  |
| --- |
| Data of function test |
| Parameter |
| Return address |
| Old rbp |
| Buffer |
| Buffer |
| Buffer |
| Buffer |
| Buffer |
| Buffer |
| Buffer |
| Buffer |

우리는 smoke 함수의 주소를 gdb에서 disas로 쉽게 알 수 있다. 실제 값은 0x08048a44 이다.

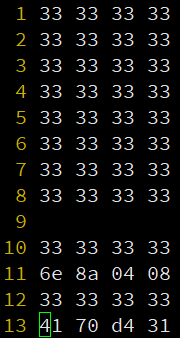


위가 우리가 만든 smoke.txt에 저장된 buffer 값이다. 여기서 주의할 점은 우리가 사용하는 Intel CPU는 little endian을 사용하므로 주소 입력 시 두 자릿수씩 잘라서 거꾸로 입력해야 한다. 이를 이용해 bufbomb를 실행하면 다음과 같은 결과를 얻을 수 있다.

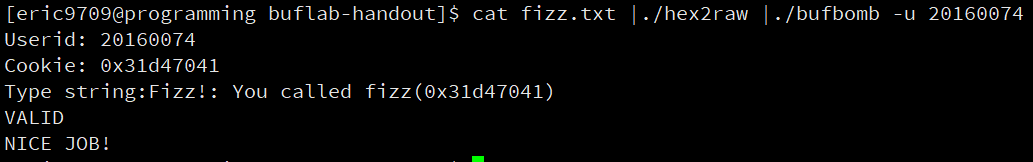


* **Level 1: Sparkler**

Level 1은 특정 함수 fizz를 호출하고, 아이디별로 고유의 값을 가지는 cookie와 같은 크기를 val이 가지도록 해야 한다. 이 때 test 함수에 val이라는 parameter가 하나 있으므로, 이 val 값이 위의 stack의 도식에서 parameter 위치에 저장된다. 이 부분의 값 역시 우리가 임의로 조절할 수 있다. 내가 가지는 고유 cookie의 값은 0x31d47041이고, 이 값을 거꾸로 parameter 위치에 저장하면 된다. Fizz의 주소는 0x08048a6e이므로, 이 값을 return address에 넣고 그 다음으로 cookie의 값을 넣으면 된다. 이 값을 fizz.txt에 저장하면 다음과 같다.

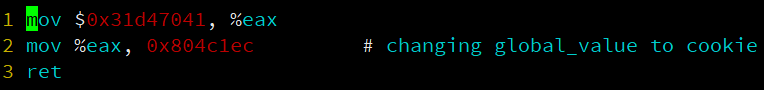


위의 그림에서 처음 32byte는 buffer에 저장되고 10에 위치한 값은 old ebp, 11에 위치한 값은 return address이다. 원래는 그 다음에 바로 argument에 넣을 값을 적어야 하지만 실제로 gdb를 이용해 분석한 결과 getbuf 함수가 return을 진행한 후 저장된 값에서 return address가 사라져 4byte가 당겨졌다. 이 때문에 4byte를 추가로 채운 뒤 우리가 원하는 cookie 값을 넣어줘야 한다. 이를 이용해 실행한 결과는 다음과 같다.

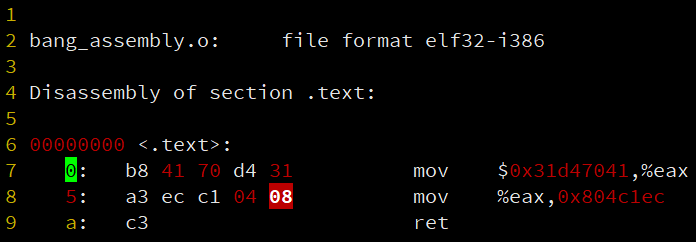
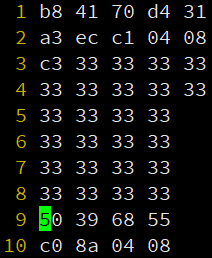


* **Level 2: Firecracker**

Level 2는 global value를 원하는 값(cookie)으로 바꿔야 하는 문제이다. 이 global value는 우리가 buffer에 입력함으로써 바로 바꿀 수 있는 값이 아니기 때문에 다른 방법을 사용해야 한다. 이 방법을 위해서는 우리가 buffer 영역에 이 global value를 cookie 값으로 바꾸는 간단한 어셈블리 코드를 작성하고, 이를 실행시켜야 한다. 이 어셈블리 코드는 buffer의 맨 끝부분에 저장하고, 위의 data stack을 나타낸 표에서 return address에 해당하는 위치에 이 buffer의 끝부분의 주소를 저장한다. 그러면 getbuf 함수가 return 되면 우리가 임의로 작성한 어셈블리 코드가 실행된다. 이 때 esp는 parameter 부분을 가리키고 있을 것이다. 따라서 우리가 임의로 짠 어셈블리 코드에서 esp 값을 따로 건드리지 않는다면 ret 수행 시 pop %eip 가 수행되어 parameter에 있는 값이 %eip 레지스터로 넘어가 해당 위치의 코드를 수행할 것이다. 우리는 이 위치(주소)가 함수 bang의 위치가 되도록 해야 한다.

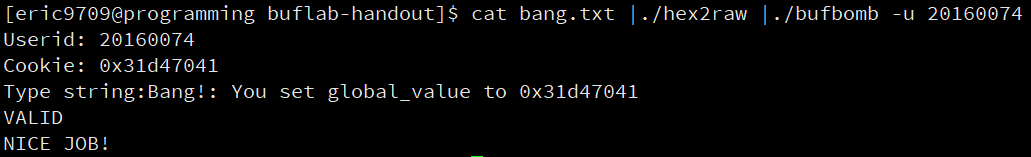


먼저 위가 우리가 임의로 작성한 어셈블리 코드이다. 이 때 global value의 저장 위치가 0x804c1ec임을 gdb를 통해 알 수 있다. 이를 변환하면 하단 왼쪽과 같다.

왼쪽의 값을 buffer 입력값에 넣은 뒤, 36 byte까지 남는 값은 임의의 코드로 채우고 return address 부분을 buffer의 시작 부분에 넣으면 오른쪽과 같은 bang.txt를 만들 수 있다.

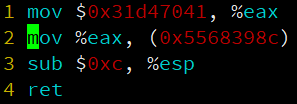
Line 7까지가 buffer의 범위인 32byte, line 8이 old ebp, line 9가 return address, line 10이 parameter의 위치이다. 우리는 gdb를 통해서 buffer의 저장 시작 위치가 0x55683950 임을 알 수 있다. 또한 두 번째 return address로 사용할 함수 bang의 주소 역시 gdb를 이용해 0x8048ac0 임을 알 수 있다. 이를 이용해 프로그램을 실행하면 다음과 같은 결과를 얻을 수 있다.



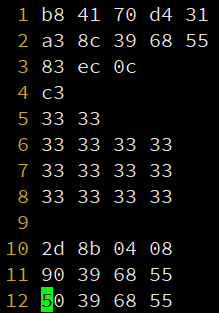
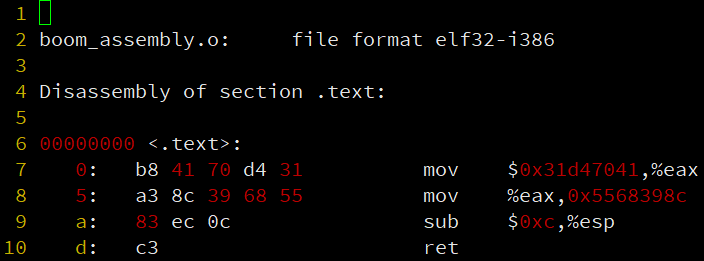
* **Level 3: Dynamite**

Level 3은 여태까지와는 다르게 test 함수로 다시 돌아가서 끝을 내야 하는 경우이다. 여태까지는 우리가 임의로 호출한 함수에서 바로 프로그램을 종료할 수 있었으나 이번에는 그렇지 않다.

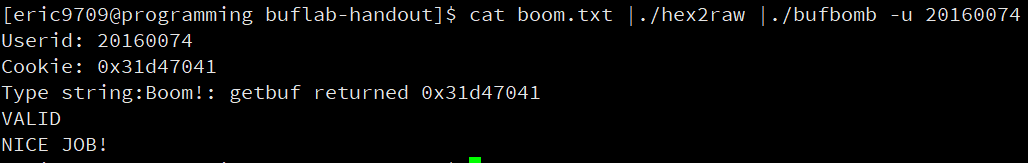
Test 함수를 보면 val에 getbuf 함수로 값을 입력받지만 우리는 이 val이 cookie 값과 일치하도록 해야 한다. 또한 local value에 uniqueval이 랜덤으로 생성되어 저장되므로 이 부분 역시 건드리지 말아야 한다. 우리가 해야 할 일은 val의 값을 cookie로 바꾸고, 이외의 test 부분의 stack data는 건드리지 말아야 한다. 이를 위해서 기본적으로 level 2에서 했던 것과 비슷한 방법을 쓸 것이나 이번에는 esp를 살짝 바꿔 test 부분의 stack data를 보존할 것이다.



위는 우리가 buffer의 맨 앞에 넣을 어셈블리 코드이다. Line 2까지는 val에 cookie를 넣는 과정이고, line 3은 esp에서 12를 빼 esp가 buffer의 마지막 4 byte 영역을 가리키게 해 두었다. 이 영역에 기존에 test에서 getbuf 함수를 호출한 바로 다음 부분의 주소를 넣으면, 함수가 정상적으로 실행되는 것처럼 속일 수 있다. 위를 변환하면 아래의 왼쪽과 같다.



변환한 값을 boom.txt에 옮긴 뒤 buffer의 28byte까지 임의의 값을 채우고, 이후 순서대로 test 함수의 return address, test 함수의 ebp, buffer의 시작 address를 4 byte씩 넣는다. 그러면 왼쪽과 같아진다. 여기서 기존에 있던 test 함수의 ebp를 원래대로 돌려주어야 하는 이유는, test로 돌아간 이후의 과정에서 ebp의 값을 필요로 하는 과정이 존재하기 때문이다. Esp의 경우는 이를 이용해 수행되는 과정이 없으므로 돌릴 필요가 없고, 돌리기도 어렵다. 이를 이용해 프로그램을 실행하면 다음과 같은 결과를 얻는다.



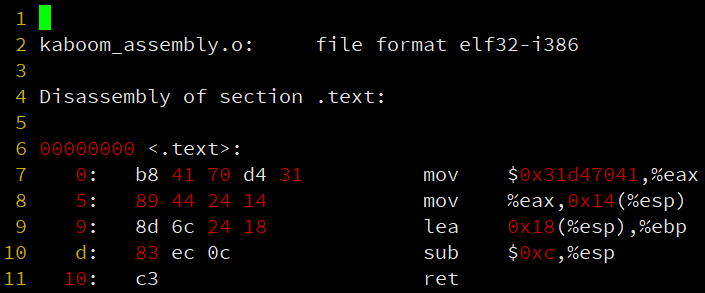
* **Level 4: Nitroglycerin**

Level 4는 level 3과 비슷하지만 test의 stack address가 바뀐다. 이는 실행 시마다 바뀌지는 않으며, 실행 후 연속으로 5회 실행되는데 이 때 연속으로 실행되는 과정에서 바뀜을 gdb를 통해 확인할 수 있다. 또한 nitro mode를 사용하기 때문에 buffer의 크기가 512 byte로 늘어나게 된다. 계속해서 stack address가 바뀌지만 다행히 범위가 존재하기 때문에 이를 이용해 어떤 상황에서도 우리가 원하는 행동을 할 수 있다.

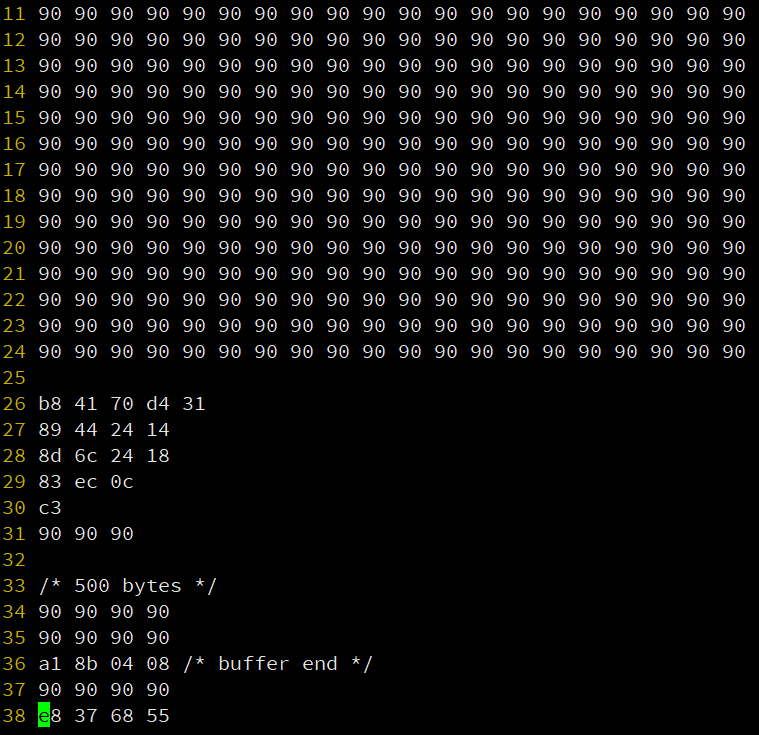
Gdb를 이용해 찾으면, 5번의 수행 동안 나타나는 testn 함수의 ebp 값은 0x55683990, 0x556839c0, 0x556839e0, 0x55683a00, 0x55683930 이다. 또한 초기 esp는 ebp-0x18이다. 이를 통해 우리는 getbufn 함수에 대해서도 esp와 ebp를 계산 가능하다. Getbufn의 경우에서는 ebp와 초기 esp의 차이가 0x20e이다.

우리가 할 일은, 처음 return address를 getbufn에서 buffer의 주소가 시작하는 부분 중 가장 큰 부분으로 해 두고, buffer 부분은 전부 nop에 해당하는 90으로 채운다. 이후 buffer의 마지막 4byte는 testn의 돌아가야 하는 주소로 바꾼다.

Level 3과의 차이점은, ebp가 매 회마다 바뀌므로 일단 old ebp 영역을 임의로 설정한 뒤, 우리가 임의로 짠 어셈블리에서 esp를 이용해 다시 바꿔주면 된다. 이 때 esp는 우리가 건드리기 전이어야 한다. 이후, esp에서 적절한 값을 빼 return address를 조작한 부분으로 옮긴다. 이 과정은 level 3과 같다. 이를 구현한 어셈블리가 다음과 같다.



Line 7~8은 val에 cookie를 옮기는 과정이고 이 과정 역시 esp가 매번 바뀌므로 esp에 dependent하게 해야 한다. 상수로 하면 한 경우만 정상적으로 수행될 것이다. Line 9가 우리가 망가뜨린 ebp를 복구하는 과정이고, 마찬가지 이유로 esp dependent해야 한다. Line 10은 우리가 미리 바꿔놓은, testn의 돌아가야 하는 주소로 이동시킨 것이다. 이후 ret하면 원래 수행되던 순서대로 testn이 수행될 것이다.



위 이미지는 kaboom.txt이다. 잘린 윗부분은 90이 줄마다 20개씩 적혀 있다. 이는 어느 지점으로 처음에 getbufn에서 return해도 쭉 nop를 수행하다 결국 위에서 만든 어셈블리 코드가 수행되도록 한 것이다. Line 36은 돌아갈 testn의 주소, line 37은 원래 old ebp가 있었던 자리, line 38은 어떤 경우에도 nop를 타고 내려올 수 있는 주소이다. 이를 이용해 프로그램을 실행하면 다음과 같은 결과를 얻을 수 있다.

