Stack smash protector message를 이용한 memory leak

Sanguine@leaveret

릭이 필요한 상황인데, 도저히 릭할 벡터가 없다면? 한번 ssp 메시지를 이용해보자.



위 그림과 같이 간단한 stack memory corruption 류의 취약점을 가진 프로그램이 있다.

컴파일 옵션은 –mpreferred-stack-boundary=2를 주었다.

ssp가 설정되어 있으므로, 14를 입력하는 순간 canary가 덮히게 되고 \_\_stack\_chk\_fail함수가 불려지게 되어 ssp메세지가 출력되고 프로그램은 abort된다.

여기서 재미있는 사실은, ssp메시지의 프로그램명 부분이 argv[0]의 영향을 받는다는 것이다.



그렇다면 ret에서 argv[0]포인터가 저장된 주소의 offset이 일정하다면, 그 부분을 덮어서 원하는 메모리 값을 읽어 올 수 있을 것이다.

그렇다면 offset이 항상 일정한지, 정확한 값은 어떤지 한번 entry point에서 main함수를 호출하기 이전까지 보자.

’

ep는 당연히 \_start부터 시작하고, bp를 걸고 실행시켜보면,



esp는 argc와 \*argv, \*envp들이 차례로 스택에 쌓여져 있는 모습을 확인 할 수 있다.



이 부분 때문에 정확한 offset을 단정할 수 없다.

\_\_libc\_start\_main에서는 main함수를 호출하는 코드는



이부분으로써, esp-70으로 값이 고정이 되어있는데, esp+0x70에 저장된 값은



에서 \_\_libc\_start\_main을 call하기전에 push해준 값이다.

그렇다면, \_start의 코드를 한번 보자.

 0x080483a0 <+0>: xor ebp,ebp

 0x080483a2 <+2>: pop esi //실행되고 나면 현재 esp가 argv[0]을 가르킴

 0x080483a3 <+3>: mov ecx,esp

 0x080483a5 <+5>: and esp,0xfffffff0 // esp가 0~15만큼 감소

 0x080483a8 <+8>: push eax // esp -4

 0x080483a9 <+9>: push esp //esp -4

 0x080483aa <+10>: push edx //esp -4

 0x080483ab <+11>: push 0x8048560 //esp -4

 0x080483b0 <+16>: push 0x80484f0 //esp -4

 0x080483b5 <+21>: push ecx //esp -4

 0x080483b6 <+22>: push esi //esp -4

 0x080483b7 <+23>: push 0x804849d //esp -4

 0x080483bc <+28>: call 0x8048390 <\_\_libc\_start\_main@plt>

 그러면 한번 공식을 세워보자.

&(main’s ret) = &argv[0] – [0x0~0xf] – 0x20 – 0x70 – 4

– [0x0~0xf] – 0x20 은 \_start루틴에서 esp가 감소되는 값이다.

– 0x70 – 4은 \_\_libc\_start\_main에서 esp가 감소되는 값이다. -4는 main이 call되면서 ret이 저장되면서 감소되는 값

이런 공식이 성립되고, 이것을 &argv[0]에 대한 식으로 바꾸면,

&argv[0] = &(main’s ret) + [0x0~0xf] + 0x20 + 0x70 + 4

&argv[0] = &(main’s ret) + [0x0~0xf] + 0x94

가 된다.

내 환경에서는 0x0~0xf가 4로 일정했기 때문에, 한번 테스트를 해보면,



다음과 같이 정확이 ret위치부터 argv[0]까지의 위치를 계산해 낼 수 있다.

그럼 한번 got 메모리 릭을 해보자.



다음과 같이 파일 명이 나와야 할 부분에 릭이 되는 것을 볼 수 있다. 그리고 이는 리모트 환경이라면 쉽게 읽어 들일 수 있고, 로컬환경이라면 subprocess나 pipe등을 이용해서 처리 할 수 있을 것이다.

간단하게 데몬으로 만들어서 테스트 해 보았다.



릭이 잘 되는 것을 알 수 있다.

2번째 방법은 \_\_stack\_check\_fail이 어떻게 argv[0]의 주소를 가져오는지 분석을 해 보았다.

\_\_stack\_check\_fail은 내부에서 다시 \_\_fortify\_fail이란 함수를 호출하고

\_\_fortify\_fail은 libc에 저장된 argv를 사용해서 메시지를 출력한다.

<\_\_stack\_check\_fail>

 0xb7f212f0 <+0>: push ebx

 0xb7f212f1 <+1>: call 0xb7f4c46b //ebx 에 0xb7f212f6(call 다음 주소) 를 넣는다.

 0xb7f212f6 <+6>: add ebx,0xaed0a //ebx + 0xaed0a

 0xb7f212fc <+12>: sub esp,0x8

 0xb7f212ff <+15>: lea eax,[ebx-0x46e65] //eax에 “stack smash detect”주소 들어감

 0xb7f21305 <+21>: call 0xb7f21310 <\_\_fortify\_fail>

<\_\_fortify\_fail>

 0xb7f21310 <+0>: push ebp

 0xb7f21311 <+1>: mov ecx,0x5

 0xb7f21316 <+6>: push edi

 0xb7f21317 <+7>: mov ebp,eax

 0xb7f21319 <+9>: push esi

 0xb7f2131a <+10>: mov esi,eax

 0xb7f2131c <+12>: push ebx

 0xb7f2131d <+13>: call 0xb7f4c46b // ebx 에 call 다음 주소를 넣는다.

 0xb7f21322 <+18>: add ebx,0xaecde

 0xb7f21328 <+24>: sub esp,0x2c

 0xb7f2132b <+27>: lea edi,[ebx-0x47e09]

 0xb7f21331 <+33>: repz cmps BYTE PTR ds:[esi],BYTE PTR es:[edi]

 0xb7f21333 <+35>: lea edi,[ebx-0x46e4d]

 0xb7f21339 <+41>: seta al

 0xb7f2133c <+44>: setb dl

 0xb7f2133f <+47>: sub eax,edx

 0xb7f21341 <+49>: movsx eax,al

 0xb7f21344 <+52>: cmp eax,0x1

 0xb7f21347 <+55>: sbb esi,esi

 0xb7f21349 <+57>: lea eax,[ebx-0x48ad7]

 0xb7f2134f <+63>: add esi,0x2

 0xb7f21352 <+66>: mov DWORD PTR [esp+0x1c],eax

 0xb7f21356 <+70>: xchg ax,ax

 0xb7f21358 <+72>: mov eax,DWORD PTR [ebx+0x35fc] //여기서 argv[0]을 가져오는데???뭐야

 0xb7f2135e <+78>: mov eax,DWORD PTR [eax]

 0xb7f21360 <+80>: mov DWORD PTR [esp+0x8],ebp // "stack smashing detected" 문자열

 0xb7f21364 <+84>: mov DWORD PTR [esp+0x4],edi //서식문자열 "\*\*\* %s \*\*\*: %s terminated\n"

 0xb7f21368 <+88>: mov DWORD PTR [esp],esi //첫번째 인자(do\_abort)

 0xb7f2136b <+91>: test eax,eax

 0xb7f2136d <+93>: cmove eax,DWORD PTR [esp+0x1c]

 0xb7f21372 <+98>: mov DWORD PTR [esp+0xc],eax //마지막 인자(argv[0])

 0xb7f21376 <+102>: call 0xb7e8e700 // 출력하고 abort시킴

 0xb7f2137b <+107>: jmp 0xb7f21358 <\_\_fortify\_fail+72>

그럼 argv[0]을 가져오는 libc영역의 \*\*argv 는 상대적인 주소로 불려오기 때문에, 어느 루틴에서 그 값을 셋팅하는지 한번 찾아 보았다.

일단 argv[0]을 참조하는 부분에 bp를 걸고 실행시켜보면,





값을 가져올 당시 ebx값이 3번째 libc 맵핑 역역의 시작주소임을 알 수 있고, 그 주소에서 0x35fc만큼 떨어진곳에 \*\*argv가 위치함을 알 수 있다.

그럼 이 값은 현재 프로그램에서 셋팅되는 것일까? 한번 entry point에 bp를 걸고 실행시켜 보았다.



이런, 프로그램 시작과 동시에 셋팅되어진 값이였다.

한번 libc소스에서 확인해보자.



extern으로 \_\_libc\_argv가 선언되어져 있엇고, 이 값이 어디서부터 오는지 찾아보았더니,



다음과 같은 startup code에서 init을 해 준다고 한다.

그럼 프로그램 내에서 저 부분의 위치를 알아낼 방법은 없는 거 같다.

dl\_runtime\_resolve같은 코드를 분석하면 libc내의 영역을 알아낼 수 있으니 그것과 같이 사용할 수도 있을 것 같다.

<http://asm.sourceforge.net/articles/startup.html>

startup 부분 찾다가 나온건데, elf가 로딩되기까지의 부분을 잘 보여준다.

3. main함수 이전의 stack frame에서 찌꺼기 활용하기.



main함수 이전의 libc\_start\_main에서 저런식으로 항상 libc의 base addr가 남더라. 저것을 pop gadget과 add가젯을 활용해서 \*\*\_\_libc\_argv의 주소를 구할 수 있을 것 같다.

저 주소는 \_libc\_start\_main에서 내부호출함수 \_setjmp가 남기는 값이다.