|  |  |  |  |
| --- | --- | --- | --- |
| LOB GATE | | | |
| COLONY | 동아대학교 | 김영진 | 2017.01.04 |
| LOB gate 문제풀이 시작하겠습니다.    먼저 원래 있던 gremlin, gremlin.c 를 제 폴더를 하나 만들어 옴겼습니다.  옴기는 과정에서 uid gid 가 gate로 변경 됬습니다.  공격할 gremlin.c 의 소스를 확인해 봅시다.    Strcpy(buffer, argv[1]); 에서 버퍼가 받아들이는 argv의 크기가 정해지지 않아 buffer over flow가 발생합니다.  Argc = 취약한 소스코드 gremlin.c가 컴파일되어 실행되는 프로그램의 인자 개수 (안정해짐!)  Argv[0] = 실행 파일의 이름  Argv[1] = 실행 파일의 첫번재 인자  Argv[2] = 실행 파일의 두번째 인자  소스코드를 보면 [ buffer(256)] + [ ebp(4) ] + [ ret(4) ] 로 되어있을 것으로 예상이 된다.  gdb로 어셈블리어가 어떤 식으로 되어있는지 확인 해 봅시다.    인텔 방식으로 전환해서 다시 디스어셈블 한다. (다들 이방식을 사용 하시는 것 같아서) | | | |
| 인텔방식 어셈블리어 표기법  [명령어] [목적지] [출발지]  🡨-------------------  0x8048430 <main>: push %ebp -> ebp값을 스택에 넣는다.  0x8048431 <main+1>: mov %ebp,%esp -> esp값을 ebp에 넣는다  0x8048433 <main+3>: sub %esp,0x100 -> 0x100(16의2제곱=256)을 esp에서 뺀다  0x8048439 <main+9>: cmp DWORD PTR [%ebp+8],1 -> ebp+8주소에 있는 값(argc)와 1의 값을 비교  이유 : argc < 2  0x804843d <main+13>: jg 0x8048456 <main+38> -> 조건부 점프. 이유 : argc>=2인경우 | | | |
| 0x804843f <main+15>: push 0x80484e0 -> 주소값 밀어넣기  0x8048444 <main+20>: call 0x8048350 <printf> -> printf 함수가 있는 주소값 호출  0x8048449 <main+25>: add %esp,4 -> 4를 esp값에 더한다.  0x804844e <main+30>: call 0x8048360 <exit> -> exit 함수 주소값 호출  0x8048453 <main+35>: add %esp,4 -> 4를 esp값에 더한다  0x8048456 <main+38>: mov %eax,DWORD PTR [%ebp+12] -> ebp+12(argv포인터)주소를 eax에 저장  0x8048459 <main+41>: add %eax,4 -> 4를 eax값에 더함 (argv 배열 인자 주소)  0x804845c <main+44>: mov %edx,DWORD PTR [%eax] -> eax의 주소값을 edx에 넣음  0x804845e <main+46>: push %edx -> edx값을 밀어넣음.  0x804845f <main+47>: lea %eax,[%ebp-256] -> epb-256(buffer)주소를 eax에 저장  0x8048465 <main+53>: push %eax -> eax값을 저장  0x8048466 <main+54>: call 0x8048370 <strcpy> -> strcpy 함수 호출  0x804846b <main+59>: add %esp,8 -> 8을 esp의값에 더한다  0x804846e <main+62>: lea %eax,[%ebp-256] -> epb-256의 주소를 넣는다 (Eax = buffer의포인터)  0x8048474 <main+68>: push %eax -> eax의 주소 저장  0x8048475 <main+69>: push 0x80484ec -> 주소값 저장  0x804847a <main+74>: call 0x8048350 <printf> -> printf 호출  0x804847f <main+79>: add %esp,8 -> 8을 esp값에 더한다.  0x8048482 <main+82>: leave -> 지역변수 공간을 정리하고 스택을 마무리 한다.   * Main + 3 에서 버퍼가 256개 할당 되는 것을 확인 가능. * 따라서 [buffer256] + [ebp4] + [ret4]     Break를 메인함수의 시작과 종료지점에 걸어준다.  버퍼와 ebp메모리 값을 합한 260만큼 A를채우고 ret에 B를 채운다.    예상대로 Segmentation fault -> 버퍼 오버플로우가 일어 났다 똑 같은 그림이라 그림은 생략하지만  A만 256개를 넣었을땐 그냥 오류가 떳고  A만 260개를 넣었을땐 역시 Segmentation fault가 일어났다.  A260,B4개를 넣어도 역시 Segmentation fault가 일어났다.    Esp-256지점부터 100개의 메모리값을 확인.  0x41414141->0x42424242로 바뀌는 부분을 주목하자  41은 A의 아스키 코드이며 42는 B의 아스키 코드이다  Shellcode[28]+nop[232]+ret[0xbffff920]으로 페이로드를 짯다.    라고 뜨면서 core파일이 생성 되었다. (리턴 주소를 잘못 찾음)  코어파일은 프로그램이 종료되었을 때 운영체제가 디스크에 남기는 파일이며  코어파일을 분석해 gdb에서 buffer의 시작주소에 적재된 쉘코드를 실행 시킬 수 있다.  다시 찾아보았다.    A가 정확히 0xbffff918 부터 시작한다.  쉘코드를 바꿔서 쉨코드[25]+놉[235]+ret를 해보았다.    bash쉘을 땃다.  계정을 확인해보니 그대로 gate 이다.  복사한 파일을 익스플로잇 하면 복사한 파일의 setuid(gate)권한을 얻는다    따라서 원본파일을 위 방식으로 다시 진행하였다.  (원본파일과 복사본 파일의 메모리 주소는 같다)[\*특별한 보호 기법이 없다면] | | | |