|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | |  | | KITRI 모의해킹 28기 | | | |
|  | | |  | |  | | | |
|  | | |  | |  | | | |
|  | | |  | |  | | | |
|  | | |  | |  | | | |
|  | | |  | |  | | | |
|  | | |  | |  | | | |
|  | | |  | |  | | | |
|  | | |  | 작성: 보안팀 | | |  | |
|  | stack buffer overflow(primitive attack) | | | | | | |  |
|  | | 기본 공격 원리 및 이해 (진행os --> redhat9) | | | |  | | |
|  | |  | | | |  | | |
|  | | |  | |  | | | |
|  | | |  | |  | | | |
|  | | |  | |  | | | |

|  |
| --- |
| 1. 개요(공격개념/원리/취약점/공격 시 예상피해 등) |
| 가장 원시적인 오버플로우 공격인 스택 버퍼 오버플로우에 대하여 알아볼 것이다. |
| 1) 우선, 어셈블리어의 기초를 알아야 한다. 프로세서에 명령을 내리기 위한 명령어를 기계어라고 한다. 기계어와 1:1대응을 하는 언어가 어셈블리어이다. 어셈블리어를 알아야 메모리에서 함수가 스택, 힙 등의 자료구조를 통하여 운용되는 것을 이해하는 것이 가능하다. |
|  |
| 위의 기본적인 명령어들을 가지고, 메모리에서 c언어 등으로 구성된 함수가 어떤식으로 배치되고, 동작하는지 확인이 가능하다.(함수의 변수, 논리적 구성 등이 위의 어셈블리어로 대체된다.) 따라서, 위와 같은 기본적인 명령어들은 암기를 해야 한다. |
| 2) c언어 등의 사용자 언어로 작성된 소스코드를 어셈블리어로 변환하여 디버깅하려면, 디버깅 툴 gdb를 알아야 한다. |
| gdb툴의 사용법이다. |
| 3) c로 작성된 간단한 소스코드를 어셈블리어로 변환하여 구조를 분석해보자. 아래와 같이 코드를 작성하였다.  #include <stdio.h>  int main(){  int a=1;  return 0;  } --> 이코드를 컴파일 하여 나온 파일을 gdb를 사용하여 디버깅해보았다. disas main명령어로 해당 코드를  디스어셈블 한다. |
| 디스어셈블한 결과를 보고 어셈블리어와 소스코드를 비교분석 |
| main +3의 sub esp,0x10으로 스택에 0x10만큼의 공간을 할당하고 나서, main +6의 mov명령어로 스택의 한 공간인 ebp-0x4에 1의 값을 넣어주고 있다. main함수가 끝이 나고 0을 반환해주면, 프로세스 안에서 main함수가 종료된 것을 의미한다. 이것이 main +13으로 나타난 것이다.  <+3>라인을 보면, esp, 0x10으로 스택에 공간을 16만큼 할당받은 것을 알 수 있다. 보통 스택에서 ret의 위치는 esp의 위치에서 ebp의 값을 더한 위치인데 ebp의 값이 보통 4정도 된다. |
| 결과적으로 위의 함수에서 ret의 주소값은 20이라고 유추할 수 있다. 이 주소값을 공격자가 실행시키고 싶은 함수의 주소로 변조시키는 것이 스택 오버플로우 공격의 핵심이다. |
|  |
|  |
|  |

|  |
| --- |
|  |
| 2. 공격 결과(해당 취약점은 centos 7, 8 에서는 불가 / redhat9에서 테스팅) |
| 0) 모든 공격의 사전에는 사용되는 언어를 kor언어로 바꾸어주는 작업을 실행해준다.  export LANG="ko\_KR.euckr" 이렇게 언어 설정을 변경해준다.  주소값을 입력하기 전에는 반드시 언어설정을 변경해주어야 한다.  echo $LANG으로 언어설정을 확인하는 것이 가능하다.    1) 아래와 같은 소스코드를 작성하고, 컴파일을 해주도록 한다. (gcc -o bof bof.c) |
| ret주소를 func()함수의 주소로 덮어쓰기 하는것이 목표이다. |
| 2) gdb툴을 이용하여 bof파일을 분석해보자. (gdb bof하여 디버깅 진입 후 disas main) |
| <main+3>을 보면 , 0x78만큼 공간을 배치받는 것을 알 수 있다. 16진수 이므로 120바이트의 크기이다.  ret주소는 124바이트로 유추가 가능하다. |
| 3) info fu 명령어로 소스코드 상의 func()함수 주소가 무엇인지 알아내보자. |
| func()함수의 주소는 0x0804835c인 것을 확인하였다. |
| 4) ./bof `perl -e 'print "A"x124, "\x5c\x83\x04\x08"'` 이 명령어로 ret전까지의 공간을 A라는 쓰레기값으로 채워주고, ret공간에 func()함수의 주소를 덮어씌워준다. 하단은 공격 결과이다. |
| func()함수가 실행되었다. |
| 5) 또다른 공격을 진행해보자. 이번에는 강제로 루트권한의 shell을 가져오는 공격이다.(백도어 기반 오버플로우) 아래와 같은 소스코드를 우선 작성하자.(그 후 컴파일 한다. --> gcc -o stacksuid stacksuid.c) |
| ret주소를 shell\_code()주소로 덮어씌우기 하는 것이 목표이다. |
| 6) gdb툴을 이용하여 bof파일을 분석해보자. (gdb stacksuid 하여 디버깅 진입 후 disas main) |
| <main+3>을 보면 , 0x18만큼 공간을 배치받는 것을 알 수 있다. 16진수 이므로 24바이트의 크기이다. ret주소는 28바이트로 유추가 가능하다. |
| 7) info fu 명령어로 소스코드 상의 shell\_code()함수 주소가 무엇인지 알아내보자. |
| shell\_code()함수의 주소는 0x0804849c인 것을 확인하였다. |
| 8) ./stacksuid `perl -e 'print "A"x28, "\x9c\x84\x04\x08"'` 이 명령어로 ret전까지의 공간을 A라는 쓰레기값으로 채워주고, ret공간에 shell\_code()함수의 주소를 덮어씌워준다. 하단은 공격 결과이다. |
| 공격을 성공하여 결국에는 root shell을 탈취하였다. |
| 9) user9로 로그인한 상태에서도 root shell을 탈취하였다. |
|  |
|  |
|  |

|  |
| --- |
|  |
| 4. 보안진단 및 대책 |
| 1) ASLR기법을 적용해주도록 한다. ASLR기법은 프로그램이 실행될 시에 배치받는 메모리 주소를 항상 랜덤하게 배치하여, RET주소를 알아내기 힘들게 만드는 기법이다. 오버플로우 공격을 효과적으로 막는다. |
| 2) 스택 보호 기법을 사용해준다. 스택 자체에서 함수 등이 실행되는 것을 효과적으로 막는다. 스택 메모리에 실행권한을 해제함으로써, 오버플로우 공격을 효과적으로 막는다. |
|  |
| 3) 스택 가드, 스택 실드 를 사용해준다. 스택 가드는 ret주소와 메모리 공간 사이에 특정 값을 넣어주어서, 특정 값이 변할 시에 오버플로우 공격으로 판단하여 함수를 강제 종료해준다.  스택 실드는 ret주소를 global ret와 같은 특정 공간에 보관하여, ret주소의 변조를 원천적으로 막는다. |
|  |
|  |

|  |
| --- |
|  |
| 5. 보안대책 적용시 공격결과 (최종 매뉴얼에 포함) |
|  |
|  |
|  |
|  |
|  |
|  |