|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | |  | | KITRI 모의해킹 28기 | | | |
|  | | |  | |  | | | |
|  | | |  | |  | | | |
|  | | |  | |  | | | |
|  | | |  | |  | | | |
|  | | |  | |  | | | |
|  | | |  | |  | | | |
|  | | |  | |  | | | |
|  | | |  | 작성:보안팀 | | |  | |
|  | got overwrite attack | | | | | | |  |
|  | | got overwrite | | | |  | | |
|  | |  | | | |  | | |
|  | | |  | |  | | | |
|  | | |  | |  | | | |
|  | | |  | |  | | | |

|  |
| --- |
| 1. 개요(공격개념/원리/취약점/공격 시 예상피해 등) |
| got overwrite공격을 이해하려면 우선 소스코드가 작성되어 컴파일되고, 그 과정 중에서 링크라는 과정에 대하여 알아야 한다. 또한, plt와 got의 개념에 대하여 알야아 한다.  우선, 하단에 컴파일 및 링킹을 한눈에 볼 수 있는 그림을 첨부했다. |
|  |
| 우리가 작성한 소스코드는 전처리 과정과 컴파일 과정을 거쳐 컴퓨터가 이해할 수 있는 언어로 변경되어 obj파일이 된다. 해당 obj파일과 여러 라이브러리(함수들의 모음)들이 하나로 묶여서 실행가능한 파일이 되는 과정이 linking 과정이다. linking의 방법에는 두가지 방법이 있다. static linking과 dynamic linking이 있는데, 하나씩 설명해보겠다.  Static Linking은 프로그램 내부에 라이브러리를 추가하여 실행파일에 함께 묶어버리는 것이다. 이 경우, 라이브러리 자체가 프로그램 내부에 존재하기 때문에, 별도의 라이브러리 연동과정이나 의존성이 사라지는 장점이 있지만, 파일의 크기가 커지게 되고, 라이브러리를 항상 메모리에 올려야 하기 때문에 리소스 사용량이 커지게 된다.  Dynamic Linking은 Static Linking과는 다르게, 공유라이브러리 라는 것을 이용한다. 라이브러리를 하나의 메모리 공간에 올려 놓고, 해당 라이브러리를 사용하는 모든 프로그램이 공유를 하여 사용을 하는 것이다. 당연히 Static Linking 방식에 비해서 파일의 크기가 작아지게 되고, 라이브러리 내용을 메모리에 올릴 필요가 없으므로, 리소스 사용량도 줄일 수 있지만, 의존성을 지니게 된다. |
| 쉽게 이해해 보자면, static linking방식은 소스코드 자체에 헤더파일들을 넣어주어 함께 묶여서 컴파일 및 링킹이 된다고 생각하면 되고, dynamic linking방식은 헤다파일들을 컴파일 및 링킹 과정에서 찾아가는 것이라고 생각하면된다. 여기에서 plt와 got개념이 사용되는 것이다. 우선 밑의 소스코드를 살펴보자.  #include <stdio.h>  int main()  {  puts("hello world !! #1\n");  return 0;  }  해당 소스에는 어디에도 puts에 대한 정의가 없다. 실행파일을 만들 때 puts의 코드가 있는 라이브러리를 Link를 통해 연결 했기 때문에, 우리는 별도의 puts에 대한 정의를 할 필요가 없는 것이다. 그런데 static linking의 경우, 해당 라이브러리를 실행파일에 넣기 때문에, 함수 실행 코드가 파일 내부에 존재하여 찾으면 되지만, Dynamic Linking의 경우, 모두가 참조 가능한 메모리에 올리게 된다. 이때 puts함수를 찾아가려면, 해당 함수의 주소가 필요하다. 이때 바로 plt와 got를 사용하여 해당 함수를 찾아가는 것이다.  우선, 이해를 돕기위해서 plt와 got의 정의를 살펴보자. 하단에 설명을 해보겠다.  <plt>  실제 호출 코드를 담고있는 테이블이다. 이 테이블의 내용을 참조해서 \_dl\_runtime\_resolve가 수행되고, 실제 시스템 라이브러리 호출이 이루어진다. 쉽게 이야기하여 외부 라이브러리를 연결해주는 함수이다. 실제 바이너리에서도 사용하고자 하는 함수, 라이브러리 주소 대신에 plt를 호출한다.  <got>  plt가 참조하는 테이블이다. plt가 어떠한 외부 프로시저를 호출할 경우에 got를 참조하여 해당 주소로 분기된다. plt함수들은 got라는 테이블의 주소로 분기한다. got는 외부 라이브러리의 함수, 변수의 주소를 저장한다.  위 소스코드에서 puts()함수의 호출을 예로 들겠다.  puts()함수가 처음 호출된다면, puts()함수 호출 -> plt 이동 -> got 참조 -> 다시 plt이동  -> \_dl\_runtime\_resolve 수행 -> got에 실제 함수의 주소 저장후에 실제 함수주소로 이동  puts()함수가 두번째 호출된다면, pust()함수 호출 -> plt 이동 -> got 참조 -> puts()함수로 점프  --------------------------------------------------------------------------------------------------------  <got overwrite 공격>  got overwrite공격이란 지금까지 설명한 plt, got를 이용한 공격기법이다. dynamic link방식으로 컴파일된 소스코드가 공유 라이브러리를 호출할 시에 plt와 got를 사용하는데, 여기서 plt는 got를 가리키고, got에 실제 함수의 주소가 들어있다. 이 got의 값을 공격자가 원하는 함수의 주소로 변조시킨다면, 해당 함수가 실행될 것이다. 이는 현존하는 플랫폼 상에서 거의 적용되는 buffer overflow 공격 기법이다. |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

|  |
| --- |
|  |
| 2. 공격 결과(해당 취약점은 centos 7, 8 에서도 현존) |
| 1) 우선적으로 코드분석, 메모리 분석, 스택구조 파악, 어셈블리어로 디컴파일, 파일 및 함수 위치 찾기 등을 하기 위하여  gdb-peda 를 설치해준다.   gdb-peda는 gdb의 확장버전으로 gdb에 비하여 좀 더 자세하게 파일을 디버깅할 수 있는 기능을 제공하고, 파일 및 함수의 위치를 쉽게 찾을 수 있는 것이 장점이다. 아래는 gdb-peda 설치 방법이다. |
| sudo yum install git  // git 다운로드  git clone https://github.com/longld/peda.git ~/peda  // github에서 peda.git 가져와서 현재디렉터리/peda에 복사  echo "source ~/peda/peda.py" >> ~./gdbinit  // 기존 gdb실행경로인 gdbinit에 위에서 /peda에 넣어둔 peda.py 덮어씌움.  echo "Done! debug your program with gdb and enjoy"  // gdb-peda 설치가 완료되었습니다!! |
| 2) got overwrite 수행하기 위하여 간단한 코드를 작성하겠다.(got\_bof.c) |
| #include <stdio.h>  int main(void)  {  char buf[20];  gets(buf);  puts(buf);  return 0;  } |
| **<공격 원리 설명>**  위의 코드는 사용자가 입력한 값을 그대로 화면에 뿌려주는 코드이다. 여기에서 puts함수의 got주소를 system함수의 주소로 바꾸어 준 후에 /bin/sh를 함수안에 넣어주면, /bin/sh 이 실행될 것이다. |
| 3) 위에 작성한 got\_bof.c파일을 gcc로 컴파일 하기 전에 오류를 잡아주기 위하여 하단의 라이브러리를 설치해준다. |
| yum install glibc-devel.i686 libgcc.i686 libstdc++-devel.i686  yum install libgcc.x86\_64 |
| 4) 그러고 나서 컴파일을 실시해준다.  gcc -m32 --no-pie -fno-pic -o got\_bof got\_bof.c  하단은 got\_bof 파일을 실행해준 모습이다. |
|  |
| 5) gdb-peda 도구를 이용하여 소스코드에서 puts함수의 plt주소를 알아낸다. |
| 사진을 보면 알 수 있듯이, +28 라인에서 보면 puts함수의 plt상 주소는 0x8048310이다. |
| 6) main함수에 breakpoint를 걸어주고 디버깅을 지속하여 puts함수의 got상 주소와 system함수의 주소를 알아낸다.  x/3i '해당 함수의 plt주소' --> 이 명령어로 함수의 분기지점을 분석할 수 있다. (got주소 등을 알아낸다.) |
| puts()함수의 got상 주소 --> 0x804a010 / system()함수의 주소 --> 0xf7e34fa0 |
| 7) puts()함수의 got상 주소를 system()함수의 주소로 바꿔치기 한다. --> 이것을 got overwrite라고 부른다. |
| 바꿔치기 하고 나서 , c를 입력하여 디버깅을 지속한다. 그 후에 /bin/sh를 입력한다. 그렇게 되면, system함수안에 /bin/sh가 인자로 들어가서 shell이 실행될 것이다. |
| 8) /bin/sh를 입력하자 의도한 대로 디버깅 중에 shell 을 따온 것을 확인 할 수 있다. 하단 결과 첨부 |
|  |
| 9) shell안에서 각종 명령어를 입력할 시에 정상시행 되는 것을 확인하였다. |
|  |
|  |
|  |

|  |
| --- |
|  |
| 4. 보안진단 및 대책 |
| 1) dynamic linking의 취약점을 이용한 방법이므로, 중요한 소스코드의 컴파일이나 바이너리 파일의 컴파일 시에 static linking을 이용한다. 허나 이 방법은 너무 많은 리소스의 낭비를 발생시킨다. |
| 2) 현존하는 강력한 공격으로, 스택이나 힙을 이용하는 것이 아니고, 실제 함수, 라이브러리의 주소를 변조시키는 공격으로 프로그래머 단에서의 보안이 필요하다. secure coding을 해야하고, gets()등 함수를 사용하는 것에 주의를 하여야 한다. 예를 들어서 주소를 변조하지 못하게 하는 등의 방법을 취한다. |
|  |
|  |
|  |
|  |

|  |
| --- |
|  |
| 5. 보안대책 적용시 공격결과 (최종 매뉴얼에 포함) |
| 해당 공격은 보안대책을 적용하여도, 프로그래머가 실수를 범할시에 충분히 다시 일어날 수 있는 공격이다. |
|  |
|  |
|  |
|  |
|  |