/* --*/

번 역: vangelis(vangelis@wowhacker.org) - wowcode of Wowhacker Team

Home: http://www.wowhacker.org, http://www.wowsecurity.net

P.S: 혹시라도 있을지도 모르는 번역상의 오류나 오타에 대한 문제는 원문을 참고하시길...

수정된 번역은 이 문서에 대한 공부를 끝낸 후 공개하도록 할 생각이다.

/* --*/

==Phrack Inc.==

Volume 0x0b, Issue 0x3c, Phile #0x0a of 0x10

|=--------------------=[Basic Integer Overflows]=----------------------=|

|=--=|

|=-------------------=[by blexim <blexim@hush.com>]=-------------------=|

--[1.0 도입

이 글에서 나는 악의적인 사용자가 영향을 받은 프로세스의 실행 경로를 수정할 수 있도록 가끔 허용할 수 있는 두 가지 프로그래밍 버그에 대해 기술할 것이다. 이 두 가지 버그는 변수들이 예상치 않은 값을 포함하도록 하는 것에 의해 발생하며, 이 두 가지 버그는 buffer overflow 또는 format string 과 같이 메모리를 덮어쓰는 것과 같이 직접적이지는 않다. 이 글에서 제시된 모든 예는 C 로 되어 있으며, 그래서 C 에 대한 기본적인 지식이 필요하다. Integer 가 메모리에 어떻게 저장되는가에 대한 지식 또한 유용할 것이지만 필수적이지는 않다.

----[1.1 정수란 무엇인가?

전산 개념에서 보면 정수란 분수 부분이 없는 실제 수를 제시할 수 있는 변수이다. 정수는 전형적으로 그것들이 컴파일되는(예를 들어, i386 과 같은 32 비트 시스템에서 정수는 32 비트의 길이며, SPARC 와 같은 64 비트 시스템에서 정수는 64 비트 길이다) 시스템에서 포인터와 같은 크기다. 어떤 컴파일러들은 같은 크기의 정수와 포인트를 사용하지는 않는다. 그래서 단순성을 위해 이 글에서 제시된 모든 예들은 32 비트 정수, long, 포인터를 가진 32 비트 시스템을 지칭한다.

모든 변수와 마찬가지로 정수는 메모리의 영역이다. 우리가 정수에 대해 말할 때 우리는 보통 그것을 십진수로 나타내며, 그것이 대부분의 사람들이 사용하는 숫자 체계이기 때문이다. 컴퓨터는 십진수를 처리할 수 없기 때문에 컴퓨터 내부에서는 정수는 이진수로 저장된다. 이진수는 십진수에서는 열 개의 수가 사용되는 것과는 달리 1 과 0 두 수를 사용하는 숫자 체계이다. 2 진수와 10 진수뿐만 아니라 16 진수도 전산 작업에서 종종 사용되는데, 그것은 2 진수와 16 진수 사이를 전환하는 것이 쉽기 때문이다.

음수를 저장하는 것이 종종 필요하기 때문에 단지 2 진수를 사용하는 음수를 제시하는 메커니즘이 필요하다. 이것을 충족시킬 방법은 sign 을 결정하기 위해 어떤 변수의 가장 중요한 비트(MSB)를 사용하는 것이다. 만약 MSB 가 1 로 설정되어 있다면 변수는 음수로 해석되고, 0 으로 설정되면 변수는 양수이다. 이것은 signedness bug 부분에서 설명되는 것처럼 혼란스러울 수 있는데, 왜냐하면 모든 변수가 sign 화 되지 않기 때문이며, 그 변수들이 양수인지 음수인지를 결정하기 위해 모두 MSB 를 사용하지 않는다는 것을 의미한다. 이 변수들은 unsigned 로 알려져 있으며, 단지 양수 값으로 지정될 수 있으며, 반면 양수 또는 음수일 수 있는 변수들은 unsigned 라고 불린다.

----[1.2 integer overflow 란 무엇인가?

정수가 고정된 크기(이 글에서는 32 비트)이기 때문에 그것이 저장할 수 있는 고정된 최대값이 있다. 이 최대값보다 더 큰 값을 저장하려는 시도를 할 때 이것을 integer overflow 라고 한다. ISO C99 표준은 integer overflow 는 "undefined behaviour"(정의되지 않은 행위)를 야기시키며, 이것은 ISO C99 표준에

맞는 컴파일러들이 오버플로우를 완전히 무시하는 것으로부터 프로그램을 중지시키는 것까지 그들이 좋아하는 것이면 어떤 것이든 할 수 있다는 것을 의미한다. 대부분의 컴파일러들은 오버플로우를 무시하는 것처럼 보이며, 이것은 예상치 못한 결과나 또는 잘못된 결과가 저장되는 결과를 초래한다.

----[1.3 왜 위험한가?

Integer overflow 는 그것이 발생한 후에는 탐지될 수 없으며, 그래서 어떤 어플리케이션이 사전에 계산한 어떤 결과가 사실 정확한지 말할 방법이 없다. 그 계산이 어떤 버퍼의 크기와 관련되어 있거나 또는 얼마나 멀리 어떤 배열을 인덱싱할지에 관련되어 있다면 이것은 위험해질 수 있다. 물론 대부분의 integer overflow 가 공격 가능한 것은 아닌데, 이것은 메모리가 직접적으로 덮어 쓰이지 않기 때문이다. 하지만 가끔 다른 종류의 버그로 이어질 수 있으며, 주로 버퍼 오버플로우로 이어진다. 이것뿐만 아니라 integer overflow 는 탐지하기가 어려운데, 그래서 심지어 잘 결산된 코드도 놀라움을 발생시킬 수 있다.

--[2.0 Integer overflows

그래서 integer overflow 가 발생할 때 어떤 일이 일어나는가? ISO C99 에는 다음과 같은 부분이 있다.

> "unsigned 오퍼랜드를 포함하고 있는 컴퓨터 조작은 결코 오버플로우될 수 없는데, 왜냐하면 결과적으로 나오는 unsigned 정수 타입에 의해 나타낼 수 없는 결과는 결과적으로 나오는 타입에 의해 제시될 수 있는 가장 큰 값보다 하나 더 큰 숫자를 법(modulo)으로 축약된다. ”
>
> “ A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type."

NB: 수를 법으로 하는 것은 두 개의 수를 나누고, 나머지를 가지는 것을 포함한다.

예를 들어,

10 modulo 5 = 0
11 modulo 5 = 1

그래서 큰 값을 법(modulo)으로 하는 것을 줄이는 것은(MAXINT + 1) 정수에 맞을 수 없는 값의 부분을 버리고, 나머지를 유지하는 것으로 보일 수 있다. C 에서 연산자를 법으로 하는 것은 % sign 이다.
</NB>

이것은 몇 마디의 말일뿐이므로, 한 가지 예를 들면 전형적인 "정의되지 않은 행위"를 더 잘 예시할 수 있을 것이다:

두 개의 unsigned 정수 a 와 b 가 있으며, 둘 다 32 비트의 길이이다. 우리는 32 비트가 가질 수 있는 최대값을 부여하고, b 에 1 을 부여한다. 우리는 a 와 b 를 함께 더하고, 그 결과를 제 3 의 unsigned 32 비트 정수 r 에 저장한다.

```
a = 0xffffffff
b = 0x1
r = a + b
```

자, a 와 b 를 더한 것의 결과가 32 비트를 이용해 제시할 수 없기 때문에 ISO 표준에 의해 그 결과는 법(modulo) 0x100000000 로 축약되었다.

```
r = (0xffffffff + 0x1) % 0x100000000
r = (0x100000000) % 0x100000000 = 0
```

modulo 수를 사용한 결과를 줄이는 것은 기본적으로 단지 가장 낮은 32 비트의 결과가 사용된다는 것을 확인시키며, 그래서 integer overflow 는 그 결과를 변수에 의해 제시될 수 있는 어떤 크기로 줄여지도록 야기시킨다. 이것은 결과가 0 으로 감싸여 있는 것처럼 보이기 때문에 종종 "wrap around"라고 부른다.

----[2.1 Widthness 오버플로우

그래서 integer overflow 는 가지기에는 너무 작은 어떤 변수에 어떤 값을 저장하고자 시도하는 것의 결과이다. 이것의 가장 간단한 예가 더 작은 변수에 큰 변수의 내용을 단순히 부여하는 것에 의해 예시될 수 있다:

```
/* ex1.c - loss of precision */
#include <stdio.h>

int main(void){
        int l;
        short s;
        char c;
```

```
        l = 0xdeadbeef;
        s = l;
        c = l;

        printf("l = 0x%x (%d bits)\n", l, sizeof(l) * 8);
        printf("s = 0x%x (%d bits)\n", s, sizeof(s) * 8);
        printf("c = 0x%x (%d bits)\n", c, sizeof(c) * 8);

        return 0;
}
/* EOF */
```

위의 소스를 컴파일하여 실행한 것의 결과는 다음과 같다:

```
nova:signed {48} ./ex1
l = 0xdeadbeef (32 bits)
s = 0xffffbeef (16 bits)
c = 0xffffffef (8 bits)
```

각 할당 값이 각 타입에 저장될 수 있는 값의 경계가 초과되도록 야기시키기 때문에 그 값은 줄어들어 할당된 변수에 맞을 수 있다.

여기서 정수 프로모션을 언급할 가치가 있다. 다른 크기의 오퍼랜드를 포함하고 있는 어떤 계산이 수행될 때 더 작은 오퍼랜드는 더 큰 오퍼랜드의 크기로 "진척"된다. 그 계산은 그러면 이 진척된 크기로 수행되며, 그리고 만약 그 결과가 더 작은 변수에 저장된다면 그 결과는 다시 더 작은 크기로 줄어든다. 예를 들어보자.

```
int i;
short s;

s = i;
```

계산은 여기서 다른 크기화된 오퍼랜드로 수행되고 있다. 여기서 발생한 것은 변수가 int(32 비트 long)로 진척되고, 그런 다음 i 의 내용이 새로 진척된 s 로 복사된다. 이 다음 진척된 변수의 내용들은 s 에 저장되기 위해 16 비트로 다시 격하(demote)된다. 만약 격하가 s 가 가질 수 있는 최대값보다 더 크다면 이 격하는 결과를 줄여지도록 야기시킬 수 있다.

------[2.1.1 Exploiting

Integer overflow 는 대부분의 버그 종류와는 다르다. 이것은 직접적으로 메모리를 덮어쓰거나 실행 흐름을 직접적으로 통제하지는 않지만 훨씬 더 미묘하다. 문제의 근원은 어떤 프로세스가 연산의 결과를 확인할 방법이 없으며, 그래서 저장된 결과와 정확한 결과 사이에는 차이가 있을 수도 있다는 사실에 있다. 이것 때문에 대부분의 integer overflow 는 정확하게 공격할 수 있는 것은 아니다. 그럼에도 불구하고, 어떤 경우에는 중요한 변수를 잘못된 값을 담도록 하는 것이 가능하며, 그리고 이것은 나중에 코드의 문제로 이어질 수 있도록 할 수 있다.

이 버그의 미묘함 때문에 이 버그가 공격될 수 있는 아주 많은 상황들이 있으며, 그래서 나는 모든 공격 가능한 상황들을 제시하지는 않을 것이다. 대신 나는 공격 가능한 몇 가지 상황의 예들을 제시할 것이다. 이것을 활용해 스스로 연구하길 희망한다.

예 1:

```
/* width1.c - 사소한 widthness bug 공격하기 */
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]){
        unsigned short s;
        int i;
        char buf[80];

        if(argc < 3){
                return -1;
        }

        i = atoi(argv[1]);
        s = i;

        if(s >= 80){            /* [w1] */
                printf("Oh no you don't!\n");
                return -1;
        }

        printf("s = %d\n", s);
```

```
        memcpy(buf, argv[2], i);
        buf[i] = '\0';
        printf("%s\n", buf);

        return 0;
}
```

이와 같은 것은 실제 코드에서는 아마 등장하지 않을 것이며, 이것은 하나의 예로서는 좋을 것이다. 다음 입력 값을 보아라:

```
nova:signed {100} ./width1 5 hello
s = 5
hello
nova:signed {101} ./width1 80 hello
Oh no you don't!
nova:signed {102} ./width1 65536 hello
s = 0
Segmentation fault (core dumped)
```

길이 인수는 명령 라인으로부터 입력 받아, 정수 i 에 넘겨진다.이 값이 short 정수 s 에 옮겨질 때 만약 그 값이 s 에 맞기에는 너무 크다면(예를 들어, 만약 값이 65535 보다 더 크다면) 줄어들게 된다. 이것 때문에 [w1]에서 바운드 체크를 우회해서 버퍼를 오버플로우시키는 것이 가능하다. 이런 후 표준적인 stack smashing 기법이 프로세스를 공격하기 위해 사용될 수 있다.

----[2.2 Arithmetic overflows

섹션 2.0 에서 보았듯이, 만약 정수가 가질 수 있는 최대값보다 더 큰 값을 정수에 저장하고자 시도하면 그 값은 줄어들게 될 것이다. 만약 저장된 값이 수 연산의 결과라면 나중에 그 결과를 사용할 프로그램의 어떤 부분도 연산의 결과가 정확하지 않기 때문에 정확하지 않게 실행될 것이다. 앞에서 제시한 wrap around 를 보여주는 다음 예를 살펴보자.

```
/* ex2.c - an integer overflow */
#include <stdio.h>

int main(void){
```

```
        unsigned int num = 0xffffffff;

        printf("num is %d bits long\n", sizeof(num) * 8);
        printf("num = 0x%x\n", num);
        printf("num + 1 = 0x%x\n", num + 1);

        return 0;
}
/* EOF */
```

이 프로그램을 실행한 것의 출력은 다음과 같다:

```
nova:signed {4} ./ex2
num is 32 bits long
num = 0xffffffff
num + 1 = 0x0
```

Note:
눈치 빠른 독자라면 0xffffffff 가 십진수 -1 이라는 것을 눈치챘을 것이며, 그래서 우리가 단지 다음을 하고 있는 것처럼 보일 것이다.

```
1 + (-1) = 0
```

이것이 어떤 일이 일어나고 있는지를 살펴보는 한 방법이지만 변수 num 이 unsigned 이기 때문에 혼동을 초래할 수도 있고, 그래서 그것에 대해 이루어진 모든 연산(arithmetic)은 unsigned 이 될 것이다. 그것이 발생하는 것처럼 많은 signed 연산은 다음 예증(오퍼랜드 둘 다 32 비트 변수라고 가정함)처럼 integer overflow 에 의존한다:

```
-700       + 800   = 100
0xfffffd44 + 0x320 = 0x100000064
```

더하기의 결과가 변수의 범위를 초과하기 때문에 가장 낮은 32 비트가 결과로서 사용되고 있다. 이 낮은 32 비트는 0x64 이며, 그것은 십진수 100 과 동일하다.
</note>

정수가 디폴트로 signed 이므로, integer overflow 는 다음 코드에 흥미로운 영향력을 가질 수 있는 signedness 에 변화를 초래할 수 있다. 다음 예를 살펴보자:

```
/* ex3.c - change of signedness */
#include <stdio.h>

int main(void){
        int l;

        l = 0x7fffffff;

        printf("l = %d (0x%x)\n", l, l);
        printf("l + 1 = %d (0x%x)\n", l + 1 , l + 1);

        return 0;
}
/* EOF */
```

이것의 출력은 다음과 같다:

```
nova:signed {38} ./ex3
l = 2147483647 (0x7fffffff)
l + 1 = -2147483648 (0x80000000)
```

여기서 정수는 signed long 정수가 가질 수 있는 가장 높은 양수값으로 초기화되어 있다. 그것이 증가할 때 signedness 를 가리키는 MSB 가 설정되고, 그 정수는 음수로 해석된다.

더하기가 integer 가 overflow 되도록 야기시킬 수 있는 유일한 연산은 아니다. 변수의 값을 변경하는 거의 어떤 연산도 다음 예에서 볼 수 있듯이 오버플로우를 야기할 수 있다.

```
/* ex4.c - various arithmetic overflows */
#include <stdio.h>

int main(void){
        int l, x;

        l = 0x40000000;

        printf("l = %d (0x%x)\n", l, l);
```

```
        x = l + 0xc0000000;
        printf("l + 0xc0000000 = %d (0x%x)\n", x, x);

        x = l * 0x4;
        printf("l * 0x4 = %d (0x%x)\n", x, x);

        x = l - 0xffffffff;
        printf("l - 0xffffffff = %d (0x%x)\n", x, x);

        return 0;
}
/* EOF */
```

```
Output:

    nova:signed {55} ./ex4
    l = 1073741824 (0x40000000)
    l + 0xc0000000 = 0 (0x0)
    l * 0x4 = 0 (0x0)
    l - 0xffffffff = 1073741825 (0x40000001)
```

더하기는 첫 번째 예에서처럼 정확하게 같은 방식으로 오버플로우를 야기시키고 있으며, 비록 다르게 보일 수는 있지만 곱셈도 마찬가지다. 두 경우에 연산의 결과는 정수에 맞기에는 너무 크며, 위에서 기술되어 있는 것처럼 줄어든다. 빼기는 약간 다른데, overflow 보다는 underflow 를 초래하는데, 정수가 가질 수 있는 최소값보다 더 낮은 값을 저장하고자 할 때 wrap around 를 초래한다. 이와 같이 우리는 더하기를 빼기로, 곱하기를 나누기로, 또는 빼기를 더하기로 강제로 전환할 수 있다.

```
------[ 2.2.1 Exploiting
```

가장 일반적인 arithmetic overflow 가 공격 당할 수 있는 가장 일반적인 방법들 중의 하나는 어느 정도 크기로 버퍼가 할당되어야 하는지를 계산하는 것이다. 종종 어떤 프로그램은 오브젝트들의 어떤 배열을 위한 공간을 할당해야 하는데, 그래서 그것은 공간을 보존하고, 어떤 오브젝트의 크기에 의한 요소들의 수를 곱함으로써 얼마나 많은 공간이 필요한지 계산하기 위해 malloc(3) 또는 calloc(3) 루틴을 사용한다. 앞에서 본 것처럼 만약 우리가 이 오퍼랜드(요소들의 수 또는 오브젝트 크기)들의 어느 것을 통제할 수 있다면 우리는 다음 코드 부분이 보여주는 것처럼 버퍼의 크기를 잘못 계산할 수 있다:

```
int myfunction(int *array, int len){
    int *myarray, i;

    myarray = malloc(len * sizeof(int));    /* [1] */
    if(myarray == NULL){
        return -1;
    }

    for(i = 0; i < len; i++){              /* [2] */
        myarray[i] = array[i];
    }

    return myarray;
}
```

이 예에 나오는 함수 myfunction()은 len 파라미터를 점검하는 것이 부족하기 때문에 시스템의 다운을 초래할 수 있다. [1]에서 곱하기는 len 에 대한 아주 충분한 값을 제공함으로써 오버플로우될 수 있다. 그래서 우리는 버퍼가 우리가 선택한 어떤 크기가 되도록 할 수 있다. len 에 대한 적절한 값을 선택함으로써 우리는 [2]에서 루프로 하여금 myarray 버퍼의 끝을 지나 쓰게 할 수 있으며, 이것은 heap overflow 로 이어진다. 이것은 malloc 통제 구조를 덮어씀으로서 어떤 구현에서는 임의의 코드를 실행하는 효과가 나올 수 있지만 이것은 이 글의 범위를 벗어나는 것이다.

다른 예:

```
int catvars(char *buf1, char *buf2, unsigned int len1,
            unsigned int len2){
    char mybuf[256];

    if((len1 + len2) > 256){    /* [3] */
        return -1;
    }

    memcpy(mybuf, buf1, len1);      /* [4] */
    memcpy(mybuf + len1, buf2, len2);

    do_some_stuff(mybuf);
```

```
        return 0;
    }
```

이 예에서, [3]에서 체크는 len1 과 len2 에 대해 더하기는 오버플로우로 그리고 낮은 수로 wrap around 를 초래할 수 있는 적절한 값을 사용함으로써 우회될 수 있다. 예를 들어, 다음 값을 보자:

```
    len1 = 0x104
    len2 = 0xfffffffc
```

함께 더해지면 0x100(십진수 256)라는 결과로 wrap around 되는 결과를 초래한다. 이것은 [3]에서 체크를 우회하고, 그런 다음 [4]에서 memcpy(3)는 데이터를 복사하고 버퍼의 끝을 지나간다.

--[3 Signedness 버그

Signedness bug 는 unsigned 변수가 signed 변수로 해석되거나 signed 변수가 unsigned 변수로 해석될 때 발생한다. 이런 형태의 행위는 컴퓨터 내부에서 signed 와 unsigned 변수가 저장되는 방법 사이에 구분이 없기 때문에 발생할 수 있다. 최근에, 몇몇 signedness 버그가 FreeBSD 와 OpenBSD 커널에서 등장하고 있어, 우리가 쉽게 이용할 수 있는 많은 예들이 있다.

----[3.1 Signedness 버그는 어떻게 생겼는가?

Signedness 버그는 다양한 형태를 띌 수 있지만, 우리가 살펴볼 것은 다음과 같다:
* 비교에서 사용되는 signed 정수
* arithmetic 에서 사용되는 signed 정수
* signed 정수에 비교되는 unsigned 정수

다음은 고전적인 signedness bug 의 예이다:

```
    int copy_something(char *buf, int len){
        char kbuf[800];

        if(len > sizeof(kbuf)){         /* [1] */
            return -1;
        }

        return memcpy(kbuf, buf, len);  /* [2] */
    }
```

여기서 문제는 memcpy 가 len 의 파라미터로 unsigned int 를 취한다는 것이지만 signed 정수를 사용하여 memcpy 가 실행되기 전에 바운드 체킹이 이루어진다. Len 에 대한 음수 값을 전달함으로서 [1]에서 체킹을 건네주는 것이 가능하지만 그 다음으로 호출에서 [2]에서 memecpy 하기 위해 len 은 큰 unsigned 값으로 해석될 것이며, 따라서 메모리가 버퍼 kbuf 의 끝을 지나 덮어쓰여지도록 한다.

signed/unsigned 혼동에서 나올 수 있는 또 다른 하나의 문제는 연산이 실행될 때 발생한다. 다음 예를 보자:

```
int table[800];

int insert_in_table(int val, int pos){
    if(pos > sizeof(table) / sizeof(int)){
        return -1;
    }

    table[pos] = val;

    return 0;
}
```

라인 table[pos] = val;은 *(table + (pos * sizeof(int))) = val;와 동일하기 때문에 우리는 여기서 문제는 코드가 더하기를 위해 음수 오퍼랜드를 예상하지 않는다는 것을 알 수 있다. 코드는 (table + pos)가 table 보다 더 크기를 예상한다. 그래서 pos 에 음수 값을 제공하는 것은 프로그램이 그것을 예상치 못하고, 그래서 처리하지 못하는 상황을 초래한다.

------[3.1.1 Exploiting

이런 종류의 버그는 공격하기에 문제가 될 수도 있는데,이것은 signed 정수가 unsigned 로 해석될 때 거대해지는 경향이 있다는 사실 때문이다. 예를 들어, 16 진수로 제시될 때 -1 은 0xffffffff 이다. Unsigned 로 해석될 때 이것은 정수로 나타내는 것이 가능한 가장 큰 값(4,294,967,295)이 되어 만약 이 값이 예를 들어 len 의 파라미터로 memcpy 에 전달되면 memcpy 는 4GB 의 데이터를 목적지 버퍼에 복사하려고 시도할 것이다. 분명 이것은 세그먼트 폴트를 야기하거나, 만약 그렇지 않다면 스택이나 힙에 있는 데이터 중 아주 많은 양을 쓰레기로 만들어버릴 것이다. 가끔 source address 에 아주 낮은 값을 전달함으로써 이 문제를 해결하는 것이 가능하지만 항상 가능한 것은 아니다.

----[3.2 integer oveflow 에 의해 야기되는 Signedness bug

가끔, 정수를 덮어쓰는 것이 가능해 음수를 wrap around 한다. 어플리케이션이 그와 같은 값을 예상할 것 같지는 않기 때문에 앞에서 기술된 대로 signedness 버그를 공략하는 것이 가능할 수 있다.

이런 형태의 버그의 예는 다음과 같다:

```
    int get_two_vars(int sock, char *out, int len){
        char buf1[512], buf2[512];
        unsigned int size1, size2;
        int size;

        if(recv(sock, buf1, sizeof(buf1), 0) < 0){
            return -1;
        }
        if(recv(sock, buf2, sizeof(buf2), 0) < 0){
            return -1;
        }

        /* packet begins with length information */
        memcpy(&size1, buf1, sizeof(int));
        memcpy(&size2, buf2, sizeof(int));

        size = size1 + size2;       /* [1] */

        if(size > len){             /* [2] */
            return -1;
        }

        memcpy(out, buf1, size1);
        memcpy(out + size1, buf2, size2);

        return size;
    }
```

이 예는 네트워크 데들에서 가끔 발생할 수 있는 것을 보여주는데, 특히 길이 정보가 패킷의 일부로

전달될 때가 그렇다(즉, 신뢰 받지 못한 유저에 의해 제공될 때). [1]에서 더하기는 데이터가 출력 버퍼의 바운드를 초과하지는 않는지 점검하는데 사용되는데, size1 과 size2 를 크기 변수로 하여금 음수로 wrap around 하도록 야기할 값으로 설정하는 것에 의해 악용될 수 있다. 다음 예가 이런 경우다.

```
size1 = 0x7fffffff
size2 = 0x7fffffff
(0x7fffffff + 0x7fffffff = 0xfffffffe (-2)).
```

이런 일이 발생할 때 [2]에서 바운드 체킹은 통과되고, 그리고 훨씬 더 많은 out 버퍼가 의도한 것보다 쓰여질 수 있다. (사실, 임의의 메모리가 쓰여질 수 있는데, 이것은 두 번째 memcpy 호출에서 (out + size1)의 dest 파라미터가 메모리의 어떤 위치라도 갈 수 있도록 허용하기 때문이다.)

이런 버그는 보통의 signedness 버그와 정확하게 같은 방식으로 공략될 수 있으며, signedness 버그와 관련된 같은 문제를 가지고 있는데, 예를 들면, 음수 값이 거대한 양수 값으로 전환되고, 이것은 쉽게 세그먼트 폴트를 초래할 수 있다.

--[4 실제 예들

integer overflow 와 signedness 버그를 가지고 있는 많은 실제 어플리케이션들이 있으며, 특히 네트워크 데몬과 운영 체제 커널에서 자주 발견된다.

----[4.1 Integer overflow

이 예(공격할 수 없음)는 리눅스용 보안 모듈로부터 가지고 온 것이다. 이 코드는 커널 환경에서 실행된다:

```
int rsbac_acl_sys_group(enum  rsbac_acl_group_syscall_type_t call,
                        union rsbac_acl_group_syscall_arg_t arg)
  {
...
    switch(call)
      {
        case ACLGS_get_group_members:
          if(   (arg.get_group_members.maxnum <= 0) /* [A] */
             || !arg.get_group_members.group
```

```
            )
            {
...
            rsbac_uid_t * user_array;
            rsbac_time_t * ttl_array;

            user_array = vmalloc(sizeof(*user_array) *
            arg.get_group_members.maxnum);   /* [B] */
            if(!user_array)
              return -RSBAC_ENOMEM;
            ttl_array = vmalloc(sizeof(*ttl_array) *
            arg.get_group_members.maxnum); /* [C] */
            if(!ttl_array)
              {
                vfree(user_array);
                return -RSBAC_ENOMEM;
              }

            err =
            rsbac_acl_get_group_members(arg.get_group_members.group,
                                        user_array,
                                        ttl_array,

                                        arg.get_group_members.max
                                        num);
...
}
```

이 예에서, [A]에서 바운드 체킹은 [B]와 [C]에서 integer oveflow 를 막기에는 충분하지 않다. arg.get_group_members.maxnum 에 대해 아주 충분한 값(예를 들어, 0xffffffff / 4 보다 더 큰)을 전달하면 우리는 [B]와 [C]에서 곱셈으로 하여금 오버플로우하도록 초래하거나 어플리케이션이 예상한 것 보다 버퍼 ttl_array 와 user_array 를 더 작도록 강제한다. rsbac_acl_get_group_members 가 사용자가 통제한 데이터를 이 버퍼들에 복사하기 때문에 ttl_array 와 user_array 버퍼의 끝을 지나 쓰는 것이 가능하다. 이 경우에, 어플리케이션은 버퍼에 할당하기 위해 vmalloc()을 사용했고, 그래서 버퍼의 끝을 지나 쓰려고 하는 시도는 단순히 에러를 일으킬 것이며, 그래서 공략될 수 없다. 그렇지만 그것은 이 버그들이 실제 코드에서 어떻게 생겼는지의 예를 제공한다.

또 다른 하나의 실제 integer overflow 취약점의 예는 ISS X-Force 에 의해 발견된 XDR RPC 라이브러리에서의 문제이다. 이 경우, 사용자가 제공한 데이터는 동적으로 할당되는 버퍼(사용자가 제공한 데이터로 가득 차 있음)의 크기를 계산하는데 사용된다. 취약한 코드는 다음과 같다:

```
bool_t
xdr_array (xdrs, addrp, sizep, maxsize, elsize, elproc)
     XDR *xdrs;
     caddr_t *addrp;    /* array pointer */
     u_int *sizep;              /* number of elements */
     u_int maxsize;             /* max numberof elements */
     u_int elsize;              /* size in bytes of each element */
     xdrproc_t elproc;  /* xdr routine to handle each element */
{
  u_int i;
  caddr_t target = *addrp;
  u_int c;             /* the actual element count */
  bool_t stat = TRUE;
  u_int nodesize;

  ...

  c = *sizep;
  if ((c > maxsize) && (xdrs->x_op != XDR_FREE))
    {
      return FALSE;
    }
  nodesize = c * elsize;    /* [1] */

  ...

  *addrp = target = mem_alloc (nodesize);   /* [2] */

  ...

  for (i = 0; (i < c) && stat; i++)
    {
      stat = (*elproc) (xdrs, target, LASTUNSIGNED);   /* [3] */
```

```
          target += elsize;
        }
```

독자 여러분들도 볼 수 있듯이, elsize 와 c (sizep)에 대해 커다란 값을 제공하면 [1]에서 곱하기로 하여금 오버플로우하게 하고, nodesize 가 어플리케이션이 예상한 것보다 훨씬 더 적게 만들어버린다. nodesize 가 [2]에서 버퍼를 할당하기 위해 사용되기 때문에 버퍼는 크기가 잘못 되어 [3]에서 힙 오버플로우로 이어진다. 이 취약점에 대한 더 많은 정보는 마지막 부분에 나오는 참고 문헌에 열거된 CERT 권고문을 참고해라.

----[4.2 Signedness bug

최근, 여러 signedness bug 가 FreeBSD 커널에서 발견되고 있다. 이 취약점들은 커널 메모리의 많은 부분이 음의 길이 파라미터를 다한 syscall 에 전달함으로써 읽혀지도록 허용한다. getpeername(2) 함수는 그와 같은 문제를 가지고 있으며, 다음과 같다:

```
    static int
    getpeername1(p, uap, compat)
        struct proc *p;
        register struct getpeername_args /* {
            int    fdes;
            caddr_t asa;
            int    *alen;
    } */ *uap;
    int compat;
    {
        struct file *fp;
        register struct socket *so;
        struct sockaddr *sa;
        int len, error;

        ...

        error = copyin((caddr_t)uap->alen, (caddr_t)&len, sizeof (len));
        if (error) {
            fdrop(fp, p);
            return (error);
        }
```

```
        ...

        len = MIN(len, sa->sa_len);    /* [1] */
        error = copyout(sa, (caddr_t)uap->asa, (u_int)len);
        if (error)
            goto bad;
    gotnothing:
        error = copyout((caddr_t)&len, (caddr_t)uap->alen, sizeof (len));
    bad:
        if (sa)
            FREE(sa, M_SONAME);
        fdrop(fp, p);
        return (error);
    }
```

이것은 signedness 버그의 고전적인 예이다. [1]에서 체킹은 len 이 음수가 될 수 있다는 사실을 고려하지 않았으며, 이 경우 MIN 매크로는 항상 len 을 리턴할 것이다. 이 음수 len 파라미터가 copyout 으로 전달되었을 때 그것은 copyout 로 하여금 커널 메모리의 4GB 까지 사용자의 공간에 복사하도록 초래하는 거대한 양의 정수로 해석된다.

--[결론

Integer overflow 는 아주 위험할 수 있는데, 부분적으로 integer overflow 가 발생한 후 탐지하는 것이 불가능하기 때문이다. 만약 integer overflow 가 발생하면 어플리케이션은 그것이 수행한 계산이 정확한지 알 수 없으며, 그리고 어플리케이션은 계산이 정확할 거라는 추측 하에서 계속될 것이다. 비록 이 버그가 공략하기에 어려울 수 있지만, 그리고 전혀 공격할 수 없는 경우도 자주 있지만 예상치 못한 행위를 초래할 수 있으며, 이것은 안전한 시스템에는 결코 좋은 일이 아니다.

--[참고문헌

CERT advisory on the XDR bug:
http://www.cert.org/advisories/CA-2002-25.html
FreeBSD advisory: http://online.securityfocus.com/advisories/4407