Integer overflows

Александр Юрченко <grange@rt.mipt.ru>

13 апреля 2003

Содержание

1 Введение

С тех пор как обнаружили первые ошибки в программах, связаные с простейшими переполнениями буфера в стеке, было открыто много новых классов уязвимостей: heap overflows, format string, tmp races, signal races. Каждый следующий класс был изощренней как по своей сути, так и по способам использования.

Последние несколько лет ознаменованы обнаружением ошибок совершенно нового класса: integer overflows. Их легко допустить в программе, тяжело обнаружить и еще сложнее использовать для атак.

2 Что такое integer overflow

В языке C переменные типа integer используются для хранения целых чисел. Обычно размер переменной типа integer равен размеру регистров общего назначения того процессора, для которого компилируется программа. На многих архитектурах (в том числе и i386) этот размер составляет 32 бита. По умолчанию переменная типа integer является знаковой (signed), знак хранится в самом старшем бите: он равен нулю для положительных значений и единице для отрицательных. Переменные без знака объявляются явно указанием ключевого слова unsigned. Таким образом модуль знаковой переменной не может превышать $2^{31} - 1$, а беззнаковая представляет собой число от 0 до $2^{32} - 1$.

Под термином „integer overflow “понимается ситуация, когда переменной типа integer пытаются присвоить значение большее, чем она может вместить. Согласно стандарту языка C ISO C99, при таком переполнении компилятор вправе делать все что угодно, например полностью его игнорировать, что часто и происходит, приводя к неожиданным результатам.

3 Виды ошибок типа integer overflow

3.1 Переполнение размера переменной

Посмотрим на такой пример кода:

```
$ cat prog.c
int main()
{
        int a = 0x12345678;
        short b = a;
        char c = b;

        printf("a = 0x%x, b = 0x%x, c = 0x%x\n", a, b, c);
}
```

Результат его работы выглядит следующим образом:

```
$ gcc prog.c -o prog
$ ./prog
a = 0x2345678, b = 0x5678, c = 0x78
```

В данном случае размер переменной `a` равен 32 бита, переменной `b` 16 бит, а `c` — 8 бит. Видно, что при попытке присвоить переменной значение большее, чем она может вместить, происходит просто отбрасывание старшей части значения, которое не помещается.

Чтобы продемонстрировать возможность использования подобной ошибки, приведем такую программу:

```
$ cat prog.c
#include <err.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char buf[256];

void check_and_copy(char *tmpbuf, unsigned short len)
```

```
{
        printf("copying %d bytes\n", len);
        if (len > sizeof(buf) - 1)
                errx(1, "string too long");
        strcpy(buf, tmpbuf);
}

int main(int argc, char *argv[])
{
        size_t len;
        char *tmpbuf;

        if (argc != 2) {
                fprintf(stderr, "usage: %s len\n", argv[0]);
                exit(1);
        }
        len = atoi(argv[1]);
        if ((tmpbuf = malloc(len + 1)) == NULL)
                err(1, "malloc()");
        memset(tmpbuf, 'x', len);
        tmpbuf[len] = '\0';
        check_and_copy(tmpbuf, len);
        printf("%s\n", buf);
}
$ gcc prog.c -o prog
$ ./prog 5
copying 5 bytes
xxxxx
$ ./prog 65570
copying 34 bytes
Segmentation fault (core dumped)
```

В этом примере из-за неправильного выбора размера переменной произошло переполнение буфера.

Следует заметить одну тонкость, что в случае, когда присваемое значение отрицательно, знаковый бит, хоть и содержится в отбрасываемой части, переносится в результирующее значение. Пример:

```
$ cat prog.c
int main()
{
        int a = -0x102030;
        short b = a;

        printf("a = %d, b = %d\n", a, b);
}
```

```
$ gcc prog.c -o prog
$ ./prog
a = -1056816, b = -8240
```

3.2 Переполнение при арифметических операциях

Значение переменной может превысить предел, который она вмещает, не только при присвоении, но и при арифметических операциях. Следующий код демонстрирует такую ошибку:

```
$ cat prog.c
int main()
{
        unsigned int a = 0xffffff00;

        a += 0x120;
        printf("a = 0x%x\n", a);
}
$ cc prog.c -o prog
$ ./prog
a = 0x20
```

Здесь также происходит отбрасывание старшей части, так как результат операции 0x100000020 не помещается в 32 бита. Пример уязвимой программы:

```
$ cat prog.c
#include <err.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char pattern[] = "XX0000XX";

int main(int argc, char *argv[])
{
        int num, i;
        size_t len;
        char *buf;

        if (argc != 2) {
                fprintf(stderr, "usage: %s num\n", argv[0]);
                exit(1);
        }
        num = atoi(argv[1]);
        len = num * strlen(pattern) + 1;
```

```
        printf("total length %u bytes\n", len);
        if ((buf = malloc(len)) == NULL)
                err(1, "malloc()");
        for (i = 0; i < num; i++)
                strcat(buf, pattern);
        printf("%s\n", buf);
}
$ gcc prog.c -o prog
$ ./prog 5
total length 41 bytes
XX0000XXXX0000XXXX0000XXXX0000XXXX0000XX
$ ./prog 536870913
total length 9 bytes
Segmentation fault (core dumped)
```

Переполнение произошло в результате операции умножения при вычислении значения переменной `len`. Из-за этого не было выделено достаточное количество памяти, что привело к переполнению буфера.

3.3 Переполнение в знаке

Переполнения в знаке возникают при рассматривании знаковой переменной как беззнаковой и наоборот. Для демонстрации такого поведения приведем следующий код:

```
$ cat prog.c
int main()
{
        int a = 0x70000010;

        printf("a = %d\n", a);
        a <<= 1;
        printf("a = %d\n", a);
}
$ cc prog.c -o prog
$ ./prog
a = 1879048208
a = -536870880
```

Легко понять, что произошло. При операции сдвига число `a` трактовалось как беззнаковое 32-битное число, после сдвига старший бит стал равен единице, число стало отрицательным.

Часто ошибки, связаные со знаком переменных, приводят к возможности обойти различные проверки, а так же позволяют производить некорректную индексацию в массивах, как например в этой программе:

```
$ cat prog.c
#include <err.h>
#include <stdio.h>
#include <stdlib.h>

char abc[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

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

        if (argc != 2) {
                fprintf(stderr, "usage: %s num\n", argv[0]);
                exit(1);
        }
        num = atoi(argv[1]);
        if (num > 26)
                errx(1, "%d: invalid number", num);
        printf("%c\n", abc[num - 1]);
}
$ gcc prog.c -o prog
$ ./prog 5
E
$ ./prog 30
prog: 30: invalid number
$ ./prog -30000
Segmentation fault (core dumped)
```

4 Реальные примеры ошибок типа integer overflow

На сегодняшний день список успешно использованных для атак уязвимостей типа integer overflow довольно большой. Вот некоторые примеры: уязвимость в OpenSSH, уязвимости в ядрах OpenBSD и FreeBSD, уязвимость в Apache, уязвимости в sendmail. Рассмотрим несколько примеров для демонстрации изложеного выше материала.

4.1 Уязвимость в OpenSSH

Уязвимость в OpenSSH связана с механизмом challenge response протокола ssh версии 2. Уязвимый участок кода находится в файле auth2-chall.c:

```
static void
input_userauth_info_response(int type, u_int32_t seq, void *ctxt)
{
```

```
        Authctxt *authctxt = ctxt;
        KbdintAuthctxt *kbdintctxt;
        int i, authenticated = 0, res, len;
        u_int nresp;

...

        nresp = packet_get_int();
        if (nresp > 0) {
                response = xmalloc(nresp * sizeof(char*));
                for (i = 0; i < nresp; i++)
                        response[i] = packet_get_string(NULL);
        }
```

Видно, что в цикле происходит сравнение знаковой переменной `i` с беззнаковой `nresp`. Поэтому если `nresp` >= 0x80000000, переменная `i` при очередной итерации станет отрицательной (переполнение в знаке), и проверка не сработает, присвоение значения элементу `response[i]` будет произведено некорректно.

4.2 Уязвимость в Apache

Уязвимость в Apache связана с механизмом chunk encoding. Уязвимый участок кода находится в файле http_protocol.c:

```
API_EXPORT(long) ap_get_client_block(request_rec *r, char *buffer, int bufsiz)
{
    int c;
    long len_read, len_to_read;
    long chunk_start = 0;
    unsigned long max_body;

    if (!r->read_chunked) {     /* Content-length read */
        len_to_read = (r->remaining > bufsiz) ? bufsiz : r->remaining;
        len_read = ap_bread(r->connection->client, buffer, len_to_read);
```

Сначала переменная `bufsiz` рассматривается как знаковая, и если ее значение отрицательно, то переменная `len_to_read` будет равна `bufsiz`. Когда же `len_to_read` будет передана в качестве параметра функции `ap_bread()`, она будет рассматриваться как большое беззнаковое число, что приведет к переполнению при копировании данных.

5 Заключение

В отличии от переполнений при использовании функций типа strcpy или уязвимостей format string не существует каких-либо способов автоматизировать аудит кода на предмет уязвимостей типа integer overflow, хотя в некоторых случаях подобные ошибки могут быть выявлены на этапе компиляции. Поэтому их обнаружение очень сложно, а использование для атак еще сложнее. Каждый эксплойт для integer overflow — произведение искусства. Но несмотря на это подобные уязвимости будут доминировать в ближайшее время, так как из-за своей новизны позволяют находить бреши в, казалось бы, много раз проверенных программах.

Список литературы

[1] Basic Integer Overflows, blexim <blexim@hush.com>, http://www.phrack.org/show.php?p=60&a=10

[2] Smashing The Kernel Stack For Fun And Profit, Sinan "noir"Eren <noir@olympos.org>, http://www.phrack.org/show.php?p=60&a=6

[3] Big Loop Integer Protection, Oded Horovitz <ohorovitz@entercept.com>, http://www.phrack.org/show.php?p=60&a=9

[4] OpenSSH Security Advisory, http://www.openssh.com/txt/preauth.adv

[5] Apache Security Advisory, http://httpd.apache.org/info/security_bulletin_20020617.txt