

Basic Integer Overflows*

blexim

blexim@hush.com

Contents

1	Introduction	1
1.1	What is an integer?	1
1.2	What is an integer overflow?	2
1.3	Why can they be dangerous?	2
2	Integer overflows	2
2.1	Widthness overflows	3
2.2	Arithmetic overflows	5
3	Signedness Bugs	8
3.1	What do they look like?	8
3.2	Signedness bugs caused by integer overflows	9
4	Real world examples	10
4.1	Integer overflows	10
4.2	Signedness bugs	12
5	Conclusion	12
A	Appendix	13

1 Introduction

In this paper I'm going to describe two classes of programming bugs which can sometimes allow a malicious user to modify the execution path of an affected process. Both of these classes of bug work by causing variables to contain unexpected values, and so are not as "direct" as classes which overwrite memory, e.g. buffer overflows or format strings. All the examples given in the paper are in C, so a basic familiarity with C is assumed. A knowledge of how integers are stored in memory is also useful, but not essential.

1.1 What is an integer?

An integer, in the context of computing, is a variable capable of representing a real number with no fractional part. Integers are typically the same size as a pointer on the system they are compiled on (i.e. on a 32 bit system, such as i386, an integer is 32 bits long, on a 64 bit system, such as SPARC, an integer is 64 bits long). Some compilers don't use integers

*<http://www.phrack.org/issues/60/10.html>

and pointers of the same size however, so for the sake of simplicity all the examples refer to a 32 bit system with 32 bit integers, longs and pointers.

Integers, like all variables are just regions of memory. When we talk about integers, we usually represent them in decimal, as that is the numbering system humans are most used to. Computers, being digital, cannot deal with decimal, so internally to the computer integers are stored in binary. Binary is another system of representing numbers which uses only two numerals, 1 and 0, as opposed to the ten numerals used in decimal. As well as binary and decimal, hexadecimal (base sixteen) is often used in computing as it is very easy to convert between binary and hexadecimal.

Since it is often necessary to store negative numbers, there needs to be a mechanism to represent negative numbers using only binary. The way this is accomplished is by using the most significant bit (MSB) of a variable to determine the sign: if the MSB is set to 1, the variable is interpreted as negative; if it is set to 0, the variable is positive. This can cause some confusion, as will be explained in the section on signedness bugs, because not all variables are signed, meaning they do not all use the MSB to determine whether they are positive or negative. These variable are known as unsigned and can only be assigned positive values, whereas variables which can be either positive or negative are called signed.

1.2 What is an integer overflow?

Since an integer is a fixed size (32 bits for the purposes of this paper), there is a fixed maximum value it can store. When an attempt is made to store a value greater than this maximum value it is known as an integer overflow. The ISO C99 standard says that an integer overflow causes “undefined behaviour”, meaning that compilers conforming to the standard may do anything they like from completely ignoring the overflow to aborting the program. Most compilers seem to ignore the overflow, resulting in an unexpected or erroneous result being stored.

1.3 Why can they be dangerous?

Integer overflows cannot be detected after they have happened, so there is not way for an application to tell if a result it has calculated previously is in fact correct. This can get dangerous if the calculation has to do with the size of a buffer or how far into an array to index. Of course most integer overflows are not exploitable because memory is not being directly overwritten, but sometimes they can lead to other classes of bugs, frequently buffer overflows. As well as this, integer overflows can be difficult to spot, so even well audited code can spring surprises.

2 Integer overflows

So what happens when an integer overflow does happen? ISO C99 has this to say:

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.

Modulo arithmetic involves dividing two numbers and taking the remainder, e.g.

$$10 \text{ modulo } 5 = 0$$

$$11 \bmod 5 = 1$$

so reducing a large value modulo ($\text{MAXINT} + 1$) can be seen as discarding the portion of the value which cannot fit into an integer and keeping the rest. In C, the modulo operator is a `%` sign.

This is a bit wordy, so maybe an example will better demonstrate the typical “undefined behaviour”:

We have two unsigned integers, `a` and `b`, both of which are 32 bits long. We assign to `a` the maximum value a 32 bit integer can hold, and to `b` we assign 1. We add `a` and `b` together and store the result in a third unsigned 32 bit integer called `r`:

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

Now, since the result of the addition cannot be represented using 32 bits, the result, in accordance with the ISO standard, is reduced modulo $0x100000000$.

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

Reducing the result using modulo arithmetic basically ensures that only the lowest 32 bits of the result are used, so integer overflows cause the result to be truncated to a size that can be represented by the variable. This is often called a “wrap around”, as the result appears to wrap around to 0.

2.1 Widthness overflows

So an integer overflow is the result of attempting to store a value in a variable which is too small to hold it. The simplest example of this can be demonstrated by simply assigning the contents of large variable to a smaller one:

```
1  /* ex1.c - loss of precision */
2  #include <stdio.h>
3
4  int main(void){
5      int l;
6      short s;
7      char c;
8
9      l = 0xdeadbeef;
10     s = l;
11     c = l;
12
13     printf("l = 0x%x (%d bits)\n", l, sizeof(l) * 8);
14     printf("s = 0x%x (%d bits)\n", s, sizeof(s) * 8);
15     printf("c = 0x%x (%d bits)\n", c, sizeof(c) * 8);
16
17     return 0;
18 }
19 /* EOF */
```

The output of which looks like this:

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

Since each assignment causes the bounds of the values that can be stored in each type to be exceeded, the value is truncated so that it can fit in the variable it is assigned to.

It is worth mentioning integer promotion here. When a calculation involving operands of different sizes is performed, the smaller operand is “promoted” to the size of the larger one. The calculation is then performed with these promoted sizes and, if the result is to be stored in the smaller variable, the result is truncated to the smaller size again. For example:

```
int i;
short s;

s = i;
```

A calculation is being performed with different sized operands here. What happens is that the variable `s` is promoted to an `int` (32 bits long), then the contents of `i` is copied into the new promoted `s`. After this, the contents of the promoted variable are “demoted” back to 16 bits in order to be saved in `s`. This demotion can cause the result to be truncated if it is greater than the maximum value `s` can hold.

2.1.1 Exploiting

Integer overflows are not like most common bug classes. They do not allow direct overwriting of memory or direct execution flow control, but are much more subtle. The root of the problem lies in the fact that there is no way for a process to check the result of a computation after it has happened, so there may be a discrepancy between the stored result and the correct result. Because of this, most integer overflows are not actually exploitable. Even so, in certain cases it is possible to force a crucial variable to contain an erroneous value, and this can lead to problems later in the code.

Because of the subtlety of these bugs, there is a huge number of situations in which they can be exploited, so I will not attempt to cover all exploitable conditions. Instead, I will provide examples of some situations which are exploitable, in the hope of inspiring the reader in their own research :)

```
1  /* width1.c - exploiting a trivial widthness bug */
2  #include <stdio.h>
3  #include <string.h>
4
5  int main(int argc, char *argv[]){
6      unsigned short s;
7      int i;
8      char buf[80];
9
10     if(argc < 3){
11         return -1;
12     }
13
14     i = atoi(argv[1]);
15     s = i;
16
17     if(s >= 80){                /* [w1] */
18         printf("Oh no you don't!\n");
19         return -1;
20     }
21
22     printf("s = %d\n", s);
23
24     memcpy(buf, argv[2], i);
25     buf[i] = '\0';
26     printf("%s\n", buf);
27 }
```

```

28     return 0;
29 }

```

Listing 1: Example 1

While a construct like this would probably never show up in real life code, it serves well as an example. Take a look at the following inputs:

```

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)

```

The length argument is taken from the command line and held in the integer `i`. When this value is transferred into the short integer `s`, it is truncated if the value is too great to fit into `s` (i.e. if the value is greater than 65535). Because of this, it is possible to bypass the bounds check at `[w1]` and overflow the buffer. After this, standard stack smashing techniques can be used to exploit the process.

2.2 Arithmetic overflows

As shown in section 2.0, if an attempt is made to store a value in an integer which is greater than the maximum value the integer can hold, the value will be truncated. If the stored value is the result of an arithmetic operation, any part of the program which later uses the result will run incorrectly as the result of the arithmetic being incorrect. Consider this example demonstrating the wrap around shown earlier:

```

1  /* ex2.c - an integer overflow */
2  #include <stdio.h>
3
4  int main(void){
5      unsigned int num = 0xffffffff;
6
7      printf("num is %d bits long\n", sizeof(num) * 8);
8      printf("num = 0x%x\n", num);
9      printf("num + 1 = 0x%x\n", num + 1);
10
11     return 0;
12 }
13 /* EOF */

```

The output of this program looks like this:

```

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

```

The astute reader will have noticed that `0xffffffff` is decimal -1, so it appears that we're just doing

$$1 + (-1) = 0$$

Whilst this is one way at looking at what's going on, it may cause some confusion since the variable `num` is unsigned and therefore all arithmetic done on it will be unsigned. As it happens, a lot of signed arithmetic depends on integer overflows, as the following demonstrates (assume both operands are 32 bit variables):

$$\begin{array}{rcl} -700 & + & 800 = 100 \\ 0xffffffff44 & + & 0x320 = 0x100000064 \end{array}$$

Since the result of the addition exceeds the range of the variable, the lowest 32 bits are used as the result. These low 32 bits are `0x64`, which is equal to decimal 100.

Since an integer is signed by default, an integer overflow can cause a change in signedness which can often have interesting effects on subsequent code. Consider the following example:

```
1  /* ex3.c - change of signedness */
2  #include <stdio.h>
3
4  int main(void){
5      int l;
6
7      l = 0x7fffffff;
8
9      printf("l = %d (0x%x)\n", l, l);
10     printf("l + 1 = %d (0x%x)\n", l + 1 , l + 1);
11
12     return 0;
13 }
14 /* EOF */
```

The output of which is:

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

Here the integer is initialised with the highest positive value a signed long integer can hold. When it is incremented, the most significant bit (indicating signedness) is set and the integer is interpreted as being negative.

Addition is not the only arithmetic operation which can cause an integer to overflow. Almost any operation which changes the value of a variable can cause an overflow, as demonstrated in the following example:

```
1  /* ex4.c - various arithmetic overflows */
2  #include <stdio.h>
3
4  int main(void){
5      int l, x;
6
7      l = 0x40000000;
8
9      printf("l = %d (0x%x)\n", l, l);
10
11     x = l + 0xc0000000;
12     printf("l + 0xc0000000 = %d (0x%x)\n", x, x);
13
14     x = l * 0x4;
15     printf("l * 0x4 = %d (0x%x)\n", x, x);
16
17     x = l - 0xffffffff;
```

```

18     printf("1 - 0xffffffff = %d (0x%x)\n", x, x);
19
20     return 0;
21 }
22 /* EOF */

```

Output:

```

nova:signed {55} ./ex4
1 = 1073741824 (0x40000000)
1 + 0xc0000000 = 0 (0x0)
1 * 0x4 = 0 (0x0)
1 - 0xffffffff = 1073741825 (0x40000001)

```

The addition is causing an overflow in exactly the same way as the first example, and so is the multiplication, although it may seem different. In both cases the result of the arithmetic is too great to fit in an integer, so it is reduced as described above. The subtraction is slightly different, as it is causing an underflow rather than an overflow: an attempt is made to store a value lower than the minimum value the integer can hold, causing a wrap around. In this way we are able to force an addition to subtract, a multiplication to divide or a subtraction to add.

2.2.1 Exploiting

One of the most common ways arithmetic overflows can be exploited is when a calculation is made about how large a buffer must be allocated. Often a program must allocate space for an array of objects, so it uses the `malloc(3)` or `calloc(3)` routines to reserve the space and calculates how much space is needed by multiplying the number of elements by the size of an object. As has been previously shown, if we are able to control either of these operands (number of elements or object size) we may be able to mis-size the buffer, as the following code fragment shows:

```

1     int myfunction(int *array, int len){
2         int *myarray, i;
3
4         myarray = malloc(len * sizeof(int));    /* [1] */
5         if(myarray == NULL){
6             return -1;
7         }
8
9         for(i = 0; i < len; i++){                /* [2] */
10             myarray[i] = array[i];
11         }
12
13         return myarray;
14     }

```

This seemingly innocent function could bring about the downfall of a system due to its lack of checking of the `len` parameter. The multiplication at [1] can be made to overflow by supplying a high enough value for `len`, so we can force the buffer to be any length we choose. By choosing a suitable value for `len`, we can cause the loop at [2] to write past the end of the `myarray` buffer, resulting in a heap overflow. This could be leveraged into executing arbitrary code on certain implementations by overwriting malloc control structures, but that is beyond the scope of this article.

Another example:

```

1     int catvars(char *buf1, char *buf2, unsigned int len1,
2                 unsigned int len2){
3         char mybuf[256];
4

```

```

5         if((len1 + len2) > 256){           /* [3] */
6             return -1;
7         }
8
9         memcpy(mybuf, buf1, len1);         /* [4] */
10        memcpy(mybuf + len1, buf2, len2);
11
12        do_some_stuff(mybuf);
13
14        return 0;
15    }

```

In this example, the check at [3] can be bypassed by using suitable values for `len1` and `len2` that will cause the addition to overflow and wrap around to a low number. For example, the following values:

```

len1 = 0x104
len2 = 0xffffffffc

```

when added together would result in a wrap around with a result of 0x100 (decimal 256). This would pass the check at [3], then the `memcpy(3)`'s at [4] would copy data well past the end of the buffer.

3 Signedness Bugs

Signedness bugs occur when an unsigned variable is interpreted as signed, or when a signed variable is interpreted as unsigned. This type of behaviour can happen because internally to the computer, there is no distinction between the way signed and unsigned variables are stored. Recently, several signedness bugs showed up in the FreeBSD and OpenBSD kernels, so there are many examples readily available.

3.1 What do they look like?

Signedness bugs can take a variety of forms, but some of the things to look out for are:

- signed integers being used in comparisons
- signed integers being used in arithmetic
- unsigned integers being compared to signed integers

Here is classic example of a signedness bug:

```

1    int copy_something(char *buf, int len){
2        char kbuf[800];
3
4        if(len > sizeof(kbuf)){           /* [1] */
5            return -1;
6        }
7
8        return memcpy(kbuf, buf, len);    /* [2] */
9    }

```

The problem here is that `memcpy` takes an unsigned `int` as the `len` parameter, but the bounds check performed before the `memcpy` is done using signed integers. By passing a negative value for `len`, it is possible to pass the check at [1], but then in the call to `memcpy` at [2], `len` will be interpreted as a huge unsigned value, causing memory to be overwritten well past the end of the buffer `kbuf`.

Another problem that can stem from signed/unsigned confusion occurs when arithmetic is performed. Consider the following example:


```

1  int table[800];
2
3  int insert_in_table(int val, int pos){
4      if(pos > sizeof(table) / sizeof(int)){
5          return -1;
6      }
7
8      table[pos] = val;
9
10     return 0;
11 }

```

Since the line

```
table[pos] = val;
```

is equivalent to

```
*(table + (pos * sizeof(int))) = val;
```

we can see that the problem here is that the code does not expect a negative operand for the addition: it expects `(table + pos)` to be greater than `table`, so providing a negative value for `pos` causes a situation which the program does not expect and can therefore not deal with.

3.1.1 Exploiting

This class of bug can be problematic to exploit, due to the fact that signed integers, when interpreted as unsigned, tend to be huge. For example, -1 when represented in hexadecimal is 0xffffffff. When interpreted as unsigned, this becomes the greatest value it is possible to represent in an integer (4,294,967,295), so if this value is passed to `memcpy` as the `len` parameter (for example), `memcpy` will attempt to copy 4GB of data to the destination buffer. Obviously this is likely to cause a segfault or, if not, to trash a large amount of the stack or heap. Sometimes it is possible to get around this problem by passing a very low value for the source address and hope, but this is not always possible.

3.2 Signedness bugs caused by integer overflows

Sometimes, it is possible to overflow an integer so that it wraps around to a negative number. Since the application is unlikely to expect such a value, it may be possible to trigger a signedness bug as described above.

An example of this type of bug could look like this:

```

1  int get_two_vars(int sock, char *out, int len){
2      char buf1[512], buf2[512];
3      unsigned int size1, size2;
4      int size;
5
6      if(recv(sock, buf1, sizeof(buf1), 0) < 0){
7          return -1;
8      }
9      if(recv(sock, buf2, sizeof(buf2), 0) < 0){
10         return -1;
11     }
12
13     /* packet begins with length information */
14     memcpy(&size1, buf1, sizeof(int));
15     memcpy(&size2, buf2, sizeof(int));
16
17     size = size1 + size2;          /* [1] */
18
19     if(size > len){                /* [2] */

```



```

24         vfree(user_array);
25         return -RSBAC_ENOMEM;
26     }
27
28     err =
29         rsbac_acl_get_group_members(arg.get_group_members.group,
30                                   user_array,
31                                   ttl_array,
32
33                                   arg.get_group_members.max
34                                   num);
35     ...
36 }

```

In this example, the bounds checking at [A] is not sufficient to prevent the integer overflows at [B] and [C]. By passing a high enough (i.e. greater than $0xffffffff / 4$) value for `arg.get_group_members.maxnum`, we can cause the multiplications at [B] and [C] to overflow and force the buffers `ttl_array` and `user_array` to be smaller than the application expects. Since `rsbac_acl_get_group_members` copies user controlled data to these buffers, it is possible to write past the end of the `user_array` and `ttl_array` buffers. In this case, the application used `vmalloc()` to allocate the buffers, so an attempt to write past the end of the buffers will simply raise an error, so it cannot be exploited. Even so, it provides an example of what these bugs can look like in real code.

Another example of a recent real world integer overflow vulnerability was the problem in the XDR RPC library (discovered by ISS X-Force). In this case, user supplied data was used in the calculation of the size of a dynamically allocated buffer which was filled with user supplied data. The vulnerable code was this:

```

1  bool_t
2  xdr_array (xdrs, addrp, sizep, maxsize, elsize, elproc)
3      XDR *xdrs;
4      caddr_t *addrp;    /* array pointer */
5      u_int *sizep;      /* number of elements */
6      u_int maxsize;     /* max numberof elements */
7      u_int elsize;      /* size in bytes of each element */
8      xdrproc_t elproc;  /* xdr routine to handle each element */
9
10 {
11     u_int i;
12     caddr_t target = *addrp;
13     u_int c;      /* the actual element count */
14     bool_t stat = TRUE;
15     u_int nodesize;
16
17     ...
18
19     c = *sizep;
20     if ((c > maxsize) && (xdrs->x_op != XDR_FREE))
21     {
22         return FALSE;
23     }
24     nodesize = c * elsize;    /* [1] */
25
26     ...
27
28     *addrp = target = mem_alloc (nodesize);    /* [2] */
29
30     ...
31
32     for (i = 0; (i < c) && stat; i++)
33     {
34         stat = (*elproc) (xdrs, target, LASTUNSIGNED);    /* [3] */
35         target += elsize;
36     }
37 }

```

As you can see, by supplying large values for `elsize` and `c (sizep)`, it was possible to cause the multiplication at [1] to overflow and cause `nodesize` to be much smaller than the application expected. Since `nodesize` was then used to allocate a buffer at [2], the buffer could be mis-sized leading to a heap overflow at [3]. For more information on this hole, see the CERT advisory listed in the appendix.

4.2 Signedness bugs

Recently, several signedness bugs were brought to light in the freebsd kernel. These allowed large portions of kernel memory to be read by passing negative length parameters to various syscalls. The `getpeername(2)` function had such a problem and looked like this:

```

1  static int
2  getpeername1(p, uap, compat)
3      struct proc *p;
4      register struct getpeername_args /* {
5          int fdes;
6          caddr_t asa;
7          int *alen;
8      } */ *uap;
9      int compat;
10 {
11     struct file *fp;
12     register struct socket *so;
13     struct sockaddr *sa;
14     int len, error;
15
16     ...
17
18     error = copyin((caddr_t)uap->alen, (caddr_t)&len, sizeof (len));
19     if (error) {
20         fdrop(fp, p);
21         return (error);
22     }
23
24     ...
25
26     len = MIN(len, sa->sa_len);    /* [1] */
27     error = copyout(sa, (caddr_t)uap->asa, (u_int)len);
28     if (error)
29         goto bad;
30 gotnothing:
31     error = copyout((caddr_t)&len, (caddr_t)uap->alen, sizeof (len));
32 bad:
33     if (sa)
34         FREE(sa, M_SONAME);
35     fdrop(fp, p);
36     return (error);
37 }
```

This is a classic example of a signedness bug - the check at [1] did not take into account the fact that `len` could be negative, in which case the `MIN` macro would always return `len`. When this negative `len` parameter was passed to `copyout`, it was interpreted as a huge positive integer which caused `copyout` to copy up to 4GB of kernel memory to user space.

5 Conclusion

Integer overflows can be extremely dangerous, partly because it is impossible to detect them after they have happened. If an integer overflow takes place, the application cannot know that the calculation it has performed is incorrect, and it will continue under the

assumption that it is. Even though they can be difficult to exploit, and frequently cannot be exploited at all, they can cause unexpected behaviour, which is never a good thing in a secure system.

A Appendix

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