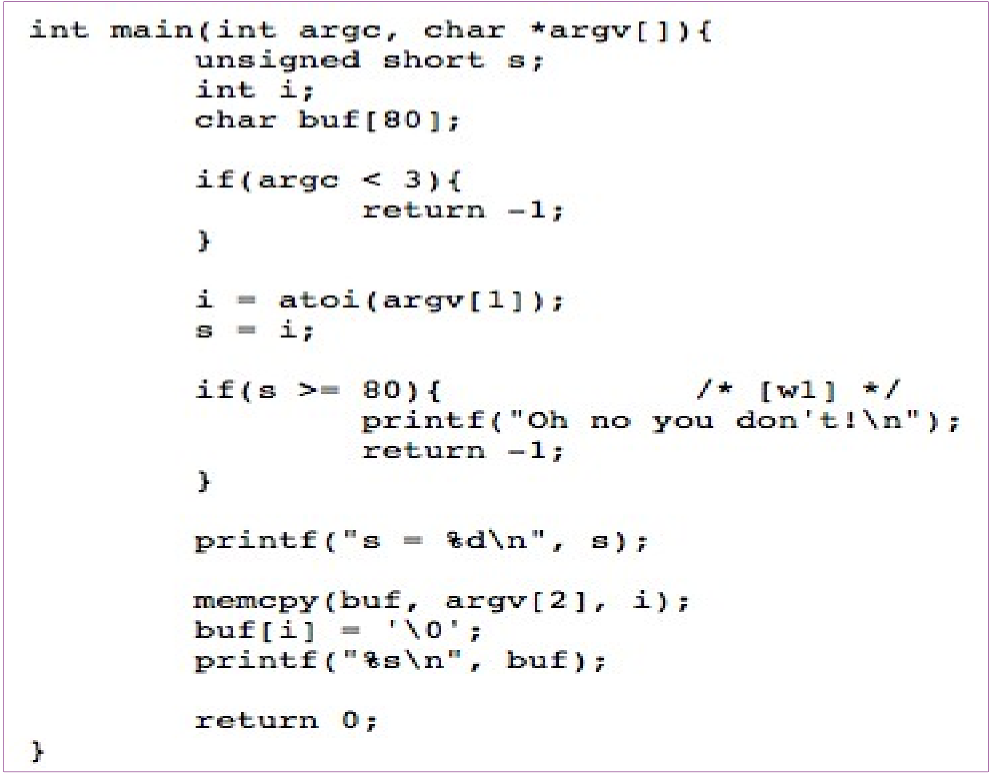
Integer Overflow Vulnerability

Chester Rebeiro

Indian Institute of Technology Madras

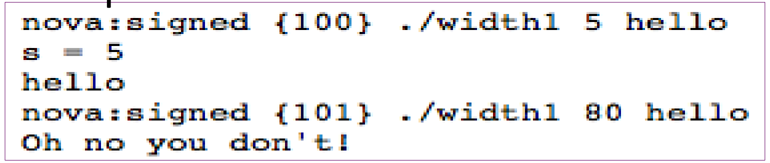
http://phrack.org/issues/60/10.html

What’s wrong with this code?



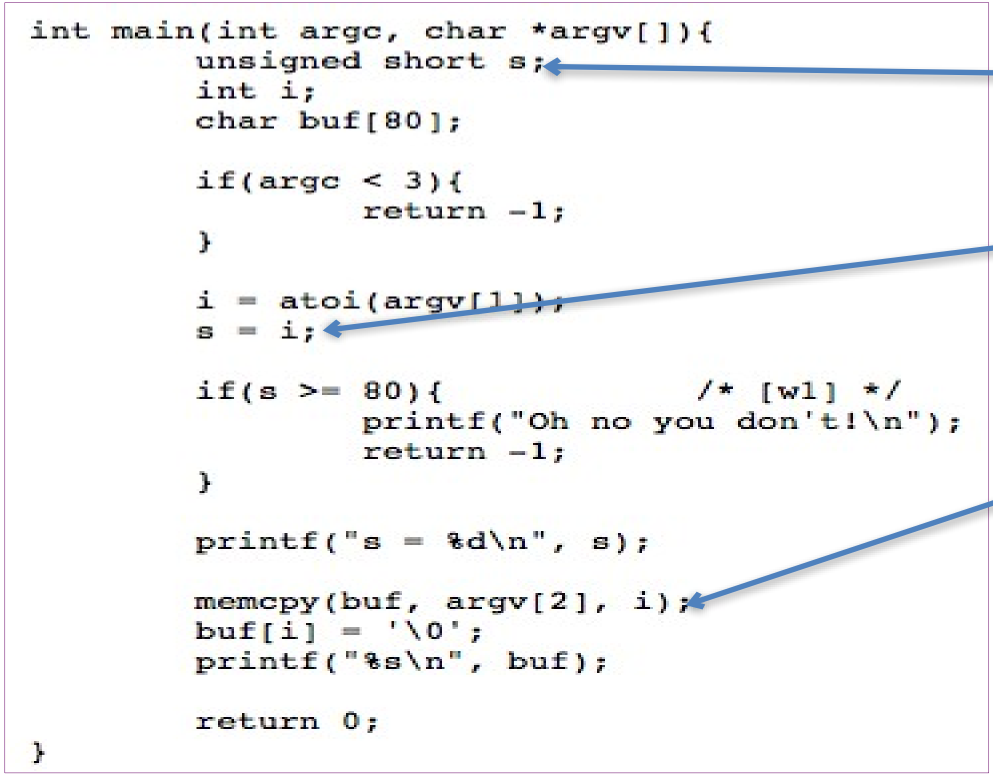
```
int main(int argc, char *argv[]){
        unsigned short s;
        int i;
        char buf[80];

        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;
}
```

Expected behavior



```
nova:signed {100} ./width1 5 hello
s = 5
hello
nova:signed {101} ./width1 80 hello
Oh no you don't!
```

What’s wrong with this code?



```
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;
}
```

Defined as short. Can hold a max value of 65535

If i > 65535, s overflows, therefore is truncated. So, the condition check is likely to be bypassed.

Will result in an overflow of buf, which can be used to perform nefarious activities

Integer Overflow Vulnerability

- Due to widthness overflow
- Due to arithmetic overflow
- Due to sign/unsigned problems

Widthness Overflows

Occurs when code tries to store a value in a variable that is too small (in the number of bits) to handle it.

For example: a cast from int to short

```
int a1 = 0x11223344;
char a2;
short a3;

a2 = (char) a1;
a3 = (short) a1;
```

```
a1 = 0x11223344
a2 = 0x44
a3 = 0x3344
```

Arithmetic Overflows

```
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;
}
```

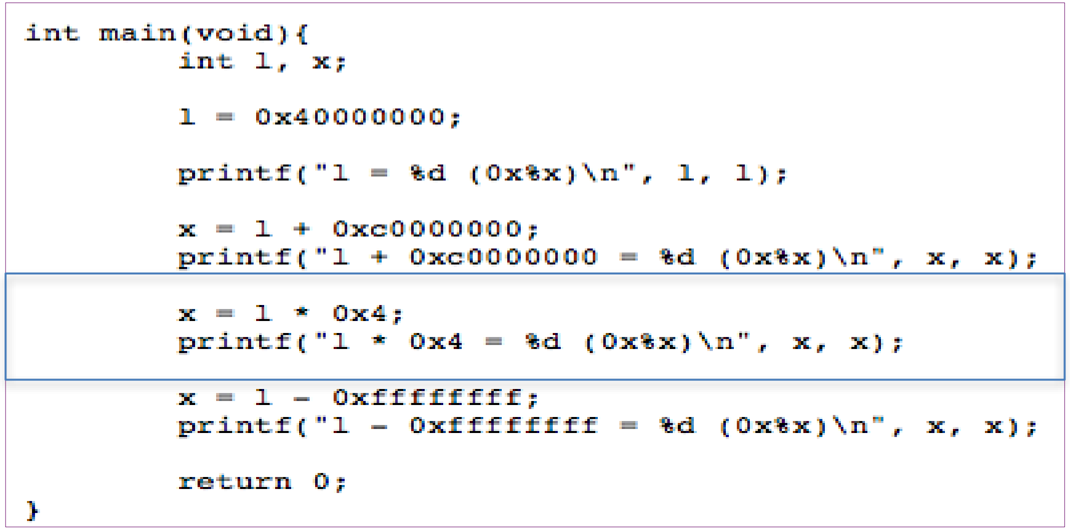
```
nova:signed {55} ./ex4
l = 1073741824 (0x40000000)
l + 0xc0000000 = 0 (0x0)
l * 0x4 = 0 (0x0)
l - 0xffffffff = 1073741825 (0x40000001)
```

Arithmetic Overflows

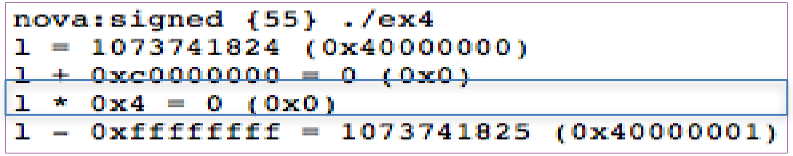
```
int main(void){
        int l, x;

        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;
}
```

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

Arithmetic Overflows

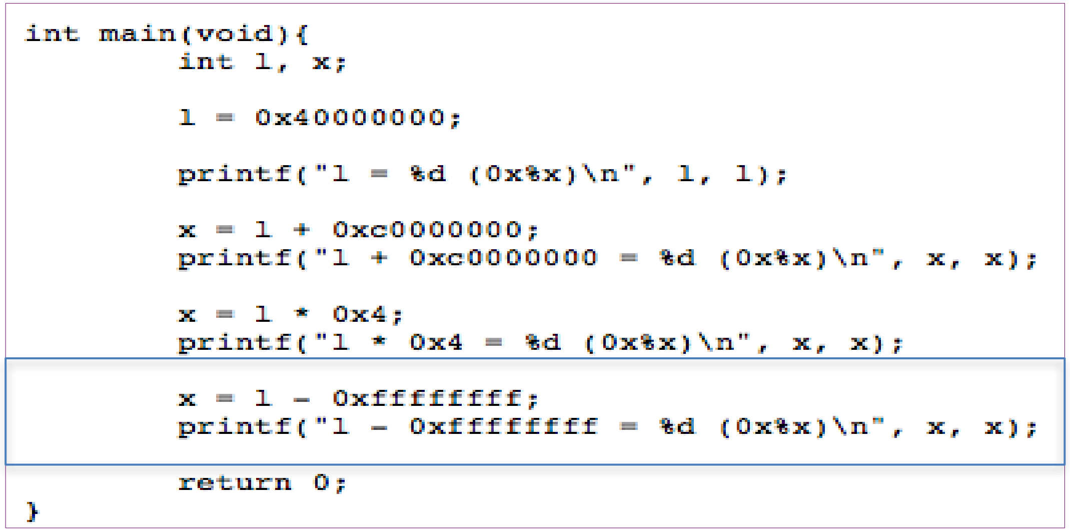


```
int main(void){
        int l, x;

        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;
}
```

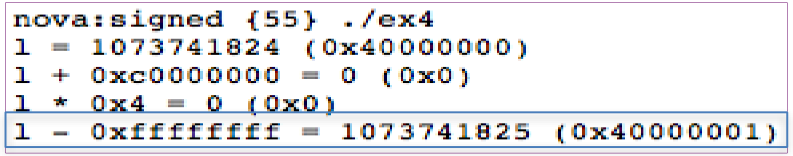


```
nova:signed {55} ./ex4
l = 1073741824 (0x40000000)
l + 0xc0000000 = 0 (0x0)
l * 0x4 = 0 (0x0)
l - 0xffffffff = 1073741825 (0x40000001)
```

Arithmetic Overflows

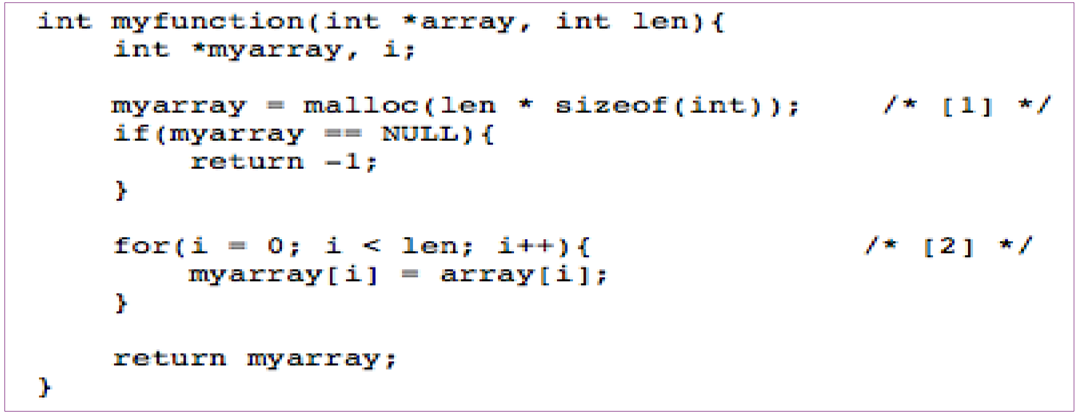


```
int main(void){
        int l, x;

        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;
}
```



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

Exploit 1
(manipulate space allocated by malloc)



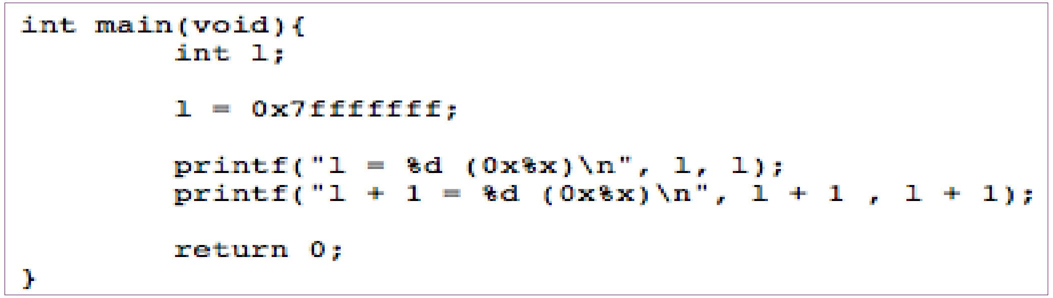
```
int myfunction(int *array, int len){
    int *myarray, i;

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

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

    return myarray;
}
```

Space allocated by malloc depends on len. If we choose a suitable value of len such that len*sizeof(int) overflows, then,

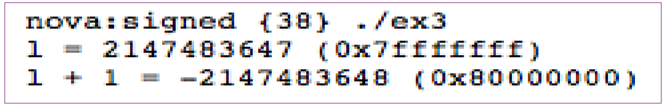
(1) myarray would be smaller than expected
(2) thus leading to a heap overflow
(3) which can be exploited

(Un)signed Integers

- Sign interpreted using the most significant bit.
- This can lead to unexpected results in comparisons and arithmetic



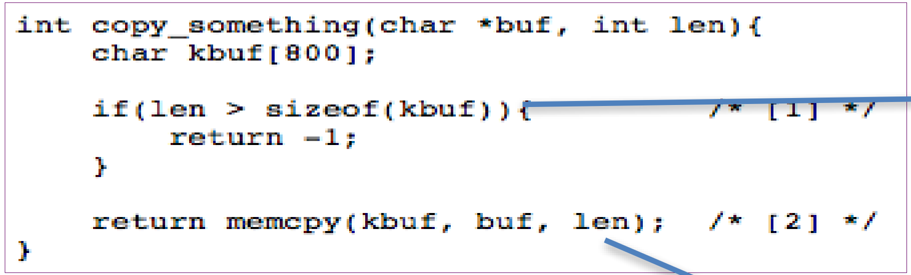
```
int main(void){
        int l;

        l = 0x7fffffff;

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

        return 0;
}
```



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

l is initialized with the highest positive value that a signed 32 bit integer can take.
When incremented, the MSB is set, and the number is interpreted as negative.

Sign Interpretations in compare



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

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

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

This test is with signed numbers. Therefore a negative len will pass the 'if' test.

In memcpy, len is interpreted as unsigned. Therefore a negative len will be treated as positive.

This could be used to overflow kbuf.

From the man pages
void ***memcpy**(void *restrict dst, const void *restrict src, size_t n);

Sign interpretations in arithmetic

```
int table[800];

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

    table[pos] = val;

    return 0;
}
```

If table + pos is greater than entries in table then return -1

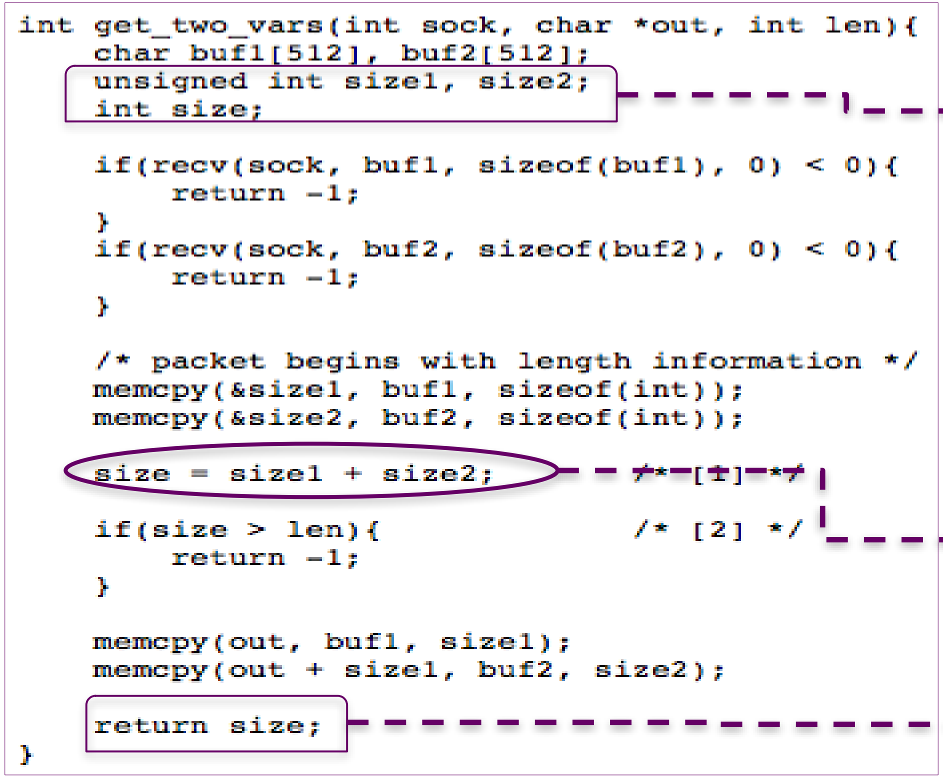
If *pos* is negative, this is not the case.

Causing *val* to be written to a location beyond the table

```
Since the line
     table[pos] = val;
is equivalent to
     *(table + (pos * sizeof(int))) = val;
```

This arithmetic done considering unsigned

exploiting overflow due to sign in a network deamon



```
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;
}
```

size1 and size2 are unsigned
Size is signed.

if size1 and size2 are large enough, size may end up being negative.

Size is returned, which may cause an out to overflow in the callee function

Ponder about

```
#define MAX_BUF_SIZE 64 * 1024

void store_into_buffer(const void *src, int num)
{
  char global_buffer[MAX_BUF_SIZE];

  if (num > MAX_BUF_SIZE)
    return;

  memcpy(global_buffer, src, num);

  [...]
}
```

Find the vulnerability in this function

Sign could lead to memory overreads.

```
#define MAX_BUF_SIZE 64 * 1024

void store_into_buffer(const void *src, int num)
{
  char global_buffer[MAX_BUF_SIZE];

  if (num > MAX_BUF_SIZE)
    return;

  memcpy(global_buffer, src, num);

  [...]
}
```

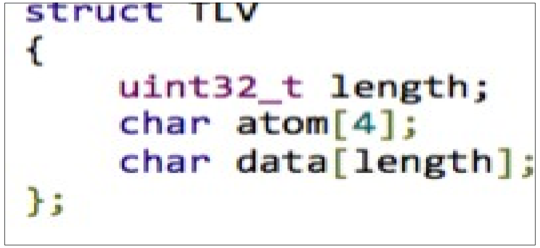
- num is a signed int
- If comparison assumes signed numbers
  - Any negative num is less than MAX_BUF_SIZE
- memcpy's 3rd parameter is unsigned. So, the negative number is interpreted as positive. Resulting in memory overreads.

Stagefright Bug

- Discovered by Joshua Drake and disclosed on July 27th, 2015
- Stagefright is a software library implemented in C++ for Android
- Stagefright attacks uses several integer based bugs to
  - execute remote code in phone
  - Achieve privilege escalation
- Attack is based on a well crafted MP3, MP4 message sent to the remote Android phone
  - Multiple vulnerabilities exploited:
    - One exploit targets MP4 subtitles that uses tx3g for timed text.
    - Another exploit targets covr (cover art) box

- Could have affected around one thousand million devices
  - Devices affected inspite of ASLR

MPEG4 Format



```
struct TLV
{
        uint32_t length;
        char atom[4];
        char data[length];
};
```

tx3g exploit

```
status_t MPEG4Source::parseChunk(off64_t *offset) {

  [...]

  uint64_t chunk_size = ntohl(hdr[0]);
  uint32_t chunk_type = ntohl(hdr[1]);
  off64_t data_offset = *offset + 8;

  if (chunk_size == 1) {
    if (mDataSource->readAt(*offset + 8, &chunk_size, 8) < 8) {
      return ERROR_IO;
    }
  chunk_size = ntoh64(chunk_size);

  [...]

  switch(chunk_type) {
  [...]

  case FOURCC('t', 'x', '3', 'g'):
  {
    uint32_t type;
    const void *data;
    size_t size = 0;
    if (!mLastTrack->meta->findData(
            kKeyTextFormatData, &type, &data, &size)) {
      size = 0;
    }

    uint8_t *buffer = new (std::nothrow) uint8_t[size + chunk_size];
    if (buffer == NULL) {
      return ERROR_MALFORMED;
    }

    if (size > 0) {
      memcpy(buffer, data, size);
    }
```

offset into file

int hdr[2] is the first two words read from offset

chunksize of 1 has a special meaning.

(1) chunk_size is uint64_t,
(2) it is read from a file
(3) it is used to allocate a buffer in heap.

All ingredients for an integer overflow vulnerability

Buffer could be made to overflow here. Resulting in a heap based exploit.
This can be used to control …
… Size written
… What is written
… Predict where objects are allocated

https://github.com/CyanogenMod/android_frameworks_av/blob/6a054d6b999d252ed87b4224f3aa13e69e3c56e0/media/libstagefright/MPEG4Extractor.cpp#L1954

Integer Overflows

```
uint64_t chunk_size = ntohl(hdr[0]);

uint8_t *buffer = new (std::nothrow) uint8_t[size + chunk_size];
```

On 32 bit platforms

widthness overflow

(chunk_size + size) is uint64_t however new takes a 32 bit value

On 64 bit platforms

arithmetic overflow

(chunk_size + size) can overflow by setting large values for chunk_size