pwnaccelerator BLOG

security and maybe more

SSHBleed - Initial Analysis

Jan 14, 2016

UPDATE: There is now an advisory from Qualys with full details: https://www.qualys.com/2016/01/14/cve-2016-0777-cve-2016-0778/openssh-cve-2016-0777-cve-2016-0778.txt

This blog post was written before the Qualys advisory was released and is based on my own analysis.

DISCLAIMER: This is a quick analysis based on a good amount of speculation and looking very quickly at some amount of unknown code. This is a 30 minute writedown and not a code audit! Take it with a grain of salt and correct me if I'm wrong.

On 2015-01-14 (his time) Damien Miller made a scary announcement on the OpenSSH development list.

He basically advised to turn off roaming entirely with "UseRoaming no". Roaming you say? In SSH? Exactly, I myself have never heard of such a feature and the documentation says: None.

I had a quick look at the code and found the following in roaming_client.c:

```
void
roaming_reply(int type, u_int32_t seq, void *ctxt)
{
        if (type == SSH2_MSG_REQUEST_FAILURE) {
                logit("Server denied roaming");
                return;
        }
        verbose("Roaming enabled");
        roaming_id = packet_get_int();
        cookie = packet_get_int64();
        key1 = oldkey1 = packet_get_int64();
        key2 = oldkey2 = packet_get_int64();
        set_out_buffer_size(packet_get_int() + get_snd_buf_size());
        roaming_enabled = 1;
}
```

Especially interesting is the line:

```
        set_out_buffer_size(packet_get_int() + get_snd_buf_size());
```

As you can see an integer is taken from the wire and added to the return value of get_snd_buf_size(). If nothing else is set, get_snd_buf_size() will return DEFAULT_ROAMBUF which is 65536.

You can spot the integer overflow here quickly as being "packet_get_int() + get_snd_buf_size()".

I was fairly quick to announce a possible find for the bug on Twitter.

Interestingly the overflow is mitigated as pointed out by @aris_ada.

There is actually a check for insane buffer sizes in set_out_buffer_size which mitigates the overflow (as we can not wrap due to the cast to size_t):

```
void
set_out_buffer_size(size_t size)
```

```
{
        if (size == 0 || size > MAX_ROAMBUF)
                fatal("%s: bad buffer size %lu", __func__, (u_long)size);
        /*
         * The buffer size can only be set once and the buffer will live
         * as long as the session lives.
         */
        if (out_buf == NULL) {
                out_buf_size = size;
                out_buf = xmalloc(size);
                out_start = 0;
                out_last = 0;
        }
}
```

However buffer size can still be farily large, up to DEFAULT_ROAMBUF (which will be important later).

As we can get unsigned values from packet_get_int() that get promoted to signed during the addition (thanks C!) we could have a pretty small buffer. Giving a negative value of -65535 would result in the value 1 being passed to set_out_buffer_size which is not caught by the check. The global variable out_buf_size is set to 1 and out_buf is allocated.

Okay so now we have a pretty small out_buf. Is that a problem? Normally it should not be, maybe it might make things slow.

Where is out_buf_size used?

Pretty often:

```
roaming_common.c:38:static size_t out_buf_size = 0;
roaming_common.c:83:            out_buf_size = size;
roaming_common.c:118:   if (count > out_buf_size) {
roaming_common.c:119:           buf += count - out_buf_size;
roaming_common.c:120:           count = out_buf_size;
roaming_common.c:122:   if (count < out_buf_size - out_last) {
roaming_common.c:129:           size_t chunk = out_buf_size - out_last;
roaming_common.c:145:           if (out_buf_size > 0)
roaming_common.c:148:   if (out_buf_size > 0 &&
roaming_common.c:169:   } else if (out_buf_size > 0 &&
roaming_common.c:204:           available = out_buf_size;
roaming_common.c:212:           atomicio(vwrite, fd, out_buf + out_buf_size - chunkend,
```

One place is the function roaming_write:

```
ssize_t
roaming_write(int fd, const void *buf, size_t count, int *cont)
{
        ssize_t ret;

        ret = write(fd, buf, count);
        if (ret > 0 && !resume_in_progress) {
                write_bytes += ret;
                if (out_buf_size > 0)
                        buf_append(buf, ret);
        }
```

It checks if there are bytes to append to some buffer and if out_buf_size is greater zero. It then calls buf_append:

```
static void
buf_append(const char *buf, size_t count)
{
        if (count > out_buf_size) {
                buf += count - out_buf_size;
                count = out_buf_size;
        }
        if (count < out_buf_size - out_last) {
```

```
                memcpy(out_buf + out_last, buf, count);
                if (out_start > out_last)
                        out_start += count;
                out_last += count;
        } else {
                /* data will wrap */
                size_t chunk = out_buf_size - out_last;
                memcpy(out_buf + out_last, buf, chunk);
                memcpy(out_buf, buf + chunk, count - chunk);
                out_last = count - chunk;
                out_start = out_last + 1;
        }
}
```

This code looks pretty ugly as pointed out by Lucas Todesco. But after all nothing found…

So I looked at the following function:

```
void
resend_bytes(int fd, u_int64_t *offset)
{
        size_t available, needed;

        if (out_start < out_last)
                available = out_last - out_start;
        else
                available = out_buf_size;
        needed = write_bytes - *offset;
        debug3("resend_bytes: resend %lu bytes from %llu",
            (unsigned long)needed, (unsigned long long)*offset);
        if (needed > available)
                fatal("Needed to resend more data than in the cache");
        if (out_last < needed) {
                int chunkend = needed - out_last;
                atomicio(vwrite, fd, out_buf + out_buf_size - chunkend,
                    chunkend);
                atomicio(vwrite, fd, out_buf, out_last);
        } else {
                atomicio(vwrite, fd, out_buf + (out_last - needed), needed);
        }
}
```

The function resend_bytes is called from the roaming_resume function in roaming_client.c:

```
        recv_bytes = packet_get_int64() ^ oldkey2;
        debug("Peer received %llu bytes", (unsigned long long)recv_bytes);
        resend_bytes(packet_get_connection_out(), &recv_bytes);
```

What it does is it will let the peer tell us how much data it received yet (oldkey2 is also read from the peer and is uint64!). It passes this to resend_bytes(…).

As we can see above resend bytes will treat this as an unsigned value “*offset” and use it to calculate “needed = write_bytes - *offset” with write_bytes being the bytes already written to the peer.

If the peer gives a value for *offset that is larger than write_bytes we have an integer underflow which will result in a value larger than write_bytes as a result!

You are probably just now thinking about: What if the peer lies to us and gives us an offset that is too great, can we “heartbleed” the buffer? Luckily there is a check:

```
        if (needed > available)
                fatal("Needed to resend more data than in the cache");
```

What is available? Available is either “available = out_last - out_start” or available is “out_buf_size”. So let’s assume it is out_buf_size. The problem: out_buf_size is also external input! It is controlled by the peer and

between 0 and MAX_ROAMBUF (2*10241*024). We can fabricate a value greater than write_bytes but still less than the maximum buffer size of MAX_ROAMBUF. And we will happily send too much data to the peer:

```
if (out_last < needed) {
                int chunkend = needed - out_last;
                atomicio(vwrite, fd, out_buf + out_buf_size - chunkend,
                    chunkend);
                atomicio(vwrite, fd, out_buf, out_last);
        } else {
                atomicio(vwrite, fd, out_buf + (out_last - needed), needed);
        }
```

Depending on what was allocated before in the SSH process all kinds of data would be leaked, from secret keys to pointer values.

This code is reached when the functions roaming_write or roaming_read fail before all data is written.

If I did not miss anything (feel free to point it out!) I think the uninitialized buffer contents of out_buf might be leaked similar to the heartbleed bug!

Interestingly while the integer overflow I spotted in roaming_reply is real, it is not the culprit because it is mitigated. The integer underflow found in resend_bytes seems to be the real deal in conjunction with the externally controllable output buffer size.

pwnaccelerator BLOG

- pwnaccelerator BLOG

security and maybe more