

Exploit for an Out of Bounds Read Vulnerability in Libssh2

Srikar Raju and Vishal Peter

Indian Institute of Science

Abstract. Over the last two decades there has been a multitude of vulnerabilities discovered and exploited in applications developed in C/C++ due to these languages being memory unsafe. In our project, we aim to exploit one such vulnerability discovered in libssh2 - a client side C library implementing the SSHv2 protocol - due to an out of bounds read (CVE 2019-13115[1]). Specifically, we aim to create a malicious SSH server, which will cause an ssh client program using libssh2 to read unintended sections in its memory and thereby trigger a crash.

Project Link: <https://github.com/viz27/Libssh2-Exploit>

1 Introduction

1.1 Brief Overview on SSH, OpenSSH and libssh2

Secure Shell (SSH)[2] is a cryptographic network protocol, which provides a secure channel over an unsecured network in a clientserver architecture, connecting an SSH client application with an SSH server. Typical applications include remote command-line, login, and remote command execution. It recommends Diffie-Hellman as Key exchange protocol to securely share a key between client and server.

OpenSSH[3] is a suite of secure networking utilities based on the Secure Shell (SSH) protocol. It implements both the server side as well as client side functionalities of SSH.

libssh2[4] is a client-side library that implements SSH2 protocol. It is used to remotely execute commands, transfer files etc., It serves as an alternative to OpenSSH, libssh. As libssh2 only implements the client side functionality of SSH, it is light weight compared to others.

In this project, we have used libssh2 on client side and OpenSSH at server end.

1.2 Diffie-Hellman Key Exchange SHA-256 (RFC-4419)

Diffie Hellman Key exchange is a means for two parties to agree upon a shared secret in such a way that the secret will be unavailable to eavesdroppers. This secret may then be converted into cryptographic keying material for other (symmetric) algorithms.

There are multiple protocols specified by IETF to implement this and a widely used one is Diffie-Hellman Key Exchange SHA-256[5]. We briefly describe how it works below.

The exchange starts with the client requesting a modulus(p) from the server indicating the preferred size. In the following description (C is the client, S is the server, the modulus p is a large safe prime, and g is a generator for a subgroup of $GF(p)$, min is the minimal size of p in bits that is acceptable to the client, n is the size of the modulus p in bits that the client would like to receive from the server, max is the maximal size of p in bits that the client can accept, V_S is S 's version string, V_C is C 's version string, K_S is S 's public host key, I_C is C 's KEXINIT message, and I_S is S 's KEXINIT message that has been exchanged before this part begins)

1. C sends $min||n||max$ to S , indicating the minimal acceptable group size, the preferred size of the group, and the maximal group size in bits the client will accept.

2. S finds a group that best matches the client's request, and sends $p||g$ to C .

3. C generates a random number x , where $1 < x < (p-1)/2$. It computes $e = g^x \text{ mod } p$, and sends " e " to S .

4. S generates a random number y , where $0 < y < (p-1)/2$, and computes $f = g^y \text{ mod } p$. S receives " e ". It computes $K = e^y \text{ mod } p$, $H = \text{hash}(V_C||V_S||I_C||I_S||K_S||min||n||max||p||g||e||f||K)$, and signature s on H with its private host key. S sends $K_S||f||s$ to C . The signing operation may involve a second hashing operation.

5. C verifies that K_S really is the host key for S (e.g., using certificates or a local database to obtain the public key). C is also allowed to accept the key without verification; however, doing so will render the protocol insecure against active attacks (but may be desirable for practical reasons in the short term in many environments). C then computes $K = f^x \text{ mod } p$, $H = \text{hash}(V_C||V_S||I_C||I_S||K_S||min||n||max||p||g||e||f||K)$, and verifies the signature s on H .

2 Vulnerability Description

The vulnerability is a potential out of bounds read which occurs in libssh2 code. During the Diffie Hellman Key Exchange protocol, the client receives the prime number p and generator g from server(step 2 of the key exchange protocol described above). As p and g can be very large numbers(upto 8K bits), the server sends the length of these numbers as well along with p and g . ie. The client receives the data which looks like this $[p_len, p, g_len, g]$ from server in a buffer. It first reads p_len to get the length of the number p and then extracts p_len bits from the buffer to get the value of p . It reads g as well in a similar fashion(p and g will be stored as BIGNUM data type, as specified by OpenSSL[6]). However the issue is that libssh2(up to version 1.8.2), after reading the value of p_len , goes ahead to read p_len bytes from the buffer without performing any sanitization even though p_len was an untrusted value received from the server side.

2.1 Version 1.8.2

```

if (key_state->state == libssh2_NB_state_sent1) {
    unsigned char *s = key_state->data + 1;
    p_len = _libssh2_ntohu32(s);
    s += 4;
    _libssh2_bn_from_bin(key_state->p, p_len, s);
    s += p_len;

    g_len = _libssh2_ntohu32(s);
    s += 4;
    _libssh2_bn_from_bin(key_state->g, g_len, s);
}

```

Fig. 1. Vulnerable code in released version 1.8.2

Fig. 1 above shows the vulnerability in the released version 1.8.2 of libssh2. The data $[p_len, p, g_len, g]$ is received in the buffer named *s*. First it reads the value of *p_len* from the buffer (done by the function `_libssh2_ntohu32()`, which reads the first 4 bytes from the buffer and converts it into an unsigned 32 bit integer). Then it reads *p_len* bytes from the buffer to extract *p* value and store it in `key_state->p` (done by the function `_libssh2_bn_from_bin()`). If the *p_len* value received from the server is incorrect, say a very large value - the function may read from outside the bounds of the buffer. On making *p_len* value sufficiently large, the client program will eventually try to read from unallocated memory locations and crash with a segmentation fault.

2.2 An insufficient fix by libssh2 team

A fix was introduced for this issue by adding bounds check on the length fields received from the server. However even that had the same vulnerabilities, even though it is not clear on a first glance. In Fig. 2 the new function introduced to perform the bounds check is shown. The buffer from which *p* will be read is passed as argument *buf*, along with argument *p_len* - which is the number of bytes to be read. *buf* has 2 additional fields *buf->data* : a pointer to the start of the entire buffer and *buf->dataptr* : a pointer to the point from which data will be begun to be read. The function tries to ensure that the after reading *p_len* bytes from the buffer beginning from *buf->dataptr*, the pointer should not go beyond the bounds of the buffer. ie ensure $p_len \leq buf->len - (buf->dataptr - buf->data)$. or $buf->dataptr - buf->data \leq buf->len - p_len$ (). This check is performed in the function `_libssh2_check_length()`

```

int _libssh2_check_length(struct string_buf *buf, size_t p_len)
{
    return ((int)(buf->dataptr - buf->data) <= (int)(buf->len - p_len)) ? 1 : 0;
}

```

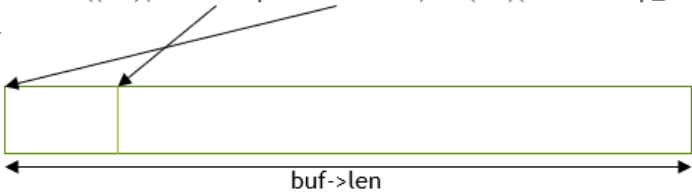


Fig. 2. Vulnerable code in the proposed fix for the problem

The issue in this is that `buf->len` and `p_len` are unsigned integers by definition and their difference is being typecast to a signed int in the expression `(int)(buf->len - p_len)`. Typically when `p_len` is larger than `buf->len`, their difference will lead to integer overflow and the value will end up being a large unsigned positive number, which when cast to signed int will become a negative number and the check will fail. However, on choosing a sufficiently large `p_len` value, we can make the value of `buf->len - p_len` a large positive number which when typecast to signed int will still remain a positive number, causing the check to falsely succeed. In other words, even with the introduction of this bounds check, we can specify large values in the `p_len` field and get a success from the bounds check function - which will again trigger an out of bounds read just like in the previous scenario.

The CVE(2019-13115) was based on this badly written code segment in the development branch. The exploit we created works on both the released version 1.8.2 as well as the development branch code mentioned in the CVE.

3 Attack model

The attack is aimed at applications which uses libssh2 to implement the ssh client side functionality. The attacker here is someone who is able to compromise a legitimate SSH server(which the client is trying to connect to) or someone who can masquerade as the legitimate server that the client is expecting. The attacker will be able to cause a crash(denial of service) at the client side when it tries to connect to the server.

To exploit the vulnerability, we created a malicious ssh server which on receiving connect requests from client side, sends a malicious value in the `p_len` field in the second step of the key exchange protocol. This was done by modifying OpenSSH source code to include the changes needed for the exploit. When a client program tries to connect to our malicious server, it receives this malicious value in the `p_len` field which in turn will trigger an out of bounds read leading to a crash. SSH provides client and server applications the functionality to authenticate each other(eg. using public keys). But this typically happens towards

the end of the key exchange. ie Since this vulnerability occurs in the very initial stages of Diffie Hellman Key Exchange protocol, the client would die even before verifying the authenticity of the server it is connecting to. We tested our exploit by running several example programs included in libssh2 development suite. All these programs would first try to establish an SSH connection with the server and we were able to observe the crash occurring every time.

We explored the possibilities to exploit this further to retrieve confidential data from the client side. This is theoretically possible as a carefully crafted `p_len` value will cause the client to read data outside the buffer bounds and store it in `key_state->p`, ie the `p` value stored at client side. And the memory outside the buffer bounds may contain data such as passwords, cryptographic keys etc. ie If the server can convince the client to send the value of `key_state->p` to the server, it will result in a data breach.

However with this vulnerability alone and without making any code changes at the client side, there doesn't seem to be a way to do this. In the key exchange phase `p` and `g` values are sent from the server to the client, but never back to the server.

4 Conclusion

We have presented a detailed analysis of the CVE-2019-13115 and a way to exploit it to cause a denial of service at the client. In the later releases of libssh2, this has been fixed. Most such applications/libraries developed in C/C++ are prone to such vulnerabilities due to the lack of memory safety. However because of the performance benefit offered by C/C++ and the difficulty in porting legacy code, they are still very much in use. And more vulnerabilities of the same flavour keep on being discovered.

5 Acknowledgement

We would like to thank Prof. Vinod Ganapathy for providing us the opportunity, to work on this project and guiding us through his comments and suggestions during the course of this project. We would like to acknowledge Kevin Backhouse, who reported this vulnerability and wrote an excellent blog[7] about this CVE that helped us in understanding it better.

References

1. (December 6, 2019), <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-13115>
2. (December 6, 2019), https://en.wikipedia.org/wiki/Secure_Shell
3. (December 6, 2019), <https://www.openssh.com/>
4. (December 6, 2019), <https://www.libssh2.org/>
5. (December 6, 2019), <https://www.ietf.org/rfc/rfc4419.txt>
6. (December 6, 2019), <https://www.openssl.org/>
7. (December 6, 2019), <https://blog.semmle.com/libssh2-integer-overflow/>