Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Pentest-Report libssh C Library 09.2019

Cure53, Dr.-Ing. M. Heiderich, MSc. N. Krein, Prof. N. Kobeissi, MSc. D. Weißer, BSc. J. Hector, B. Walny

Index

Introduction

"libssh is a multiplatform C library implementing the SSHv2 protocol on client and server side. With libssh, you can remotely execute programs, transfer files, use a secure and transparent tunnel, manage public keys and much more ..."

From https://www.libssh.org/

This report documents the findings of a security assessment targeting the libssh software. Carried out by Cure53 in September and October 2019, this project entailed both a penetration test and a source code audit. The focus was placed on the libssh software, which is available as open source and currently shipped in version v0.9. Fourteen security-relevant findings, including one marked as "*Critical*" in terms of impact, have been spotted by Cure53 on the scope.

It needs to be clarified that this assessment was generously funded by Mozilla. The budget stemmed from the framework of the Mozilla Open Source Support (MOSS) initiative, specifically the Secure Open Source funding track. Cure53 was introduced to the libssh team and its maintainers by Mozilla. From there, the assessment was planned collaboratively. Resources-wise, Cure53 approached this assessment of libssh with a team of six senior testers. The total time invested into the completion of this examination came up to thirty-two person-days. All work - spanning core tests, documentation, reporting and write-up - was done in late September and early October 2019.

In order to fulfill the objectives of this assessment in an organized fashion, three work packages (WPs) have been created. WP1 encompassed the libssh source code audit, during which Cure53 focused on investigating the available sources with the overarching goal of identifying vulnerable sections. Code-assisted penetration testing, reliant on the use of the reference server and client implementations, as well as local setups created by the Cure53, has been performed during WP2. Finally, WP3 centered on libssh protocol Fuzzing and formal verification using AFL. The latter partly happened in parallel to the test and was also accompanied by formal verification with the ProVerif tool. To clarify the goal, it should be stated that Cure53 sought to find out whether the protocol handshakes are as secure as they are supposed to be.

To prepare for the test, several briefing meetings were held. In addition, the libssh team supplied Cure53 with a scope document with instructions pertinent to the test setup, interesting areas and general info about the expectations regarding the test and assessment's results. Consequently, the project started on time and progressed efficiently. Communications during the assessment took place in an IRC channel on Freenode, which was made available by the maintainers of the libssh project. Cure53 furnished regular status updates and shared relevant findings as they emerged rather than upon waiting for the final completion of this report.

Among the spotted fourteen findings, eight were classified as security vulnerabilities and six were noted as general weaknesses with lower exploitation potential. While one item received a "*Critical*" marker, it must be noted that its exploitability does not extend to every piece of software that utilizes the libssh API. Nevertheless, it was demonstrated that popular software such as *cURL* is prone to the proposed attack scenario, thus causing the vulnerability to be marked as higher than originally proposed. Quite impressively, all other problems had only "*Medium*" and lower grades, indicating a good overall outcome of this Cure53 assessment of the libssh software.

In the following sections, the report will first briefly reiterate the scope and three specific WPs. It then moves on to dedicated, chronologically discussed tickets, which shed light on the discoveries one-by-one. Alongside technical aspects like PoCs, Cure53 furnishes mitigation advice for improving libssh going forward. The report closes with a broader conclusion in which Cure53 summarizes this autumn 2019 project and issues a verdict about the tested scope. Further, detailed recommendations and notes on the security and privacy posture of the libssh software are supplied in the final section of this document.

Scope

- **libssh server- and client-side code**
  - **WP1**: libssh source code audits
    - https://www.libssh.org/files/0.9/
  - **WP2**: libssh penetration tests, code-assisted (using reference server & client)
  - **WP3**: libssh protocol fuzzing & formal verification (using AFL & ProVerif)
  - A detailed scope document was made available to Cure53 by the libssh maintainers
  - Build instructions were provided

Identified Vulnerabilities

The following sections list both vulnerabilities and implementation issues spotted during the testing period. Note that findings are listed in chronological order rather than by their degree of severity and impact. The aforementioned severity rank is simply given in brackets following the title heading for each vulnerability. Each vulnerability is additionally given a unique identifier (e.g. *SSH-01-001*) for the purpose of facilitating any future follow-up correspondence.

SSH-01-003 Client: Missing *NULL* check leads to crash in erroneous state *(Low)*

It was discovered that a *crypto* function does not properly check for a *null* pointer, allowing for a malicious server to cause a *libssh*-based client application to crash. In certain scenarios, this can lead to information leakage, for instance when core dumps are written and made available to the attacker.

When an SSH client establishes a connection, the server sends back its public key. In case this key cannot be parsed correctly, the *libssh* client cannot setup the required *crypto* parameters. As such, it causes the *ssh_connect()* function to return an error. If the *ssh_disconnect()* function is called in this erroneous state, the client crashes while encrypting the disconnect-message. This is because the required data structs have not been initialized. The following code snippet shows where the crash occurs. It can be seen that *out_cipher* is not initialized, causing a Null Pointer Dereference upon access to *out_cipher->blocksized* being attempted.

Affected File:
libssh-0.9.0/src/packet.c

Affected Code:
```
static int packet_send2(ssh_session session)
{
   [...]
    crypto = ssh_packet_get_current_crypto(session, SSH_DIRECTION_OUT);
    if (crypto) {
        blocksize = crypto->out_cipher->blocksize;
        lenfield_blocksize = crypto->out_cipher->lenfield_blocksize;
```

Although the library returns an error when the connection fails, no crash should occur when the *disconnect* function is called in this state. Upon further investigation, it was found that the issue affects applications like cURL when compiled with *libssh*. It is recommended to check the *out_cipher* pointer against a *NULL* value.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

SSH-01-004 SCP: Unsanitized location leads to command execution (*Critical*)

When the *libssh* SCP client connects to a server, the *scp* command, which includes a user-provided path, is executed on the server-side. In case the library is used in a way where users can influence the third parameter of *ssh_scp_new(),* it would become possible for an attacker to inject arbitrary commands, leading to a compromise of the remote target. As shown in the following snippet, the provided location is passed directly without prior escaping.

Affected File:
libssh-0.9.0/src/scp.c

Affected Code:
```
int ssh_scp_init(ssh_scp scp)
{
        [...]
        if(scp->mode == SSH_SCP_WRITE)
                snprintf(execbuffer,sizeof(execbuffer),"scp -t %s %s",
                scp->recursive ? "-r":"", scp->location);
        else
                snprintf(execbuffer,sizeof(execbuffer),"scp -f %s %s",
                scp->recursive ? "-r":"", scp->location);
        if(ssh_channel_request_exec(scp->channel,execbuffer) == SSH_ERROR){
```

Dangerous Library Call:
```
ssh_scp scp = ssh_scp_new(session, SSH_SCP_READ, [USER CONTROLLED]);
ssh_scp_init(scp);
```

Although this issue requires an application to pass unsanitized user-input to the *libssh* API, a setup where user-controlled data reaches that sink is realistic. This especially holds because the *libssh* documentation[1] does not mention any form of security risk when supplying the *location* parameter.

Further analysis of the dangerous usage of the affected API call also demonstrated the vulnerability of the *libcurl* compiled with *libssh* support.

Proof-of-Concept with *libcurl* + “*--with-libssh*”:
```
$ curl -u user scp://localhost:"/etc/passwd;touch /tmp/xxx"
```

[1] http://api.libssh.org/stable/group__libssh__scp.html#ga9fcd39a2bb6438e39cf19ff859dc2f2e

Proof-of-Concept with PHP + *libcurl* + "*--with-libssh*":

```
$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, "scp://localhost/;touch /tmp/bla789");
curl_setopt($ch, CURLOPT_USERPWD, "user:pass");
curl_exec($ch);
```

Therefore, it is recommended to properly escape the *location* parameter and place it between single quotes. This should also solve issues with paths containing certain characters.

SSH-01-006 General: Various unchecked *Null-derefs* cause DOS *(Low)*

While applying Semmle's extended taint tracking queries, it was possible to notice a few *null pointer dereferences* resulting from unchecked function calls. The following Semmle query indicates the potential results in full.

Used Semmle Query:

```
import cpp

from VariableAccess access
where maybeNull(access) and  dereferenced(access)
select access, "Potential Nullderef?"
```

LGTM Link:
https://lgtm.com/query/8016580153808805310/

The list below enumerates a few noteworthy findings that explain the underlying problem. Whenever one of the highlighted functions is called, its return value is left unchecked. This results in the variable on the left side of the assignment being uninitialized or *NULL*. The next dereference thus accesses the *NULL* pointer, causing a segmentation fault and subsequent crash of the application reliant on the underlying codepath.

Noteworthy Findings:

- *libssh-0.9.0/src/gssapi.c:*
```
for (i=0; i<n_oids; ++i){
    oids[i] = ssh_string_new(selected->elements[i].length + 2);
    ((unsigned char *)oids[i]->data)[0] = SSH_OID_TAG;
```
- *libssh-0.9.0/src/messages.c:*
```
crypto = ssh_packet_get_current_crypto(session, SSH_DIRECTION_IN);
[...]
rc = ssh_buffer_pack(buffer,
                     "dPbsssbsS",
                     crypto->digest_len,
```

- *libssh-0.9.0/src/packet_crypt.c:*
  ```
  crypto = ssh_packet_get_current_crypto(session, SSH_DIRECTION_IN);
  cipher = crypto->in_cipher;
  ```

It is recommended to go through the Semmle findings above and ensure that the returned values of all function calls are correctly checked. They also need to be accordingly bailed out of the control flow whenever the invocation fails.

SSH-01-007 PKI Gcrypt: Potential UAF/double free with RSA pubkeys *(Medium)*

The *libssh*'s memory allocation functionalities provide a macro called *SAFE_FREE*. Upon freeing a pointer, this macro sets the pointer variable to *NULL*. However, it is still possible to have a pointer variable be non-*NULL* should the macro be used in a wrapper function. One such wrapper function is *ssh_string_free* whereas the caller of that function retains a pointer of non-*NULL*. This can lead to double-free or Use-after-Free (UAF) vulnerabilities.

Below is a code excerpt that shows a potential double-free vulnerability and highlights its relevant parts.

Affected File:
libssh-0.9.0/src/pki_gcrypt.c

Affected Code:
```
ssh_string pki_publickey_to_blob(const ssh_key key)
{
[...]
    ssh_string e = NULL;
    ssh_string n = NULL;
[...]
    switch (key->type) {
[...]
        case SSH_KEYTYPE_RSA:
[...]

            ssh_string_burn(e);
            ssh_string_free(e);
            ssh_string_burn(n);
            ssh_string_free(n);

            break;
        case SSH_KEYTYPE_ED25519:
[...]
    }
```

```
makestring:
    str = ssh_string_new(ssh_buffer_get_len(buffer));
    if (str == NULL) {
        goto fail;
    }

    rc = ssh_string_fill(str, ssh_buffer_get(buffer),
ssh_buffer_get_len(buffer));
    if (rc < 0) {
        goto fail;
    }
    ssh_buffer_free(buffer);

    return str;
fail:
    ssh_buffer_free(buffer);
    ssh_string_burn(str);
    ssh_string_free(str);
    ssh_string_burn(e);
    ssh_string_free(e);
[...]
    ssh_string_burn(n);
    ssh_string_free(n);

    return NULL;
}
```

After handling a particular case for the given *switch* statement, the used pointers *e* and *n* are "freed" by the *ssh_string_free* in use. However, should one of the two functions executed after the *switch* statement fail, then the same pointers are liberated again, thus leading to a double-free.

It is recommended to explicitly set the pointer variables to *NULL* after calling *ssh_string_free*, as it is done in a similar function defined in *src/pki_crypto.c*.

SSH-01-010 SSH: Deprecated hash function in fingerprinting *(Low)*

The SSH protocol standard currently defines a method for out-of-band public key authentication through fingerprints. This is done through *RFC4251*[2], which cites FIPS-180-2, and is represented as hexadecimal-encoded bytes of a SHA-1 hash on the public key, as well as on the related identity information. This out-of-band mutual authentication method is also supported as default on a large share of SSH implementations, outside of *libssh*.

[2] https://tools.ietf.org/html/rfc4251

While no flagrant security issue exists as a result of using *SHA-1* as a fingerprint hash function, it is noted that *SHA-1* is currently in the process of being deprecated across virtually all major Internet protocols, most notably TLS (both on the protocol[3] and certificate[4] levels). Furthermore, many practical attacks have been uncovered on *SHA-1,* rendering collisions feasible[5][6].

For these reasons, it is recommended to deprecate *SHA-1'*s usage in *libssh* as much as it is possible. Unfortunately, this is constrained by notions of cross-compatibility with other SSH implementations, and as such may not be achievable without coordinating across other major implementations in the SSH ecosystem.

SSH-01-011 SSH: Lack of point validation on *X25519* and *Ed25519* (Medium)

libssh implements an extension to *RFC4253* that allows for the usage of more modern Diffie-Hellman and signature primitives, namely *X25519* and *Ed25519*, during the key exchange phase. However, these primitives are implemented in a way that no point/key validation occurs during the scalar multiplication step. In the case of *Curve25519*, this opens the possibility for small subgroup attacks, while posing a variety of malleability issues in *Ed25519*.

Practical attacks based on these weaknesses have been demonstrated in two recent papers by Cas Cremers et al, in particular in *Section 3.4* of the first paper[7], and *Section 7.1* of the second one[8]. Considerable effort was made during the timeframe of this audit to making a determination on whether similar issues apply to the Diffie-Hellman key exchange described in *Section 8* of *RFC4253*. So far, no definitive confirmation for any flaws could be obtained. However, this might entirely be due to the need for more research on the requirements for obtaining a colliding signature in *Ed25519*. Other parts of the analysis allude to more certainty. For example, it seems likely that there may be a reduction in forward-secrecy if a low order subgroup is forced.

If it is possible to obtain a colliding signature without modifying or Man-in-the-Middling the server's public key (as *Section 3.4* of the paper cited above strongly suggests), then there may be a non-negligible chance that using *Curve25519* and *Ed25519* for some protocol executions will result in a degradation of security for all of SSH, not just *libssh*. Coupled with the potential of forcing certain cipher-suites in SSH implementations via an active attack, this can become a serious problem if these faults are confirmed.

[3] https://tools.ietf.org/id/draft-lvelvindron-tls-md5-sha1-deprecate-01.html
[4] https://security.googleblog.com/2014/09/gradually-sunsetting-sha-1.html
[5] https://shattered.io/static/shattered.pdf
[6] https://sites.google.com/site/itstheshappening/
[7] https://eprint.iacr.org/2019/779.pdf
[8] https://eprint.iacr.org/2019/526.pdf

Given that not enough time within the scope of the audit permits research into confirming such a sophisticated attack, this issue is filed in order to strongly encourage point validation in *Curve25519.* In doing so, the potential attack would be avoided regardless. It is noted that performing point validation on both primitives does not break interoperability with other SSH implementations - except for cases of extremely low probability. This means that the approach can be deployed without creating any issues.

SSH-01-013 Conf-Parsing: Recursive wildcards in hostnames lead to DOS *(Low)*

While fuzzing *parse_config.c* and *knownhosts.c* it was found that the use of wildcard characters such as “*” (asterisk) cause a recursion in the underlying parsing logic, ultimately resulting in a Denial-of-Service. The root cause of this issue is that the *match_pattern* function in *match.c* tries to recursively match sub-patterns against the hostname if two wildcard characters ensue. Two PoCs for the affected file and lines of code are shown below.

PoC 1 *(known_hosts file)***:**
```
**************************cure53
```

PoC 2 *(ssh config file):*
```
Host ******************************cure53
```

Affected File:
libssh-0.9.0/src/match.c

Affected Code:
```
static int match_pattern(const char *s, const char *pattern) {
  [..]

  for (;;) {
       [..]
       /*
       * Move ahead one character at a time and try to
       * match at each position.
       */
       for (; *s; s++) {
              if (match_pattern(s, pattern)) {
                     return 1;
              }
       }
       [..]
```

It is recommended to either write the code in an iterative fashion, or to make use of static counters which track the recursion's depth, thus bailing out if the depth is too great. Sensible values have to be found for that counter.

SSH-01-014 Conf-Parsing: Integer underflow leads to OOB array access *(Low)*

While fuzzing the parsing of SSH *config* files, it was found that empty lines, i.e. lines with nothing but a *null* byte, lead to an integer underflow. In turn, this signifies an out-of-bounds array access. The affected file and code are shown below.

Affected File:
libssh-0.9.0/src/config.c

Affected Code:
```
static int ssh_config_parse_line(..,const char *line,..)
{
  size_t len;
  char *s = NULL, *x = NULL;
  x = s = strdup(line);
  [...]
  /* Remove trailing spaces */
  for (len = strlen(s) - 1; len > 0; len--) {
       if (! isspace(s[len])) {
       break;
       }
       s[len] = '\0';
  }
```

In order to catch such edge cases, it is recommended to check if the supplied line has a length greater than 0, or to refactor the loop initialization.

Miscellaneous Issues

This section covers those noteworthy findings that did not lead to an exploit but might aid an attacker in achieving their malicious goals in the future. Most of these results are vulnerable code snippets that did not provide an easy way to be called. Conclusively, while a vulnerability is present, an exploit might not always be possible.

SSH-01-001 State Machine: Initial machine states should be set explicitly *(Info)*

Throughout the codebase, different variables are used for tracking different states. This includes the overall session state, i.e. being in the authentication state or the session being authenticated. Another example is a variable that keeps track of the state for handshakes.

It was discovered that whenever these variables are first declared, no initial state is explicitly set. Rather, the initial state is implicitly imposed due to the fact that allocated memory is initialized with *null* bytes upon allocation.

This did not lead to any issues during the test. However, explicitly setting these variables to their initial state aids in the overall understanding of the code and can prevent issues where a certain state is assumed but not given. It is therefore recommended to be more explicit about this matter by setting initial states upon declaration.

SSH-01-002 Kex: Differently bound macros used to iterate same array *(Info)*

When reviewing the handler functions for the different SSH messages, it was discovered that two different macros are used as upper bounds when iterating over the same array. For example, two for-loops iterate an array called *strings*, while the first does so using the upper bound *KEX_METHODS_SIZE* and the *SSH_KEX_METHODS* second time around. This can potentially lead to an out-of-bound read/write, since both bounds are defined separately from one another. The code excerpt below shows the two iterations and highlights the relevant parts.

Affected File:
libssh-0.9.0/src/kex.c

Affected Code:
```
SSH_PACKET_CALLBACK(ssh_packet_kexinit)
{
[...]
    char *strings[KEX_METHODS_SIZE] = {0};
[...]
    for (i = 0; i < KEX_METHODS_SIZE; i++) {
[...]
```

```
            strings[i] = ssh_string_to_char(str);
            if (strings[i] == NULL) {
                ssh_set_error_oom(session);
                goto error;
            }
            ssh_string_free(str);
            str = NULL;
        }

        /* copy the server kex info into an array of strings */
        if (server_kex) {
            for (i = 0; i < SSH_KEX_METHODS; i++) {
                session->next_crypto->client_kex.methods[i] = strings[i];
            }
    [...]
```

During this test both upper bounds were defined to be the same size, thus not causing any issues. However, should one of these defines change with future changes to the code, an out-of-bound issue could emerge.

It is therefore recommended to either use the same upper boundary for both loop iterations, or to make these two definitions interdependent. For example, let *KEX_METHODS_SIZE* always be the same as *SSH_KEX_METHODS*.

SSH-01-005 Code-Quality: Integer sign confusion during assignments *(Low)*

During more general source code analysis, it was noticed that variable declarations and assignments often change signs. This often leads to unexpected control flow, especially when the conversion happens implicitly. To spot all potential issues where a signed integer gets converted to an unsigned one (and the other way around), the following Semmle query was used.

Semmle Query:
```
import cpp

class Signed extends IntType { Signed() { this.isSigned() }}
class Unsigned extends IntType { Unsigned() { this.isUnsigned() }}

from AssignExpr e, Signed s, Unsigned u
where (
        (e.getLValue().getUnspecifiedType() = s and
        e.getRValue().getUnspecifiedType() = u) or
        (e.getLValue().getUnspecifiedType() = u and
        e.getRValue().getUnspecifiedType() = s) and not
    (e.getLValue().isConstant() or e.getRValue().isConstant())
)
```

```
select e.getRValue(), "Accidental signed/unsigned int conversion? " +
e.getLValue() + " = " + e.getRValue()
```

LGTM Results:
https://lgtm.com/query/6155169963772014869/

Noteworthy Findings:

- *libssh-0.9.0/src/sftp.c (count is signed, sftp->ext->count is unsigned):*
  `int count = sftp->ext->count;`
- *ibssh-0.9.0/src/sftpserver.c (val is unsigned, i i signed):*
  `val = i;`
- *ibssh-0.9.0/src/sftp.c (ssh_buffer_get_len returns unsigned, packetlen is signed):*
  `packetlen=ssh_buffer_get_len(buffer);`

Although none of the findings mentioned above and in the LGTM link result in security vulnerabilities, they are still the result of a bad coding practice that can lead to bugs in the future. It is recommended to go through each of the Semmle findings above and make sure implicit casts do not happen. This can be accomplished by declaring the left-hand side variable with the type it is supposed to hold.

SSH-01-008 SCP: Protocol Injection via unescaped File Names *(Low)*

It was discovered that file names are not properly encoded when used in the *SCP* protocol messages. If unsanitized user-input is passed to the related library functions, an attacker can inject arbitrary *SCP* protocol messages and create arbitrary files in the current SCP destination. The following code snippets show how file-names are included in the *SCP* protocol. Besides extracting the path's basename, no other sanitization is carried out.

Affected Code:
```
int ssh_scp_push_directory(ssh_scp scp, const char *dirname, int mode){
[...]
    dir=ssh_basename(dirname);
    perms=ssh_scp_string_mode(mode);
    snprintf(buffer, sizeof(buffer), "D%s 0 %s\n", perms, dir);

[...]
int ssh_scp_push_file64(ssh_scp scp, const char *filename, uint64_t size, int
mode){
[...]
    file=ssh_basename(filename);
    perms=ssh_scp_string_mode(mode);
    SSH_LOG(SSH_LOG_PROTOCOL,"SCP pushing file %s, size %" PRIu64 " with
permissions '%s'",file,size,perms);
    snprintf(buffer, sizeof(buffer), "C%s %" PRIu64 " %s\n", perms, size, file);
```

It is recommended to follow the same approach as OpenSSH and to encode paths using the *vis-encoding*[9].

SSH-01-009 SSH: *RFC4255* not Implemented *(Info)*

It was discovered that *RFC4255*[10], which describes implementing SSH fingerprint verification through DNS record checks, is not supported in *libssh* despite it appearing on the *libssh* list of the supported Internet standards[11].

It is recommended to either implement the standard or to mark it as not currently supported by *libssh*, as the suggested functionality does not occur at any point of the user-flow.

SSH-01-012 PKI: Information leak via uninitialized stack buffer *(Low)*

It was discovered that during the decryption of private keys a stack buffer is not initialized, thus leading to the potential leakage of information. In order to read the password into the buffer, the *auth_fn()* callback function is called and basically equals a user-defined callback function. In the example applications, this callback just passes the buffer to *ssh_getpass()*.

Affected File:
libssh-0.9.0/src/pki_container_openssh.c

Affected Code:
```
static int pki_private_key_decrypt([...])
{
    [...]
    char passphrase_buffer[128];
    [...]
    if (passphrase == NULL) {
        [...]
        rc = auth_fn("Passphrase",
                     passphrase_buffer,
                     sizeof(passphrase_buffer),
                     0,
                     0,
                     auth_data);
        [...]
        passphrase = passphrase_buffer;
    }
```

[9] https://linux.die.net/man/3/vis
[10] https://tools.ietf.org/html/rfc4255
[11] http://api.libssh.org/stable/

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

The *ssh_getpass()* function disables the display of *stdin* and calls *ssh_gets()* where the provided buffer is printed in case its first byte is not zero. Since the buffer is not initialized by *pki_private_key_decrypt(),* it might happen that stack contents are leaked.

Affected File:
libssh-0.9.0/src/getpass.c

Affected Code:
```
static int ssh_gets(const char *prompt, char *buf, size_t len, int verify) {
    [...]

    /* read the password */
    while (!ok) {
        if (buf[0] != '\0') {
            fprintf(stdout, "%s[%s] ", prompt, buf);
        } else {
            fprintf(stdout, "%s", prompt);
        }
```

It is not safe to rely on the user to properly initialize the buffer in the provided callback function. Therefore, it is recommended to prevent the potential leakage of stack bytes by initializing the passphrase buffers in *pki_private_key_decrypt()* and *pki_private_key_encrypt()* with zeroes.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Conclusions

As noted in the *Introduction,* this assessment of the libssh software concludes with positive results. After spending thirty-two days on the scope in September and October of 2019, six senior members of the Cure53 team can confirm that the libssh software is mostly secure and generally free from major risks. At the same time, the testing demonstrated vulnerabilities across all scope items and Work Packages, signaling that some improvements and more granular approaches can be considered and deployed by the maintainers of the libssh software.

It should be noted that the generous funding from the Mozilla MOSS project made this assignment possible. Further, thanks to the libssh team's excellent preparation and availability throughout the test, Cure53 was able to efficiently work on the codebase, providing and receiving thorough feedback.

Despite the large functionality and a rather large attack surface, Cure53 found the codebase of libssh to be surprisingly clean, easy to understand and audit. Only one exception should be made in this realm, namely as regards the libssh's confusing state machine that already exposed the project to vulnerabilities in the past. During this assessment, Cure53 also spent considerable effort on auditing all of the control flow possibilities spread throughout the entire source code. Even though the general feeling shared by the Cure53 team is that the state machine is now securely implemented, it is still highly recommended to devise and offer a cleaned-up and centralized rewrite.

At first glance, it appears that the number of fourteen findings is quite excessive. However, the findings must be read in context and, notably, an aggregation of risk still results in a rather low criticality score. A vast majority of problems must be evaluated as minor, low-impact mistakes. Nevertheless, some noteworthy findings concern SSH-01-004, which is a command injection via *scp*, as well as SSH-01-007, which demonstrates a potentially exploitable double free in key parsing. While these discoveries would be a hard target for the attackers to hit, especially SSH-01-004 is worth-fixing since major applications out there insecurely implement the scenario responsible for this flaw.

Beyond the above, Cure53 also spotted a number of DOS issues that are mostly a result of some unclean code patterns. It has been established, for example, that libssh is riddled with unchecked function calls and integer sign confusions. Cure53 highly recommends going through all Semmle queries and fixing these errors one-by-one before they result in actually exploitable conditions in the future.

Regarding the libssh's cryptographic implementations, Cure53 actually had no serious concerns upon review. Almost all specifications (with the exception of RFC4255, documented in SSH-01-009) are standards-compliant. Cryptographic primitives used by libssh were sound and resistant to side-channel attacks. However, one minor protocol-level issue was identified (see SSH-01-010) and relates to the usage of deprecated hash functions. It should be noted that this issue is not specific to libssh and likely cannot be avoided without breaking compatibility with other SSH clients. Therefore, coordinating across all SSH libraries in order to mitigate this issue is hereby encouraged. Finally, in SSH-01-011, Cure53 describes a proposal for implementing point validation on *Curve25519*-based cryptographic primitives. This is driven by the goal of avoiding a number of issues that might be valid and serious but could not be confirmed in full due to time constraints imposed on this audit, as well as their degree of intricacy.

All in all, the Cure53 team has gained a positive impression about the libssh software. Praise needs to be extended to the development team and applies to the entire process of this audit, starting with the preparatory phase and including communications and follow-up work. Clearly, some bugs and security concerns can be spotted in the codebase of libssh, yet Cure53 is generally confident about the tested software's reliability and its well-written codebase. It is vital that the documented issues can be resolved rather easily, particularly as no major logic errors compromising the entire library could be unveiled. Conclusively, the libssh software has been evaluated as secure during this Cure53 September-October 2019 assessment.

Cure53 would like to thank Jochai Ben-Avie of Mozilla for his excellent project coordination, support and assistance, both before and during this assignment. Cure53 would further like to express gratitude to Andreas Schneider of Red Hat as well as the rest of the maintainer team who aided the assignment with valuable advice and input.