A gentle technical analysis of CVE-2018-10933
TLP:GREEN

blueintel

October 19, 2018

Abstract

In this document we analyze the vulnerability described in CVE-2018-10933 in the context of two example SSH server programs, which are shipped with the `libssh` library. We prove that the threat is real and demonstrate how the approach to programming an SSH server with `libssh` affects the extent of exploitation.

Program 1: SSH server

The first of the two programs we analyze is a simple demo SSH server, available in the `libssh` source tree under `examples/ssh_server_fork.c`. For brevity, we call this program *SSH server* in the rest of this document.

In our analysis, we will assume that the basic step of exploitation has already been performed by the client, i.e., the client has already sent the server an `SSH2_MSG_USERAUTH_SUCCESS` message, as a result of which the `libssh` library on the server has authenticated the client. We deliberately state that it is the library that has authenticated the client for the reasons that will become apparent later.

Let us now have a look at the code of the function `channel_open()` which is set up to be invoked whenever a client requests a new channel from *SSH server* (the line numbers on the left-hand side correspond to the line numbers in the source file `ssh_server_fork.c`):

```
437  static ssh_channel channel_open(ssh_session session, void *userdata) {
438      struct session_data_struct *sdata = (struct session_data_struct *) userdata;
439
440      sdata->channel = ssh_channel_new(session);
441      return sdata->channel;
442  }
```

We see that *SSH server* merely calls `ssh_channel_new()` to open a new channel and stores its handle for later reference (line 440). Notice that no other operations on the channel are performed at this point. What this means is that, even though the library on the server has authenticated the client and will therefore open a new channel when asked to do so, the server is still not able to fulfill client requests, such as allocating a pty or spawning a shell. To enable this functionality, the server must first talk to the library and register a set of custom callback functions that will handle those client requests. The following piece of code shows how exactly *SSH server* does this:

```
504      struct ssh_channel_callbacks_struct channel_cb = {
505          .userdata = &cdata,
506          .channel_pty_request_function = pty_request,
507          .channel_pty_window_change_function = pty_resize,
508          .channel_shell_request_function = shell_request,
509          .channel_exec_request_function = exec_request,
510          .channel_data_function = data_function,
511          .channel_subsystem_request_function = subsystem_request
512      };
...
```

```
534     while (sdata.authenticated == 0 || sdata.channel == NULL) {
...         /* Authenticate the user and allocate a channel */
546     }
547
548     ssh_set_channel_callbacks(sdata.channel, &channel_cb);
```

To set up custom callbacks for various channel operations, the server calls `ssh_set_channel_callbacks()`. As is evident from the code (lines 534-548), however, *SSH server* won't do that until it has made sure that the client has been properly authenticated and that a channel has been successfully allocated. *SSH server* does not merely rely on the `libssh` library for client authentication, but employs its own authentication mechanism and keeps track of whether the client has been authenticated or not through a custom state variable (see `authenticated` on line 534). The only way to change `authenticated` to a non-zero value is by entering a valid username and password combination, as we can see from the following code:

```
if (strcmp(user, USER) == 0 && strcmp(pass, PASS) == 0) {
    sdata->authenticated = 1;
    return SSH_AUTH_SUCCESS;
}
```

So even if a client is able to trick the server into allocating a new channel, there is no way to pass the loop on lines 534-546 without supplying valid user credentials. As a result, *SSH server* will not set up custom channel callbacks defined in `channel_cb` (lines 504-512), and all the client requests on the open channel will instead be fulfilled by the default callback functions provided by the library. The default callbacks, however, are noops (more precisely, the client is denied the requested service), which renders full exploitation impossible.

Program 2: SSH proxy

Now we look into another example program that is also part of `libssh`. The program implements a simple SSH proxy server and is available under `examples/proxy.c` in the `libssh` source tree. In the rest of this document, we refer to this program by the name of *SSH proxy*.

We again assume that the client has already carried out the first step of exploitation, which is tricking the library on the server side to authenticate it. From that point on, we analyze how the server handles the request for opening a new channel. The code that is relevant for our analysis is contained in the following lines:

```
 94 struct ssh_channel_callbacks_struct channel_cb = {
 95     .channel_pty_request_function = pty_request,
 96     .channel_shell_request_function = shell_request
 97 };
 98
 99 static ssh_channel new_session_channel(ssh_session session, void *userdata) {
100     (void) session;
101     (void) userdata;
102     if(chan != NULL)
103         return NULL;
104     printf("Allocated session channel\n");
105     chan = ssh_channel_new(session);
106     ssh_callbacks_init(&channel_cb);
107     ssh_set_channel_callbacks(chan, &channel_cb);
108     return chan;
109 }
```

As in the case of *SSH server*, here we also see a callback function that is invoked when a new channel is requested by the client. This function, named `new_session_channel()` (lines 99-109) begins by calling `ssh_channel_new()` (line 105), which is exactly the same behavior exhibited by *SSH server*. Notice, however, that in addition to allocating a new channel, *SSH proxy* immediately registers its custom callbacks for handling channel operations by invoking `ssh_set_channel_callbacks()` on line 107.

It seems as *SSH proxy* relies on the `libssh` library to ensure that no channel can be opened if the session has not been previously authenticated. Nevertheless, given that the library has already been tricked into authenticating the client, opening the new channel on line 105 will succeed. As a result, the subsequent call to `ssh_set_channel_callbacks()` on line 107 will successfully register the custom *SSH proxy* callbacks to handle requests for opening a pty (line 95) and spawning a shell (line 96).

The client can therefore request a shell from *SSH proxy* immediately after opening a channel, which will result in *SSH proxy* executing its callback function `shell_request()`, thereby making the exploitation successful.

Conclusion

Even though both programs that we have analyzed in this document are only demos, they definitely serve as samples for writing more complex SSH servers based on `libssh`. It is therefore likely that other implementations exist, which may be based on these example programs and techniques showcased in them.

We have seen that the extent of exploitation in these two examples depends on the point where the channel callbacks are being initialized. The defensive approach of initializing the callbacks after ensuring that the user has been authenticated and not relying on the library to handle authentication proved to be crucial for making *SSH server* non-exploitable. On the opposite side, a too early initialization of the channel callbacks and putting too much trust into the library made the exploitation in the case of *SSH proxy* possible. This, again, demonstrates the great value of defensive programming and a layered security design.