New critical denial of service vulnerability in the SQLCipher SQL command processing (CVE-2020-27207)

by Robert Hörr (e-mail: robert.hoerr@t-systems.com)
(Penetration Tester of the Deutsche Telekom Security GmbH)

A new critical denial of service vulnerability (Use CVE-2020-27207) in the SQLCipher SQL command processing of the master branch (https://github.com/sqlcipher) was discovered with a self-developed *SQLCipher-FAST* (Fast Automated Software Testing) framework. The versions 4.x.x include the vulnerability, too. The vulnerability forces the database application to read some unexpected RAM data, which leads to an undefined database behavior. The vulnerability was fixed in the version 4.4.1.

What is the SQLCipher library?

The company Zetetic provides several security frameworks. One of them is the open source SQLCipher library. It is an extension for the open source SQLite database providing complete database encryption. Several companies like Samsung, Motorola and SAP are using this library. According to Zetetic, "*SQLCipher is widely used, protecting data for thousands of apps on hundreds of millions of devices[...]* ." (https://www.zetetic.net/sqlcipher/)

How is the SQLCipher library tested?

The code size of SQLite and SQLCipher library is roughly 250,000 lines of code. It is hardly possible to check all code paths by a manual source code review. Hence, dynamic automated machine testing must be performed. An efficient way to perform this kind of testing is the code coverage fuzzing approach. For that, the *SQLCipher-FAST* framework was developed. This framework unites the strengths of several fuzzing tools and detects issues like buffer overflows.

Which issue was discovered?

The *SQLCipher-FAST* framework detects several issues in the SQL command processing. One of the issues is a *heap-use-after-free*. This issue occurs if the following specially crafted SQL command sequence is executed in the API function *sqlite3_exec(...)*:

```
00000000: 5052 4147 4d41 2063 6970 6865 725f 6465  PRAGMA cipher_de
00000010: 6661 756c 745f 706c 6169 6e74 6578 745f  fault_plaintext_
00000020: 6865 6164 6572 5f73 697a 6520 0000 6170  header_size ..ap
00000030: 001a 1300 3d20 303b 220a 206b 6579 2043  ....= 0;". key C
00000040: 2024 6865 786b 6579 7370 6563 3b0a 2020   $hexkeyspec;.
00000050: 2020 5052 4147 4d41 2063 6970 6865 725f    PRAGMA cipher_
00000060: 706c 6169 6e74 6578 745f 6865 6164 6572  plaintext_header
00000070: 5f73 697a 6520 3d20 3332 3b0a 2020 2020  _size = 32;.
00000080: 7052 4147 4d41 206a 6f75 726e 616c 5f6d  pRAGMA journal_m
00000090: 6f64 653b 0a20 2020 5241 4700 0000 0000  ode;.   RAG.....
000000a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000d0: 0000 0000 0000 00                        .......
```

This sequence includes the string "*cipher_default_plaintext_header_size*" which is processed in the following source code parts of the file sqlite3.c:

```
function: sqlcipher_codec_pragma (...) {
if( sqlite3StrICmp(zLeft,")==0 ) {
  if( zRight ) {
     sqlcipher_set_default_plaintext_header_size(atoi(zRight));
  } else {
     char *size =
sqlite3_mprintf("%d", sqlcipher_get_default_plaintext_header_size());
     codec_vdbe_return_string(pParse, "cipher_default_plaintext_header_size",
size, P4_DYNAMIC);
     sqlite3_free(size);
}...}...}
```

```
function: sqlite3Strlen30(const char *z) {
if( z==0 ) return 0;
return 0x3fffffff & (int)strlen(z);
...
}
```

The pointer *size* gets an address of a dynamically allocated memory area. In the function *codec_vdbe_return_string(...)* the pointer *size* is copied and the pointer *size* is freed by the function execution *sqlite3_free(size)*. Later the address of the copied pointer is read in the function *strlen(z)* of the function *sqlite3Strlen30(...)* which leads to an undefined database behavior. At the end of the program the function *freeP4(...)* frees the copied pointer.

How is the discovered issue exploitable?

Exploiting this issue can result in a remote denial of service attack. For example, a SQL injection can be used to execute the specially crafted SQL command sequence. After that, some unexpected RAM data is read that leads to an undefined database behavior.