SELECT code_execution FROM * USING SQLite;

By Omri Herscovici

Published: 2019-08-10 · Archived: 2026-05-05 02:33:37 UTC

Gaining code execution using a malicious SQLite database

Research By: Omer Gull

tl;dr

SQLite is one of the most deployed software in the world. However, from a security perspective, it has only been examined through the lens of WebSQL and browser exploitation. We believe that this is just the tip of the iceberg.

In our long term research, we experimented with the exploitation of memory corruption issues within SQLite without relying on any environment other than the SQL language. Using our innovative techniques of Query Hijacking and Query Oriented Programming, we proved it is possible to reliably exploit memory corruptions issues in the SQLite engine. We demonstrate these techniques a couple of real-world scenarios: pwning a password stealer backend server, and achieving iOS persistency with higher privileges.

We hope that by releasing our research and methodology, the security research community will be inspired to continue to examine SQLite in the countless scenarios where it is available. Given the fact that SQLite is practically built-in to every major OS, desktop or mobile, the landscape and opportunities are endless. Furthermore, many of the primitives presented here are not exclusive to SQLite and can be ported to other SQL engines. Welcome to the brave new world of using the familiar Structured Query Language for exploitation primitives.

Motivation

This research started when omriher and I were looking at the leaked source code of some notorious password stealers. While there are plenty of password stealers out there (Azorult, Loki Bot, and Pony to name a few), their modus operandi is mostly the same:

A computer gets infected, and the malware either captures credentials as they are used or collects stored credentials maintained by various clients.
It is not uncommon for client software to use SQLite databases for such purposes.
After the malware collects these SQLite files, it sends them to its C2 server where they are parsed using PHP and stored in a collective database containing all of the stolen credentials.

Skimming through the leaked source code of such password stealers, we started speculating about the attack surface described above.
Can we leverage the load and query of an untrusted database to our advantage?
Such capabilities could have much bigger implications in countless scenarios, as SQLite is one of the most widely deployed pieces of software out there.

A surprisingly complex code base, available in almost any device imaginable. is all the motivation we needed, and so our journey began.

SQLite Intro

The chances are high that you are currently using SQLite, even if you are unaware of it.
To quote its authors:

SQLite is a C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine. SQLite is the most used database engine in the world. SQLite is built into all mobile phones and most computers and comes bundled inside countless other applications that people use every day.

Unlike most other SQL databases, SQLite does not have a separate server process. SQLite reads and writes directly to ordinary disk files. A complete SQL database with multiple tables, indices, triggers, and views is contained within a single disk file.

Attack Surface

The following snippet is a fairly generic example of a password stealer backend.

Given the fact that we control the database and its content, the attack surface available to us can be divided into two parts: The load and initial parsing of our database, and the SELECT query performed against it.

The initial loading done by sqlite3_open is actually a very limited surface; it is basically a lot of setup and configuration code for opening the database. Our surface is mainly the header parsing which is battle-tested against AFL.

Things get more interesting as we start querying the database.
Using SQLite authors' words:

"The SELECT statement is the most complicated command in the SQL language."

Although we have no control over the query itself (as it is hardcoded in our target), studying the SELECT process carefully will prove beneficial in our quest for exploitation.

As SQLite3 is a virtual machine, every SQL statement must first be compiled into a byte-code program using one of the sqlite3_prepare* routines.
Among other operations, the prepare function walks and expands all SELECT subqueries. Part of this process is verifying that all relevant objects (like tables or views) actually exist and locating them in the master schema.

sqlite_master and DDL

Every SQLite database has a `sqlite_master` table that defines the schema for the database and all of its objects (such as tables, views, indices, etc.).
The sqlite_master table is defined as:

The part that is of special interest to us is the sql column.
This field is the DDL (Data Definition Language) used to describe the object.
In a sense, the DDL commands are similar to C header files. DDL commands are used to define the structure, names, and types of the data containers within a database, just as a header file typically defines type definitions, structures, classes, and other data structures.

These DDL statements actually appear in plain-text if we inspect the database file:

During the query preparation, sqlite3LocateTable() attempts to find the in-memory structure that describes the table we are interested in querying.

sqlite3LocateTable() reads the schema available in sqlite_master, and if this is the first time doing it, it also has a callback for every result that verifies the DDL statement is valid and build the necessary internal data structures that describe the object in question.

DDL Patching

Learning about this preparation process, we asked, can we simply replace the DDL that appears in plain-text within the file? If we could inject our own SQL to the file perhaps we can affect its behaviour.

Based on the code snippet above, it seems that DDL statements must begin with "create ".
With this limitation in mind, we needed to assess our surface.
Checking SQLite's documentation revealed that these are the possible objects we can create:

The CREATE VIEW command gave us an interesting idea. To put it very simply, VIEWs are just pre-packaged SELECT statements. If we replace the table expected by the target software with a compatible VIEW, interesting opportunities reveal themselves.

Hijack Any Query

Imagine the following scenario:
The original database has a single TABLE called dummy that is defined as:

The target software queries it with the following:

We can actually hijack this query if we craft dummy as a VIEW:

This “trap” VIEW enables us to hijack the query – meaning we generate a completely new query that we **totally control.**

This nuance greatly expands our attack surface, from the very minimal parsing of the header and an uncontrollable query performed by the loading software, to the point where we can now interact with vast parts of the SQLite interpreter by patching the DDL and creating our own views with sub-queries.

Now that we can interact with the SQLite interpreter, our next question was what exploitation primitives are built into SQLite? Does it allow any system commands, reading from or writing to the filesystem?

As we are not the first to notice the huge SQLite potential from an exploitation perspective, it makes sense to review prior work done in the field. We started from the very basics.

SQL Injections

As researchers, it's hard for us to even spell SQL without the "i", so it seems like a reasonable place to start. After all, we want to familiarize ourselves with the internal primitives offered by SQLite. Are there any system commands? Can we load arbitrary libraries?

It seems that the most straightforward trick involves attaching a new database file and writing to it using something along the lines of:

We attach a new database, create a single table and insert a single line of text. The new database then creates a new file (as databases are files in SQLite) with our web shell inside it.
The very forgiving nature of the PHP interpreter parses our database until it reaches the PHP open tag of "<?".
Writing a webshell is definitely a win in our password stealers scenario, however, as you recall, DDL cannot begin with "ATTACH"

Another relevant option is the load_extension function. While this function should allow us to load an arbitrary shared object, it is disabled by default.

Memory Corruptions In SQLite

Like any other software written in C, memory safety issues are definitely something to consider when assessing the security of SQLite.
In his great blog post, Michał Zalewski described how he fuzzed SQLite with AFL to achieve some impressive results: 22 bugs in just 30 minutes of fuzzing.

Interestingly, SQLite has since started using AFL as an integral part of their remarkable test suite.

These memory corruptions were all treated with the expected gravity (Richard Hip and his team deserve tons of respect).
However, from an attacker's perspective, these bugs would prove to be a difficult path to exploitation without a decent framework to leverage them.
Modern mitigations pose a major obstacle in exploiting memory corruption issues and attackers need to find a more flexible environment.

The Security Research community would soon find the perfect target!

Web SQL

Web SQL Database is a web page API for storing data in databases that can be queried using a variant of SQL through JavaScript.The W3C Web Applications Working Group ceased working on the specification in November 2010, citing a lack of independent implementations other than SQLite.

Currently, the API is still supported by Google Chrome, Opera and Safari.
All of them use SQLite as the backend of this API.

Untrusted input into SQLite, reachable from any website inside some of the most popular browsers, caught the security community's attention and as a result, the number of vulnerabilities began to rise.
Suddenly, bugs in SQLite could be leveraged by the JavaScript interpreter to achieve reliable browser exploitation.

Several impressive research reports have been published:

- Low hanging fruits like CVE-2015-7036

    - Untrusted pointer dereference **fts3_tokenizer()**
- More complex exploits presented in Blackhat 17 by the Chaitin team
    - Type confusion in **fts3OptimizeFunc()**
- The recent Magellan bugs exploited by Exodus
    - Integer overflow in **fts3SegReaderNext()**

A clear pattern in past WebSQL research reveals that a virtual table module named "FTS" might be an interesting target for our research.

FTS

Full-Text Search (FTS) is a virtual table module that allows textual searches on a set of documents.
From the perspective of an SQL statement, the virtual table object looks like any other table or view. But behind the scenes, queries on a virtual table invoke callback methods on shadow tables instead of the usual reading and writing on the database file.

Some virtual table implementations, like FTS, make use of real (non-virtual) database tables to store content.

For example, when a string is inserted into the FTS3 virtual table, some metadata must be generated to allow for an efficient textual search. This metadata is ultimately stored in real tables named "%_segdir" and "%_segments", while the content itself is stored in ""%_content" where "%" is the name of the original virtual table.
These auxiliary real tables that contain data for a virtual table are called "shadow tables"

Due to their trusting nature, interfaces that pass data between shadow tables provide a fertile ground for bugs. CVE-2019-8457,- a new OOB read vulnerability we found in the RTREE virtual table module, demonstrates this well.

RTREE virtual tables, used for geographical indexing, are expected to begin with an integer column. Therefore, other RTREE interfaces expect the first column in an RTREE to be an integer. However, if we create a table where the first column is a string, as shown in the figure below, and pass it to the **rtreenode()** interface, an OOB read occurs.

Now that we can use query hijacking to gain control over a query, and know where to find vulnerabilities, it's time to move on to exploit development.

SQLite Internals For Exploit Development

Previous publications on SQLite exploitation clearly show that there has always been a necessity for a wrapping environment, whether it is the PHP interpreter seen in this awesome blog post on abusing SQLite tokenizers or the more recent work on Web SQL from the comfort of a JavaScript interpreter.

As SQLite is pretty much everywhere, limiting its exploitation potential sounded like low-balling to us and we started exploring the use of SQLite internals for exploitation purposes.
The research community became pretty good at utilizing JavaScript for exploit development. Can we achieve similar primitives with SQL?

Bearing in mind that SQL is Turing complete ([1], [2]), we started creating a primitive wish-list for exploit development based on our pwning experience.
A modern exploit written purely in SQL has the following capabilities:

- Memory leak.
- Packing and unpacking of integers to 64-bit pointers.
- Pointer arithmetics.
- Crafting complex fake objects in memory.
- Heap Spray.

One by one, we will tackle these primitives and implement them using nothing but SQL.

For the purpose of achieving RCE on PHP7, we will utilize the still unfixed 1-day of CVE-2015-7036.

Wait, what? How come a 4-year-old bug has never been fixed? It is actually an interesting story and a great example of our argument.
This feature was only ever considered vulnerable in the context of a program that allows arbitrary SQL from an untrusted source (Web SQL), and so it was mitigated accordingly.
However, SQLite usage is so versatile that we can actually still trigger it in many scenarios 🙂

Exploitation Game-plan

CVE-2015-7036 is a very convenient bug to work with.
In a nutshell, the vulnerable **fts3_tokenizer()** function returns the tokenizer address when called with a single argument (like "simple", "porter" or any other registered tokenizer).

When called with 2 arguments, fts3_tokenizer overrides the tokenizer address in the first argument with the address provided by a blob in the second argument.
After a certain tokenizer has been overridden, any new instance of the fts table that uses this tokenizer allows us to hijack the flow of the program.

Our exploitation game-plan:

- Leak a tokenizer address
- Compute the base address
- Forge a fake tokenizer that will execute our malicious code
- Override one of the tokenizers with our malicious tokenizer
- Instantiate an fts3 table to trigger our malicious code

Now back to our exploit development.

Query Oriented Programming ©

We are proud to present our own unique approach for exploit development using the familiar structured query language. We share QOP with the community in the hope of encouraging researchers to pursue the endless possibilities of database engines exploitation.
Each of the following primitives is accompanied by an example from the sqlite3 shell.

While this will give you a hint of what want to achieve, keep in mind that our end goal is to plant all those primitives in the sqlite_master table and hijack the queries issued by the target software that loads and queries our malicious SQLite db file

Memory Leak – Binary

Mitigations such as ASLR definitely raised the bar for memory corruptions exploitation. A common way to defeat it is to learn something about the memory layout around us.

This is widely known as Memory Leak.
Memory leaks are their own sub-class of vulnerabilities, and each one has a slightly different setup.

In our case, the leak is the return of a BLOB by SQLite.
These BLOBs make a fine leak target as they sometimes hold memory pointers.

The vulnerable fts3_tokenizer() is called with a single argument and returns the memory address of the requested tokenizer.
hex() makes it readable by humans.
We obviously get some memory address, but it is reversed due to little-endianity.
Surely we can flip it using some SQLite built-in string operations.

substr() seems to be a perfect fit! We can read little-endian BLOBs but this raises another question: how do we store things?

QOP Chain

Naturally, storing data in SQL requires an INSERT statement. Due to the hardened verification of sqlite_master, we can't use INSERT as all of the statements must start with "CREATE ". Our approach to this challenge is to simply store our queries under a meaningful VIEW and chain them together.
The following example makes it a bit clearer:

This might not seem like a big difference, but as our chain gets more complicated, being able to use pseudo-variables will surely make our life easier.

Unpacking of 64-bit pointers

If you've ever done any pwning challenges, the concept of packing and unpacking of pointers should not be foreign.
This primitive should make it easy to convert our hexadecimal values (like the leak we just achieved) to integers. Doing so allows us to perform various calculations on these pointers in the next steps.

This query iterates a hexadecimal string char by char in a reversed fashion using substr().

A translation of this char is done using this clever trick with the minor adjustment of instr() which is 1-based.
All that is needed now is the proper shift that is on the right of the * sign.

Pointer arithmetics

Pointer arithmetics is a fairly easy task with integers at hand. For example, extracting the image base from our leaked tokenizer pointer is as easy as:

Packing of 64-bit pointers

After reading leaked pointers and manipulating them to our will, it makes sense to pack them back to their little-endian form so we can write them somewhere.
SQLite char() should be of use here as its documentation states that it will "return a string composed of characters having the Unicode code point values of an integer."
It proved to work fairly well, but only on a limited range of integers

Larger integers were translated to their 2-bytes code-points.
After banging our heads against SQLite documentation, we suddenly had a strange epiphany: our exploit is actually a database.
We can prepare beforehand a table that maps integers to their expected values.

Now our pointer packing query is the following:

Crafting complex fake objects in memory

Writing a single pointer is definitely useful, but still not enough. Many memory safety issues exploitation scenarios require the attackers to forge some object or structure in memory or even write a ROP chain.

Essentially, we will string several of the building blocks we presented earlier.
For example, let's forge our own tokenizer, as explained here.
Our fake tokenizer should conform to the interface expected by SQLite defined here:

Using the methods described above and a simple JOIN query, we are able to fake the desired object quite easily.

Verifying the result in a low-level debugger, we see that indeed a fake tokenizer object was created.

Heap Spray

Now that we crafted our fake object, it is sometimes useful to spray the heap with it.
This should ideally be some repetitive form of the latter.

Unfortunately, SQLite does not implement the REPEAT() function like MySQL.
However, this thread gave us an elegant solution.

The zeroblob(N) function returns a BLOB consisting of N bytes while we use replace() to replace those zeros with our fake object.

Searching for those 0x41s shows we also achieved a perfect consistency. Notice the repetition every 0x20 bytes.

Memory Leak – Heap

Looking at our exploitation game plan, it seems like we are moving in the right direction.
We already know where the binary image is located, we were able to deduce where the necessary functions are, and spray the heap with our malicious tokenizer.

Now it's time to override a tokenizer with one of our sprayed objects. However, as the heap address is also randomized, we don't know where our spray is allocated.
A heap leak requires us to have another vulnerability.

Again, we will target a virtual table interface.
As virtual tables use underlying shadow tables, it is quite common for them to pass raw pointers between different SQL interfaces.

Note: This exact type of issue was mitigated in SQLite 3.20. Fortunately, PHP7 is compiled with an earlier version. In case of an updated version, CVE-2019-8457 could be used here as well.

To leak the heap address, we need to generate an fts3 table beforehand and abuse its MATCH interface.

Just as we saw in our first memory leak, the pointer is little-endian so it needs to be reversed. Fortunately, we already know how to do so using SUBSTR().
Now that we know our heap location, and can spray properly, we can finally override a tokenizer with our malicious tokenizer!

Putting It All Together

With all the desired exploitation primitives at hand, it's time to go back to where we started: exploiting a password stealer C2.

As explained above, we need to set up a "trap" VIEW to kickstart our exploit. Therefore, we need to examine our target and prepare the right VIEW.

As seen in the snippet above, our target expects our db to have a table called Notes with a column called BodyRich inside it. To hijack this query, we created the following VIEW

After Notes is queried, 3 QOP Chains execute. Let's analyze the first one of them.

heap_spray

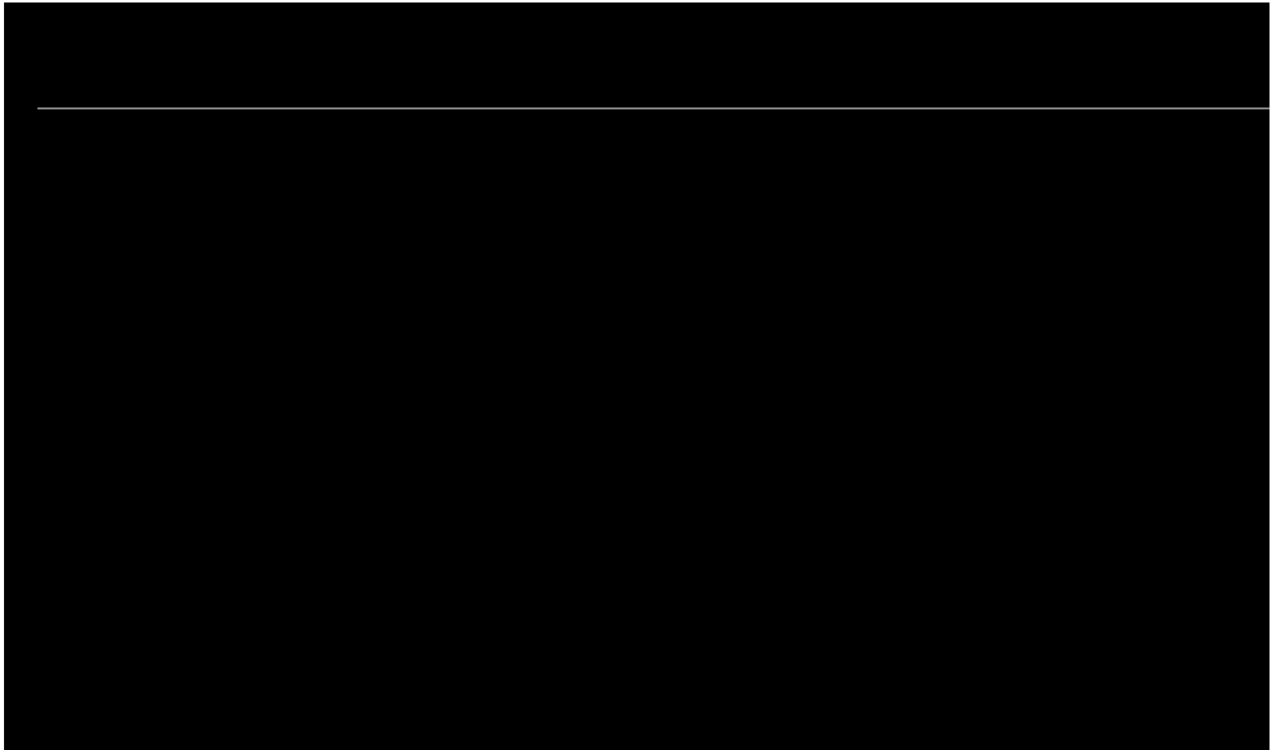
Our first QOP chain should populate the heap with a large amount of our malicious tokenizer.

p64_simple_create, p64_simple_destroy, and p64_system are essentially all chains achieved with our leak and packing capabilities.

For example, p64_simple_create is constructed as:

As these chains get pretty complex, pretty fast, and are quite repetitive, we created QOP.py.
QOP.py makes things a bit simpler by generating these queries in pwntools style.
Creating the previous statements becomes as easy as:

Demo



COMMIT;

Now that we have established a framework to exploit any situation where the querier cannot be sure that the database is non-malicious, let's explore another interesting use case for SQLite exploitation.

iOS Persistency

Persistency is hard to achieve on iOS as all executable files must be signed as part of Apple's Secure Boot. Luckily for us, SQLite databases are not signed.

Utilizing our new capabilities, we will replace one of the commonly used databases with a malicious version. After the device reboots and our malicious database is queried, we gain code execution.

To demonstrate this concept, we replace the Contacts DB "AddressBook.sqlitedb". As done in our PHP7 exploit, we create two extra DDL statements. One DDL statement overrides the default tokenizer "simple", and the other DDL statement triggers the crash by trying to instantiate the overridden tokenizer. Now, all we have to do is re-write every table of the original database as a view that hijacks any query performed and redirect it toward our malicious DDL.

Replacing the contacts db with our malicious contacts db and rebooting results in the following iOS crashdump:

As expected, the contacts process crashed at 0x4141414141414149 where it expected to find the xCreate constructor of our false tokenizer.

Furthermore, the contacts db is actually shared among many processes. Contacts, Facetime, Springboard, WhatsApp, Telegram and XPCProxy are just some of the processes querying it. Some of these processes are more privileged than others. Once we proved that we can execute code in the context of the querying process, this technique also allows us to expand and elevate our privileges.

Our research and methodology have all been responsibly disclosed to Apple and were assigned the following CVEs:

- CVE-2019-8600
- CVE-2019-8598
- CVE-2019-8602
- CVE-2019-8577

Future Work

Given the fact that SQLite is practically built-in to almost any platform, we think that we've barely scratched the tip of the iceberg when it comes to its exploitation potential. We hope that the security community will take this innovative research and the tools released and push it even further. A couple of options we think might be interesting to pursue are

- Creating more versatile exploits. This can be done by building exploits dynamically by choosing the relevant QOP gadgets from pre-made tables using functions such as sqlite_version() or sqlite_compileoption_used().
- Achieving stronger exploitation primitives such as arbitrary R/W.
- Look for other scenarios where the querier cannot verify the database trustworthiness.

Conclusion

We established that simply querying a database may not be as safe as you expect. Using our innovative techniques of Query Hijacking and Query Oriented Programming, we proved that memory corruption issues in SQLite can now be reliably exploited. As our permissions hierarchies become more segmented than ever, it is clear that we must rethink the boundaries of trusted/untrusted SQL input. To demonstrate these concepts, we achieved remote code execution on a password stealer backend running PHP7 and gained persistency with higher privileges on iOS. We believe that these are just a couple of use cases in the endless landscape of SQLite.

Check Point IPS Product Protects against this threat: "SQLite fts3_tokenizer Untrusted Pointer Remote Code Execution (CVE-2019-8602)."

Source: https://research.checkpoint.com/2019/select-code_execution-from-using-sqlite/