The outstanding stealth of Operation Triangulation

By Georgy Kucherin

Published: 2023-10-23 · Archived: 2026-04-05 14:26:06 UTC

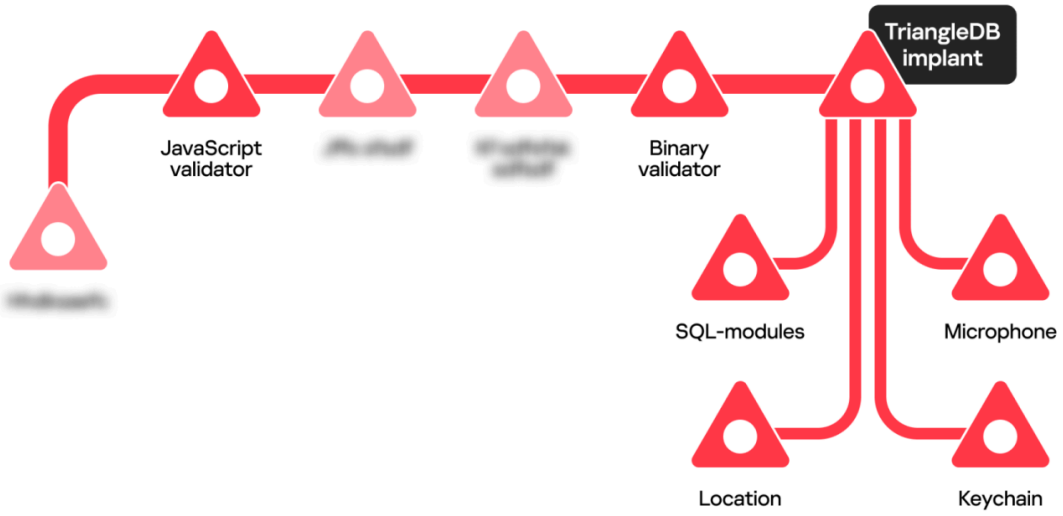
UPD 23.04.2025: MITRE created a page for Operation Triangulation as part of its ATT&CK framework.

Introduction

In our previous blogpost on Triangulation, we discussed the details of TriangleDB, the main implant used in this campaign, its C2 protocol and the commands it can receive. We mentioned, among other things, that it is able to execute additional modules. We also mentioned that this operation was quite stealthy. This article details one important aspect of this attack – the stealth that was exercised by the threat actor behind it. Along the way, we will also reveal more information about the components used in this attack.

Validation components

In our previous blogposts, we outlined the Operation Triangulation infection chain: a device receives a malicious iMessage attachment that launches a chain of exploits, and their execution ultimately results in the launch of the TriangleDB implant. In more detail, the infection chain can be summarized with the following graph:



Apart from the exploits and components of the TriangleDB implant, the infection chain contains two “validator” stages, namely “JavaScript Validator” and “Binary Validator”. These validators collect various information about

the victim device and send it to the C2 server. This information is then used to assess if the iPhone or iPad to be implanted with TriangleDB could be a research device. By performing such checks, attackers can make sure that their 0-day exploits and the implant do not get burned.

JavaScript Validator

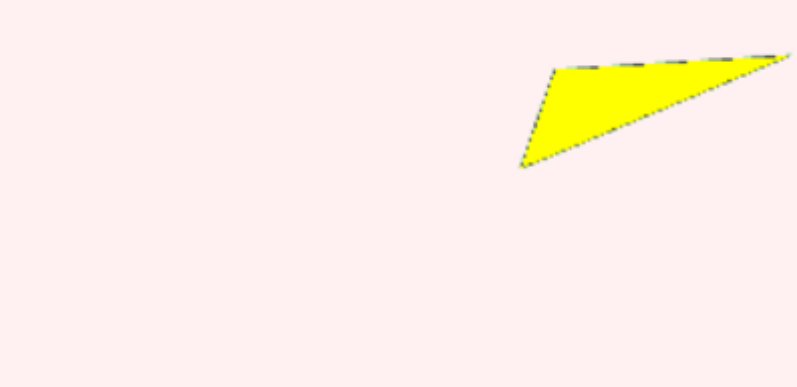
At the beginning of the infection chain, the victim receives an invisible iMessage attachment with a zero-click exploit. The ultimate goal of this exploit is to silently open a unique URL on the backuprabbit[.]com domain. The HTML page hosted on that URL contains obfuscated JavaScript code of the NaCl cryptography library, as well as an encrypted payload. This payload is the JavaScript validator. This validator performs a lot of various checks, including different arithmetic operations like Math.log(-1) or Math.sqrt(-1), availability of components such as Media Source API, WebAssembly and others.

And, as we already mentioned, it performs a fingerprinting technique called Canvas Fingerprinting by drawing a yellow triangle on a pink background with WebGL and calculating its checksum:

```
1  context.bufferData(context.ELEMENT_ARRAY_BUFFER, l, context.STATIC_DRAW);
2  context.useProgram(C);
3  context.clearColor(0.5, 0.7, 0.2, 0.25);
4  context.clear(context.COLOR_BUFFER_BIT);
5  context.drawElements(context.TRIANGLES, l.length, context.UNSIGNED_SHORT, 0);
6  C.L = context.getAttribLocation(C, Z('VE'));
7  C.W = context.getUniformLocation(C, Z('Zv'));
8  context.enableVertexAttribArray(C.L);
9  context.vertexAttribPointer(C.L, 3, context.FLOAT, !1, 0, 0);
10 context.uniform2f(C.W, 1, 1);
11 context.drawArrays(context.TRIANGLE_STRIP, 0, 3);
12 var h = new Uint8Array(262144);
13 context.readPixels(0, 0, 256, 256, context.RGBA, context.UNSIGNED_BYTE, h);
14 data['xT'] = h[88849];
15 data['jHWOO'] = h[95054];
16 data['aRR'] = h[99183];
```

```
data['ffJEi'] = h[130012];
for (var p = 0, _ = 0; _ < h.length; _++)
p += h[_];
data['WjOn'] = p;
```

Code drawing the triangle



The drawn triangle

This triangle is, in fact, why we dubbed this whole campaign Operation Triangulation.

After running the validator, it encrypts and sends all collected information to another unique URL on backuprabbit[.]com in order to receive (or not) the next stage of the infection chain.

Binary Validator

As we see from the infection chain graph, this validator gets launched prior to deployment of the TriangleDB implant. As opposed to the JavaScript Validator, which is a script, this validator is a Mach-O binary file (hence the name Binary Validator). When launched, it decrypts its configuration using AES. This configuration is a plist:

```
<key>sco</key>
  <array>
     <string>DeleteLogs</string>
     <string>DeleteArtifacts</string>
     <string>ProcessList</string>
     <string>InterfaceList</string>
     <string>JailbreakDetect</string>
```

```
8       <string>EnableAdTracking</string>
9       <string>DeviceInfo</string>
10      <string>InstalledApps</string>
11    </array>
12    <key>sda</key>
13    <dict>
14      <key>sdf</key>
15      <array/>
16      <key>sdi</key>
17      <true/>
18      <key>sdk</key>
19      <array>
20    <string>c99218578c03cfe347fababc838dd9f2</string>
21    <string>3d527800ad9418b025340775eaf6454c</string>
22    <string>07d2143cea9fe70f7a0fcc653a002403</string>
23    <string>c66cc1d90cce4e9cb6b631e063c83d61</string>
```

This plist contains a list of actions (such as DeleteLogs, DeleteArtifacts, etc.) that have to be performed by the validator. Specifically, it:

- Removes crash logs from the /private/var/mobile/Library/Logs/CrashReporter directory that could have been created during the exploitation process;
- Searches for traces of the malicious iMessage attachment in various databases, such as ids-pub-id.db or knowledgeC.db, and then removes them. To be able to do that, the validator's configuration contains 40 MD5 hashes of Apple IDs that are used for sending the malicious iMessages. We managed to crack the majority of these hashes, thus obtaining a list of attacker-controlled Apple ID email addresses:

mailto:travislong544[at]yahoo.com	mailto:tinyjax89[at]gmail.com
mailto:norsarall87[at]outlook.com	mailto:nonbaguette[at]yahoo.com
mailto:jesteristhebestband[at]gmail.com	mailto:slbrimms96[at]outlook.com
mailto:christineashleysmith[at]gmail.com	mailto:costamaria91[at]outlook.com
mailto:homicidalwombat[at]yahoo.com	mailto:hyechink97[at]gmail.com
mailto:nigelmlevy[at]gmail.com	mailto:greatoleg9393[at]mail.com

mailto:supercatman15[at]hotmail.com	mailto:popanddangle[at]outlook.com
mailto:shannonkelly404[at]gmail.com	mailto:maxjar90[at]mail.com
mailto:superhugger21[at]gmail.com	mailto:chongwonnam[at]gmail.com
mailto:parkourdiva[at]yahoo.com	mailto:wopperplopper1[at]aol.com
mailto:naturelover1972[at]outlook.com	mailto:bajablaster101[at]gmail.com
mailto:sasquatchdreams[at]outlook.com	mailto:carlson31773[at]outlook.com
mailto:trunkfullofbeans[at]yahoo.com	mailto:fsozgur[at]outlook.com
mailto:danielhbarnes2[at]gmail.com	mailto:soccerchk835[at]gmail.com
mailto:patriotsman121[at]gmail.com	mailto:stephamartinez122[at]gmail.com
mailto:wheelsordoors[at]yahoo.com	mailto:popcornkerner[at]gmail.com
mailto:janahodges324[at]gmail.com	mailto:pupperoni1989[at]outlook.com
mailto:mibarham[at]outlook.com	mailto:biglesterjames5[at]gmail.com

- Gets a list of processes running on the device, as well as a list of network interfaces;
- Checks whether the target device is jailbroken. The validator implements checks for a wide range of jailbreak tools: Pangu, xCon, Evasion7, Electra, unc0ver, checkra1n and many more;
- Turns on personalized ad tracking;
- Collects a wide range of information about the victim, such as username, phone number, IMEI and Apple ID;
- Retrieves a list of installed applications.

What is interesting about these actions is that the validator implements them both for iOS and macOS systems:

```
formatStr = stringWithUtf8String(
              &OBJC_CLASS___NSString,
              v33,
              "/Users/%s/Library/Preferences/com.apple.AdLib.plist");
v39 = getlogin();
adLib_plist_path = stringWithFormat(&OBJC_CLASS___NSString, v40, formatStr, v39);
```

We also found that the validator implements an unused action, which was dubbed PSPDetect by the attackers.

```
v70[0] = stringWithUtf8String(&OBJC_CLASS___NSString, v9, "ProcessList");
v71[0] = valueWithPointer(&OBJC_CLASS___NSValue, v10, ProcessListFunc);
v70[1] = stringWithUtf8String(&OBJC_CLASS___NSString, v11, "PSPDetect");
v71[1] = valueWithPointer(&OBJC_CLASS___NSValue, v12, PSPDetectFunc);
v70[2] = stringWithUtf8String(&OBJC_CLASS___NSString, v13, "InterfaceList");
v71[2] = valueWithPointer(&OBJC_CLASS___NSValue, v14, InterfaceListFunc);
v70[3] = stringWithUtf8String(&OBJC_CLASS___NSString, v15, "JailbreakDetect'
v71[3] = valueWithPointer(&OBJC_CLASS___NSValue, v16, JailbreakDetectFunc);
v70[4] = stringWithUtf8String(&OBJC_CLASS___NSString, v17, "DeviceInfo");
v71[4] = valueWithPointer(&OBJC_CLASS___NSValue, v18, DeviceInfoFunc);
v70[5] = stringWithUtf8String(&OBJC_CLASS___NSString, v19, "InstalledApps");
v71[5] = valueWithPointer(&OBJC_CLASS___NSValue, v20, sub_10000A3A0);
```

This action retrieves a list of files from the validator's configuration (this list was empty for the validator configurations that we analyzed), checks if they are present in the file system and produces a list of found files as output.

The abbreviation PSP in the name of this action may mean “personal security product,” or, in simpler terms, a security solution. It is thus possible that this action may be launched on macOS devices in order to detect installed antivirus products.

Having executed all these actions, the validator encrypts and sends the obtained data (list of processes, user information, etc.) to the C2 server. In response, the server returns the TriangleDB implant, which we described before.

Looking for traces in logs, again

The threat actor behind Operation Triangulation exercises stealth not only by introducing two validators in the infection chain. In fact, they perform all operations with the TriangleDB implant very carefully. This can be observed from our analysis of commands sent by the attackers to the infected devices via this implant.

After the implant establishes communication with the C2 server and sends a heartbeat, it receives multiple CRXShowTables and CRXFetchRecord commands from the C2 server. These commands are related to retrieval of logs that might show traces of the infection chain and/or the malware itself. Some of the files that are retrieved are:

- Crash log files (e.g. those in the /var/mobile/Library/Logs/CrashReporter);
- Database files (e.g. /private/var/mobile/Library/IdentityServices/ids-gossip.db). These database files may contain the Apple ID used by the attackers to send the malicious iMessage.

Once the attackers receive these files, they delete them from the device, so that the victim can’t examine them and potentially find signs of compromise. Having completed log collection and deletion, the attackers send multiple CRXPollRecords commands to the implant, instructing it to periodically exfiltrate files from the /private/var/tmp directory. The names of the files to be uploaded to the C2 server should match one of the following regular expressions:

Regular expression	**Type of data**
`^(kng|dky).+\.dat$`	Location data
`^adr3.+\.dat$`	SQL-related data
`^sr6d.+\.(dat|srm)$`	Microphone-recorded data
`^S5L.+\.kcd$`	Keychain data
`^ntc.+\.db2$`	Unknown
`^(\w[247F][023A][24BC]|\w[4AEF][349B][169D]){2}-\w[05AF][3468][124C]-4[123A][09AD][356A]-[89AB]\w\w\w-(\w[126A][24CE][348B]|\w[29DE][168D][156D]){3}$`	Unknown

The files with these names contain execution results produced by modules. These modules are uploaded to the infected device through the CRXUpdateRecord and CRXRunRecord commands – we describe them below.

Microphone recording

One of the most privacy-invading modules is the microphone-recording module, which goes by the name of “msu3h” (we believe 3h stands for three hours, the default recording duration). Upon execution, it decrypts (using a custom algorithm derived from GTA IV hashing) its configuration, but it performs further actions only if the battery is more than 10% charged.

The configuration file itself contains typical configuration data, such as how long to record for and the AES encryption key used to encrypt recordings, but also more menacing parameters, such as:

- suspendOnDeviceInUse: sets whether recording should be stopped if the device screen is turned on;
- syslogRelayOverride: sets whether audio should be recorded when system logs are being captured.

The recording takes place using the Audio Queue API, and sound chunks are compressed using the Speex codec, then encrypted using AES. Apart from sound data, each recording contains diagnostic messages, which have a four-byte type identifier, which can either be:

Identifier	Message
0x6265676E	recording started
0x736C726E	recording stopped because of syslog monitoring
0x6465766E	recording stopped because device screen got turned on
0x6964736E	recording stopped because of insufficient disk space
0x656E646E	recording finished

Keychain exfiltration

For an unknown reason, the attackers decided to add an additional keychain exfiltration module, despite such functionality already being present in TriangleDB. This keychain module has the same logic as that in TriangleDB, but is largely based on code from the iphone-dataprotection.keychainviewer project.

SQLite stealing modules

Many apps on iOS use SQLite to store their internal data. It is thus of no surprise that the attackers implemented modules capable of stealing data from various SQLite databases. All these modules have the same codebase, and contain different SQL queries to be executed. Again, they have a configuration that is encrypted. When this is decrypted, only standard variables such as file path, AES key, query string, etc. can be found.

The code of these modules is quite peculiar. For example, the attackers implemented a wrapper around the fopen() function, adding the Z flag (indicating that a created file should be AES-encrypted and zlib-compressed) used in combination with the standard w (write) flag, as can be seen in the image below.

```
time_now = time(0LL);
srand(time_now);
generated_filename = generate_random_name(filepath, ext, 6LL);
generated_filename_1 = generated_filename;
if ( !generated_filename )
{
  ky_value = 0LL;
  random_name = 0LL;
  file_ptr = 0LL;
  _err = 0xE0000002LL;
  goto LABEL_96;
}
file_ptr = custom_fopen(generated_filename, "wZ", AES_key);
```

What is also interesting is that the SQLite stealing modules contain three branches of code for different iOS versions: lower than 8.0, between 8.0 and 9.0, and 9.0 and later.

Each module that we found performs different SQL database queries. For example, there is a module that processes application usage data from the knowledgeC.db database. Another module extracts photo-related metadata, such as whether a child is in the picture or not, if the person is male or female (see image below) and text that was automatically OCRed from media files.

```
1   ...
2   END AS 'Face(s) Detected',
3   CASE face.ZAGETYPE
4      WHEN 1 THEN 'Baby / Toddler'
5      WHEN 2 THEN 'Baby / Toddler'
6      WHEN 3 THEN 'Child / Young Adult'
7      WHEN 4 THEN 'Young Adult / Adult'
8      WHEN 5 THEN 'Adult'
9      else 'Unknown'
10  END AS 'Subject Age Estimate',
11  CASE face.ZGENDERTYPE
12     WHEN 1 THEN 'Male'
13     WHEN 2 THEN 'Female'
```

```
14      else 'Unknown'
15  END AS 'Subject Gender',
16  person.ZDISPLAYNAME AS 'Subject Name'
17  ...
```

It should also come as no surprise that the attackers expressed interest in WhatsApp, SMS and Telegram messages as well – we found modules exfiltrating this data too.

Location-monitoring module

This module runs in a separate thread and tries to impersonate the bundle that is authorized to use the location services specified in the configuration (e.g. /System/Library/LocationBundles/Routine.bundle). Apart from using GPS to determine the location, it also uses GSM, retrieving the MCC (MobileCountryCode), MNC (MobileNetworkCode), LAC (LocationAreaCode) and CID (CellID) values through the CoreTelephony framework.

One reason for using GSM-related data is to estimate the victim’s location when GPS data is not available.

```
if ( getValue(dict, kCTCellMonitorMCC, &mcc) )
{
  snprintf(&a1->MCC, 4uLL, "%u", mcc);
  a1->flag |= 1u;
}
else
{
  ret_code = 2LL;
}
if ( getValue(dict, kCTCellMonitorMNC, &mnc) )
{
  snprintf(&a1->MNC, 4uLL, "%u", mnc);
  a1->flag |= 2u;
}
else
{
  ret_code = 2LL;
}
if ( getValue(dict, kCTCellMonitorLAC, &lac) )
{
  a1->LAC = lac;
  a1->flag |= 4u;
}
else
{
  ret_code = 2LL;
}
if ( getValue(dict, kCTCellMonitorCellId, &cid) )
{
  a1->cellID = cid;
```

Conclusion

The adversary behind Triangulation took great care to avoid detection. They introduced two validators in the infection chain in order to ensure that the exploits and the implant do not get delivered to security researchers. Additionally, microphone recording could be tuned in such a way that it stopped when the screen was being used. The location tracker module may not use standard GPS functionality if this is unavailable, but rather metadata from the GSM network.

The attackers also showed a great understanding of iOS internals, as they used private undocumented APIs in the course of the attack. They additionally implemented in some modules support for iOS versions prior to 8.0. Recall that these were widely used before 2015, which gives an indication of just how long the code of the modules has been in use.

Last but not least, some of the components used in this attack contain code that may indicate that they are targeting macOS systems as well, although, as of the publication date, no Triangulation traces have been encountered on macOS devices.

Even though Operation Triangulation was executed with a high degree of stealth, we were still able to extract the full exploitation chain, as well as the implant and its plugins. If you want to find how we managed to circumvent all the protections introduced by the attackers, we encourage you to attend the SAS conference, where Igor Kuznetsov will present a talk entitled “Operation Triangulation: Connecting the Dots” and the story of how this long-lasting attack was put to a stop. If you are not able to make it to Phuket, you can read the blogpost summarizing this talk that we will release shortly after SAS.

Indicators of Compromise

Keychain module

MD5 527bb38d4716c019b65da64d0f851a70
SHA-1 a468613d31c90ac94bbd313bc70c5c6638c91603
SHA-256 64f36b0b8ef62634a3ec15b4a21700d32b3d950a846daef5661b8bbca01789dc

Location module

MD5 da5d3c0d3ad8df77ff6f331066636e42
SHA-1 a5a93e8d48fdef8c02066b9020445b50ebc81a8f
SHA-256 7e779a019f250d8cec9761d1230296236a8b714743df42c49ce8daf818d542e7

SMS-stealing module

MD5 adb9e4b7a75eccc37f6941a5cbc7685b
SHA-1 6e9cd17fcc8b14cc860ce980c5e919494a10eec9
SHA-256 c2393fceab76776e19848c2ca3c84bea0ed224ac53206c48f1c5fd525ef66306

Microphone module

MD5 ac2444e7f7b0a4b084ad8c9ae8ac26c8
SHA-1 10509067ba5d9d985e932ea77f089491dee1611d
SHA-256 ff2f223542bbc243c1e7c6807e4c80ddad45005bcd78a77f8ec91de29deb2f6e

---