Reverse engineering of Flutter applications

Axelle Apvrille, Fortinet

Oct 23, 2023

Contents

This paper explores the challenges associated with reverse engineering Flutter applications and presents effective techniques to overcome them.

Beginning with an introduction to Flutter applications and the Dart programming language, we embark on the task of reversing a basic Dart program and a Flutter application built in release mode. Despite our efforts, we encounter significant difficulties in recovering our original code with current tools.

These challenges arise from key concepts that we delve into, namely the Dart Object Pool, object serialization, and Flutter obfuscation. Armed with this understanding, we proceed to the subsequent sections of the paper where we propose techniques for function identification, string recovery, byte array analysis, and the identification of simple loops and algorithms. With the application of these techniques, we successfully reverse our initial programs and applications.

Short introduction to Flutter

Flutter ("Flutter," n.d.) is an open source UI software development kit created by Google. It is particularly attractive for its ability to develop with a *single codebase*, and then compile *natively* to various mobile - or non-mobile - platforms (Android, iOS, Windows, MacOS, Linux, Web… see list).

The applications are developed in *Dart* ("Dart," n.d.), an object-oriented programming language with a C-style syntax and a few features such as sound null safety. They can then be compiled into various formats, depending on desired properties such as portability, performance. Flutter uses *kernel snapshots* for debug builds, and Ahead Of Time (AOT) snapshots for *release* builds. The former are cross platform, the latter initialize quicker.

Dart program formats	Size	Exec time	Description
hello.dart	266 bytes	0m0,320s	Source code
hello.exe	5.8 M	0m0,008s	Self contained executable
hello.aot	863 K	0m0,008s	AOT snapshot
hello.jit	4.7 M	0m0,242s	JIT snapshot
hello.dill	936 bytes	0m0,245s	Kernel snapshot

Anatomy of a Flutter application

Flutter applications consist in at least 2 snapshots: one for the VM, and one or more snapshots for the program. The **VM snapshot** contains the base functionality of the Dart VM and common Dart libraries. The other snapshots are called **isolate snapshots**: one snapshot per isolate. An *isolate* is an independent unit of execution that runs concurrently with other isolates within the same Dart process. Each isolate has it own memory heap, stack and event loop - contrary to OS threads which share the same memory space.

Dart programs have at least one isolate, to run the main "thread", and possibly more. For instance, the developer may decide to create more isolate to handle seperately UI rendering, or network requests etc. So, Flutter applications at least have *one isolate snapshot* (possibly more). It basically freezes the state of the Dart VM before the main is called.

When bundled as Android applications, a Flutter application typically has the following files:

```
./AndroidManifest.xml
./assets/
     flutter_assets/
        AssetManifest.json
        FontManifest.json
        fonts/
        ...
./classes.dex
./kotlin/
     ...
./lib/
      ./arm64-v8a/
         libapp.so
         libflutter.so
      ./armeabi-v7a/
         libapp.so
         libflutter.so
      ./x86_64/
         libapp.so
         libflutter.so
./META-INF/
    ...
./res/
    ...
./resources.arsc
```

The Dalvik code inside `classes.dex` has no interest (apart from loading the Dart runtime). The interesting Dart payload is located in the native libraries `libapp.so`, which are compiled for arm32, arm64 and x86_64. `libapp.so` contains the Flutter snapshots. They are visible from dynamic symbols.

```
$ objdump -T libapp.so

libapp.so:     file format elf64-x86-64

DYNAMIC SYMBOL TABLE:
0000000000170000 g    DO .text      00000000000047a0
    _kDartVmSnapshotInstructions
00000000001747a0 g    DO .text      00000000002848e0
    _kDartIsolateSnapshotInstructions
00000000000001f0 g    DO .rodata    0000000000003890 _kDartVmSnapshotData
0000000000003a80 g    DO .rodata    0000000000169bb0
    _kDartIsolateSnapshotData
00000000000001c8 g    DO .note.gnu.build-id 0000000000000020
    _kDartSnapshotBuildId
```

We see 2 snapshots (VM and Isolate), and each snapshot has a code section (`.text`) and a data section (`.rodata`). To reverse engineer a Flutter application, we are interested in `_kDartIsolateSnapshotInstructions` and `_kDartIsolateSnapshotData`.

Status for Reversing Dart programs

Basic Dart program

We write a basic Dart program. The syntax of Dart is close to C, with a few built-in types such as **`int`**, **`double`**, `bool`, `List`, `Set`, `String`. Contrary to many programming languages though, Dart does not have any type to represent *bytes* nor *single characters*. Bytes are simply represented as arrays of integers: `List<int>`, or a more optimized form as `Uint8List`.

```
import 'dart:typed_data';

void xor_stage2() {
      String header= 'ph0wn{';
      String footer = '}';
      // plain = list('Dart_is_soooo_opaque_isnt_it')
      // [ ord(i) ^ 0x43 for i in plain ]
      List <int> core = [7, 34, 49, 55, 28, 42, 48, 28,
           48, 44, 44, 44, 44, 28, 44, 51,
           34, 50, 54, 38, 28, 42, 48, 45,
           55, 28, 42, 55];
      int i = 0;

      for (i = 0; i < core.length; i++) {
        core[i] = core[i] ^ 0x43;
      }
      print('stage2: '+ header + String.fromCharCodes(core) + footer);
}

void xor_stage1() {
  Uint8List flag = Uint8List.fromList([98, 101, 97, 117, 116,
     105, 102, 117, 108, 45, 115, 116, 97, 103, 101, 49]);
  print('stage1: ' + String.fromCharCodes(flag));
}

void main(List<String> arguments) {
  xor_stage1();
  xor_stage2();
}
```

Like in C, the program begins with `main`. Then, we basically print 2 strings: one in `xor_stage1()`, creating a `String` object from a list of bytes, and one in `xor_stage2()` where the string is decrypted from an XOR-encrypted byte array.

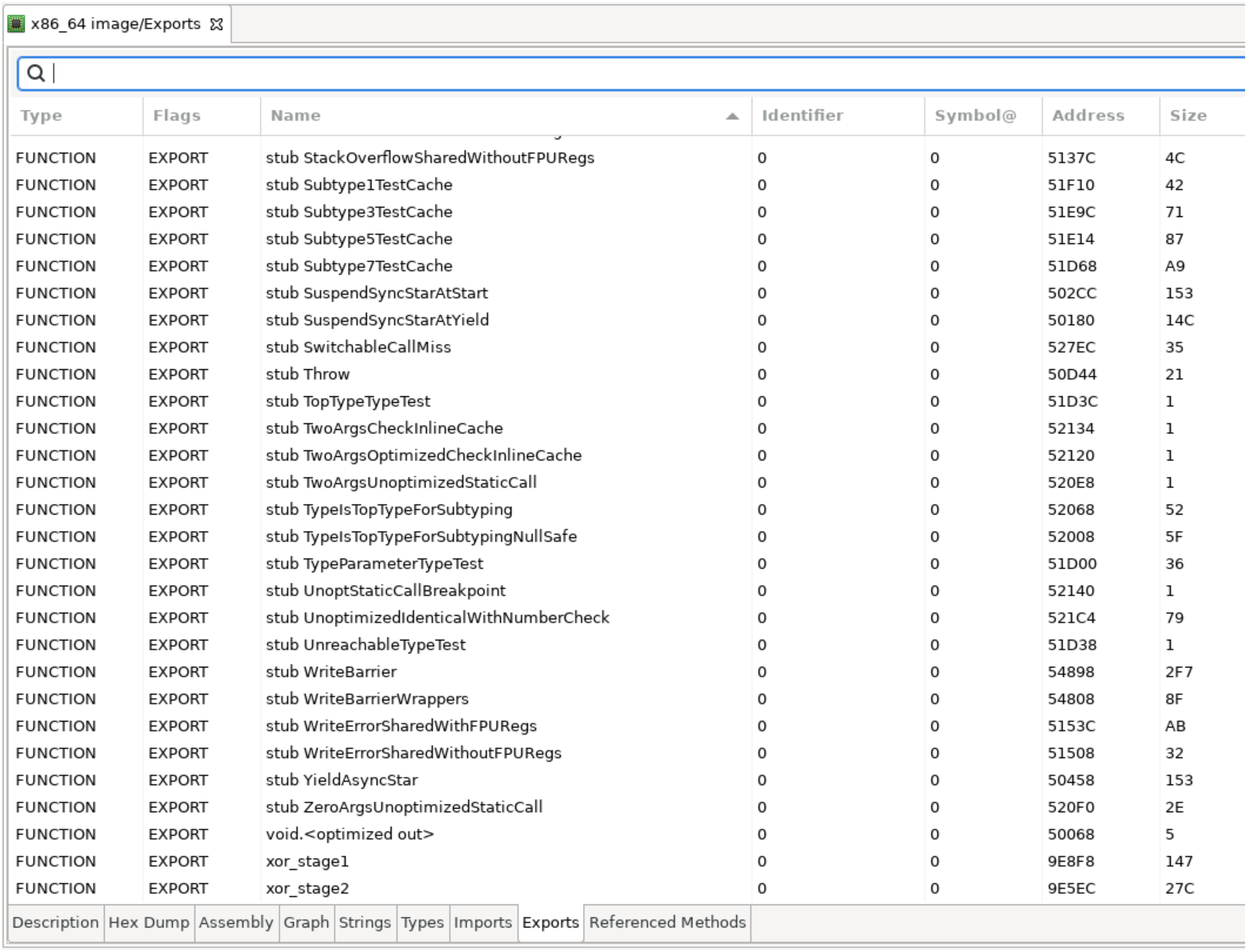
We compile an AOT snapshot (`dart compile aot-snapshot file`). When we run it, the output is the following:

```
stage1: beautiful-stage1
stage2: ph0wn{Dart_is_soooo_opaque_isnt_it}
```

What disassemblers understand of Dart programs

As we mentioned previously, Dart programs may be compiled for various platforms. Reverse engineering tools generally having better support for x86_64, we maximize our reversing chances by working over a Dart AOT snapshot for x86_64.

The Unix `strings` command very few interesting strings: `ph0wn{`, `stage1:`, `stage2:`, `xor_stage1`, `xor_stage2`, `main`, but **none of the strings built from byte arrays**. Ghidra v10.2.2 is completely lost and only manages to find function names. IDA Pro, Radare and JEB perform better: they find function names and correctly disassemble them.



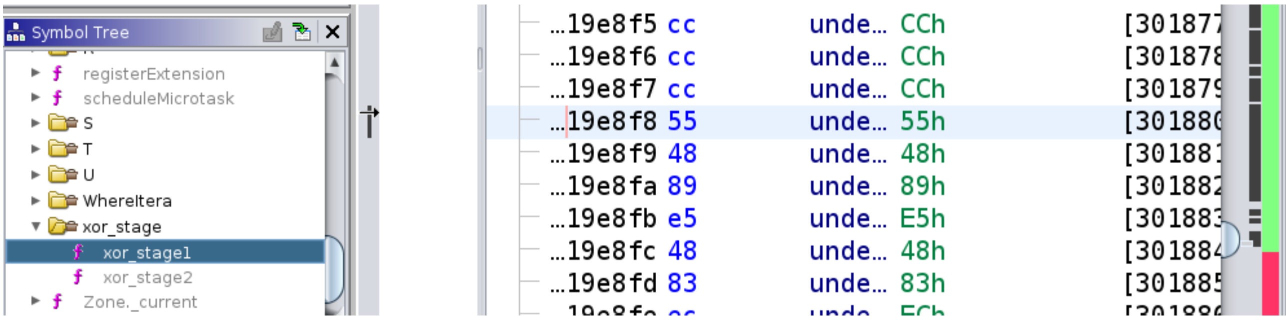
x86_64 image/Exports

Type	Flags	Name	Identifier	Symbol@	Address	Size
FUNCTION	EXPORT	stub StackOverflowSharedWithoutFPURegs	0	0	5137C	4C
FUNCTION	EXPORT	stub Subtype1TestCache	0	0	51F10	42
FUNCTION	EXPORT	stub Subtype3TestCache	0	0	51E9C	71
FUNCTION	EXPORT	stub Subtype5TestCache	0	0	51E14	87
FUNCTION	EXPORT	stub Subtype7TestCache	0	0	51D68	A9
FUNCTION	EXPORT	stub SuspendSyncStarAtStart	0	0	502CC	153
FUNCTION	EXPORT	stub SuspendSyncStarAtYield	0	0	50180	14C
FUNCTION	EXPORT	stub SwitchableCallMiss	0	0	527EC	35
FUNCTION	EXPORT	stub Throw	0	0	50D44	21
FUNCTION	EXPORT	stub TopTypeTypeTest	0	0	51D3C	1
FUNCTION	EXPORT	stub TwoArgsCheckInlineCache	0	0	52134	1
FUNCTION	EXPORT	stub TwoArgsOptimizedCheckInlineCache	0	0	52120	1
FUNCTION	EXPORT	stub TwoArgsUnoptimizedStaticCall	0	0	520E8	1
FUNCTION	EXPORT	stub TypeIsTopTypeForSubtyping	0	0	52068	52
FUNCTION	EXPORT	stub TypeIsTopTypeForSubtypingNullSafe	0	0	52008	5F
FUNCTION	EXPORT	stub TypeParameterTypeTest	0	0	51D00	36
FUNCTION	EXPORT	stub UnoptStaticCallBreakpoint	0	0	52140	1
FUNCTION	EXPORT	stub UnoptimizedIdenticalWithNumberCheck	0	0	521C4	79
FUNCTION	EXPORT	stub UnreachableTypeTest	0	0	51D38	1
FUNCTION	EXPORT	stub WriteBarrier	0	0	54898	2F7
FUNCTION	EXPORT	stub WriteBarrierWrappers	0	0	54808	8F
FUNCTION	EXPORT	stub WriteErrorSharedWithFPURegs	0	0	5153C	AB
FUNCTION	EXPORT	stub WriteErrorSharedWithoutFPURegs	0	0	51508	32
FUNCTION	EXPORT	stub YieldAsyncStar	0	0	50458	153
FUNCTION	EXPORT	stub ZeroArgsUnoptimizedStaticCall	0	0	520F0	2E
FUNCTION	EXPORT	void.<optimized out>	0	0	50068	5
FUNCTION	EXPORT	xor_stage1	0	0	9E8F8	147
FUNCTION	EXPORT	xor_stage2	0	0	9E5EC	27C

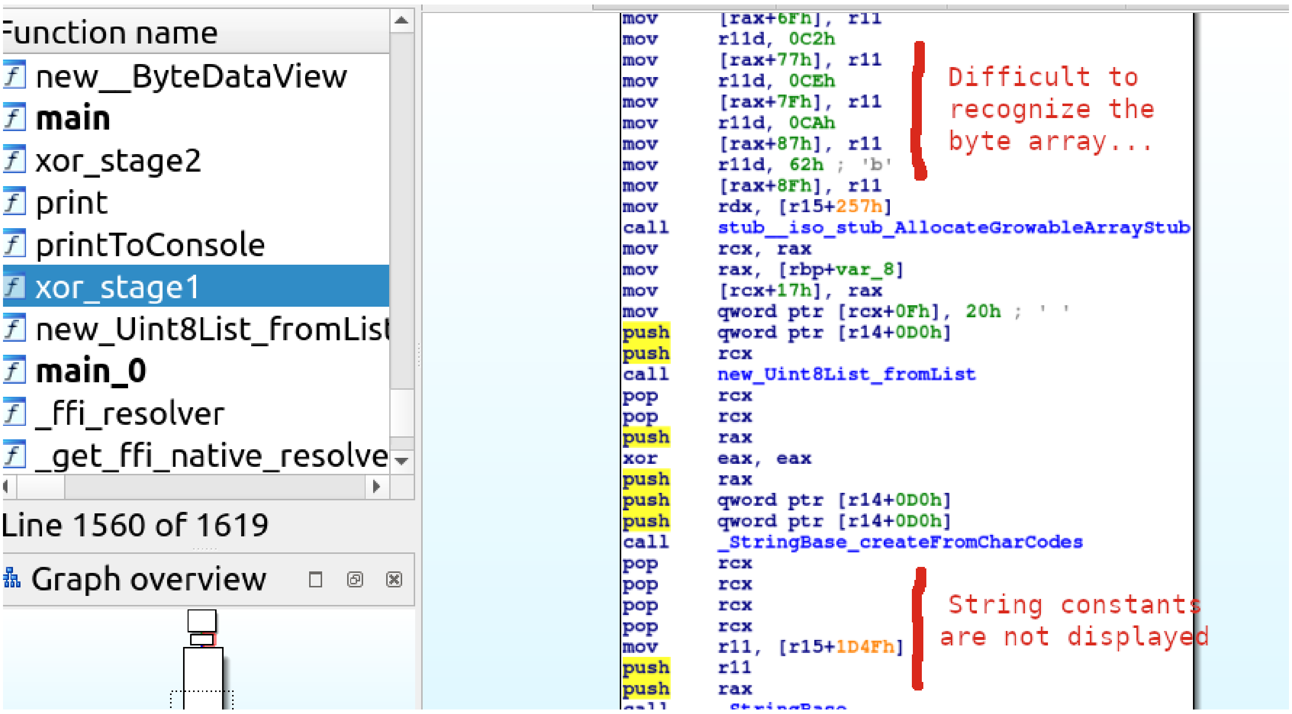
Description | Hex Dump | Assembly | Graph | Strings | Types | Imports | Exports | Referenced Methods

JEB lists Dart functions in the Exports tab. Radare 2 lists function names with `afl`

They however remain unable to recover strings and byte arrays. For example, in the screenshots below, no disassembler recognizes use of the string `stage1:`. The byte array is not recognizable either: each individual byte do not match the bytes of the source code... As for decompilation, only JEB manages to produce a decent output - yet, it remains mostly useless.



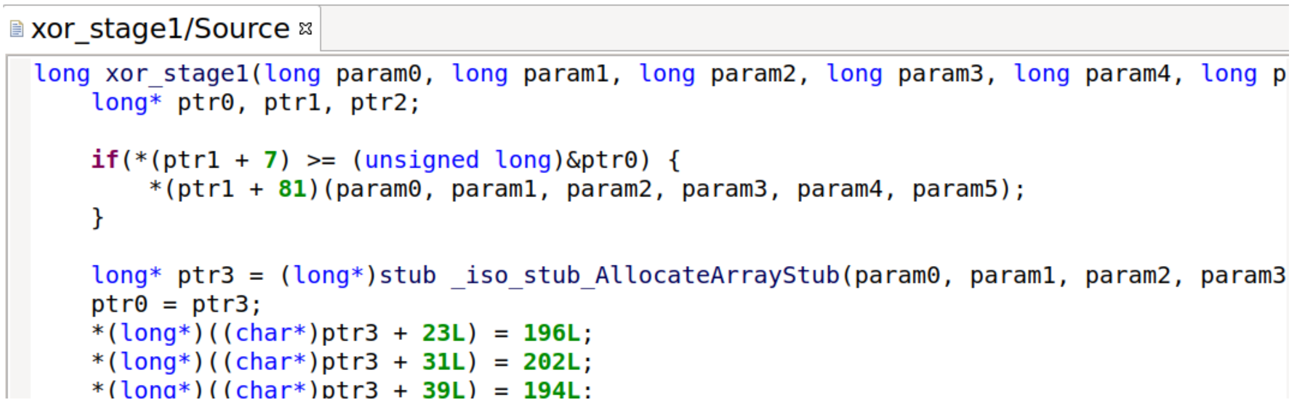
Ghidra is completely lost when disassembling a Dart snapshot



Disassembly of xor_stage1 by IDA Pro. Note the byte array is difficult to recognize: the bytes do not match.

```
; arg int64_t arg10 @ xmm3
; var int64_t var_8h @ rbp-0x8
0x0009e8f8      55              push rbp                ; xorpi.dart:0
0x0009e8f9      4889e5          mov rbp, rsp
0x0009e8fc      4883ec08        sub rsp, 8
0x0009e900      493b6638        cmp rsp, qword [r14 + 0x38]
0x0009e904      0f8629010000    jbe 0x9ea33
; CODE XREF from sym.xor_stage1 @ 0x9ea3a(x)
0x0009e90a      498b9f570200.   mov rbx, qword [r15 + 0x257]
0x0009e911      41ba20000000    mov r10d, 0x20          ; "@"
0x0009e917      e8f06dffff      call sym.stub__iso_stub_AllocateArrayStub
0x0009e91c      488945f8        mov qword [var_8h], rax
0x0009e920      41bbc4000000    mov r11d, 0xc4
0x0009e926      4c895817        mov qword [rax + 0x17], r11
0x0009e92a      41bbca000000    mov r11d, 0xca
0x0009e930      4c89581f        mov qword [rax + 0x1f], r11
0x0009e934      41bbc2000000    mov r11d, 0xc2
0x0009e93a      4c895827        mov qword [rax + 0x27], r11
0x0009e93e      41bbea000000    mov r11d, 0xea
0x0009e944      4c89582f        mov qword [rax + 0x2f], r11
0x0009e948      41bbe8000000    mov r11d, 0xe8
0x0009e94e      4c895837        mov qword [rax + 0x37], r11
0x0009e952      41bbd2000000    mov r11d, 0xd2
```

Disassembly of the xor_stage1 function by Radare 2. The assembly is correct, but raw.



```
xor_stage1/Source
long xor_stage1(long param0, long param1, long param2, long param3, long param4, long p
    long* ptr0, ptr1, ptr2;

    if(*(ptr1 + 7) >= (unsigned long)&ptr0) {
        *(ptr1 + 81)(param0, param1, param2, param3, param4, param5);
    }

    long* ptr3 = (long*)stub__iso_stub_AllocateArrayStub(param0, param1, param2, param3
    ptr0 = ptr3;
    *(long*)((char*)ptr3 + 23L) = 196L;
    *(long*)((char*)ptr3 + 31L) = 202L;
    *(long*)((char*)ptr3 + 39L) = 194L;
```

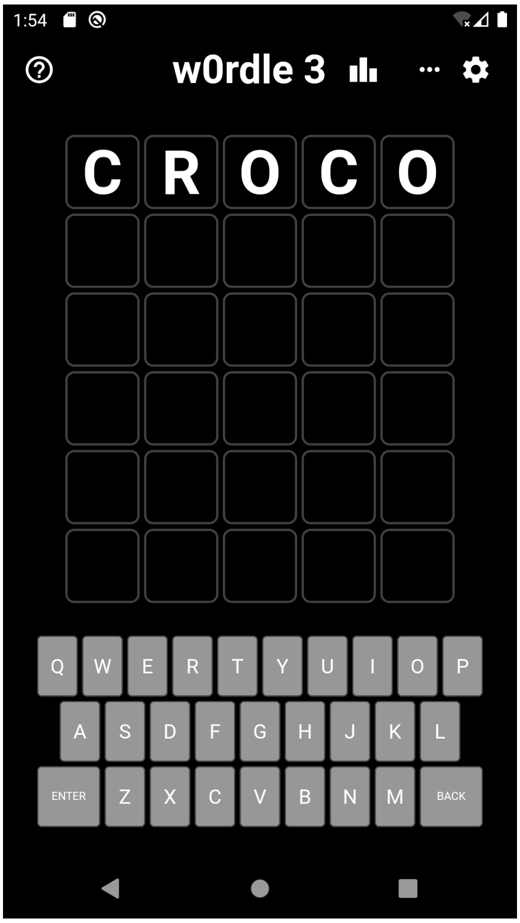
Decompiled code for xor_stage1, by JEB. Though far from excellent, it's the best decompiled output we can hope for.

Disassembler	Function names	Function disassembly	User strings	Byte array recovery	Decompiler output
Ghidra v10.2.2	Yes	No	No	No	Not usable
IDA Pro 8.2	Yes	Yes	No	No	-
Radare 5.8.7	Yes	Yes	No	No	Not usable
JEB 4.31.0	Yes	Yes	No	No	Basic

Status for reversing Flutter application

w0rdle stage 3

In a second attempt, we try to reverse a Flutter application. For that, we use a challenge I released in January 2023 and which remains, up to now, unsolved. The Flutter application is downloadable from (Apvrille 2023e). Its SHA256 hash is `99068666d845bdf2513bca731ad95e4b21f6d0460925beaf881a649ce16a9c2e`. The application features a customized wordle game: this is a game where you have 6 chances to guess a 5-letter word. It is implemented using Flutter and offers 3 reverse engineering stages: stage 1 and stage2 were solved during Ph0wn CTF 2022 and they can be solved with no to little Flutter knowledge. *Stage 3*, however, is meant to require Flutter reverse engineering skills.



The wordle game, stage 3, on Android.

In stage 3, the word to guess consists of characters which are not present in the game's keyboard. Contrary to stage 2, it is difficult to modify the word to guess (because it is encrypted in the code). Thus, there is no way to win by playing the game. We will investigate a solution where we *reverse engineer* the function that displays the flag. It is (intentionally) extremely similar to the Dart program we built previously.

```
import 'package:flutterdle/domain.dart';
...

class Flutterdle {
String _winningMessage() {
```

```
    if (stage == 2) {
      // return flag for stage 2
      ...
    }

    if (stage == 3) {
      String header = 'ph0wn{';
      String footer = '}';
      // encrypted body of the flag
      List <int> core = [81, 94, 89, 83, 104, ...];
      int i = 0;

      for (i = 0; i < core.length; i++) {
        core[i] = core[i] ^ 0x37;
      }
      print(header + String.fromCharCodes(core) + footer);
      return header + String.fromCharCodes(core) + footer;
    }

    // else, stage 1
    String flag = 'FLAG for STAGE1';
    print(flag); // we want this to print on logcat
    return flag;
}
```

The flag is decrypted from a very basic XOR algorithm. **This would typically represent no issue to reverse engineer**. However, we'll see in the next paragraph Flutter makes things complicated.

Disassemblers understand less from Flutter apps

As we mentioned in Flutter application's anatomy, the library to reverse is `libapp.so`. Again, to maximize chances of successful disassembly, we attempt to reverse the *x86_64* version.

For stage 3, the `strings` command finds `ph0wn`{ and `_winningMessage@739388852`. There are no other interesting strings for stage 3. We load the library in a disassembler and analyze. It's worse than for a Dart program: disassemblers do not find *any* function name [^1].

[^1] JEB manages to recover function names in some cases, but it did not recover them for w0rdle 3.

In the example below, we ask Radare 2 to list functions (`afl`). All have a dummy name `fcn.address`.

```
> afl
...
0x0036f024    14     232 fcn.0036f024
0x00332678    14     232 fcn.00332678
0x00332760     9     105 fcn.00332760
0x003328b4     9     105 fcn.003328b4
0x00178f1c     1      67 fcn.00178f1c
```

There are many functions. It's unlikely we'll pick one randomly and by chance disassemble `_winningMessage` . If ever we did, we'd notice disassemblers have difficulties finding the correct boundaries. The assembly below is produced by Radare 2, who failed to understand the beginning of `_winningMessage` is at 0x003**ce09c**.

```
0x003ce08b      e9d7fdffff     jmp 0x3cde67
; CODE XREF from fcn.003cde3c @ 0x3cde82(x)
0x003ce090      498bb72f9d00.  mov rsi, qword [r15 + 0x9d2f]
0x003ce097      e838500200     call fcn.003f30d4
; CALL XREF from fcn.003cde3c @ 0x3cdff5(x)
0x003ce09c      55             push rbp
0x003ce09d      4889e5         mov rbp, rsp
0x003ce0a0      4883ec10       sub rsp, 0x10
0x003ce0a4      493b6638       cmp rsp, qword [r14 + 0x38]
0f8697040000   jbe 0x3ce545
; CODE XREF from fcn.003cde3c @ 0x3ce54c(x)
0x003ce0ae      488b4510       mov rax, qword [var_10h_2]
0x003ce0b2      488b482b       mov rcx, qword [rax + 0x2b]
0x003ce0b6      4883f902       cmp rcx, 2
```

This is due to Flutter's *obfuscation*. By default, it *renames* all Dart module names, class names and function names for release builds on Android and iOS. While this is not strong obfuscation (no encryption, no modification of instruction), it hardens reverse engineering.

Improving Reverse Engineering

> In the rest of this paper, we use **Radare 5.8.7** and **JEB 4.31.0**, which are the current versions at the time of writing this article. I am in contact with both authors / contributors, both disassemblers are likely to improve in the next few months.

The Object Pool and its impact

In our initial attempts, we are able to find string constants (`strings`, or `iz` command in Radare) but disassemblers are unable to annotate code when they are used. This is **because Dart assembly accesses strings indirectly through an Object Pool**.

The Object Pool is a table-like structure which stores and references frequently used objects, immediates and constants within a Dart program. (Batteux 2022) explains how the Object Pool is serialized in a AOT snapshot, and how to access objects at runtime. Basically, all objects are serialized in a snapshot. The Object Pool itself is an object and is serialized too. At runtime, each object - including the object pool - is de-serialized. Then, to access a given object, the code provides its pool index. The Object Pool looks up for the corresponding object and returns it.

To illustrate the mechanism, let's take an example from our Dart program.

```
void xor_stage2() {
   String header= 'ph0wn{'; // access to constant "ph0wn{"
   ...
}
```

When we affect constant `ph0wn{`, the assembly actually says "please get me index XXX". The object pool looks up this particular entry and returns the corresponding object `ph0wn{`. **Dart uses a dedicated register for requests to the Object Pool**: it's r15 for x86_64, r5 for Aarch32, and x27 for Aarch64 (see Appendix register layout)

```
; get pool object index 936
0x0009e7f0       4d8b9f471d00.  mov r11, qword [r15 + 0x1d47]
0x0009e7f7       4153           push r11
```

(Software 2022) explains the pool index is computed from `address //8`. So, the assembly above asks for pool index *936*. If the pool index is high, computing its address may be split on several instructions. For example, the following instructions compute address 0x8000 + 0xbb8 = 0x8bb8. It fetches pool index 4471 (0x8bb8 // 8).

```
; x17 = x27 + (8 << 12)
add x17, x27, 8, lsl 12
; x17 = x17 + 0xbb8
ldr x17, [x17, 0xbb8]
```

This explains our reverse engineering difficulties: the strings are visible in the ELF executable because the string constant is serialized at snapshot creation, but disassemblers are unable to detect their access and properly annotate code with a message like `/* accessing pool string: ph0wn{ */`[1] because they are not aware of the Object Pool, the use of a specific register and the computation of the pool index.

Architecture	Load Pool Object	Opcode bytes
x86_64	`mov rbx, qword ptr ds:[r15+847h]`[2]	498B9F470800
arm7eabi	`ldr r1, [r5, #433h]`	331495E5
arm64	`ldr x16, [x27, #7D50h]`	70AB7EF9

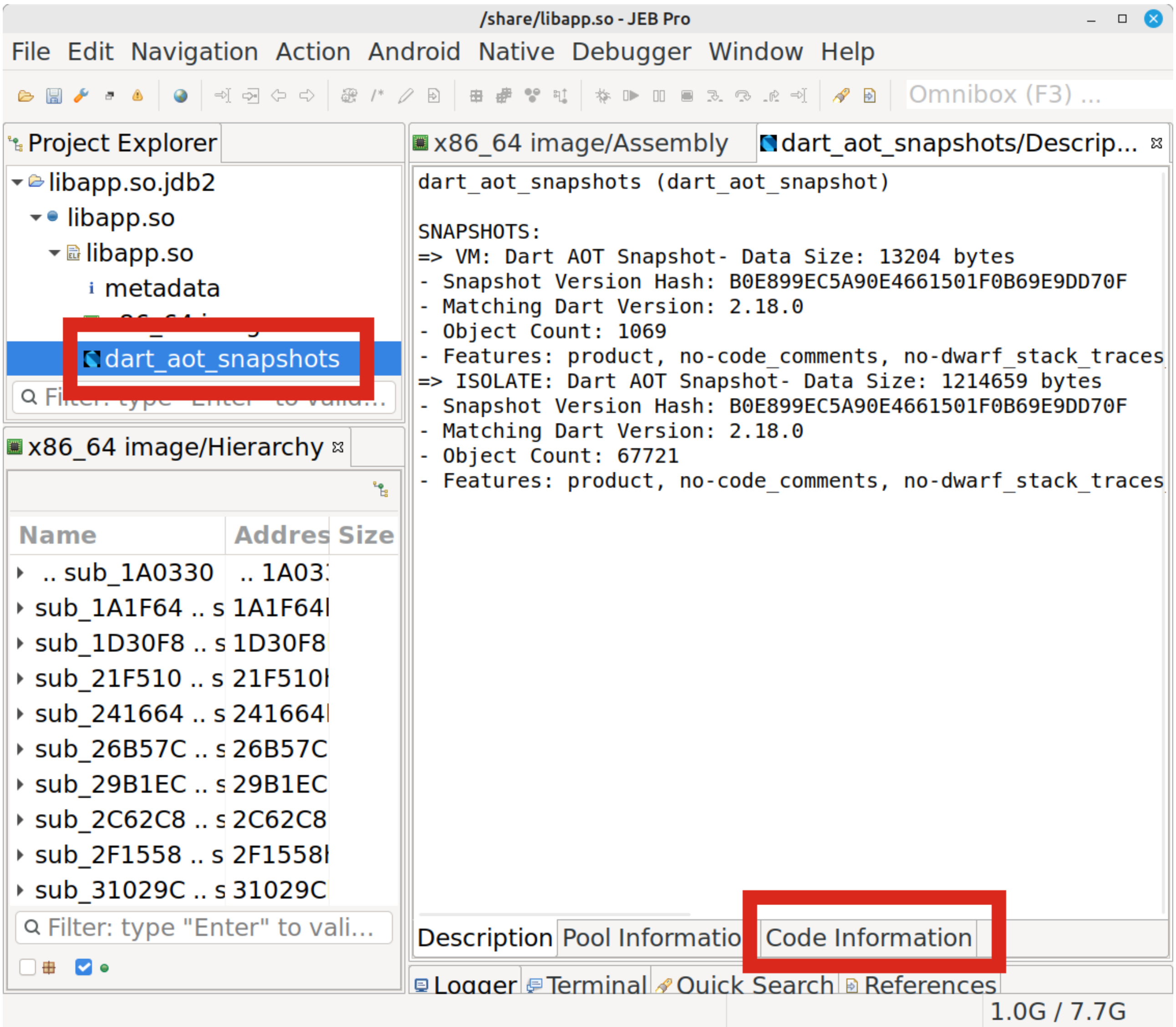
How to locate Flutter functions with JEB

JEB manages to recover Flutter application function names and addresses in a few cases, and provides the necessary elements to retrieve names manually in other cases. This features stems from JEB's ability to parse the Object Pool[3]. Click on the Dart AOT snapshot, then on the "Code Information".

[1] JEB does it for the x86_64 platform. However, there are bugs: it doesn't recognize all strings, and the feature does not work for ARM platforms.

[2] With Radare2, the produced instructions differ slightly: `mov rbx, qword [r15 + 0x847]`

[3] Currently, there are bugs, especially for ARM platforms.

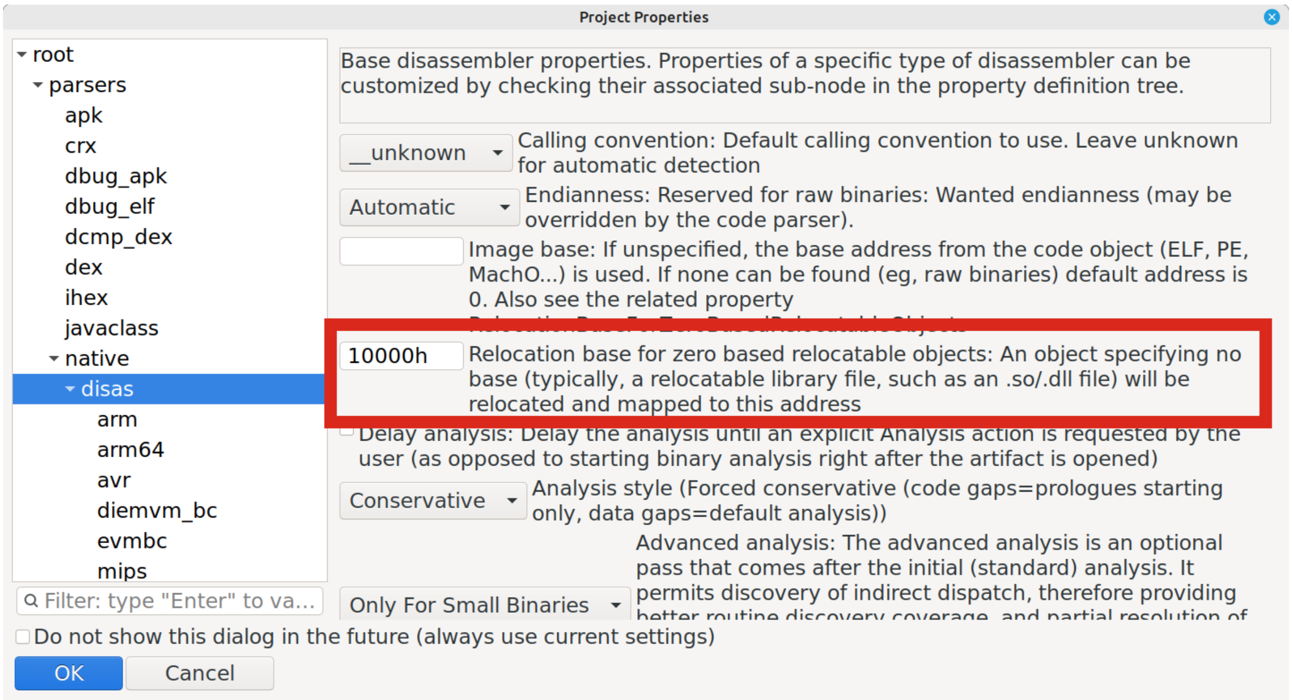


Accessing Flutter function addresses with JEB: head to Code Information

Then, search for `_winningMessage`. The list displays functions and their relative virtual offset.

```
updateAfterSuccessfulGuess @ 0x3CDE3C
_winningMessage@739388852 @ 0x3CE09C
_updateStats@739388852 @ 0x3CE56C
```

So, `_winningMessage` should be located at 0x3CE09C. Except JEB uses a default setting where all objects with unknown base address are automatically re-located at 0x10000. Consequently, `_winningMessage` is not at 0x3CE09C but at 0x3**D**E09C.



Default relocation address in JEB. This is customizable in Options/Backend properties: root/parsers/native/disas/

```
sub_3DE09C      proc

                push    rbp                             ; xref: sub
                mov     rbp, rsp
                sub     rsp, 10h
                cmp     rsp, qword ptr ds:[r14+38h]
                jbe     loc_3DE545
loc_3DE0AE:
                mov     rax, qword ptr ss:[rbp+10h]     ; xref: sub
                mov     rcx, qword ptr ds:[rax+2Bh]
                cmp     rcx, 2
                jnz     loc_3DE2FC
                mov     rbx, qword ptr ds:[r15+A5Fh]
                mov     r10d, 38h
                call    sub_40277C
                mov     qword ptr ss:[rbp-8], rax
                mov     r11d, Eh
                mov     dword ptr ds:[rax+Fh], r11d
                mov     r11d, 44h
                mov     dword ptr ds:[rax+13h], r11d
```

Disassembly of `_winningMessage()`*. The address of the function is computed from the snapshot code information added to the base relocation address.*

The encoding of Small Integers, and its impact to byte arrays

The assembly for a byte array consists in the following steps:

1. Allocate an array of the necessary size.
2. Load each byte.
3. Create the String. This consists in a call to `createFromCharCodes` which is called by `String.fromCharCodes`

There is an unexpected challenge: **Dart uses a special representation for bytes**. It represents differently:

- Small Integers (SMI). Those are integers which can fit on 31 bits (for 32-bit architectures) or 63 bits (for 64-bit architecture). They are represented with their least significant bit set to `0`. The value is encoded on the remaining bits. Example: to push a SMI value of 5 (0x5) to a register, we do not move `mov ip, 0x05` but `mov ip, 0x0a`.
- Medium Integers (Mint). Those which need more bits than 31/63.

For example, let's suppose we have this array ("Pico was there") in Dart:

```
List <int> message = [ 0x50, 0x69, 0x63, 0x6f, 0x20, 0x77, 0x61, 0x73,
      0x20, 0x74, 0x68, 0x65, 0x72, 0x65];
```

We allocate an array of 14 bytes. To represent SMI 14 (`1110` in binary), we set the least significant bit to 0 and obtain `11100`, which consists in doubling the value. The result equals 28 in decimal, or 0x1c in hexadecimal. Below, the assembly pushes 0x1c to the lower 32 bits of general purpose register r10.

```
 ; size of array = 0x1c / 2 = 14
mov        r10d, 1Ch
call       stub _iso_stub_AllocateArrayStub
```

The same occurs for each byte of the array. We are not pushing 0x50, 0x69, 0x63, 0x6f as expected, but their double, 0xa0, 0xd2, 0xc6, 0xde, to the data segment *rax+x*. Rax is a general purpose register.

```
000AE650                  mov        r11d, A0h                      ; P
000AE656                  mov        qword ptr ds:[rax+17h], r11
000AE65A                  mov        r11d, D2h                      ; i
000AE660                  mov        qword ptr ds:[rax+1Fh], r11
000AE664                  mov        r11d, C6h                      ; c
000AE66A                  mov        qword ptr ds:[rax+27h], r11
000AE66E                  mov        r11d, DEh                      ; o
```

and finally, create the String:

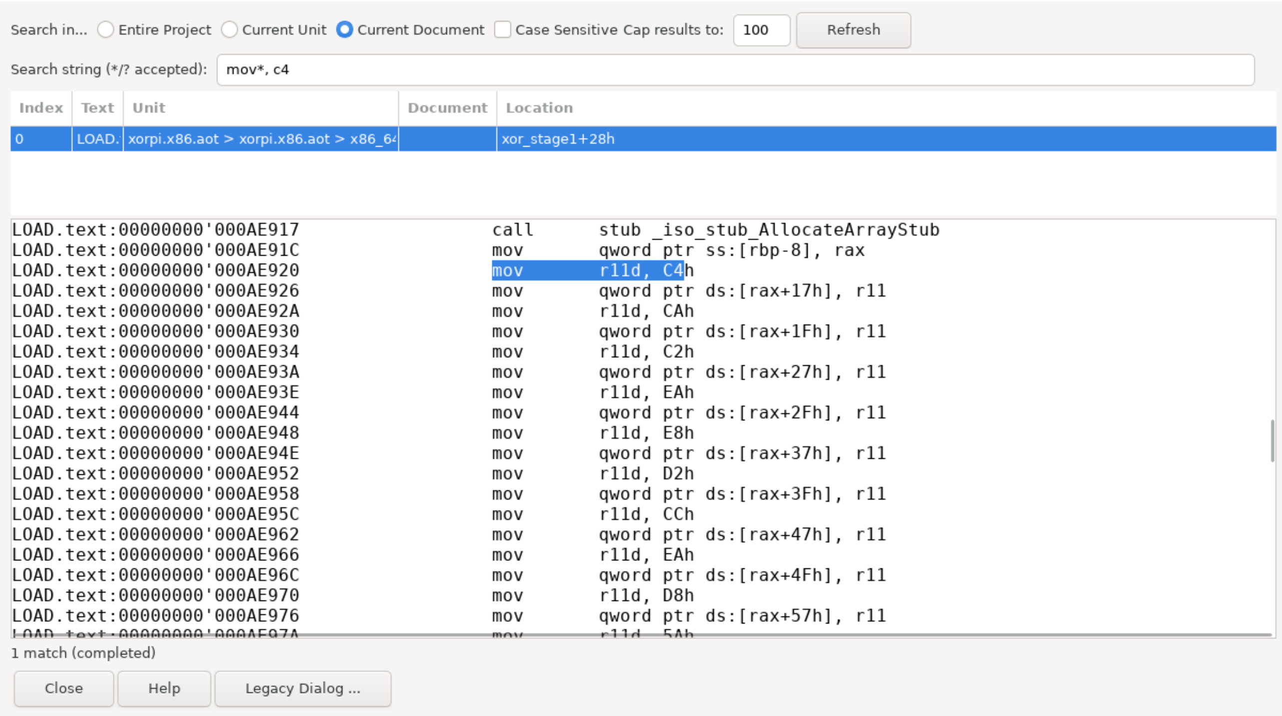
```
000AE70D                  call       _StringBase.createFromCharCodes
```

If we need to concatenate strings (e.g. `'stage2: '+ header`), the assembly is straight forward: get the parts to concatenate and call the String concatenation function. In the code below, we are concatenating 2 strings from the object pool, respectively indexes 938 and 939.

```
mov        r11, qword ptr ds:[r15+1D57h]
```

```
push        r11
mov         r11, qword ptr ds:[r15+1D5Fh]
push        r11
call        _StringBase.+
```

Knowing about SMI representation, the byte array of our Dart program or our Flutter application are easier to locate. Our Dart program loads `[98, 101, 97,...]`, so we search for `mov something, C4`.



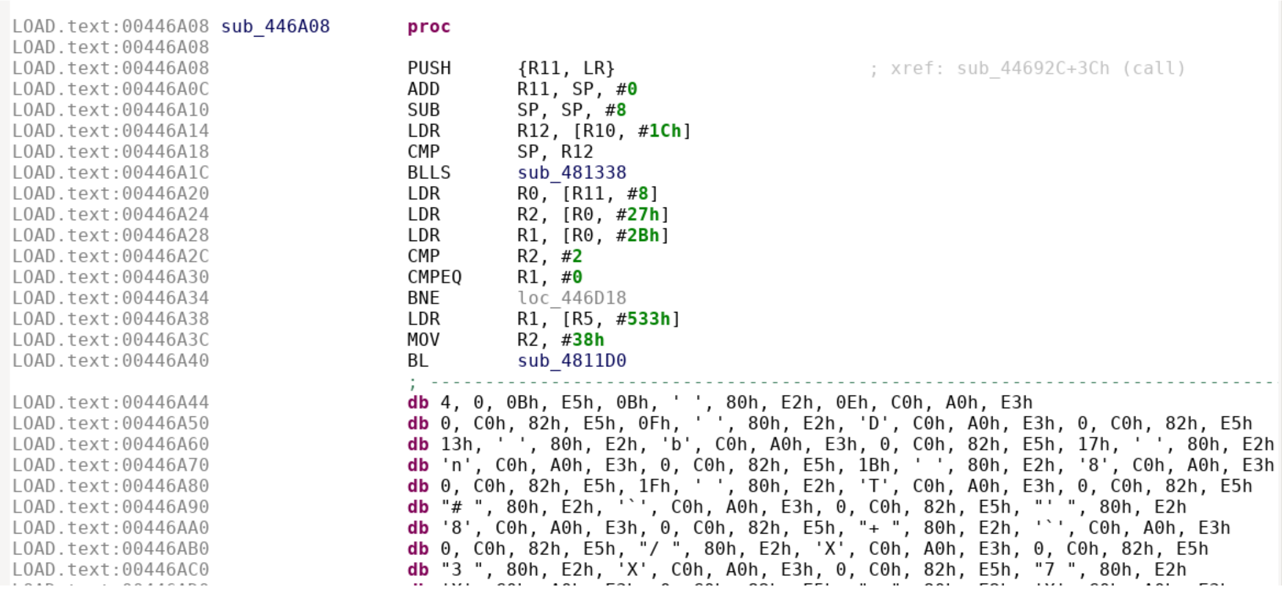
JEB understands joker characters in search patterns. We can search for the first SMI of our byte array, represented as 0xC4

Same, in stage 3 of our Flutter application, we create a byte array `[81, 94, 89...]`. This will be encoded as 0xa2, 0xbc, 0xb2. We can ask the disassembler to search for `mov r11d, 0xa2`. This is how to do it with Radare 2, then we can confirm which hit is correct by disassembling the next few instructions:

```
[0x003ce09c]> /ad mov r11d, 0xa2
0x003ce31e          41bba2000000  mov r11d, 0xa2
0x003ce382          41bba2000000  mov r11d, 0xa2
0x003ce38c          41bba2000000  mov r11d, 0xa2
0x0006f329          41bb800225a2  mov r11d, 0xa2250280
[0x00170000]> pd 4 @ 0x003ce31e
;-- hit0_1:
0x003ce31e      41bba2000000   mov r11d, 0xa2
0x003ce324      4489580f       mov dword [rax + 0xf], r11d
0x003ce328      41bbbc000000   mov r11d, 0xbc
```

Custom register for Dart VM thread

In some cases - particularly for Aarch32 and Aarch64 - your favorite disassembler **fails to correctly recognizes the boundaries of a function**. In that situation, we need to manually define the beginning of the function. To do so, it helps to know what the beginning of a Dart function looks like (see Appendix Prologue Snippet).



```
LOAD.text:00446A08 sub_446A08       proc
LOAD.text:00446A08
LOAD.text:00446A08                  PUSH      {R11, LR}              ; xref: sub_44692C+3Ch (call)
LOAD.text:00446A0C                  ADD       R11, SP, #0
LOAD.text:00446A10                  SUB       SP, SP, #8
LOAD.text:00446A14                  LDR       R12, [R10, #1Ch]
LOAD.text:00446A18                  CMP       SP, R12
LOAD.text:00446A1C                  BLLS      sub_481338
LOAD.text:00446A20                  LDR       R0, [R11, #8]
LOAD.text:00446A24                  LDR       R2, [R0, #27h]
LOAD.text:00446A28                  LDR       R1, [R0, #2Bh]
LOAD.text:00446A2C                  CMP       R2, #2
LOAD.text:00446A30                  CMPEQ     R1, #0
LOAD.text:00446A34                  BNE       loc_446D18
LOAD.text:00446A38                  LDR       R1, [R5, #533h]
LOAD.text:00446A3C                  MOV       R2, #38h
LOAD.text:00446A40                  BL        sub_4811D0
                                    ; ---------------------------------------------------------------
LOAD.text:00446A44                  db 4, 0, 0Bh, E5h, 0Bh, ' ', 80h, E2h, 0Eh, C0h, A0h, E3h
LOAD.text:00446A50                  db 0, C0h, 82h, E5h, 0Fh, ' ', 80h, E2h, 'D', C0h, A0h, E3h, 0, C0h, 82h, E5h
LOAD.text:00446A60                  db 13h, ' ', 80h, E2h, 'b', C0h, A0h, E3h, 0, C0h, 82h, E5h, 17h, ' ', 80h, E2h
LOAD.text:00446A70                  db 'n', C0h, A0h, E3h, 0, C0h, 82h, E5h, 1Bh, ' ', 80h, E2h, '8', C0h, A0h, E3h
LOAD.text:00446A80                  db 0, C0h, 82h, E5h, 1Fh, ' ', 80h, E2h, 'T', C0h, A0h, E3h, 0, C0h, 82h, E5h
LOAD.text:00446A90                  db "# ", 80h, E2h, '`', C0h, A0h, E3h, 0, C0h, 82h, E5h, "' ", 80h, E2h
LOAD.text:00446AA0                  db '8', C0h, A0h, E3h, 0, C0h, 82h, E5h, "+ ", 80h, E2h, '`', C0h, A0h, E3h
LOAD.text:00446AB0                  db 0, C0h, 82h, E5h, "/ ", 80h, E2h, 'X', C0h, A0h, E3h, 0, C0h, 82h, E5h
LOAD.text:00446AC0                  db "3 ", 80h, E2h, 'X', C0h, A0h, E3h, 0, C0h, 82h, E5h, "7 ", 80h, E2h
```

JEB 4.31.0 does not detect 0x446A44 and further as code. This needs to be fixed manually by telling it is is code (C).

As usual, a function begins by (1) saving various registers on the stack, (2) moving the frame pointer and (3) allocating necessary space on the stack.

In addition, in Dart assembly code, there is a **stack overflow check**. The assembly **compares the stack pointer with a register Dart dedicates to the current VM thread pointer**. Tthis register is different for each platform, see Appendix register layout. If an overflow is detected, the function branches to an error exit.

```
; push base pointer on the stack
push rbp
; the new value for the base pointer is the stack pointer
mov rbp, rsp
; allocate 16 bytes
sub rsp, 10h
; r14 holds the current Dart VM thread pointer
cmp rsp, qword [r14 + 0x38]
; if stack pointer is <= [r14 + 0x38 ]: jump stack overflow error
jbe 0x9e850
```

Dart's custom calling convention (ABI)

The assembly generated for a Dart program demonstrates an unconventional method of passing parameters to functions, which poses a challenge for disassemblers attempting to analyze the code.

For example, this is the assembly of a Dart program concatenating 2 object pool constants ("stage 2:" and "ph0wn{").

```
mov r11, qword [r15 + 0x1d3f]
push r11
mov r11, qword [r15 + 0x1d47]
```

```
push r11
call fcn.string_concat
```

Usually, on x86_64, the first few arguments of a function are passed in specific registers, and only subsequent arguments are pushed on the stack. For example, see here: the first four arguments on Microsoft Windows are passed using registers, the first six arguments on Unix (System V AMD64 ABI).

Dart does it differently and **passes all arguments on the stack** (`push r11` in both cases).

The same behavior occurs for AAarch32: the first string is loaded in LR register, the second in SB. **Both registers are pushed on the stack** (`stm sp, {sb, lr}`) and provided to the concatenation function.

```
ldr lr, [r5, 0xe9f] ; "stage2: "
ldr sb, [r5, 0xea3] ; "ph0wn{"
stm sp, {sb, lr}    ; push them on the stack
bl fcn.concat       ; concatenate strings
```

For Aarch64, the difference is even more striking because **Dart uses a custom stack register**, X15. This is because program counter (PC) and stack pointer (SP) aren't indexed registers on Aarch64, unlike AArch32. Then, same as other platforms, all arguments are pushed on the stack, whereas, normally, Aarch64 uses X0-X7 for the first 8 arguments, and only uses the stack for the additional arguments.

```
LDR        X16, [X27, #1C90h]              ; "stage2: "
LDR        X30, [X27, #1C98h]              ; ph0wn{
STP        X30, X16, [X15]                 ; we push them on the stack
BL         _StringBase.+                   ; concatenate both strings: '
    stage2: ph0wn{'
```

XOR encryption loop

The assembly of a Dart (or Flutter) XOR encryption loop is not difficult to identify, but interesting to read in detail because it shows the differences the compiler needs to perform between *small* integers and *medium* integers.

This is our program's `xor_stage2` loop Dart source code, and after, its corresponding assembly for x86_64.

```
      for (i = 0; i < core.length; i++) {
        core[i] = core[i] ^ 0x43;
      }
```

The counter is represented by register `rdi`. It is initialized to 0 (XOR-ing with itself is a common way to null a value). The next 2 lines are specific to Dart: it checks for stack overflow, r14 register being dedicated to the pointer to running VM thread.

```
; initialize edi=0
xor        edi, edi
loop:
cmp        rsp, qword ptr ds:[r14+38h]
jbe        overflow_check2
```

Then, it checks for the loop's end condition. The byte array has 28 bytes (0x1c).

```
cmp        rdi, 1Ch
jnl        loop_finished
```

The following is specific to retrieving a SMI. Remember we stored in the array double its value. When we want to XOR it, the assembly pays attention to XOR it with the real value, i.e it divides it back by 2.

```
mov        rax, qword ptr ds:[rcx+8*rdi+17h] ; load core[i]
sar        rax, 1                            ; core[i] / 2
jnb        not_negative                      ; jump if not negative
mov        rax, qword ptr ds:[rax+rax+8]     ; we won't get here

not_negative:
mov        rdx, rax
xor        rdx, 43h                          ; core[i] ^ 0x43
```

The compiler has some extra work: it does not know if the XOR result fits in a small or a medium integer. Consequently, it writes code for both cases. It tests if the result fits in a SMI by doubling it and checking if there's an overflow. If there's no overflow, this is a SMI. If it overflow, it must be stored in a Mint.

```
; rdx contains XOR result: core[i] ^ 0x43
mov        rax, rdx
; compute rax * 2
add        rax, rax
; no overflow: SMI case, overflow: Mint case.
jno        no_overflow
; Mint case: create Mint containing XOR result value
call       stub _iso_stub_AllocateMintSharedWithoutFPURegsStub
mov        qword ptr ds:[rax+7], rdx
```

Then, it stores the XOR result in `core[i]`:

```
no_overflow:
mov        rdx, rcx
; get address of core[i]
lea        r13, qword ptr ds:[rdx+8*rdi+17h]
; store XOR result in core[i]
mov        qword ptr ds:[r13], rax
```

The lines above are run in SMI and Mint case. In the case of a Mint, there is again some extra steps that we can skip as we have SMI. To differentiate between SMI and Mint, the assembly tests the least significant bit of rax (al): it should be 0 for a SMI, and 1 for a Mint. Finally, we increment the counter and loop back to the beginning.

```
test       al, 1
jz         loop_increment
; Mint case
...
loop_increment:
; increment counter
add        rdi, 1
; loop
jmp        loop
```

Dealing with Aarch32 and Aarch64

We have mostly dealt with x86_64 so far, because disassemblers are usually better for that platform. In theory, reversing Aarch32 and Aarch64 is not more difficult: it is just a matter of knowing the different instructions. The issue mostly comes from badly disassembled code.

```
LOAD.text:00446CF8 sub_446CF8        proc
LOAD.text:00446CF8
LOAD.text:00446CF8                   ADD      SP, SP, #8
LOAD.text:00446CFC                   PUSH     {R0}
LOAD.text:00446D00                   LDR      LR, [R5, #943h]
LOAD.text:00446D04                   PUSH     {LR}
LOAD.text:00446D08                   BL       sub_1DAEE0
LOAD.text:00446D08
LOAD.text:00446D08 sub_446CF8        endp
                                     ; ---------------------------------------------------------------------------

LOAD.text:00446D0C                   db 8, D0h, 8Dh, E2h
LOAD.text:00446D10                   db 0, D0h, 'K', E2h, 0, 88h, BDh, E8h
LOAD.text:00446D18 loc_446D18:
LOAD.text:00446D18                   MOV      R0, #0                     ; xref: sub_446A08+2Ch (cbr)
LOAD.text:00446D1C                   MOV      R3, #0
LOAD.text:00446D20                   CMP      R2, #3
LOAD.text:00446D24                   CMPEQ    R1, #0
LOAD.text:00446D28                   BNE      sub_446FEC
LOAD.text:00446D2C                   LDR      R1, [R5, #533h]
LOAD.text:00446D30                   MOV      R2, #2Ch
LOAD.text:00446D34                   BL       sub_4811D0
LOAD.text:00446D34
LOAD.text:00446D34 sub_446A08        endp
                                     ; ---------------------------------------------------------------------------

LOAD.text:00446D38                   db 4, 0, 0Bh, E5h, 0Bh, ' ', 80h, E2h
LOAD.text:00446D40                   db A2h, C0h, A0h, E3h, 0, C0h, 82h, E5h, 0Fh, ' ', 80h, E2h, BCh, C0h, A0h, E3h
LOAD.text:00446D50                   db 0, C0h, 82h, E5h, 13h, ' ', 80h, E2h, B2h, C0h, A0h, E3h, 0, C0h, 82h, E5h
LOAD.text:00446D60                   db 17h, ' ', 80h, E2h, A6h, C0h, A0h, E3h, 0, C0h, 82h, E5h, 1Bh, ' ', 80h, E2h
LOAD.text:00446D70                   db D0h, C0h, A0h, E3h, 0, C0h, 82h, E5h, 1Fh, ' ', 80h, E2h, BCh, C0h, A0h, E3h
LOAD.text:00446D80                   db 0, C0h, 82h, E5h, "# ", 80h, E2h, 86h, C0h, A0h, E3h, 0, C0h, 82h, E5h
LOAD.text:00446D90                   db "' ", 80h, E2h, D0h, C0h, A0h, E3h, 0, C0h, 82h, E5h, "+ ", 80h, E2h
LOAD.text:00446DA0                   db E6h, C0h, A0h, E3h, 0, C0h, 82h, E5h, "/ ", 80h, E2h, BCh, C0h, A0h, E3h
LOAD.text:00446DB0                   db 0, C0h, 82h, E5h, "3 ", 80h, E2h, A2h, C0h, A0h, E3h, 0, C0h, 82h, E5h
LOAD.text:00446DC0                   db "7 ", 80h, E2h, A2h, C0h, A0h, E3h, 0, C0h, 82h, E5h, "; ", 80h, E2h
LOAD.text:00446DD0                   db BCh, C0h, A0h, E3h, 0, C0h, 82h, E5h, "? ", 80h, E2h, A8h, C0h, A0h, E3h
LOAD.text:00446DE0                   db 0, C0h, 82h, E5h, "C ", 80h, E2h, 84h, C0h, A0h, E3h, 0, C0h, 82h, E5h
LOAD.text:00446DF0                   db "G ", 80h, E2h, B6h, C0h, A0h, E3h, 0, C0h, 82h, E5h, "K ", 80h, E2h
LOAD.text:00446E00                   db 86h, C0h, A0h, E3h, 0, C0h, 82h, E5h, "O ", 80h, E2h, D0h, C0h, A0h, E3h
LOAD.text:00446E10                   db 0, C0h, 82h, E5h, "S ", 80h, E2h, B2h, C0h, A0h, E3h, 0, C0h, 82h, E5h
LOAD.text:00446E20                   db "W ", 80h, E2h, 0Eh, C0h, A0h, E3h, 0, C0h, 82h, E5h, "[ ", 80h, E2h
LOAD.text:00446E30                   db 80h, C0h, A0h, E3h, 0, C0h, 82h, E5h, "_ ", 80h, E2h, 10h, C0h, A0h, E3h
LOAD.text:00446E40                   db 0, C0h, 82h, E5h, "35", 95h, E5h, FCh, E4h, 0, EBh, 0, '0', A0h, E1h
LOAD.text:00446E50                   db 4, ' ', 1Bh, E5h, 8, '0', 0Bh, E5h, 0Bh, ' ', 83h, E5h, 0, C0h, A0h, E3h
LOAD.text:00446E60                   db 7, C0h, 83h, E5h, ',', C0h, A0h, E3h, 7, C0h, 83h, E5h
```

Addresses 0x00446D38 are incorrectly disassembled. Define them as Code (C) to recover the instructions. In that case, it is storing SMIs in the byte array.

To identify the XOR loop, do not forget the instruction is named *EOR* (not XOR). Moreover, for Aarch64, pay attention that the XOR is done on a register not an immediate value.

```
MOVZ      X16, #37h     ; load XOR Key 0x43 in register X16
EOR       X5, X1, X16   ; XOR byte with register X16
```

Consequently, if you wish to search for the XOR loop for Aarch64 platform , you need to adjust search instructions. Searching for `EOR*37h` won't find anything, you need to search for `MOVZ*37h` instead (and check there is an EOR after).

Finally, as we mentioned earlier, pay attention to Dart's specific calling convention on AArch64: X15 is used as custom stack register + all arguments are pushed on the stack.

Flutter reverse examples

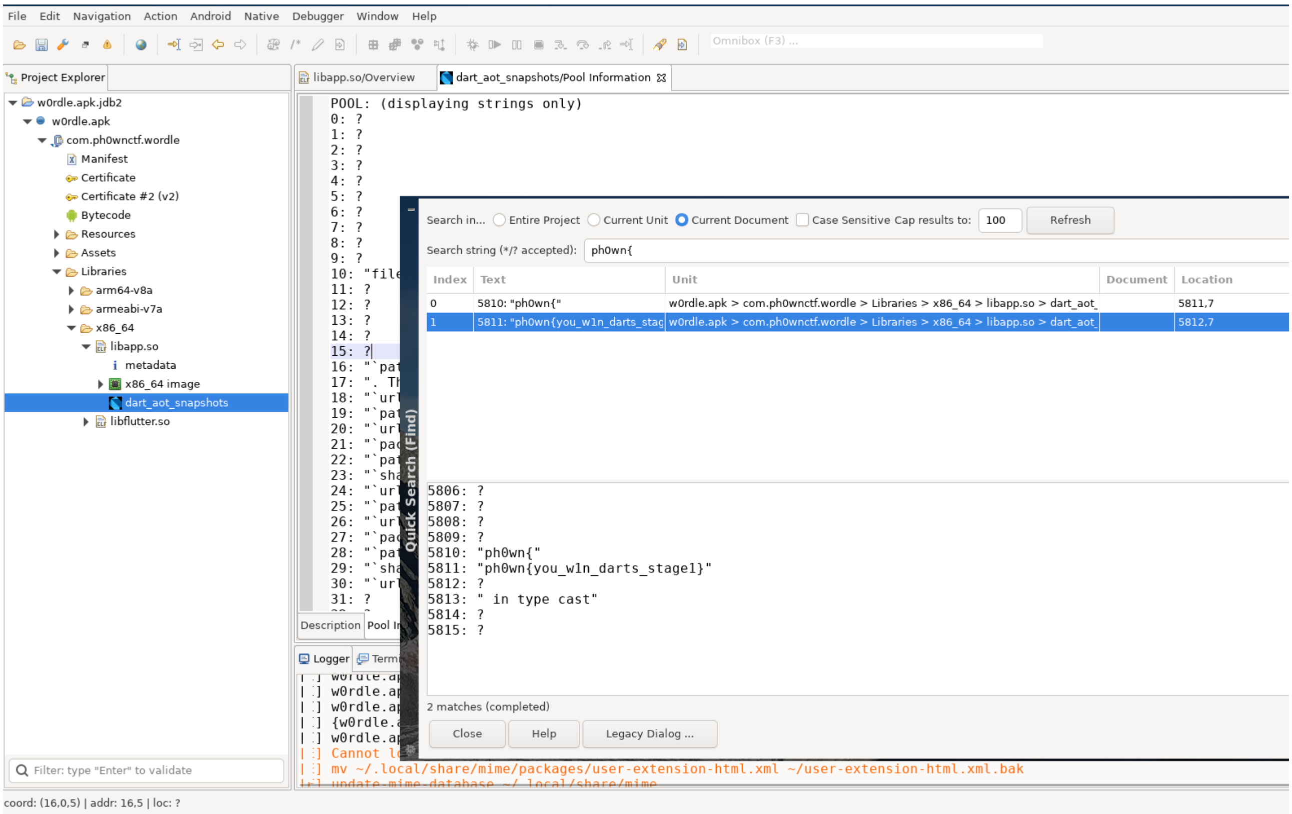
W0rdle stage 3

This paragraph is a quick write-up to solve w0rdle 3, using knowledge acquired through this research.

We quickly identify that `w0rdle.apk` uses Flutter, with the presence of `lib/<platform>/libapp.so` and `lib/<platform>/libflutter.so`. The strings show the stage 1 flag “ph0wn{you_w1n_darts_stage1}” and the beginning of another flag “ph0wn{”.

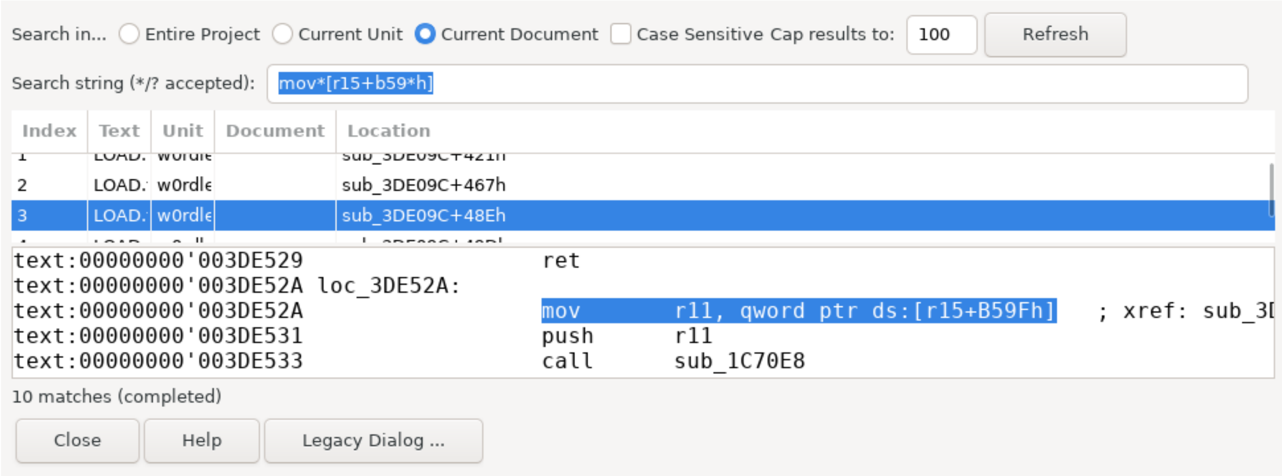
```
$ strings libapp.so | grep ph0wn
ph0wn
file:///home/axelle/prog/challenges/ph0wn2022/challenges/w0rdle/.dart_tool/
    flutter_build/dart_plugin_registrant.dart
ph0wn{
ph0wn w0rdle
ph0wn CTF is awesome
ph0wn{you_w1n_darts_stage1}
I love ph0wn CTF. I guessed w0rdle
```

We load the application in JEB and disassemble the x86_64 library. We search for the stage 1 flag with the Dart AOT snapshot element.



Stage 1 flag is in the Object Pool at index 5811

We search for code using this pool index (index 5810 with the beginning of a flag is interesting too). We know the instruction is going to be something like `mov *ds:[r15+XXh]`, where pool index = XX // 8. So XX can be 0xb598-0xb59f. We disassemble the x86_64 code (wait till analysis has finished) and search for that.



This piece of code uses the stage 1 flag.

```
; load ph0wn{you_w1n_darts_stage1}
003DE52A                    mov       r11, qword ptr ds:[r15+B59Fh]    ; xref:
    sub_3DE09C+266h (cbr)
003DE531                    push      r11
003DE533                    call      sub_1C70E8
```

We notice the same function uses constant `ph0wn{` with index 5810.

```
; load "ph0wn{" from pool objects
003DE503                    mov       r11, qword ptr ds:[r15+B597h]
003DE50A                    push      r11
003DE50C                    push      rax
003DE50D                    call      sub_196B98
```

We quickly identify 2 arrays in the function:

```
003DE0B6                    cmp       rcx, 2
003DE0BA                    jnz       loc_3DE2FC
003DE0C0                    mov       rbx, qword ptr ds:[r15+A5Fh]
; array size
003DE0C7                    mov       r10d, 38h
003DE0CD                    call      sub_40277C
003DE0D2                    mov       qword ptr ss:[rbp-8], rax
; load array bytes
003DE0D6                    mov       r11d, Eh
003DE0DC                    mov       dword ptr ds:[rax+Fh], r11d
003DE0E0                    mov       r11d, 44h
003DE0E6                    mov       dword ptr ds:[rax+13h], r11d
003DE0EA                    mov       r11d, 62h
...
003DE2FE                    cmp       rcx, 3
003DE302                    jnz       loc_3DE52A
003DE308                    mov       rbx, qword ptr ds:[r15+A5Fh]
; array 2 size
003DE30F                    mov       r10d, 2Ch
003DE315                    call      sub_40277C
003DE31A                    mov       qword ptr ss:[rbp-8], rax
; load bytes of array2
```

```
003DE31E                mov      r11d, A2h
003DE324                mov      dword ptr ds:[rax+Fh], r11d
003DE328                mov      r11d, BCh
003DE32E                mov      dword ptr ds:[rax+13h], r11d
003DE332                mov      r11d, B2h
```

The first array is XORed with 0x43:

```
003DE243                xor      rdx, 43h
003DE247                lea      rax, qword ptr ds:[rdx+rdx]
```

The second array is XORed with 0x37:

```
003DE44F                xor      rdx, 37h
```

We notice the first array is preceded by a comparison with 2, while the second array is compared to 3. We guess this is the stage indicator. So, stage 3 is the second array, and it is XORed with 0x37. We recover the SMIs: `[0xa2, 0xbc, 0xb2, ...]`, convert them to integers, apply the XOR and read the flag:

```
length = 0x2c // 2
table = [0xa2, 0xbc, 0xb2, 0xa6, 0xd0, 0xbc, 0x86, 0xd0, 0xe6, 0xbc, 0xa2, 0xa2, 0xbc, 0xa8, 0x84, 0xb6, 0x86,0xd0, 0xb2, 0x0e, 0x80, 0x10]

assert len(table) == length, f"The table does not have the expected length: {length}"

ciphertext = [i // 2 for i in table]
print("Ciphertext: ", ciphertext)

plaintext = [chr(i ^ 0x37) for i in ciphertext]
print("Flag: ", ''.join(plaintext))
```

The plaintext is `find_it_Difficult_n0w?`. As we have another pool object which is `ph0wn{` (and that the official flag format for Ph0wn CTF is `ph0wn{xxxxx}`), we assume the beginning and trailing part will be concatenated to the main plaintext to create the full flag `ph0wn{find_it_Difficult_n0w?}`

There are several other ways to solve the challenge. For instance, a participant might spot the method name `_winningMessage` in Code Information.

	x86_64	armeabi-v7a	arm64-v8a
pool index for `ph0wn{`	5810	5901	5797
pool index for `ph0wn{you_w1n_darts_stage1}`	5811	5902	5798
`_winningMessage` address	0x003CE09C	0x00436A08	0x003C71A0

Android/MoneyMonger malware

The MoneyMonger family was discovered in May 2022 and consists in loan scams. The cybercriminals attract victims with easy money, collect personal data and uses the information to blackmail them later. Victims have reported being harrassed by phone calls, SMS to re-pay their loan even before the due date. Some even reported being harrassed without concluding any contract. The criminals go as far as threatening victims to leak sensitive information.

In December 2022, a Flutter version of MoneyMonger was discovered and detailed in (Ortega 2022). The researcher used static analysis to reverser malicious parts within the Dalvik layer, and fell back to dynamic analysis for the difficult parts, in particular parts implemented in Dart. With our learned knowledge, we are able to go further into the reversing of Dart.

We study malicious sample `e7fdecbac151325ad88054fb2287d7a5b48234f63d81fc35e2425b57ebb559e3`. It has only been compiled for AArch64, which makes the task more difficult with regards to the current status of tools.

The manifest lists an entry point `com.fastrupee.fastrupeepeefa.MainApplication` which inherits from `FlutterApplication` and initiates the initialization of the Flutter framework.

Platform Channels

Communication between the Dalvik layer and Flutter is perform through *Platform Channels*. Platform channels are implemented by Flutter and are the standard way to communication between a Flutter client application and its host (Android in our case). In particular, Flutter provides a class named `MethodChannel` to help *Dart* code call *Java or Kotlin* code.

MoneyMonger's `MainActivity.kp()` method defines the mapping between both worlds:

- `getGaid()` calls `getAdvertisingIdInfo().getId()`
- `setConfig()` adds an entry named `smsCount` to a shared preferences file
- `removeCrashLog()` removes appropriate entries from the shared preferences file
- `getCrashLog()` reads the `crash_message` and `crash_stacktrace` entries from the shared preferences file
- `aesEncrypt()` calls `b2.dp()` with appropriate arguments provided by the Dart code
- same for several other functions like `getLocation()`, `toHex()`, `fromHex()`, `aesDecrypt()`, `getAllDeviceInfo()`, `reportBranchEvent()`

```
    private final Object kp(MethodCall methodCall0, Result
        methodChannel$Result0) {
        SharedPreferences.Editor sharedPreferences$Editor0;
        if(l8.a6(methodCall0.method, "getGaid")) {
            try {
                return AdvertisingIdClient.getAdvertisingIdInfo(this).getId
                    ();
            }
            catch(Exception exception0) {
```

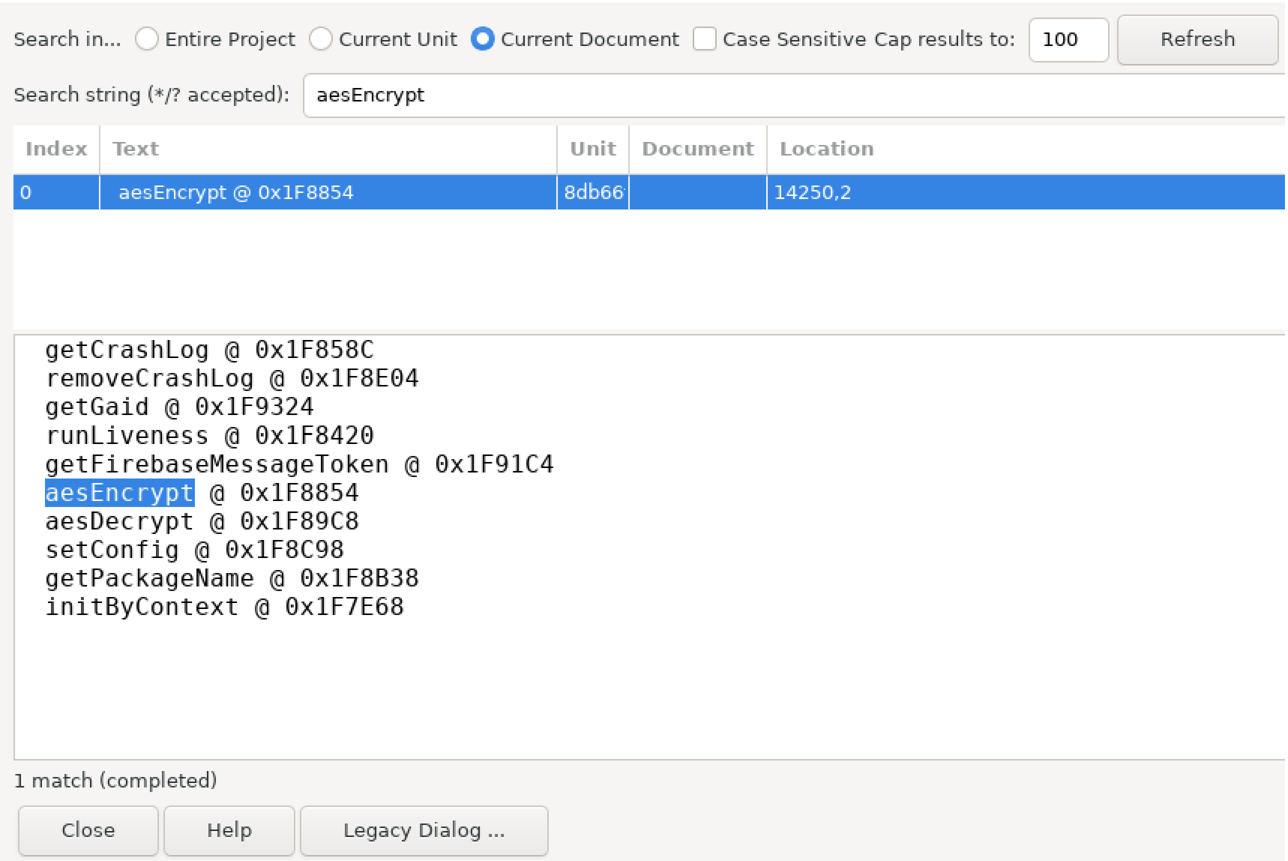
```
                exception0.printStackTrace();
                return null;
            }
        }

        int v = 0;
        if(l8.a6(methodCall0.method, "setConfig")) {
            String s = (String)methodCall0.argument("smsCount");
            if(s == null) {
                return null;
            }

            sharedPreferences$Editor0 = this.getSharedPreferences("data", 0)
                .edit().putString("smsCount", s);
            sharedPreferences$Editor0.commit();
            return null;
        }
```

Dart AOT Snapshot

In the Dart AOT snapshot, the Code Information displayed by JEB (or searching for strings in `libapp.so`) shows the application mostly consists in classes to handle the UI: `Widgets`, `UserPageConfig`, `WebViewController`, `StartupPageConfig`… Class `MyUtilsPlugin` references several methods mentioned by [11]: `aesDecrypt`, `setConfig`, `getAllDeviceInfo`.



The following methods are part of a class named MyUtilsPlugin

When the Flutter method `getGaid` is called, method channel glue between Flutter and Dalvik gives the information to the Java code, and this triggers the call of `getAdvertisingIdInfo()` on Java side.

We are able to locate the (Dart) implementation of `getGaid`, `setConfig`, `aesEncrypt`, `aesDecrypt` etc.

method name	address
aesDecrypt	0x1F89C8
aesEncrypt	0x1F8854
setConfig	0x1F8C98

setConfig

For example, we located `setConfig()` and recognize the typical ARM prologue:

```
com_fastrupee_fastrupeepeefaMyUtilsPlugin__setConfig proc

                  STP         X29, X30, [X15, #FFFFFFF0h]!     ; xref: sub_4196E8+1D0h (call) / 45CDDCh (call)
                  MOV         X29, X15
                  SUB         X15, X15, #18h
                  LDR         X16, [X26, #40h]
                  CMP         X15, X16
                  B.LS        loc_208DF8
```

Actually, this function is a wrapper function to communicate with the Dalvik layer. We set it fetches the string `setConfig` at index 10185 from the Object Pool. The access to the index is performed in 2 instructions: (1) compute 0x13000, then (2) add 0xE48. The resulting offset divided by 8 is 10185.

```
                  BL          sub_20EBC
                  ADD         X17, X27, #13h, LSL #12          ; setConfig
                  LDR         X17, [X17, #E48h]
                  STUR        X17, [X0, #1Fh]
                  LDUR        X1, [X29, #FFFFFFF0h]
                  STUR        X1, [X0, #27h]
                  STUR        X0, [X1, #5Fh]
                  STR         X0, [X15, #FFFFFFF8h]!
                  BL          ::___asyncThenWrapperHelper@4048458
```

The `setConfig` method is called by the code below which (1) gets the current SMS count and (2) provides this count as argument to the `setConfig` method. As a matter of fact, from the Dalvik glue, we saw that `setConfig` was taking one argument named `smsCount`.

```
BL          com_fastrupee_fastrupeepeefaPermissionConfig__get:getSmsCount
STR         X0, [X15, #FFFFFFF8h]!
BL          com_fastrupee_fastrupeepeefaMyUtilsPlugin__setConfig
ADD         X15, X15, #8
```

The function `getSmsCount` calls `getConfigValue`:

```
LDR         X4, [X4, #6D8h]
BL
    com_fastrupee_fastrupeepeefaLtConfig___com_fastrupee_fastrupeepeefagetConfigValue@
```

AES encryption/decryption

As for the Flutter `aesEncrypt` and `aesDecrypt` functions, we see the actual AES encryption/decryption is performed from the Dalvik layer, and that it takes two arguments: a password (secret key) and a buffer.

The password can be recovered by different methods:

1. Search for 16-byte strings in `libapp.so`. We notice `D4JcGjcw489iiEq1` (actually, `strings` does not output the string boundary correctly and adds a trailing 8, but keys are 16 bytes long for AES, so we can remove it)
2. Search for 16-byte strings in the Object Pool: the key is at index *4102*.
3. Use a Frida hook on the Java layer to display the password. The same hoom can also be used to print decrypted values.

```
r.parseInt(\"1022137011\");\n            if (Math.random() > x) {\n\n
              } else {\n\n            }\n        }\n{\n            ret = (
    int)(Double.MAX_VALUE * Math.random());\n        }\n{\n            int a
     = (int)(Integer.MAX_VALUE * Math.random());\n            int b = (int)(
    Integer.MAX_VALUE * Math.random());\n            ret  = a + b;\n
    }\n{\n            ret = (WindowManager) this.getApplicationContext().
    getSystemService(Context.WINDOW_SERVICE);\n        }\n{\n            ret
     = (WindowManager) this.getApplicationContext().getSystemService(Context
    .WINDOW_SERVICE);\n        }\n}\n@Override\n    protected void onCreate(
    Bundle savedInstanceState) {\n        super.onCreate(savedInstanceState)
    ;\ntry {com_fastrupee_fastrupeepeefaActivity40();} catch(Exception e) {}
     \n    }\n}"}}}
dp: password=D4JcGjcw489iiEq1
dp: plaintext={}
dp: password=D4JcGjcw489iiEq1
dp: plaintext={"AID":"419325ce0c5f52d2","UUID":"cd4842a0-b89b-11ed-8d26-87
    e38aff63fe","GAID":"f1db4ccc-af6a-42db-a428-a7f6155068cb","VERSION":"
    1.0.18","SE":"true"}
```

- The password is `D4JcGjcw489iiEq1`.
- The text above `dp: password=D4JcGjcw489iiEq1` is the *decrypted* configuration. It is dumped in `FlutterSharedPreferences.xml`
- The last plaintext is a JSON value containing device information. This plaintext is *encrypted* on request by the Dart code by the Dalvik layer. And indeed, if we look at the code of the routine which calls `aesEncrypt` in Dart, we notice it operates on JSONs:

```
STP         X1, X16, [X15, #FFFFFFF0h]!
LDR         X4, [X27, #4A8h]
```

```
BL          JsonCodec__encode
ADD         X15, X15, #10h
...
BL          com_fastrupee_fastrupeepeefaMyUtilsPlugin__aesEncrypt
```

With MoneyMonger, the malicious code is located on Dalvik's side, which makes it easier to reverse than we could have expected. Malware such as **FluHorse** implement the malicious parts in Dart and are more tricky to reverse. See (Apvrille 2023c).

State of the Art

Research on reversing Flutter consists in at most a handful of interesting articles and tools.

Doldrum and Darter dump Dart AOT snapshots. However, due to frequent format changes, these projects became outdated and could not keep up with the modifications, rendering them obsolete. For example, Doldrum works until Dart version 2.12. It throws an exception for any more recent version (here 2.19.1). The fix is far more complex than just adding the new snapshot hash [4].

```
    raise Exception('Unsupported Dart SDK, snapshot hash: ' + snapshotHash)
Exception: Unsupported Dart SDK, snapshot hash:
    adb4292f3ec25074ca70abcd2d5c7251
```

An alternative approach is employed by reFlutter, which instruments code to dynamically dump memory addresses of objects. By design, it only dump code it *visits*. Even so, it was unable to dump the address of any useful method of the w0rdle 3 challenge.

```
Library:'package:flutterdle/game.dart' Class: Flutterdle extends Object {
    // missing dump
}

// successful dump of address if Stats.fromJson in domain.dart
Library:'package:flutterdle/domain.dart' Class: Stats extends Object {
  Function 'toJson':. (Stats) => Map<String, dynamic> {
                Code Offset: _kDartIsolateSnapshotInstructions + 0
                    x0000000000109648
  }
}
```

Among the articles, (Lipke 2020) stands out as one of the first to address in detail Dart reverse engineering. [4] does an excellent job at outlining the core challenges and providing a viable solution with IDA Pro. However, the solution is complex: memory dump of the sample, use of several scripts, and IDA Pro plugins to automate function name recovery.

Other articles, such as (Loura 2020), focus on specific aspects, while (Software 2022), (Alexander 2023), ("Reverse Engineering a Flutter App by Recompiling Flutter Engine" 2021) and my own (Apvrille 2022a) explore practical cases and show how to use *reFlutter*.

[4] For Darter too: changing the `EXPECTED_VERSION` constant is in `darter/constants.py` only superficially fixes the error message.

One of the key *novelties* of this paper are the **disassembly of byte arrays in Dart**. This hasn't ever been covered. *It is particularly useful to* find secrets in CTF challenges, CrackMes or malicious code.* The other contributions are:

1. **Dart's calling convention specificities**. (Batteux 2022) had noticed it for Aarch64, but we see it is also true for Aarch32 and x86_64 to some extent.
2. **A comprehensive approach to reversing Dart or Flutter** with Radare or JEB. There are fragments of information in various articles, none provide specific details on the task. This paper details how to spot the beginning and end of functions when disassemblers fail to identify them, highlight the stack overflow instructions which are specifically inserted in Dart assembly, or tricks to search for XOR loops over x86_64, arm32 and arm64 platforms.

Conclusion and Future Work

The current support for assembly produced by Flutter applications is limited. Besides Flutter's intentional and default obfuscation, this is due to several concepts of Dart: the Object Pool with indirect access to constant strings, the special representation of Small Integers, the use of custom registers (object pool, current Dart VM thread, and even custom stack register for AAarch64) and an unusual calling convention where all parameters are pushed on the stack whatever their number.

With research from this paper, we able to tweak disassemblers manually to assist reverse engineering. We are able to reverse a basic Dart release program, solve a Flutter CTF challenge and dig in Flutter parts of a malware of the MoneyMonger family.

The next step consists in enhancing disassemblers. For instance, the following would particularly interesting:

- Recognize encoding of Small Integers and automatically annotate code with the corresponding integer value
- Link strings of the Object Pool to their use in assembly code, instead of manually computing the pool index and looking it up
- Organize Flutter methods in the classes they belong to. This would help recover the object oriented aspect of the source code.
- Detect Dart's calling convention, where arguments are pushed on the stack. On Aarch64, detect Dart's custom stack register. Understanding the calling convention will help produce better decompiled code.
- Automatically detect and annotate Dart's stack overflow check lines.

Appendix

Dart commands

Create a standalone Dart project: `dart create -t console projectname`

Dart program formats

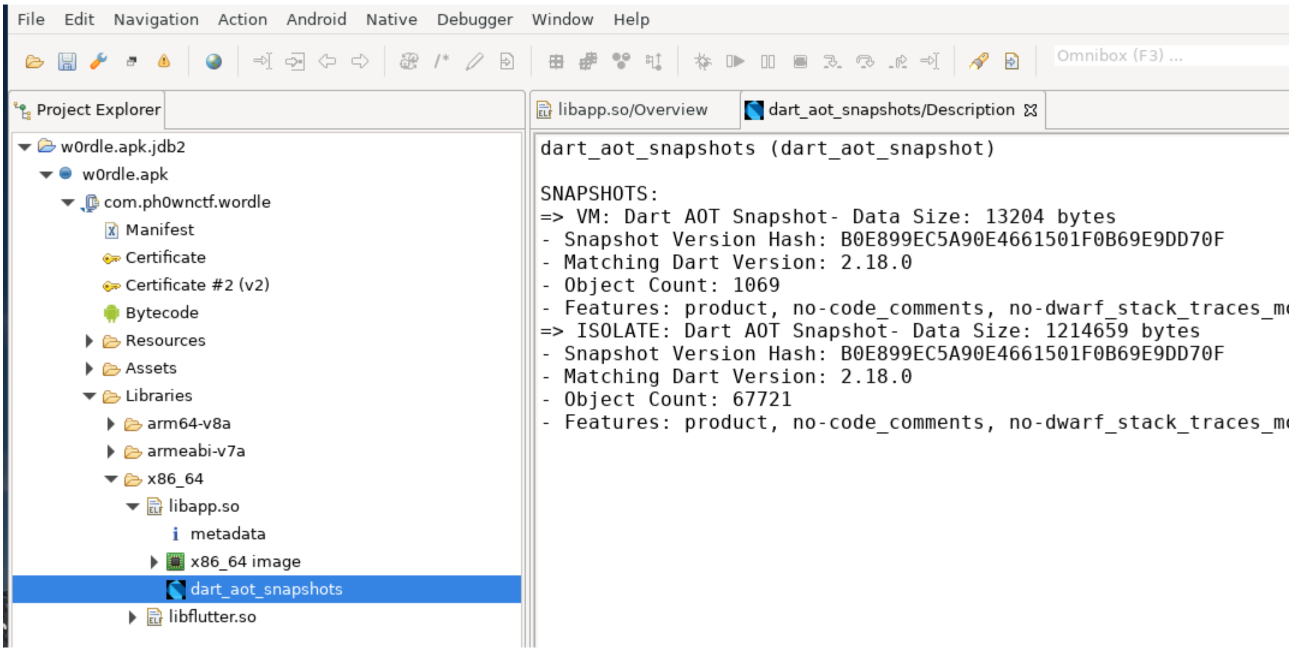
Dart formats	Command to create it	Run it…
Source code		`dart run hello.dart`
Self contained executable	`dart compile exe hello.dart`	`./hello.exe`
AOT snapshot	`dart compile aot-snapshot hello.dart`	`FLUTTER_DIR/flutter/bin/cache/dart-sdk/bin/dartaotruntime hello.aot`
JIT snapshot	`dart compile jit-snapshot hello.dart`	`dart run hello.jit`
Kernel snapshot	`dart compile kernel hello.dart`	`dart run hello.dill`

Dart formats	Portable	Contains a Dart Runtime	Description
Source code	Yes	No	`dart run sourcecode` uses Dart VM's JIT compiler to run the code
Self contained executable	No	Yes	
AOT snapshot	No	No	Code is *natively* compiled for the platform. Used for Flutter *release* builds. The command `dartaotruntime` contains the runtime.
JIT snapshot	No	No?	Usually slower than AOT snapshots.
Kernel snapshot	Yes	No	Portable representation of code. Used for Flutter *debug* builds.

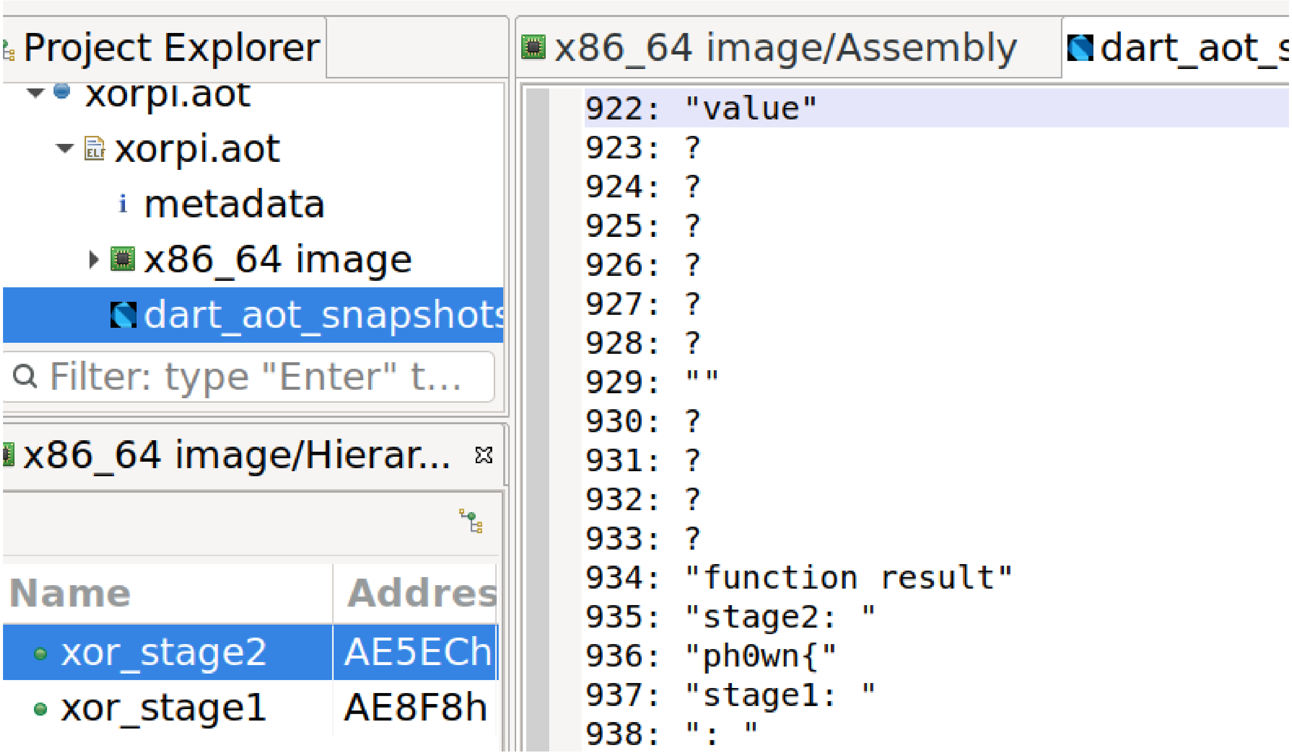
AOT snapshot format

Dart AOT snapshot formats have evolved through time. Therefore, one of the first steps is to read the *version*, which is present in the snapshot header. It can be read with JEB, my Python script (Apvrille 2022b) or using my ImHex pattern (Apvrille 2023a)

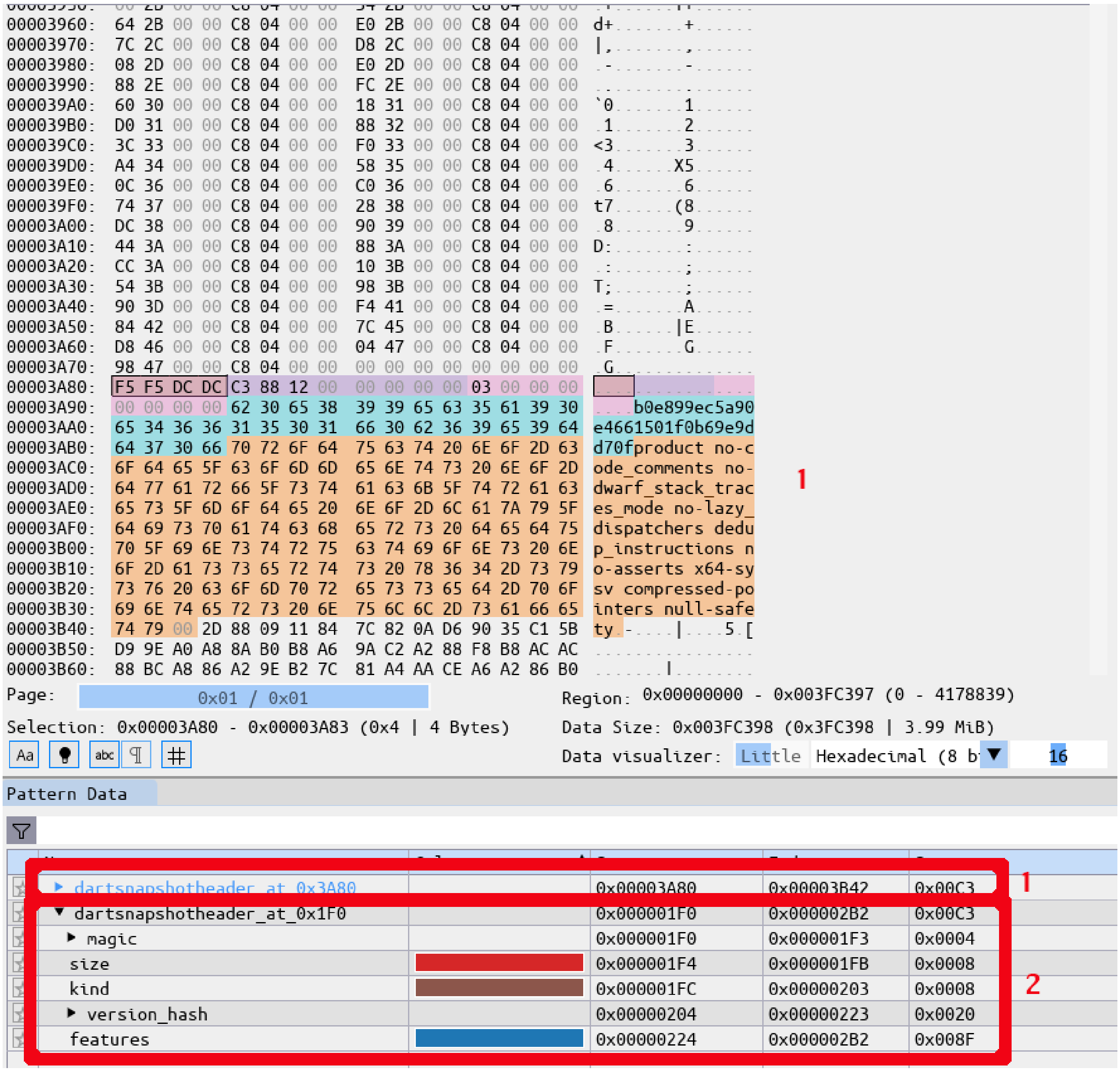
```
$ python3 flutter-header.py -i ./hello.aot
========== Flutter Snapshot Header Parser ============
Snapshot
    offset  = 512 (0x200)
    size    = 3519
    kind    = SnapshotKindEnum.kMessage
    version = 2.19.1
    features= product no-code_comments no-dwarf_stack_traces_mode no-lazy_dispatchers dedup_instructions no-asserts x64 linux no-compressed-pointers null-safety
-----------
Snapshot
    offset  = 34240 (0x85c0)
    size    = 98694
    kind    = SnapshotKindEnum.kMessage
    version = 2.19.1
    features= product no-code_comments no-dwarf_stack_traces_mode no-lazy_dispatchers dedup_instructions no-asserts x64 linux no-compressed-pointers null-safety
-----------
```



JEB reading a Dart AOT snapshot header



JEB shows the Object Pool of a Dart program and display readable strings for each index. There are a few bugs though, and this doesn't work all the time



ImHex showing 2 Dart AOT snapshots in libapp.so

Description	Value	Size	Comments
Magic number	`f5f5dcdc`	4 bytes	kMagicValue
Size		8 bytes	
Snapshot kind		8 bytes	`SnapshotKindEnum`
Version hash		32 bytes	

Features	Variable: Null terminated string

For example, this is a Dart AOT Snapshot at 0x140:

```
00000140  f5 f5 dc dc c2 0d 00 00  00 00 00 00 03 00 00 00  |................|
00000150  00 00 00 00 61 64 62 34  32 39 32 66 33 65 63 32  |....adb4292f3ec2|
```

- `f5 f5 dc dc c2`: Magic number
- size: 4 bytes (0d 00 00 00)
- kind: SnapshotKindEnum.kMessage (00 00 00 03)
- version_hash: `adb4292f3ec25074ca70abcd2d5c7251`
- features: null terminated string

After the Snapshot header, there is *Cluster Info*:

Description	Size	Comments
Base Object Count	ULEB128	
Object Count	ULEB128	
Cluster Count	ULEB128	
Code order length	ULEB128	

- ClassId: U32

Dart strings

Dart strings are created one of following ways:

1. Using `String`, e.g. `String flag = 'flag{congrats}'`
2. `String` are immutable, so if they need to be manipulated, use the `StringBuffer` class, and, if needed, convert to a `String` with `toString()`.

```
void main() {
     final buffer = StringBuffer('Pico le Croco');
     buffer.write(' has big teeth');
     print(buffer);
     print(buffer.toString());
}
```

3. A string is also a sequence of Unicode UTF-16 code units, which are represented as integers. So, they can also be created from list of integers, or types derived from integers (e.g Uint8List). Note that if UTF-8 conversion is needed, there are `encode()` and `decode()` methods from the `dart:convert` library.

```
// String to bytes
String foo = 'Hello world';
List<int> bytes = foo.codeUnits;
// Bytes to String
String bar = String.fromCharCodes(bytes);
```

Object Pool serialization

When you compile a Flutter app,

1. Each object is serialized
2. A reference to each Dart object is stored in the *Dart object pool*
3. The object pool itself is *serialized* too
4. The Dart code is then modified, and each access to an object is replaced with an indirect access via the pool index.

At runtime, to access objects:

1. Retrieve the serialized objects from the Dart AOT snapshot format
2. De-serialize all objects in Flutter
3. Whenever the code asks to access a given pool index, the object pool looks up for the corresponding object and returns it.

Register layout

Architecture	Register	Use
arm7eabi	r5	**Object Pool**
	r10	Pointer to running VM thread
	r11	Frame Pointer
	r13	Stack Pointer
	r14	Link Register
	r15	Program Counter
arm64	X15	**Custom Stack Pointer**

Architecture	Register	Use
	X26	Pointer to running VM thread
	X27	**Object Pool**
	X29	Frame Pointer
	X30	Link Register
x86_64	r10	Arguments descriptor register
	r12	Code register
	r14	Pointer to running VM thread
	r15	**Object Pool**

Where is it implemented?

	URL
ClassId enumeration	`sdk/runtime/vm/class_id.h`
ObjectPool class	runtime/vm/object.h
Register enumeration	runtime/vm/constants_arm.h, runtime/vm/constants_arm64.h, runtime/vm/constants_x64.h
Snapshot class	runtime/vm/snapshot.h
Snapshot serialization	`sdk/runtime/vm/app_snapshot.cc`
Serialization of integers	runtime/vm/app_snapshot.cc
Class Smi	`runtime/vm/object.h`
Cluster Info serialization	runtime/vm/app_snapshot.cc
Read/Write Uint	`runtime/vm/kernel_binary.h`
Read/Write LEB128	`runtime/vm/datastream.h` L173

Code snippets for various platforms

Storing *representations* of SMIs

On Aarch32,

```
MOV         R12, #A2h
STR         R12, [R2]
```

On Aarch64,

```
; Assign value A2 to X17
MOVZ        X17, #A2h
; Store Unsigned Register
; We are writing W17 (least significant 32 bits of X17) at
; the address of X0 + Fh
STUR        W17, [X0, #Fh]
```

On x86_64,

```
mov r11d, A2h
mov dword ptr ds:[rax+Fh], r11d
```

Storing a SMI in a byte array

We need (1) to double the SMI to represent it, and (2) store the representation.

On Aarch32, with:

- the value to store is R3
- the byte array is R2
- the counter is R6

```
LSL         R0, R6, #1
...
MOV         R8, R0
...
; R0 = R1
EOR         R0, R1, #0
MOV         R9, R0
; Logical Shift Left: multiply result by 2
LSL         R0, R3, #1
; Check negative value
CMP         R3, R0, ASR #1
CMPEQ       R9, R0, ASR #31
BEQ         not_negative2
; we should not get here
BL          sub_481430
STR         R3, [R0, #7]
STR         R9, [R0, #Bh]
not_negative2:
MOV         R1, R2
```

```
; compute the offset in the array: R9 = R2 + R6*4
; because R8 = 2*R6
ADD         R9, R1, R8, LSL #1
ADD         R9, R9, #Bh
; store the representation of the SMI in the array
STR         R0, [R9]
; R0 is a SMI representation, so R0 & 1 = 0
TST         R0, #1                          ; tests R0 & 1
; Branch if previous test was 0
BEQ         increment
```

On Aarch64 with:

- the value to store is X5
- the byte array is X1
- the counter is X4

```
; Signed BitField Insert Zeroes
; copies the 31 lower bits of X5 at position 1 in X0
; this doubles X5!
SBFIZ       X0, X5, #1, #1Fh
; Shift right X0 and compare with X5
CMP         X5, X0, ASR #1
; branch if equal
B.EQ        equal_case
; we should not get here
BL          sub_402188
STUR        X5, [X0, #7]
equal_case:
MOV         X1, X2
; offset = array+(X4*4)
ADD         X25, X1, X4, LSL #2
ADD         X25, X25, #Fh
; store the XORed value (W0) at the address of X25
STR         W0, [X25]
; bit 0 of W0 will be 0 because it's an SMI, so we will branch to increment.
TBZ         W0, #0, increment
```

On x86_66 with:

- the value to store is eax
- the byte array is rdx
- the counter is rdi

```
; Load Effective Address works on addresses (no access to memory)
; So, rax = 2 * rdx
lea         rax, qword ptr ds:[rdx+rdx]
mov         rbx, rdx
; Right shift 31 bits: we only keep the sign bit
sar         rbx, 1Eh
add         rbx, 1
; at this point if rdx was negative, then rbx = 2
; if rdx was positive, then rbx = 1
cmp         rbx, 2
```

```
; branch if below 2
jb          is_below
; we should not get here
call        sub_402B00
mov         qword ptr ds:[rax+7], rdx
is_below:
; compute the offset where to store the value
lea         r13, qword ptr ds:[rdx+4*rdi+Fh]
; store value
mov         dword ptr ds:[r13], eax
```

Getting SMI from a byte array

This consists in

1. Reading the value from the byte array
2. Dividing it by 2 for a SMI

For Aarch32:

```
; compute offset address R12 = R2 + R6 * 4
; because R8 = R6 * 2 previously
ADD         R12, R2, R8, LSL #1
; load byte from that position
LDR         R0, [R12, #Bh]
; Arithmetic Shift Right: store sign bit of R0 is R1
ASR         R1, R0, #31
; perform division by 2
ASRS        R9, R0, #1
```

For Aarch64:

```
; X2 holds the array
; X4 is the counter
; we compute offset address X16 = X2 + X4*4
ADD         X16, X2, X4, LSL #2
; Load Unsigned Register W0 = &(X16 + Fh)
LDUR        W0, [X16, #Fh]
;
ADD         X0, X0, X28, LSL #32
; Sign-Extended BitField Extract
; Extract bits 1 to 31 (with sign extension) and copies to X1
; This actually performs a division by 2
SBFX        X1, X0, #1, #31
```

For x86_64:

```
; rcx holds the array
; rdi is the counter
; we computer offset address rcx+4*rdi+Fh
; the content of this address will be in eax
mov         eax, dword ptr ds:[rcx+4*rdi+Fh]
; don't care
```

```
add       rax, qword ptr ds:[r14+48h]
; the content is now in rdx
movsxd    rdx, eax
; shift arithmetic right
; this divides by 2
sar       rdx, 1
```

Prologue snippet + stack check

For Aarch32:

```
; push frame pointer (r11) and link register on the stack
PUSH      {R11, LR}
; move frame pointer to the bottom of the stack
ADD       R11, SP, #0
SUB       SP, SP, #8
MOV       R0, #2Ch
; check stack overflow
; r10 holds the current VM thread pointer
LDR       R12, [R10, #1Ch]
CMP       SP, R12
BLLS      sub_32FCF4
```

For Aarch64:

```
STP       X29, X30, [X15, #FFFFFFF0h]!
MOV       X29, X15
SUB       X15, X15, #10h
LDR       X16, [X26, #38h]
CMP       X15, X16
B.LS      loc_3D75DC
```

For x86_64:

```
; push base pointer on the stack
push rbp
; the new value for the base pointer is the stack pointer
mov rbp, rsp
; allocate 16 bytes
sub rsp, 10h
; r14 holds the current Dart VM thread pointer
cmp rsp, qword [r14 + 0x38]
; if stack pointer is <= [r14 + 0x38 ]: jump stack overflow error
jbe 0x9e850
```

XOR

For Aarch32: `EOR R3, R9, #37h`

For Aarch64:

```
MOVZ      X16, #37h
```

```
EOR        X4, X3, X16
```

For x86_64: `xor rdx, 0x37`

Load a Pool Object

- For Aarch32: `LDR R1, [R5, #433h]`
- For Aarch64: `LDR X16, [X27, #433h]`
- For x86_64: `mov rbx, qword ptr ds:[r15+433h]`

String concatenation

For Aarch32:

```
ldr lr, [r5, 0xe9f] ; "stage2: "
ldr sb, [r5, 0xea3] ; "ph0wn{"
stm sp, {sb, lr}    ; push them on the stack
bl fcn.concat       ; concatenate strings
```

For Aarch64:

```
LDR        X16, [X27, #1C90h]               ; "stage2: "
LDR        X30, [X27, #1C98h]               ; ph0wn{
STP        X30, X16, [X15]                  ; we push them on the stack
BL         _StringBase.+                    ; concatenate both strings: '
    stage2: ph0wn{'
```

For x86_64:

```
mov r11, qword [r15 + 0x1d3f]
push r11
mov r11, qword [r15 + 0x1d47]
push r11
call fcn.string_concat
```

Useful commands

Unix

Task	How
Get the base address of snapshots	`objdump -T libapp.so`

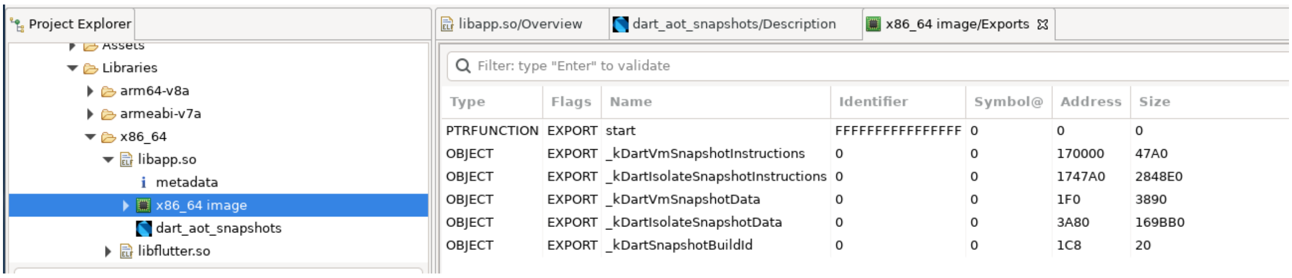
Task	How
List packages from a reFlutter dump	`grep -Eo "package:[a-z_A-Z]*/"dump.dart | grep -v "flutter/src"| sort | uniq`
Searching for strings constants in a Flutter app or Dart program	`strings` or `bgrep -t hex 'deadbeef'file` bgrep or `binwalk -R '\xde\xad\xbe\xef'file`

Radare 2

Task	Example
Search for instructions	`/ad mov~0xa0`
Modify instruction delimiter for search	`e asm.cmt.token=#`
Search for opcodes	`/x 379021e2`
Get address of the beginning of a function	`afi`
Define a function	`af`
Show only the instructions, no opcode, no arrows	`pif`

JEB

Task	How
Get Dart version	Go to Dart AOT snapshot > Description
Get the base address of snapshots	Platform image > Exports
Get address of Flutter functions	Search for the function name in "Code Information". Add base relocation address to the virtual address
Search for initialization of a byte array	Search for the instruction that moves double of the byte value. Wild card * is allowed in the search
View opcodes	Edit > Rendering Options > Show bytes count (6)



JEB displays the addresses of snapshots

MoneyMonger Frida hook

```
Java.perform(function(){
    var aes = Java.use("mpdi.v4.b2");
    // aesDecrypt
    aes.b8.implementation = function(password, ciphertext) {
    console.log("b8: password="+password);
    var decrypted = this.b8(password, ciphertext);
    console.log("decrypted="+decrypted);
    return decrypted;
    }
    // aesEncrypt
    aes.dp.implementation = function(password, plaintext) {
    console.log("dp: password="+password);
    console.log("dp: plaintext="+plaintext);

    return this.dp(password, plaintext);
    }

});
```

References

Alexander, Felix. 2023. “Flutter Hackers: Uncovering the Dev’s Myopia.” https://infosecwriteups.com/flutter-hackers-uncovering-the-devs-myopia-part-2-598a44942b5e.

Apvrille, Axelle. 2022a. “Reversing an Android Sample Which Uses Flutter.” https://cryptax.medium.com/reversing-an-android-sample-which-uses-flutter-23c3ff04b847.

———. 2022b. “Source Code to Find Flutter Snapshots and Read Their Header.” https://github.com/cryptax/misc-code/blob/master/flutter/flutter-header.py.

———. 2023a. “Dart AOT Snapshot ImHex Pattern.” https://github.com/cryptax/ImHex-Patterns/blob/master/patterns/dartaot.hexpat.

———. 2023b. “Dart’s Custom Calling Convention.” https://cryptax.medium.com/darts-custom-calling-convention-8aa96647dcc6.

———. 2023c. “Fortinet Reverses Flutter-Based Android Malware "Fluhorse".” https://www.fortinet.com/blog/threat-research/fortinet-reverses-flutter-based-android-malware-fluhorse.

———. 2023d. "Reversing Flutter Apps: Dart's Small Integers." https://cryptax.medium.com/reversing-flutter-apps-darts-small-integers-b922d7fae7d9.

———. 2023e. "W0rdle 3 Challenge." https://github.com/ph0wn/writeups/blob/master/2022/reverse/w0rdle/w0rdle.apk.

Batteux, Boris. 2022. "The Current State & Future of Reversing Flutter Apps." https://www.guardsquare.com/blog/current-state-and-future-of-reversing-flutter-apps.

"Dart." n.d. https://dart.dev.

"Flutter." n.d. https://flutter.dev.

Lipke, Andre. 2020. "Reverse Engineering Flutter Apps." https://blog.tst.sh/reverse-engineering-flutter-apps-part-1/.

Loura, Ricardo. 2020. "Reverse Engineering Flutter for Android." https://rloura.wordpress.com/2020/12/04/reversing-flutter-for-android-wip/.

Ortega, Fernando. 2022. "MoneyMonger: Predatory Loan Scam Campaigns Move to Flutter." https://www.zimperium.com/blog/moneymonger-predatory-loan-scam-campaigns-move-to-flutter/.

"Reverse Engineering a Flutter App by Recompiling Flutter Engine." 2021. https://tinyhack.com/2021/03/07/reversing-a-flutter-app-by-recompiling-flutter-engine/.

Software, PNF. 2022. "Dart AOT Snapshot Helper Plugin to Better Analyze Flutter-Based Apps." https://www.pnfsoftware.com/blog/dart-aot-snapshot-helper-plugin-to-better-analyze-flutter-based-apps/.