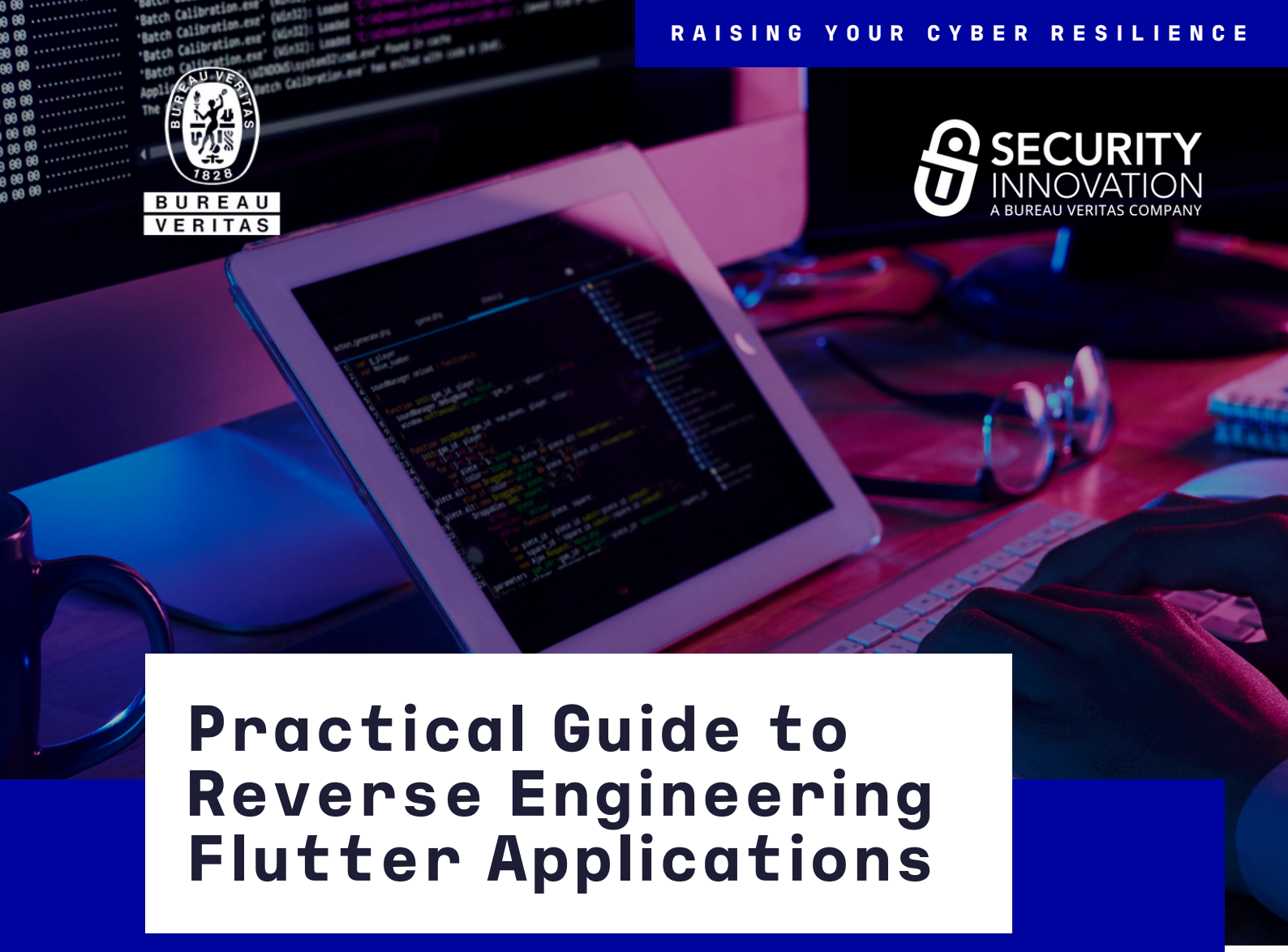


Practical Guide to Reverse Engineering Flutter Applications

By: Neel Patel (Security Engineer I @SecurityInnovation)
Asif Iqbal Gazi (Security Engineer II @SecurityInnovation)

Abstract

Flutter's popularity as a cross-platform framework has made it a go-to choice for building mobile applications with a single codebase, simplifying development across Android and iOS. However, Flutter's architecture and compilation process introduce unique challenges for reverse engineering, especially compared to native Android and iOS apps.

This whitepaper explores these differences and provides a step-by-step guide to reverse engineering Flutter applications—from understanding the app structure and decompiling code to analyzing Dart artifacts. Using an open-source Flutter app called **Minesweeper**, this guide demonstrates tools and techniques for reverse engineering Flutter Application. Whether you are a security researcher or a curious developer, this guide offers practical insights into the process.

Overview of Reverse Engineering Flutter Apps

Disassembling Flutter Binaries with Ghidra

For iOS applications, the core logic is typically found in the binary located in the
located in the <app_name>.app directory. However, in Flutter apps the logic resides in <app_name>.app > Frameworks > App.framework > App.

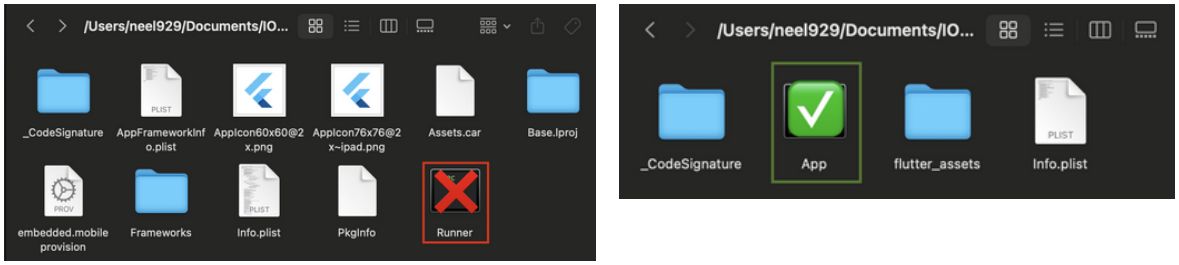


Figure 1: Screenshot of Minesweeper App Extraction/ Unzip

1. Extract the binary and load it into Ghidra.
2. Observe that, the exported symbols in the symbol tree include only:
   _kDartIsolateSnapshotData
   _kDartIsolateSnapshotInstructions
   _kDartVmSnapshotData
   _kDartVmSnapshotInstructions

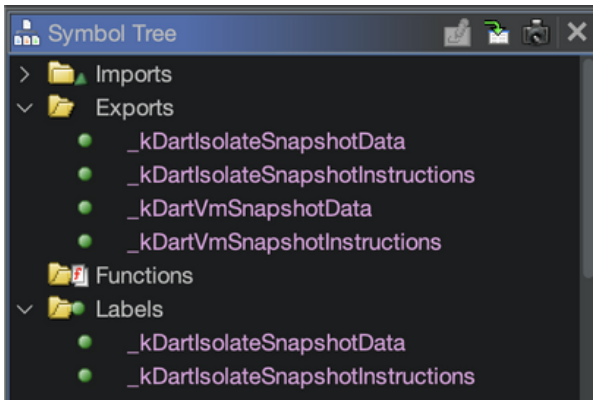


Figure 2: Screenshot of Symbol Tree in Ghidra

These symbols relate to the Dart runtime and its snapshot files, which manage compiled code and data.

Decoding Dart Runtime and Snapshots

Understanding the following components is critical for effective analysis:

Dart VM and Snapshots: Flutter applications are powered by the Dart Virtual Machine (VM), which enables the execution of Dart code, managing everything from memory to runtime operations. In production builds, Dart code is compiled Ahead-Of-Time (AOT) into native code, serialized into snapshot files that the Dart VM loads at runtime. Snapshots are serialized representations of compiled Dart code and data, allowing efficient runtime initialization.

- *VM Snapshots:* These contain shared code and data required by all isolates, like core libraries and commonly used functions. VM snapshots load at app startup, providing foundational instructions and data.
- *Isolate Snapshots:* Each isolate, an independent Dart execution context, has its own isolate snapshot with isolate-specific code and data. This allows multiple isolates to run concurrently without shared memory.

Isolates: In Dart and Flutter, concurrency is managed through isolates rather than traditional threads. Each isolate has its own memory space and runs independently, avoiding the risks of shared memory. The Dart VM manages the lifecycle of isolates, including creating, communicating, and terminating them. This architecture enhances stability but adds complexity to reverse engineering, as each isolate operates independently, making shared application state harder to track.

Dart Pool: The Dart Pool (or **Object Pool**) is an internal component of the Dart VM that manages frequently used objects and data structures. Acting as a cache, the Dart Pool reduces redundant memory allocations and optimizes performance. It manages reusable resources like pointers, precompiled instructions, and certain serialized data elements frequently referenced across the application.
The Dart Pool is closely related to Dart VM snapshots and isolates because it holds reusable resources that isolates can access. When a new isolate is created, it doesn't need to load resources from scratch; instead, it can reference objects in the Dart Pool, improving memory use and isolate startup times. This design is especially beneficial in Flutter, where applications often use multiple isolates (e.g., the main UI isolate and background isolates for async tasks).

With the background information provided, the 5 exported symbols have the following significance:

_kDartIsolateSnapshotData represents the initial state of the Dart heap and includes isolate-specific information.
_kDartIsolateSnapshotInstructions contains the Ahead-of-Time (AOT) code that is executed by the Dart isolate.
_kDartVmSnapshotData represents the initial state of the Dart heap shared between isolates. This helps launch Dart isolates faster but doesn't contain any isolate-specific information.
_kDartVmSnapshotInstructions contains AOT instructions for common routines shared between all Dart isolates in the VM. This snapshot is typically extremely small and mostly contains stubs.
_kDartIsolateSnapshotInstructions contains the application code that is executed by Dart Runtime.

We will not delve further into the technical details of the Dart Virtual Machine (DartVM) or Flutter architecture in this guide. However, we have provided references and external resources at the end of the whitepaper for readers interested in exploring these topics in more depth.

Metadata Extraction

Patching and Preparing the Binary

With information obtained from the previous section, if we want to continue our static analysis journey using Ghidra, we need symbolic information like Function Name, and Class Name. To do that, the app needs to be patched manually by modifying the source code of DartVM, compiling it, and replacing the runtime binary in the app. However, this is a tedious process. We can utilize an open-source framework called reFlutter to automate this process.

Using reFlutter framework:

1. Patch the app binary following **reFlutter's GitHub instructions.**
2. Extract the patched application.
3. Update the info.plist file to support document browsing.
4. Compress the Payload folder, rename it to .ipa, and install the app on a device.
5. Retrieve the dumped metadata file after running the app.

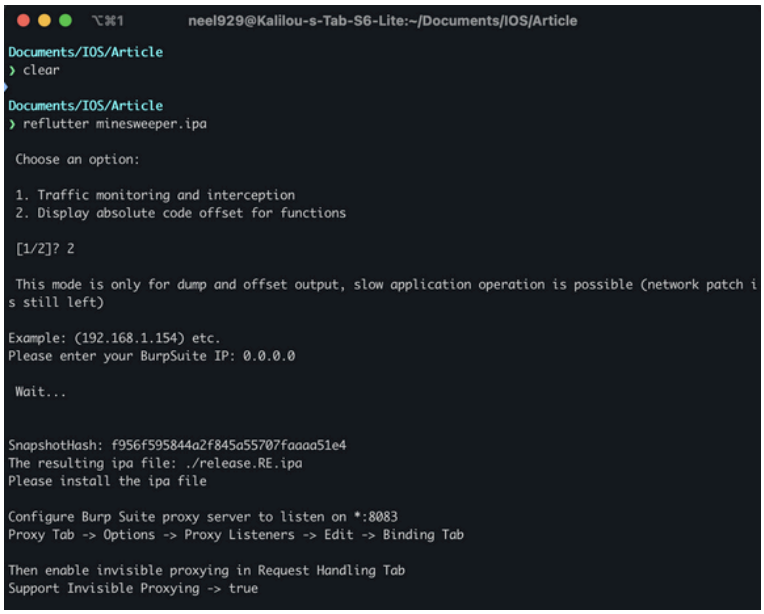


Figure 3: Screenshot of Patching App Binary using reFlutter

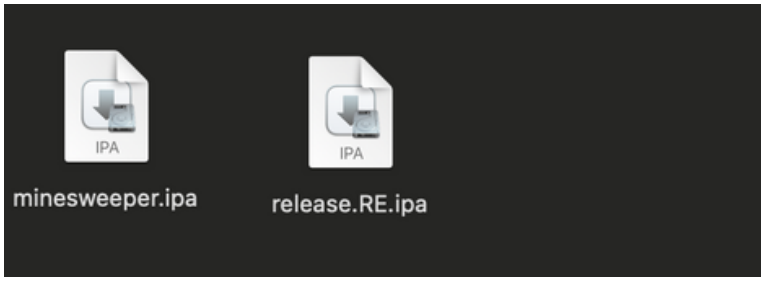


Figure 4: Screenshot of Patched App Binary release.RE.ipa)

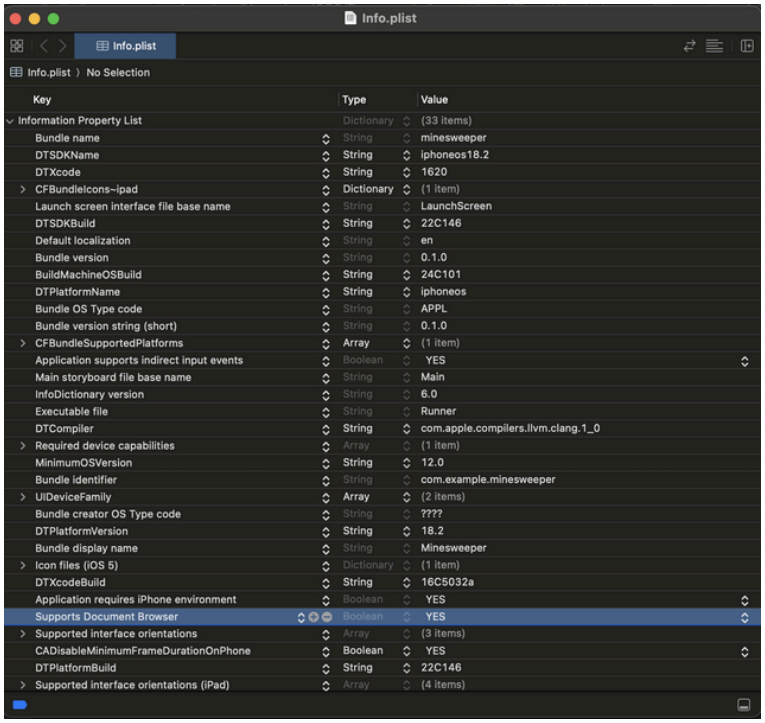


Figure 5: Screenshot of modified info.plist

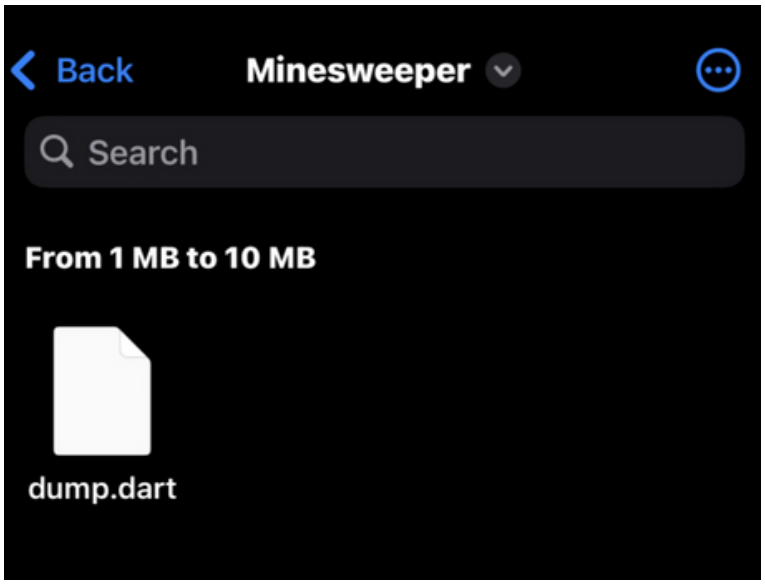


Figure 6: Screenshot of metadata dump after running the patched application for a while

Resolving Symbols in Ghidra using Dumped Metadata File

We can now resolve the symbols using the dump.dart that we recovered previously.

1. Load the dumped metadata (dump.dart) into Ghidra.
2. Use the Ghidra script shared below to resolve function names and populate the symbol tree.

Head over to Window > Script Manager and Create a new script based on Java and paste the following code:

```
// Java Script to resolve symbols in Flutter App using Dart metadata file.
// @Neel Patel

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.HashSet;
import java.util.Set;
import ghidra.app.script.GhidraScript;
import ghidra.app.cmd.disassemble.DisassembleCommand;
import ghidra.app.cmd.function.CreateFunctionCmd;
import ghidra.program.model.address.Address;
import ghidra.program.model.symbol.SourceType;

public class ResolveFunctions extends GhidraScript {

  /**
   * Creates a function at a given address with the specified name.
   *
   * @param offset The offset where the function is located.
   * @param functionName The name to assign to the function.
   */
  private void createFunction(long offset, String functionName) {
      // Convert offset to Address object
      Address functionAddress = toAddr(offset);
      println("Creating function: " + functionName);

      // Disassemble the function at the given address
      DisassembleCommand disassembleCmd = new
DisassembleCommand(functionAddress, null, true);
      if (disassembleCmd.applyTo(currentProgram)) {
          println("Disassembly successful at address: " + functionAddress);
      } else {
          println("Disassembly failed at address: " + functionAddress);
          return;
      }

      // Check if a function already exists at the address
      if (getFunctionAt(functionAddress) != null) {
          println("A function already exists at address " + functionAddress);
          return;
      }
```

```
        // Create the function at the specified address
        CreateFunctionCmd createFunctionCmd = new CreateFunctionCmd(functionName,
functionAddress, null, SourceType.USER_DEFINED);
        if (createFunctionCmd.applyTo(currentProgram)) {
            println("Function created successfully at address: " + functionAddress);
        } else {
            println("Failed to create function at address: " + functionAddress);
        }
    }

    @Override
    public void run() {
        // Path to the Dart metadata file
        String metadataFilePath = "/Path/to/dump.dart";

        // Hex value to be added to the offset
        String dartSnapshotInstructionHexOffset = "0x0000e900";

        try (BufferedReader reader = new BufferedReader(new FileReader(metadataFilePath)))
{
// Set to store unique JSON objects
            Set<String> uniqueJsonObjects = new HashSet<>();
            StringBuilder jsonContent = new StringBuilder();
            String line;

// Read all lines from the file and accumulate them into jsonContent
            while ((line = reader.readLine()) != null) {
jsonContent.append(line);
            }

// Split the JSON content into separate JSON objects
            String[] jsonObjects = jsonContent.toString().split("\\}\\{");

// Print the table header for debugging
            printf("%-20s | %-20s | %-20s | %-20s%n", "Method Name", "Offset", "Library URL",
"Class Name");
            printf("--------------------------------------------------------------------------------");

// Iterate over each JSON object
            for (String jsonObject : jsonObjects) {
// Clean up JSON object by removing leading '{' and trailing '}'
jsonObject = jsonObject.replaceFirst("^\\{", "").replaceAll("\\}$", "");

// Only process the JSON object if it's unique
if (uniqueJsonObjects.add(jsonObject)) {
// Split the JSON object into key-value pairs
String[] keyValuePairs = jsonObject.split(",");

String methodName = null, offsetHex = null, libraryUrl = null, className = null;
```

```
 // Extract values from key-value pairs
for (String pair : keyValuePairs) {
                String[] keyValue = pair.split(":");
                String key = keyValue[0].replaceAll("\"", "").trim();
                String value = keyValue[1].replaceAll("\"", "").trim();

                // Assign the extracted values to respective variables
                switch (key) {
                    case "method_name":
                        methodName = value;
                        break;
                    case "offset":
                        // Convert offset to hexadecimal and add the snapshot instruction offset
                        int originalOffset = Integer.parseInt(value.substring(2), 16);
                        int updatedOffset = originalOffset +
Integer.parseInt(dartSnapshotInstructionHexOffset.substring(2), 16);
                        offsetHex = "0x" + String.format("%08x", updatedOffset);
                        break;
                    case "library_url":
                        libraryUrl = value;
                        break;
                    case "class_name":
                        className = value;
                        break;
                }
}

// Print details for debugging or processing
// Uncomment the line below if you want to print the details to the console
// printf("%-20s | %-20s | %-20s | %-20s%n", methodName, offsetHex, libraryUrl,
className);

// Create the function using the parsed offset and method name
if (methodName != null && offsetHex != null) {
                createFunction(Long.parseLong(offsetHex.substring(2), 16), methodName);
}
}
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
  }
}
```

In the above script, replace the filePath with the Path of dump.dart file and also change the offset address of ***_kDartIsolateSnapshotInstructions*** which can be retrieved as shown below:

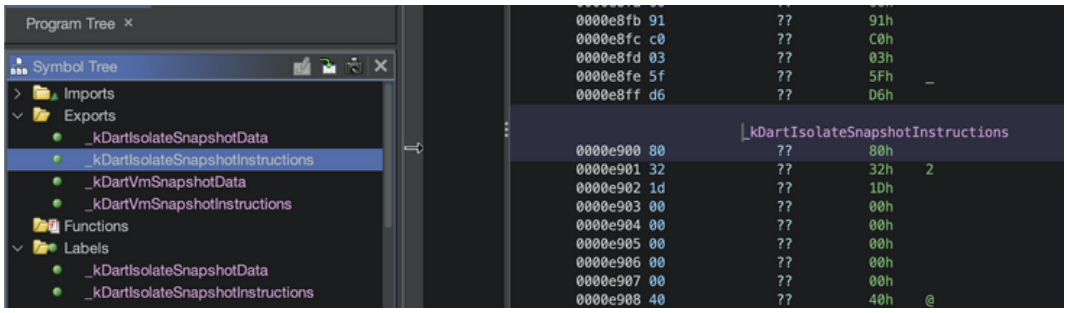


Figure 7: Screenshot of retrieving offset address in Ghidra using dump.dart

After making required changes and running the script we can see that Symbol Tree is populated with the original function names:

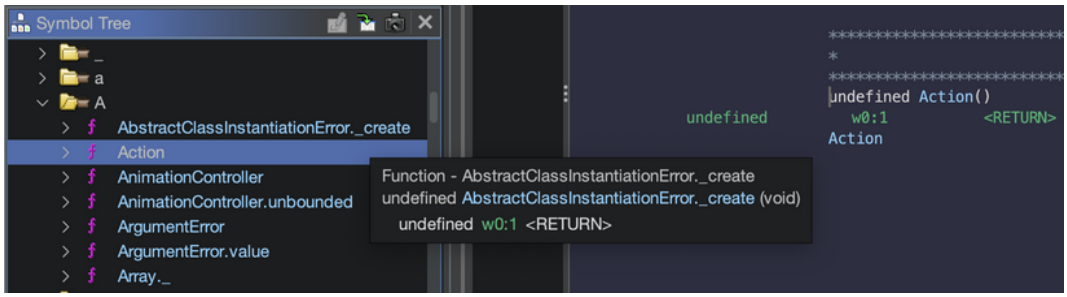


Figure 8: Screenshot of populated symbol tree in Ghidra using dump.dart and Ghidra Script

> ***Note:*** The App Binary until this point can be patched. The changes made to the binary after this point will result in an unpatchable binary. Changes made beyond this point are only to make analysis easier.

Resolving Code and Data References

After the previous steps, observe that there are no references between the code and data in the disassembly, as shown in the figure below. (Searching for string “T I M E” from the source code):

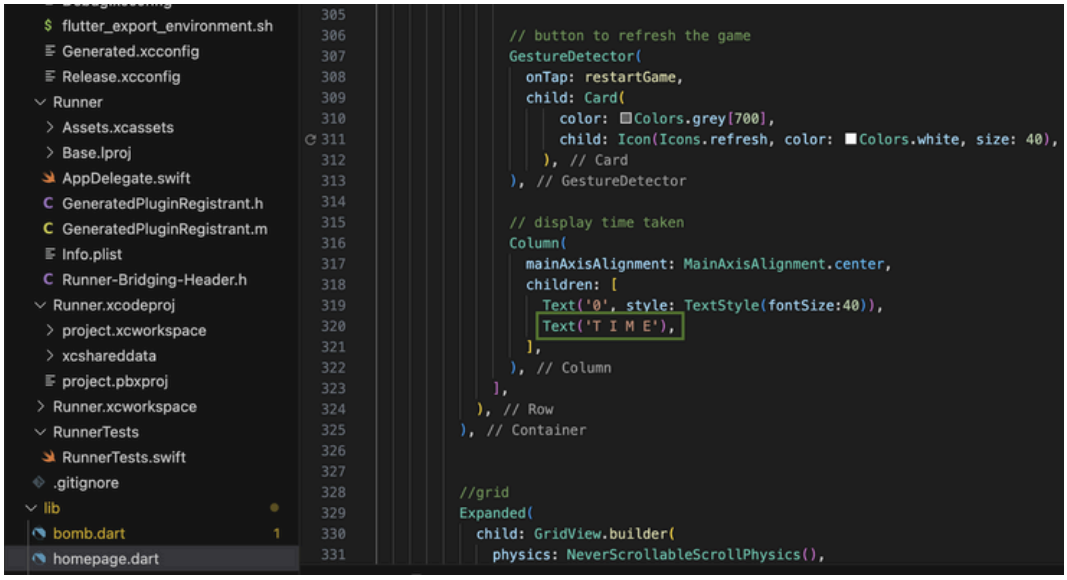


Figure 9: Screenshot of the string “T I M E” in the source code

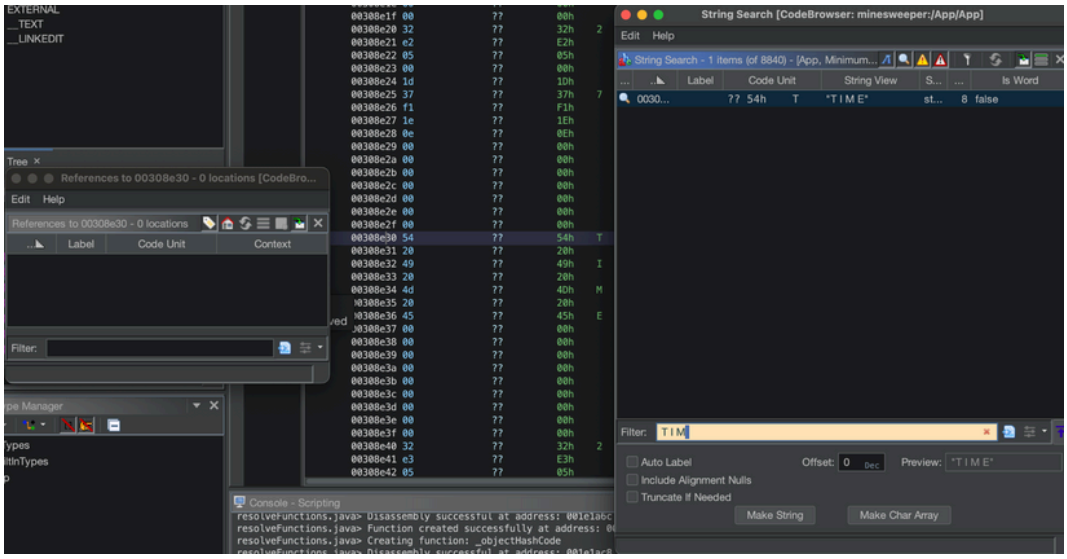


Figure 10: Screenshot of searching the string “T I M E” in Ghidra

Enhancing Dart Code Analysis

The challenge discussed above arise due to:

- **Object Pool Indirection** - as discussed in section 22.

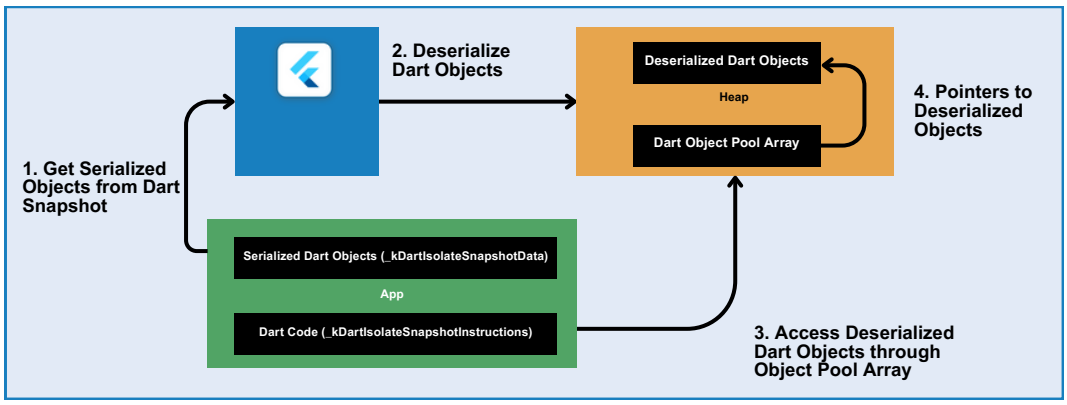


Figure 11: Screenshot of Dart Serialization and Deserialization

- **Non-standard stack pointer usage:** Dart VM uses register `X15` as a stack pointer instead of the standard SP

The Dart object pool is a big array with pointers to the Dart objects. During runtime, the Dart object pool contains pointers to the deserialized dart objects like string and integers; these objects are accessed using the `X27` register. Register `X27` points to the Dart Object Pool, and the deserialized Dart objects are accessed through this pointer. As shown in the disassembly below, the object is loaded into the `X1` register.

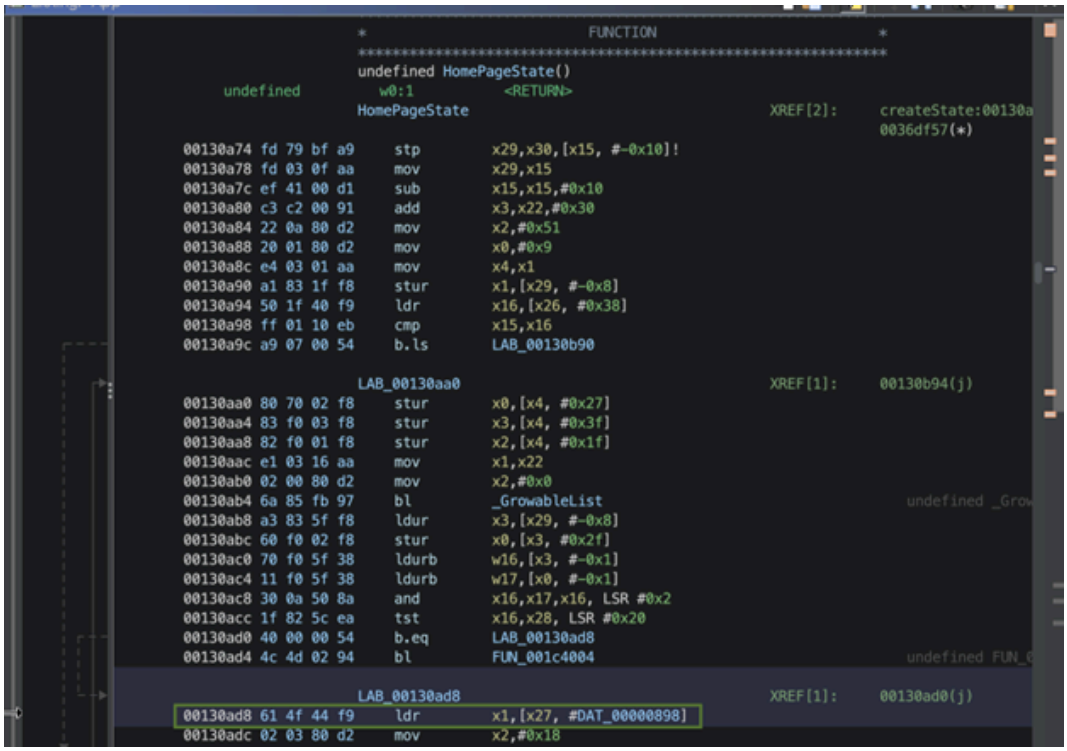


Figure 12: Screenshot of Non-Standard Stack pointer in use

Also, Dart VM uses X15 as the Stack register instead of the SP register. During the disassembly, decompilers deduce parameters based on the SP register. This is the reason why, in the decompiled code, there are no local variables detected and functions don't have any parameters.

In ARM64, registers X0-X7 are used for passing function parameters. If more than 8 parameters are used, the subsequent values are stored on the stack. However in Dart, function parameters are passed only on the stack. So reverse engineering tools like Ghidra often incorrectly assume the number of function parameters used when performing decompilation.

Steps to address these challenges:

- **Add cross-references** to deserialized objects.
- **Replace X15 with SP** for stack pointer recognition.
- **Adjust function signatures** to properly account for parameters passed on the stack.

Mapping Dart Objects

Establishing cross-references between the Dart code and the deserialized objects in the heap is a challenging task, as the Dart objects are stored in the runtime heap. To retrieve these objects in Ghidra, the following steps must be performed:

1. The Dart object pool and other Dart objects must be dumped from the application heap at runtime.
2. This memory dump needs to be imported to Ghidra.
3. The data types of the Dart object pool and other Dart objects must be defined in Ghidra.
4. References to the X27 register must be replaced with the static memory location of the Dart object pool array in the memory dump.

DartVM Registers

Registers used by Dart VM can be found here: https://github.com/dart-lang/sdk/blob/main/runtime/vm/constants_arm64.h

```
enum Register {
  R0 = 0,
  R1 = 1,
  R2 = 2,
  R3 = 3,
  R4 = 4,
  R5 = 5,
  R6 = 6,
  R7 = 7,
  R8 = 8,
  R9 = 9,
  R10 = 10,
  R11 = 11,
  R12 = 12,
  R13 = 13,
  R14 = 14,
  R15 = 15,  // SP in Dart code.
  R16 = 16,  // IP0 aka TMP
  R17 = 17,  // IP1 aka TMP2
  R18 = 18,  // reserved on iOS, shadow call stack on Fuchsia, TEB on Windows.
  R19 = 19,
  R20 = 20,
  R21 = 21,  // DISPATCH_TABLE_REG (AOT only)
  R22 = 22,  // NULL_REG
  R23 = 23,
  R24 = 24,  // CODE_REG
  R25 = 25,
  R26 = 26,  // THR
  R27 = 27,  // PP
  R28 = 28,  // HEAP_BITS
  R29 = 29,  // FP
  R30 = 30,  // LR
  R31 = 31,  // ZR, CSP
  kNumberOfCpuRegisters = 32,
  kNoRegister = -1,
  kNoRegister2 = -2,

  // These registers both use the encoding R31, but to avoid mistakes we give
  // them different values, and then translate before encoding.
  CSP = 32,
  ZR = 33,

  // Aliases.
  IP0 = R16,
  IP1 = R17,
  SP = R15,
```

Figure 13: Screenshot of Dart-SDK source code Non-Standard Stack pointer

Dumping Flutter Heap to Retrieve Object Pool

To get deserialized objects which are present in the memory heap, heap memory must be dumped and then it can be imported in Ghidra for further analysis. Frida will be used in this tutorial for the same purpose. First of all, we need to get information about object pool and heap start address which can be retrieved by running the following Frida script:

```
//frida -U -f <package> -l frida.js
function hookFunc() {
 var dumpOffset = '0x000f9bdc' // Function to hook
   var argBufferSize = 150
   var baseApp = Module.findBaseAddress('App') // libapp.so (Android) or App (IOS)

   var codeOffset = baseApp.add(dumpOffset)
   console.log('')
   console.log('Wait..... ')
   Interceptor.attach(codeOffset, {
      onEnter: function(args) {

         var objectPool = this.context.x27;
         var baseHeap = this.context.x23;
         var bitMask = 0xF00000000;
         var baseHeapMasked = baseHeap.and(0xF00000000);
         let modules = Process.enumerateRanges("r-");
         for (var i = 0; i < modules.length; i++) {
         if((modules[i].base.and(bitMask)).compare(baseHeapMasked) == 0)
console.log(modules[i].base);
         }
         console.log("App Binary Base Address: "+baseApp)
         console.log("Heap Base Address (X23): "+ baseHeap);
         console.log("Object Pool Address (X27): " + objectPool);
         console.log("Number of Objects in the Object Pool:
"+Memory.readPointer(objectPool.add(0x8)));
         console.log("Heap bitmask: "+baseHeap.and(0xF00000000))
      },
      onLeave: function(retval) {

      }
    });
  }
   setTimeout(hookFunc, 1000)
```

The output of the above script is as follows:

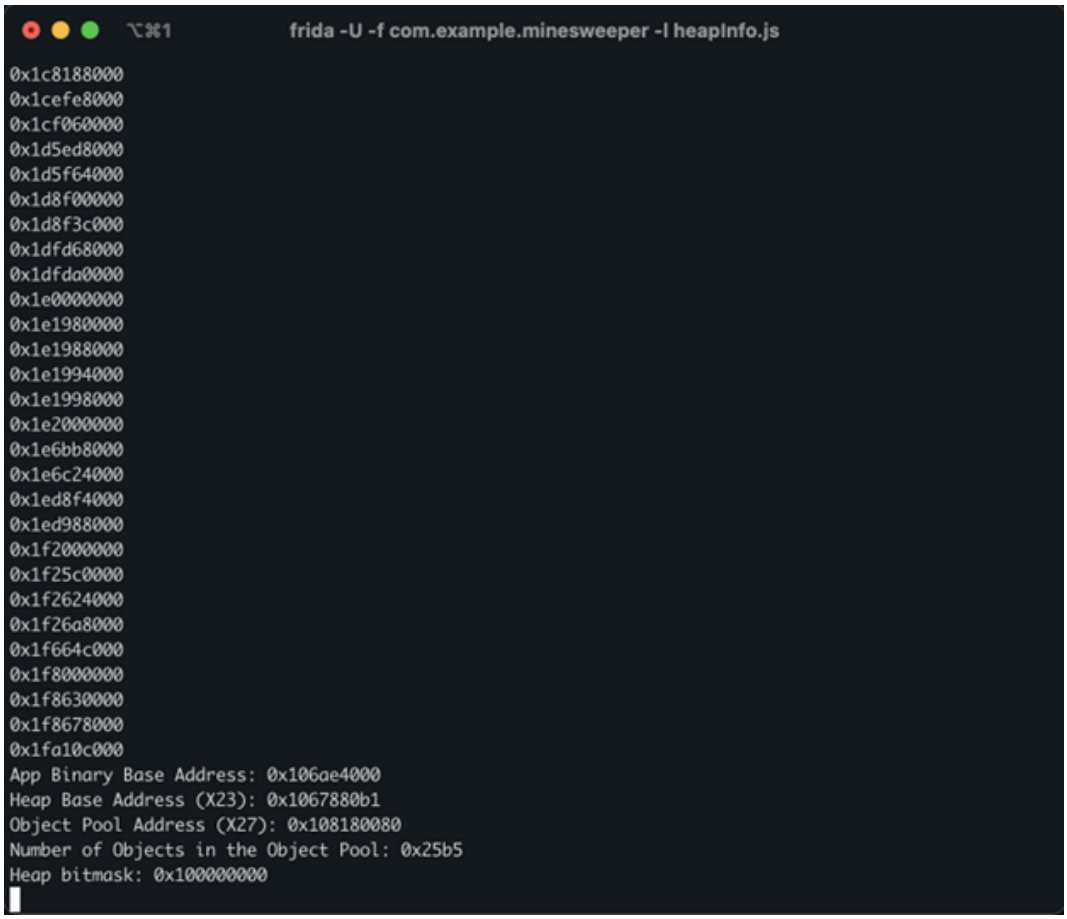


Figure 14: Screenshot of Output from Frida Hook

When the script is executed multiple times, it can be observed that the Heap Base Address and Object Pool Address will change every time. The problem with dumping the entire application memory is that it can be very large and impractical. To address this issue, the script retrieves the heap base address, which in the example case is 0x1067880b1. It should be possible to dump memory addresses that start with 0x1…….. which will contain the heap and object pool. It can be accomplished using Bitmask. Using this logic the script below dumps the flutter heap:

```
//frida -U -f <package> -l frida.js
var APP_DATA_DIR = "/var/mobile/Containers/Data/Application/--/Documents/" // make
sure that the directory is writable
function dump_memory(baseHeap, dump_directory){
  let modules = Process.enumerateRanges("r--");
  let i, module;
  let module_file;
  var bitMask = 0xF00000000; //bitMask to retain msb and set remaining to zero
  var baseHeapMasked = baseHeap.and(0xF00000000);
  module_file = new File(dump_directory + "ranges.json", "wb");
  module_file.write(JSON.stringify(modules, null, 2));
  module_file.close();
  for (i = 0; i < modules.length; i++) {
    try {
      module = modules[i];
      if ((modules[i].base.and(bitMask)).compare(baseHeapMasked) == 0) {
console.log(`Dumping memory into ${dump_directory + module.base}`);
module_file = new File(dump_directory + module.base, "wb");
module_file.write(module.base.readByteArray(module.size));
module_file.close();
      }
    } catch(ex) {
      console.log(ex);
      console.log(JSON.stringify(module, null, 2));
    }
  }
}
function hookFunc() {
  var dumpOffset = '0x000f9bdc' //restartGame (Function to hook)
    var baseApp = Module.findBaseAddress('App') // libapp.so (Android) or App (IOS)

    var codeOffset = baseApp.add(dumpOffset)
    console.log('')
    console.log('Wait..... ')

    Interceptor.attach(codeOffset, {
      onEnter: function(args) {

var objectPool = this.context.x27;
var baseHeap = this.context.x23;
var bitMask = 0xF00000000;
var baseHeapMasked = baseHeap.and(0xF00000000);

let modules = Process.enumerateRanges("r-");
for (var i = 0; i < modules.length; i++) {
if((modules[i].base.and(bitMask)).compare(baseHeapMasked) == 0)
console.log(modules[i].base);
}
```

```
dump_memory(baseHeap, APP_DATA_DIR);
console.log("App Binary Base Address: "+baseApp)
console.log("Heap Base Address (X23): "+ baseHeap);
console.log("Object Pool Address (X27): " + objectPool);
console.log("Number of Objects in the Object Pool:
"+Memory.readPointer(objectPool.add(0x8)));
console.log("Heap bitmask: "+baseHeap.and(0xF00000000))       },
onLeave: function(retval) {

}
       });
    }

    setTimeout(hookFunc, 1000)
```

The directory where the Flutter application is installed on the device is usually writable by the app itself. To determine the location of the app installation, the r2Frida tool can be used to print the relevant information, as shown in the figure below.

Figure 15: Screenshot of r2Frida output

```
    module_file = new File(dump_directory + "ranges.json", "wb");
    module_file.write(JSON.stringify(modules, null, 2));
    module_file.close();
    for (i = 0; i < modules.length; i++) {
        try {
            module = modules[i];
            if ((modules[i].base.and(bitMask)).compare(baseHeapMasked) == 0) {
                console.log(`Dumping memory into ${dump_directory + module.base}`);
                module_file = new File(dump_directory + module.base, "wb");
                module_file.write(module.base.readByteArray(module.size));
                module_file.close();
            }
        } catch(ex) {
            console.log(ex);
            console.log(JSON.stringify(module, null, 2));
        }
    }
}
function hookFunc() {

    var dumpOffset = '0x000f9bdc' //restartGame
    var baseApp = Module.findBaseAddress('App') // libapp.so (Android) or App (IOS)

    var codeOffset = baseApp.add(dumpOffset)
    console.log('')
    console.log('Wait..... ')

    Interceptor.attach(codeOffset, {
        onEnter: function(args) {
```

Figure 16: Screenshot of Frida dumpMemory.js script

```
⌥⌘1                frida -U -f com.example.minesweeper -l dumpMemory.js
Dumping memory into /var/mobile/Containers/Data/Application/A42EB282-BC11-4584-8A45-751C49564BE5/Documents/0x1c8188000
Dumping memory into /var/mobile/Containers/Data/Application/A42EB282-BC11-4584-8A45-751C49564BE5/Documents/0x1cefe8000
Dumping memory into /var/mobile/Containers/Data/Application/A42EB282-BC11-4584-8A45-751C49564BE5/Documents/0x1cf060000
Dumping memory into /var/mobile/Containers/Data/Application/A42EB282-BC11-4584-8A45-751C49564BE5/Documents/0x1d5ed8000
Dumping memory into /var/mobile/Containers/Data/Application/A42EB282-BC11-4584-8A45-751C49564BE5/Documents/0x1d5f64000
Dumping memory into /var/mobile/Containers/Data/Application/A42EB282-BC11-4584-8A45-751C49564BE5/Documents/0x1d8f00000
Dumping memory into /var/mobile/Containers/Data/Application/A42EB282-BC11-4584-8A45-751C49564BE5/Documents/0x1d8f3c000
Dumping memory into /var/mobile/Containers/Data/Application/A42EB282-BC11-4584-8A45-751C49564BE5/Documents/0x1dfd68000
Dumping memory into /var/mobile/Containers/Data/Application/A42EB282-BC11-4584-8A45-751C49564BE5/Documents/0x1dfda0000
Dumping memory into /var/mobile/Containers/Data/Application/A42EB282-BC11-4584-8A45-751C49564BE5/Documents/0x1e0000000
Dumping memory into /var/mobile/Containers/Data/Application/A42EB282-BC11-4584-8A45-751C49564BE5/Documents/0x1e1980000
Dumping memory into /var/mobile/Containers/Data/Application/A42EB282-BC11-4584-8A45-751C49564BE5/Documents/0x1e1988000
Dumping memory into /var/mobile/Containers/Data/Application/A42EB282-BC11-4584-8A45-751C49564BE5/Documents/0x1e1994000
Dumping memory into /var/mobile/Containers/Data/Application/A42EB282-BC11-4584-8A45-751C49564BE5/Documents/0x1e1998000
Dumping memory into /var/mobile/Containers/Data/Application/A42EB282-BC11-4584-8A45-751C49564BE5/Documents/0x1e2000000
Dumping memory into /var/mobile/Containers/Data/Application/A42EB282-BC11-4584-8A45-751C49564BE5/Documents/0x1e6bb8000
Dumping memory into /var/mobile/Containers/Data/Application/A42EB282-BC11-4584-8A45-751C49564BE5/Documents/0x1e6c24000
Dumping memory into /var/mobile/Containers/Data/Application/A42EB282-BC11-4584-8A45-751C49564BE5/Documents/0x1ed8f4000
Dumping memory into /var/mobile/Containers/Data/Application/A42EB282-BC11-4584-8A45-751C49564BE5/Documents/0x1ed988000
Dumping memory into /var/mobile/Containers/Data/Application/A42EB282-BC11-4584-8A45-751C49564BE5/Documents/0x1f2000000
Dumping memory into /var/mobile/Containers/Data/Application/A42EB282-BC11-4584-8A45-751C49564BE5/Documents/0x1f25c0000
Dumping memory into /var/mobile/Containers/Data/Application/A42EB282-BC11-4584-8A45-751C49564BE5/Documents/0x1f2624000
Dumping memory into /var/mobile/Containers/Data/Application/A42EB282-BC11-4584-8A45-751C49564BE5/Documents/0x1f26a8000
Dumping memory into /var/mobile/Containers/Data/Application/A42EB282-BC11-4584-8A45-751C49564BE5/Documents/0x1f664c000
Dumping memory into /var/mobile/Containers/Data/Application/A42EB282-BC11-4584-8A45-751C49564BE5/Documents/0x1f8000000
Dumping memory into /var/mobile/Containers/Data/Application/A42EB282-BC11-4584-8A45-751C49564BE5/Documents/0x1f8630000
Dumping memory into /var/mobile/Containers/Data/Application/A42EB282-BC11-4584-8A45-751C49564BE5/Documents/0x1f8678000
Dumping memory into /var/mobile/Containers/Data/Application/A42EB282-BC11-4584-8A45-751C49564BE5/Documents/0x1fa10c000
App Binary Base Address: 0x106058000
Heap Base Address (X23): 0x101f20e00
Object Pool Address (X27): 0x107780080
Number of Objects in the Object Pool: 0x25b5
Heap bitmask: 0x100000000
```

Figure 17: Screenshot of dumpMemory.js script in action

As shown in the figure above, all the memory addresses that start with 0x1…….. were dumped as Heap Base Address starts with 0x1…….. also note that Object Pool Address should be a part of heap. It is important to **note down all the above** addresses as they are required to be used in Ghidra. Copy the memory dump from the device to the PC.

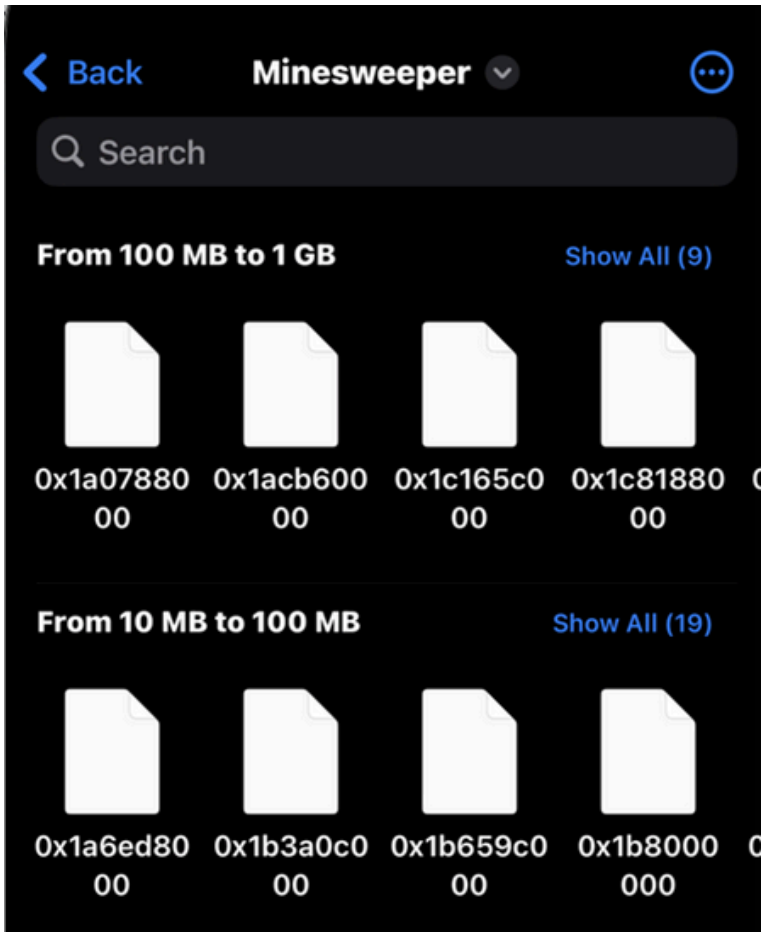


Figure 18: Screenshot of all the memory dumps

Importing Memory Dump in Ghidra

After copying the memory dump to the PC, it's time to start importing the memory dumps to Ghidra. Before doing that, the Base Address of the binary needs to be set to match it with the addresses in the dump file. To do that, go to Window > Memory Map

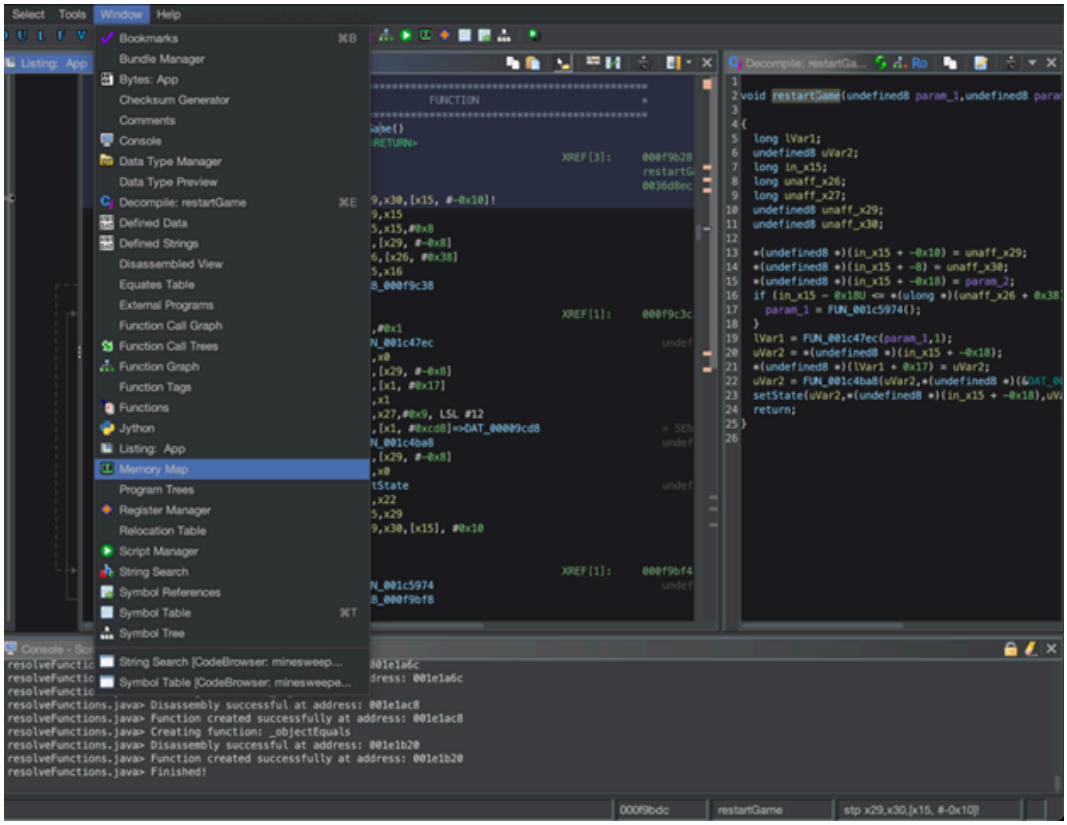


Figure 19: Screenshot of App Binary in Ghidra

Click on Set Image Base on the top right:

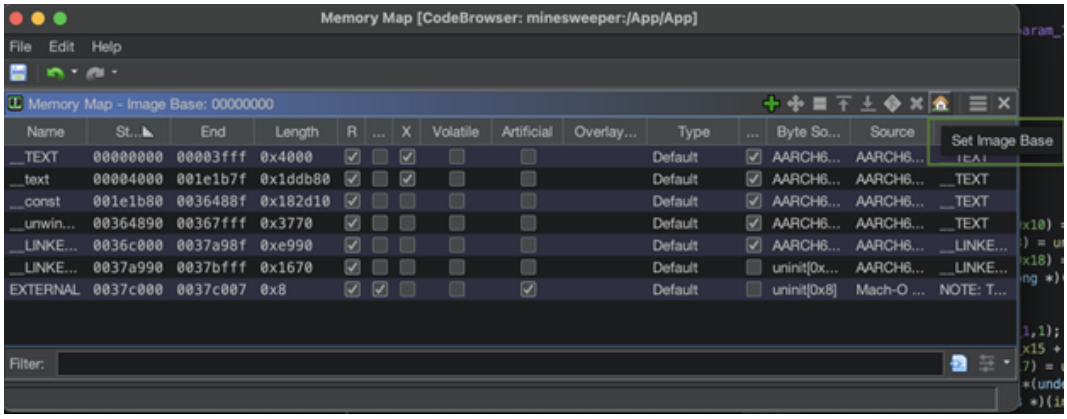


Figure 20: Screenshot of App Binary in Ghidra Memory Map Window

Enter the Base Image Address that was retrieved during memory dump output:

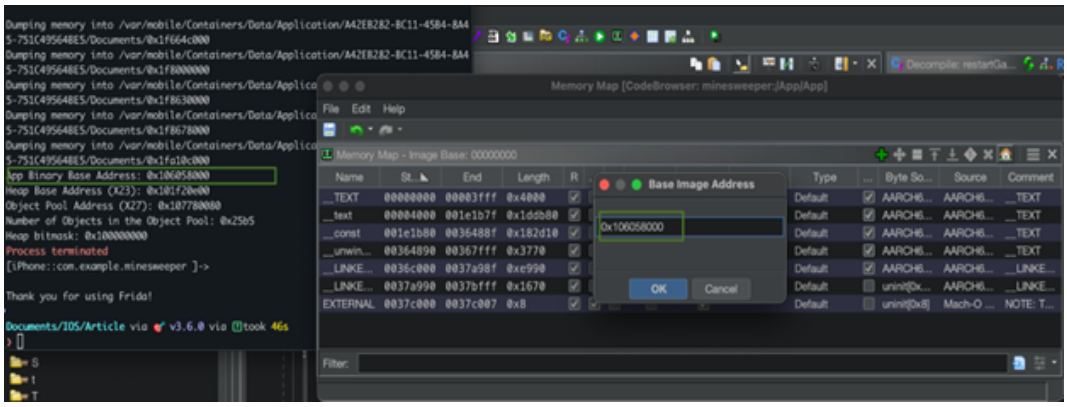


Figure 21: Screenshot of App Binary rebasing in Ghidra Memory Map Window

> ***Note:*** Make sure to set the Base Image Address before importing the memory dump into Ghidra.

Copy the memory dumps to a dedicated folder:

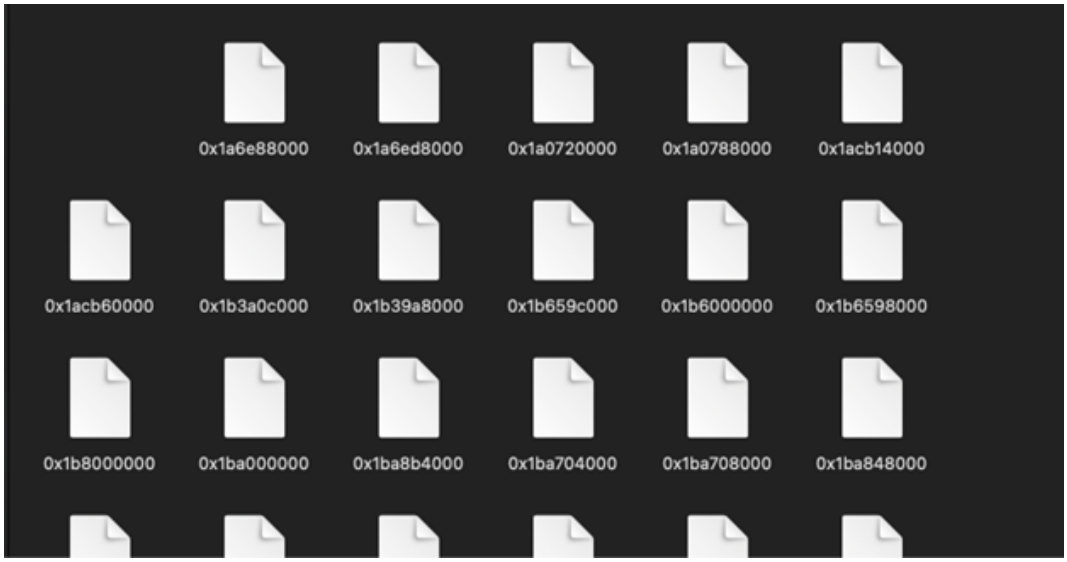


Figure 22: Screenshot of Memory dumps in dedicated folder

The following Ghidra script can be used to import memory dumps:

```
import ghidra.app.script.GhidraScript;
import ghidra.program.model.mem.Memory;
import ghidra.program.model.mem.MemoryBlock;
import ghidra.program.model.address.Address;
import ghidra.util.task.TaskMonitor;
import java.io.File;
import java.io.FileInputStream;
import java.io.BufferedInputStream;
import java.io.DataInputStream;
import java.io.IOException;

public class ImportMemoryDump extends GhidraScript {

  @Override
  public void run() {
     try {
        // Ask the user to select a folder containing the memory dump files
        File folder = askDirectory("Select folder", "Import Memory Dump Files");

        // Early exit if folder is not selected or invalid
if (folder == null || !folder.exists() || !folder.isDirectory()) {
println("Error: Invalid folder selected. Script aborted.");
return;
        }

        // List all files in the selected folder
        File[] dumpFiles = folder.listFiles();

        // Check if folder contains memory dump files
if (dumpFiles == null || dumpFiles.length == 0) {
println("No memory dump files found in the folder.");
return;
        }

        // Process each memory dump file
for (File dumpFile : dumpFiles) {
try {
importMemoryFromFile(dumpFile);
} catch (Exception e) {
// Catch exceptions for each individual file and log the error without stopping the script
println("Error processing file: " + dumpFile.getName());
println("Exception: " + e.getMessage());
e.printStackTrace();
}
        }

        println("All memory dump files processed.");
```

```
        } catch (Exception e) {
            // Catch any unexpected exceptions that occur during the entire script execution
            println("An unexpected error occurred during script execution.");
            println("Exception: " + e.getMessage());
            e.printStackTrace();
        }
    }

    /**
     * Imports a memory dump file into the current program.
     *
     * @param dumpFile The file containing the memory dump.
     */
    private void importMemoryFromFile(File dumpFile) {
        try (FileInputStream fis = new FileInputStream(dumpFile);
BufferedInputStream bis = new BufferedInputStream(fis);
DataInputStream dis = new DataInputStream(bis)) {

            // Get the size of the memory dump file
            long fileSize = dumpFile.length();
            long baseAddress = extractBaseAddressFromFileName(dumpFile.getName());

            // Import the file as a memory block using the InputStream approach
            importMemoryBlockToGhidra(dumpFile.getName(), baseAddress, fis, fileSize);

            println("Successfully imported memory dump file: " + dumpFile.getName());

        } catch (IOException e) {
            // Handle file read exceptions
            println("Error reading memory dump file: " + dumpFile.getName());
            println(e.getMessage());
            e.printStackTrace();
        } catch (NumberFormatException e) {
            // Handle address parsing exceptions
            println("Error parsing address from file name: " + dumpFile.getName());
            println(e.getMessage());
            e.printStackTrace();
        } catch (Exception e) {
            // Catch any other unforeseen exceptions
            println("An unexpected error occurred with file: " + dumpFile.getName());
            println(e.getMessage());
            e.printStackTrace();
        }
    }
```

```
    /**
     * Extracts the base address from the file name, assuming the base address is
     * encoded as hexadecimal starting from the 3rd character of the file name.
     *
     * @param fileName The name of the memory dump file.
     * @return The base address as a long.
     * @throws NumberFormatException If the address string is not a valid hexadecimal
number.
     */
    private long extractBaseAddressFromFileName(String fileName) throws
NumberFormatException {
        // Extract address from file name (assumed to start from the 3rd character)
        String addressStr = fileName.substring(2);
return Long.parseLong(addressStr, 16);
    }

    /**
     * Imports the memory dump data into the Ghidra program's memory at the specified
address.
     *
     * @param blockName The name of the memory block (usually the dump file's name).
     * @param baseAddress The base address at which to import the memory dump.
     * @param inputStream The InputStream containing the memory dump data.
     * @param length The length of the memory dump data.
     */
    private void importMemoryBlockToGhidra(String blockName, long baseAddress,
FileInputStream inputStream, long length) {
        try {
            Address address = toAddr(baseAddress);
            Memory memory = currentProgram.getMemory();

            // Create a new memory block and initialize it with the InputStream
            TaskMonitor monitor = getMonitor(); // You can optionally pass a TaskMonitor to
track progress
            MemoryBlock block = memory.createInitializedBlock(blockName, address,
inputStream, length, monitor, false);

            println("Memory block successfully created for file: " + blockName);
        } catch (Exception e) {
            // Handle errors related to importing memory
            println("Error importing memory block: " + blockName);
            println(e.getMessage());
            e.printStackTrace();
        }
    }
}
```

After running the above script Memory Map should look like this:

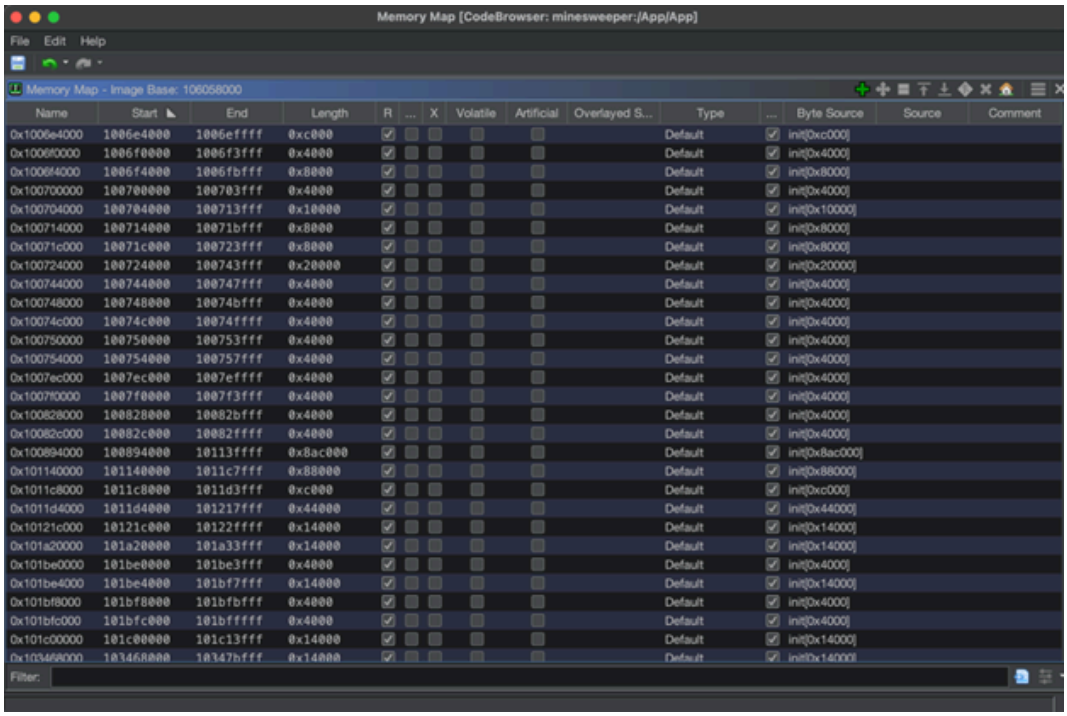
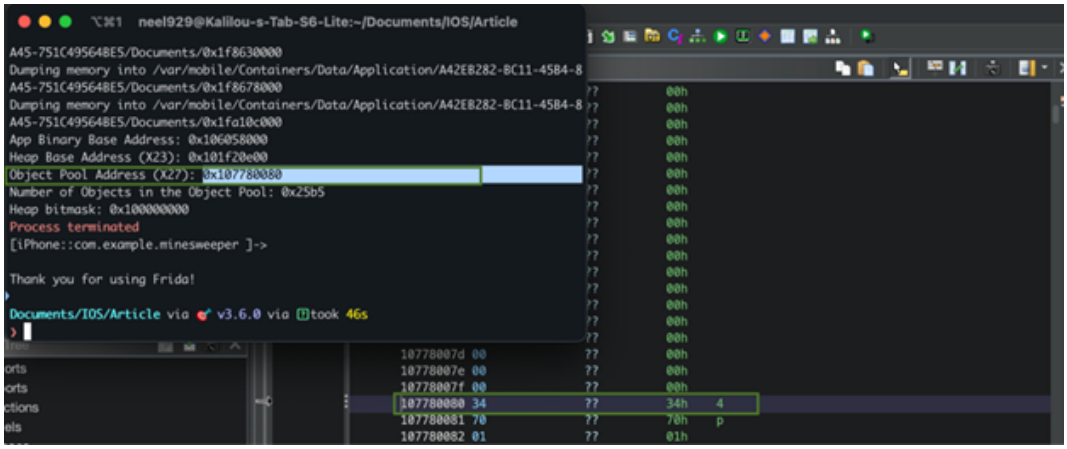


Figure 23: Screenshot of Resolved Memory Map

Resolving Object Pool Array

After Importing Memory Dump in Ghidra and Jumping onto the Address where object pool array is located:



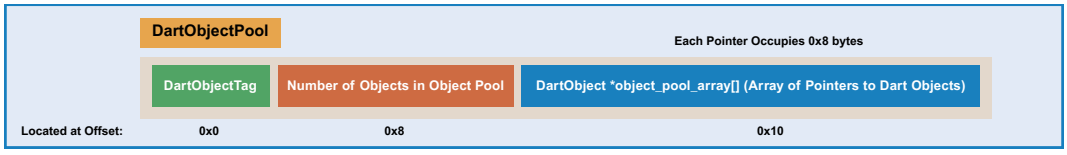

```
struct DartObjectTag
{
 charis_canonical_and_gc;
 charsize_tag;
 __int16cid;
};
```

The Dart object pool is a Dart object itself thus it starts with a DartObjectTag followed by the number of Dart objects in the pool (at offset 8) and ends with an array of pointers to Dart objects.

```
struct DartObjectPool
{
 DartObjectTagtag;

 __int64nb_dart_objects_in_object_pool;
 DartObject*object_pool_array[];
};
```

The addresses stored in the Dart object pool array are odd. Along with addresses, Small Integers (smi) are also stored on the array. To differentiate between Small Integers and Pointers, Small Integers have their LSB set to 0 and Pointers are made odd. Thus odd values on the Dart object pool array are pointers and even values are small integers. The real value of a Pointer is the value of Pointer subtracted by 1 (i.e. if ox1078881 is the value stored on Dart object pool, the real address would be 0x1078880). In case of Small Integers, the value must be right shifted by 1 (i.e. SMI value 50 resolves to integer value 50>>1 = 25).



Ghidra needs to be instructed to create the object pool array. The following script can be used to create object pool array (Make sure to replace object pool address in the script) :

```
import ghidra.app.script.GhidraScript;
import ghidra.program.model.address.Address;
import ghidra.program.model.data.PointerDataType;
import ghidra.program.model.data.*;

public class DartObjectPoolPointers extends GhidraScript {
  @Override
  public void run() {
     try {
        // Define the base address of the object pool
        Address objectPoolBaseAddress = toAddr(0x107780080L);

        // Define the start address of the array (object pool base + 0x10)
        Address arrayStartAddress = objectPoolBaseAddress.add(0x10);

        // Define the end address of the array
        Address arrayEndAddress = objectPoolBaseAddress.add(0x8 * 0x25B5);

        // Define the data type for a pointer
        DataType pointerDataType = new PointerDataType();

        int pointerIndex = 0;

        // Traverse the memory addresses in the array
while (arrayStartAddress.compareTo(arrayEndAddress) <= 0) {
pointerIndex++;
try {
Address pointerAddress = toAddr(getLong(arrayStartAddress));

// Check if the pointer address is misaligned (odd)
if ((pointerAddress.getOffset() & 1) != 0) {
                println("Before: "+arrayStartAddress.toString());
                  // Clear the existing data at this address
                  clearListing(
                     arrayStartAddress,
                     arrayStartAddress.add(pointerDataType.getLength() - 1)
                  );

                  // Correct the misalignment
                  setLong(arrayStartAddress, pointerAddress.getOffset());

                  // Create a new pointer data type at this address
                  createData(arrayStartAddress, pointerDataType);

                  //println("Corrected pointer: " + pointerAddress);
                  println("Pointer index: " + pointerIndex);
```

```
 }
} catch (Exception e) {
println("Error processing address " + arrayStartAddress + ": " + e.getMessage());
}

// Move to the next 8-byte aligned address
arrayStartAddress = arrayStartAddress.add(8);
          }
      } catch (Exception e) {
          println("An unexpected error occurred: " + e.getMessage());
      }
   }
}
```

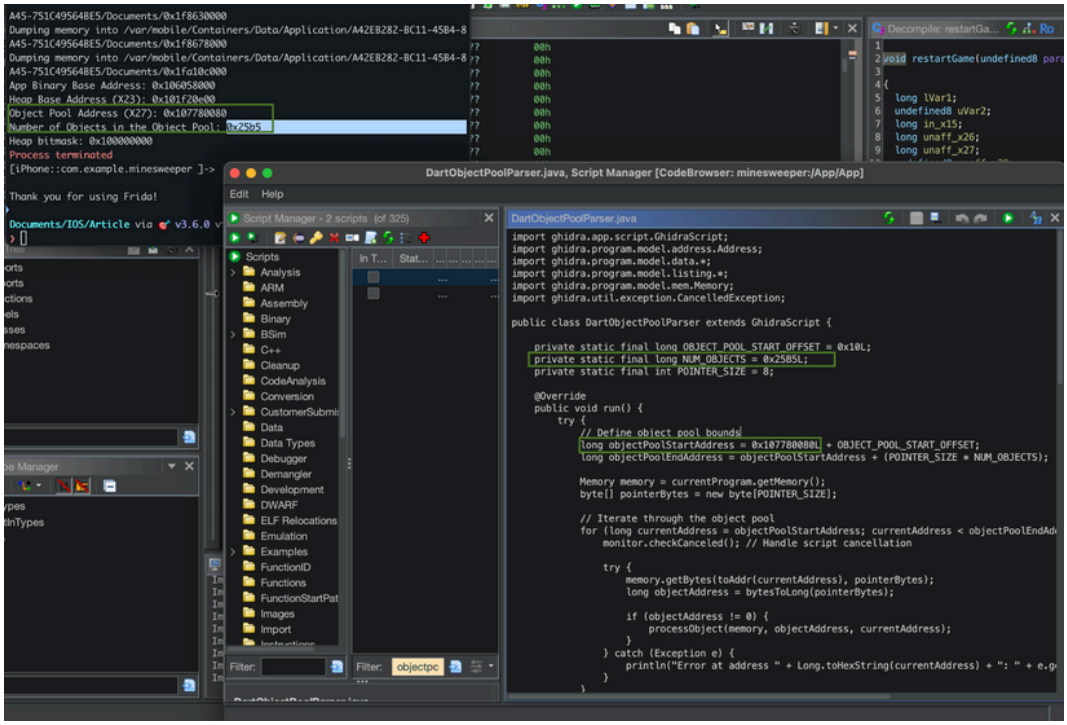


Figure 24: Screenshot of DartObjectPoolParser Ghidra Script

After running the above script, object pool array gets created and pointers to dart objects should be visible as shown below:

```
107780090 61 40 d1        addr        DAT_105d14061
          05 01 00
          00 00
107780098 e1 40 d1        addr        DAT_105d140e1
          05 01 00
          00 00
1077800a0 61 41 d1        addr        DAT_105d14161
          05 01 00
          00 00
1077800a8 e1 41 d1        addr        DAT_105d141e1
          05 01 00
          00 00
1077800b0 61 42 d1        addr        DAT_105d14261
          05 01 00
          00 00
1077800b8 61 6c d1        addr        DAT_105d16c61
```

Figure 25: Screenshot of Dart Object Pool Array in Ghidra

Creating Dart Objects

After creating the pointers in the Dart Object Pool Array, Ghidra doesn't have any information about the structure of Dart Objects as shown below:

```
                    DAT_105d140e1
105d140e1 28              ??         28h    (
105d140e2 01              ??         01h
105d140e3 00              ??         00h
105d140e4 e2              ??         E2h
105d140e5 3f              ??         3Fh    ?
105d140e6 06              ??         06h
105d140e7 00              ??         00h
105d140e8 d8              ??         D8h
105d140e9 26              ??         26h    &
105d140ea 06              ??         06h
105d140eb 06              ??         06h
105d140ec 01              ??         01h
105d140ed 00              ??         00h
105d140ee 00              ??         00h
105d140ef 00              ??         00h
105d140f0 d8              ??         D8h
```

Figure 26: Screenshot of unresolved Dart Object in Ghidra

Identification of Dart Objects

The structure of Dart Deserialized Object is as follows (as evident from dart-sdk source code):

```
struct DartDeserializedObject
{
    char is_canonical_and_gc;
    char size_tag;
    __int16 cid;
    <intpadding>
    __int64 s_len; //s_len is stored as Small Integer so real length is s_len>>1;
    char s[];
};
```

In order to create the objects, the object pool array has to be traversed and the structure of the objects need to be created manually by writing Ghidra script.
We will utilize the ClassId which is 0x5 or 5L (source link)

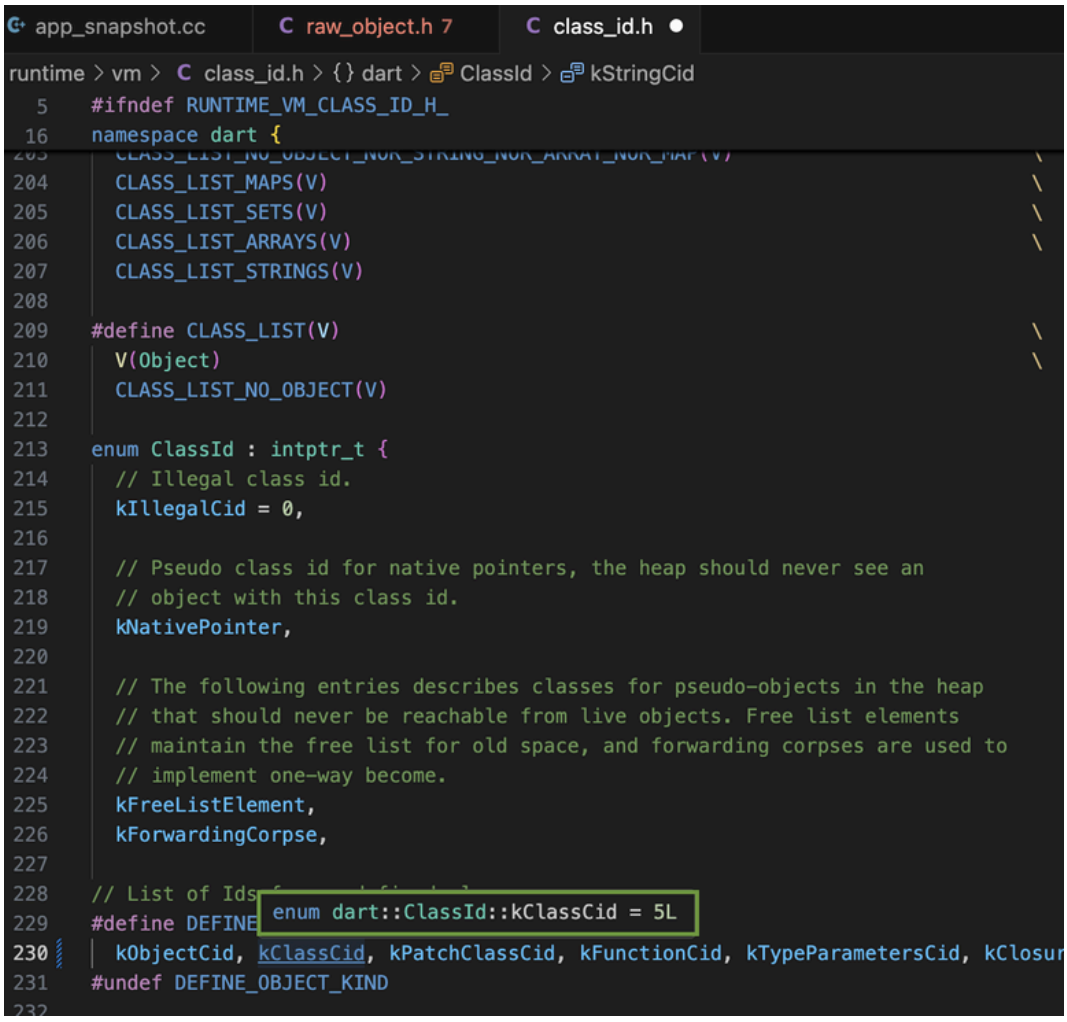


Figure 27: Screenshot of ClassId from Dart-SDK Source Code

The below script can be used to create Dart String Objects:

```
import ghidra.app.script.GhidraScript;
import ghidra.program.model.address.Address;
import ghidra.program.model.data.*;
import ghidra.program.model.listing.*;
import ghidra.program.model.mem.Memory;
import ghidra.util.exception.CancelledException;

public class DartObjectPoolParser extends GhidraScript {
   private static final long OBJECT_POOL_START_OFFSET = 0x10L;
// Change this to the number of section as per your dump file
   private static final long NUM_OBJECTS = 0x25B5L;
// Pointer size is 8 for aarch64
   private static final int POINTER_SIZE = 8;

   @Override
   public void run() {
      try {
         // Define object pool boundary
         long objectPoolStartAddress = 0x107780080L + OBJECT_POOL_START_OFFSET;
         long objectPoolEndAddress = objectPoolStartAddress + (POINTER_SIZE *
NUM_OBJECTS);

         Memory memory = currentProgram.getMemory();
         byte[] pointerBytes = new byte[POINTER_SIZE];

         // Iterate through the object pool
for (long currentAddress = objectPoolStartAddress; currentAddress <
objectPoolEndAddress; currentAddress += POINTER_SIZE) {
monitor.checkCanceled(); // Handle script cancellation

try {
memory.getBytes(toAddr(currentAddress), pointerBytes);
long objectAddress = bytesToLong(pointerBytes);

if (objectAddress != 0) {
                  processObject(memory, objectAddress, currentAddress);
}
} catch (Exception e) {
println("Error at address " + Long.toHexString(currentAddress) + ": " + e.getMessage());
}
         }
```

```
            println("Dart object pool structures created successfully!");
        } catch (CancelledException e) {
            println("Script canceled by user.");
        } catch (Exception e) {
            println("Error: " + e.getMessage());
        }
    }

    private void processObject(Memory memory, long objectAddress, long pointerAddress)
throws Exception {
        byte[] structBytes = new byte[4];
        memory.getBytes(toAddr(objectAddress), structBytes);

        // Extract CID field
        short cid = (short) ((structBytes[2] & 0xFF) | ((structBytes[3] & 0xFF) << 8));

        // Filter objects based on CID values
        if (cid == 0x5) {
            println("CID: " + Integer.toHexString(cid));
            println("Pointer value at " + Long.toHexString(pointerAddress) + ": " +
Long.toHexString(objectAddress));
            createStringDataStructure(toAddr(objectAddress));
        }
    }

    private void createStringDataStructure(Address structAddress) {
        try {
            // Read string length and content
            long stringLength = getLong(structAddress.add(0x8)) >> 1;
            StringBuilder stringContent = new StringBuilder();

for (int i = 0; i < stringLength; i++) {
byte b = getByte(structAddress.add(0x10 + i));
stringContent.append((char) (b & 0xFF));
            }

            String stringName = stringContent.toString();

            // Define DartString structure
            Structure dartStringStruct = new StructureDataType(stringName, 0);
            dartStringStruct.add(ByteDataType.dataType, "is_canonical_and_gc", null);
            dartStringStruct.add(ByteDataType.dataType, "size_tag", null);
            dartStringStruct.add(ShortDataType.dataType, "cid", null);
            dartStringStruct.add(DWordDataType.dataType, "padding", null); // Padding for
alignment
            dartStringStruct.add(QWordDataType.dataType, "string_length", null);
            dartStringStruct.add(new ArrayDataType(CharDataType.dataType, (int)
stringLength, 1), "string_content", null);
```

```
            DataUtilities.createData(
currentProgram,
structAddress,
dartStringStruct,
dartStringStruct.getLength(),
DataUtilities.ClearDataMode.CLEAR_ALL_UNDEFINED_CONFLICT_DATA
            );

            println("DartString structure created successfully at address " + structAddress);
        } catch (Exception e) {
            println("Failed to create DartString structure: " + e.getMessage());
        }
    }

    private long bytesToLong(byte[] bytes) {
        long value = 0;
        for (int i = bytes.length - 1; i >= 0; i--) {
            value = (value << 8) | (bytes[i] & 0xFF);
        }
        // Adjust for object pool pointer odd values
        if (value > 0 && (value % 2L == 1L)) {
return value - 1;
        }
return 0L;
    }
}
```

After running the above script, it can be seen that Dart String Objects have been successfully created:

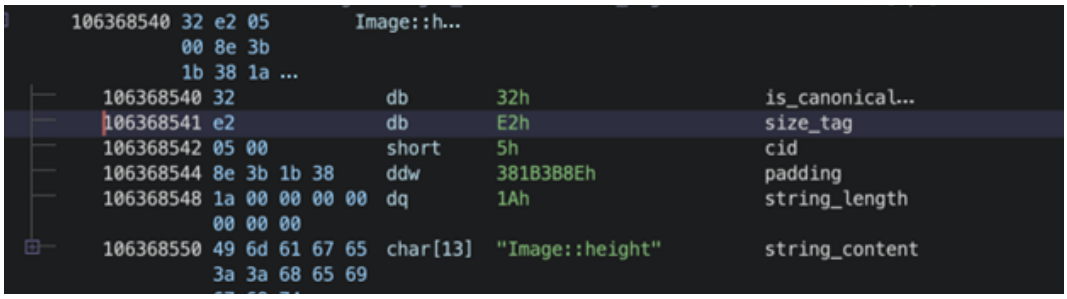


Figure 28: Screenshot of Dart String Object

Before moving forward, one more change needs to be applied to the Dart Object Pool array as the pointers are odd in order to differentiate between Integer and the pointers in the array. In order for Ghidra to set the correct reference to string, the pointers should be pointing to start of the string. In order to resolve that, one should subtract all the pointers in the Dart Object Pool using the below script:

```
import ghidra.app.script.GhidraScript;
import ghidra.program.model.address.Address;
import ghidra.program.model.data.PointerDataType;
import ghidra.program.model.data.*;

public class DartObjectPoolPointers extends GhidraScript {
  @Override
  public void run() {
     try {
        // Define the base address of the object pool
        Address objectPoolBaseAddress = toAddr(0x107780080L);

        // Define the start address of the array (object pool base + 0x10)
        Address arrayStartAddress = objectPoolBaseAddress.add(0x10);

        // Define the end address of the array
        Address arrayEndAddress = objectPoolBaseAddress.add(0x8 * 0x25B5);

        // Define the data type for a pointer
        DataType pointerDataType = new PointerDataType();

        int pointerIndex = 0;

        // Traverse the memory addresses in the array
while (arrayStartAddress.compareTo(arrayEndAddress) <= 0) {
pointerIndex++;
try {
Address pointerAddress = toAddr(getLong(arrayStartAddress));

// Check if the pointer address is misaligned (odd)
if ((pointerAddress.getOffset() & 1) != 0) {
                    //println("Misaligned address found at: " + arrayStartAddress);
                   pointerAddress = pointerAddress.subtract(1);
                  println("Before: "+arrayStartAddress.toString());
                   // Clear the existing data at this address
                   clearListing(
                      arrayStartAddress,
                      arrayStartAddress.add(pointerDataType.getLength() - 1)
                   );

                   // Correct the misalignment
                   setLong(arrayStartAddress, pointerAddress.getOffset());
```

```
                // Create a new pointer data type at this address
                createData(arrayStartAddress, pointerDataType);

                //println("Corrected pointer: " + pointerAddress);
                println("Pointer index: " + pointerIndex);
}
} catch (Exception e) {
println("Error processing address " + arrayStartAddress + ": " + e.getMessage());
}

// Move to the next 8-byte aligned address
arrayStartAddress = arrayStartAddress.add(8);
        }
    } catch (Exception e) {
        println("An unexpected error occurred: " + e.getMessage());
    }
  }
}
```

It can be seen that Object Pool Array pointers now point to the correct address:

```
          00 00
107780108 00 63 88      addr      DAT_107886300                  = 34h
          07 01 00
          00 00
107780110 60 0d 34      addr      key_106340d60
          06 01 00
          00 00
107780118 80 80 d0      addr      DAT_105d08080                  = 72h
          05 01 00
          00 00
107780120 c0 c5 23      addr      value_10623c5c0
          06 01 00
          00 00
107780128 10 86 73      addr      DAT_106738610
          06 01 00
          00 00
107780130 40 69 88      addr      DAT_107886940                  = 34h
          07 01 00
          00 00
107780138 10 b0 23      addr      DAT_10623b010                  = 32h
          06 01 00
          00 00
107780140 c0 95 30      addr      dart:ui_1063095c0
          06 01 00
          00 00
107780148 40 85 36      addr      Image::height_106368540
          06 01 00
          00 00
```

Figure 29: Screenshot of Dart Pool Array with Correct Address

Adding References Between Code and Data

To add references between code and data, the register X27, which points to the Object Pool Array, needs to be provided a default value. Then, Ghidra will be able to add references between the code and data automatically.

In the figure shown below, it is still not known to Ghidra what object is being loaded, as the value of X27 is unavailable to Ghidra.

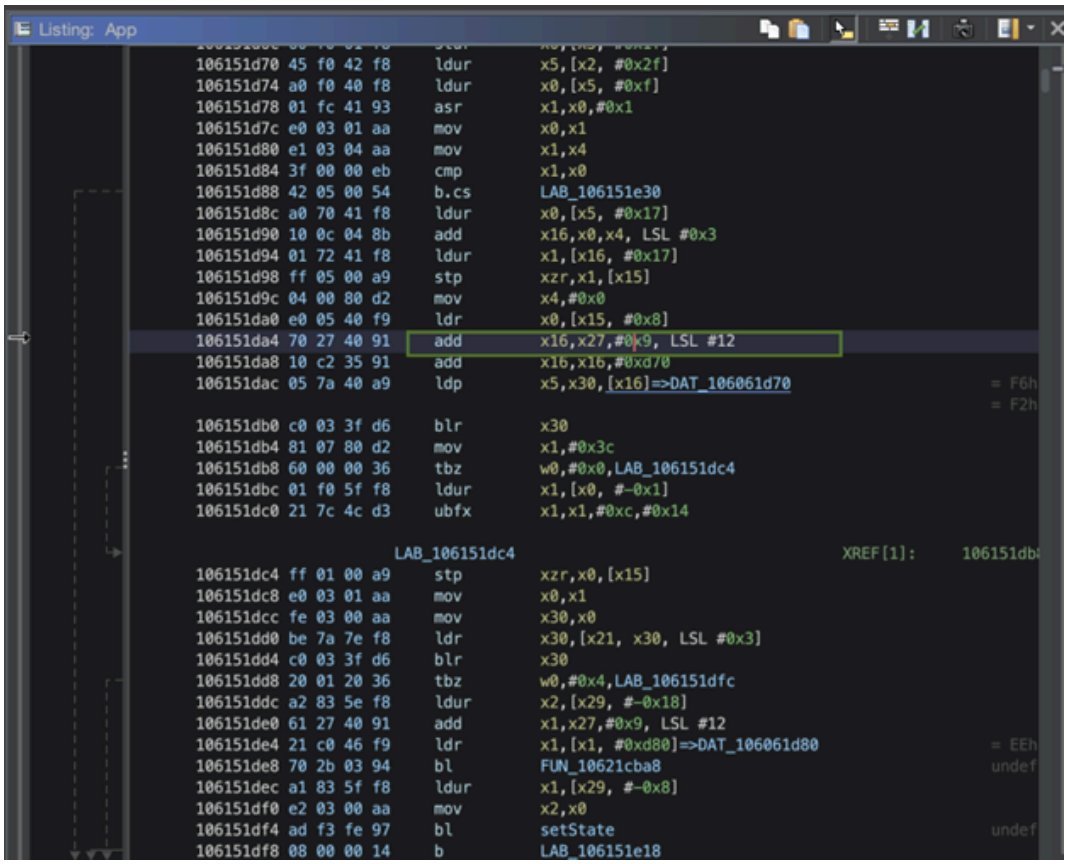


Figure 30: Screenshot of X27 register mystery

The value of the X27 register, which was retrieved during the memory dump, can be set.

As the TEXT/text section contains the code, the value of X27 can be replaced for the addresses that contain the TEXT/text section. Head over to the Memory Map to find the addresses. As shown in the figure below, the start address would be 0x106058000 and end address 0x106239b7f.

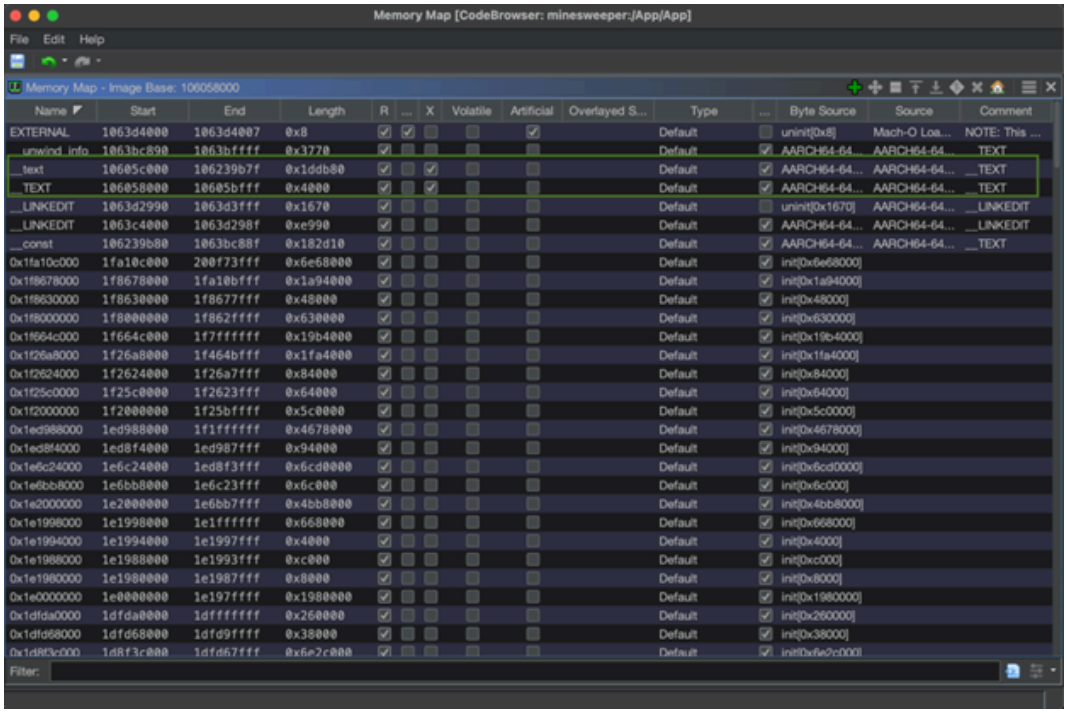


Figure 31: Screenshot of entire memory map

Run the script below:

```
import java.math.BigInteger;
import ghidra.app.script.GhidraScript;
import ghidra.program.model.address.Address;
import ghidra.program.model.lang.Register;
import ghidra.program.model.lang.RegisterValue;
import ghidra.program.model.listing.ProgramContext;

public class SetRegisterX27 extendsGhidraScript {

  @Override
  public void run() {
    try {
      // Define the register to modify
      Register x27Register= currentProgram.getRegister("x27");
      if (x27Register== null) {
        println("Register 'x27' not found in the current program.");
        return;
      }

      // Define the value to set for the register
      BigInteger registerValue= new BigInteger("107780080", 16); // Object Pool Address
      RegisterValue x27Value= new RegisterValue(x27Register, registerValue);
```

```
        // Define the address range to apply the value
        Address startAddress= toAddr(0x106058000L); // Replace with your start address
        Address endAddress= toAddr(0x106239B7FL); // Replace with your end address

        // Set the register value within the specified range
        ProgramContext programContext= currentProgram.getProgramContext();
        programContext.setRegisterValue(startAddress, endAddress, x27Value);

        println("Successfully set the register 'x27' value in the specified range.");
    } catch (Exception e) {
        println("Error: " + e.getMessage());
    }
  }
}
```

Go to register manager as shown below:

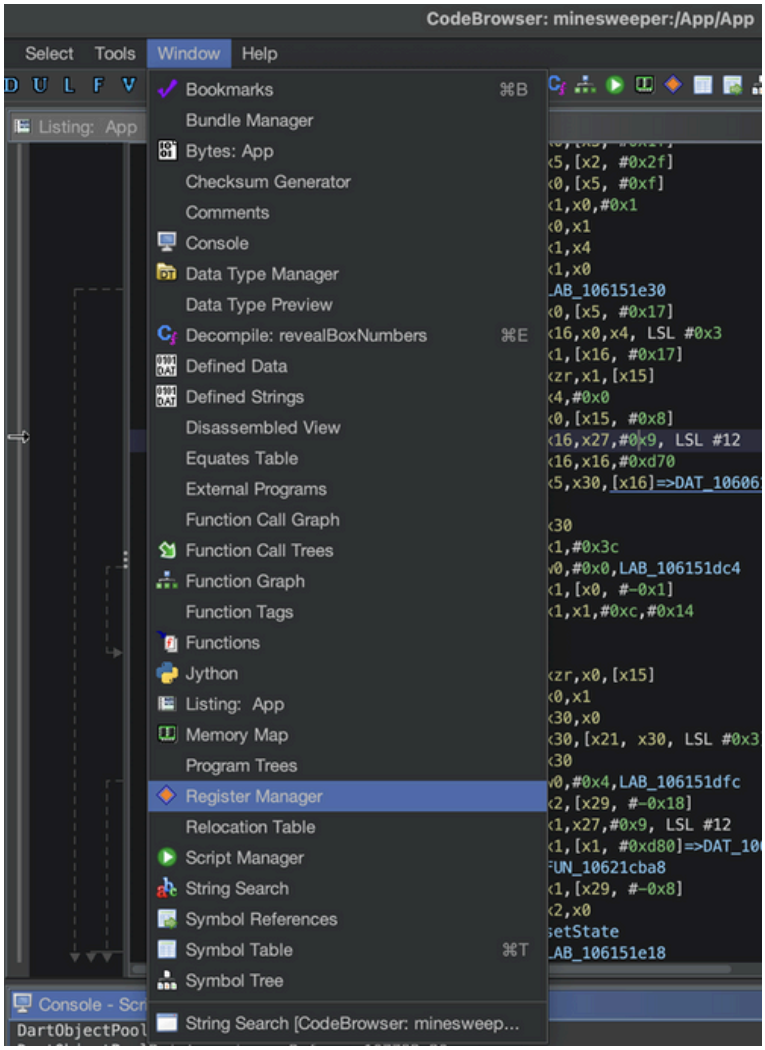


Figure 31b: Screenshot of Register Manager

Make sure that the Start, and the Value should be set to the address of the Object Pool:

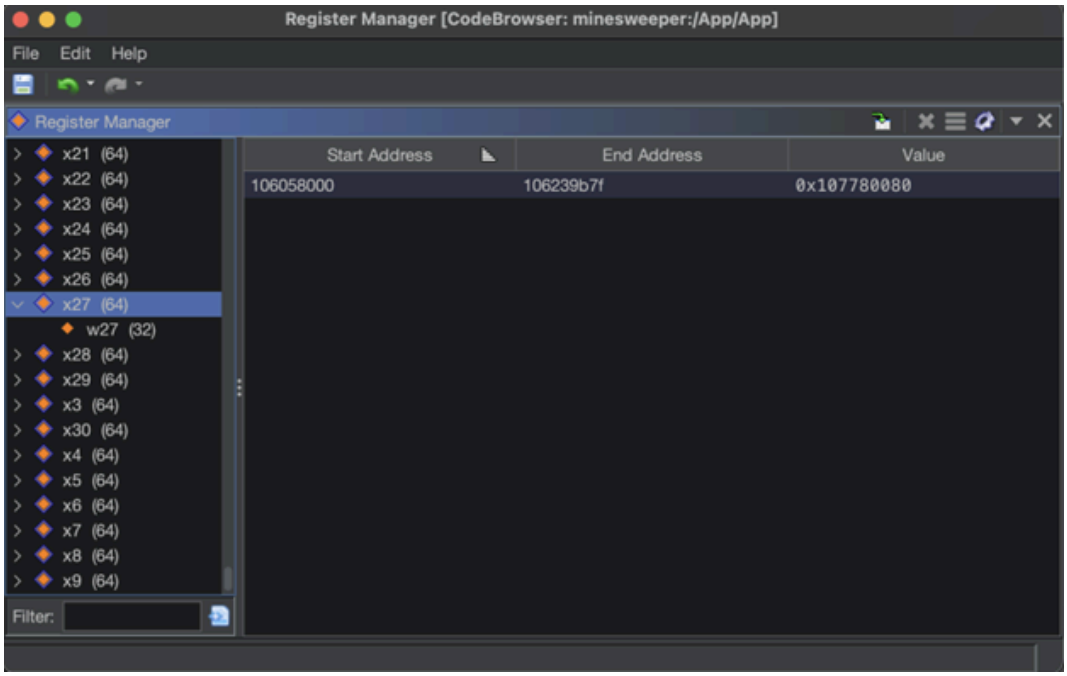


Figure 32: Screenshot of Register Manager Start and End Address

Now Auto Analysis needs to be done:

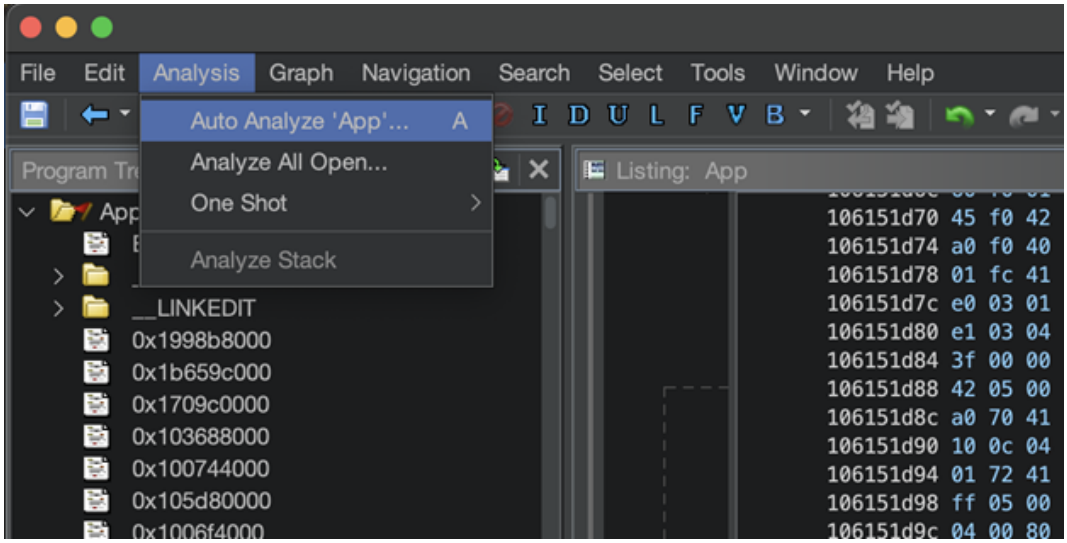


Figure 33: Screenshot of Auto Analysis

It can be observed that the references between code and data have been resolved:

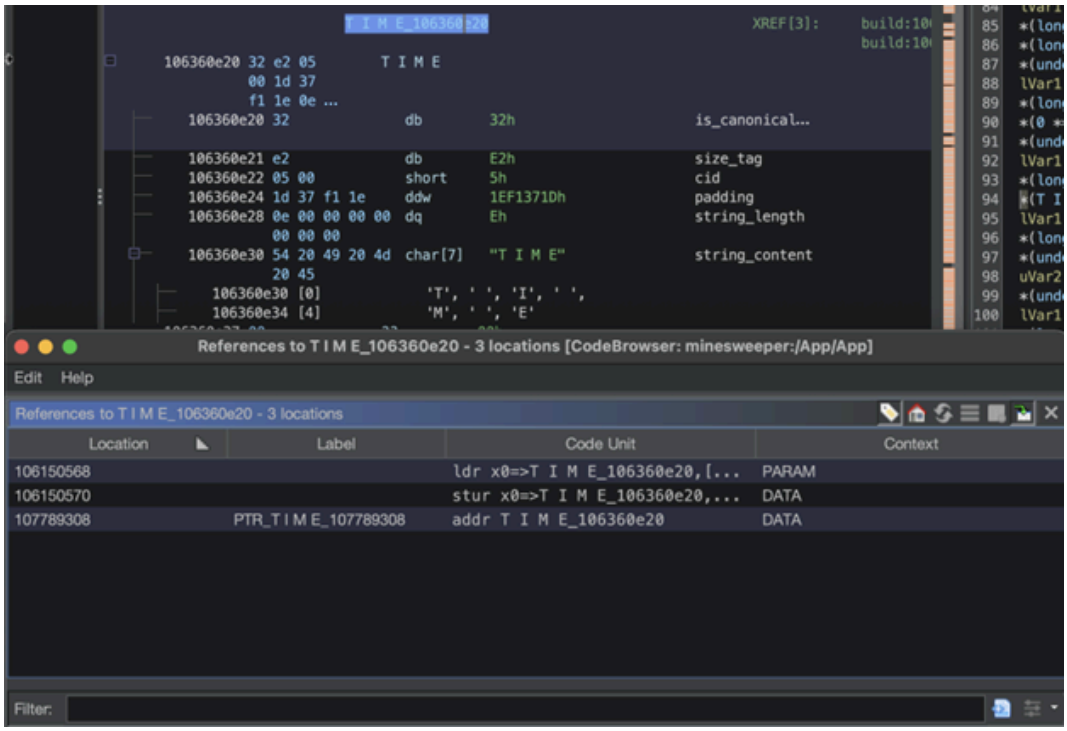


Figure 33b: Screenshot of Auto Analysis Result

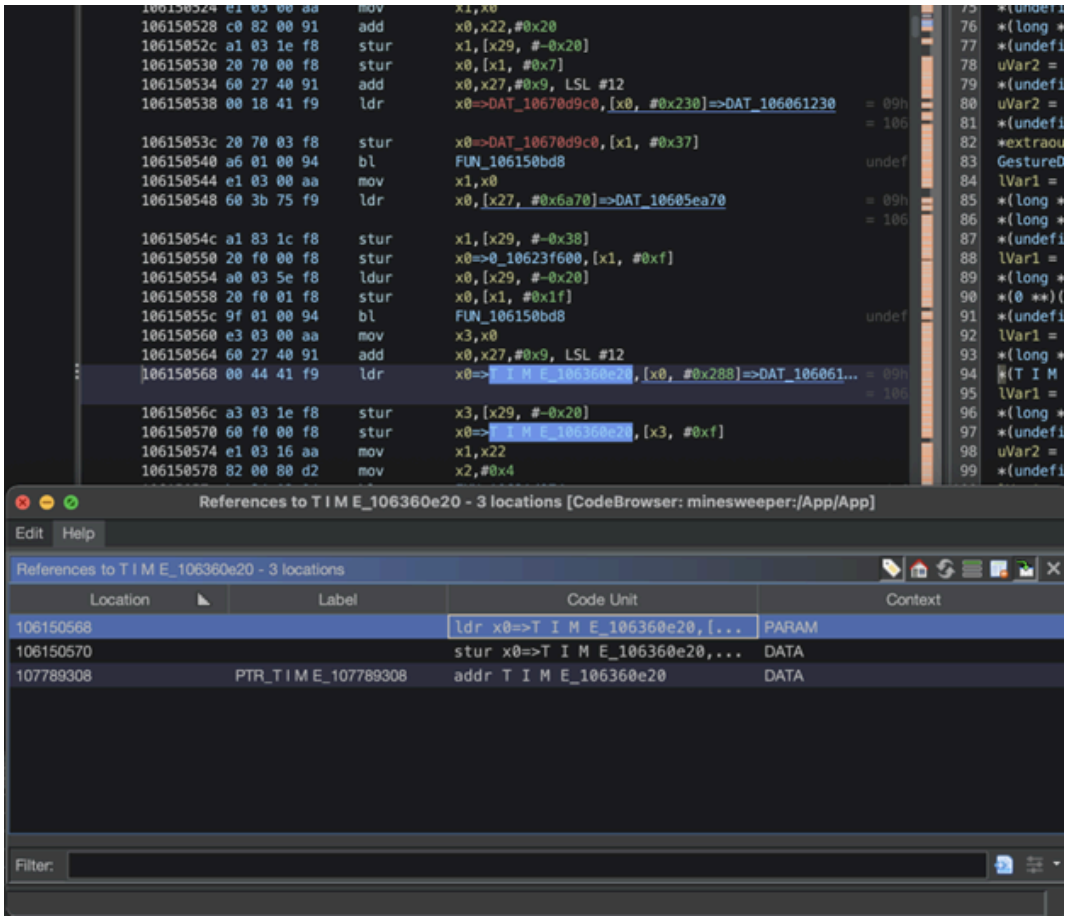


Figure 34: Screenshot of Auto Analysis Result Cont.

Handling Dart VM's Custom Stack

Since Dart VM uses X15 as the stack pointer, tools like Ghidra fail to identify local variables and parameters. To resolve this:

1. Replace X15 with SP across relevant code ranges.
2. Set X27 (object pool register) to a known value in Ghidra's memory map.
3. Perform auto-analysis to establish cross-references between code and data.

The script below replaces X15 register with SP register for a given range of addresses:

```
import ghidra.app.script.GhidraScript;
import ghidra.program.model.address.Address;
import ghidra.program.model.address.AddressSet;
import ghidra.program.model.address.AddressSetView;
import ghidra.program.model.listing.Instruction;
import ghidra.program.model.listing.InstructionIterator;
import ghidra.program.model.listing.Listing;
import ghidra.program.model.lang.Register;
import ghidra.app.plugin.assembler.Assembler;
import ghidra.app.plugin.assembler.Assemblers;

public class ReplaceX15WithSP extends GhidraScript {

  @Override
  public void run() {
     // Retrieve the SP register
     Register spRegister = currentProgram.getProgramContext().getRegister("sp");

     // Retrieve the x15 register
     Register x15Register = currentProgram.getProgramContext().getRegister("x15");

     // Define the address range
     AddressSet addressRange = new AddressSet(
        toAddr("0x106058000L"),
        toAddr("0x106239B7FL")
     );

     // Replace occurrences of x15 with SP in the defined address range
     replaceRegisterInRange(addressRange, spRegister, x15Register);
  }

  private void replaceRegisterInRange(AddressSetView addressRange, Register
spRegister, Register x15Register) {
     Listing listing = currentProgram.getListing();
     Assembler assembler = Assemblers.getAssembler(currentProgram);
     InstructionIterator instructions = listing.getInstructions(addressRange, true);
```

```
      // Iterate through instructions in the address range
while (instructions.hasNext() && !monitor.isCancelled()) {
        Instruction instruction = instructions.next();

        // Check each operand of the instruction for the x15 register
for (int operandIndex = 0; operandIndex < instruction.getNumOperands(); operandIndex++)
{
Object[] operands = instruction.getOpObjects(operandIndex);

// Replace x15 with SP if found
for (Object operand : operands) {
if (operand instanceof Register && operand.equals(x15Register)) {
                    String patchedInstruction = instruction.toString()
                       .replace(x15Register.getName(), spRegister.getName());
                    println("Patching: " + patchedInstruction);

                    // Get the instruction's address
                    Address instructionAddress = instruction.getAddress();

                    try {
                       // Use the assembler to write the patched instruction to memory
                       assembler.assemble(instructionAddress, patchedInstruction);
                    } catch (Exception e) {
                       println("Failed to patch instruction at address: " + instructionAddress);
                       e.printStackTrace();
                    }
}
}
        }
     }
  }
}
```

Before running the script:

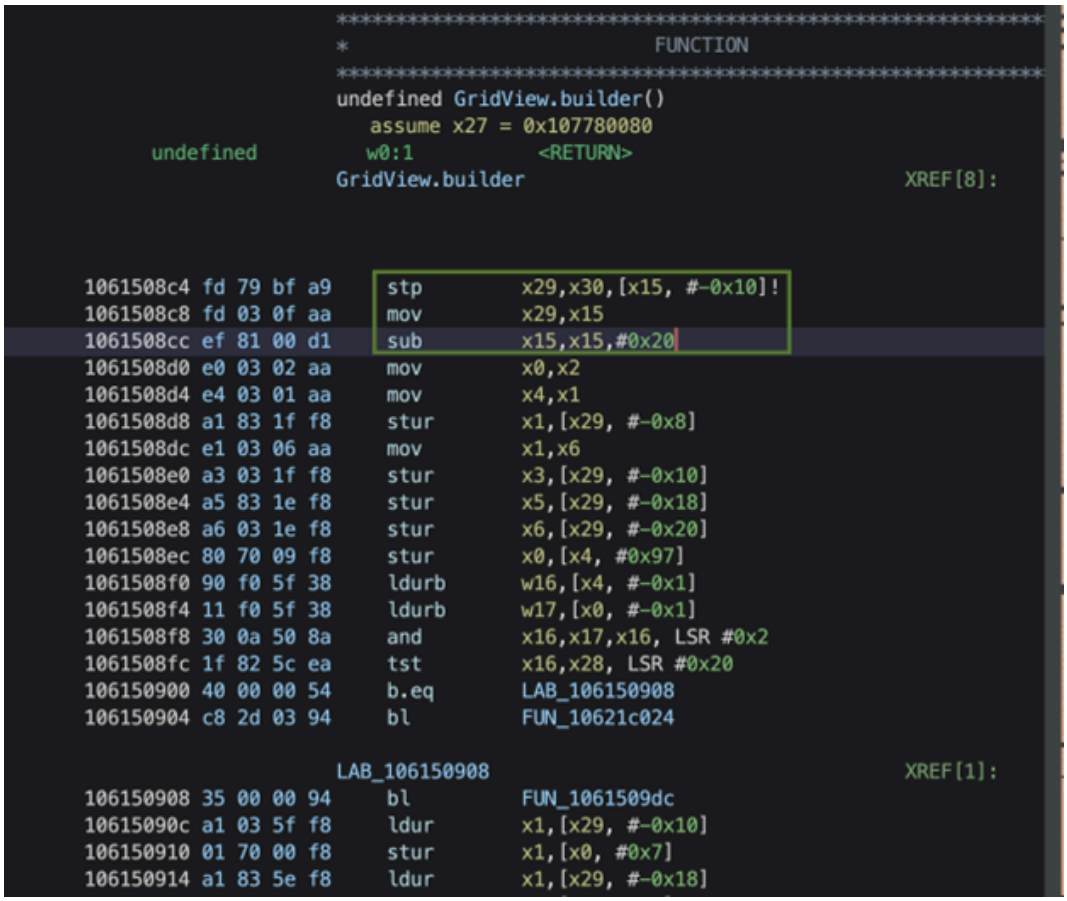


Figure 35: Screenshot of Disassembly Window before Stack handling script execution

After running the script:

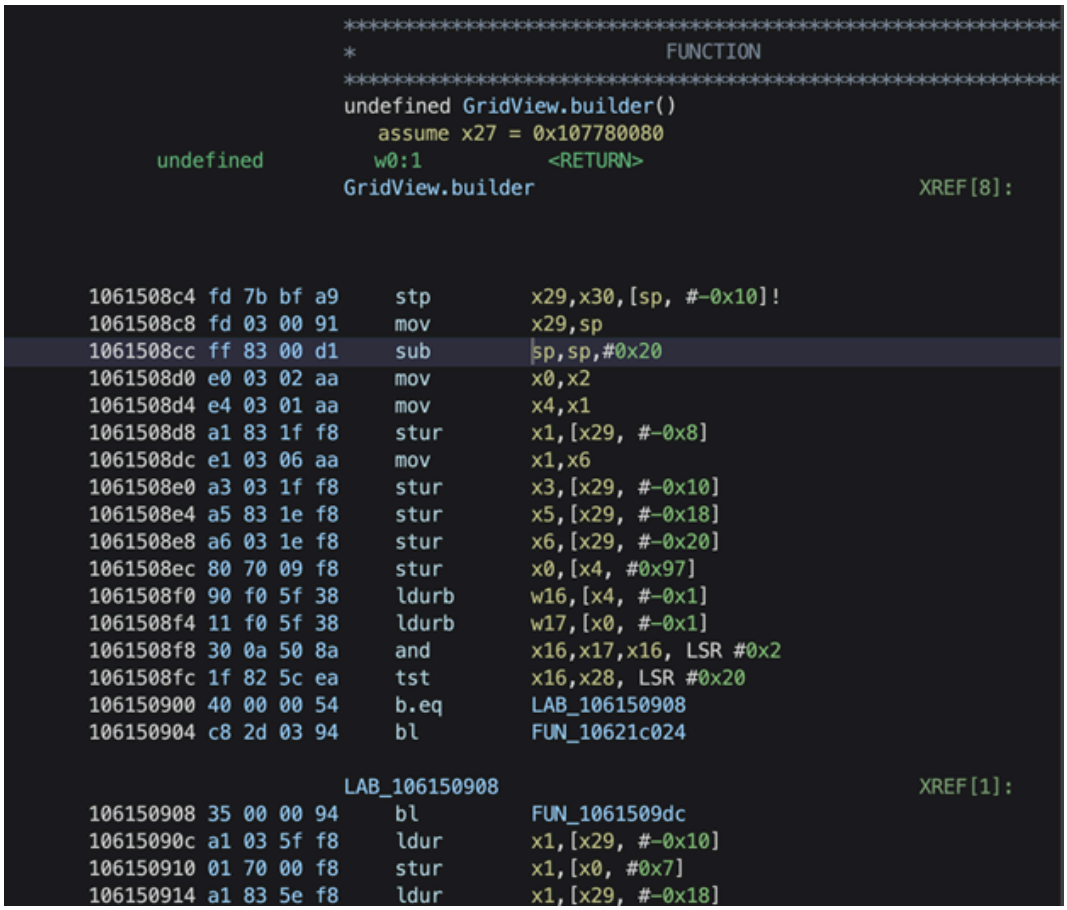


Figure 36: Screenshot of Disassembly Window After Stack handling script execution

After patching the stack pointer, the Dart stack is considered to be the regular stack by Ghidra. Next, let Ghidra know that the parameters it is looking for are in the stack, not in the registers that it would usually use.

Correcting Function Parameter Interpretation

Unlike ARM64's standard use of X0-X7 for parameters, Dart passes all parameters on the stack. Adjust function signatures in Ghidra manually to reflect this.

1. Navigate to the function signature editor.
2. Define parameters based on stack values.

This can be accomplished by using Edit Function Signature as shown below:

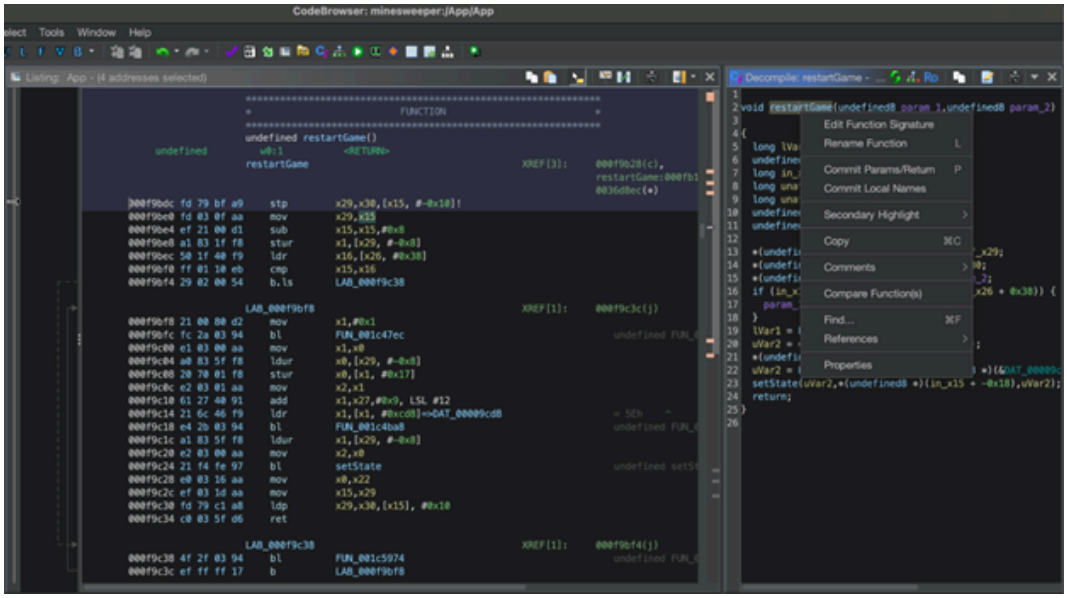


Figure 37: Screenshot of edit function signature in Ghidra

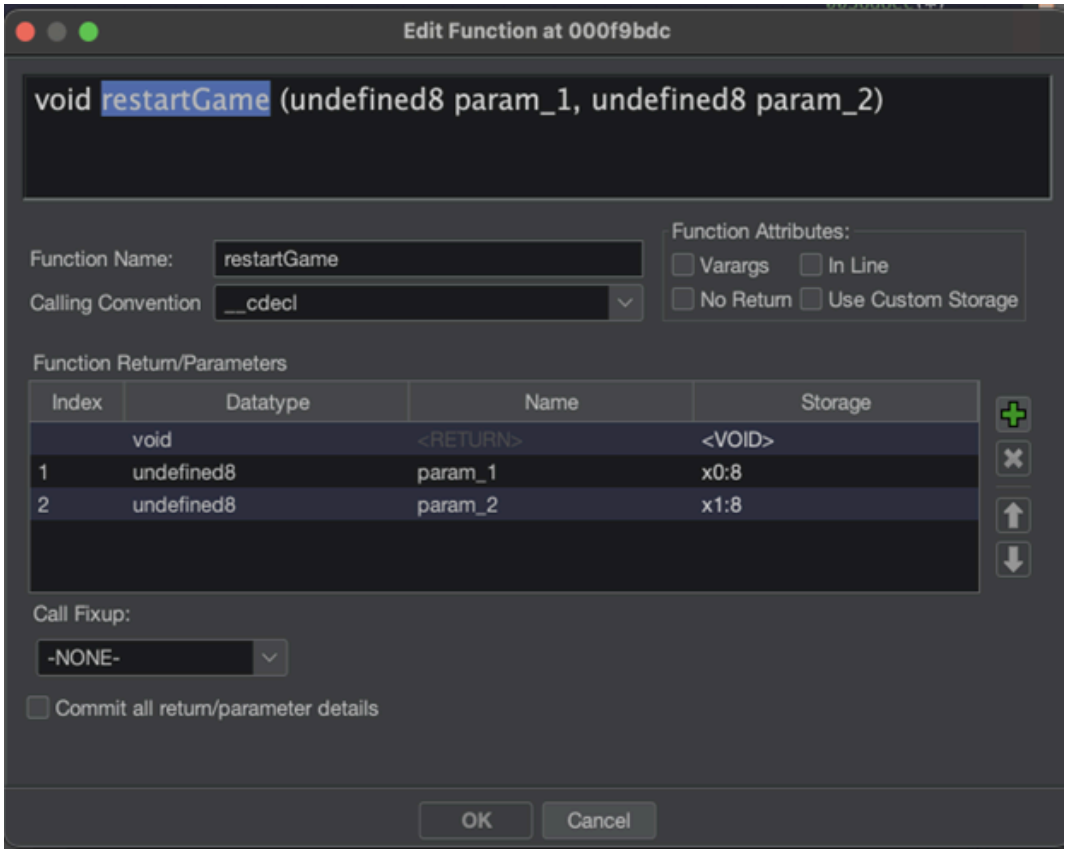


Figure 37b: Screenshot of edit function signature in Ghidra cont.

Conclusion

In this guide, we improved Ghidra's ability to decompile Flutter applications by:

- Establishing cross-references between Dart code and deserialized objects.
- Correcting the stack pointer usage for proper decompilation.
- Adjusting function signatures to align with Dart's parameter-passing model.

This methodology provides a strong foundation for reverse engineering Flutter apps on iOS, enabling deeper insights into their architecture and functionality.

SECURITY INNOVATION
A BUREAU VERITAS COMPANY

About Security Innovation/ Bureau Veritas

Security Innovation is a leader in software security, providing comprehensive assessment solutions to secure software from design to deployment, across all environments, including web, cloud, IoT, and mobile. Leveraging decades of expertise and as part of Bureau Veritas, a global leader in Testing, Inspection, and Certification, we seamlessly integrate world-class security into development processes, safeguarding the way companies build and deliver products.

Security Innovation is a Bureau Veritas company. Bureau Veritas (BV) is a publicly listed company specialized in testing, inspection and certification. BV was founded in 1828, has over 80,000 employees and is active in 140 countries.

References

- Flutter and Dart Source Code/Documentation
- reFlutter Framework
- Guardsquare Blog
- Guardsquare Blog 2
- Minesweeper Source YT
- Dart Blog - Numeric Computation

BUREAU VERITAS 1828

Interested?

Contact us today to start raising your cyber resilience.

sisales@securityinnovation.com

+1 877 839 7598

securityinnovation.com