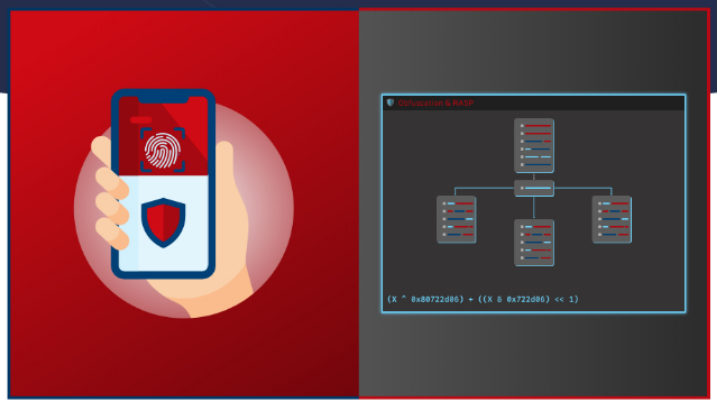
IOS REVERSE ENGINEERING

Part 1 – SingPass RASP Analysis

Romain Thomas August 29, 2022 31 min read



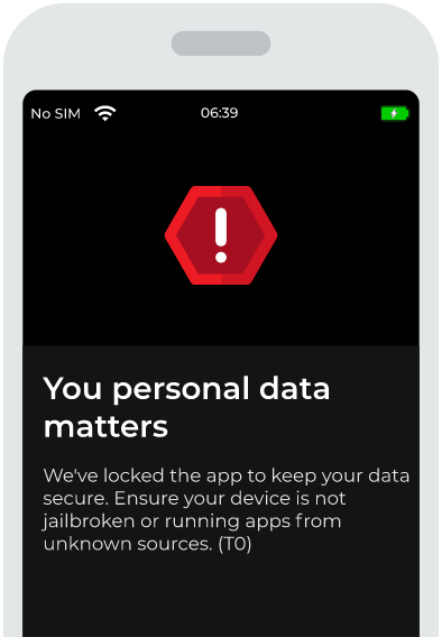
This is an **ephemeral** blog post. If you want to keep access to this content, you can download the archive which contains the blog post and the assets here: http://dl.romainthomas.fr or ios-rasp-blog-post.zip

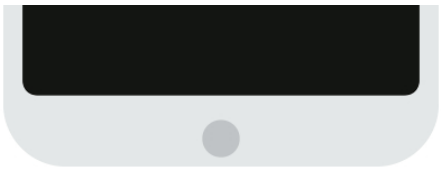
Introduction

I started to dig into the SingPass application which turned out to be obfuscated and protected with Runtime Application Self-Protection (RASP).

Retrospectively, this application is pretty interesting to analyze RASP functionalities since:

1. It embeds advanced RASP functionalities (Jailbreak detection, Frida Stalker detection, ...).
2. The native code is *lightly* obfuscated.
3. The application starts by showing an error message which is a good *oracle* to know whether we managed to circumvent the RASP detections.





Context

All the findings and the details of this blog post has been shared with the editor of the obfuscator. The overall results have also been shared with SingPass. In addition, SingPass is part of a bug bounty program on HackerOne. Bypassing these RASP checks are a prerequisite to go further in the security assessment of this application.

By grepping some keywords in the install directory of the application, we actually get two results which reveal the name of the obfuscator:

```
iPhone:/private/[ ... ]/SingPass.app root# grep -Ri +++++++ *
Binary file Frameworks/NuDetectSDK.framework/NuDetectSDK matches
Binary file SingPass matches
```

The NuDetectSDK binary also uses the same obfuscator but it does not seem involved in the early jailbreak detection shown in the previous figure. On the other hand, `SingPass` is the main binary of the application and we can observe strings related to threat detections:

```
$ SingPass.app strings ./SingPass|grep -i +++++++
+++++++ThreatLogAPI(headers:)
+++++++CallbackHandler(context:)
```

Binary

For those who would like to follow this blog post with the original binary, you can download the decrypted SingPass Mach-O binary here . The name of the obfuscator has been redacted but it does not impact the content of the code.

Unfortunately, the binary does not leak other strings that could help to identify where and how the application detects jailbroken devices but fortunately, the application does not crash ...

If we assume that the obfuscator decrypts strings at runtime, we can try to dump the content of the `__data` section when the error message is displayed. At this point of the execution, the strings used for detecting jailbroken devices are likely decoded and clearly present in the memory.

This is actually quite the same technique used in **PokemonGO**: *What About LIEF*

1. We run the application and we wait for the *jailbreak* message
2. We attach to SingPass with Frida and we inject a library that:
   - Parses in-memory the `SingPass` binary (thanks to LIEF)
   - Dumps the content of the `__data` section
   - Write the dump in the iPhone's `/tmp` directory

Once the data section is dumped, we end up with the following changes in some parts of the `__data` section:

```
__data:00000001010B97FA qword_1010B97FA DCQ 0x942752767D91608E
__data:00000001010B97FA
__data:00000001010B9802 byte_1010B9802  DCB 0x98
__data:00000001010B9802
__data:00000001010B9803 dword_1010B9803 DCD 0x2318E9A2
__data:00000001010B9803
__data:00000001010B9807                 DCQ 0x222AD8A21325DEE5
__data:00000001010B980F byte_1010B980F  DCB 0xE0
__data:00000001010B980F
__data:00000001010B9810 byte_1010B9810  DCB 0xE5
__data:00000001010B9810
__data:00000001010B9811 byte_1010B9811  DCB 0xB7
__data:00000001010B9811
__data:00000001010B9812 dword_1010B9812 DCD 0x8491650B
__data:00000001010B9812
```

```
__data:00000001010B9816                 DCQ 0x4B7D0892EFAB11BBF
__data:00000001010B981E                 DCQ 0x4B9FF985643894C5
__data:00000001010B9826                 DCQ 0x38D0ABE65C497CF0
__data:00000001010B9826
__data:00000001010B982E                 DCQ 0xD769CDDBCB49C500
__data:00000001010B9836                 DCQ 0xF1C4BCB1A563CD53
__data:00000001010B983E                 DCD 0x52BB5CBC
__data:00000001010B9842 byte_1010B9842  DCB 0x42
__data:00000001010B9842
__data:00000001010B9843 dword_1010B9843 DCD 0x2451DD96
__data:00000001010B9843
__data:00000001010B9847                 DCQ 0x6248099506559926
__data:00000001010B984F                 DCD 0xFAF2AF09
__data:00000001010B9853 dword_1010B9853 DCD 0xD8B63318
__data:00000001010B9853
__data:00000001010B9857                 DCQ 0xD1CA2820CCC72C5F
__data:00000001010B985F                 DCQ 0xE7CE3B62D1C62F5E
__data:00000001010B9867                 DCD 0x77CE475E
__data:00000001010B986B dword_1010B986B DCD 0x7AB7291C
__data:00000001010B986B
__data:00000001010B986F                 DCQ 0x75C9251C6AC41E5F
__data:00000001010B9877                 DCQ 0x71CF165D64C42C4E
__data:00000001010B987F                 DCQ 0x6EC22E5133BA165C
__data:00000001010B9887 word_1010B9887  DCW 0xB54F
__data:00000001010B9887
__data:00000001010B9889 dword_1010B9889 DCD 0xDEDB09A0
__data:00000001010B9889
```

```
__data:00000001010B97FA aTaurine        DCB "/taurine",0
__data:00000001010B97FA
__data:00000001010B9803 aTaurineCstmp   DCB "/taurine/cstmp",0
__data:00000001010B9803
__data:00000001010B9812 aTaurineJailbre DCB "/taurine/jailbreakd",0
__data:00000001010B9812
__data:00000001010B9812
__data:00000001010B9826                 DCD 0
__data:00000001010B982A aTaurineLaunchj DCB "/taurine/launchjailbreak",0
__data:00000001010B982A
__data:00000001010B982A
__data:00000001010B9843 aTaurineJbexec  DCB "/taurine/jbexec",0
__data:00000001010B9843
__data:00000001010B9853 aTaurineAmfideb DCB "/taurine/amfidebilitate",0
__data:00000001010B9853
__data:00000001010B9853
__data:00000001010B986B aTaurinePspawnP DCB "/taurine/pspawn_payload.dylib",0
__data:00000001010B986B
__data:00000001010B986B
__data:00000001010B9889 aInstalledTauri DCB "/.installed_taurine",0
__data:00000001010B9889
__data:00000001010B9889
```

Fig 1. Slices of the __data section before and after the dump

> **Note**
> The string encoding routines will be analyzed in the **second** part of this series of blog posts

In addition, we can observe the following strings which seem to be related to the RASP functionalities of the obfuscator:

```
__data:1010B8D10 aIxgCodeTracing DCB "EVT_CODE_TRACING",0          ; XREF: on_rasp_detection
__data:1010B8D30 aIxgCodeSystemL DCB "EVT_CODE_SYSTEM_LIB",0       ; XREF: on_rasp_detection
__data:1010B8D50 aIxgCodeSymbolT DCB "EVT_CODE_SYMBOL_TABLE",0     ; XREF: on_rasp_detection
__data:1010B8D70 aIxgCodePrologu DCB "EVT_CODE_PROLOGUE",0         ; XREF: on_rasp_detection
__data:1010B8D90 aIxgAppLoadedLi DCB "EVT_APP_LOADED_LIBRARIES",0 ; XREF: on_rasp_detection
__data:1010B8DB0 aIxgAppSignatur DCB "EVT_APP_SIGNATURE",0         ; XREF: on_rasp_detection
__data:1010B8DD0 aIxgEnvDebugger DCB "EVT_ENV_DEBUGGER",0          ; XREF: on_rasp_detection
__data:1010B8DF0 aIxgEnvJailbrea DCB "EVT_ENV_JAILBREAK",0         ; XREF: on_rasp_detection
__data:1010B8E10 aUsersChinweeDe DCB "/Users/***/ndi-sp-mobile-ios-swift/SingPass/*******.swift",0
__data:1010B8E10                                                   ; XREF: on_rasp_detection
```

Fig 2. Strings Related to the RASP Features

All the `EVT_*` strings are referenced by one **and only one** function that I named `on_rasp_detection`. This function turns out to be the threat detection callback used by the app's developers to perform action(s) when a RASP event is triggered.

To better understand the logic of the checks behind these strings, let's start with `EVT_CODE_PROLOGUE` which is used to detect hooked functions.

EVT_CODE_PROLOGUE: Hook Detection

While going through the assembly code closes to the cross-references of `on_rasp_detection`, we can spot several times this pattern:

To detect if a given function is hooked, the obfuscator loads the **first byte** of the function and compares this byte with the value `0×FF`. `0×FF` might seem – at first glance – arbitrary but it's not. Actually, regular functions start with a prologue that allocates space on stack for saving registers defined by the calling convention and stack variables required by the function. In AArch64, this allocation can be performed in two ways:

```
stp REG, REG, [SP, 0×AA]!
; or
sub SP, SP, 0×BB
stp REG, REG, [SP, 0×CC]
```

These instructions are **not equivalent**, but somehow and with the good offsets, they could lead to the same result. In the second case, the instruction `sub SP, SP, #CST` is encoded with the following bytes:

```
</>  0×ff ** 0×00 0×d1
```

As we can see, the encoding of this instruction starts with `0×FF`. If it is not the case, then either the function starts with a different stack-allocation prologue or potentially starts with a hooking trampoline. Since the code of the application is compiled through obfuscator's compiler, the compiler is able to distinguish these two cases and insert the right check for the correct function's prologue.

If the first byte of the instruction of the function does not pass the check, it jumps to the red basic block. The purpose of this basic block is to trigger a user-defined callback that will process the detection according to the application's design and the developers' choices:

- Printing an error

- Crashing the application
- Corrupting internal data
- ...

From the previous figure, we can observe that the detection callback is loaded from **a static variable** located at `#hook_detect_cbk_ptr`. When calling this detection callback, the obfuscator provides the following information to the callback:

1. A detection code: `0x400` for `EVT_CODE_PROLOGUE`
2. A *corrupted* pointer which could be used to crash the application.

Let's now take a closer look at the design of the detection callback(s) as a whole.

Detection Callbacks

As explained in the previous section, when the obfuscator detects tampering, it reacts by calling a detection callback stored in the **static variable** at the address: `0x10109D760`

```
__data:000000010109D758 off_10109D758       DCQ sub_100ED9F00
__data:000000010109D760 hook_detect_cbk_ptr DCQ hook_detect_cbk ; Hook Detection Callback
__data:000000010109D768 word_10109D768      DCW 0xDBE3
__data:000000010109D76A byte_10109D76A      DCB 1
__data:000000010109D76B byte_10109D76B      DCB 1
```

By statically analyzing `hook_detect_cbk`, the implementation seems to corrupt the pointer provided in the callback's parameters. On the other hand, when running the application we observe a jailbreak detection message and not a crash of the application.

If we look at the cross-references which read or write at this address, we get this list of instructions:

```
...
R init_and_check_rasp+1D8C   LDR   X8,  [X20,#hook_detect_cbk_ptr@PAGEOFF]
R init_and_check_rasp+1DE4   LDR   X8,  [X20,#hook_detect_cbk_ptr@PAGEOFF]
R init_and_check_rasp+1E3C   LDR   X8,  [X20,#hook_detect_cbk_ptr@PAGEOFF]
R init_and_check_rasp+1E94   LDR   X8,  [X20,#hook_detect_cbk_ptr@PAGEOFF]
R init_and_check_rasp+1EEC   LDR   X8,  [X20,#hook_detect_cbk_ptr@PAGEOFF]
R init_and_check_rasp+1F44   LDR   X8,  [X20,#hook_detect_cbk_ptr@PAGEOFF]

W init_and_check_rasp+01BC   STR   X23, [X20,#hook_detect_cbk_ptr@PAGEOFF]
```

So actually **only one** instruction – `init_and_check_rasp+01BC` – is **overwriting** the *default* detection callback with another function:

```
__text:0000000100ED7E4C ADRP            X8, #sub_100206C68@PAGE
__text:0000000100ED7E50 LDRB            W8, [X8,#sub_100206C68@PAGEOFF]
__text:0000000100ED7E54 ADRL            X23, hook_detect_cbk_user_def
__text:0000000100ED7E5C STR             X23, [X20,#hook_detect_cbk_ptr@PAGEOFF]
__text:0000000100ED7E60 CMP             W8, #0xFF
__text:0000000100ED7E64 B.EQ            loc_100ED7EB0
```

Compared to the default callback: `hook_detect_cbk`, the overridden function, `hook_detect_cbk_user_def` does not corrupt a pointer that would make the application crash. Instead, it calls the function `on_rasp_detection` which references all the strings `EVT_CODE_TRACING`, `EVT_CODE_SYSTEM_LIB`, etc, listed in the figure 2.

> `hook_detect_cbk_user_def` is called on a RASP event. That's why this application does not crash.

By looking at the function `init_and_check_rasp` as a whole, we can notice that the `X23` register is also used to initialize other static variables:

```
W 0x00100ED7E5C: STR   X23, [X20, #hook_detect_cbk_ptr@PAGEOFF]
```

```
W 0x00100ED81F0: STR   X23, [X25, #EVT_CODE_TRACING_cbk_ptr@PAGEOFF]
W 0x00100ED86A0: STR   X23, [X25, #EVT_CODE_SYMBOL_TABLE_cbk_ptr@PAGEOFF]
W 0x00100ED8B48: STR   X23, [X25, #EVT_CODE_SYSTEM_LIB_cbk_ptr@PAGEOFF]
W 0x00100ED8C64: STR   X23, [X24, #EVT_ENV_JAILBREAK_cbk_ptr@PAGEOFF]
W 0x00100ED8E40: STR   X23, [X24, #EVT_APP_SIGNATURE_cbk_ptr@PAGEOFF]
W 0x00100ED91D4: STR   X23, [X24, #EVT_ENV_DEBUGGER_cbk_ptr@PAGEOFF]
W 0x00100ED9408: STR   X23, [X24, #EVT_APP_LOADED_LIBRARIES_cbk_ptr@PAGEOFF]
W 0x00100ED9694: STR   X23, [X24, #EVT_APP_MACHO_cbk_ptr@PAGEOFF]
```

Fig 3. X23 Writes Instructions

These memory writes mean that the callback `hook_detect_cbk_user_def` is used to initialize other static variables. In particular, these other static variables are likely used for the other RASP checks. By looking at the cross-references of these static variables `#EVT_CODE_TRACING_cbk_ptr`, `#EVT_ENV_JAILBREAK_cbk_ptr` etc, we can locate where the other RASP checks are performed and under which conditions they are triggered.

EVT_CODE_SYSTEM_LIB

EVT_ENV_DEBUGGER

Thanks to the `#EVT_*` cross-references, we can go **statically** through all the basic blocks that use these `#EVT_*` variables and highlight the underlying checks that could trigger the RASP callback(s). Before detailing the checks, it is worth mentioning the following points:

1. Whilst the application uses a commercial obfuscator which provides native code obfuscation in addition to RASP, the code is **lightly obfuscated** which makes static assembly code analysis doable very easily.
2. As it will be discussed in "*RASP Weaknesses*", the application setups the **same callback** for **all** the RASP events. Thus, it eases the RASP bypass and the dynamic analysis of the application.

Anti-Debug

The version of the obfuscator used by SingPass implements two kinds of debug check. First, it checks if the parent process id (`ppid`) is the same as `/sbin/launchd` which should be **1**.

```
static constexpr pid_t LAUNCHD_PID = 1;
pid_t ppid = getppid();
if (ppid ≠ LAUNCHD_PID) {
  // Trigger EVT_ENV_DEBUGGER
}
```

`getppid` is called either through a function or with a syscall.

If it is not the case, it triggers the `EVT_ENV_DEBUGGER` event. The second check is based on `sysctl` which is used to access the `extern_proc.p_flag` value. If this flag contains the `P_TRACED` value, the RASP routine triggers the `EVT_ENV_DEBUGGER` event.

```
int names[] = {
  CTL_KERN,
  KERN_PROC,
  KERN_PROC_PID,
  getpid(),
```

```
};
kinfo_proc info;
int sizeof_info = sizeof(kinfo_proc);
int ret = sysctl(names, 4, &info, &sizeof_info, nullptr, nullptr);
if (info.kp_proc.p_flag  & P_TRACED) {
  // Trigger EVT_ENV_DEBUGGER
}
```

In the SingPass binary, we can find an instance of these two checks in the following ranges of addresses:

```
ppid:   0x10071F420 – 0x10071F474
sysctl: 0x100151668 – 0x100151730
```

Jailbreak Detection

As for most of the jailbreak detections, the obfuscator tries to detect if the device is jailbroken by checking if some files exist (or not) on the device.

Files or directories are checked with syscalls or a regular functions thanks to the following *helpers*:

```
pathconf:    0x100008EB0 -- 0x100008F28
utimes:      0x10000D8D4 -- 0x10000D948
stat:        0x100012188 -- 0x10001221C
open:        0x10002D478 -- 0x10002D4D8
fopen:       0x1000474E4 -- 0x100047554
stat64:      0x10006AA30 -- 0x10006AAD8
getfsstat64: 0x10047E82C -- 0x10047E914
```

While in the introduction, I mentioned that a dump of the section `__data` reveals strings related to jailbreak detection, the dump does not reveal **all the strings** used by the obfuscator.

By looking closely at the strings encoding mechanism, it turns out that some strings are decoded *just-in-time* in a *temporary variable*. I'll explain the strings encoding mechanism in the second part of this series of blog posts but at this point, we can uncover the strings by setting hooks on functions like `fopen`, `utimes` and dumping the `__data` section right after these calls. Then, we can iterate over the different dumps to see if new strings appear.

```
$ python dump_analysis.py

Processing __data_0.raw
0x01010b935c h/.installed_unc0ver
0x01010b986a w/taurine/pspawn_payload.dylib

Processing __data_392.raw
0x01010b910e y__TEXT
0x01010b91b3 /System/Library/dyld/dyld_shared_cache_arm64e
0x01010b9174 /System/Library/Caches/com.apple.dyld/dyld_shared_cache_arm64e
0x01010b9136 /System/Library/Caches/com.apple.dyld/dyld_shared_cache_arm64
0x01010b9126 dyld_v1  arm64e
0x01010b9116 dyld_v1   arm64

Processing __data_393.raw
0x01010afb90 /Users/xxxxxxx/Desktop/Xcode/ndi-sp-mobile-ios-swift/SingPass/[ ... ]
0x01010b942c /var/jb
0x01010af910 https://bio-stream.singpass.gov.sg
0x01010af6a0 https://api.myinfo.gov.sg/spm/v3
0x01010b93b0 /.mount_rw
```

In the end, the approach does not enable to have all the strings decoded but it enables to have a *good coverage*. The list of the files used for detecting jailbreak is given the Annexes.

There is also a particular check for detecting the `unc0ver` jailbreak which consists in trying to unmount `/.installed_unc0ver`:

```
0x100E4D814: _unmount("/.installed_unc0ver")
```

Environment

The obfuscator also checks environment variables that trigger the `EVT_ENV_JAILBREAK` event.

Some of these checks seem to be related to code lifting detection while still triggering the `EVT_ENV_JAILBREAK` event.

```
if (strncmp(_dyld_get_image_name(0), "/private/var/folders", 0×14)) {
  → Trigger EVT_ENV_JAILBREAK
}
```

```
if (strncmp(getenv("HOME"), "/Users", 6) == 0) {
  → Trigger EVT_ENV_JAILBREAK
}
```

```
if (strncmp(getenv("HOME"), "mobile", 6) ≠ 0) {
  → Trigger EVT_ENV_JAILBREAK
}
```

```
char buffer[0×400];
size_t buff_size = 0×400;
_NSGetExecutablePath(buffer, &buff_size);
if (buffer.startswith("/private/var/folders")) {
  → Trigger EVT_ENV_JAILBREAK
}
```

startswith()

From a reverse engineering perspective, startswith() is actually implemented as a succession of xor that are "or-ed" to get a boolean. This might be the result of an optimization from the compiler. You can observe this patter in the basic block located at the address: 0x100015684.

Advanced Detections

In addition to regular checks, the obfuscator performs advanced checks like verifying the current status of the SIP (**S**ystem **I**ntegrity **P**rotection), and more precisely, the KEXTS code signing status.

From my weak experience in iOS jailbreaking, I think that no Jailbreak disables the `CSR_ALLOW_UNTRUSTED_KEXTS` flag. Instead, I guess that it is used to detect if the application is running on an Apple M1 which allows such deactivation.

```
csr_config_t buffer = 0;
if (__csrctl(CSR_ALLOW_UNTRUSTED_KEXTS, buffer, sizeof(csr_config_t)) {
  /*
   * SIP is disabled with CSR_ALLOW_UNTRUSTED_KEXTS
   * → Trigger EVT_ENV_JAILBREAK
   */
}
```

☰ Assembly range: 0×100004640 – 0×1000046B8

The obfuscator also uses the Sandbox API to verify if some paths exist:

```
int ret = __mac_syscall("Sandbox", /* Sandbox Check */ 2,
                        getpid(), "file-test-existence", SANDBOX_FILTER_PATH,
                        "/opt/homebrew/bin/brew");
```

The paths checked through this API are OSX-related directories, so I guess it is also used to verify that the current code has not been lifted on an Apple Silicon. Here is, for instance, a list of directories checked with the *Sandbox* API:

```
/Applications/Xcode.app/Contents/MacOS/Xcode
/System/iOSSupport/
/opt/homebrew/bin/brew
/usr/local/bin/brew
```

☰ Assembly range: 0×100ED7684 (function)

In addition, it uses the Sandbox attribute `file-read-metadata` as an alternative to the `stat()` function.

☰ Assembly range: 0×1000ECA5C – 0×1000ECE54

The application uses the sandbox API through private syscalls to determine whether some jailbreak artifacts exists. This is very smart but I guess it's not really compliant with the Apple

policy.

Code Symbol Table

The purpose of this check is to verify that the addresses of the resolved imports point to the right library. In other words, this check verifies that the import table is not tampered with pointers that could be used to hook imported functions.

```
☰ Initialization: part of sub_100E544E8
```

```
☰ Assembly range: 0×100016FC4 – 0×100017024
```

During the RASP checks initialization (`sub_100E544E8`), the obfuscator **manually** resolves the imported functions. This manual resolution is performed by iterating over the symbols in the `SingPass` binary, checking the library that imports the symbol, accessing (in-memory) the `__LINKEDIT` segment of this library, parsing the exports trie, etc. This manual resolution fills a table that contains the **absolute address** of the resolved symbols.

In addition, the initialization routine setups – what I called – a metadata structure that follows this layout:

```
__data:000000010109F0C8 nb_symbols      DCD 0×399
__data:000000010109F0C8
__data:000000010109F0D8                 ALIGN 0×20
__data:000000010109F0E0 ; symbols_metadata_t metadata
__data:000000010109F0E0 metadata        DCQ symbols_index       ; symbols_index
__data:000000010109F0E0                 DCQ origins             ; origins
__data:000000010109F0E0                 DCQ 0                   ; resolved_la_syms
__data:000000010109F0E0                 DCQ 0                   ; resolved_got_syms
__data:000000010109F0E0                 DCQ 0                   ; macho_la_syms
__data:000000010109F0E0                 DCQ 0                   ; macho_got_syms
__data:000000010109F0E0                 DCQ 0                   ; stub_helper_start
__data:000000010109F0E0                 DCQ 0                   ; stub_helper_end
__data:000000010109F0E0                 DCQ 0                   ; field_unknown
```

`symbols_index` is a kind of translation table that converts an index known by the obfuscator into an index in the `__got` or the `__la_symbol_ptr` section. The index's origin (i.e `__got` or `__la_symbol_ptr`) is determined by the `origins` table which contains enum-like integers:

```
enum SYM_ORIGINS : uint8_t {
  NONE      = 0,
  LA_SYMBOL = 1,
  GOT       = 2,
};
```

The length of both tables: `symbols_index` and `origins`, is defined by the static variable `nb_symbols` which is set to `0×399`. The metadata structure is followed by two pointers: `resolved_la_syms` and `resolved_got_syms` which point to the imports address table manually filled by the obfuscator.

```
ℹ There is a dedicated table for each section: __got and __la_symbol_ptr.
```

Then, `macho_la_syms` points to the beginning of the `__la_symbol_ptr` section while `macho_got_syms` points to the `__got` section.

Finally, `stub_helper_start / stub_helper_end` holds the memory range of the `__stub_helper` section. I'll describe the purpose of these values later.

All the values of this metadata structure are set during the initialization which takes place in the function `sub_100E544E8`.

In different places of the `SingPass` binary, the obfuscator uses this metadata information to verify the integrity of the resolved import(s). It starts by accessing the `symbols_index` and the `origins` with a fixed value:

```
__text:00100016FC4 LDR              W26, [X22,#0×CA8] ; X22 → symbols_index
```

```
__text:00100016FC8 LDR             X8, [X19,#0×498]
__text:00100016FCC STP             XZR, XZR, [X19,#0×20]
__text:00100016FD0 STP             X22, X23, [X19,#0×58]
__text:00100016FD4 LDP             Q0, Q1, [X19,#0×30]
__text:00100016FD8 STUR            Q0, [X19,#0×68]

__text:00100016F54 LDR             X25, [X19,#0×488]
__text:00100016F58 LDR             X24, [X19,#0×490]
__text:00100016F5C LDRB            W21, [X23,#0×32A] ; X23 → origins table
__text:00100016F60 ADRL            X0, check_region_cbk ; cbk
__text:00100016F68 BL              iterate_system_region
```

Since the `symbols_index` table contains `uint32_t` values, `#0×CA8` matches `#0×32A` (index for the origins table) when divided by `sizeof(uint32_t)`: `0×CA8 = 0×32A * sizeof(uint32_t)`

In other words, we have the following operations:

```
const uint32_t sym_idx   = metadata.symbols_index[0×32a];
const SYM_ORIGINS origin = metadata.origins[0×32a]
```

Then, given the `sym_idx` value and depending on the origin of the symbol, the function accesses either the resolved `__got` table or the resolved `__la_symbol_ptr` table. This access is done with a helper function located at `sub_100ED6CC0`. It can be summed up with the following pseudo-code:

```
uintptr_t* section_ptr       = nullptr;
uintptr_t* manually_resolved = nullptr;

if      (origin == /* 1 */ SYM_ORIGINS::LA_SYMBOL) {
  section_ptr       = metadata.macho_la_syms;
  manually_resolved = metadata.resolved_la_syms;
}
else if (origin == /* 2 */ SYM_ORIGINS::GOT) {
  section_ptr       = metadata.macho_got_syms;
  manually_resolved = metadata.resolved_got_syms;
}
```

The entries at the index `sym_idx` of `section_ptr` and `manually_resolved` are compared and if they don't match, the event `#EVT_CODE_SYMBOL_TABLE` is triggered.

Actually, the comparison covers different cases. First, the obfuscator handles the case where the symbol at `sym_idx` is not yet resolved. In that case, `section_ptr[sym_idx]` points to the symbols resolution stub located in the section `__stub_helper`. That's why the `metadata` structure contains the memory range of this section:

```
const uintptr_t addr_from_section = section_ptr[sym_idx];
if (metadata.stub_helper_start ≤ addr && addr < metadata.stub_helper_end) {
  // Skip
}
```

In addition, if the pointers do not match, the function verifies their location using `dladdr`:

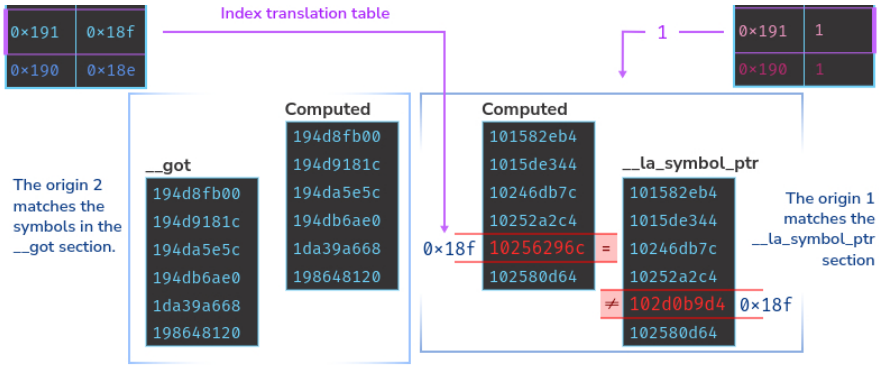
```
const uintptr_t addr_from_section    = section_ptr[sym_idx];
const uintptr_t addr_from_resolution = manually_resolved[sym_idx];

if (addr_from_section ≠ addr_from_resolution) {
  Dl_info info_section;
  Dl_info info_resolution;

  dl_info(addr_from_section,    &info_section);
  dl_info(addr_from_resolution, &info_resolution);
  if (info_section.dli_fbase ≠ info_resolution.dli_fbase) {
    // ⟶ Trigger EVT_CODE_SYMBOL_TABLE;
  }
}
```

Two pointers might not match if, for instance, an imported function is hooked with Frida.

In the case where the `origin[sym_idx]` is set to `SYM_ORIGINS::NONE` the function skips the check. Thus, we can simply disable this RASP check by filling the original table with 0. The number of symbols is close to the metadata structure and the address of the metadata structure is leaked by the `___atomic_load` and `___atomic_store` functions.



Code Tracing

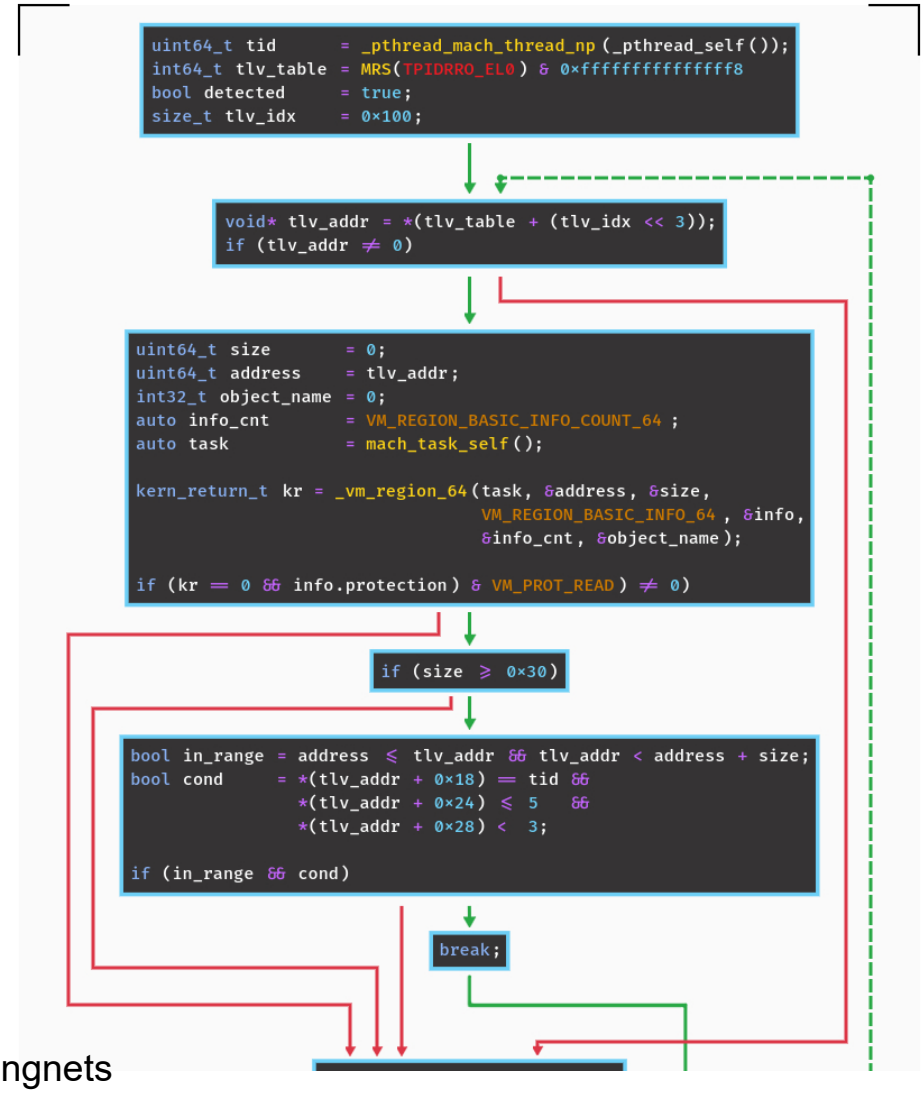
The *Code Tracing* check aims to verify that the current is not *traced*. By looking at the cross-references of `#EVT_CODE_TRACING_cbk_ptr`, we can identify two kinds of verification.

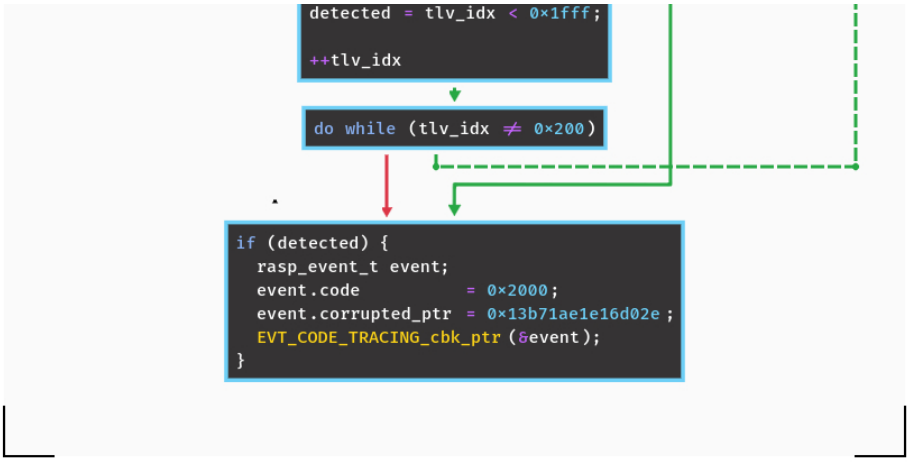
GumExecCtx

`EVT_CODE_TRACING` seems able to **detect** if the **Frida's Stalker** is running. It's the first time I can observe this kind of check and it's very smart. For those who would like to follow this analysis with the raw assembly code, I will use this range of addresses from the SingPass binary:

```
☰ 0×10019B6FC – 0×10019B82C
```

Here is the graph of the function that performs the Frida Stalker check:





Code associated with Frida Stalker Detection

Yes, this code is able to detect the Stalker. How? Let's start with the first basic block. `_pthread_mach_thread_np(_pthread_self())` aims at getting the thread id of the function that invokes this check.

Then more subtly, `MRS(TPIDRRO_EL0) & #-8` is used to **manually** access the thread local storage area. On ARM64, Apple uses the least significant byte of `TPIDRRO_EL0` to store the number of CPU while the MSB contains the TLS base address.

> See also: dyld – threadLocalHelpers.s

Then, the second basic block – which is the loop's entry – accesses the thread local variable with the *key* `tlv_idx` which ranges from `0x100` to `0x200` in the loop:

```
*(tlv_table + (tlv_idx << 3))
```

The following basic block which calls `_vm_region_64(…)` is used to verify that the `tlv_addr` variable contains a valid address with a *correct* size (i.e. larger than `0x30`). Under these conditions, it jumps into the following basic block with these *strange* memory accesses:

```
bool in_range = address <= tlv_addr && tlv_addr < address + size;
bool cond     = *(tlv_addr + 0x18) == tid &&
                *(tlv_addr + 0x24) <= 5   &&
                *(tlv_addr + 0x28) <  3;

if (in_range && cond)
```

Condition that (somehow) Triggers EVT_CODE_TRACING

To figure out the meaning of these memory accesses, let's remind that this function is associated with the `EVT_CODE_TRACING` event. Which well-known public tool could be associated with code tracing? Without too much risk, we can assume the Frida's Stalker.

If we look at the implementation of the Stalker, we can notice this kind of initialisation (in `gumstalker-arm64.c`):

```
void gum_stalker_init (GumStalker* self) {
  [ ... ]
  self->exec_ctx = gum_tls_key_new();
  [ ... ]
}

void* _gum_stalker_do_follow_me(GumStalker* self, ... ) {
  GumExecCtx* ctx = gum_stalker_create_exec_ctx( ... );
  gum_tls_key_set_value (self->exec_ctx, ctx);
}
```

So the Stalker creates a thread local variable that is used to store a pointer to the `GumExecCtx` structure which has the following layout:

```
struct _GumExecCtx {
  volatile gint state;
  gint64 destroy_pending_since;

  GumStalker * stalker;
  GumThreadId thread_id;
```

```
  GumArm64Writer code_writer;
  GumArm64Relocator relocator;
  [ ... ]
}
```

If we add the offsets of this layout and if we *virtually* inline the `GumArm64Writer` structure, we can get this representation:

```
struct _GumExecCtx {
  /* 0x00 */ volatile gint state;
  /* 0x08 */ gint64 destroy_pending_since;

  /* 0x10 */ GumStalker * stalker;
  /* 0x18 */ GumThreadId thread_id;

  GumArm64Writer code_writer {
  /* 0x20 */ volatile gint ref_count;
  /* 0x24 */ GumOS target_os;
  /* 0x28 */ GumPtrauthSupport ptrauth_support;
  ...
  };
}
```

`destroy_pending_since` is located at the offset **`0x08`** and not **`0x04`** because of the alignment enforced by the compiler.

With this representation, we can observe that:

- `*(tlv_table + 0x18)` effectively matches the `GumThreadId thread_id` attribute.
- `*(tlv_table + 0x24)` matches `GumOS target_os`
- `*(tlv_table + 0x28)` matches `GumPtrauthSupport ptrauth_support`

`GumOS` and `GumPtrauthSupport` are enums defined in `gumdefs.h` and `gummemory.h` with these values:

```
enum _GumOS {
  GUM_OS_WINDOWS,
  GUM_OS_MACOS,
  GUM_OS_LINUX,
  GUM_OS_IOS,
  GUM_OS_ANDROID,
  GUM_OS_QNX
};


enum _GumPtrauthSupport {
  GUM_PTRAUTH_INVALID,
  GUM_PTRAUTH_UNSUPPORTED,
  GUM_PTRAUTH_SUPPORTED
};
```

`GumOS` contains 6 entries starting from `GUM_OS_WINDOWS = 0` up to `GUM_OS_QNX = 5` and similarly, `GUM_PTRAUTH_INVALID = 0` while the last entry is associated with `GUM_PTRAUTH_SUPPORTED = 2`

Therefore, the previous *strange* conditions are used to fingerprint the `GumExecCtx` structure:

```
bool in_range = address ⩽ tlv_addr && tlv_addr < address + size;
bool cond     = _GumExecCtx→thread_id                   == tid &&
                _GumExecCtx→code_writer.target_os       ⩽ 5   &&
                _GumExecCtx→code_writer.ptrauth_support <  3;

if (in_range && cond) {
  → Trigger EVT_CODE_TRACING
}
```

One way to prevent this Stalker detection would be to recompile Frida with swapped fields in the `_GumExecCtx` structure.

Thread Check

An alternative to the previous Frida stalker check consists in accessing the current thread status

through the following call:

```
thread_read_t target = pthread_mach_thread_np(pthread_self());
uint32_t count = ARM_UNIFIED_THREAD_STATE_COUNT;
arm_unified_thread_state state;
thread_get_state(target, ARM_UNIFIED_THREAD_STATE, &state, &count);
```

Then, it checks if `state→ts_64.__pc` is within the `libsystem_kernel.dylib` thanks to the following comparison:

```
const auto mach_msg_addr = reinterpret_cast<uintptr_t>(&mach_msg);
const uintptr_t delta = abs(state→ts_64.__pc - mach_msg_addr)
if (delta > 0×4000) {
  rasp_event_info info;
  info.event = 0×2000; // EVT_CODE_TRACING;
  info.ptr = (uintptr_t*)0×13b71a24724edfe;
  EVT_CODE_TRACING_cbk_ptr(info);
}
```

In other words, `state→ts_64.__pc` is considered to be in `libsystem_kernel.dylib`, if its distance from `&mach_msg` is smaller than `0×4000`.

At first sight, I was a bit confused by this RASP check but since the previous checks, associated with `EVT_CODE_TRACING`, aims at detecting the Frida Stalker, this check is also likely designed to detect the Frida Stalker.

To confirm this hypothesis, I developed a small test case that reproduces this check, in a standalone binary and we can observe a difference depending on whether it runs through the Frida stalker or not:

```
iPhone:/tmp root# ./stalker
/usr/lib/system/libsystem_platform.dylib:0×1dafa21f0: stp x14, x15, [x3, #0×30]
/usr/lib/system/libsystem_platform.dylib:0×1dafa21f4: ret
/usr/lib/system/libsystem_kernel.dylib:0×1bf97735c: mov w21, #0
/usr/lib/system/libsystem_kernel.dylib:0×1bf977360: str w22, [x19]
/usr/lib/system/libsystem_kernel.dylib:0×1bf977364: b #0×1bf9772d4
/usr/lib/system/libsystem_kernel.dylib:0×1bf9772d4: mov x0, x21
/usr/lib/system/libsystem_kernel.dylib:0×1bf9772d8: add sp, sp, #1, lsl #12
/usr/lib/system/libsystem_kernel.dylib:0×1bf9772dc: add sp, sp, #0×470
/usr/lib/system/libsystem_kernel.dylib:0×1bf9772e0: ldp x29, x30, [sp, #0×30]
/usr/lib/system/libsystem_kernel.dylib:0×1bf9772e4: ldp x20, x19, [sp, #0×20]
/usr/lib/system/libsystem_kernel.dylib:0×1bf9772e8: ldp x22, x21, [sp, #0×10]
/usr/lib/system/libsystem_kernel.dylib:0×1bf9772ec: ldp x28, x27, [sp], #0×40
/usr/lib/system/libsystem_kernel.dylib:0×1bf9772f0: ret
/tmp/stalker:0×102638208: ldr x19, [sp, #0×158]
/tmp/stalker:0×10263820c: mov x0, x20
/tmp/stalker:0×102638210: bl #0×10264b560
pc: 0×10291caa0 → (distance: 0×bd03cf60)
Done!
(3) romain1:romain*                                    3.96 5.47 5.03 06:31
```

*Output of the Test Case **with** the Stalker*

```
iPhone:/tmp root# ./stalker
Stalker check starting ..
pc: 0×1bf95a644 → /usr/lib/system/libsystem_kernel.dylib (distance: 0×c44)
Done!
(3) romain1:romain*                                    3.96 5.47 5.03 06:31
```

*Output of the Test Case **without** the Stalker*

This check can be bypassed without too much difficulty by using the function `gum_stalker_exclude` to exclude the library `libsystem_kernel.dylib` from the stalker:

```
GumStalker* stalker = gum_stalker_new();
exclude(stalker, "libsystem_kernel.dylib");
{
  // Stalker Check
}
```

As a result of this exclusion, `state→ts_64.__pc` is located in `libsystem_kernel.dylib`:

```
iPhone:/tmp root# ./stalker
/usr/lib/system/libsystem_kernel.dylib:0×1bf9772dc: add sp, sp, #0×470
/usr/lib/system/libsystem_kernel.dylib:0×1bf9772e0: ldp x29, x30, [sp, #0×30]
/usr/lib/system/libsystem_kernel.dylib:0×1bf9772e4: ldp x20, x19, [sp, #0×20]
/usr/lib/system/libsystem_kernel.dylib:0×1bf9772e8: ldp x22, x21, [sp, #0×10]
/usr/lib/system/libsystem_kernel.dylib:0×1bf9772ec: ldp x28, x27, [sp], #0×40
/usr/lib/system/libsystem_kernel.dylib:0×1bf9772f0: ret
/tmp/stalker:0×104d00250: ldr x19, [sp, #0×158]
/tmp/stalker:0×104d00254: mov x0, x20
/tmp/stalker:0×104d00258: bl #0×104d137b4
pc: 0×1bf95a644 → /usr/lib/system/libsystem_kernel.dylib (distance: 0×c44)

(3) romain1:romain*                                    3.96 5.47 5.03 06:31
```

Output of the Test Case with Excluded Memory Ranges

App Loaded Libraries

The RASP event `EVT_APP_LOADED_LIBRARIES` aims at checking the integrity of the Mach-O's dependencies. In other words, it checks that the Mach-O imported libraries have not been altered.

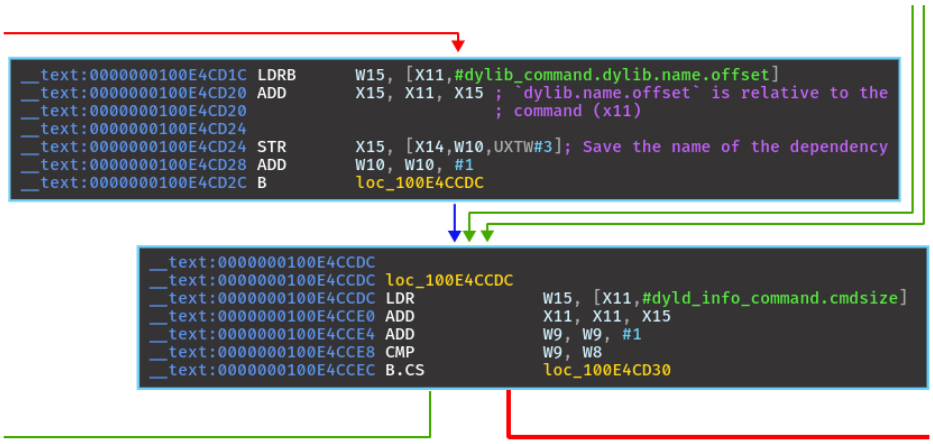
> ☰ Assembly ranges: 0×100E4CDF8 – 0×100e4d39c

The code associated with this check starts by accessing the Mach-O header thanks to the `dladdr` function:

```
Dl_info dl_info;
dladdr(&static_var, &dl_info);
```

`Dl_info` contains the base address of the library which encompasses the address provided in the first parameter and since, a Mach-O binary is loaded with its header, `dl_info.dli_fbase` actually points to a `mach_header_64`.

Then the function iterates over the `LC_ID_DYLIB`-like commands to access dependency's name:



This name contains the path to the dependency. For instance, we can access this list as follows:

```
import lief

singpass = lief.parse("./SingPass")

for lib in singpass.libraries:
  print(lib.name)

# Output:
/System/Library/Frameworks/AVFoundation.framework/AVFoundation
/System/Library/Frameworks/AVKit.framework/AVKit
...
/System/Library/Frameworks/leveldb.framework/leveldb
```

```
@rpath/nanopb.framework/nanopb
```

The dependency's names are used to fill a hash table in which a hash value in encoded on 32 bits:

```
// Pseudo code
uint32_t TABLE[0×6d]
for (size_t i = 0; i < 0×6d; ++i) {
  TABLE[i] = hash(lib_names[i]);
}
```

Later in the the code, this computed table is compared with another hash table – **hard-coded in the code** – which looks like this:

```
__text:0000000100E4CF38 loc_100E4CF38                   ; CODE XREF: sub_100E4CC54+230↑j
__text:0000000100E4CF38                 MOV             X10, SP
__text:0000000100E4CF3C                 SUB             X8, X10, #0×1C0
__text:0000000100E4CF40                 MOV             SP, X8
__text:0000000100E4CF44                 MOV             X9, #0
__text:0000000100E4CF48                 MOV             X11, #0×1D2A29E0195DDC1
__text:0000000100E4CF58                 MOV             X12, #0×4AA0A7902C19769
__text:0000000100E4CF68                 STP             X11, X12, [X8]
__text:0000000100E4CF6C                 MOV             X11, #0×64CAED105C09EBA
__text:0000000100E4CF7C                 MOV             X12, #0×EDC68A50A44D7D1
__text:0000000100E4CF8C                 STP             X11, X12, [X8,#0×10]
__text:0000000100E4CF90                 MOV             X11, #0×128801E010DCF774
__text:0000000100E4CFA0                 MOV             X12, #0×14EDACA112DF984A
__text:0000000100E4CFB0                 STP             X11, X12, [X8,#0×20]
__text:0000000100E4CFB4                 MOV             X11, #0×166DA22D164DF42A
__text:0000000100E4CFC4                 MOV             X12, #0×1B2CAF8A1ACDDCCF
__text:0000000100E4CFD4                 STP             X11, X12, [X8,#0×30]
__text:0000000100E4CFD8                 MOV             X11, #0×1CF2CE101BB56374
__text:0000000100E4CFE8                 MOV             X12, #0×235D2E461E37FF16
__text:0000000100E4CFF8                 STP             X11, X12, [X8,#0×40]
__text:0000000100E4CFFC                 MOV             X11, #0×28F80E87260C94F3
__text:0000000100E4D00C                 MOV             X12, #0×2CBB87222BFB0F4D
__text:0000000100E4D01C                 STP             X11, X12, [X8,#0×50]
```

Fig 4. Examples of Hashes

If some libraries have been modified to inject, for instance, `FridaGadget.dylib` then the hash **dynamically** computed will not match the hash hard-coded in the code.

Whilst the implementation of this check is pretty *"standard"*, there are a few points worth mentioning:

- Firstly, the hash function seems be a derivation of the MurmurHash.
- Secondly, the hash is encoded on **32 bits** but the code in the Figure 4 references the X11/X12 registers which are 64 bits. This is actually a compiler optimization to limit the number of memory accesses.
- Thirdly, the hard coded hash values are duplicated in the binary for each instance of the check. In SingPass, this RASP check is present twice thus, we find these values at the following locations: `0×100E4CF38`, `0×100E55678`. This duplication is likely used to prevent a single spot location that would be easy to patch.

Code System Lib

This check is associated with the event `EVT_CODE_SYSTEM_LIB` which consists in verifying the integrity of the **in-memory** system libraries with their content in the dyld shared cache (**on-disk**).

```
☰ Assembly ranges: 0×100ED5BF8 – 0×100ED5D6C and 0×100ED5E0C – 0×100ED62D4
```

This check usually starts with the following pattern:

```
__text:00100E80AF0      ADR         X0, check_region_cbk ; cbk
__text:00100E80AF4      NOP
__text:00100E80AF8      BL          iterate_system_region
__text:00100E80AFC      ORR         W8, W0, W21
__text:00100E80B00      CBZ         W8, loc_100E80B50
__text:00100E80B04      MOV         X21, SP
__text:00100E80B08      ADRP        X8, #EVT_CODE_SYSTEM_LIB_cbk_ptr@PAGE
```

```
__text:00100E80B0C     LDR         X8, [X8,#EVT_CODE_SYSTEM_LIB_cbk_ptr@PAGEOFF]
__text:00100E80B10     MOV         X9, SP
__text:00100E80B14     SUB         X10, X9, #0x10
__text:00100E80B18     MOV         SP, X10
__text:00100E80B1C     MOV         X11, #0x13B851C07E9DBCD
__text:00100E80B2C     STUR        X11, [X9,#-0x10]
__text:00100E80B30     MOV         X9, SP
__text:00100E80B34     SUB         X0, X9, #0x10
__text:00100E80B38     MOV         SP, X0
__text:00100E80B3C     MOV         W11, #0x1000
__text:00100E80B40     STUR        W11, [X9,#-0x10]
__text:00100E80B44     STUR        X10, [X9,#-8]
__text:00100E80B48     BLR         X8
__text:00100E80B4C     MOV         SP, X21
```

If the result of `iterate_system_region` with the given `check_region_cbk` callback is not 0, it triggers the `EVT_CODE_SYSTEM_LIB` event:

```
if (iterate_system_region(check_region_cbk) != 0) {
// Trigger `EVT_CODE_SYSTEM_LIB`
}
```

To understand the logic behind this check, we need to understand the purpose of the `iterate_system_region` function and its relationship with the callback `check_region_cbk`.

iterate_system_region

> As for all the functions referenced in the blog post, their names come from my own analysis and might be inaccurate. Most of the functions related to the RASP checks were obviously stripped. In this case, `iterate_system_region` matches the original *sub_100ED5BF8*

This function aims to call the system function `vm_region_recurse_64` and then, filter its output on conditions that could trigger the callback given in the first parameter: `check_region_cbk`.

`iterate_system_region` starts by accessing the base address of the dyld shared cache thanks to the `SYS_shared_region_check_np` syscall. This address is used to read and **memoize** a few attributes from the `dyld_cache_header` structure:

1. The shared cache header
2. The shared cache end address
3. Other limits related to the shared cache

The following snippet gives an overview of these computations:

```
static dyld_shared_cache* header = nullptr; /* At: 0x1010DE940 */
static uintptr_t g_shared_cache_end;        /* At: 0x1010DE948 */
static uintptr_t g_overflow_address;        /* At: 0x1010DE950 */
static uintptr_t g_module_last_addr;        /* At: 0x1010DE958 */

if (header == nullptr) {
// return;
}

uintptr_t shared_cache_base;
syscall(SYS_shared_region_check_np, &shared_cache_base);
header = shared_cache_base;

g_shared_cache_end = shared_cache_addr + header->mappings[0].size;
g_overflow_address = -1;
g_module_last_addr = g_shared_cache_end;
if (header->imagesTextCount > 0) {
  uintptr_t slide = shared_cache_addr - header->mappings[0].address;
  uintptr_t tmp_overflow_address = -1;
  uintptr_t shared_cache_end_tmp = shared_cache_end;

  for (size_t i = 0; i < header->imagesTextCount; ++i) {
    const uintptr_t txt_start_addr = slide      + header->imagesText[i].loadAddress;
    const uintptr_t txt_end_addr   = start_addr + header->imagesText[i].textSegmentSize;

    if (txt_start_addr >= shared_cache_end_tmp && txt_start_addr < tmp_overflow_address) {
      g_overflow_address   = start_addr;
      tmp_overflow_address = start_addr;
    }
```

```
    if (txt_end_addr ≥ shared_cache_end_tmp) {
      g_module_last_addr   = txt_end_addr;
      shared_cache_end_tmp = txt_end_addr;
    }
  }
}
```

From a reverse engineering point of view, the stack variable used to memoize these information is aliased with the parameter `info` of `vm_region_recurse_64` that is called later. I don't know if this aliasing is on purpose, but it makes the reverse engineering of the structures a bit more complicated.

Following this memoization, there is a loop on `vm_region_recurse_64` which queries the `vm_region_submap_info_64` information for these addresses in the range of the dyld shared cache. We can identify the type of the query (`vm_region_submap_info_64`) thanks to the `mach_msg_type_number_t *infoCnt` argument which is set to `19`:

```
; In this basic block, the stack variable `#0×B0+info` is aliased with
; the variable used for, saving (temporarily) the shared cache information
; c.f. loc_100ED5C68

__text:0000000100ED5D24 loc_100ED5D24
__text:0000000100ED5D24                 X9 <- shared cache base address
__text:0000000100ED5D24 STR             X9, [SP,#0×B0+pAddr]
__text:0000000100ED5D28 MOV             W8, #0×13 ; → 19 ⟺ vm_region_submap_info_64
__text:0000000100ED5D2C STR             W8, [SP,#0×B0+infoCnt]
__text:0000000100ED5D30 ADD             X1, SP, #0×B0+pAddr      ; address
__text:0000000100ED5D34 SUB             X2, X29, #-size          ; size
__text:0000000100ED5D38 SUB             X3, X29, #-nesting_depth ; nesting_depth
__text:0000000100ED5D3C ADD             X4, SP, #0×B0+info       ; info
__text:0000000100ED5D40 ADD             X5, SP, #0×B0+infoCnt    ; infoCnt
__text:0000000100ED5D44 MOV             X0, X20                  ; target_task
__text:0000000100ED5D48 BL              _vm_region_recurse_64
__text:0000000100ED5D4C CBZ             W0, loc_100ED5D70
```

This loop breaks under certain conditions and the callback is triggered with other conditions. As it is explained a bit later, the callback verifies the in-memory integrity of the library present in the dyld shared cache.

The verification and the logic behind this check is prone to take time, that's why the authors of the check took care of filtering the addresses to check to avoid useless (heavy) computations.

Basically, the callback that performs the in-depth inspection of the shared cache is triggered if:

```
if (info.pages_swapped_out ≠ 0 ||
    info.pages_swapped_out == 0 && info.protection & VM_PROT_EXECUTE)
{
  bool integrity_failed = check_region_cbk(address);
}
```

check_region_cbk

When the conditions are met, `iterate_system_region` calls the `check_region_cbk` with the suspicious address in the first parameter:

```
int iterate_system_region(callback_t cbk) {
  int ret = 0;
  if (cond(address)) {
    ret = cbk(address) {
      // Checks on the dyld_shared_cache
    }
  }
  return ret;
}
```

During the analysis of SingPass, only **one** callback is used in pair with `iterate_system_region`, and its code is not **especially obfuscated** (except the strings). Once we know that the checks are related to the dyld shared cache, we can quite easily figure out the structures involved in this function. This callback is located at the address `0×100ed5e0c` and renamed `check_region_cbk`.

Firstly, it starts by accessing the information about the address:

```
int check_region_cbk(uintptr_t address) {
  Dl_info info;
  dladdr(address, info);
  // ...
}
```

This information is used to read the content of the `__TEXT` segment associated with the `address` parameter:

```
auto* header = reinterpret_cast<mach_header_64*>(info.dli_fbase);

segment_command_64 __TEXT = get_text_segment(header);

vm_offset_t data = 0;
mach_msg_type_number_t* dataCnt = 0;

vm_read(task_self_trap(), info.dli_fbase, __TEXT.vmsize, &data, &dataCnt);
```

> The `__TEXT` strings is encoded as well as the different paths of the shared cache like `/System/Library/Caches/com.apple.dyld/dyld_shared_cache_arm64e` and the header's magic values: `0x01010b9126: dyld_v1 arm64e` or `0x01010b9116: dyld_v1 arm64`

On the other hand, the function opens the `dyld_shared_cache` and looks for the section of the shared cache that contains the library associated with the `address` parameter:

```
int fd = open('/System/Library/Caches/com.apple.dyld/dyld_shared_cache_arm64');
(1) mmap(nullptr, 0x100000, VM_PROT_READ, MAP_NOCACHE | MAP_PRIVATE, fd, 0x0): 0x109680000

// Look for the shared cache entry associated with the provided address
(2) mmap(nullptr, 0xad000, VM_PROT_READ, MAP_NOCACHE | MAP_PRIVATE, fd, 0x150a9000): 0x109681000
```

The purpose of the second call to `mmap()` is to load the slice of the shared cache that contains the code of the library. Then, the function checks **byte per byte** that the `__TEXT` segment's content matches the in-memory content. The loop which performs this comparison is located between these addresses: `0x100ED6C58 - 0x100ED6C70`.

As we can observe from the description of this RASP check, the authors paid a lot of attention to avoid performance issues and memory overhead. On the other hand, the callback `check_region_cbk` was never called during my experimentations (even when I hooked system function). I don't know if it's because I misunderstood the conditions but in the end, I had to manually force the conditions (by forcing the `pages_swapped_out` to 1).

> `vm_region_recurse_64` seems also always paired with an anti-hooking verification that is slightly different from the check described at the beginning of this blog post. Its analysis is quite easy and can be a good exercise.

RASP Design Weaknesses

Thanks to the different `#EVT_*` static variables that hold function pointers, the obfuscator enables to have dedicated callbacks for the supported RASP events. Nevertheless, the function `init_and_check_rasp` defined by the application's developers setup **all** these pointers **to the same** callback: `hook_detect_cbk_user_def`. In such a design, all the RASP events end up in a single function which weakens the strength of the different RASP checks.

It means that we only have to target this function to disable or bypass the RASP checks.

Using Frida Gum, the bypass is as simple as using `gum_interceptor_replace` with an empty function:

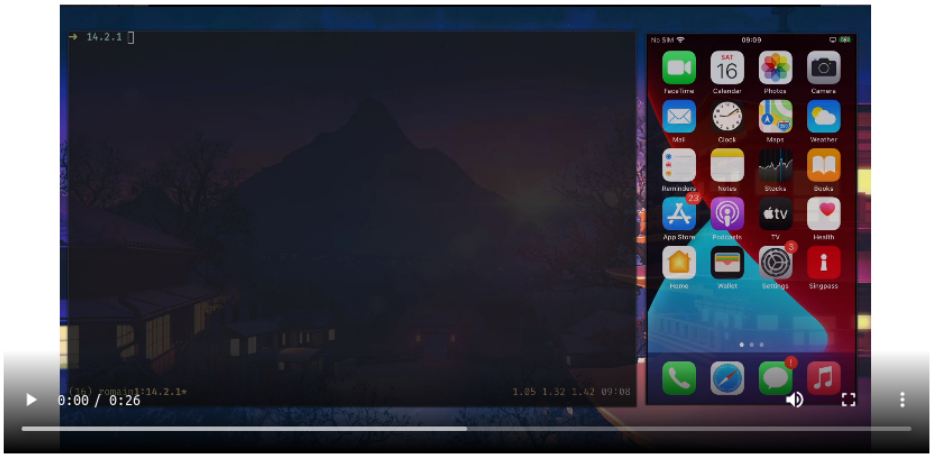
```
enum class RASP_EVENTS : uint32_t {
  EVT_ENV_JAILBREAK        = 0x1,
  EVT_ENV_DEBUGGER         = 0x2,
  EVT_APP_SIGNATURE        = 0x20,
  EVT_APP_LOADED_LIBRARIES = 0x40,
  EVT_CODE_PROLOGUE        = 0x400,
  EVT_CODE_SYMBOL_TABLE    = 0x800,
  EVT_CODE_SYSTEM_LIB      = 0x1000,
  EVT_CODE_TRACING         = 0x2000,
```

```
struct event_info_t {
  RASP_EVENTS event;
  uintptr_t** ptr_to_corrupt;
};

void do_nothing(event_info_t info) {
  RASP_EVENTS evt = info.event;
  // ...
  return;
}

// This is **pseudo code**
gum_interceptor_replace(
  listener->interceptor,
  reinterpret_cast<void*>(&hook_detect_cbk_user_def)
  do_nothing,
  reinterpret_cast<void*>(&hook_detect_cbk_user_def)
);
```

Thanks to this weakness, I could prevent the error message from being displayed as soon as the application starts.



SingPass Jailbreak & RASP Bypass

> It exists two other RASP checks: `EVT_APP_MACHO` and `EVT_APP_SIGNATURE` which were not enabled by the developers and thus, are not present in SingPass.

Conclusion

This first part is a good example of the challenges when using or designing an obfuscator with RASP features. On one hand, the commercial solution implements strong and advanced RASP functionalities with, for instance, inlined syscalls spread in different places of the application. On the other hand, the app's developers weakened the RASP functionalities by setting the **same callback** for all the events. In addition, it seems that the application **does not use the native code obfuscation** provided by the commercial solution which makes the RASP checks un-protected against static code analysis. It could be worth to enforce code obfuscation on these checks regardless the configuration provided by the user.

From a developer point of view, it can be very difficult to understand the impact in term of reverse-engineering when choosing to setup the same callback while it can be a good design decision from an architecture perspective.

In the second part of this series about iOS code obfuscation, we will dig a bit more in native code obfuscation through another application, where the application reacts differently to the RASP events and where the code is obfuscated with MBA, Control-Flow Flattening, etc.

If you have questions feel free to ping me .

Annexes

JB Detection Files	Listed in PokemonGO
/.bootstrapped	No
/.installed_taurine	No
/.mount_rw	No
/Library/dpkg/lock	No
/binpack	Yes
/odyssey/cstmp	No
/odyssey/jailbreakd	No
/payload	No
/payload.dylib	No
/private/var/mobile/Library/Caches/kjc.loader	No
/private/var/mobile/Library/Sileo	No
/taurine	No
/taurine/amfidebilitate	No
/taurine/cstmp	No
/taurine/jailbreakd	No
/taurine/jbexec	No
/taurine/launchjailbreak	No
/taurine/pspawn_payload.dylib	No
/var/dropbear	No
/var/jb	No
/var/lib/undecimus/apt	No
/var/motd	No
/var/tmp/cydia.log	No

Flagged Packages
/Applications/AutoTouch.app/AutoTouch
/Applications/iGameGod.app/iGameGod
/Applications/zxtouch.app/zxtouch
/Library/Activator/Listeners/me.autotouch.AutoTouch.ios8
/Library/LaunchDaemons/com.rpetrich.rocketbootstrapd.plist
/Library/LaunchDaemons/com.tigisoftware.filza.helper.plist
/Library/MobileSubstrate/DynamicLibraries/ATTweak.dylib
/Library/MobileSubstrate/DynamicLibraries/GameGod.dylib
/Library/MobileSubstrate/DynamicLibraries/LocalIAPStore.dylib
/Library/MobileSubstrate/DynamicLibraries/Satella.dylib
/Library/MobileSubstrate/DynamicLibraries/iOSGodsiAPCracker.dylib

```
/Library/MobileSubstrate/DynamicLibraries/pccontrol.dylib
/Library/PreferenceBundles/SatellaPrefs.bundle/SatellaPrefs
/Library/PreferenceBundles/iOSGodsiAPCracker.bundle/iOSGodsiAPCracker
```