iOS Swift Anti-Jailbreak Bypass with Frida



May 13, 2020 | Posted by Raffaele Sabato

Link: https://syrion.me/blog/ios-swift-antijailbreak-bypass-frida/

References:

- <u>https://syrion.me/</u>
- <u>https://twitter.com/syrion89</u>
- <u>https://www.linkedin.com/in/raffaelesabato/</u>

Frida

Frida is a dynamic binary instrumentation framework that has been around for a while. In a nutshell, Frida allows reverse engineers to perform activities such as function hooking/trancing and runtime code modification. If your target is an iOS application, Frida provides you with powerful Objective-C API, making painless reverse engineering tasks. Unfortunately, out of the box, Frida lacks of Swift API, and some community contributions are <u>outdated</u>. Analysing Swift iOS applications with Frida hasn't been an easy task for me so far. Fortunately, my friend <u>r3dxOf</u> provided me with some suggestions about how to approach the problem, and I will share with you what I learnt during this journey.

Swift Security Checks

In this blogpost, I will describe how to bypass a number of jailbreak and reverse engineering detection mechanisms implemented within the **IOSSecuritySuite**. The aforementioned project is written in Swift and is hosted on <u>Github</u>. However, we will perform the analysis against a non-stripped iOS application which implements such library. The analysis will be conducted without the support of the source code in order to simulate a real-life scenario. When the application is executed, it shows the message below, telling us that our iPhone is jailbroken. Some suspicious files are found and the application is considered "reversed".



Figure 1 - Security Checks

We have two modules defined within the binary:

- FrameworkClientApp
- IOSSecuritySuite

Because we know that the **IOSSecuritySuite** modules contains the jailbreak detection mechanism logic, we will reverse it first.

Reverse Engineering

When I'm looking for jailbreak detection mechanisms, I usually start searching for strings and functions containing the word "**jailbr**" (jail, jailbreak or jailbroken) or "**root**". We have a lot of matches as shown below:

Q∼ jailbr		۵
 Tag Scope 		¥
rag scope		•
Address	Туре	Name
Oxacac	Р	static IOSSecuritySuite.IOSSecuritySuite.amlJailbroken() -> Swift.Bool
0xacf4	Р	static IOSSecuritySuite.IOSSecuritySuite.amlJailbrokenWithFailMessage() -> (jailbroken: S
0xad38	Р	static IOSSecuritySuite.IOSSecuritySuite.amlJailbrokenWithFailedChecks() -> (jailbroken:
0xe628	Р	static IOSSecuritySuite.JailbreakCheckderived_enum_equals(IOSSecuritySuite.Jailbreak
0xe798	Р	IOSSecuritySuite.JailbreakCheck.hashValue.getter : Swift.Int
0xe80c	Р	lazy protocol witness table accessor for type IOSSecuritySuite.JailbreakCheck and confor
0xe870	Р	IOSSecuritySuite.JailbreakCheck.hash(into: inout Swift.Hasher) -> ()
0xe968	Р	static IOSSecuritySuite.JailbreakCheck.allCases.getter : [IOSSecuritySuite.JailbreakCheck]
0xe9c0	Р	protocol witness for static Swift.Equatable.== infix(A, A) -> Swift.Bool in conformance IOS
0xe9e4	Р	protocol witness for Swift.Hashable.hashValue.getter : Swift.Int in conformance IOSSecuri
0xea00	Р	protocol witness for Swift.Hashable.hash(into: inout Swift.Hasher) -> () in conformance IO
0xea1c	Р	protocol witness for Swift.HashablerawHashValue(seed: Swift.Int) -> Swift.Int in conform
0xea38	Р	protocol witness for static Swift.CaseIterable.allCases.getter : A.AllCases in conformance I
0xea60	Р	IOSSecuritySuite.JailbreakChecker.JailbreakStatus.passed.getter : Swift.Bool
0xea68	Р	IOSSecuritySuite.JailbreakChecker.JailbreakStatus.failMessage.getter : Swift.String
0xeaa0	Р	IOSSecuritySuite.JailbreakChecker.JailbreakStatus.failedChecks.getter : [(check: IOSSecu
0xeadc	Р	IOSSecuritySuite.JailbreakChecker.JailbreakStatus.init(passed: Swift.Bool, failMessage: S
0xeae4	Р	static IOSSecuritySuite.JailbreakChecker.amlJailbroken() -> Swift.Bool
0xeb38	Р	static IOSSecuritySuite. JailbreakChecker. (performChecks in _F8E503CD913F87B6FC3E9
0xf198	Р	static IOSSecuritySuite. JailbreakChecker.amlJailbrokenWithFailMessage() -> (jailbroken:
0xf21c	Р	static IOSSecuritySuite. JailbreakChecker.amlJailbrokenWithFailedChecks() -> (jailbroken:
0xf298	Р	static IOSSecuritySuite. JailbreakChecker. (checkURLSchemes in _F8E503CD913F87B6FC
0xf718	Р	static IOSSecuritySuite. JailbreakChecker. (checkExistenceOfSuspiciousFiles in _F8E503C
0x10280	Р	static IOSSecuritySuite. JailbreakChecker. (checkSuspiciousFilesCanBeOpened in _F8E503
0x10824	Р	static IOSSecuritySuite. JailbreakChecker. (checkRestrictedDirectoriesWriteable in _F8E50
0x112a8	Р	static IOSSecuritySuite. JailbreakChecker. (checkFork in _F8E503CD913F87B6FC3E966D6
0x11574	Р	static IOSSecuritySuite. JailbreakChecker. (checkSymbolicLinks in _F8E503CD913F87B6F
0x11c0c	Р	static IOSSecuritySuite. JailbreakChecker. (checkDYLD in _F8E503CD913F87B6FC3E966D
0x123a4	Р	static IOSSecuritySuite. JailbreakChecker. (canOpenUrlFromList in _F8E503CD913F87B6F
1112 Jabola		

Figure 2 - Functions containing the string "Jailbr"

My first goal is to hook each function and see if it does affect the alert message that we saw before.

Bypass canOpenUrlFromList

The first function we are going to analyse is

_\$s16IOSSecuritySuite16JailbreakCheckerC18canOpenUrlFromList33_F8E503CD913F87B6F 3E966D69D 813ABLL10urlSchemesSb6passed_SS11failMessagetSaySSG_tFZ.

By reversing it, we can see the canOpenURL method call at the address offset **0x126bc**. According with the AArch64 ABI, the **x0** register will contain the value returned from the function call. This function usually tries to open suspicious URL scheme such as "**cydia://**" in order to identify jailbreak artefact. When it founds any suspicious file, the function returns "true". Because the value returned is a Boolean it should contain the values **0x01** (**true**) or **0x00** (**false**), therefore we have to change it to **0x00** in order to bypass this check.

012688	blr	x9	
01268c	adrp	×8. #0×19000	: 0x19138@PAGE
012690	add	x8, x8, #0x138	; 0x19138@PAGEOFF, &@selector(canOpenURL:)
012694	ldr	x1, [x8]	; "canOpenURL:",@selector(canOpenURL:)
012698	sub	x8, x29, #0x68	, campenoner (escretor (campenoner)
01269c	ldur	x0, [x8, #-0x100]	: 0x19038
0126a0	sub	x8, x29, #0x58	
0126a4	ldur	x2, [x8, #-0x100]	: 0x19038
0126a8	sub	x8, x29, #0x78	
0126ac	stur	x0, [x8, #-0x100]	: 0x19038
0126b0	mov	x0, x2	
0126b4	sub	x8, x29, #0x78	
012668	ldur	x2. [x8. #-0x100]	: 0x19038
0126bc	bl	<pre>impstubs_objc_msgSend</pre>	; objc_msgSend
012660	sub	x8, x29, #0x68	
0126c4	ldur	×1, [×8, #-0×100]	; 0×19038
0126c8	sub	x8, x29, #0x7c	
0126cc	stur	w0, [x8, #-0x100]	; 0×19038
0126d0	mov	x0, x1	
0126d4	bl	impstubsobjc_release	; objc_release
0126d8	sub	x8, x29, #0x58	
0126dc	ldur	×0, [×8, #-0×100]	; 0x19038
0126e0	bl	impstubsobjc_release	
0126e4	sub	x8, x29, #0x7c	
0126e8	ldur	w11, [x8, #-0x100]	; 0×19038
0126ec	tbz	w11, 0x0, loc_12874	

Figure 3 - canOpenURL

We can use Frida's Module.getBaseAddress() function to obtain the base address where the library is loaded in memory (Frida calculate the ASLR Shift for us) and then we have to add the **0x126c** offset to it. Then we ask Frida to hook and replace the code at that specific address (baseAddress+offset). Our implementations will change the **x0** register value from **0x01** (true) to **0x00** (false). In Frida we can access register's values by using the this.context object.

```
console.log("Bypass canOpenUrlFromList");
}
},
});
```

Running the script, we can see that the target instruction is reached and the check is bypassed as shown in the images below.

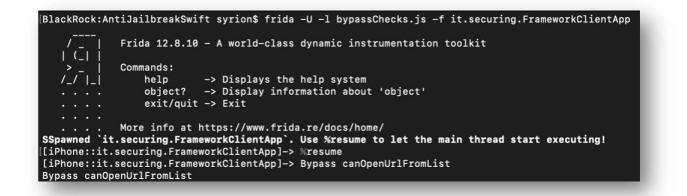


Figure 4 - Frida Script

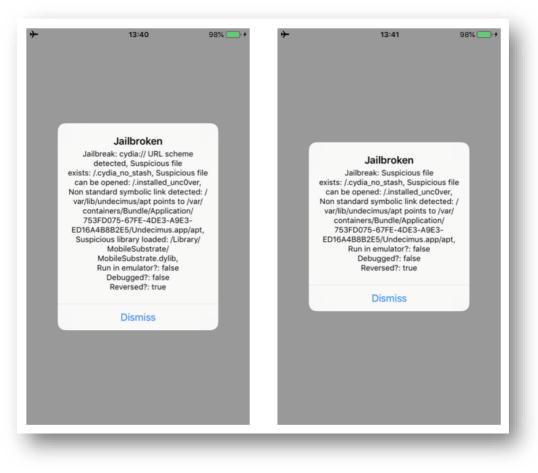


Figure 5 – Security Checks

N.B. During the analysis it was observed that some Swift functions defined within the **IOSSecuritySuite** library, executed **Objective-C** code by "bridging" them. It should be noted that in such circumstances, controls can also be bypassed by using the "**ObjC**" object defined within Frida's API.

Bypass checkExistenceOfSuspiciousFiles

Function:

_\$s16IOSSecuritySuite16JailbreakCheckerC31checkExistenceOfSuspiciousFiles33_F8E503C 913F87 B6FC3E966D69D813ABLLSb6passed_SS11failMessagetyFZ

In the disassembled code below we can see that the function calls the **fileExistsAtPath** method at the offset address **0x100a8**, so we need to change the return value as we did before.

0010074	bl	<pre>impstubs\$sSS10FoundationE19_bridgeTo</pre>	<pre>ObjectiveCSo8NSStringCyF ; (extension in Foundation):Sv</pre>
0010078	ldr	x8, [sp, #0x1e0 + var_128]	
001007c	str	x0, [sp, #0x1e0 + var_148]	
0010080	mov	x0, x8	
0010084	bl	<pre>impstubsswift_bridgeObjectRelease</pre>	; swift_bridgeObjectRelease
0010088	adrp	x8, #0x19000	; 0x19120@PAGE
001008c	add	x8, x8, #0x120	; 0x19120@PAGEOFF, &@selector(fileExistsAtPath:)
0010090	ldr	x1, [x8]	; "fileExistsAtPath:",@selector(fileExistsAtPath:)
0010094	ldr	x0, [sp, #0x1e0 + var_148]	
0010098	ldr	x2, [sp, #0x1e0 + var_138]	
001009c	str	x0, [sp, #0x1e0 + var_150]	
00100a0	mov	x0, x2	
00100a4	ldr	x2. [sp. #0x1e0 + var 150]	
00100a8	bl	imo stubs obic msøSend	: obic msgSend
00100ac	ldr	x1, [sp, #0x1e0 + var_148]	
00100b0	str	w0, [sp, #0x1e0 + var_154]	
00100b4	mov	x0, x1	
00100b8	bl	<pre>impstubsobjc_release</pre>	; objc_release
00100bc	ldr	x0, [sp, #0x1e0 + var_138]	
00100c0	bl	<pre>impstubsobjc_release</pre>	; objc_release
00100c4	ldr	w11, [sp, #0x1e0 + var_154]	
00100c8	tbz	w11, 0x0, loc_101f8	

Figure 6 - fileExistsAtPath

Using the same Frida code, we can set the new target address, and bypass the check.

Frida Code

Running the updated script, we can bypass the two methods. As shown in the images below. Moreover, the message in the alert box will change as well.

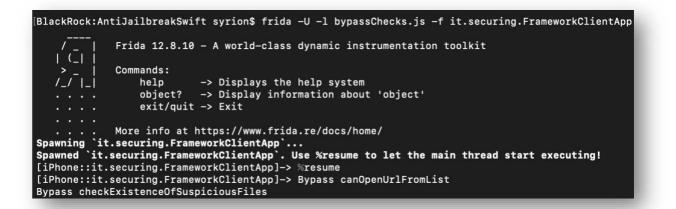


Figure 7 - Frida Script

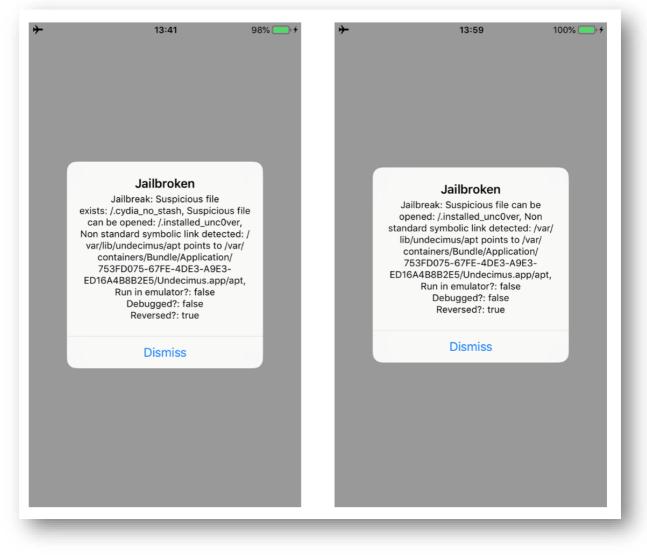


Figure 8 - Security Checks

Bypass checkSuspiciousFilesCanBeOpened

Function:

_\$s16IOSSecuritySuite16JailbreakCheckerC31checkSuspiciousFilesCanBeOpened33_F8E50 3CD913F87 B6FC3E966D69D813ABLLSb6passed_SS11failMessagetyFZ

As we can see in the disassembled code below, the **isReadableFileAtPath** method is called at the offset address **0x1064c** and the return value is stored in the **x0** register as usual.

0010628 001062c	bl	<pre>impstubsswift_bridgeObjectRelease</pre>	; swift_bridgeObjectRelease
	adrp	x8, #0x19000	; 0x19140@PAGE
010630	add	x8, x8, #0x140	; 0x19140@PAGEOFF, &@selector(isReadableFileAtPath:)
010634	ldr	×1, [x8]	
010638	ldr	x0, [sp, #0x1e0 + var_148]	
01063c	ldr	x2, [sp, #0x1e0 + var_138]	
010640	str	x0, [sp, #0x1e0 + var_150]	
010644	mov	x0, x2	
010648	ldr	x2. [sn. #0x1e0 + var 150]	
01064c	bl	<pre>impstubsobjc_msgSend</pre>	; objc_msgSend
010020	Lar	xi, [sp, #0xie0 + var_i48]	
010654	str	w0, [sp, #0x1e0 + var_154]	
010658	mov	x0, x1	
01065c	bl	<pre>impstubsobjc_release</pre>	
010660	ldr	x0, [sp, #0x1e0 + var_138]	
010664	bl	<pre>impstubsobjc_release</pre>	; objc_release
010668	ldr	w11, [sp, #0x1e0 + var 154]	
01066c	tbz	w11, 0x0, loc 1079c	

Figure 9 - isReadableFileAtPath

Using the same script with the new address we can bypass the check.

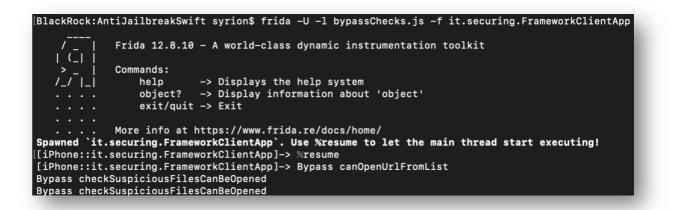


Figure 10 - Frida Script

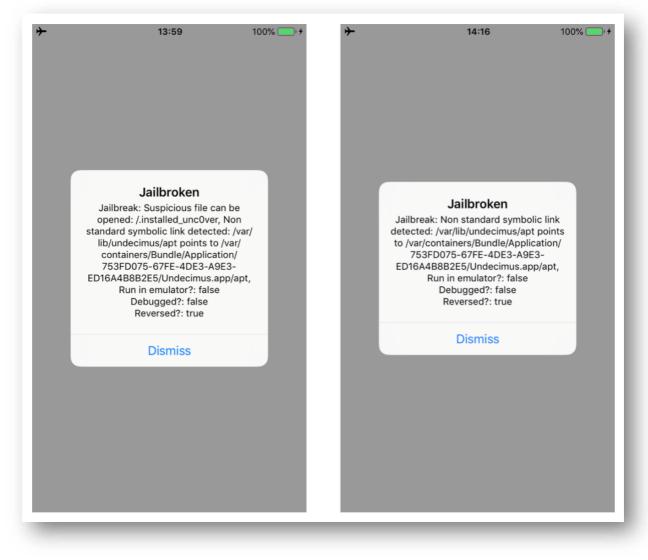


Figure 11 - Security Checks

Bypass checkSymbolicLinks

Function:

_\$s16IOSSecuritySuite16JailbreakCheckerC18checkSymbolicLinks33_F8E503CD913F87B6F C3E966 D69D813ABLLSb6passed_SS11failMessagetyFZ

Our iPhone is still recognized as **"jailbroken**", so we need to bypass other methods. As we can see below, the **destinationOfSymbolicLinkAtPath** method is called at the offset address **0x118ac**, so we can modify the **x0** register by replacing the its value with **0x00** as we did before.

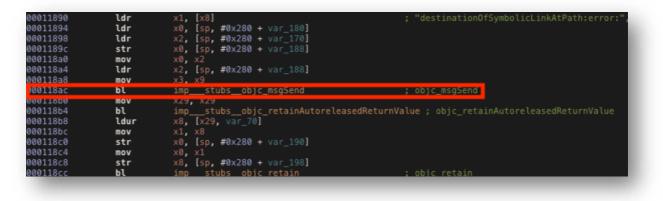


Figure 12 - destinationOfSymbolicLinkAtPath

Using the same script we can change the return value contained in the **x0** register.

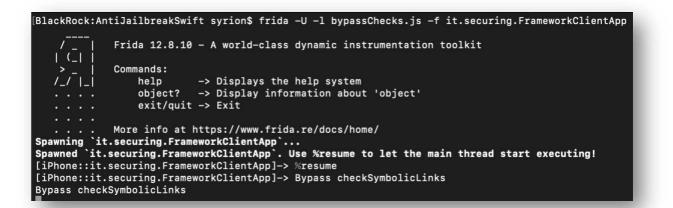


Figure 13 - Frida Script

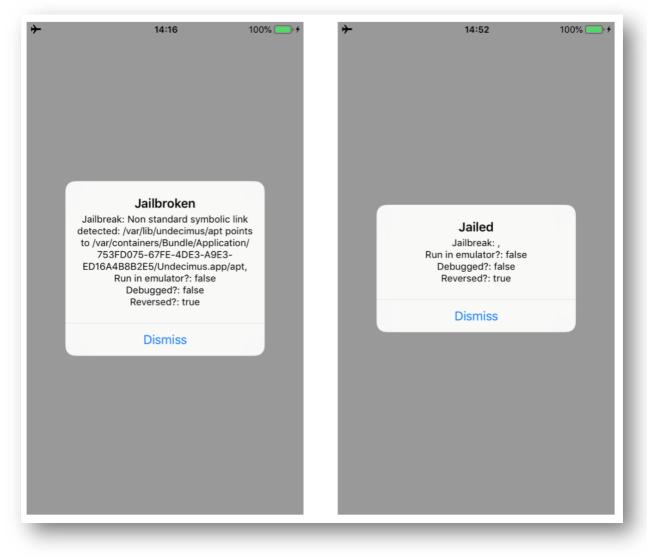


Figure 14 - Security Checks

We successfully bypassed all the Anti-Jailbreak checks.

Bypass amIReversed

Function: _\$s16IOSSecuritySuiteAAC20amIReverseEngineeredSbyFZ

The method **amIReversed** checks if the application is debugged or tampered using instrumentation tools like Frida. Again, we can change its return value contained in the **x0** register, before the **RET** instruction is executed at the offset address **0xaea8**.

	_\$s16I0SSecu	ritySuiteAAC20amIReverseEngineeredSbyFZ:	<pre>// static IOSSecuritySuite.IOSSecurit</pre>
000ae64	sub	sp, sp, #0x30	
00ae68	stp	x20, x19, [sp, #0x10]	
100ae6c	stp	x29, x30, [sp, #0x20]	
00ae70	add	x29, sp, #0x20	
00ae74	movz	×8, #0×0	
00ae78	str	x8, [sp, #0x20 + var_18]	
100ae7c	str	x20, [sp, #0x20 + var_18]	
100ae80	mov	x0, x8	
00ae84	bl	\$s16I0SSecuritySuite30ReverseEngineeringT	oolsCheckerCMa ; type metadata accessor
00ae88	mov	x20, x0	
100ae8c	str	x1, [sp, #0x20 + var_20]	
100ae90	bl	\$s16I0SSecuritySuite30ReverseEngineeringT	oolsCheckerC20amIReverseEngineeredSbyFZ
00ae94	mov	x2, x0	
00ae98	and	w0, w0, #0×1	
00ae9c	ldp	x29, x30, [sp, #0x20]	
100aea0	ldp	x20, x19, [sp, #0x10]	
00aea4	add	sp, sp, #0x30	
00aea8	ret		
	; endp		

Figure 15 - amIReverseEngineered

addr = ptr(0xaea8);	
moduleBase = Module.getBaseAddress(targetModule);	
targetAddress = moduleBase.add(addr);	
Interceptor.attach(targetAddress, {	
onEnter: function(args) {	
if(this.context.x0 == 0x01){	
this.context.x0=0x00	
console.log("Bypass amIReverseEngineered");	
}	
},	
});	

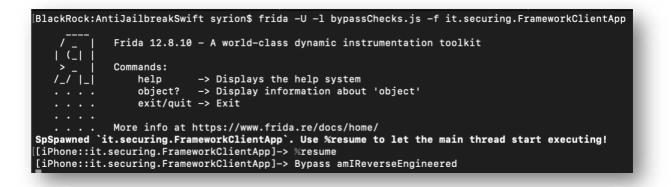


Figure 16 - Frida Script

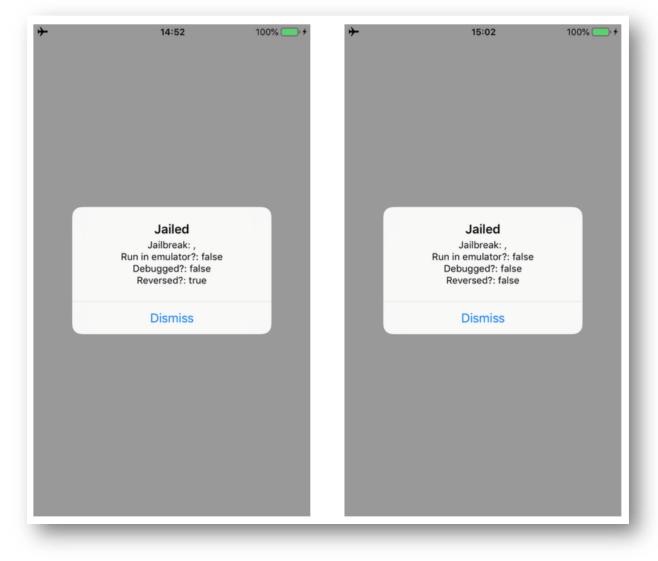


Figure 17 - Security Checks

Bypass AmIDebugged

Function: _\$s16IOSSecuritySuiteAAC11amIDebuggedSbyFZ

	_\$s16I0SSecu	ritySuiteAAC11amIDebuggedSbyFZ:	<pre>// static IOSSecurityS</pre>
000adc4	sub	sp, sp, #0x30	
000adc8	stp	x20, x19, [sp, #0x10]	
000adcc	stp	x29, x30, [sp, #0x20]	
00add0	add	x29, sp, #0x20	
000add4	movz	×8, #0×0	
000add8	str	x8, [sp, #0x20 + var_18]	
000addc	str	x20, [sp, #0x20 + var_18]	
000ade0	mov	x0, x8	
000ade4	bl	\$s16IOSSecuritySuite15Debugger(<pre>CheckerCMa ; type metadat</pre>
000ade8	mov	x20, x0	
aaaadac	e+ P	v1 [cn #0v20 + var 20]	
000adf0	bl	\$s16I0SSecuritySuite15Debugger(CheckerC11amIDebuggedSbyFZ
000ad†4	mov	X2, X0	
000adf8	and	w0, w0, #0×1	
000adfc	ldp	x29, x30, [sp, #0x20]	
000ae00	ldp	x20, x19, [sp, #0x10]	
000ae04	add	sp, sp, #0x30	
000ae08	ret		
	; endp		

Figure 18 - AmIDebugged

The function AmIDebugged checks if the application is debugged, following the method call, we can see there is another method

_\$s16IOSSecuritySuite15DebuggerCheckerC11amIDebuggedSbyFZ looking for a debugger using sysctl and getPid.

	loc_6ae0:		
6ae0	ldr	x0, [sp, #0x710 + var_520]	; CODE XREF=_\$s16IOSSecuritySuite1
6ae4	bl	<pre>impstubsswift_unknownObjectRetain</pre>	; swift_unknownObjectRetain
6ae8	ldr	x8, [sp, #0x710 + var_300]	
6aec	add	x9, sp, #0x410	
6af0	str	x0, [sp, #0x710 + var_540]	
6af4	mov	x0, x9	
6af8	str	x8, [sp, #0x710 + var_548]	
6afc	bl	_\$sSpys5Int32VGW0h	; outlined destroy of Swift.Unsafe
6b00	ldr	x1, [sp, #0x710 + var_520]	
6b04	str	x0, [sp, #0x710 + var_550]	
6b08	mov	x0, x1	
6b0c	bl	<pre>impstubsswift_unknownObjectRelease</pre>	; swift_unknownObjectRelease
6b10	ldr	x0, [sp, #0x710 + var_488]	
6b14	bl	<pre>impstubsswift_bridgeObjectRelease</pre>	; swift_bridgeObjectRelease
6b18	add	x8, sp, #0x470	
6b1c	add	x9, sp, #0x460	
6b20	ldr	x0, [sp, #0x710 + var_548]	
6b24	ldr	w1, [sp, #0x710 + var_42C]	
6b28	mov	x2, x8	
6b2c	mov	x3, x9	
6b30	movz	×8, #0×0	
6b34	mov	x4, x8	
6b38	movz	×8, #0×0	
6b3c	mov	x5, x8	
6b40	bl	imp stubs sysctl	; sysctl
6b44	ldr	x2, [sp, #0x710 + var_520]	
6b48	str	w0, [sp, #0x710 + var_554]	
6b4c	mov	x0, x2	
6b50	bl	<pre>impstubsswift_unknownObjectRelease</pre>	; swift_unknownObjectRelease
6b54	ldr	w10, [sp, #0x710 + var_554]	

Figure 19 - sysctl

If we try to debug the application, the check will return true but we can hook the ret instruction at address **0xae08** and change the return value contained in **x0** to **0x00** with the following Frida script.

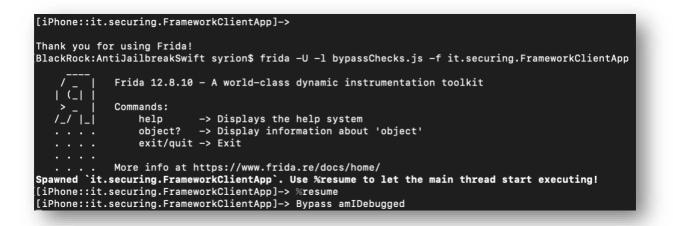


Figure 20 - Frida Script

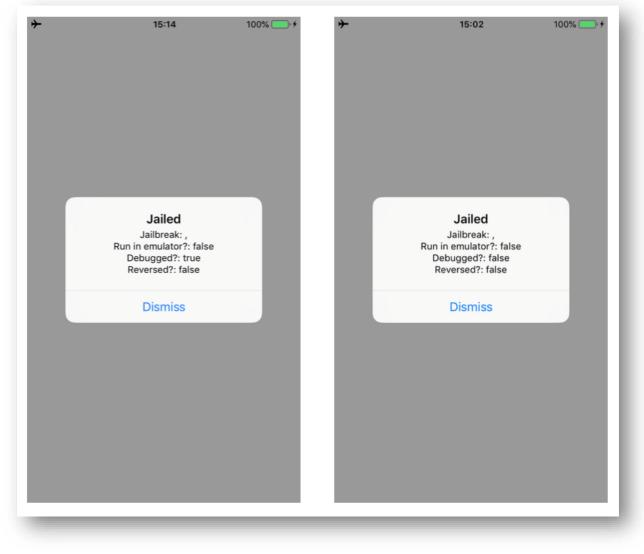


Figure 21 - Security Checks

Bypass amIRunInEmulator

Function: _\$s16IOSSecuritySuite15EmulatorCheckerC08amIRunInC0SbyFZ

Because the application is running on a real iPhone, the **amIrunInEmulator** check can't be triggered, therefore what we can do is to inject a true value and let the application believe that is running in an emulator. The image below shows the method. As always, we have to change the **x0** register value before the **RET** instruction at the offset address **0xa880** is executed.

0a814	_\$5161055ect	<pre>iritySuite15EmulatorCheckerC08amIRunInC sp, sp, #0x40</pre>	<pre>0SbyFZ: // static IOSSecuritySuite.EmulatorChecker.amI ; End of unwind block (FDE at 0x17a48), CODE XREF=_\$</pre>
0a818	stp	x20, x19, [sp, #0x20]	
0a81c	stp	x29, x30, [sp, #0x30]	
0a820	add	x29, sp, #0x30	
0a824	add	x8, sp, #0x18	
0a828 0a82c	mov	x0, x8 w9. #0x0	
0a620 0a830	movz uxtb	w9, #0X0 w1, w9	
0a834	movz	×2, #0×8	
0a838	str	x20, [sp, #0x30 + var_20]	
0a83c	bl	imp stubs memset	: memset
0a840	ldr	x8, [sp, #0x30 + var 20]	
0a844	str	x8, [sp, #0x30 + var 18]	
0a848	mov	x20, x8	
0a84c	bl	_\$s16I0SSecuritySuite15EmulatorChec	kerC12checkCompile33_37AA336693AE6C493BA3A8BB4A701225LLSbyFZ
0a850	tbz	w0, 0x0, loc_a860	
0a854	movz	w8, #0×1	
0a858	str	w8, [sp, #0x30 + var 24]	
0a85c	b		
0a860	loc_a860: ldr	x20, [sp, #0x30 + var_20]	; CODE XREF=_\$s16IOSSecuritySuite15EmulatorCheckerC0
0a864	bl		kerC12checkRuntime33 37AA336693AE6C493BA3A8BB4A701225LLSbyFZ
0a868	str	w0. [sp. #0x30 + var 24]	xelc12clieckkullc1iie32_37kH330033KE0C433DK3K0DD4K701223EE3Dy12
	loc_a86c:	o ["o co o ci]	
0a86c	ldr	w8, [sp, #0x30 + var_24]	; CODE XREF=_\$s16IOSSecuritySuite15EmulatorCheckerC0
0a870 0a874	and	w0, w8, #0x1	
0a874 0a878	ldp ldp	x29, x30, [sp, #0x30] x20, x19, [sp, #0x20]	
0a070 0a87c	add	sp, sp, #0x40	
0a880	ret	sh! sh! #0X40	
	; endp		

Figure 22 - amIRunInEmulator

We can "enable" the emulator with the following Frida script.

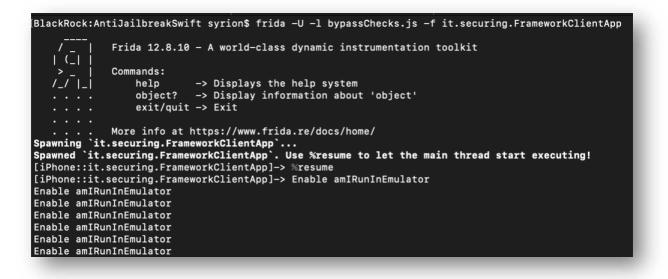


Figure 23 - Frida Script

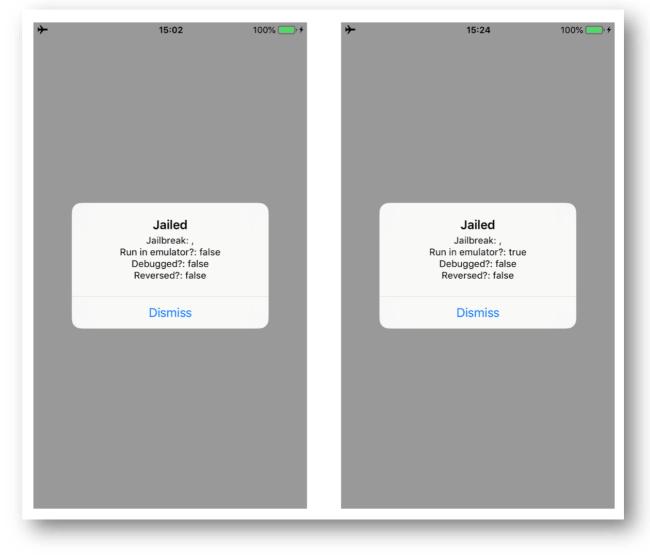


Figure 24 - Security Checks

Change the "Jailed" string

Finally, I wanted to change "Jailed" string for fun. However, its address is no writable so we can't modify its value. What we can do, is creating our own string and put its address in the register that point to the string message. In order to do so, we need to reverse the FrameworkClientApp module. In the

_\$s18FrameworkClientApp14ViewControllerC13viewDidAppearyySbF methods, we can find the instruction where the "Jailed" string address is put into the x0 register. We can see it at address 0x4348.

004340 004344	adrp add	×0, #0×100006000 ×0, ×0, #0×550	: 0x100006550@PAGE, CODE XREF= \$s18Frame : 0x100006550@PAGE0FF, "Jailed"
004348	movz	w8, #0x6	, extended550(i AdEorr, 541(cd
00434c	mov	x1, x8	
004350	movz	w2, #0×1	
004354	bl		ringLiteral17utf8CodeUnitCount7isASCIISSBp_BwBi1_tcfC ;
004358	str	x0, [sp, #0x2f0 + var 138]	
00435c	str	x1, [sp, #0x2f0 + var_140]	

Figure 25 - Jailed String

We can create our own string and put its address into the **x0** register after the instruction at address **0x4348**.



Figure 26 - Security Checks

This is the complete Frida script (which potentially can be written in a better way, so I will let this to you as exercise).

Complete Frida Script

```
var targetModule = 'IOSSecuritySuite';
var addr = ptr(0x126c0);
var moduleBase = Module.getBaseAddress(targetModule);
var targetAddress = moduleBase.add(addr);
  Interceptor.attach(targetAddress, {
       onEnter: function(args) {
                if(this.context.x0 == 0x01) {
                    this.context.x0=0x00
                    console.log("Bypass canOpenUrlFromList");
            }
        },
    });
addr = ptr(0x100ac);
moduleBase = Module.getBaseAddress(targetModule);
targetAddress = moduleBase.add(addr);
   Interceptor.attach(targetAddress, {
        onEnter: function(args) {
                if (this.context.x0 == 0x01) {
                    this.context.x0=0x00
                    console.log("Bypass checkExistenceOfSuspiciousFiles");
```

```
},
    });
addr = ptr(0x10650);
moduleBase = Module.getBaseAddress(targetModule);
targetAddress = moduleBase.add(addr);
    Interceptor.attach(targetAddress, {
        onEnter: function(args) {
                if (this.context.x0 == 0x01) {
                    this.context.x0=0x00
                    console.log("Bypass checkSuspiciousFilesCanBeOpened");
                }
        },
    });
addr = ptr(0x118b0);
moduleBase = Module.getBaseAddress(targetModule);
targetAddress = moduleBase.add(addr);
    Interceptor.attach(targetAddress, {
        onEnter: function(args) {
                if(this.context.x0 != 0x00) {
                    this.context.x0 = 0x00
                    console.log("Bypass checkSymbolicLinks");
                }
       },
    });
addr = ptr(0xaea8);
moduleBase = Module.getBaseAddress(targetModule);
targetAddress = moduleBase.add(addr);
    Interceptor.attach(targetAddress, {
        onEnter: function(args) {
                if(this.context.x0 == 0x01) {
                   this.context.x0=0x00
                    console.log("Bypass amIReverseEngineered");
                }
        },
    });
addr = ptr(0xae08);
moduleBase = Module.getBaseAddress(targetModule);
targetAddress = moduleBase.add(addr);
    Interceptor.attach(targetAddress, {
        onEnter: function(args) {
                if (this.context.x0 == 0x01) {
                   this.context.x0=0x00
                    console.log("Bypass amIDebugged");
                }
        },
    });
addr = ptr(0xa880);
moduleBase = Module.getBaseAddress(targetModule);
targetAddress = moduleBase.add(addr);
    Interceptor.attach(targetAddress, {
        onEnter: function(args) {
                if(this.context.x0 == 0x00) {
```

Conclusion

As we have seen, Frida provides a very powerful way to attach each arm instruction and interact with register and memory. Its documentation is very clear and full of examples. I hope you enjoined it. If you find any mistakes in my write-up, please contact me and I will be more than happy to fix them.