Debugging iOS Applications with IDA Pro

Table of Contents

 2. Getting Started. 2.1. Preparing a Debugging Environment. 3. Source Level Debugging. 4. Debugging DYLD. 	1
3. Source Level Debugging	1
	2
4. Debugging DYLD	4
	6
5. Debugging the DYLD Shared Cache	10
5.1. Initial Analysis	10
5.2. Debugger Configuration	12
5.3. Further Analysis	13
6. Debugging System Applications	14
6.1. Patching the debugserver	14
6.2. IDA Configuration	
6.3. Conclusion	17
7. Troubleshooting	
8. Notes	18

Last updated on June 25, 2020 - v0.1

1. Overview

This tutorial discusses optimal strategies for debugging native iOS applications with IDA Pro.

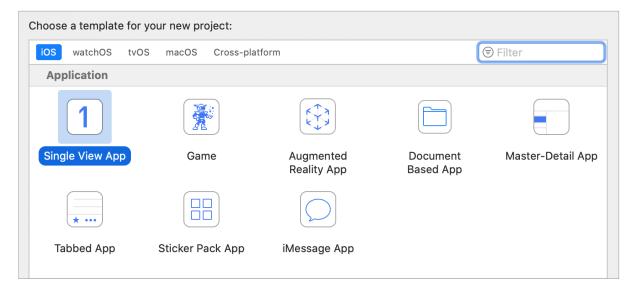
IDA Pro supports remote debugging on any iOS version since iOS 9 (including iPadOS). Debugging is generally device agnostic so it shouldn't matter which hardware you're using as long as it's running iOS. The debugger itself can be used on any desktop platform that IDA supports (Mac/Windows/Linux), although using the debugger on Mac makes more features available.

Note that IDA supports debugging on both jailbroken and non-jailbroken devices. Each environment provides its own unique challenges and advantages, and we will discuss both in detail in this writeup.

2. Getting Started

The quickest way to get started with iOS debugging is to use Xcode to install a sample app on your device, then switch to IDA to debug it.

In this example we'll be using an iPhone SE 2 with iOS 13.4 (non-jailbroken) while using IDA 7.5 SP1 on OSX 10.15 Catalina. Start by launching Xcode and use menu **File>New>Project...** to create a new project from one of the iOS templates, any of them will work:



After selecting a template, set the following project options:

Product Name:	idatest
Team:	None
Organization Name:	
Organization Identifier:	primer
Bundle Identifier:	primer.idatest
Language:	Objective-C
l la su luta ufa sa	
User Interface:	Storyboard 🗘

Note the bundle identifier **primer.idatest**, it will be important later. For the **Team** option choose the team associated with your iOS Developer account, and click OK. Before building be sure to set the target device in the top left of the Xcode window:

		► ► idatest > ■ iPhone SE 2
--	--	-----------------------------

Now launch the build in Xcode. If it succeeds then Xcode will install the app on your device automatically.

2.1. Preparing a Debugging Environment

Now that we have a test app installed on our device, let's prepare to debug it. First we must ensure that the iOS debugserver is installed on the device. Since our device is not jailbroken, this is not such a trivial task. By default iOS restricts all remote access to the device, and such operations are managed by special MacOS Frameworks.

Fortunately Hex-Rays provides a solution. Download the ios_deploy utility from our downloads page. This is a commandline support utility that can perform critical tasks on iOS devices without requiring a jailbreak. Try running it with the **listen** phase. If ios_deploy can detect your device it will print a message:

Use the mount phase to install DeveloperDiskImage.dmg, which contains the debugserver:

\$ export DEVELOPER=/Applications/Xcode.app/Contents/Developer \$ export DEVTOOLS=\$DEVELOPER/Platforms/iPhoneOS.platform/DeviceSupport \$ ios_deploy mount -d \$DEVTOOLS/13.4/DeveloperDiskImage.dmg

The device itself is now ready for debugging. Now let's switch to IDA and start configuring the debugger. Load the **idatest** binary in IDA, Xcode likely put it somewhere in its **DerivedData** directory:

\$ alias ida64="/Applications/IDA\ Pro\ 7.5\ sp1/ida64.app/Contents/MacOS/ida64"
\$ export XCDATA=~/Library/Developer/Xcode/DerivedData
\$ ida64 \$XCDATA/idatest/Build/Products/Debug-iphoneos/idatest.app/idatest

Then go to menu Debugger>Select debugger... and select Remote iOS Debugger:

	👧 Select a de	ebugger
Available deb	ouggers	
Remot	ougger e GDB debugger e XNU debugger e iOS debugger replayer	

When debugging a binary remotely, IDA must know the full path to the executable on the target device. This is another task that iOS makes surprisingly difficult. Details of the filesystem are not advertised, so we must use ios_deploy to retrieve the executable path. Use the **path** phase with the app's bundle ID:

\$ ios_deploy path -b primer.idatest
/private/var/containers/Bundle/Application/<UUID>/idatest.app/idatest

Use this path for the fields in Debugger>Process options...

	👷 Debug application setup: ios
NOTE: all paths	must be valid on the remote computer
<u>Application</u>	ers/Bundle/Application/CD1B48CF-D847-430A-A960-5110F6925C50/idatest.app/idatest 🚬
<u>I</u> nput file	ers/Bundle/Application/CD1B48CF-D847-430A-A960-5110F6925C50/idatest.app/idatest 🔁 💷
<u>P</u> arameters	
	Help Cancel OK

NOTE: the path contains a hex string representing the application's 16-byte UUID. This id is regenerated every time you reinstall the app, so you must update the path in IDA whenever the app is updated on the device.

Now go to Debugger>Debugger options>Set specific options.	and ensure the following fields are s	et:
---	---------------------------------------	-----

	👷 iOS configuration
Ar <u>c</u> hitecture	ARM64 (AArch64)
Ma <u>x</u> packet size	-1
Time <u>o</u> ut	1000
Syslog <u>f</u> lags	
Symbol <u>p</u> ath	/Users/troy/Library/Developer/Xcode/iOS DeviceSupport/13.4 (17E8255)/Symbols
<u>D</u> evice	iPhone SE 2 (iPhone SE 2, iOS 13.4) 🗘 🗸 Launch debugserver automatically
Disable dyld not	tify
	Help Cancel OK

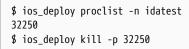
Make special note of the **Symbol path** option. This directory contains symbol files extracted from your device. Both IDA and Xcode use these files to load symbol tables for system libraries during debugging (instead of reading the tables in process memory), which will dramatically speed up debugging.

Xcode likely already created this directory when it first connected to your device, but if not you can always use ios_deploy to create it yourself:

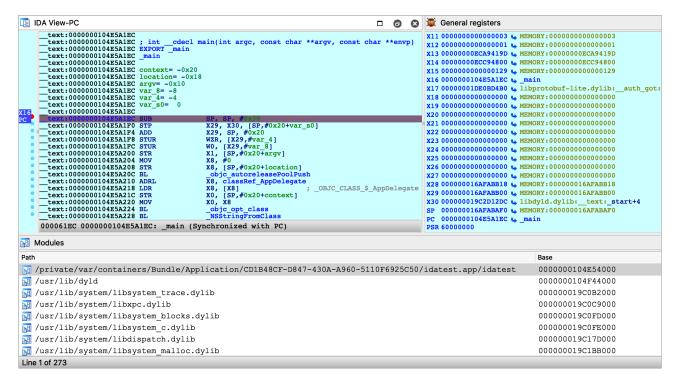
\$ ios_deploy symbols Downloading /usr/lib/dyld Downloading 0.69 MB of 0.69 MB Downloading /System/Library/Caches/com.apple.dyld/dyld_shared_cache_arm64e Downloading 1648.38 MB of 1648.38 MB Extracting symbol file: 1866/1866 /Users/troy/Library/Developer/Xcode/iOS DeviceSupport/13.4 (17E8255)/Symbols: done

Also ensure that the **Launch debugserver automatically** option is checked. This is required for non-jailbroken devices since we have no way to launch the server manually. This option instructs IDA to establish a connection to the debugserver itself via the MacOS Frameworks, which will happen automatically at debugging start.

Lastly, Xcode might have launched the test application after installing it. Use the **proclist** phase to retreive the app's pid and terminate it with the **kill** phase:



Finally we are ready to launch the debugger. Go to **main** in IDA's disassembly view, use **F2** to set a breakpoint, then **F9** to launch the process, and wait for the process to hit our breakpoint:



You are free to single step, inspect registers, and read/write memory just like any other IDA debugger.

3. Source Level Debugging

You can also use IDA to debug the source code of your iOS application. Let's rebuild the **idatest** application with the **DWARF with dSYM File** build setting:

Debug Information Format	DWARF with dSYM File \diamondsuit	DWARF with dSYM File 🗘	DWARF with dSYM File \Diamond
Debug	DWARF with dSYM File \diamondsuit		DWARF with dSYM File 🗘
Release	DWARF with dSYM File \diamondsuit		DWARF with dSYM File 🗘

Since the app is reinstalled, the executable path will change. We'll need to update the remote path in IDA:

\$ ios_deploy p	bath -b primer.idatest
	😭 Debug application setup: ios
NOTE: all paths	must be valid on the remote computer
<u>Application</u>	ers/Bundle/Application/DA785612-F361-4BC9-9EDA-6C75702ACC0E/idatest.app/idatest ど 🛛
<u>I</u> nput file	ers/Bundle/Application/DA785612-F361-4BC9-9EDA-6C75702ACC0E/idatest.app/idatest 🕥 📖

Be sure to enable **Debugger>Use source-level debugging**, then launch the process. At runtime IDA will be able to load the DWARF source information:

Source view: /Users/troy/Xce	ode/idatest/idatest/main.m			Θ	Θ
<pre>1 /// main.m 3 // idatest 4 // 5 // import <uikit uikit.h=""> 6 # import 'AppDelegate.h" 8 // int main(int argc, char * argv[]) { 10 NSString * appDelegateClassName; 11 @ eautoreleasepool { 12 // Setup code that might create autoreleased objects goes here. 13 appDelegateClassName = NSStringFromClass([AppDelegate class]); 14 } 15 return UIApplicationMain(argc, argv, nil, appDelegateClassName); 16 }</uikit></pre>					
0000624C 15 (102F0E24	C)				
🔄 Locals (DWARF)	Second State S				
Name	Value	Туре	Location		
argc	1	int	X29-8		
▼ argv	0x16CEF7B28LL:0x16CEF7C6	char **	SP+10		
▶ [0]	0x16CEF7C60LL:"/var/cont	char *	@16CEF'	7в28	
appDelegateClassName	0x2832F4040LL:{baseclass	Foundation::NSString::NSString *	SP+8		
▶ baseclass_0	<pre>{isa=0x1A1E9BADA69LL}</pre>	ObjectiveC::NSObject::NSObject	@2832F	4040	

Note that the debugserver does not provide DWARF information to IDA - instead IDA looks for dSYM bundles in the vicinity of the idb on your local filesystem. Thus if you want IDA to load DWARF info for a given module, both the module binary and its matching dSYM must be in the same directory as the idb, or in the idb's parent directory.

For example, in the case of the idatest build:

\$ tree	
•	
├─── idatest.app	
idatest	
└─── idatest.app.dSYM	
└──── Contents	
Len Resources	
L DWARF	
└─── idatest	

IDA was able to find the idatest binary next to idatest.i64, as well as the dSYM bundle next to the parent app directory.

If IDA can't find DWARF info on your filesystem for whatever reason, try launching IDA with the command-line option **-z440010**, which will enable much more verbose logging related to source-level debugging:

```
Looking for Mach-O file "idatest.app/idatest.dSYM/Contents/Resources/DWARF/idatest"

File "idatest.app/idatest.dSYM/Contents/Resources/DWARF/idatest" exists? -> No.

Looking for Mach-O file "idatest.app.dSYM/Contents/Resources/DWARF/idatest"

File "idatest.app.dSYM/Contents/Resources/DWARF/idatest" exists? -> Yes.

Looking for cpu=16777228:0, uuid=7a09f307-7503-3c0d-a182-ab552c1bf182.

Candidate: cpu=16777228:0, uuid=7a09f307-7503-3c0d-a182-ab552c1bf182.

Found, with architecture #0

DWARF: Found DWARF file "idatest.app.dSYM/Contents/Resources/DWARF/idatest"
```

4. Debugging DYLD

IDA can also be used to debug binaries that are not user applications. For example, dyld.

The ability to debug dyld is a nice advantage because it allows us to observe critical changes in the latest versions of iOS (especially regarding the shared cache) before a jailbreak is even available. We document this functionality here in the hopes it will be useful to others as well.

In this example we'll be using IDA to discover how dyld uses ARMv8.3 Pointer Authentication to perform secure symbol bindings. Start by loading the dyld binary in IDA. It is usually found here:

~/Library/Developer/Xcode/iOS DeviceSupport/13.4 (17E8255)/Symbols/usr/lib/dyld

The target application will be a trivial helloworld program:

```
#include <stdio.h>
int main(void)
{
    puts("hello, world!\n");
    return 0;
}
```

Compile and install this app on your device, then set the following fields in Debugger>Process options...

	👷 Debug application setup: ios	
NOTE: all paths r	must be valid on the remote computer	
<u>Application</u>	ndle/Application/A5701C8F-7CF4-458F-B7CC-43F2963C1C90/helloworld.app/helloworld	
<u>I</u> nput file	/usr/lib/dyld	
<u>P</u> arameters		

Under **Debugger >Debugger options**, enable **Suspend on debugging start**. This will instruct IDA to suspend the process at dyld's entry point, before it has begun binding symbols. Now launch the process with **F9** - immediately the process will be suspended at **__dyld_start**:

1	IDA View-PC		👿 General registers	
20	<pre>text:000000100C81000 text:000000100C81000 text:000000100C81000 text:000000100C81000 text:00000100C81000 text:000000100C81000 text:000000100C81000 text:000000100C81000 var_10=-C text:000000100C81000 var_10=-C text:000000100C81000 var_10=-C text:000000100C8100 NOV text:000000100C8100 NOV text:000000100C8100 NOV text:000000100C8101 NOV text:000000100C8102 LDR text:000000100C8102 LDR text:000000100C8103 NOV text:000000100C8103 NOV text:000000100C8103 NOV</pre>	yld_start rt ; DATA XREF:dyld_find_unwind_ x20 x10 x10 x28, SP SP, X28, #0xFFFFFFFFFFF x0, #0 x1, #0 x1, x0 x1, x	X14 00000000000000 MEMC X15 00000000000000 MEMC X16 0000000000000 MEMC X17 00000000000000 MEMC X19 0000000000000 MEMC X20 00000000000000 MEMC X21 00000000000000 MEMC X22 00000000000000 MEMC X22 00000000000000 MEMC X22 00000000000000 MEMC X24 000000000000000 MEMC X25 000000000000000 MEMC X26 000000000000000 MEMC X28 000000000000000 MEMC X29 0000000000000000 MEMC SP 0000000000000000 MEMC SP 00000000000000000 MEMC SP 00000000000000000 MEMC SP 000000016F43BB28 MEMC PSR 000000016F43BB28 MEMC	RY : 00000000000000000000000000000000000
20	Modules			
Path	1			Base
28	/private/var/containers/Bundle	/Application/A5701C8F-7CF4-458F-B7CC-43F2963C1C90/hell	loworld.app/helloworld	0000001009C4000
	/usr/lib/dyld			0000000100C80000

Double-click on the **helloworld** module to bring up its symbol list and go to the **_main** function:

helloworld:text:00000001009CBF74	
helloworld:text:00000001009CBF74 _main	
helloworld: text:00000001009CBF74	
helloworld: text:00000001009CBF74 var s	0= 0
helloworld: text:00000001009CBF74	
helloworld: text:00000001009CBF74 PACIB	SP
helloworld: text:00000001009CBF78 STP	X29, X30, [SP,#-0x10+var_s0]!
helloworld: text:00000001009CBF7C MOV	X29, SP
helloworld: text:00000001009CBF80 ADR	X0, aHelloWorld ; "hello, world!\n"
helloworld: text:00000001009CBF84 NOP	
helloworld: text:00000001009CBF88 BL	sub_1009CBF98
helloworld: text:00000001009CBF8C MOV	WO, #O
helloworld: text:00000001009CBF90 LDP	X29, X30, [SP+var_s0],#0x10
helloworld: text:00000001009CBF94 RETAB	
helloworld: text:00000001009CBF94 ; End	of function main
helloworld: text: 00000001009CBF94	

Note that function sub_1009CBF98 is the stub for puts:

helloworld:auth_stubs:00000001009CBF98 helloworld:auth_stubs:00000001009CBF98 helloworld:auth_stubs:0000000009CBF98 helloworld:auth_stubs:00000001009CBFA0	sub_1009CBF98 ADRL X17, off_1009CC000	<pre>; CODE XREF: _main+141p ; unk C001000000000000</pre>
helloworld: auth stubs:0000001009CBFA4	BRAA X16, X17	; unk C00100000000000
helloworld:auth_stubs:0000001009CBFA4 helloworld:auth_stubs:0000001009CBFA4	; End of function sub_1009CBF98	

The stub reads a value from off_109CC000, then performs a branch with pointer authentication. We can assume that at some point, dyld will fill off_109CC000 with an authenticated pointer to puts. Let's use IDA to quickly track down this logic in dyld.

The iOS debugger supports watchpoints. Now would be a good time to use one:

```
ida_dbg.add_bpt(0x1009CC000, 8, BPT_WRITE)
```

Resume the process and wait for dyld to trigger our watchpoint:

100CA5DE4 MOV	X0, X19	; this		X17 000000100CA5958
100CA5DE8 MOV	X1, X19	; void *		x18 000000000000000000000000000000000000
100CA5DEC BL		ded25ChainedFixupPc	interOnDisk6Arm64e11signPoir	terEPvy x19 00000001009CC000
100CA5DF0 MOV	<mark>x21</mark> , x0			
100CA5DF4				X20 00000016F439F18
100CA5DF4 loc_100CA5DF4		; CODE XREF:	_ZNK5dyld311MachOLoaded21fix	
100CA5DF4		;ZNK5dyld31	1MachOLoaded21fixupAllChaine	dFixupsE x22 00000001009C4000
100CA5DF4 LDR	X0, [X20,#0x20]			X23 0000001009C4000
100CA5DF8 CBZ	X0, loc 100CA5E10			X23 0000001009C4000
100CA5DFC MOV	x8, x0 -			x24 0000001009CC000
100CA5E00 LDR	X9, [X8,#0x10]!			x25 00000016F439F28
100CA5E04 MOV	X1, X19			X26 00000016F43B0C0
100CA5E08 MOV	X2, <mark>X21</mark>			
100CA5E0C BLRAA	x9, x8			X27 000000100CF1610
100CA5E10				x28 00000000000000000
100CA5E10 loc_100CA5E10		; CODE XREF:	ZNK5dyld311MachOLoaded21fix	upAllCha x29 000000016F439E10
100CA5E10 STR	<mark>X21</mark> , [X19]			x30 A901E18100CA5E10
100CA5E14 B	loc_100CA5E44			
100CA5E18 ;				SP 00000016F439DE0
				PC 000000100CA5E14

The instruction **STR X21 [X19]** triggered the watchpoint, and note the value in X21 (BB457A81BA95ADD8) which is the authenticated pointer to **puts**. Where did this value come from? We can see that X21 was previously set with **MOV X21**, **X0** after a call to this function:

dyld3::MachOLoaded::ChainedFixupPointerOnDisk::Arm64e::signPointer

It seems like we're on the right track. Also note that IDA was able to extract a nice stack trace despite dyld's heavy use of PAC instructions to authenticate return addresses on the stack:

Address	Module	Function
100CA5E14	dyld	ZNK5dyld311MachOLoaded21fixupAllChainedFixups_block_invoke
100CA5EEC	dyld	dyld3::MachOLoaded::walkChain
100CA5BF0	dyld	dyld3::MachOLoaded::forEachFixupInAllChains
100CA5B50	dyld	dyld3::MachOLoaded::fixupAllChainedFixups
100CA2210	dyld	ZN5dyld36Loader18applyFixupsToImage_block_invoke.68
100CB0218	dyld	dyld3::MachOAnalyzer::withChainStarts
100CA2004	dyld	ZN5dyld36Loader18applyFixupsToImage_block_invoke_3
100CB3314	dyld	dyld3::closure::Image::forEachFixup
100CA15EC	dyld	dyld3::Loader::applyFixupsToImage
100CA0A00	dyld	dyld3::Loader::mapAndFixupAllImages
100C88784	dyld	dyld::launchWithClosure
100C86BE0	dyld	dyld::_main
100C81228	dyld	dyldbootstrap::start
100C81034	dyld	dyld_start

This leads us to the following logic in the dyld-733.6 source:

```
// authenticated bind
newValue = (void*)(bindTargets[fixupLoc->arm64e.bind.ordinal]);
if (newValue != 0)
newValue = (void*)fixupLoc->arm64e.signPointer(fixupLoc, newValue);
```

Here, fixupLoc (off_109CC00) and newValue (address of puts) are passed as the loc and target arguments for Arm64e::signPointer:

```
uint64_t discriminator = authBind.diversity;
if ( authBind.addrDiv )
discriminator = __builtin_ptrauth_blend_discriminator(loc, discriminator);
switch ( authBind.key ) {
  case 0: // IA
    return __builtin_ptrauth_sign_unauthenticated(target, 0, discriminator);
  case 1: // IB
    return __builtin_ptrauth_sign_unauthenticated(target, 1, discriminator);
  case 2: // DA
    return __builtin_ptrauth_sign_unauthenticated(target, 2, discriminator);
  case 3: // DB
    return __builtin_ptrauth_sign_unauthenticated(target, 3, discriminator);
}
```

Thus, the pointer to **puts** is signed using its destination address in helloworld:__auth_got as salt for the signing operation. This is quite clever because the salt value is subject to ASLR and therefore cannot be guessed, but at this point the executable has already been loaded into memory — so it won't change by the time the pointer is verified in the stub.

To see this in action, use F4 to run to the BRAA instruction in the stub and note the values of the operands:

1	DA View-PC					Θ	0
	helloworld:auth_s helloworld:auth_s helloworld:auth_s helloworld:auth_s helloworld:auth_s	tubs:00000001009CBF tubs:00000001009CBF tubs:00000001009CBF tubs:00000001009CBF	98 98 sub_1009CBF98 98 ADRL A0 LDR	X17, off X16, [X1	7]	00	
Ţ	helloworld:auth_s helloworld:auth_s helloworld:auth_s UNKNOWN 0000000100	tubs:00000001009CBF	A4 ; End of functio A4		09CBF98		
1	Call Stack					Θ	Θ
Addr	ess	Module	Function				
Ş	00000001009CBFA4	helloworld	sub_1009CBF98+C	;			
i	00000001009CBF88	helloworld	_main+14				
5	00000001BAAED2DC	libdyld.dylib	_start+4				

X16 BB457A81BA95ADD8 L MEMORY:unk_BB457A81BA95ADD8 X17 0000001009CC000 L helloworld: auth got:off_1009CC000

The branch will use the operands to verify that the target address has not been modified after it was originally calculated by dyld. Since we haven't done anything malicious, one more single step should take us right to **puts**:

	DA View-PC							Θ	Θ
	libsystem_c.dylib: libsystem_c.dylib: libsystem_c.dylib: libsystem_c.dylib: libsystem_c.dylib: libsystem_c.dylib: libsystem_c.dylib: libsystem_c.dylib: libsystem_c.dylib:	text:0000001BA95ADD8 text:0000001BA95ADD8 text:0000001BA95ADD8 text:0000001BA95ADD0 text:0000001BA95ADE0 text:0000001BA95ADE4 text:00000001BA95ADE8 text:0000001BA95ADE6 text:0000001BA95ADF4 text:0000001BA95ADF8 A95ADD8: libsystem_c.	PACIBSP SUB STP ADD ADRL LDR STUR	X20, X29, X29, X8, X8, X8,	X30, SP, # sta [X8] [X29,#	[SP,#0 [SP,#0 0x50 ; ck_chk -0x18]	x50] 'P' guard		:)
i	Call Stack		0					Θ	Θ
									_
Addr	ess	Module	Function						-
	ess 00000001BA95ADD8	Module libsystem_c.dylib	Functionputs						
		1							

Just for fun, let's rewind the process back to the start of the stub:

IDC>PC = 0x1009CBF98

Then overwrite the authenticated pointer to puts with a raw pointer to printf:

ida_bytes.put_qword(0x1009CC000, ida_name.get_name_ea(BADADDR, "_printf"))

Now when we step through the stub, the BRAA instruction should detect that the authenticated pointer has been modified, and it will purposefully crash the application by setting PC to an invalid address:

	DA View-PC
20	MEMORY:2000001BA95AB80 % 1 MEMORY:2000001BA95AB81 % 1 MEMORY:2000001BA95AB82 % 1 MEMORY:2000001BA95AB83 % 1 MEMORY:2000001BA95AB84 % 1 MEMORY:2000001BA95AB85 % 1
	MEMORY: 2000001BA95AB86 % 1 MEMORY: 2000001BA95AB87 % 1 MEMORY: 2000001BA95AB88 % 1 MEMORY: 2000001BA95AB88 % 1
	MEMORY: 2000001BA95AB8A % 1 MEMORY: 2000001BA95AB8B % 1 MEMORY: 2000001BA95AB8B % 1 MEMORY: 2000001BA95AB8C % 1

Any attempt to resume execution will inevitably fail:



It seems we now have an understanding of secure symbol bindings in dyld. Fascinating!

5. Debugging the DYLD Shared Cache

This section discusses how to optimally debug system libraries in a dyld_shared_cache.

NOTE: full support for dyld_shared_cache debugging requires IDA 7.5 SP1

Debugging iOS system libraries is a challenge because the code is only available in the dyld cache. IDA allows you to load a library directly from the cache, but this has its own complications. A single module typically requires loading several other modules before the analysis becomes useful. Fortunately IDA is aware of these annoyances and allows you to debug such code with minimal effort.

To start, consider the following sample application that uses the CryptoTokenKit framework:

```
#import <CryptoTokenKit/CryptoTokenKit.h>
int main(void)
{
    TKTokenWatcher *watcher = [[TKTokenWatcher alloc] init];
    NSArray *tokens = [watcher tokenIDs];
    for ( int i = 0; i < [tokens count]; i++ )
        printf("%s\n", [[tokens objectAtIndex:i] UTF8String]);
    return 0;
}</pre>
```

Assume this program has been compiled and installed on the device as ctk.app.

Instead of debugging the test application, let's try debugging the CryptoTokenKit framework itself - focusing specifically on the **-[TKTokenWatcher init]** method.

5.1. Initial Analysis

First we'll need access to the dyldcache that contains the CryptoTokenKit framework. The best way to obtain the cache is to extract it from the ipsw package for your device/iOS version. This ensures that you are working with the original untouched cache that was installed on your device.

When opening the cache in IDA, choose the load option **Apple DYLD cache for arm64e (single module)** and select the CryptoTokenKit module:

Choose a module to load	
File name	Address
/System/Library/Frameworks/AutomaticAssessmentConfiguration.framework/AutomaticA	0x1B8176000
/System/Library/Frameworks/AutomaticAssessmentConfiguration.framework/Frameworks	0x1B8178000
/System/Library/Frameworks/CryptoTokenKit.framework/CryptoTokenKit	0x1B8181000
/System/Library/Frameworks/MediaToolbox.framework/Support/libSTS-N.dylib	0x1B81BD000
/System/Library/Frameworks/SwiftUI.framework/SwiftUI	0x1B81C2000

Wait for IDA to finish the initial analysis of CryptoTokenKit. Immediately we might notice that the analysis suffers because of references to unloaded code. Most notably many Objective-C methods are missing a prototype, which is unusual:

text:00000001B81ADCF4 text:00000001B81ADCF4 text:00000001B81ADCF4 text:00000001B81ADCF4	; -[TKTokenWatc] TKTokenWatche:		;	; DATA XREF: CryptoTokenKit:objc_con	st:
text:00000001B81ADCF4 text:00000001B81ADCF4 text:00000001B81ADCF4	var_10 var_s0	= -0x10 = 0			
		PACIBSP STP STP		<pre>9, [SP,#-0x10+var_10]! 0, [SP,#0x10+var_s0]</pre>	

However this is expected. Modern dyld caches store all Objective-C class names and method selectors inside the libobjc module. Objective-C analysis is practically useless without these strings, so we must load the libobjc module to access them. Since a vast majority of modules depend on libobjc in such a way, it is a good idea to automate this in a script.

For a quick fix, save the following idapython code as init.py:

```
# improve functions with branches to unloaded code
idaapi.cvar.inf.af &= ~AF_ANORET
def dscu_load_module(module):
    node = idaapi.netnode()
    node.create("$ dscu")
    node.supset(2, module)
    load_and_run_plugin("dscu", 1)
# load libobjc, then analyze objc types
dscu_load_module("/usr/lib/libobjc.A.dylib")
load_and_run_plugin("objc", 1)
```

Then reopen the cache with:

\$ ida64 -Sinit.py -Oobjc:+1 dyld_shared_cache_arm64e

This will tell IDA to load libobjc immediately after the database is created, then perform the Objective-C analysis once all critical info is in the database. This should make the initial analysis acceptable in most cases. In the case of CryptoTokenKit, we see that the Objective-C prototypes are now correct:

text:0000001B81ADCF4		
text:0000001B81ADCF4 ; NSXPCListene	rEndpoint *c	_cdecl -[TKTokenWatcher endpoint](TKTokenWatcher *self, SEL)
text:0000001B81ADCF4 TKTokenWatch	er endpoint	; DATA XREF: CryptoTokenKit: objc const:000000
text:0000001B81ADCF4		
text:0000001B81ADCF4 var_10	= -0x10	
text:0000001B81ADCF4 var s0	= 0	
text:0000001B81ADCF4		
text:0000001B81ADCF4	PACIBSP	
text:0000001B81ADCF8	STP	X20, X19, [SP,#-0x10+var_10]!
text:0000001B81ADCFC	STP	x29, x30, [SP, #0x10+var s0]

Now let's go to the -[TKTokenWatcher init] method invoked by the ctk application:

text:0000001B81ADC54		
text:00000001B81ADC54	; TKTokenWatcher * cdecl	-[TKTokenWatcher init](TKTokenWatcher *self, SEL)
text:00000001B81ADC54	TKTokenWatcher init	; DATA XREF: CryptoTokenKit: objc const:0000000
text:00000001B81ADC54	ADRP	X8, #sel initWithClient @PAGE ; "initWithClient:"
text:0000001B81ADC58	ADD	X1, X8, #sel initWithClient @PAGEOFF ; "initWithClient:"
text:0000001B81ADC5C	MOV	x2, #0
text:0000001B81ADC60	В	0x1B271C01C
text:00000001B81ADC60	; End of function -[TKToke	enWatcher init]
text:00000001B81ADC60	·	-

If we right-click on the unmapped address **0x1B271C01C**, IDA provides two options in the context menu:

Load ProVideo:__auth_stubs Load ProVideo

In this case the better option is **Load ProVideo:__auth_stubs**, which loads only the stubs from the module and properly resolves the names:

text:00000001B81ADC54		
text:0000001B81ADC54		<pre>kenWatcher init](TKTokenWatcher *self, SEL)</pre>
text:0000001B81ADC54	TKTokenWatcher init	; DATA XREF: CryptoTokenKit: objc const:0000000
text:00000001B81ADC54		X8, #sel_initWithClient_@PAGE ; "initWithClient:"
text:0000001B81ADC58	ADD	X1, X8, #sel_initWithClient_@PAGEOFF ; "initWithClient:"
text:0000001B81ADC5C	MOV	X2, #0
text:0000001B81ADC60	В	j objc msgSend
text:0000001B81ADC60	; End of function -[TKTokenWatch	er init]
text:0000001B81ADC60	· ·	-

This is a common pattern in the latest arm64e dyldcaches, and it is quite convenient for us. Loading a handful of __auth_stubs sections is enough to resolve most of the calls in CryptoTokenKit, which gives us some nice analysis for **-[TKTokenWatcher init]** and its helper method:

3		Pseudocode-A
	12	TKTokenWatcher *cdecl -[TKTokenWatcher init](TKTokenWatcher *self, SEL a2) {
•		<pre>return -[TKTokenWatcher initWithClient:](self, "initWithClient:", 0LL);</pre>
•	4	}

E Pseudocode-A

	TKTokenWatcher *cdecl -[TKTokenWatcher initWithClient:](TKTokenWatcher *self, SEL a2, id client)
	{ 3 // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
•	client = jobjc_retain(client);
• •	super.receiver = self;
•	<pre>/ super_super_class = &OBJC_CLASSTKTokenWatcher;</pre>
• 8	
• •	
1(
• 1:	
12	
• 13	
• 14	
15	
10	else
1	
• 18	
• 19	
• 20	
21	
• 22	
• 2:	
• 24	j_objc_release(_client);

5.2. Debugger Configuration

Now that the static analysis is on par with a typical iOS binary, let's combine it with dynamic analysis. We can debug this database by setting the following options in **Debugger>Process options**:

	🕒 🔘 🔹 🧕 👷 Debug application setup: ios		
NOTE: all paths must be valid on the remote computer			
<u>Application</u>	containers/Bundle/Application/C4118B7F-DC92-4D92-9E46-1AB33D65905C/ctk.app/ctk 💙		
<u>I</u> nput file	/System/Library/Frameworks/CryptoTokenKit.framework/CryptoTokenKit		

Here we set the **Input file** field to the full path of the CryptoTokenKit module. This allows IDA to easily detect the dyldcache slide at runtime. When CryptoTokenKit is loaded into the process, IDA will compare its runtime load address to the imagebase in the current idb, then rebase the database accordingly.

By default the imagebase in the idb corresponds to the first module that was loaded:

IDC>msg("%a", get_imagebase())
CryptoTokenKit:HEADER:0000001B8181000

Thus, it is easiest to set Input file to the module corresponding to the default imagebase.

Note however that we could also use this configuration:

	👷 Debug application setup: ios		
NOTE: all paths must be valid on the remote computer			
Application	containers/Bundle/Application/C4118B7F-DC92-4D92-9E46-1AB33D65905C/ctk.app/ctk 💙		
<u>I</u> nput file	/usr/lib/libobjc.A.dylib		

Provided that we update the imagebase in the idb to the base of the libobjc module:

```
ida_nalt.set_imagebase(ida_segment.get_segm_by_name("libobjc.A:HEADER").start_ea)
```

This will result in the same dyld slide and should work just as well, because the the imagebase and the **Input file** field both correspond to the same module. This is something to keep in mind when debugging dyldcache idbs that contain multiple libraries.

Now let's try launching the debugger. Set a breakpoint at -[TKTokenWatcher initWithClient:], use F9 to launch the process, then wait for our breakpoint to be hit:

IDA was able to map our database (including CryptoTokenKit, libobjc, and the satellite __auth_stubs sections) into process memory. We can single step, resume, inspect registers, and perform any other operation that is typical of an IDA debugging session.

5.3. Further Analysis

Note that after terminating the debugging session you can continue to load new modules from the cache. If a dyld slide has been applied to the database, new modules will be correctly loaded into the rebased address space. This did not work in previous versions of IDA.

For example, after a debugging session we might notice some more unresolved calls:

CryptoTokenKit: text:00000001F2A3DD64	MOV	X19, X0
CryptoTokenKit: text:0000001F2A3DD68	LDR	WO, [XO,#8]
CryptoTokenKit: text:0000001F2A3DD6C	BL	0x1EFF0CB20
CryptoTokenKit: text:00000001F2A3DD70	LDR	X0, [X19,#0x10] ; id
CryptoTokenKit: text:00000001F2A3DD74	ADRP	X8, #sel_invalidate@PAGE ; "invalidate"

IDA is aware that the address space has shifted, and it will load the new code at the correct address:

CryptoTokenKit: text:0000001F2A3DD64	MOV	X19, X0
CryptoTokenKit: text:0000001F2A3DD68	LDR	WO, [XO,#8]
CryptoTokenKit: text:00000001F2A3DD6C	BL	j notify cancel 0
CryptoTokenKit: text:0000001F2A3DD70	LDR	X0, [X19,#0x10] ; id
CryptoTokenKit: text:0000001F2A3DD74	ADRP	<pre>X8, #sel invalidate@PAGE ; "invalidate"</pre>

You are free to load new modules and relaunch debugging sessions indefinitely.

6. Debugging System Applications

The previous examples used custom applications to demonstrate IDA's debugging capabilities. In this case IDA can utilize the debugserver included in Apple's iOS developer tools, but there are situations in which this server is not sufficient for our needs.

The debugserver will refuse to debug any application that we didn't build ourselves. To demonstrate this, try launching IDA with an empty database and use **Debugger>Attach>Remote iOS Debugger** to attach to one of the system daemons:

	Choose process to attach to
ID	Name
36476	sensorkitd
36384	biometrickitd
36433	maild
36338	contextstored
36482	itunesstored
51	identityservicesd
36436	com.apple.sbd
21499	locationd
Line 74 of 227	,

You will likely get this error message:

0	The debugger could not attach to the selected process. This can perhaps indicate the process was just terminated, or that you don't have the necessary privileges.
	ОК

It is possible to install a custom version of the debugserver that can debug system processes, but this requires a jailbroken device. We document the necessary steps and IDA configuration here. The device used in this example is an iPhone 8 with iOS 13.2.2, jailbroken with checkra1n 0.10.1.

6.1. Patching the debugserver

First we must obtain a copy of the debugserver binary from the DeveloperDiskImage.dmg:

- \$ export DEVELOPER=/Applications/Xcode.app/Contents/Developer
- \$ export DEVTOOLS=\$DEVELOPER/Platforms/iPhoneOS.platform/DeviceSupport
- \$ hdiutil mount \$DEVTOOLS/13.2/DeveloperDiskImage.dmg
- \$ cp /Volumes/DeveloperDiskImage/usr/bin/debugserver .

Now save the following xml as entitlements.plist:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/ PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
<key>task_for_pid-allow</key> <true/>
<key>get-task-allow</key> <true/>
<key>platform-application</key> <true/>
<key>com.apple.springboard.debugapplications</key> <true/>
<key>com.apple.system-task-ports</key> <true/>
</dict>
</dict>
</dict>
```

Then use Idid to codesign the server:

```
$ ldid -Sentitlements.plist debugserver
```

This will grant the debugserver permission to debug any application, including system apps. Now we can copy the server to the device and run it:

```
$ scp debugserver root@iphone-8:/usr/bin/
$ ssh root@iphone-8
iPhone-8:~ root# /usr/bin/debugserver 192.168.1.7:1234
debugserver-@(#)PROGRAM:LLDB PROJECT:lldb-900.3.98 for arm64.
Listening to port 1234 for a connection from 192.168.1.7...
```

Note that we specified **192.168.1.7** which is the IP of the host machine used in this example. Be sure to replace this with the IP of your host so that the server will accept incoming connections from IDA.

6.2. IDA Configuration

To enable debugging with the patched debugserver, set the following options in dbg_ios.cfg:

```
// don't launch the debugserver. we did it manually
AUTOLAUNCH = N0
// your device's UUID. this is used when fetching the remote process list
DEVICE_ID = "";
// debugging symbols extracted by Xcode
SYMBOL_PATH = "~/Library/Developer/Xcode/iOS DeviceSupport/13.2.2 (17B102)/Symbols";
```

We're now ready to open a binary in IDA and debug it. Copy the **itunesstored** binary from your device, it is typically found here:

/System/Library/PrivateFrameworks/iTunesStore.framework/Support/itunesstored

After loading the binary use **Debugger>Select debugger** and choose **Remote iOS Debugger**, then under **Debugger>Process options** set the following fields:

	👷 Debug application setup: ios		
NOTE: all paths must be valid on the remote computer			
<u>Application</u>	/System/Library/PrivateFrameworks/iTunesStore.framework/Support/itunesstored		
Input file	/System/Library/PrivateFrameworks/iTunesStore.framework/Support/itunesstored		
<u>P</u> arameters			
<u>H</u> ostname	iphone-8 Po <u>r</u> t 1234 V		

Since we set **AUTOLAUNCH = NO**, IDA now provides the **Hostname** and **Port** fields so we can specify how to connect to our patched debugserver instance.

Now use **Debugger>Attach to process** and choose **itunesstored** from the process list. Since we have modified the debugserver it should agree to debug the target process, allowing IDA to create a typically robust debugging environment:

IDA View-PC	👿 General registers	
<pre>libsystem_kernel.dylib:_text:0000001A201862C _mach_msg_trap DCB 0xD0 libsystem_kernel.dylib:_text:0000001A201862D DCB 3 libsystem_kernel.dylib:_text:0000001A201862D DCB 3 libsystem_kernel.dylib:_text:0000001A201863D DCB 0x80 libsystem_kernel.dylib:_text:0000001A201863D DCB 1 libsystem_kernel.dylib:_text:0000001A201863D DCB 0x10 libsystem_kernel.dylib:_text:0000001A201863D DCB 0x10 libsystem_kernel.dylib:_text:0000001A201863D DCB 0x10 libsystem_kernel.dylib:_text:0000001A2018633 DCB 0x14 libsystem_kernel.dylib:_text:0000001A2018634 ;</pre>	X19 0000000000000 MEMORY:000000000000000000000000000000000000	
B Modules		
Path	Base	
🔢 /System/Library/PrivateFrameworks/iTunesStore.framework/Support	t/itunesstored 0000001045E0000	
🔢 /usr/lib/libobjc-trampolines.dylib	000000104B4C000	
🔢 /usr/lib/dyld	000000104D2C000	
<pre>/usr/lib/system/libsystem_trace.dylib</pre>	0000001A1DEC000	
usr/lib/system/libxpc.dylib	0000001A1E03000	
Line 1 of 535		

Note that although we're not using the debugserver from DeveloperDiskImage.dmg, IDA still depends on other developer tools to query the process list. We discuss how to install the DeveloperDiskImage in the Getting Started section above, but for a quick workaround you can always just specify the PID manually:

	Choose process to attach to
ID	A Name
45050	AppStore
45052	com.apple.accessibility.mediaaccessibilityd
45055	PassbookUIService
45060	debugserver
100002	<enter a="" attach="" pid="" to=""></enter>
Line 286 c	286
	Help Search Cancel OK

Now that we've successfully attached to a system process, let's do something interesting with it. Consider the method -**[PurchaseOperation initWithPurchase:]**. This logic seems to be invoked when a transaction is performed in the AppStore. Set a breakpoint at this method, then open the AppStore on your device and try downloading an app (it can be any app, even a free one).

Immediately our breakpoint is hit, and we can start unwinding the logic that brought us here:

📑 IDA View-PC			0	🗊 General registers
text:00000001045F text:00000001045F text:00000001045F text:00000001045F	9AE8 ; Attributes: bp-based 9AE8 ; PurchaseOperation * 9AE8 ; PurchaseOperation_ini 9AE8 ; PurchaseOperation_ini 9AE8 var_50= -0x60 9AE8 var_50= -0x50 9AE8 var_40= -0x40 9AE8 var_20= -0x20 9AE8 var_10= -0x10 9AE8 var_10= -0x10 9AE8 var_90= 0 9AE8 var_90= 0 9AE8 var_80= 0 9AE8 SP, SP, SP, SP, SPE	cdecl -[PurchaseOperation initWithPurch tWithPurchase ; DATA XREF:objc_cor	ase:] X hst:(X X X X X X X X X X X X X X X X X X X	17 00000001045F9AE8 -[PurchaseOperation init 18 0000000000000 MEMORY:0000000130108E20 19 00000013C108E20
text:0000001045F text:0000001045F text:0000001045F 00019AE8 000000010	9AF4 STP X24, X2 9AF8 STP X22, X2	3, [SP,#0x60+var_30] 1, [SP,#0x60+var_20] ion initWithPurch: (Synchronized with	S P	<pre>S0 0000001688A6A20 MEMORY:00000001688A6A20 C 0000001045F9AE8 -[PurchaseOperation init SR 60000000</pre>
<pre>text:0000001045F text:0000001045F 00019AE8 00000010 Call Stack</pre>	9AF4 STP X24, X2 9AF8 STP X22, X2 45F9AE8: -[PurchaseOperat	3, [SF,#0x60+var_30] 11, [SF,#0x60+var_20] ion initWithPurch≀ (Synchronized with	S P	p 000000016B8A6A20 MEMORY:000000016B8A6A20 c 00000001045F9AE8 - [PurchaseOperation init
text:0000001045F text:00000001045F 00019AE8 00000010 Call Stack Address	9AF4 STP X24, X2 9AF8 STP X22, X2 (45F9AE8: -[PurchaseOperat Module	<pre>[3, [SP,#0x60+var_30] 11, [SP,#0x60+var_20] ion initWithPurch: (Synchronized with Function</pre>	PC) P	p 00000016B8A6A20 , MEMORY:000000016B8A6A20 C 0000001045F9AE8 , -[PurchaseOperation init SR 60000000
 text:0000001045F text:00000001045F 00019AE8 00000010 Call Stack Address 0000001045F9AE8 	9AF4 STP X24, X2 9AF8 STP X22, X2 145F9AE8: -[PurchaseOperat Module itunesstored	<pre>13, [SP,#0x60+var_30] 11, [SP,#0x60+var_20] ion initWithPurch: (Synchronized with Function PurchaseOperation *cdecl -[Pu</pre>	PC) P	<pre>p 00000016B8A6A20</pre>
text:0000001045F text:00000001045F 00019AE8 00019AE8 0000001045F 4ddress 10000001045F9AE8 100000001045F9AE8 100000001047D195C	9AF4 STP X24, X2 99AF8 STP X22, X2 145F9AE8: -[PurchaseOperat Module itunesstored	<pre>3; [SP,#0x60+var_30] 1; [SP,#0x60+var_20] ion initWithPurch: (Synchronized with Function PurchaseOperation *cdec1 -[Pu -[PurchaseManagerOperation _new</pre>	PC) P PC) P Purch	<pre>p 00000016B8A6A20</pre>
text:0000001045F text:0000001045F 00019AE8 00019AE8 0000001045F 0000001045F 0000001045F 0000001045F 0000001045F 0000001045F 00000001045F 0000001045F 0000001045F 0000001045F 000000010475F 00000001047CFDE0	9AF4 STP X24, X2 99AF8 STP X22, X2 145F9AE8: -[PurchaseOperat Module itunesstored itunesstored itunesstored	<pre>13, [SP,#0x60+var_30] 11, [SP,#0x60+var_20] ion initWithPurch: (Synchronized with Function PurchaseOperation *cdecl - [Pu - [PurchaseManagerOperation _new - [PurchaseManagerOperation run]</pre>	PC) P PC) P Purch	<pre>p 00000016B8A6A20</pre>
text:0000001045F text:0000001045F 00019AE8 00019AE8 0000001045F 0000001045F 0000001045F 0000001045F 0000001045F 0000001045F 00000001045F 0000001045F 00000001045F 00000001045F 00000001045F 00000001047CFDE0 000000018993C444	9AF4 STP X24, X2 9AF8 STP X22, X2 145F9AE8: -[PurchaseOperat Module itunesstored itunesstored itunesstored itunesstored	<pre>13, [SP,#0x60+var_30] 11, [SP,#0x60+var_20] ion initWithPurch: (Synchronized with Function PurchaseOperation *cdecl - [Pu - [PurchaseManagerOperation _new - [PurchaseManagerOperation run] - [ISOperation _main:]+128</pre>	PC) P PC) P Purch	<pre>p 00000016B8A6A20</pre>
text:0000001045F text:0000001045F 0019AE8 00100000000045F 0000001045F 0000001045F 0000001045F 0000001045F 00000001045F 00000001045F 00000001045F 00000001045F 00000001045F 00000001045F 00000001045F 00000001047CFDE0 000000018993C444 000000018993B074	9AF4 STP X24, X2 99AF8 STP X22, X2 145F9AE8: -[PurchaseOperat Module itunesstored itunesstored itunesstored	<pre>13, [SP,#0x60+var_30] 11, [SP,#0x60+var_20] ion initWithPurch: (Synchronized with Function PurchaseOperation *cdecl - [Pu - [PurchaseManagerOperation _new - [PurchaseManagerOperation run]</pre>	PC) P PC) P Purch	<pre>p 00000016B8A6A20</pre>
text:0000001045F text:0000001045F 00019AE8 00019AE8 0000001045F 0000001045F 0000001045F 0000001045F 0000001045F 0000001045F 00000001045F 0000001045F 00000001045F 00000001045F 00000001045F 00000001047CFDE0 000000018993C444	9AF4 STP X24, X2 9AF8 STP X22, X2 145F9AE8: -[PurchaseOperat Module itunesstored itunesstored itunesstored itunesstored	<pre>13, [SP,#0x60+var_30] 11, [SP,#0x60+var_20] ion initWithPurch: (Synchronized with Function PurchaseOperation *cdecl - [Pu - [PurchaseManagerOperation _new - [PurchaseManagerOperation run] - [ISOperation _main:]+128</pre>	PC) PC) Prchas Purch +0x31	<pre>p 00000016B8A6A20</pre>
text:0000001045F text:0000001045F 0019AE8 00100000000045F 0000001045F 0000001045F 0000001045F 0000001045F 00000001045F 00000001045F 00000001045F 00000001045F 00000001045F 00000001045F 00000001045F 00000001047CFDE0 000000018993C444 000000018993B074	9AF4 STP X24, X2 9AF8 STP X22, X2 145F9AE8: -[PurchaseOperat Module itunesstored itunesstored itunesstored itunesstored itunesstored itunesstored	<pre>i3, [SP,#0x60+var_30] i1, [SP,#0x60+var_20] ion initWithPurch: (Synchronized with</pre>	PC) PC) Prchas Purch +0x31	<pre>p 00000016B8A6A20</pre>
text:0000001045F text:0000001045F 0019AE8 00100000000045F 0000001045F 0000001045F9AE8 00000001045F9AE8 000000010475PAE8 000000010475PAE8	9AF4 STP X24, X2 9AF8 STP X22, X2 445F9AE8: -[PurchaseOperat Module itunesstored itunesstored itunesstored itunesstored itunesstore Foundation	<pre>ii; [SP,#0x60+var_30] ii; [SP,#0x60+var_20] ion initWithPurch: (Synchronized with PurchaseOperation *cdecl - [Pu - [PurchaseManagerOperationnew - [PurchaseManagerOperation run] - [ISOperationmain:]+128 - [ISOperationmain]+4D4 NSOPERATION_IS_INVOKING_MAIN</pre>	PC) P PC) P Purch +0x31	<pre>p 000000016B8A6A20</pre>
text:0000001045F text:0000001045F 0019AE8 001000001045F 32 Call Stack Address 32 0000001045F9AE8 32 0000001047D195C 32 0000001047CFDE0 32 00000018993C444 32 00000012608684 32 0000001250FDFC	9AF4 STP X24, X2 9AF8 STP X22, X2 445F9AE8: -[PurchaseOperat Module itunesstored itunesstored itunesstored iTunesStore Foundation Foundation	<pre>i3, [SP,#0x60+var_30] i1, [SP,#0x60+var_20] ion initWithPurch: (Synchronized with PurchaseOperation *cdecl - [Pu - [PurchaseManagerOperationnew - [PurchaseManagerOperation run] - [ISOperationmain:]+128 - [ISOperation main]+4D4 NSOPERATION_IS_INVOKING_MAIN - [NSOPeration start]+2D8</pre>	PC) P PC) P Purch +0x31	<pre>p 000000016B8A6A20</pre>

Stepping through this function, we see many Objective-C method call sites:

	text:00000001045F9B9C		X8, #selRef_initWithAccountIdentifier_@PAGE
•	text:00000001045F9BA0	LDR	X1, [X8,#selRef_initWithAccountIdentifier_@PAGEOFF] ; SEL
	text:00000001045F9BA4	MOV	x0, x22 ; id
-	text:00000001045F9BA8	MOV	x2, x23
PC •	text:0000001045F9BAC		_objc_msgSend
	text:00000001045F9BB0		X8, # OBJC IVAR \$ PurchaseOperation. authenticationContext@PAGE
•	text:00000001045F9BB4	LDRSW	X22, [X8,#_OBJC_IVAR \$_PurchaseOperationauthenticationContext@
	text:00000001045F9BB8	LDR	x8, [x20, x22]

Instead of using **F7** to step into the _objc_msgSend function, we can use shortcut **Shift-O** to take us directly to the Objective-C method that is being invoked:

		StoreServices: text:0000001B1D4C458		
		StoreServices: text:0000001B1D4C458	; -[SSAuthentica	<pre>ationContext initWithAccountIdentifier:]</pre>
X 1	7	StoreServices: text:0000001B1D4C458	SSAuthenticat:	ionContext_initWithAccountIdentifier
PC	P.	<pre>StoreServices: text:0000001B1D4C458</pre>	STP	X24, X23, [SP,#-0x40]!
	•	StoreServices: text:0000001B1D4C45C	STP	X22, X21, [SP,#0x10]
	•	StoreServices: text:0000001B1D4C460	STP	X20, X19, [SP,#0x20]
	•	StoreServices: text:0000001B1D4C464	STP	X29, X30, [SP,#0x30]
		StoreServices: text:0000001B1D4C468	ADD	X29, SP, #0x30 ; '0'

We discuss the **Shift-O** action in detail in our mac debugger tutorial, but it is worth demonstrating that this action works just as well in arm64/iOS environments.

It seems that we're well on our way to reverse-engineering transactions in the AppStore. The remaining work is left as an exercise for the reader :)

6.3. Conclusion

Hopefully by now we've shown that IDA's iOS Debugger is quite versatile. It can play by Apple's rules when debugging on a non-jailbroken device, and it can also be configured to use an enhanced debugserver when a jailbreak is available.

Also keep in mind that all previous examples in this writeup should work equally well with the patched debugserver. We encourage you to go back and try them.

7. Troubleshooting

IDA uses the Remote GDB Protocol to communicate with the iOS debugserver. Thus, the best way to diagnose possible issues is to log the packets transmitted between IDA and the server. You can do this by running IDA with the **-z10000** command-line option:

\$ ida64 -z10000 -L/tmp/ida.log

Often times these packets contain messages or error codes that provide clues to the issue.

For more enhanced troubleshooting, you can also enable logging on the server side. Go to **Debugger>Debugger** options>Set specific options and set the Syslog flags field:

	👷 iOS configuration	
Ar <u>c</u> hitecture	ARM64 (AArch64)	
Ma <u>x</u> packet size	-1	
Time <u>o</u> ut	1000	
Syslog <u>f</u> lags	LOG_DEFAULT LOG_RNB_PACKETS	~
Symbol <u>p</u> ath	/Users/troy/Library/Developer/Xcode/iOS DeviceSupport/13.2.2 (17B102)/Symbols	~

This will instruct the debugserver to log details about the debugging session to the iOS system log (all valid flags are documented under the **SYSLOG_FLAGS** option in dbg_ios.cfg).

Start collecting the iOS system log with:

\$ ios_deploy syslog -f /tmp/sys.log

Then launch the debugger. Now both the client (/tmp/ida.log) and the server (/tmp/sys.log) will log important events in the debugger session, which will often times reveal the issue.

8. Notes

This tutorial replaces the old iOS debugging tutorial, which is available here.