# 4

# BUILDING YOUR TEST PLATFORM

In this chapter, I'll outline the tools you need to review your code and test your iOS applications, and I'll show you how to build a robust and useful test platform. That test platform will include a properly set up Xcode instance, an interactive network proxy, reverse engineering tools, and tools to bypass iOS platform security checks.

I'll also cover the settings you need to change in Xcode projects to make bugs easier to identify and fix. You'll then learn to leverage Xcode's static analyzer and compiler options to produce well-protected binaries and perform more in-depth bug detection.

## Taking Off the Training Wheels

A number of behaviors in a default OS X install prevent you from really digging in to the system internals. To get your OS to stop hiding the things you need, enter the following commands at a Terminal prompt:

```
$ defaults write com.apple.Finder AppleShowAllFiles TRUE
$ defaults write com.apple.Finder ShowPathbar -bool true
$ defaults write com.apple.Finder _FXShowPosixPathInTitle -bool true
$ defaults write NSGlobalDomain AppleShowAllExtensions -bool true
$ chflags nohidden ~/Library/
```

These settings let you see all the files in the Finder, even ones that are hidden from view because they have a dot in front of their name. In addition, these changes will display more path information and file extensions, and most importantly, they allow you to see your user-specific *Library*, which is where the iOS Simulator will store all of its data.

The `chflags` command removes a level of obfuscation that Apple has put on directories that it considers too complicated for you, such as */tmp* or */usr*. I'm using the command here to show the contents of the iOS Simulator directories without having to use the command line every time.

One other thing: consider adding *$SIMPATH* to the Finder's sidebar for easy access. It's convenient to use *$SIMPATH* to examine the iOS Simulator's filesystem, but you can't get to it in the Finder by default. To make this change, browse to the following directory in the Terminal:

```
$ cd ~/Library/Application\ Support
$ open .
```

Then, in the Finder window that opens, drag the iPhone Simulator directory to the sidebar. Once you're riding without training wheels, it's time to choose your testing device.

## Suggested Testing Devices

My favorite test device is the Wi-Fi only iPad because it's inexpensive and easy to jailbreak, which allows for testing iPad, iPhone, and iPod Touch applications. Its lack of cellular-based networking isn't much of a hindrance, given that you'll want to intercept network traffic most of the time anyway.

But this configuration does have some minor limitations. Most significantly, the iPad doesn't have GPS or SMS, and it obviously doesn't make phone calls. So it's not a bad idea to have an actual iPhone of some kind available.

I prefer to have at least two iPads handy for iOS testing: one jailbroken and one stock. The stock device allows for testing in a legitimate, realistic end-user environment, and it has all platform security mechanisms still intact. It can also register properly for push notifications, which has proven problematic for jailbroken devices in the past. The jailbroken device allows you to more closely inspect the filesystem layout and more detailed workings of iOS; it also facilitates black-box testing that wouldn't be feasible using a stock device alone.

## Testing with a Device vs. Using a Simulator

Unlike some other mobile operating systems, iOS development uses a *simulator* rather than an emulator. This means there's no full emulation of the iOS device because that would require a virtualized ARM environment. Instead, the simulators that Apple distributes with Xcode are compiled for the x64 architecture, and they run natively on your development machine, which makes the process significantly faster and easier. (Try to boot the Android emulator inside a virtual machine, and you'll appreciate this feature.)

On the flip side, some things simply don't work the same way in the iOS Simulator as they do on the device. The differences are as follows:

**Case-sensitivity** Unless you've intentionally changed this behavior, OS X systems operate with case-insensitive HFS+ filesystems, while iOS uses the case-sensitive variant. This should rarely be relevant to security but can cause interoperability issues when modifying programs.

**Libraries** In some cases, iOS Simulator binaries link to OS X frameworks that may behave differently than those on iOS. This can result in slightly different behavior.

**Memory and performance** Since applications run natively in the iOS Simulator, they'll be taking full advantage of your development machine's resources. When gauging the impact of things such as PBKDF2 rounds (see Chapter 13), you'll want to compensate for this or test on a real device.

**Camera** As of now, the iOS Simulator does not use your development machine's camera. This is rarely a huge issue, but some applications do contain functionality such as "Take a picture of my check stub or receipt," where the handling of this photo data can be crucial.

**SMS and cellular** You can't test interaction with phone calls or SMS integration with the iOS Simulator, though you can technically simulate some aspects, such as toggling the "in-call" status bar.

Unlike in older versions of iOS, modern versions of the iOS Simulator do in fact simulate the Keychain API, meaning you can manage your own certificate and store and manipulate credentials. You can find the files behind this functionality in *$SIMPATH/Library/Keychains*.

## Network and Proxy Setup

Most of the time, the first step in testing any iOS application is to run it through a proxy so you can examine and potentially modify traffic going from the device to its remote endpoint. Most iOS security testers I know use BurpSuite[1] for this purpose.

1. *http://www.portswigger.net*

### *Bypassing TLS Validation*

There's one major catch to running an app under test through a proxy: iOS resolutely refuses to continue TLS/SSL connections when it cannot authenticate the server's certificate, as well it should. This is, of course, the correct behavior, but your proxy-based testing will screech to a halt rather quickly if iOS can't authenticate your proxy's certificate.

For BurpSuite specifically, you can obtain a CA certificate simply by configuring your device or iOS Simulator to use Burp as a proxy and then browsing to *http://burp/cert/* in Mobile Safari. This should work either on a real device or in the iOS Simulator. You can also install CA certificates onto a physical device by either emailing them to yourself or navigating to them on a web server.

For the iOS Simulator, a more general approach that works with almost any web proxy is to add the fingerprint of your proxy software's CA certificate directly into the iOS Simulator trust store. The trust store is a SQLite database, making it slightly more cumbersome to edit than typical certificate bundles. I recommend writing a script to automate this task. If you want to see an example to get you started, Gotham Digital Science has already created a Python script that does the job. You'll find the script here: *https://github.com/GDSSecurity/Add-Trusted-Certificate-to-iOS-Simulator/*.

To use this script, you need to obtain the CA certificate you want to install into the trust store. First configure Firefox[2] to use your local proxy (127.0.0.1, port 8080 for Burp). Then attempt to visit any SSL site; you should get a familiar certificate warning. Navigate to **Add Exception** → **View** → **Details** and click the **PortSwigger CA** entry, as shown in Figure 4-1.

Click **Export** and follow the prompts. Once you've saved the CA certificate, open *Terminal.app* and run the Python script to add the certificate to the store as follows:

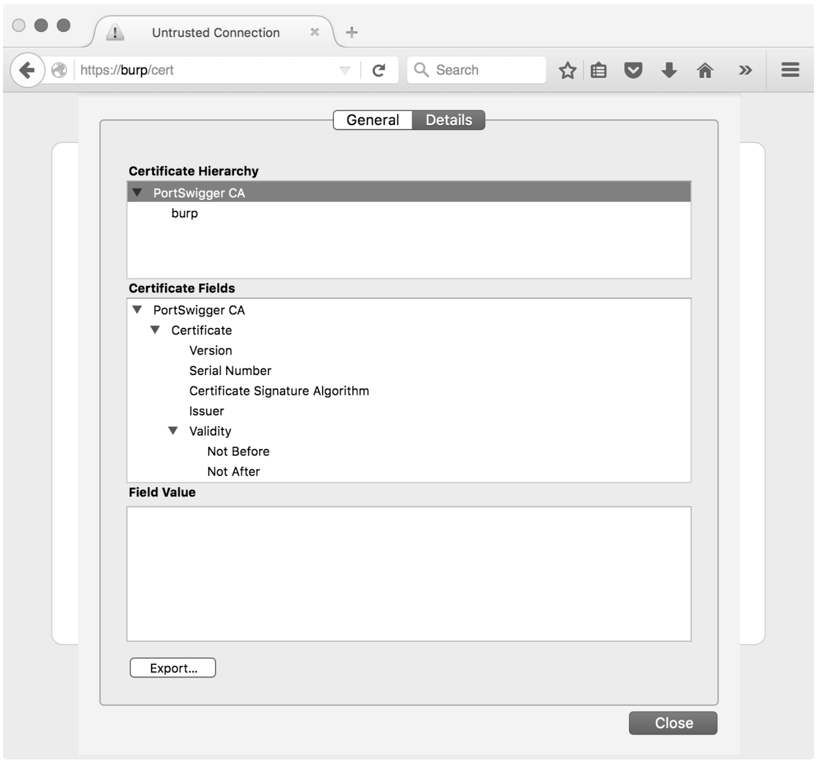
```
$ python ./add_ca_to_iossim.py ~/Downloads/PortSwiggerCA.pem
```

Unfortunately, at the time of writing, there isn't a native way to configure the iOS Simulator to go through an HTTP proxy without also routing the rest of your system through the proxy. Therefore, you'll need to configure the proxy in your host system's Preferences, as shown in Figure 4-2.

If you're using the machine for both testing and other work activities, you might consider specifically configuring other applications to go through a separate proxy, using something like FoxyProxy[3] for your browser.

2. I generally consider Chrome a more secure daily browser, but the self-contained nature of Firefox does let you tweak proxy settings more conveniently.

3. *http://getfoxyproxy.org*


Untrusted Connection

https://burp/cert

Search

General | Details

Certificate Hierarchy

- PortSwigger CA
  - burp

Certificate Fields

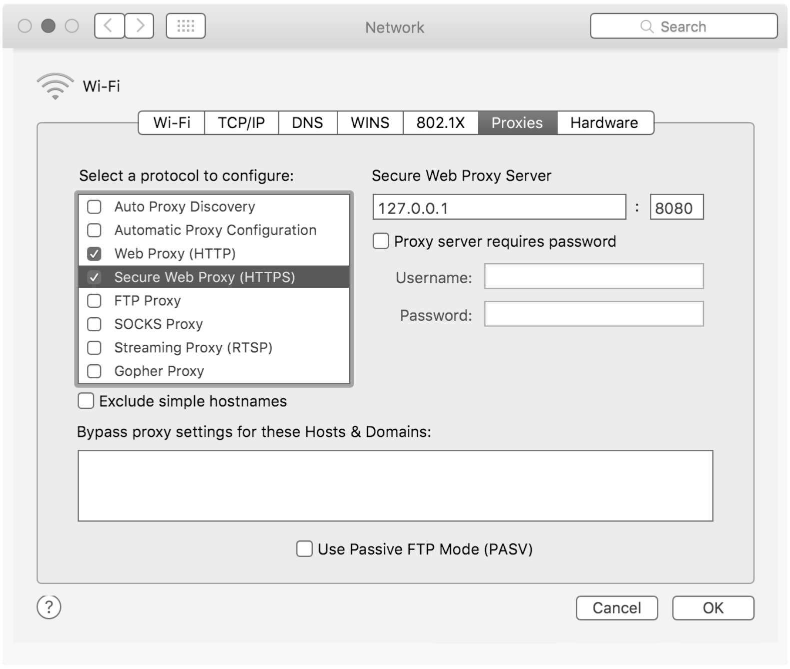
- PortSwigger CA
  - Certificate
    - Version
    - Serial Number
    - Certificate Signature Algorithm
    - Issuer
    - Validity
      - Not Before
      - Not After

Field Value

Export...

Close

Figure 4-1: Selecting the PortSwigger CA for export


Network

Search

Wi-Fi

Wi-Fi | TCP/IP | DNS | WINS | 802.1X | Proxies | Hardware

Select a protocol to configure:

- [ ] Auto Proxy Discovery
- [ ] Automatic Proxy Configuration
- [✓] Web Proxy (HTTP)
- [✓] Secure Web Proxy (HTTPS)
- [ ] FTP Proxy
- [ ] SOCKS Proxy
- [ ] Streaming Proxy (RTSP)
- [ ] Gopher Proxy

Secure Web Proxy Server

127.0.0.1 : 8080

- [ ] Proxy server requires password

Username:

Password:

- [ ] Exclude simple hostnames

Bypass proxy settings for these Hosts & Domains:

- [ ] Use Passive FTP Mode (PASV)

Cancel | OK

Figure 4-2: Configuring the host system to connect via Burp

## Bypassing SSL with stunnel

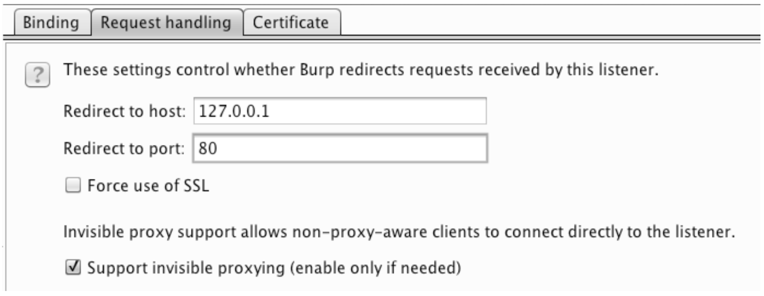
One method of bypassing SSL endpoint verification is to set up a termination point locally and then direct your application to use that instead. You can often accomplish this without recompiling the application, simply by modifying a plist file containing the endpoint URL.

This setup is particularly useful if you want to observe traffic easily in plaintext (for example, with Wireshark), but the Internet-accessible endpoint is available only over HTTPS. First, download and install stunnel,[4] which will act as a broker between the HTTPS endpoint and your local machine. If installed via Homebrew, stunnel's configuration file will be in */usr/local/etc/stunnel/stunnel.conf-sample*. Move or copy this file to */usr/local/etc/stunnel/stunnel.conf* and edit it to reflect the following:

```
; SSL client mode
client = yes

; service-level configuration
[https]
accept  = 127.0.0.1:80
connect = 10.10.1.50:443
TIMEOUTclose = 0
```

This simply sets up stunnel in client mode, instructing it to accept connections on your loopback interface on port 80, while forwarding them to the remote endpoint over SSL. After editing this file, set up Burp so that it uses your loopback listener as a proxy, making sure to select the **Support invisible proxying** option, as shown in Figure 4-3. Figure 4-4 shows the resulting setup.



These settings control whether Burp redirects requests received by this listener.

Redirect to host: 127.0.0.1

Redirect to port: 80

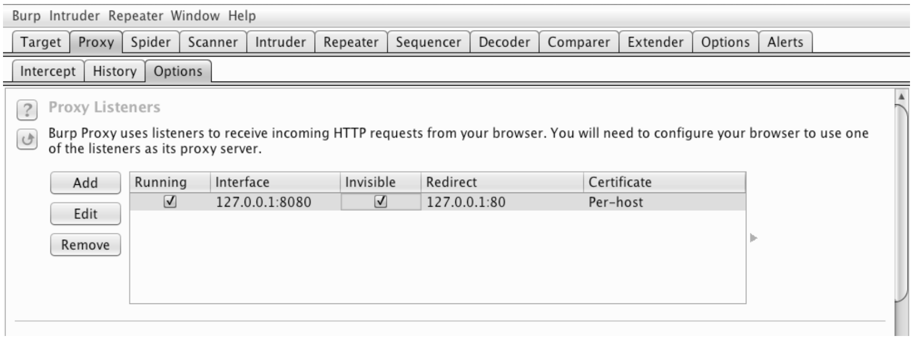
☐ Force use of SSL

Invisible proxy support allows non-proxy-aware clients to connect directly to the listener.

☑ Support invisible proxying (enable only if needed)

*Figure 4-3: Setting up invisible proxying through the local stunnel endpoint*

4. *http://www.stunnel.org/*


Burp Intruder Repeater Window Help
Target Proxy Spider Scanner Intruder Repeater Sequencer Decoder Comparer Extender Options Alerts
Intercept History Options
Proxy Listeners
Add
Edit
Remove

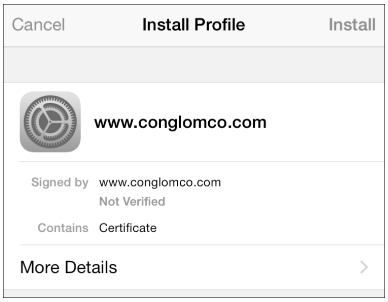
Burp Proxy uses listeners to receive incoming HTTP requests from your browser. You will need to configure your browser to use one of the listeners as its proxy server.

| Running | Interface | Invisible | Redirect | Certificate |
|---|---|---|---|---|
| ✓ | 127.0.0.1:8080 | ✓ | 127.0.0.1:80 | Per-host |

*Figure 4-4: Final Burp/stunnel setup*

### Certificate Management on a Device

To install a certificate on a physical iOS device, simply email the certificate to an account associated with the device or put it on a public web server and navigate to it using Mobile Safari. You can then import it into the device's trust store, as shown in Figure 4-5. You can also configure your device to go through a network proxy (that is, Burp) hosted on another machine. Simply install the CA certificate (as described earlier) of the proxy onto the device and configure your proxy to listen on a network-accessible IP address, as in Figure 4-6.

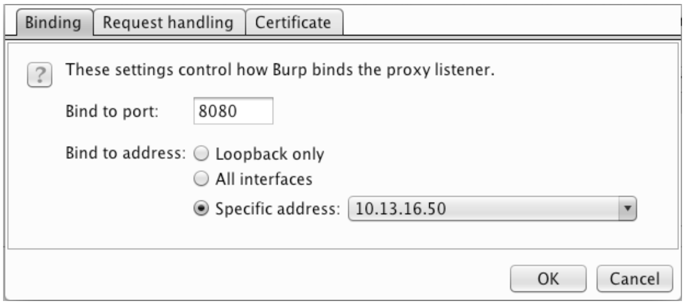


www.conglomco.com

Signed by www.conglomco.com
Not Verified

Contains Certificate

*Figure 4-5: The certificate import prompt*



These settings control how Burp binds the proxy listener.

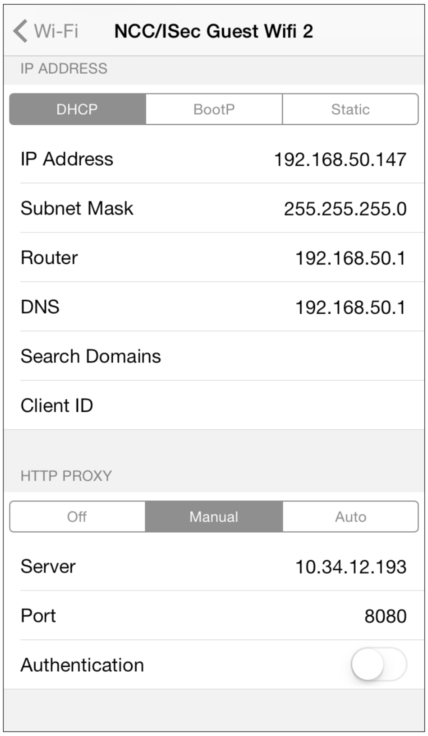
Bind to port: 8080

Bind to address: Loopback only / All interfaces / [✓ Specific address: 10.13.16.50]

*Figure 4-6: Configuring Burp to use a nonlocalhost IP address*

### Proxy Setup on a Device

Once you've configured your certificate authorities and set up the proxy, go to **Settings** → **Network** → **Wi-Fi** and click the arrow to the right of your currently selected wireless network. You can enter the proxy address and port from this screen (see Figure 4-7).



IP Address 192.168.50.147
Subnet Mask 255.255.255.0
Router 192.168.50.1
DNS 192.168.50.1
Search Domains
Client ID

HTTP PROXY
Off [✓ Manual] Auto
Server 10.34.12.193
Port 8080
Authentication

*Figure 4-7: Configuring the device to use a test proxy on an internal network*

Note that when configuring a device to use a proxy, only connections initiated by `NSURLConnection` or `NSURLSession` will obey the proxy settings; other connections such as `NSStream` and `CFStream` (which I'll discuss further in Chapter 7) will not. And of course, since this is an HTTP proxy, it works only for HTTP traffic. If you have an application using `CFStream`, you can edit the codebase with the following code snippet to route stream traffic through the same proxy as the host OS:

```
CFDictionaryRef systemProxySettings = CFNetworkCopySystemProxySettings();

CFReadStreamSetProperty(readStream, kCFStreamPropertyHTTPProxy, systemProxySettings
    );

CFWriteStreamSetProperty(writeStream, kCFStreamPropertyHTTPProxy,
    systemProxySettings);
```

If you're using `NSStream`, you can accomplish the same by casting the `NSInputStream` and `NSOutputStream` to their Core Foundation counterparts, like so:

```
CFDictionaryRef systemProxySettings = CFNetworkCopySystemProxySettings();

CFReadStreamSetProperty((CFReadStreamRef)readStream, kCFStreamPropertyHTTPProxy, (
    CFTypeRef)systemProxySettings);

CFWriteStreamSetProperty((CFWriteStreamRef)writeStream, kCFStreamPropertyHTTPProxy,
     (CFTypeRef)systemProxySettings);
```

If you're doing black-box testing and have an app that refuses to honor system proxy settings for HTTP requests, you can attempt to direct traffic through a proxy by adding a line to */etc/hosts* on the device to point name lookups to your proxy address, as shown in Listing 4-1.

```
10.50.22.11    myproxy api.testtarget.com
```

*Listing 4-1: Adding a hosts file entry*

You can also configure the device to use a DNS server controlled by you, which doesn't require jailbreaking your test device. One way to do this is to use Tim Newsham's dnsRedir,[5] a Python script that will provide a spoofed answer for DNS queries of a particular domain, while passing on queries for all other domains to another DNS server (by default, 8.8.8.8, but you can change this with the `-d` flag). The script can be used as follows:

```
$ dnsRedir.py 'A:www.evil.com.=1.2.3.4'
```

This should answer queries for *www.evil.com* with the IP address 1.2.3.4, where that IP address should usually be the IP address of the test machine you're proxying data through.

For non-HTTP traffic, things are a little more involved. You'll need to use a TCP proxy to intercept traffic. The aforementioned Tim Newsham has written a program that can make this simpler—the aptly named tcpprox.[6] If you use the `hosts` file method in Listing 4-1 to point the device to your proxy machine, you can then have tcpprox dynamically create SSL certificates and proxy the connection to the remote endpoint. To do this, you'll need to create a certificate authority certificate and install it on the device, as shown in Listing 4-2.

5. *https://github.com/iSECPartners/dnsRedir/*

6. *https://github.com/iSECPartners/tcpprox/*

```
$ ./prox.py -h
Usage: prox.py [opts] addr port

Options:
  -h, --help    show this help message and exit
  -6            Use IPv6
  -b BINDADDR   Address to bind to
  -L LOCPORT    Local port to listen on
  -s            Use SSL for incoming and outgoing connections
  --ssl-in      Use SSL for incoming connections
  --ssl-out     Use SSL for outgoing connections
  -3            Use SSLv3 protocol
  -T            Use TLSv1 protocol
  -C CERT       Cert for SSL
  -A AUTOCNAME  CName for Auto-generated SSL cert
  -1            Handle a single connection
  -l LOGFILE    Filename to log to

$ ./ca.py -c
$ ./pkcs12.sh ca
  (install CA cert on the device)
$ ./prox.py -s -L 8888 -A ssl.testtarget.com ssl.testtarget.com 8888
```

*Listing 4-2: Creating a certificate and using tcpprox to intercept traffic*

The *ca.py* script creates the signed certificate, and the *pkcs12.sh* script produces the certificate to install on the device, the same as shown in Figure 4-5. After running these and installing the certificate, your application should connect to the remote endpoint using the proxy, even for SSL connections. Once you've performed some testing, you can read the results with the *proxcat.py* script included with tcpprox, as follows:

```
$ ./proxcat.py -x log.txt
```

Once your application is connected through a proxy, you can start setting up your Xcode environment.

## Xcode and Build Setup

Xcode contains a twisty maze of project configuration options—hardly anyone understands what each one does. This section takes a closer look at these options, discusses why you would or wouldn't want them, and shows you how to get Xcode to help you find bugs before they become real problems.

### Make Life Difficult

First things first: treat warnings as errors. Most of the warnings generated by clang, Xcode's compiler frontend, are worth paying attention to. Not only do they often help reduce code complexity and ensure correct syntax, they also catch a number of errors that might be hard to spot, such as signedness issues or format string flaws. For example, consider the following:

```
- (void) validate:(NSArray*) someTribbles withValue:(NSInteger) desired {

    if (desired > [someTribbles count]) {
        [self allocateTribblesWithNumberOfTribbles:desired];
    }
}
```

The count method of NSArray returns an unsigned integer, (NSUInteger). If you were expecting the number of desired tribbles from user input, a submitted value might be –1, presumably indicating that the user would prefer to have an anti-tribble. Because desired is an integer being compared to an unsigned integer, the compiler will treat both as unsigned integers. Therefore, this method would unexpectedly allocate an absurd number of tribbles because –1 is an extremely large number when converted to an unsigned integer. I'll discuss this type of integer overflow issue further in Chapter 11.

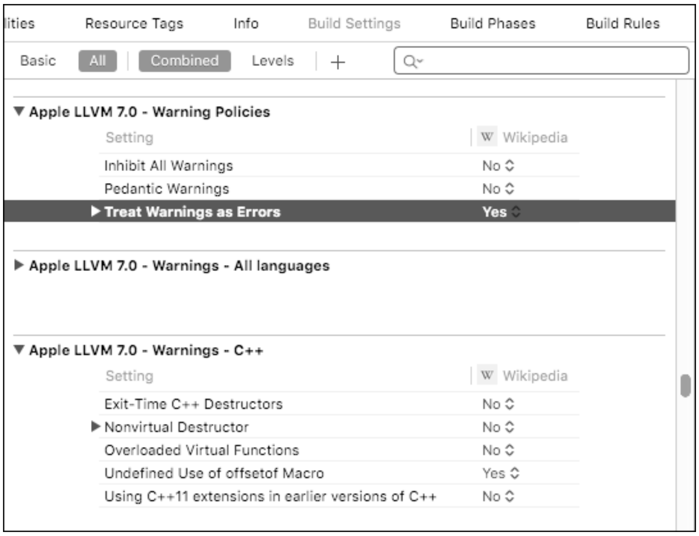
You can have clang flag this type of of bug by enabling most warnings and treating them as errors, in which case your build would fail with a message indicating "Comparison of integers of different signs: 'int' and 'NSUInteger' (aka 'unsigned int')".

**NOTE** *In general, you should enable all warnings in your project build configuration and promote warnings to errors so that you are forced to deal with bugs as early as possible in the development cycle.*

You can enable these options in your project and target build settings. To do so, first, under Warning Policies, set Treat Warnings as Errors to **Yes** (Figure 4-8). Then, under the Warnings sections, turn on all the desired options.

Note that not every build warning that clang supports has an exposed toggle in the Xcode UI. To develop in "hard mode," you can add the -Wextra or -Weverything flag, as in Figure 4-9. Not all warnings will be useful, but it's best to try to understand exactly what an option intends to highlight before disabling it.

-Weverything, used in Figure 4-9, is probably overkill unless you're curious about clang internals; -Wextra is normally sufficient. To save you a bit of time, Table 4-1 discusses two warnings that are almost sure to get in your way (or that are just plain bizarre).



▼ Apple LLVM 7.0 - Warning Policies

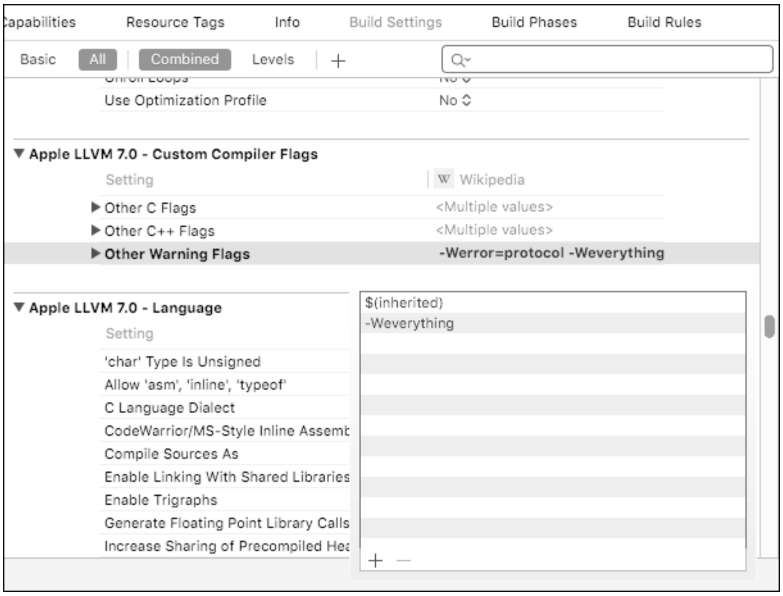
| Setting | Wikipedia |
|---|---|
| Inhibit All Warnings | No ◇ |
| Pedantic Warnings | No ◇ |
| ▶ **Treat Warnings as Errors** | **Yes** ◇ |

▶ Apple LLVM 7.0 - Warnings - All languages

▼ Apple LLVM 7.0 - Warnings - C++

| Setting | Wikipedia |
|---|---|
| Exit-Time C++ Destructors | No ◇ |
| ▶ Nonvirtual Destructor | No ◇ |
| Overloaded Virtual Functions | No ◇ |
| Undefined Use of offsetof Macro | Yes ◇ |
| Using C++11 extensions in earlier versions of C++ | No ◇ |

*Figure 4-8: Treating all warnings as errors*



| | |
|---|---|
| Use Optimization Profile | No ◇ |

▼ Apple LLVM 7.0 - Custom Compiler Flags

| Setting | Wikipedia |
|---|---|
| ▶ Other C Flags | <Multiple values> |
| ▶ Other C++ Flags | <Multiple values> |
| ▶ **Other Warning Flags** | **-Werror=protocol -Weverything** |

```
$(inherited)
-Weverything
```

▼ Apple LLVM 7.0 - Language

Setting
'char' Type Is Unsigned
Allow 'asm', 'inline', 'typeof'
C Language Dialect
CodeWarrior/MS-Style Inline Assemb
Compile Sources As
Enable Linking With Shared Libraries
Enable Trigraphs
Generate Floating Point Library Calls
Increase Sharing of Precompiled Hea

*Figure 4-9: This setting enables all warnings, including options for which there is no exposed UI.*

**Table 4-1:** Obnoxious Warnings to Disable in Xcode

| Compiler warning | Justification for disabling |
|---|---|
| Implicit synthesized properties | Since property synthesis is now automatic, this isn't really an error unless your development guidelines require explicit synthesis. |
| Unused parameters/functions/variables etc. | These can be supremely irritating when writing code, since your code is obviously not completely implemented yet. Consider enabling these only for nondebug builds. |

### Enabling Full ASLR

In iOS 4.3, Apple introduced *address space layout randomization (ASLR)*. ASLR ensures that the in-memory structure of the program and its data (libraries, the main executable, stack and heap, and memory-mapped files) are loaded into less predictable locations in the virtual address space. This makes code execution exploits more difficult because many rely on referencing the virtual addresses of specific library calls, as well as referencing data on the stack or heap.

For this to be fully effective, however, the application must be built as a *position-independent executable (PIE)*, which instructs the compiler to build machine code that can function regardless of its location in memory. Without this option, the location of the base executable and the stack will remain the same, even across reboots,[7] making an attacker's job much easier.

To ensure that full ASLR with PIE is enabled, check that Deployment Target in your Target's settings is set to at least iOS version 4.3. In your project's Build Settings, ensure that Generate Position-Dependent Code is set to No and that the bizarrely named Don't Create Position Independent Executable is also set to No. So don't create position-independent executables. Got it?

For black-box testing or to ensure that your app is built with ASLR correctly, you can use otool on the binary, as follows:

```
$ unzip MyApp.ipa
$ cd Payload/MyApp.app
$ otool -vh MyApp

MyApp (architecture armv7):
Mach header
      magic cputype cpusubtype caps    filetype ncmds sizeofcmds            flags
   MH_MAGIC     ARM         V7 0x00     EXECUTE    21       2672  NOUNDEFS DYLDLINK
                                                                       TWOLEVEL PIE
```

7. *http://www.trailofbits.com/resources/ios4_security_evaluation_paper.pdf*

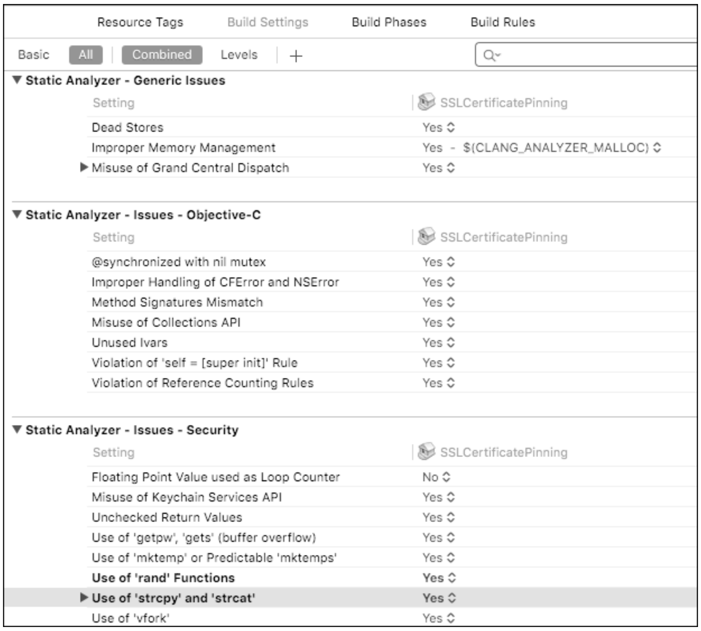
```
MyApp (architecture armv7s):
Mach header
      magic cputype cpusubtype caps    filetype ncmds sizeofcmds            flags
   MH_MAGIC     ARM        V7S 0x00     EXECUTE    21       2672  NOUNDEFS DYLDLINK
                                                                     TWOLEVEL PIE
```

At the end of each `MH_MAGIC` line, if you have your settings correct, you should see the `PIE` flag, highlighted in bold. (Note that this must be done on a binary compiled for an iOS device and will not work when used on iOS Simulator binaries.)

## Clang and Static Analysis

In computer security, *static analysis* generally refers to using tools to analyze a codebase and identify security flaws. This could involve identifying dangerous APIs, or it might include analyzing data flow through the program to identify the potentially unsafe handling of program inputs. As part of the build tool chain, clang is a good spot to embed static analysis language.

Beginning with Xcode 3.2, clang's static analyzer[8] has been integrated with Xcode, providing users with a UI to trace logic, coding flaws, and general API misuse. While clang's static analyzer is handy, several of its important features are disabled by default in Xcode. Notably, the checks for classic dangerous C library functions, such as `strcpy` and `strcat`, are oddly absent. Enable these in your Project or Target settings, as in Figure 4-10.



**Static Analyzer - Generic Issues**

| Setting | SSLCertificatePinning |
|---|---|
| Dead Stores | Yes |
| Improper Memory Management | Yes - $(CLANG_ANALYZER_MALLOC) |
| Misuse of Grand Central Dispatch | Yes |

**Static Analyzer - Issues - Objective-C**

| Setting | SSLCertificatePinning |
|---|---|
| @synchronized with nil mutex | Yes |
| Improper Handling of CFError and NSError | Yes |
| Method Signatures Mismatch | Yes |
| Misuse of Collections API | Yes |
| Unused Ivars | Yes |
| Violation of 'self = [super init]' Rule | Yes |
| Violation of Reference Counting Rules | Yes |

**Static Analyzer - Issues - Security**

| Setting | SSLCertificatePinning |
|---|---|
| Floating Point Value used as Loop Counter | No |
| Misuse of Keychain Services API | Yes |
| Unchecked Return Values | Yes |
| Use of 'getpw', 'gets' (buffer overflow) | Yes |
| Use of 'mktemp' or Predictable 'mktemps' | Yes |
| **Use of 'rand' Functions** | **Yes** |
| **Use of 'strcpy' and 'strcat'** | **Yes** |
| Use of 'vfork' | Yes |

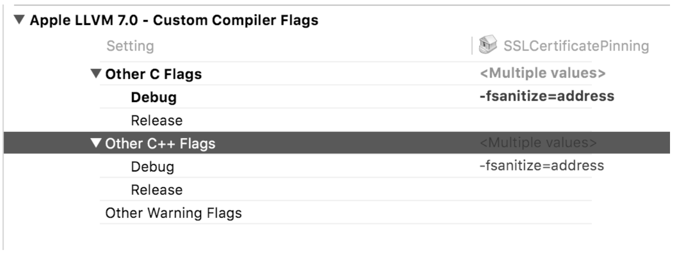
*Figure 4-10: Enabling all clang static analysis checks in Xcode*

8. *http://clang-analyzer.llvm.org/*

### Address Sanitizer and Dynamic Analysis

Recent versions of Xcode include a version of clang/llvm that features the Address Sanitizer (ASan). ASan is a dynamic analysis tool similar to Valgrind, but ASan runs faster and has improved coverage.[9] ASan tests for stack and heap overflows and use-after-free bugs, among other things, to help you track down crucial security flaws. It does have a performance impact (program execution is estimated to be roughly two times slower), so don't enable it on your release builds, but it should be perfectly usable during testing, quality assurance, or fuzzing runs.

To enable ASan, add `-fsanitize=address` to your compiler flags for debug builds (see Figure 4-11). On any unsafe crashes, ASan should write extra debug information to the console to help you determine the nature and severity of the issues. In conjunction with fuzzing,[10] ASan can be a great help in pinning down serious issues that may be security-sensitive and in giving an idea of their exploitability.



▼ Apple LLVM 7.0 - Custom Compiler Flags

| Setting | SSLCertificatePinning |
|---|---|
| ▼ Other C Flags | <Multiple values> |
| Debug | -fsanitize=address |
| Release | |
| ▼ Other C++ Flags | <Multiple values> |
| Debug | -fsanitize=address |
| Release | |
| Other Warning Flags | |

*Figure 4-11: Setting the ASan compiler flags*

## Monitoring Programs with Instruments

Regardless of whether you're analyzing someone else's application or trying to improve your own, the DTrace-powered Instruments tool is extremely helpful for observing an app's activity on a fine-grained level. This tool is useful for monitoring network socket usage, finding memory allocation issues, and watching filesystem interactions. Instruments can be an excellent tool for discovering what objects an application stores on local storage in order to find places where sensitive information might leak; I use it in that way frequently.

### Activating Instruments

To use Instruments on an application from within Xcode, hold down the **Run** button and select the **Build for Profiling** option (see Figure 4-12). After building, you will be presented with a list of preconfigured templates tailored for monitoring certain resources, such as disk reads and writes, memory allocations, CPU usage, and so on.

9. *http://clang.llvm.org/docs/AddressSanitizer.html*

10. *http://blog.chromium.org/2012/04/fuzzing-for-security.html*

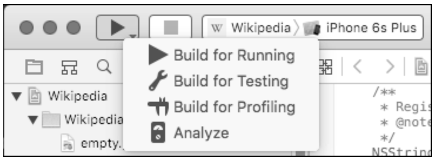
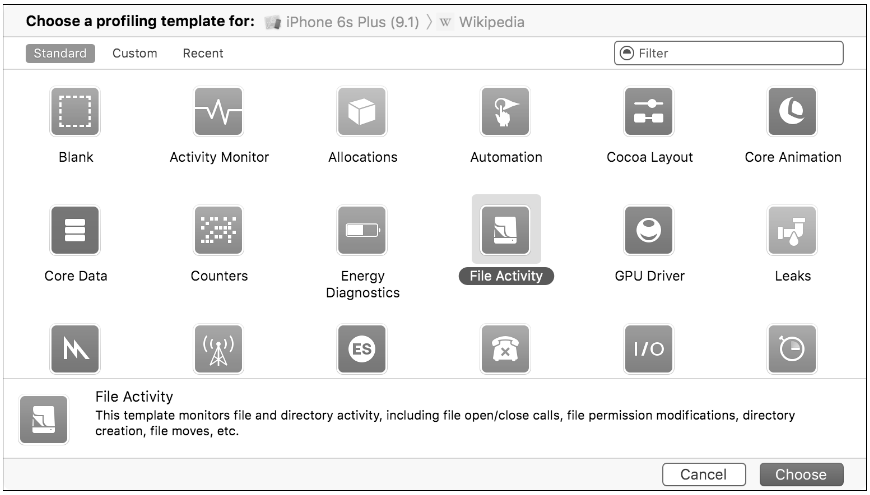


*Figure 4-12: Selecting the Build for Profiling option*

The File Activity template (shown in Figure 4-13) will help you monitor your application's disk I/O operations. After selecting the template, the iOS Simulator should automatically launch your application and begin recording its activity.



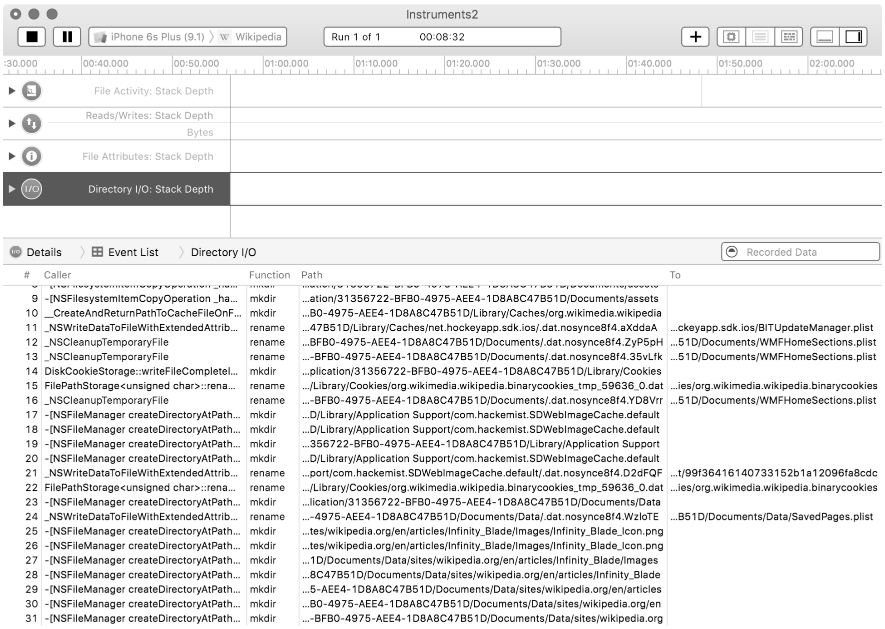
File Activity
This template monitors file and directory activity, including file open/close calls, file permission modifications, directory creation, file moves, etc.

*Figure 4-13: Selecting the File Activity profiling template*

There are a few preset views in Instruments for monitoring file activity. A good place to start is Directory I/O, which will capture all file creation or deletion events. Test your application the way you normally would and watch the output here. Each event is listed with its Objective-C caller, the C function call underlying it, the file's full path, and its new path if the event is a rename operation.

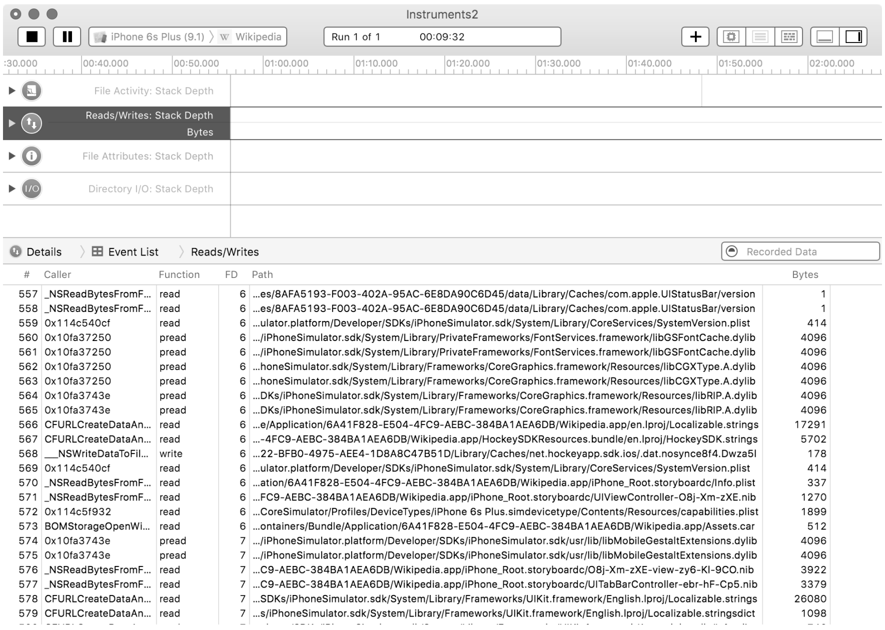
You'll likely notice several types of cache files being written here (see Figure 4-14), as well as cookies or documents your application has been asked to open. If you suspend your application, you should see the application screenshot written to disk, which I'll discuss in Chapter 10.

For a more detailed view, you can select the Reads/Writes view, as shown in Figure 4-15. This will show any read or write operations on files or sockets, along with statistics on the amount of data read or written.


Instruments2
iPhone 6s Plus (9.1) 〉 Wikipedia
Run 1 of 1 00:08:32
:30.000 00:40.000 00:50.000 01:00.000 01:10.000 01:20.000 01:30.000 01:40.000 01:50.000 02:00.000
File Activity: Stack Depth
Reads/Writes: Stack Depth
Bytes
File Attributes: Stack Depth
Directory I/O: Stack Depth
Details 〉 Event List 〉 Directory I/O
Recorded Data

| # | Caller | Function | Path | To |
|---|---|---|---|---|
| 9 | -[NSFilesystemItemCopyOperation _ha... | mkdir | ...ation/31356722-BFB0-4975-AEE4-1D8A8C47B51D/Documents/assets | |
| 10 | __CreateAndReturnPathToCacheFileOnF... | mkdir | ...B0-4975-AEE4-1D8A8C47B51D/Library/Caches/org.wikimedia.wikipedia | |
| 11 | _NSWriteDataToFileWithExtendedAttrib... | rename | ...47B51D/Library/Caches/net.hockeyapp.sdk.ios/.dat.nosynce8f4.aXddaA | ...ckeyapp.sdk.ios/BITUpdateManager.plist |
| 12 | _NSCleanupTemporaryFile | rename | ...BFB0-4975-AEE4-1D8A8C47B51D/Documents/.dat.nosynce8f4.ZyP5pH | ...51D/Documents/WMFHomeSections.plist |
| 13 | _NSCleanupTemporaryFile | rename | ...-BFB0-4975-AEE4-1D8A8C47B51D/Documents/.dat.nosynce8f4.35vLfk | ...51D/Documents/WMFHomeSections.plist |
| 14 | DiskCookieStorage::writeFileCompletel... | mkdir | ...plication/31356722-BFB0-4975-AEE4-1D8A8C47B51D/Library/Cookies | |
| 15 | FilePathStorage<unsigned char>::rena... | rename | .../Library/Cookies/org.wikimedia.wikipedia.binarycookies_tmp_59636_0.dat | ...ies/org.wikimedia.wikipedia.binarycookies |
| 16 | _NSCleanupTemporaryFile | rename | ...-BFB0-4975-AEE4-1D8A8C47B51D/Documents/.dat.nosynce8f4.YD8Vrr | ...51D/Documents/WMFHomeSections.plist |
| 17 | -[NSFileManager createDirectoryAtPath... | mkdir | ...D/Library/Application Support/com.hackemist.SDWebImageCache.default | |
| 18 | -[NSFileManager createDirectoryAtPath... | mkdir | ...D/Library/Application Support/com.hackemist.SDWebImageCache.default | |
| 19 | -[NSFileManager createDirectoryAtPath... | mkdir | ...356722-BFB0-4975-AEE4-1D8A8C47B51D/Library/Application Support | |
| 20 | -[NSFileManager createDirectoryAtPath... | mkdir | ...D/Library/Application Support/com.hackemist.SDWebImageCache.default | |
| 21 | _NSWriteDataToFileWithExtendedAttrib... | rename | ...port/com.hackemist.SDWebImageCache.default/.dat.nosynce8f4.D2dFQF | ...t/99f36416140733152b1a12096fa8cdc |
| 22 | FilePathStorage<unsigned char>::rena... | rename | .../Library/Cookies/org.wikimedia.wikipedia.binarycookies_tmp_59636_0.dat | ...ies/org.wikimedia.wikipedia.binarycookies |
| 23 | -[NSFileManager createDirectoryAtPath... | mkdir | ...lication/31356722-BFB0-4975-AEE4-1D8A8C47B51D/Documents/Data | |
| 24 | _NSWriteDataToFileWithExtendedAttrib... | rename | ...-4975-AEE4-1D8A8C47B51D/Documents/Data/.dat.nosynce8f4.WzloTE | ...B51D/Documents/Data/SavedPages.plist |
| 25 | -[NSFileManager createDirectoryAtPath... | mkdir | ...tes/wikipedia.org/en/articles/Infinity_Blade/Images/Infinity_Blade_Icon.png | |
| 26 | -[NSFileManager createDirectoryAtPath... | mkdir | ...tes/wikipedia.org/en/articles/Infinity_Blade/Images/Infinity_Blade_Icon.png | |
| 27 | -[NSFileManager createDirectoryAtPath... | mkdir | ...1D/Documents/Data/sites/wikipedia.org/en/articles/Infinity_Blade/Images | |
| 28 | -[NSFileManager createDirectoryAtPath... | mkdir | ...8C47B51D/Documents/Data/sites/wikipedia.org/en/articles/Infinity_Blade | |
| 29 | -[NSFileManager createDirectoryAtPath... | mkdir | ...5-AEE4-1D8A8C47B51D/Documents/Data/sites/wikipedia.org/en/articles | |
| 30 | -[NSFileManager createDirectoryAtPath... | mkdir | ...B0-4975-AEE4-1D8A8C47B51D/Documents/Data/sites/wikipedia.org/en | |
| 31 | -[NSFileManager createDirectoryAtPath... | mkdir | ...-BFB0-4975-AEE4-1D8A8C47B51D/Documents/Data/sites/wikipedia.org | |

*Figure 4-14: Directory I/O view showing files created or deleted*


Instruments2
iPhone 6s Plus (9.1) 〉 Wikipedia
Run 1 of 1 00:09:32
:30.000 00:40.000 00:50.000 01:00.000 01:10.000 01:20.000 01:30.000 01:40.000 01:50.000 02:00.000
File Activity: Stack Depth
Reads/Writes: Stack Depth
Bytes
File Attributes: Stack Depth
Directory I/O: Stack Depth
Details 〉 Event List 〉 Reads/Writes
Recorded Data

| # | Caller | Function | FD | Path | Bytes |
|---|---|---|---|---|---|
| 557 | _NSReadBytesFromF... | read | 6 | ...es/8AFA5193-F003-402A-95AC-6E8DA90C6D45/data/Library/Caches/com.apple.UIStatusBar/version | 1 |
| 558 | _NSReadBytesFromF... | read | 6 | ...es/8AFA5193-F003-402A-95AC-6E8DA90C6D45/data/Library/Caches/com.apple.UIStatusBar/version | 1 |
| 559 | 0x114c540cf | read | 6 | ...ulator.platform/Developer/SDKs/iPhoneSimulator.sdk/System/Library/CoreServices/SystemVersion.plist | 414 |
| 560 | 0x10fa37250 | pread | 6 | .../iPhoneSimulator.sdk/System/Library/PrivateFrameworks/FontServices.framework/libGSFontCache.dylib | 4096 |
| 561 | 0x10fa37250 | pread | 6 | .../iPhoneSimulator.sdk/System/Library/PrivateFrameworks/FontServices.framework/libGSFontCache.dylib | 4096 |
| 562 | 0x10fa37250 | pread | 6 | ...honeSimulator.sdk/System/Library/Frameworks/CoreGraphics.framework/Resources/libCGXType.A.dylib | 4096 |
| 563 | 0x10fa37250 | pread | 6 | ...honeSimulator.sdk/System/Library/Frameworks/CoreGraphics.framework/Resources/libCGXType.A.dylib | 4096 |
| 564 | 0x10fa3743e | pread | 6 | ...DKs/iPhoneSimulator.sdk/System/Library/Frameworks/CoreGraphics.framework/Resources/libRIP.A.dylib | 4096 |
| 565 | 0x10fa3743e | pread | 6 | ...DKs/iPhoneSimulator.sdk/System/Library/Frameworks/CoreGraphics.framework/Resources/libRIP.A.dylib | 4096 |
| 566 | CFURLCreateDataAn... | read | 6 | ...e/Application/6A41F828-E504-4FC9-AEBC-384BA1AEA6DB/Wikipedia.app/en.lproj/Localizable.strings | 17291 |
| 567 | CFURLCreateDataAn... | read | 6 | ...-4FC9-AEBC-384BA1AEA6DB/Wikipedia.app/HockeySDKResources.bundle/en.lproj/HockeySDK.strings | 5702 |
| 568 | ___NSWriteDataToFil... | write | 6 | ...22-BFB0-4975-AEE4-1D8A8C47B51D/Library/Caches/net.hockeyapp.sdk.ios/.dat.nosynce8f4.Dwza5l | 178 |
| 569 | 0x114c540cf | read | 6 | ...ulator.platform/Developer/SDKs/iPhoneSimulator.sdk/System/Library/CoreServices/SystemVersion.plist | 414 |
| 570 | _NSReadBytesFromF... | read | 6 | ...ation/6A41F828-E504-4FC9-AEBC-384BA1AEA6DB/Wikipedia.app/iPhone_Root.storyboardc/Info.plist | 337 |
| 571 | _NSReadBytesFromF... | read | 6 | ...FC9-AEBC-384BA1AEA6DB/Wikipedia.app/iPhone_Root.storyboardc/UIViewController-O8j-Xm-zXE.nib | 1270 |
| 572 | 0x114c5f932 | read | 6 | ...CoreSimulator/Profiles/DeviceTypes/iPhone 6s Plus.simdevicetype/Contents/Resources/capabilities.plist | 1899 |
| 573 | BOMStorageOpenWi... | read | 6 | ...ontainers/Bundle/Application/6A41F828-E504-4FC9-AEBC-384BA1AEA6DB/Wikipedia.app/Assets.car | 512 |
| 574 | 0x10fa3743e | pread | 7 | .../iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator.sdk/usr/lib/libMobileGestaltExtensions.dylib | 4096 |
| 575 | 0x10fa3743e | pread | 7 | .../iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator.sdk/usr/lib/libMobileGestaltExtensions.dylib | 4096 |
| 576 | _NSReadBytesFromF... | read | 7 | ...C9-AEBC-384BA1AEA6DB/Wikipedia.app/iPhone_Root.storyboardc/O8j-Xm-zXE-view-zy6-Kl-9CO.nib | 3922 |
| 577 | _NSReadBytesFromF... | read | 7 | ...C9-AEBC-384BA1AEA6DB/Wikipedia.app/iPhone_Root.storyboardc/UITabBarController-ebr-hF-Cp5.nib | 3379 |
| 578 | CFURLCreateDataAn... | read | 7 | ...SDKs/iPhoneSimulator.sdk/System/Library/Frameworks/UIKit.framework/English.lproj/Localizable.strings | 26080 |
| 579 | CFURLCreateDataAn... | read | 7 | ...s/iPhoneSimulator.sdk/System/Library/Frameworks/UIKit.framework/English.lproj/Localizable.stringsdict | 1098 |

*Figure 4-15: Profiling results showing detailed file reads and writes*

### *Watching Filesystem Activity with Watchdog*

Instruments should catch most iOS Simulator activity, but some file writes or network calls may actually be performed by other system services, thereby escaping the tool's notice. It's a good idea to manually inspect the iOS Simulator's directory tree to get a better feel for the structure of iOS and its applications and to catch application activity that you might otherwise miss.

One easy way to automate this is to use the Python watchdog module.[11] Watchdog will use either the kqueue or FSEvents API to monitor directory trees for file activity and can either log events or take specific actions when these events occur. To install watchdog, use the following:

```
$ pip install watchdog
```

You can write your own scripts to use watchdog's functionality, but you'll find a nice command line tool already included with watchdog called watchmedo. If you open a Terminal window and navigate to the Simulator directory, you should be able to use watchmedo to monitor all file changes under the iOS Simulator's directory tree, as follows:

```
$ cd ~/Library/Application\ Support/iPhone\ Simulator/6.1
$ watchmedo log --recursive .
on_modified(self=<watchdog.tricks.LoggerTrick object at 0x103c9b190>, event=<
     DirModifiedEvent: src_path=/Users/dthiel/Library/Application Support/iPhone
     Simulator/6.1/Library/Preferences>)
on_created(self=<watchdog.tricks.LoggerTrick object at 0x103c9b190>, event=<
     FileCreatedEvent: src_path=/Users/dthiel/Library/Application Support/iPhone
     Simulator/6.1/Applications/9460475C-B94A-43E8-89C0-285DD036DA7A/Library/Caches
     /Snapshots/com.yourcompany.UICatalog/UIApplicationAutomaticSnapshotDefault-
     Portrait.png>)
on_modified(self=<watchdog.tricks.LoggerTrick object at 0x103c9b190>, event=<
     DirModifiedEvent: src_path=/Users/dthiel/Library/Application Support/iPhone
     Simulator/6.1/Applications/9460475C-B94A-43E8-89C0-285DD036DA7A/Library/Caches
     /Snapshots>)
on_created(self=<watchdog.tricks.LoggerTrick object at 0x103c9b190>, event=<
     DirCreatedEvent: src_path=/Users/dthiel/Library/Application Support/iPhone
     Simulator/6.1/Applications/9460475C-B94A-43E8-89C0-285DD036DA7A/Library/Caches
     /Snapshots/com.yourcompany.UICatalog>)
on_modified(self=<watchdog.tricks.LoggerTrick object at 0x103c9b190>, event=<
     DirModifiedEvent: src_path=/Users/dthiel/Library/Application Support/iPhone
     Simulator/6.1/Library/SpringBoard>)
```

11. *https://pypi.python.org/pypi/watchdog/*

Entries that start with on `on_modified` indicate a file was changed, and entries that start with `on_created` indicate a new file. There are several other change indicators you might see from watchmedo, and you can read about them in the Watchdog documentation.

## Closing Thoughts

You should now have your build and test environment configured for running, modifying, and examining iOS apps. In Chapter 5, we'll take a closer look at how to debug and inspect applications dynamically, as well as how to change their behavior at runtime.