# Jailbreaking for Developers, An Overview

Much has been said about Apple’s closed approach to selling devices and running an app platform. But what few know is that behind closed doors there’s a massive ecosystem of libraries and hardware features waiting to be unlocked by developers. All of the APIs Apple uses internally to build their services, applications, and widgets are available once the locks are broken via a process called jailbreaking. Most of them are written in Objective-C, a dynamic language that provides very rich introspection capabilities and has a culture of self-describing code. Further tearing down barriers, most people install something called CydiaSubstrate shortly after jailbreaking, which allows running custom code inside every existing process on the device. This is very powerful—not only have we broken out of the walled garden into the rest of the forest, all of the footpaths are already labeled. Building code that targets jailbroken iOS devices involves unique ways of inspecting APIs, injecting code into processes, and writing code that modifies existing classes and finalized behaviors of the system.

The APIs implemented on iOS can be divided into four categories: framework-level Objective-C APIs, app-level Objective-C classes, C-accessible APIs and JavaScript-accessible APIs. Objective-C frameworks are the most easily accessible. Normally the structure of a component is only accessible to the programmer and those the source code or documentation have been made available to, but compiled Objective-C binaries include method tables describing all of the classes, protocols, methods and instance variables contained in the binary. An entire family of “class-dump” tools exists to take these method tables and convert them to header-like output for easy consumption by adventurous programmers. Calling these APIs is as simple as adding the generated headers to your project and linking with the framework or library. The second category of app internal classes may be inspected via the same tools, but are not linkable via standard tools. To get to those classes, one has to have code injected into the app in question and use the Objective-C runtime function objc\_getClass to get a reference to the class; from there, one can call APIs via the headers generated by the tool. Inspecting C-level functions are more difficult. No information about what the parameters or data structures are baked into the binaries, only the names of exported functions. The developer tools that ship with OS X come with a disassembler named “otool” which can dump the instructions used to implement the code in the device. Paired with knowledge of ARM assembly, the type information can be reconstructed by hand with much effort. This is much more cumbersome than with Objective-C code. Luckily, some of the components implemented in C are shared with OS X and have headers available in the OS X SDK, or are available as open-source from Apple. JavaScript-level APIs are most often facades over Objective-C level APIs to make additional functionality accessible to web pages hosted inside the iTunes, App Store, iCloud and iAd sections of the operating system.

Putting the APIs one has uncovered to use often requires having code run inside the process where their implementations are present. This can be done using the DYLD\_INSERT\_LIBRARIES environment variable on systems that use dyld, but this facility offers very few provisions for crash protection and can easily leave a device in a state where a restore is necessary. Instead, the gold standard on iOS devices is a system known as Cydia Substrate, a package that standardizes process injection and offers safety features to limit the damage testing new code can do. Once Cydia Substrate is installed, one needs only to drop a dynamic library compiled for the device in /Library/ MobileSubstrate/DynamicLibraries, and substrate will load it automatically in every process on the device. Filtering to only a specific process can be achieved by dropping a property list of the same name alongside it with details on which process or grouping of processes to filter to. Once inside, one can register for events, call system APIs and perform any of the same behaviors that the process normally could. This applies to apps that come preinstalled on the device, apps available from the App Store, the window manager known as SpringBoard, UI services that apps can make use of such as the mail composer, and background services such as the media decoder daemon. It is important to note that any state that the injected code has will be unique to the process it’s injected into and to share state mandates use inter-process communication techniques such as sockets, fifos, mach ports and shared memory.

Modifying existing code is where it really starts to get powerful and allows tweaking existing functionality of the device in simple or even radical ways. Because Objective-C method lookup is all done at runtime and the runtime offers APIs to modify methods and classes, it is really straightforward to replace the implementations of existing methods with new ones that add new functionality, suppress the original behavior or both. This is known as method hooking and in Objective-C is done through a complicated dance of calls to the class\_addMethod, class\_ getInstanceMethod, method\_getImplementation and method\_setImplementation runtime functions. This is very unwieldy; tools to automate this have been built. The simplest is Cydia Substrate’s own MSHookMessage function. It takes a class, the name of the method you want to replace, the new implementation, and gives back the original implementation of the function so that the replacement can perform the original behavior if necessary. This has been further automated in the Logos Objective-C preprocessor tool, which introduces syntax specifically for method hooking and is what most tweaks are now written in. Writing Logos code is as simple as writing what would normally be an Objective-C method implementation, and sticking it inside of a %hook ClassName ... %end block instead of an @implementation ClassName ... %end block, and calling %orig() instead of [super ...]. Simple tweaks to how the system behaves can often done by replacing a single method with a different implementation, but complicated adjustments often require assembling numerous method hooks. Since most of iOS is implemented in Objective-C, the vast majority of tweaks need only these building blocks to apply the modifications they require. For the lower levels of the system that are written in C, a more complicate hooking approach is required. The lowest level and most compatible way of doing so is to simply rewrite the assembly instructions of the victim function. This is very dangerous and does not compose well when many developers are modifying the same parts of the system. Again, CydiaSubstrate introduces an API to automate this in form of MSHookFunction. Just like MSHookMessage, one needs only to pass in the target function, new replacement implementation function, and it applies the hook and returns the old implementation that the new replacement can call if necessary. With the tools the community has made available, the details of the very complex mechanics of hooking have been abstracted and simplified to the point where they’re hidden from view and a developer can concentrate on what new features they’re adding.

Combining these techniques unique to the jailbreak scene, with those present in the standard iOS and OS X development communities yields a very flexible and powerful tool chest for building features and experiences that the world hasn’t seen yet.

Ryan Petrich

Code Wrangler, Father of Activator