Dynamically Overriding Mac OS X

Down the Rabbit Hole

Copyright © 2003 Red Shed Software. all rights reserved.

Jonathan 'Wolf' Rentzsch
(jon at redshed dot net).

These rules are no different than those of a computer system.
Some of them can be bent. Others... can be broken.
— Morpheus

Abstract

Mac OS X is a whole new ballgame. The problem is that while the OS got game, we're missing the balls — the ability to dynamically override software functionality. This paper details the two basic techniques that reanimate Mac OS 9-style system extensions: **function overriding** and **code injection**.

Dynamic Overriding Defined

Dynamic overriding is the ability to change software at runtime, often in ways not originally anticipated by the software's original authors.

Old-school Macintosh programmers will read the previous paragraph and immediately think "**trap patching**": the Classic Mac OS's method of providing dynamic overriding. However, when addressing Mac OS X, we must abandon this terminology for two reasons:

1. **Accuracy.** While Mac OS X still uses traps, their role is greatly diminished in the face of shared libraries. Traps play no role in dynamic overriding on Mac OS X, and thus "trap patching" is not applicable.
2. **Confusion.** "Patching" has a specific connotation on Unix systems: applying textual differences to source code prior to compilation. Dynamic overriding is wholly different as no permanent changes are made to the target software, these changes are made after compilation, at runtime.

Besides being able to suppress, change or extend functionality in prepackaged software, dynamic overriding also has the considerable advantage of being easy to remove. All that's needed is to disable the overriding software and relaunch the affected application (or — in the worst case — restarting the system). No permanent changes are made to the applications or system.

Function Overriding

Unlike the classic Mac OS, Mac OS X does not directly support dynamically overriding system-supplied functions. At best, APIs exposed with Objective C interfaces can be overridden with categories and/or posing. However, this has two major limitations:

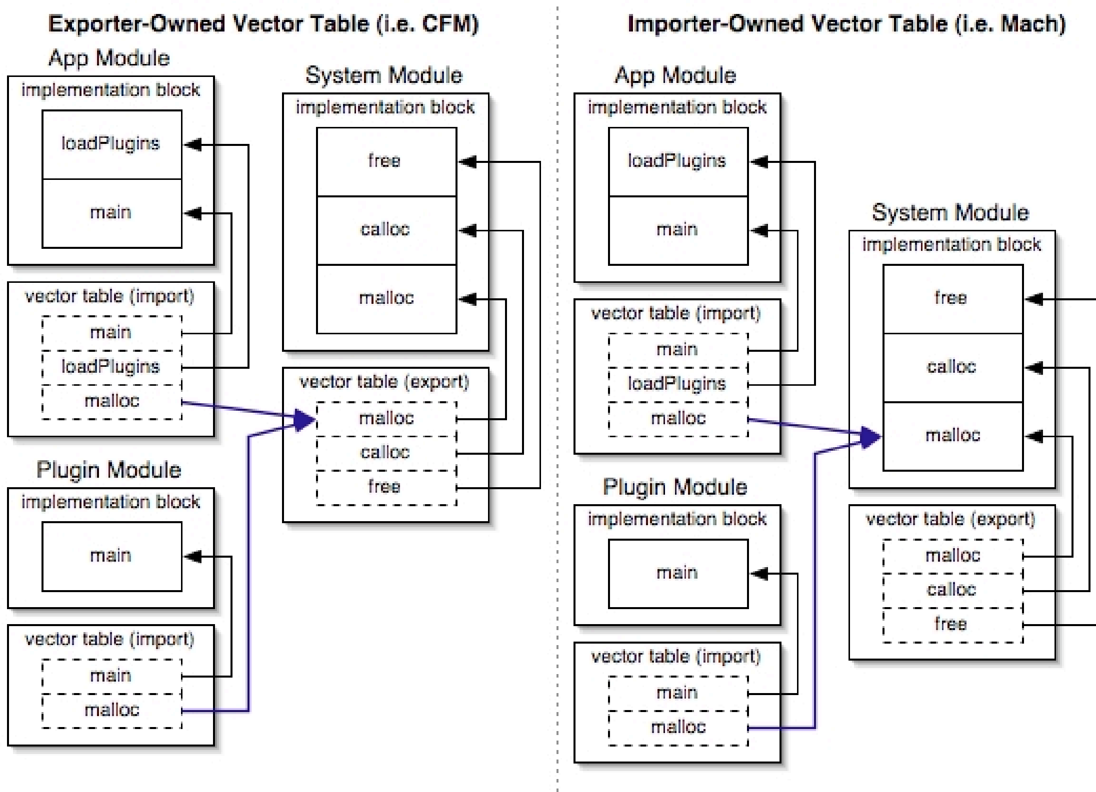
1. **It requires the desired API be exposed in Objective C.** Most of Mac OS X's functions are exposed as plain C functions. This includes all of Mach, the BSD

layer, and Carbon. Even if there are Objective C wrappers for these functions, you run into the second limitation:

2. **Overriding an Objective C wrapper does not override the original function.** If your application has a mix of code, some of which calls the Objective C wrapper, some of which calls the wrapped functions directly, your override will not be universally applied.

Ideally, overriding a system-supplied function would involve little more than discovering its entry in a universal table, saving off the pointer to the original function's code and replacing it with a pointer to the desired override code.

However, Mach's linking model lacks any sort of central bottleneck (pointer). Unlike Mac OS 9's Code Fragment Manager (CFM), which maintains a single bottleneck to a function's implementation, Mach links every imported symbol directly to its implementation. The following figure illustrates the differences between the models.



Vector Table Ownership

Before you get any crazy ideas about enumerating all loaded modules to rewrite their vector tables, you need to be aware of two things. The first is Mach's lazy binding — a vector table's symbol isn't resolved until the first time it's called. You would need to manually resolve the symbols yourself, an expensive and tricky process. The second is that rewriting the vector tables still wouldn't work for symbols looked up programmatically, or modules loaded after your manual rewrite.

So rewriting vector tables is largely out of the question. What is surprisingly feasible is to rewrite the function's implementation itself. The idea is to replace the function's first

instruction with a branch instruction to the desired override function. This technique is known as **single-instruction overwriting**.

By confining ourselves to replacing only one instruction, we reap a number of benefits:

> **Atomic replacement.** The PowerPC can atomically update one 32-bit value in memory. All PowerPC instructions just so happen to be 32 bits long. So if we have multiple preemptive threads across multiple processors replacing the same instruction, we're guaranteed predictable results, and a stale instruction will never be accidentally restored.
> **Less likely to harmfully impact the original code.** If the intent of the override is to fully replace the original function, then overwriting the first instruction of the original function can't hurt anything. However, if the intent is to reenter the original code, things get more complicated. The original code can't just be replaced — it needs to be saved off and relocated into another memory block and re-executed prior to reentry. The more code that needs to be relocated, the greater the chances that you will relocate code that shouldn't be relocated (a branch relative instruction, for example). A single-instruction swap minimizes harmful potential.
> **Compatibility.** Single-instruction overwriting is most likely to work with the widest variety of function prologs and other patching implementations, including our own!

These benefits are worthwhile, but replacing only one instruction is rather confining. It appears our only real choice is a branch instruction, to quickly jump to an area with more wiggle room. Since we're limited to only one instruction, this implies we can't load a register with any branch target address (that would take at least three instructions). Which means we only have `b` (branch relative), `ba` (branch absolute), `bl` (branch relative, update link register) and `bla` (branch absolute, update link register). However, we don't want to stomp on the link register — it houses the caller's return address! That leaves us with `b` and `ba`.

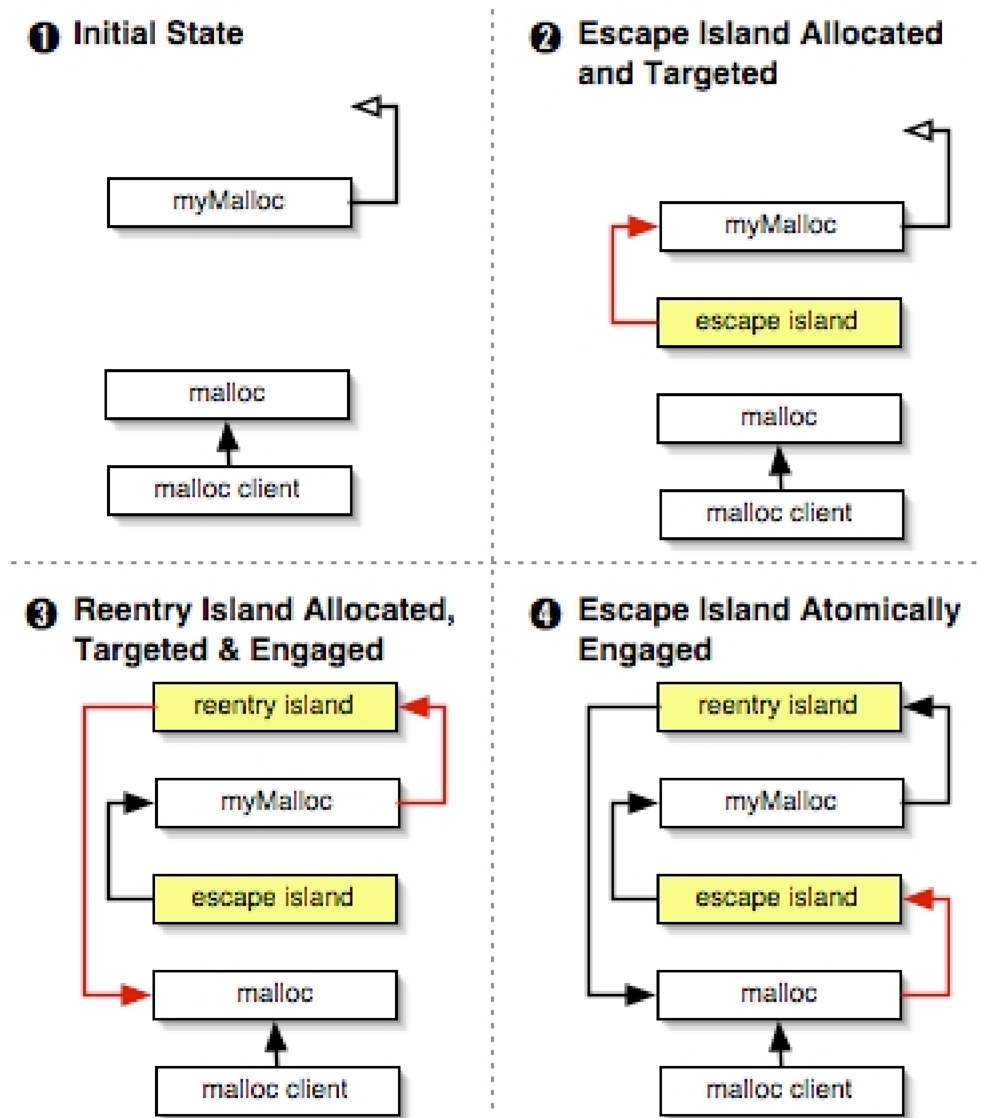
Both `b` and `ba` embed a 4-byte-aligned sign-extended 24-bit address into the 32-bit instruction. That means `b` can target ±32 megabytes relative to the current program counter, while `ba` can target the fixed low and high 32 megabytes of address space. The first 32 MB of a process's address space is rather busy with loaded code and whatnot. In addition, we often can't guarantee the overriding function is within 32 MB of the original function. Adding these up, it appears the best bet is to allocate a **branch island** at the end of the process address space. Fortunately, Mach supports a sparse memory model, which means the operating system won't try to allocate 4 gigabytes when we allocate a single page at the end of the address space.

This **escape branch island** will simply act as a layer of indirection, allowing us to use an otherwise address-limited single branch absolute instruction to effectively branch to any address. It will house seven instructions that will load the override function's address into the PowerPC's counter register and then branch to that address, preserving the link register. Here's the branch island template from this paper's supporting code (`mach_override.c`):

```
long kIslandTemplate[] = {
  0x9001FFFC,  //  stw    r0,-4(SP)      ; save off original r0 into red zone
  0x3C00DEAD,  //  lis    r0,0xDEAD      ; load the high half of address
  0x6000BEEF,  //  ori    r0,r0,0xBEEF   ; load the low half of address
```

```
  0x7C0903A6,  //  mtctr  r0           ; load target into counter register
  0x8001FFFC,  //  lwz    r0,-4(SP)    ; restore original r0
  0x60000000,  //  nop                 ; optional original first instruction
  0x4E800420   //  bctr                ; branch to the target in counter
register
};
```

Most of the time it's desirable for overriding function to call the original function. In order to perform this action, we must first execute the original's function's first instruction and then jump to its second instruction. Here, a **reentry branch island** can house the original first instruction and perform the branch to the second instruction.



Branch Islands

As alluded to in the preceding figure, the order of the allocation, targeting and engaging of the escape and reentry islands is important. By performing these operations in the proper order, the program is always in a correct, consistent state. Let's walk through the steps required to override a function:

1. **Discover the original function's address.** This can be accomplished using Mach's `_dyld_lookup_and_bind()` and `_dyld_lookup_and_bind_with_hint()` routines. A word of warning, however: if the symbol (the function's name) is not found, the link edit error handler is called. If you don't override the default handler, then and error message will be printed and the entire process will be terminated. Yikes! Fortunately you can use `NSIsSymbolNameDefined()` and/or `NSIsSymbolNameDefinedWithHint()` to discover if a symbol is defined before attempting to look up its address.
2. **Test the waters.** If reentry is desired (that is, the overriding function won't be a wholesale replacement for the original function, and will re-call the original function), then we should take some precautions. Specifically, we should be wary of overwriting instructions that cannot be relocated (such branch relative instructions, or branch instructions which update the link register) and the `mfctr` (move from counter register) instruction.

   It's weird but feasible that a function would begin with a branch relative instruction. Ideally, this instruction should be recognized, and the reentry's target should be aimed at the relative branch's destination.

   Weird and unfeasible is if the first instruction is a branch instruction that updates the link register or a `mfctr` instruction. Such a branch instruction would stomp on the caller's return address already in the link register — burning the bridge back — which makes precious little sense. Reading the counter register with a `mfctr` instruction also lacks sense, since calling conventions dictate the content of that register is unpredictable. If either of these are present, it probably means we're in uncharted waters and the overriding should be aborted.

3. **Make the original function writable.** By default, the page containing the original function's code will not be writable. Attempting to overwrite the instruction would result in process termination. The solution is straight-forward: use Mach's `vm_protect()` routine to mark the page writable.
4. **Allocate the escape branch island.** We can't simply use `malloc()` to allocate the escape island, since `malloc()` offers no control over placement of the block, and the island must be in the final 32 megabytes of the address space in order to be reachable from our single branch absolute instruction. Fortunately, Mach's `vm_allocate()` command allows that kind of control.
5. **Target the escape island and make it executable.** Once allocated, the escape island should be targeted at the overriding function and its instructions made executable via `msync(MS_INVALIDATE)`, which is largely equivalent to Carbon's `MakeDataExecuteable()` in terms of flushing the processor's data and instruction caches.
6. **Build the branch instruction.** Now that we know the escape island's address, we can craft the branch absolute instruction to replace the original first instruction.
7. **Optionally allocate and engage the reentry island.** If a reentry island is desired, now is the time to allocate it and return its pointer (usually in the form of overwriting a global function pointer to the "original" function).
8. **Atomically:**
   1. **Insert the original first instruction into the reentry island.** If a reentry island was requested, we should now put the original instruction into the reentry island to be executed upon reentry.
   2. **Target the reentry island and make it executable.** If the reentry island exists, we're now prepared to complete writing its instructions and dumping the processor's cache to make it stick.

3. **Swap the original function's first instruction with our custom-built branch instruction.** If the atomic instruction replacement fails, we'll need to loop and re-read the original first instruction and try again.

Code Injection

Function overriding is a useful technique in it's own right, but it only realizes half the goal of reattaining Mac OS 9-style system extension functionality. The other half is getting our code running where we want it.

Mac OS 9 made it straight-forward to globally replace or extend system-supplied functions with your own code. However Mac OS X, with its multiuser architecture and protected memory, makes this less obvious. Fortunately, if we explore the lower levels of Mach, we discover all the basic tools we need. Specifically, Mach offers two crucial abilities that allow us to "inject" and execute arbitrary code: remote memory allocation and remote thread creation.

Mach offers the ability for one process to allocate memory in another process's address space via the `vm_allocate()` call. You can populate to this "remote" memory block using `vm_write()`. Finally, `thread_create_running()` allows you to create a new thread in another process.

Together, these functions allow us to:

1. **Allocate the remote thread stack.** Due to the way the PowerPC passes parameters (in registers), all we have to do is set aside a buffer to be used as the thread's stack. No need to populate it with anything meaningful.
2. **Allocate and populate the remote thread's code.** While setting aside the code buffer is straight-forward, correctly populating it is less so. You first need to discover the desired code's parent Mach binary image using the dynamic loader APIs (`_dyld_image_count()`, `_dyld_get_image_header()`, `getsectbynamefromheader()`, `_dyld_get_image_vmaddr_slide()` and `_dyld_get_image_name()`) and then discover the image's size, which requires a roundtrip to the file system (to `stat()` the image file).
3. **Create and start a new thread pointing to the desired stack and code.** With the stack and code in place, we can now start the thread. The trickiest part is setting the thread's initial state, which requires we correctly set:
    - **The Program Counter.** In Mach's `ppc_thread_state_t` structure, the program counter register is named `srr0` and must point at the right offset into the remote code block.
    - **The Stack Pointer.** Here field `r1` must point into the remote stack block.
    - **Parameters.** Fields `r3` through `r10` are used for parameter passing into the remote thread's entry point.
    - **Link Register.** Since this is the entry point into a new thread, it's wise to set the thread's return address to something obviously wrong like `0xDEADBEEF`.

Once your code is running inside the target's process, you can override functions, load bundles and even use NSObject's `poseAsClass:` to override Cocoa applications. In general, you can fully control the target process and override it as you see fit.

The downside to this remote thread technique is resource leakage. While the thread can successfully stop itself, it cannot deallocate its own code and stack blocks. This leakage isn't major — it only happens once per code injection — but it's something we'd ideally avoid. One reasonable solution is to inject a new "injection manager" Mach server thread into a process. The idea is to inject a thread whose job it is to accept other injections via IPC. The "manager" thread would then be able to effectively recycle the resources used by later code injections. Indeed, the entire need to spawn a new thread per injection would disappear.

Summary

Mac OS X is a new operating system with new rules. Fortunately, there's still enough wiggle room for us to dynamically override aspects of the system that are incorrect or require enhancement.