Dynamically Overriding Mac OS X

Down the Rabbit Hole

Jonathan 'Wolf' Rentzsch
http://rentzsch.com

"These rules are no different than those of a computer system.

Some of them can be bent. Others... can be broken."

—Morpheus

What Happened to *How to Hack Mac OS X*?

- The paper's original title was "*How to Hack Mac OS X*"
- However, "hacking" is just too abused, especially in the Unix realm.
- Really, think "trap patching". However:
  - On Mac OS X, traps aren't used nearly as often.
  - "Patching" also has a specific definition in the Unix realm: applying textual differences to source code.
- Thus, "dynamic overriding".

Dynamic Overriding Defined

- The ability to modify software at runtime...
  - ...often in ways not originally anticipated by the software's original authors.
- Can suppress, change or extend existing functionality.
- Can add wholly new functionality to existing software.
- The modifications exist only in memory:
  - Never written to disk (save VM swapping).
  - Not permanent.
  - Easy to rollback — restart the app.

Dynamic Overriding: Made possible by two techniques that work together

Function Overriding	Code Injection

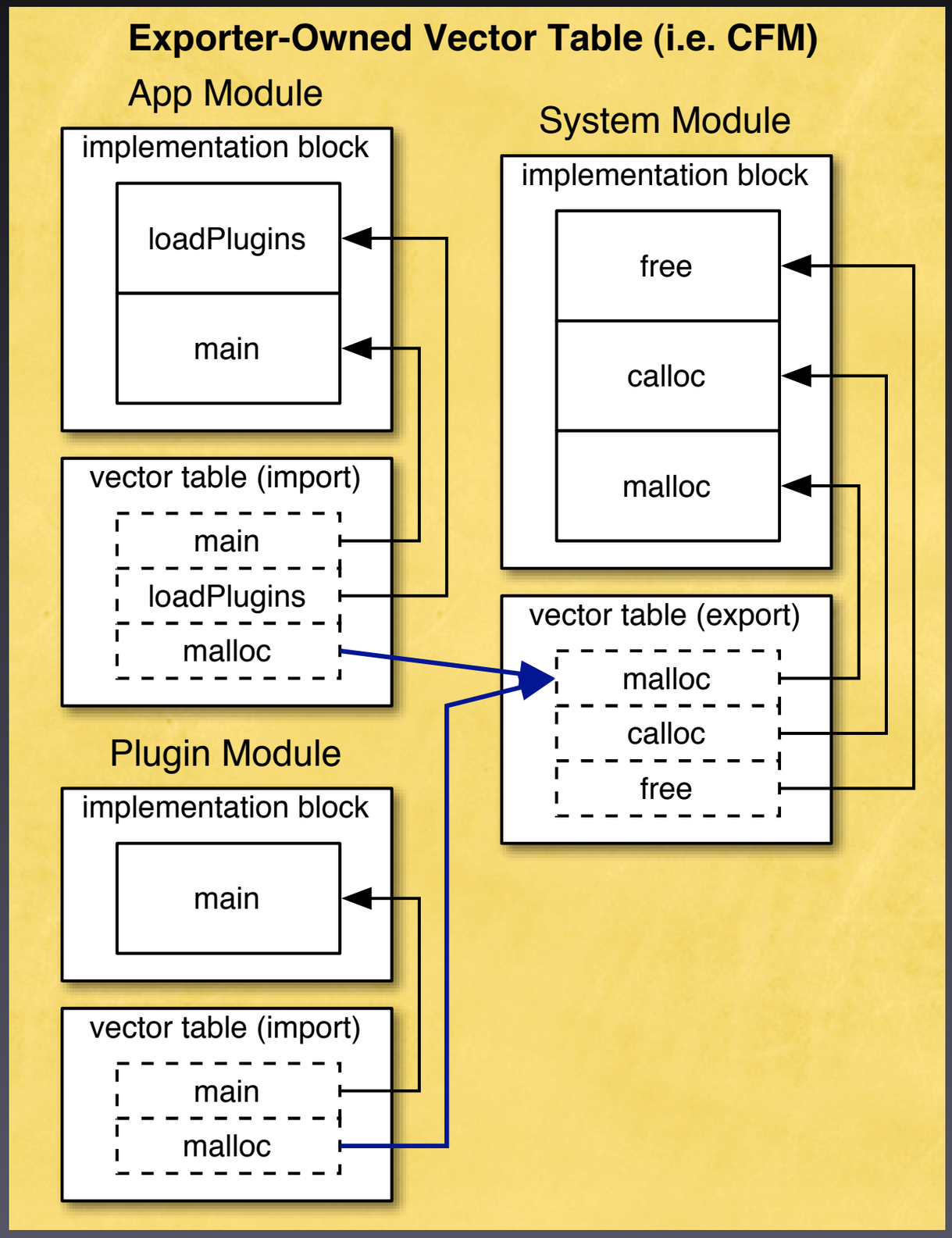
Function Overriding

- Unlike the classic Mac OS, Mac OS X does not directly support dynamically overriding system functions.
- At best, APIs exposed with Objective C interfaces can be overridden with categories and/or posing. Major limitations:
  - *Requires the API be exposed in ObjC.* Excludes all of Mach, BSD and Carbon.
  - *Overriding an ObjC wrapper does not override the original function.* Override will not effect all of the app if it has mixed procedural/objective code.

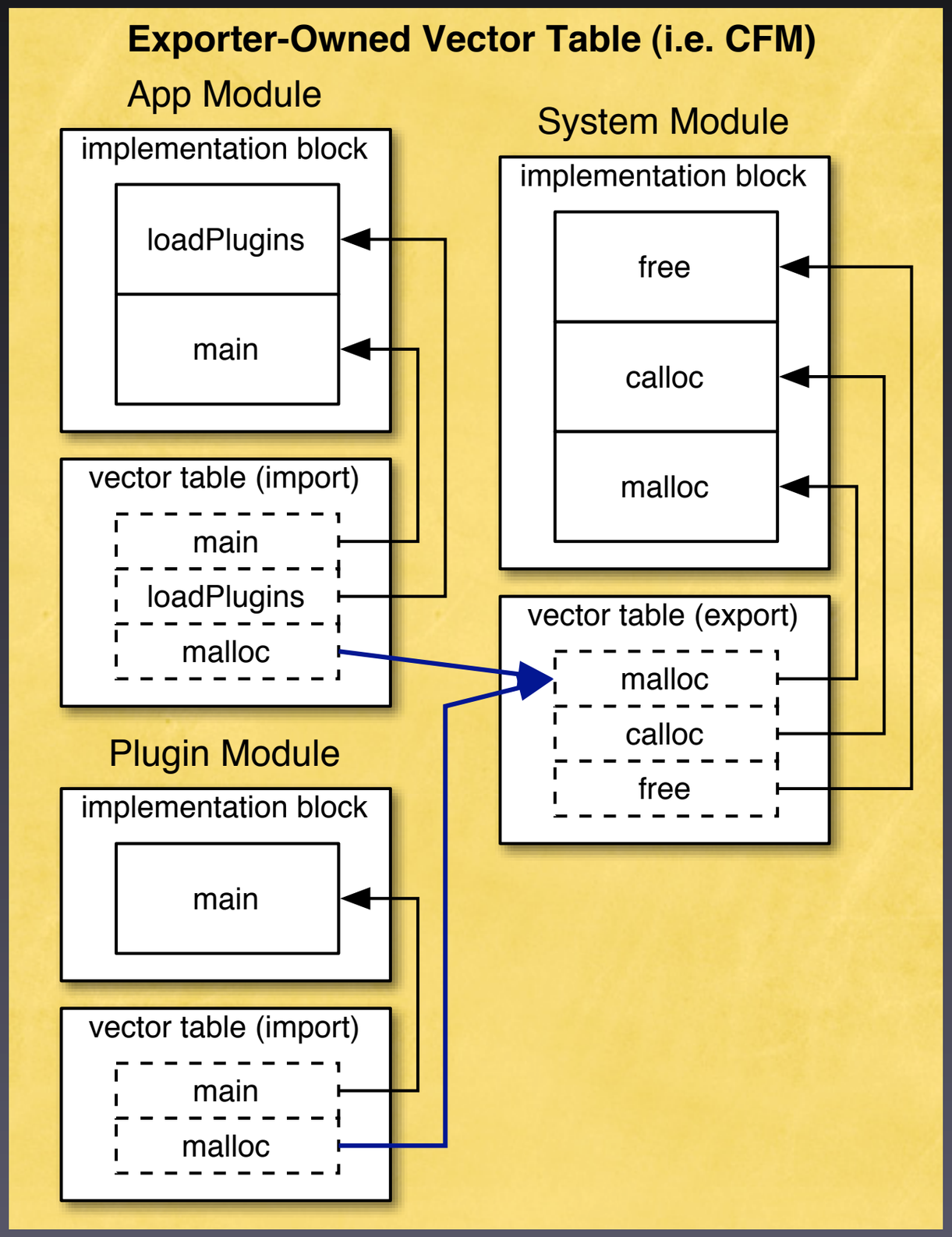
Function Overriding

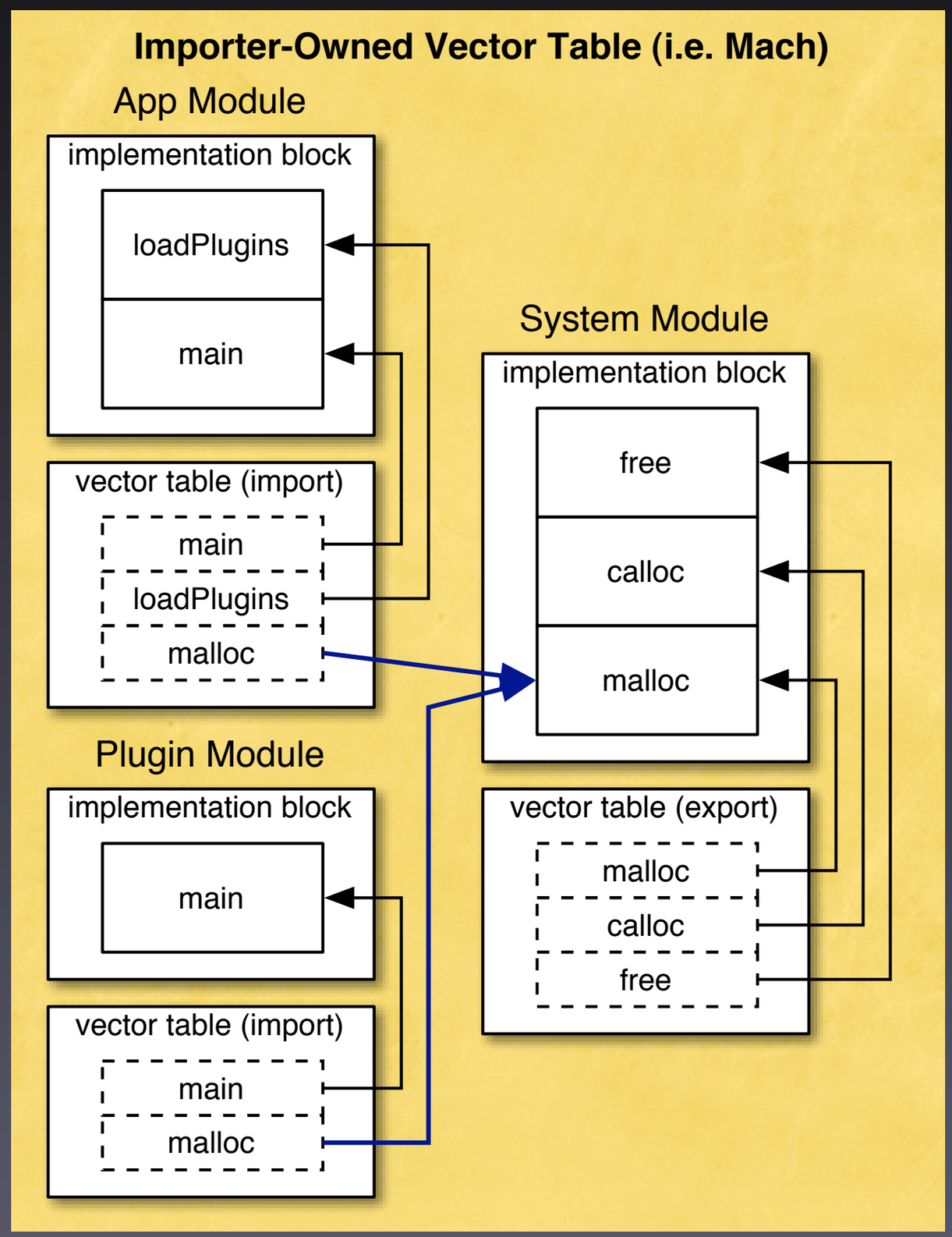
- Ideally, overriding a system function would be simple:
  - Discover the function's entry in a universal table.
  - Save off a pointer to the original code.
  - Replace it with a pointer to your overriding code.
- Alas, Mach's linking model lacks any sort of centralized bottleneck. This stands in contrast to CFM.

CFM's Model


Exporter-Owned Vector Table (i.e. CFM)
App Module
implementation block
loadPlugins
main
vector table (import)
main
loadPlugins
malloc
Plugin Module
implementation block
main
vector table (import)
main
malloc
System Module
implementation block
free
calloc
malloc
vector table (export)
malloc
calloc
free

CFM vs Mach


Exporter-Owned Vector Table (i.e. CFM)
App Module
implementation block
loadPlugins
main
vector table (import)
main
loadPlugins
malloc
System Module
implementation block
free
calloc
malloc
vector table (export)
malloc
calloc
free
Plugin Module
implementation block
main
vector table (import)
main
malloc


Importer-Owned Vector Table (i.e. Mach)
App Module
implementation block
loadPlugins
main
vector table (import)
main
loadPlugins
malloc
System Module
implementation block
free
calloc
malloc
vector table (export)
malloc
calloc
free
Plugin Module
implementation block
main
vector table (import)
main
malloc

Function Overriding

- You may think that you could walk all loaded modules to discover and rewrite all their vector tables.
- That won't work:
  - **Lazy binding** means symbols aren't resolved until they're first used. You'd have to force resolving of all symbols before rewriting: expensive and tricky.
  - All the work needs to be done again when a new module is loaded.
  - Won't work for symbols looked up programatically.

Function Overriding

- What **will** work: rewrite the original function implementation, *in memory*, itself.
- Basic premise: replace the original function's first instruction with a branch instruction to the desired override code.
- This technique is known as **single-instruction overwriting**.
- You may shudder now...

Single-Instruction Overwriting

- Benefits:
  - *Atomic replacement.* Safe in the face of multiple preemptive threads calling the original function.
  - *Less likely to harmfully impact the original code.* If you wish to reenter the original function from the override, you'll need to re-execute the replaced instruction. Moving less code around makes this more likely to work.
  - *Compatibility.* Works with the widest variety of function prologs and other patching implementations (including our own!)

Single-Instruction Overwriting

- Replacing a single instruction is good, but limiting.
- Can't branch to an arbitrary address, as that would take at least three instructions.
- Leaves us with branch instructions that encode their targets:
  - `b` (branch relative)
  - `ba` (branch absolute)
  - `bl` (branch relative, update link register)
  - `bla` (branch absolute, update link regiter)

Single-Instruction Overwriting

- `bl` and `bla` go away since they stomp on the link register — that houses the caller's return address!
- `b` and `ba` embed a 4-byte-aligned, 24-bit address.
- For `b`, that's ±32MB relative to the current program counter. We really can't guarantee our override will be within 32MB of the original function.
- For `ba`, that's the lowermost and uppermost 32MB of the current address space. The lowermost 32MB tends to be busy with loaded code and data.
- That leaves `ba`'s uppermost 32MB of address space.

The Branch Island

- We can allocate a single page at the end of the address space to hold our **branch island**.
- *(Mach's sparse memory model works nicely here.)*
- The branch island acts as a level of indirection, allowing the address-limited `ba` to effectively target any address.
- Two uses for branch islands: **escape** and **reentry**.

Branch Islands

- **Escape** branch island (required):
  - Used to jump **from** the original function **to** the overriding function.
  - Allocated at the end of the address space.
    - What our generated `ba` instruction points at.
- **Reentry** branch island:
  - Optional, only generated if you wish to reenter the original function.
  - Houses the original function's first instruction.

Inside the Branch Island

- C definition:

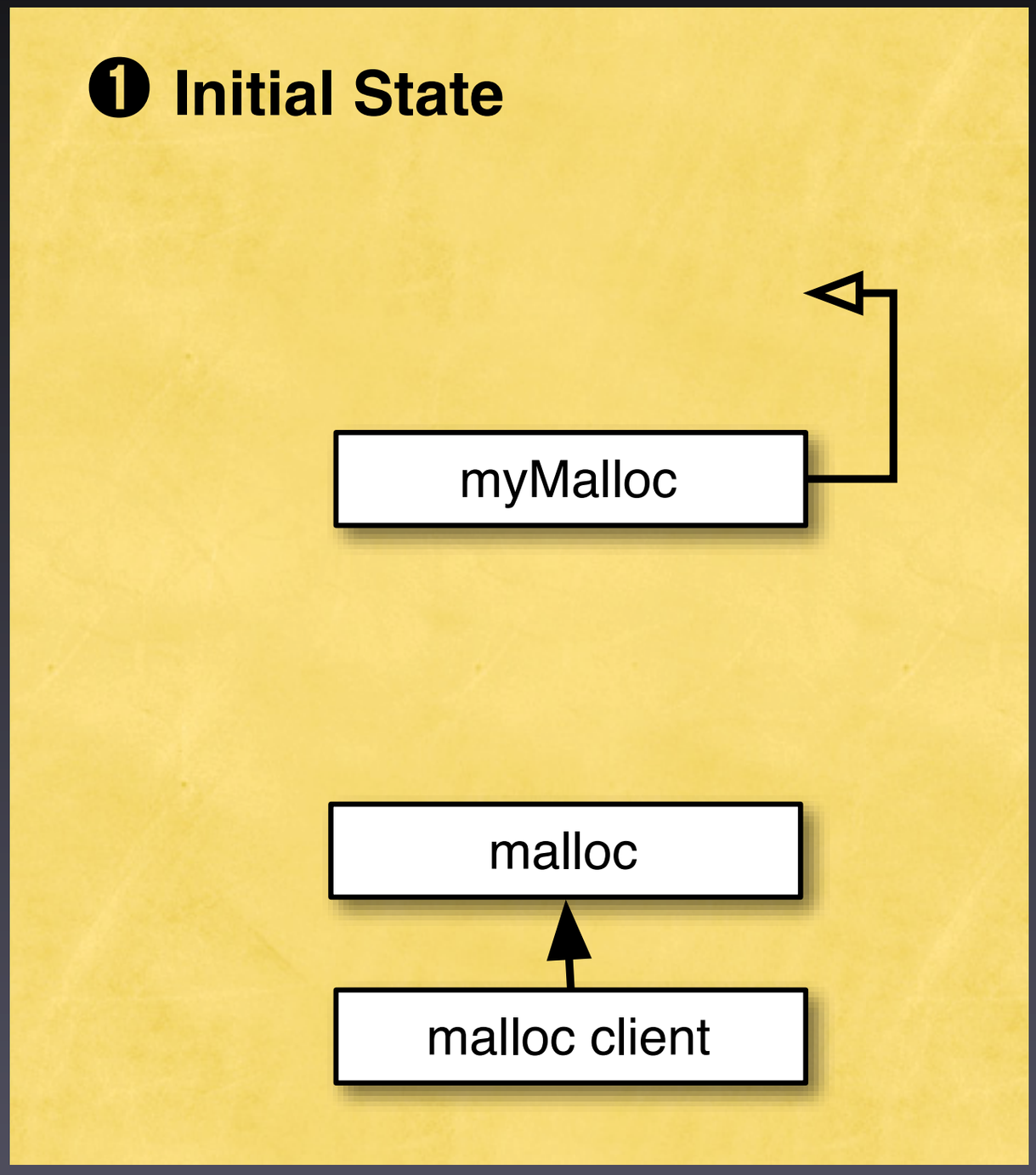
```
long kIslandTemplate[] = {
    0x9001FFFC,
    0x3C00DEAD,
    0x6000BEEF,
    0x7C0903A6,
    0x8001FFFC,
    0x60000000,
    0x4E800420
};
```

- What, you guys don't read machine language?!?

Branch Islands *for Dummies*

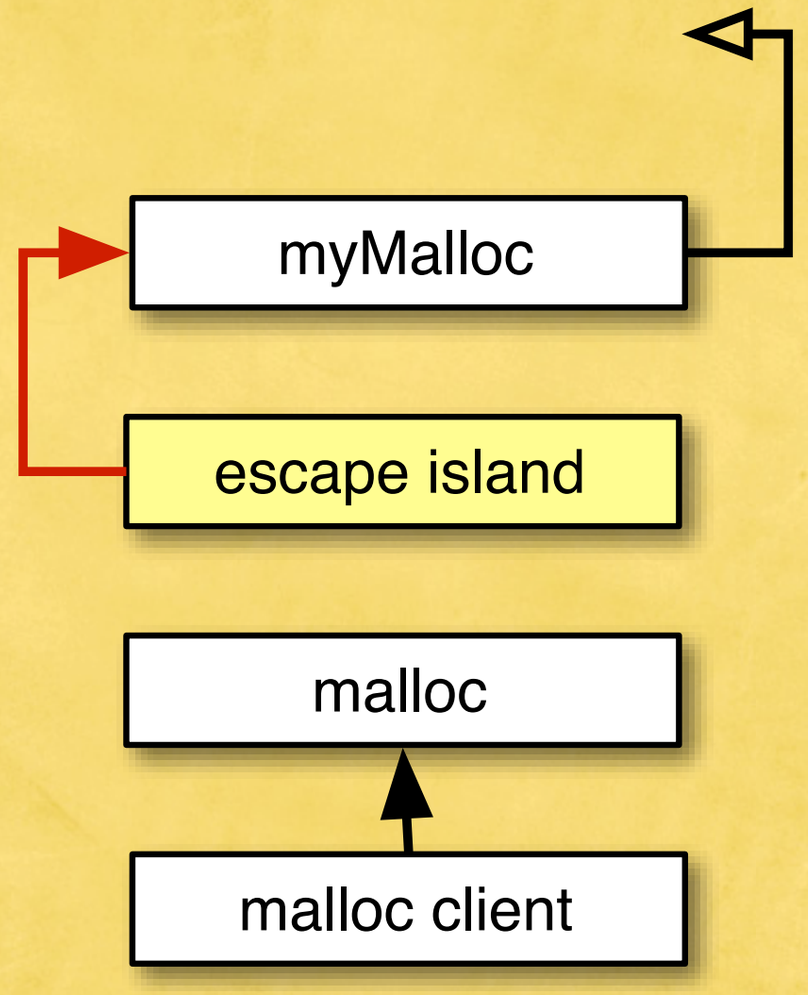
Opcode	Assembly	Comment
`0x9001FFFC`	`stw r0,-4(SP)`	save off original r0 into red zone
`0x3C00DEAD`	`lis r0,0xDEAD`	load the high half of address
`0x6000BEEF`	`ori r0,r0,0xBEEF`	load the low half of address
`0x7C0903A6`	`mtctr r0`	load target into counter register
`0x8001FFFC`	`lwz r0,-4(SP)`	restore original r0
`0x60000000`	`nop`	optional original first instruction (for
`0x4E800420`	`bctr`	branch to the target in counter register

Lame Override Animation

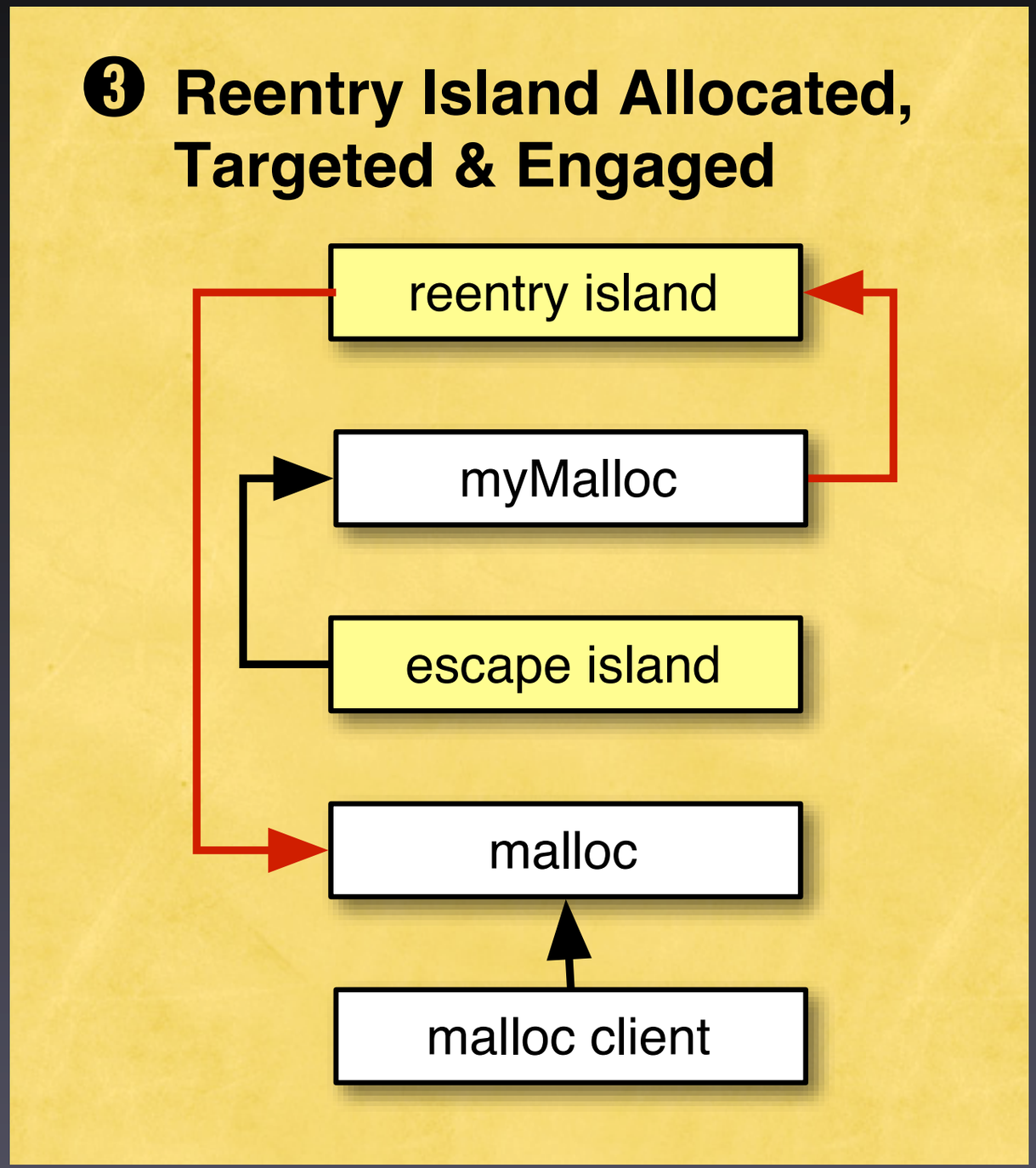


Lame Override Animation

❷ **Escape Island Allocated and Targeted**

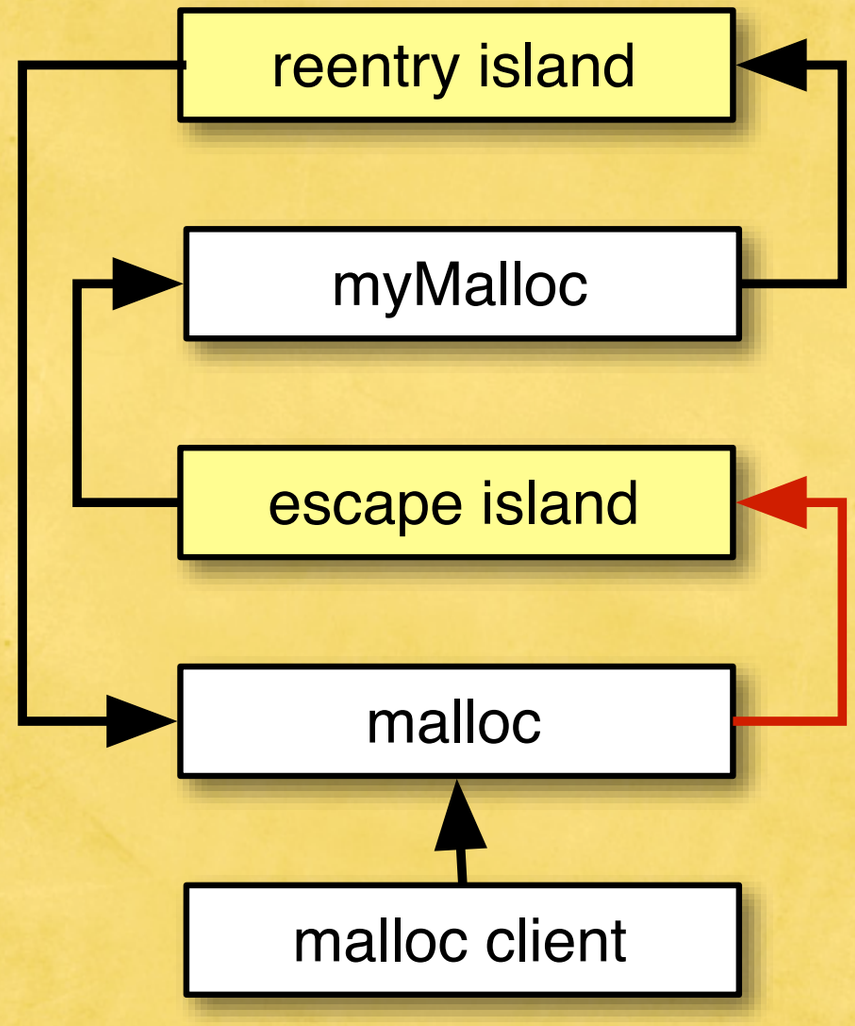


Lame Override Animation



Lame Override Animation

❹ Escape Island Atomically Engaged



Function Overriding Details

- *Discover the function's address.* Use `_dyld_lookup_and_bind[with_hint]()` and `NSIsSymbolNameDefined[WithHint]()`.
- *Test the waters.* Watch out for functions that start with the `mfctr` instruction.
- *Make the original function writable.* `vm_protect()`.
- *Allocate the escape island.* Use `vm_allocate()`.
- *Target the escape island and make it executable.* Use `msync()`.

Function Overriding Details

- *Build the branch instruction.* Target the escape island.
- *Optionally allocate and engage the reentry island.*
- Atomically:
  - *Insert the original function's first instruction into the reentry island.* If reentry is desired.
  - *Target the reentry island and make it executable.* Again, if reentry is desired.
  - *Swap the original function's first instruction with the generated `ba` instruction.* If atomic replacement fails (unlikely), loop and try again.

Code Injection Overview

- Function overriding is a powerful technique, but it's only attains half of our goal.
- By itself, we can only override system functions in our own software.
- Code injection, on the other hand, allows us override application and system functions in **any process**.
- Not just override functions, but inject new Objective C classes (example: for posing)

Code Injection Overview

- Mach supplies everything we need.
- *It's just not very well documented.* ;)
- Specifically, Mach provides APIs for:
  - Remote memory allocation. (`vm_allocate()`)
  - Remote memory I/O. (`vm_write()`)
  - Remote thread allocation. (`thread_create()`)
  - Remote thread control. (`thread_create_running()`)

Code Injection Details

- *Allocate a remote thread stack.* No need to populate it — parameters will be passed in registers.
- *Allocate and populate the thread's code block.* The gotcha here is determining the thread entry's code size. It's surprisingly hard, and requires a calling the dynamic loader APIs and `stat()`'ing the file system!
- *Allocate, populate and start the thread.* Set the thread control block's `srr0` to the code block, `r1` to the stack, `r3` through `r10` to parameters and `lr` to `0xDEADBEEF` (this thread should never return).

Code Injection Leakage

- While the injected thread can stop itself, it can't delete itself (it would need to deallocate its own stack and code while running).
- May be work-arounds, like the injected thread spawning another "normal" cleanup thread.
- Another solution is to install a permanent "injection manager" thread, that would start a Mach server to handle future injections via IPC.
  - Bonus feature: such an "injection server" would eliminate the need to start a new thread per injection.

Code & Doc Availability

- Two packages, both written in C:
  - *mach_override*: Implements function overriding.
    - http://rentzsch.com/mach_override
  - *mach_inject*: Implements code injection.
    - http://rentzsch.com/mach_inject
- Code is hosted by Extendamac *(BSD-style license)*
  - http://extendamac.sourceforge.net

Conclusion

“Whoa.”

—Neo