Code Signing – Hashed Out

Jonathan Levin

CTO
Technologeeks, LLC
@Technologeeks
Code Signing in Apple’s OSes

- Apple has introduced code signing as far back as OS X 10.5
  - Along with another major feature – then seatbelt, now sandbox

- Other OSes use signatures too, but Apple’s use is more advanced:
  - Provides the fulcrum for all system security measures
  - In OS X, creeping in as of 10.8 (via GateKeeper) and Mac App Store
  - In iOS, Mandatory as of day one*

* Proverbially. As in, around when apps started being available with app store. So maybe iOS 2.x or 3.x. But let’s not be nitpicky.
Motivation for Code Signing

- Obvious motivation: Authenticate software origin
  - Greatly mitigates any potential for malware as Apple vets its Devs

- Secondary motivation: Security profiles embedded in signature
  - OS X and iOS declarative security – entitlements – part of signature

- Unexpected bonus: Hegemony over software distribution
  - Only code signature allowed in iOS is Apple’s.
  - OS X still allows any signature (or even unsigned code). For how long?
Battle Plan

- Document technical specification of signature
  - Quick refresher on the Mach-O binary format as a prerequisite

- Explain Enforcement mechanisms (in iOS)
  - Come Face-to-Face with the adversary - AppleMobileFileIntegrity

- Examine bypass techniques up to and including iOS 8.1.2
Refresher: Mach-O binaries
Mach-O and Code Signatures

- Apple uses Mach-O as its binary format
  - Old binfmt tracing back to its NeXTSTEP origins
  - Many modifications introduced throughout OS X 10.x and iOS
  - No longer compatible with GNU HURD Mach-O
  - Never was and/or will be in any way compatible with ELF
For those familiar with ELF parlance:

<table>
<thead>
<tr>
<th>Mach-O...</th>
<th>Is ELF’s..</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment</td>
<td>Section</td>
</tr>
<tr>
<td>Section</td>
<td>N/A</td>
</tr>
<tr>
<td>/usr/lib/dyld</td>
<td>/usr/bin/ld</td>
</tr>
<tr>
<td>dylib (dynamic library)</td>
<td>so (Shared object)</td>
</tr>
</tbody>
</table>
Mach-O header consists of $n_{cmds}$ “Load commands”:

<table>
<thead>
<tr>
<th>Load command</th>
<th>Defines</th>
</tr>
</thead>
<tbody>
<tr>
<td>LC_SEGMENT[[_64]]</td>
<td>Memory regions with same r/w/x protection. Further contains sections. fileaddr [+ filesize] mapped to vmaddr [+vmsize]</td>
</tr>
<tr>
<td>LC_DYLD_INFO[ONLY]</td>
<td>Map of LINKEDIT for dynamic linker (DYLD)</td>
</tr>
<tr>
<td>LC_[DY]SYMTAB</td>
<td>Symbol tables</td>
</tr>
<tr>
<td>LC_LOAD_DYLINKER</td>
<td>Which dynamic linker to use (/usr/lib/dyld)</td>
</tr>
<tr>
<td>LC_MAIN (pre 10.8:UNIXTHREAD)</td>
<td>Entry point of executable</td>
</tr>
<tr>
<td>LC_LOAD_DYLIB</td>
<td>Dynamic library dependencies</td>
</tr>
</tbody>
</table>
Mach-O and Code Signatures

- Simple example: (/bin/ls from an ARMv8 iOS8)

Phontifex:/root # jtool -l -v /bin/ls

<table>
<thead>
<tr>
<th>LC 00: LC_SEGMENT_64</th>
<th>Mem: 0x000000000-0x100008000</th>
<th>File: Not Mapped</th>
<th>---/---</th>
<th>__PAGEZERO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mem: 0x100003cc8-0x100007288</td>
<td>File: 0x000003cc8-0x000007288</td>
<td>__TEXT.__text   (Normal)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem: 0x100007288-0x10000760c</td>
<td>File: 0x000007288-0x00000760c</td>
<td>__TEXT.__stubs (Symbol Stubs)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem: 0x10000760c-0x1000079a8</td>
<td>File: 0x00000760c-0x0000079a8</td>
<td>__TEXT.__stub_helper (Normal)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem: 0x100007b68-0x100007fb7</td>
<td>File: 0x000007b68-0x000007fb7</td>
<td>__TEXT.__cstring (C-String Literals)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem: 0x100007fb8-0x100008000</td>
<td>File: 0x000007fb8-0x000008000</td>
<td>__TEXT.__unwind_info</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LC 01: LC_SEGMENT_64</td>
<td>Mem: 0x100008000-0x10000c000</td>
<td>File: 0x8000-0xc000</td>
<td>rw-/rw-</td>
<td>__DATA</td>
</tr>
<tr>
<td>Mem: 0x00008000-0x00008038</td>
<td>File: 0x00008000-0x00008038</td>
<td>__DATA.__got (Non-Lazy Symbol Ptrs)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem: 0x00008038-0x00008290</td>
<td>File: 0x00008038-0x00008290</td>
<td>__DATA.__la_symbol_ptr (Lazy Symbol Ptrs)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem: 0x00008290-0x000084b8</td>
<td>File: 0x00008290-0x000084b8</td>
<td>__DATA.__const</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem: 0x000084c0-0x000084f0</td>
<td>File: 0x000084c0-0x000084f0</td>
<td>__DATA.__data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem: 0x000084f0-0x000085a8</td>
<td>Not mapped to file</td>
<td>__DATA.__bss (Zero Fill)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem: 0x000085a8-0x00008634</td>
<td>Not mapped to file</td>
<td>__DATA.__common (Zero Fill)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LC 02: LC_SEGMENT_64</td>
<td>Mem: 0x100008000-0x10000c000</td>
<td>File: 0xc000-0xd2f0</td>
<td>rw-/rw-</td>
<td>__LINKEDIT</td>
</tr>
<tr>
<td>Mem: 0x10000c000-0x10000e000</td>
<td>File: 0xc000-0xd2f0</td>
<td>r---/r---</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Code Signature Format
LC_CODE_SIGNATURE

- LC_CODE_SIGNATURE format is largely undocumented
  - That is, unless you go to the source: Security/libsecurity_codesigning
  - Enforcement (on OS X and XNU core) partially open source

- Apple provides two tools:
  - codesign(1): Generates, validates, and partially dumps signatures
  - codesign_allocate(1): Reserves space for load command in header

- Free, advanced tool: Jtool (http://NewOSXBook.com/files/jtool.tar)
  - otool(1) clone with many more options, esp. relevant to code signing
**LC_CODE_SIGNATURE**

- **LC_CODE_SIGNATURE** command points to a code signature “blob”
- Key component of blob is the “Code Directory”
  - Version: 20100*
  - Flags: none, or “adhoc”
  - Identifier: reverse DNS notation unique ID
  - CDHash: SHA-1 or other “mega-hash” of code slots

- Code signature can also be “detached”, i.e. separate from binary

* - Xnu 2782 sources show 20200 as well, including modifications to support team IDs
Code Slots

- File pages are individually hashed into “slots”, at indices 0+
- Ancillary data also hashed into “special slots”, at negative indices:

<table>
<thead>
<tr>
<th>Index</th>
<th>Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>−1</td>
<td>Bound Info.plist (Manifest)</td>
</tr>
<tr>
<td>−2</td>
<td>Internal requirements</td>
</tr>
<tr>
<td>−3</td>
<td>Resource Directory (_CodeResources)</td>
</tr>
<tr>
<td>−4</td>
<td>Application Specific (largely unused)</td>
</tr>
<tr>
<td>−5</td>
<td>Entitlements (bound in code signature)</td>
</tr>
</tbody>
</table>
Code Signature Format (OS X)

morpheus@Boreas (~)$ jtool --sig -v /bin/ls
Blob at offset: 29232 (5488 bytes) is an embedded signature of 4549 bytes, and 3 blobs
Blob 0: Type: 0 @36: Code Directory (261 bytes)
  Version: 20100
  Flags: none (0x0)
  Identifier: com.apple.ls
  CDHash: e8e766ea872cf682d5a5da3176f57ed140dfa75f
  # of Hashes: 8 code + 2 special
  Hashes @101 size: 20 Type: SHA-1
    Requirements blob: 34a9b54a874a8a0992f450a4d9a13f6bf3ee9edf (OK)
    Bound Info.plist: Not Bound
    Slot 0 (File page @0x0000): 75e0c5f20a84694cde3247b56ee1103a931d286a (OK)
    Slot 1 (File page @0x1000): ad20db05d1744ea7746baabb4da1d516ff91ae30 (OK)
    Slot 2 (File page @0x2000): f74b63f857ba9a44172352a146f8c213ea55afec9 (OK)
    Slot 3 (File page @0x3000): 532f0362c18af792be6d2447aad5fef07a05109f (OK)
    Slot 4 (File page @0x4000): 234b0c4fc286483a73ed2aff135ea1696ad5371 (OK)
    Slot 5 (File page @0x5000): 1de73e5f3b2587af89ce3d14c90f6e40f0eb173 (OK)
    Slot 6 (File page @0x6000): b0c138b5d7e10c104d94f49759fb447898 (OK)
    Slot 7 (File page @0x7000): 6300feb2aa97d8956596264a64f047cad3c2f638 (OK)
Blob 1: Type: 2 @297: Requirement Set (180 bytes) with 2 requirements:
  0: Designated Requirement (@28): Ident("com.apple.ls") AND Apple Anchor
  1: Library Requirement (@68): Ident("com.apple.libutil":) AND Apple Anchor OR
     Ident("libncurses.5") AND Apple Anchor OR
     Ident("libSystem.B") AND Apple Anchor
Blob 2: Type: 10000 @477: Blob Wrapper (4072 bytes) (0x10000 is CMS (RFC3852) signature)
Code Signature Format (iOS)

Phontifex:/ root# jtool -v --sig /bin/ls
Blob at offset: 53552 (448 bytes) is an embedded signature of 437 bytes, and 3 blobs
Blob 0: Type: 0 @36: Code Directory (381 bytes)
  Version: 20100
  Flags: adhoc (0x2)
  Identifier: com.apple.ls
  CDHash: bb98100b1ae8bc76f0384094542b6a1c802e742
  # of Hashes: 14 code + 2 special
  Hashes @101 size: 20 Type: SHA-1
  Requirements blob: 3a75f6db058529148e14dd7ea1b4729cc09ec973 (OK)
  Bound Info.plist: Not Bound
  Slot 0 (File page @0x0000): 4ea36bd97bfe568c38bee510bca3f3b54baaaf99 (OK)
  Slot 1 (File page @0x1000): 1ceaf73df40e531df3bfb26b4fb7cd95fb7bff1d (OK)
  Slot 2 (File page @0x2000): 1ceaf73df40e531df3bfb26b4fb7cd95fb7bff1d (OK)
  Slot 3 (File page @0x3000): f0b6158041cb2df9f9269d1490af9d8e7850d5f1 (OK)
  Slot 4 (File page @0x4000): 89dd50a17ad26c5290b0588027d87a8159855a (OK)
  Slot 5 (File page @0x5000): 8f402084bdcd6a837e9e0297a3590d8e2554dc (OK)
  Slot 6 (File page @0x6000): 9203c7ca528a8f133586e95e94c257786fa808dc (OK)
  ...
  Slot 11 (File page @0xb000): 1ceaf73df40e531df3bfb26b4fb7cd95fb7bff1d (OK)
  Slot 12 (File page @0xc000): fbeff9126c1c8de8079bbbc9c30f68d54c29c8fa (OK)
  Slot 13 (File page @0xd000): d8a6e4163274866d4fe943b8887213b18650ecac (OK)
Blob 1: Type: 2 @417: Empty requirement set
Blob 2: Type: 10000 @429: Blob Wrapper (8 bytes) (0x10000 is CMS (RFC3852) signature)
Superblob ends @36
Code Signature validation in XNU: on load

- Code directory hash is validated in its entirety – no individual hashes checked yet

Diagram:
- `mac_execve` and `posix_spawn`
- `bsd/kern/kern_exec.c`
- `exec_activate_image`
- `bsd/kern/kern_exec.c`
- `check_for_signature`
- `.to taskgated`
- `Set up main thread, etc.`
- `bsd/kern/mach_loader.c`
- `exec_mach_imgact`
- `load_machfile`
- `parse_machfile`
- `bsd/kern/mach_loader.c`
- `load_code_signature`
- `bsd/kern/mach_loader.c`
- `parse_machfile`
- `bsd/kern/mach_loader.c`
- `load_code_signature`
- `bsd/kern/ubc_subr.c`
- `ubc_cs_blob_add`
- `security/mac_vfs.c`
- `mac_vnode_check_signature`
Code Signature validation in XNU: page faults

Individual hashed checked on corresponding page’s page fault – if VM_FAULT_NEED_CS_VALIDATION

#define VM_FAULT_NEED_CS_VALIDATION(pmap, page)  \
   ((pmap) != kernel_pmap /*1*/ &&          \
   !(page)->cs_tainted /*2*/ &&              \
   (page)->object->code_signed /*3*/ &&         \
   !(page)->cs_validated || (page)->wpmapped /*4*/))
Code Signature Blob Handling

- Embedded signature blob (0xfade0cc0) copied to kernel space
  - Once verified, inaccessible by own process (loaded into UBC)

- Applications can use undocumented csops*(#169) syscall
  - Used extensively by dyld
  - Wrapped by Security.Framework SecTask* APIs
    - (primarily used for entitlements blob portion of code signature)
    - Which is why on jailbroken iOS binaries must be (pseudo)-signed for ent.

* And also csops_audittoken (#170), which functions similarly (both funnel to same internal implementation)
# CSOPS (#169)

<table>
<thead>
<tr>
<th>Flag (CS_OPS_*)</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>_STATUS _SETSTATUS</td>
<td>Return or set status of code signing for process</td>
</tr>
<tr>
<td>_MARKININVALID</td>
<td>Invalidate sig, possibly killing process on spot (if CS_KILL)</td>
</tr>
<tr>
<td>_MARKKILL</td>
<td>Set CS_KILL to Kill process if code signature is invalidated</td>
</tr>
<tr>
<td>_CDHASH</td>
<td>Return Code Directory Hash value</td>
</tr>
<tr>
<td>_ENTITLEMENTS_BLOB</td>
<td>Retrieve Entitlements</td>
</tr>
<tr>
<td>_MARKRESTRICT</td>
<td>(dyld) restrict library loading (like __RESTRICT segment)</td>
</tr>
<tr>
<td>_BLOB</td>
<td>Retrieve entire embedded blob (0xfade0cc0)</td>
</tr>
<tr>
<td>SIGPUP_INSTALL/DROP</td>
<td>10.9+: Used for CSR codesigning <em>(out of scope of this talk)</em></td>
</tr>
</tbody>
</table>
LC_DYLIB_CODE_SIGN_DRS

- LC_DYLIB_CODE_SIGN_DRS added in iOS 5.1 and OS X 10.8
- Resulting from a hack by Charlie Miller
  - Demonstrated arbitrary loading of unsigned dylibs by patching dyld

- Specifies Internal Requirements which must be satisfied on loading:
  - Commonly, logical AND/OR of (foreach library_identifier and anchor apple)
  - Requirement language supports certificates, entitlements, and much more
iOS: Text is always read-only

- In iOS (unlike OS X) mmap(2)/mprotect(2) are intentionally “broken”
  - Setting +w is disallowed on a +x segment (and vice versa)

- Apple provides an exception to this rule for JIT generated code
  - VM_FLAGS_MAP_JIT allows +wx simultaneously
  - Code “leaked”* in XNU-1699 (iOS5/OS X 10.7)

- Exception requires special dynamic-codesigning entitlement
  - Exception made for MobileSafari (iOS5+) with “nitro” javascript engine

* Code was #ifdef-ed as CONFIG_EMBEDDED. Apple has consistently (knowingly) leaked iOS code up to 1699. Why? Good question.
Enforcement
Enforcement in OS X*

- At present, code signing is **not enforced by default** on OS X
  - Gatekeeper does enforce GUI (Finder) launches, but not CLI/APIs
  - De facto useless since malware can run without LC_CODE_SIGNATURE

- Can be enforced system-wide via sysctl
  - Warning: incorrect setting will break some Apple binaries as well!

```bash
root@Zephyr (~)# sysctl vm | grep cs_
vm.cs_force_kill: 1         # set CS_KILL to kill (-9) on invalid page
vm.cs_force_hard: 0         # set CS_HARD to reject invalid page mapping
vm.cs_debug: 0              # diagnostic messages via dmesg(1)
vm.cs_enforcement: 1         # Set Require Enforcement by default (10.9/ios7)
vm.cs_enforcement_panic: 0   # Death before dishonor
```

* Greets: Patrick of SynAck – this slide is for you 😊 You missed my presentation, but I attended yours. All malware bases belongs to you!
Substrate: The MAC* framework

- Apple borrows/inherits the MAC Framework of Trusted BSD
- XNU is laced with `#if CONFIG_MACF` callouts (true in both iOS/OS X)
- Policy modules can call `mac_policy_register` to receive callouts
  - `nm/jtool -S|grep mac_policy_register` on `/System/Library/Extensions`
  - Can also check `kextstat(8)` for dependency on `com.apple.kpi.dsep`

* Mandatory Access Control: Linux/Android has similar (though incompatible) functionality in SELinux
Enter: AppleMobileFileIntegrity

- The AppleMobileFileIntegrity* kernel extension enforces codesigning
  - Also protects Mach task ports
  - Provides in-kernel entitlement support
- Works in cahoots with sandbox.kext
- Extremely paranoid (literally – panics kernel on unload or on failures)
- Very chatty (easy to decompile – plenty of verbose strings)

* Known to its friends – and enemies alike – as “AMFI” (uppercase). Lowercase (amfid) is used for daemon, discussed later
**AMFI as a MAC Policy**

- MPOs have some 330 (or so) callouts. AMFI cares about a dozen:

<table>
<thead>
<tr>
<th>Callout</th>
<th>Called by MACF when:</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpo_cred_check_label_update_execve</td>
<td>MAC Label* needs to be updated as a result of process launching (exec)</td>
</tr>
<tr>
<td>mpo_cred_label_init/associate/destroy</td>
<td>MAC Label* lifecycle</td>
</tr>
<tr>
<td>mpo_proc_check_interit_ipc_ports</td>
<td>resets task/thread ports for setuid/setgid programs</td>
</tr>
<tr>
<td>mpo_proc_check_mprotect</td>
<td>mprotect(2) invoked (iOS prevents r-x from ever getting +w)</td>
</tr>
<tr>
<td>mpo_proc_check_map_anon</td>
<td>mmap(2) invoked with MAP_ANON</td>
</tr>
<tr>
<td>mpo_proc_check_get_task</td>
<td>task_for_pid trap (the holy grail of debugging/tracing/pwning) invoked</td>
</tr>
<tr>
<td>mpo_vnode_check_exec</td>
<td>exec(2) is invoked</td>
</tr>
<tr>
<td>mpo_proc_check_cpumon</td>
<td>CPU Usage Monitoring parameters</td>
</tr>
<tr>
<td><strong>mpo_proc_check_run_cs_invalid</strong></td>
<td>Code Signature is invalid – AMFI gets a chance to save process</td>
</tr>
<tr>
<td><strong>mpo_vnode_check_signature</strong></td>
<td>Signature blob is added to Unified Buffer Cache</td>
</tr>
</tbody>
</table>

* MAC Labels are used in the implementation of sandboxing – but that’s for another presentation (and the book)
In Cache We Trust

- AMFI has a built-in “trust cache” with CDHashes of all iOS binaries
  - Used in “adhoc” (certificate-less) model – compares hashes only
  - Additional cache can be loaded from user mode, under some conditions
    - Loading process requires com.apple.private.amfi.can-load-trust-cache
    - Loaded cache must be a signed IMG3/IMG4 (encrypted/certified by Apple*)

- For all other binaries (read: App-store Apps) it uses amfid

* Loading the trust cache (via the UserClient) actually performs PKI in kernel mode, parsing DER and using corecrypto for PKCS#1
User mode process

Process performs a system call (or mach trap)

sysent/mach_trap_table

Corresponding function in kernel is called from table

Syscall/trap #n

Function calls out to Mandatory Access Control Framework

MACF

MACF checks if any policy modules requested to hook the particular functionality in their policy (recall: exec eventually called check_signature)

AMFI looks up CDHash of Mach-O in its trust cache (for adhoc binaries)

AppleMobileFile Integrity

AMFI hooks ..check_signature

If not adhoc, AMFI mach messages amfid to verify CMS (RFC3852) blob

amfid

amfid loads libmis.dylib to perform validation

sandboxd

sandbox

Sandbox uses AppleMatch for profile evaluation, and mach messages sandboxd

AppleMatch

Only if all MACF modules approve, syscall/mach_trap will be executed

The sandbox modules hooks most other functions
amfid

- User mode lackey of `AppleMobileFileIntegrity.kext`
- Communicates over host special port #18 (`HOST_AMFID_PORT`)
  - Mach port maintained by launchd, making it hard* to MiM
- Itself only a half-empty shell over `libmis.dylib`, which provides logic
  - Dylib handles “provisioning profiles”, allowing developer/enterprise apps
  - Boils down to `MISValidateSignature` and friends

* Hard – but not impossible
Reining in AMFI

- Can be disabled with several boot-args to XNU, including:

<table>
<thead>
<tr>
<th>Boot-Arg</th>
<th>Means</th>
</tr>
</thead>
<tbody>
<tr>
<td>amfi_allow_any_signature</td>
<td>Allow self-signed (basically, any signature)</td>
</tr>
<tr>
<td>cs_enforcement_disable</td>
<td>Invalid binaries may be loaded, won’t be killed (injected by TaiG)</td>
</tr>
<tr>
<td>(also unsets XNU’s vm.cs_enforcement)</td>
<td></td>
</tr>
<tr>
<td>amfi_get_out_of_my_way</td>
<td>Disables AMFI altogether</td>
</tr>
<tr>
<td>cs_debug*</td>
<td>Diagnostic IOLog() messages from AMFI</td>
</tr>
</tbody>
</table>

- iBoot no longer passes boot-args (as of iOS 5 on release devices)
  - boot-args can still be overwritten in memory (if you can exploit iBoot)

* Doesn’t actually disable, but provides information. There are other interesting boot args which AMFI uses for other purposes
AMFI: Coming to an OS X near you

- AMFI.kext and amfidi make their OS X debut in 10.10
- Presently securing kernel extensions by offering APIs
  - LC_CODE_SIGNATURE for kexts was introduced in 10.9 – but that’s kextd*
  - Main use: enforce entitlements at kernel levels
    - e.g AppleMobileFileIntegrity::AMFIEntitlementGetBool(proc*, char const*, bool*)
    - AppleMobileFileIntegrity::copyEntitlements(proc*)
- May very likely enforce everything in 10.11, or 11.0, etc.
- Likely coming soon: kernel-level entitlement support in OS X

- That kernel extension signatures are validated in user mode makes for many ways to bypass – q.v. OSXreverser, SynAck, etc.
Defeating Code Signing
A Brief History of Time (à la code signing)

- Apple’s code signature mechanism has evolved considerably
  - Some modifications introduced to support new security features
  - Most modifications coerced by successful hacks (reactive, not proactive)

- Presently (iOS 8.1.3-8.3, 8.4b) – no publicly known faults
  - But you never know about 0-days, now, do you?*

* Actually, there’s at least one more fault in the validation path. But TG won’t disclose/sell 0-days (nor bleat when others buy and use them)
Originally devised by SolarDesigner (return to libc)

Perfected as “Return Oriented Programming” to include:
- Set up of dummy stack frames (on 32-bit) or just return addresses
- Jump back into “gadgets” achieving assembly code snippets
- Gadgets **are validly signed code** – just used out of order!

- dyld_shared_cache (prelinked libraries) supplies plenty of gadgets
- Even more so on Intel architectures (instructions are variable length)
- Apple tries to secure framework code (SDK stubs), but just can’t do so.
Tried and True: Return Oriented Programming

- Wang, et. Al presented “Jekyll Apps” in USENIX ’13
  - Idea: App contains alternate code path to trigger deliberate ROP
  - Submitted to Apple, app passes code review, gets signature
  - Deployed on devices, phones home, then subverts ("hacks" itself)

- Solution: Tighter entitlements, white list model only
  - iOS 8 brings an explosive growth in entitlements, and move to XPC
Low hanging fruit: sysctl proc/vnode enforce

- Originally, the sysctl MIBs controlling code signins were read/write

  morpheus@Zephyr(~)$ sysctl -a | grep enforce
  ...
  security.mac.proc_enforce: 1
  security.mac.vnode_enforce: 1

- Root arbitrary code execution via ROP, then **sysctl –w** disable

- Solution: Apple made the MIB variables read only as of iOS 5
  - Can still be overwritten if a kernel memory overwrite is found
Forbidden fruit: stolen enterprise/dev certificates

- Apple willingly signs certain certificates, adding to chain of trust
  - Developer certificates (to deploy on device)
  - Enterprise certificates (to allow for org-internal apps)
    - Some jailbreaks set the i-Device date, which bypasses certificate expiration

- Apple can’t vet all the code that gets arbitrarily signed
  - Solution: Use restrictive entitlements, embedded in certificate
  - Problem: Still allows arbitrary code execution, within entitlement bounds
    - Allows for “failbreaks” and conducive to eventual jailbreaks
**Rotten fruit: DYLD_INSERT_LIBRARIES**

- Dyld’s equivalent of **LD_PRELOAD**
  - Just as powerful, just as nasty (especially if coupled with interposing)
- Several jailbreaks would force inject trojan libraries into processes
  - Apple got sick of that and introduced dyld restricting (ignoring variables)

<table>
<thead>
<tr>
<th>sRestrictedReason</th>
<th>Means</th>
</tr>
</thead>
<tbody>
<tr>
<td>restrictedBySetGUid</td>
<td>u+s or g+s</td>
</tr>
<tr>
<td>restrictedBySegment</td>
<td>__RESTRICT/__restrict section (e.g. amfid)</td>
</tr>
<tr>
<td>restrictedByEntitlements</td>
<td>code signed binaries with entitlements</td>
</tr>
</tbody>
</table>

- Plus, any injected library would have to be code signed.. Right?
Evasi0n (6/7): There is no TEXT

- Trojan libmis.dylib with an empty __TEXT.__text
  - No Text = No signature

- Redirect symbols:
  - _CFEqual
  - _MISValidateSignature (indirect for _CFEqual)
  - _kMISValidationInfoEntitlements (indirect for _kCFUserNotificationTokenKey)
  - _kMISValidationInfoIssuerCertificate (indirect for _kCFUserNotificationTokenKey)
  - _kMISValidationInfoSigningID (indirect for _kCFUserNotificationTokenKey)
  - _kMISValidationInfoValidatedByProfile (indirect for _kCFUserNotificationTokenKey)
  - _kMISValidationOptionAllowAdHocSigning (indirect for _kCFUserNotificationTokenKey)
  - _kMISValidationOptionExpectedHash (indirect for _kCFUserNotificationTokenKey)
  - _kMISValidationOptionLogResourceErrors (indirect for _kCFUserNotificationTokenKey)
  - _kMISValidationOptionUniversalFileOffset (indirect for _kCFUserNotificationTokenKey)
  - _kMISValidationOptionValidateSignatureOnly (indirect for _kCFUserNotificationTokenKey)
  - _CFEqual
Evasi0n (6): Overlapping Segments (Round I)

- Deliberately set two LC_SEGMENT commands to overlap
  - First command sets R-X (for executable code)
  - Second command sets R-- (not text)
  - Both commands have same vmaddr/vmsize

- `mmap(2)` called twice, and second mapping helps bypass check
  - Note segment overlap check is performed by dyld, not kernel
  - Kernel doesn’t care – code signature is only for text (+x) pages

- rdar://13145644

CVE-2013-0977: Fixed in iOS 6.1.3
Apple checks vmaddr + vmsize... But fails on an integer overflow!

Deliberately malformed negative vmaddr bypasses check!

<table>
<thead>
<tr>
<th>LC</th>
<th>Description</th>
<th>Mem:</th>
<th>File:</th>
<th>Permissions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LC_SEGMENT</td>
<td>0xffffffff000000000</td>
<td>0x0-0x1000</td>
<td>r-x/r-x</td>
<td>__FAKE_TEXT</td>
</tr>
<tr>
<td>01</td>
<td>LC_SEGMENT</td>
<td>0xffffffff000000000</td>
<td>0x2000-0x3000</td>
<td>r--/r--</td>
<td>__TEXT</td>
</tr>
<tr>
<td>02</td>
<td>LC_SEGMENT</td>
<td>0x00001000-0x00002000</td>
<td>0x1000-0x10bb</td>
<td>r--/r--</td>
<td>__LINKEDIT</td>
</tr>
<tr>
<td>03</td>
<td>LC_SYMTAB</td>
<td>Symbol table is at offset 0x0 (0), 0 entries</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>String table is at offset 0x0 (0), 0 bytes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Export info: 576 bytes at offset 4096 (0x1000-0x1240)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>06</td>
<td>LC_ID_DYLIB</td>
<td>/usr/lib/libmis.dylib (compatibility ver: 1.0.0, current ver: 1.0.0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>07</td>
<td>LC_LOAD_DYLIB</td>
<td>/System/Library/Frameworks/CoreFoundation.framework/CoreFoundation (compatibility ver: 65535.255.255, current ver: 0.0.0)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CVE unknown, but (sort of) fixed in 8
Pangu (8): Overlapping Segments (Round III)

- Apple only checks first segment.. (< 8.1.2) So Pangu8 fakes second:

```
Pademonium:/ root# ARCH=armv8 jtool -v -l libmisPangu.dylib | more
LC 00: LC_SEGMENT_64 Mem: 0x000000000-0xc0000 File: 0x0-0xc0000   r-x/r-x __TEXT
     Mem: 0x000000000-0x00000000 File: 0x00004000-0x00004000 __TEXT.__text
LC 01: LC_SEGMENT_64 Mem: 0xffffffffffffc000-0x0   File: 0xc0000-0x184000 rw-/rw- __TEXT1
LC 02: LC_SEGMENT_64 Mem: 0x0000c8000-0xcc000 File: 0xc8000-0xc8794 r--/r-- __LINKEDIT
LC 03: LC_ID_DYLIB/usr/lib/libmis.dylib (compatibility ver: 1.0.0, current ver: 255.0.0)...
```

- Note the first segment is still technically empty (no code in section)

CVE-2014-4455: Fixed in iOS 8.1.2
Pangu (8): Overlapping Segments (Round III)

- Binary is loaded, **__TEXT** segment overlaps, and “resets” protection

```bash
Pademonium:/ root# ARCH=armv8 jtool -v -l libmisPangu.dylib | more
LC 00: LC_SEGMENT_64 Mem: 0x000000000-0xc0000 File: 0x0-0xc0000 r-x/r-x __TEXT
    Mem: 0x000000000-0xc0000000 File: 0x00004000-0xc0004000 __TEXT.__text
LC 01: LC_SEGMENT_64 Mem: 0xffffffffffffc000-0x0 File: 0xc0000-0x184000 rw-/rw- __TEXT1
LC 02: LC_SEGMENT_64 Mem: 0x0000c8000-0xcc000 File: 0xc8000-0xc8794 r--/r-- __LINKEDIT
LC 03: LC_ID_DYLIB /usr/lib/libmis.dylib (compatibility ver: 1.0.0, current ver: 255.0.0)
```

```bash
PADEMONIUM:/ root# ARCH=armv8 jtool -v -l libmisPangu.dylib | more
LC 00: LC_SEGMENT_64 Mem: 0x000000000-0xc0000 File: 0x0-0xc0000 r-x/r-x __TEXT
    Mem: 0x000000000-0x000000000 File: 0x0-0xc0000 __TEXT.__text
LC 01: LC_SEGMENT_64 Mem: 0xffffffffffffc000-0x0 File: 0xc0000-0x184000 rw-/rw- __TEXT1
LC 02: LC_SEGMENT_64 Mem: 0x0000c8000-0xcc000 File: 0xc8000-0xc8794 r--/r-- __LINKEDIT
LC 03: LC_ID_DYLIB /usr/lib/libmis.dylib (compatibility ver: 1.0.0, current ver: 255.0.0)
```

---

**CVE-2014-4455:** Fixed in iOS 8.1.2
TaiG: Overlapping Segments (Round IV)

- Apple adds checks for vmsize and filesize all over, but not vmaddr...

```
morpheus@Zephyr(~)$ ARCH=armv7 jtool -l -v ~/iOS/JB/TaiG/libmis.dylib
LC 00: LC_SEGMENT Mem: 0x00000000-0x00001000 File: 0x0-0x1000 __TEXT
Mem: 0x00001000-0x00001000 File: 0x00001000-0x00001000 __TEXT.__text (Normal)
LC 01: LC_SEGMENT Mem: 0x00001000-0x00002000 __LINKEDIT
LC 02: LC_ID_DYLIB /usr/lib/libmis.dylib
...
LC 16: LC_SEGMENT Mem: 0xffffffffffffc000-0x1fffc000 File: 0xc000-0x14000 __DATA
```

```
morpheus@Zephyr(~)$ ARCH=armv8 jtool -v -l ~/iOS/JB/TaiG/libmis.dylib
LC 00: LC_SEGMENT_64 Mem: 0x00000000000-0x4000 File: 0x0-0x1000 __TEXT
Mem: 0x00001000-0x00001000 File: 0x00001000-0x00001000 __TEXT.__text (Normal)
LC 01: LC_SEGMENT_64 Mem: 0x00001000-0x00002000 File: 0xa000-0xa618 __LINKEDIT
LC 02: LC_ID_DYLIB /usr/lib/libmis.dylib
...
LC 16: LC_SEGMENT_64 Mem: 0xfffffffffffff000-0x1ffff000 File: 0xa000-0xc000 r--/r-- __DATA
```

CVE-2014-4455: Really Fixed in iOS 8.1.3
TaiG: Overlapping Segments (Round IV)

- Once again, overlap occurs.

```
morpheus@Zephyr(~)$ ARCH=armv8 jtool -v -l ~/iOS/JB/TaiG/libmis.dylib
LC 00: LC_SEGMENT_64    Mem: 0x000000000-0x4000  File: 0x0-0x1000 __TEXT
    Mem: 0x000004000-0x000004000 File: 0x00001000-0x00001000 TEXT.__text (Normal)
LC 01: LC_SEGMENT_64 Mem: 0x000004000-0x8000 File: 0xa000-0xa618 __LINKEDIT
LC 02: LC_ID_DYLIB /usr/lib/libmis.dylib
...
LC 16: LC_SEGMENT_64 Mem: 0xfffffffffffff000-0x1fffc000 File: 0xc000-0x14000 r--/r-- __DATA
Phontifex:/ root# DYLD_PRINT_SEGMENTS=1 DYLD_INSERT_LIBRARIES=/tmp/libmis.taig.dylib ls > /dev/null
dyld: Main executable mapped /bin/ls
...
dyld: Mapping /tmp/libmis.taig.dylib (slice offset=65536)
   __TEXT at 0x400000000->0x40000fff with permissions r.x
   __LINKEDIT at 0x40001000->0x40001617 with permissions r..
   __DATA at 0x3ff00000->0x40000fff with permissions r..
```

CVE-2014-4455: Really Fixed in iOS 8.1.3
Hypotheticals (?)

- SHA-1 2nd Preimage will entirely defeat signatures, but ..
  - We don’t have that yet(?)
  - Signatures decoupled from Algorithms, and could migrate to 256 or NG

- Any kernel memory overwrite – even 32-bits - will defeat enforcement
  - XNU is getting more secure, but there’s always IOKit and IOHID/IOUSB 😊
Summary & Takeaways
Apple clearly thought out code signatures..

- Elegant, cryptographically secure mechanism
  - Uses hash-of-hashes technique
  - Implementation decoupled from hash specifics
  - SHA-1 (still) secure, hash easily upgradeable to SHA-256 or NG

- Used as substrate for overall system security
  - Intricately tied to entitlements and sandboxing
But elegant design makes for multiple fractures

- User mode components are inherently weak
  - Can’t fully validate daemon identity and integrity
  - Dynamically linked binary prone to library injection/replacement
  - Loader (dyld) *still* has bugs aplenty, amfid too critical a component

- Even pure kernel mode implementation can be broken
  - Kernel arbitrary memory overwrite can disable KEXT

- Implementation faults, implementation faults, implementation faults!
For more information

- Pangu’s CANSEC-West Talk 2015 – Covers their jailbreaks in depth
  - Awesome talk straight from the horse’s mouth

- **Mac OS X and iOS Internals (1st)** – With 2nd Edition (vol I) due summer ’15
  - Still open for last minute requests: [http://NewOSXBook.com/TOC2.html](http://NewOSXBook.com/TOC2.html)

- [http://NewOSXBook.com](http://NewOSXBook.com) – Book’s Web site, lots of free tools/articles