GeekCon

Defeating Code Signing on iOS 16 using a PPL bypass

A look at the code signing bypass technique used in the Dopamine jailbreak

@opa334dev

About Me

- Researcher from Germany
- GitHub @opa334, Twitter @opa334dev
- Employed at Cellebrite Labs (We're hiring!)
- Software developer (Since 2017)
- Jailbreak developer (Since 2022)
  - Released TrollStore in 2022
  - Released Dopamine 1.x (iOS 15.0 - 15.4.1) in 2023
  - Released Dopamine 2.x (iOS 15.0 - 16.5) in 2024

01

section

Basics

Motivations

- Code signing bypass is an integral component of jailbreaks
- Previously this was done using
  - Kernel patches (Until iOS 10)
  - Expired enterprise certificates + amfid patch (Until iOS 15)
- Previous techniques do not work anymore
- Code signing is behind PPL now (PMAP_CS, introduced in iOS 15)
- Goal: Make any binary be able to be executed, even ones not signed by Apple

arm64e Post Exploit Mitigations

- Kernel Text Read-only Region (KTRR)
  - Protects kernel code and static data
- Kernel Page Protection Layer (PPL)
  - Protects important data (“hypervisor”)
- Kernel Pointer Authentication (PAC / Control Flow Integrity)
  - Protects important data (via Cryptographic Signatures)

Kernel Memory (KTRR + PAC + PPL)

Read

Constant Data (KTRR protected)

Code (KTRR protected)

Read / Execute

Read / Write

Volatile Data (PPL protected)

Volatile Data (PAC protected)

Volatile Data

Assumptions

- We have chained a Kernel exploit (landa, CVE-2023-41974) and a PPL bypass (Operation Triangulation, CVE-2023-38606)
  - Operation Triangulation bug can also in theory be used as a limited KTRR bypass, but we will only use it for PPL
- Our app therefore has PPLRW
- No PAC bypass however, therefore no Kcall primitive

- How do we use this to bypass code signing?

02

section

Primitives & Handoff

PPL R/W Handoff

- Integral component of modern jailbreaks
- Pass physical read / write primitives (including PPL protected pages) to another process
- In practice, we will archive this by modifying the page table

PPL R/W Handoff

- App exploits PPL bug to get initial primitive, then spawns jailbreak server and passes primitives to it
- Jailbreak server then preserves the primitives and uses them to provide jailbreak features to other processes
- During initial jailbreak, use PPL bypass to handoff PPL R/W to our own process

Memory Management on *OS

Page Tables

pmap->tte

0x0000000000	[Entry 1]
0x1000000000	[Entry 2]
0x2000000000	[Entry 3]
0x3000000000	[Entry 4]
0x4000000000	[Entry 5]
0x5000000000	[Entry 6]
0x6000000000	[Entry 7]
0x7000000000	[Entry 8]

Level 1

0x1000000000	[Entry 1]
0x1002000000	[Entry 2]
0x1004000000	[Entry 3]
0x1006000000	[Entry 4]
0x1008000000	[Entry 5]
	(...)
0x1FFC000000	[Entry 2047]
0x1FFE000000	[Entry 2048]

Level 2

0x1002000000	[Entry 1]
0x1002004000	[Entry 2]
0x1002008000	[Entry 3]
0x100200C000	[Entry 4]
0x1002010000	[Entry 5]
	(...)
0x1003FF8000	[Entry 2047]
0x1003FFC000	[Entry 2048]

Level 3

pmap Layer

- Provides routines to interact with physical pages
  - Every process owns a vm_map object
  - Every vm_map object owns a pmap object
- On arm64e: Implemented in PPL mode, all data structures are protected
- Backed by actual page table used by the MMU of the specific process (pmap->ttep is TTBR0 at runtime)
- Every mapping is registered in the pmap layer and then reflected in the backed page table
- Allocates and deallocates page tables as needed

pmap Layer

- Every physical page has a descriptor (pt_entry / pt_desc)
- pv_head_table: pv_entry_t[], one entry for every existent physical page (gPhysBase -> gPhysBase + gPhysSize)
- Every page has a type, indicating what it is used for
  - PVH_TYPE_NULL: Not mapped
  - PVH_TYPE_PVEP: Page mapped in multiple pmaps
  - PVH_TYPE_PTEP: Page mapped in one pmap
  - PVH_TYPE_PTDP: Page used as Page Table
- More attributes, depending on the type

VM Layer

- Interacts with pmap layer to provide virtual memory to userspace processes
- Every mapping represents one vm_entry being added to the tasks map attribute
- VM Layer handles memory faults and maps in the backing memory as needed

VM Layer + pmap Layer: Example

Userland

```
void *page = malloc(0x4000);

*(uint32_t *)page = 0x4141;
```

Kernel

vm_allocate
↳ vm_map_enter

handle_user_abort
↳ vm_fault
  ↳ pmap_enter_options

Injecting Page Table Entries

- To map arbitrary kernel pages into our process, we need to be able to allocate level 2 and level 3 page tables
- If we don’t do this, the system will panic on process exit, because it can’t clean up page tables that haven’t been properly allocated
- Avoid mapping to addresses that are managed by the VM layer
  - Our injected page table entries are neither added to the VM layer, nor the pmap layer
  - If the system attempts to map something else on top of our mapping, that will either destroy our mapping or crash the system

VM Layer Coverage

- VM layer covers only a small chunk of memory (0x102aa4000 to 0x3d8000000 on my iPhone 13 Pro on 15.1.1) of the actual page table
- Other addresses are always unused
- When the jailbreak maps a kernel page, it needs to do so in unused area to avoid colliding with regular allocations

Allocate Page Tables

- Every page table has an associated pt_desc / ptd_info
- A reference to this exists in the pv_entry structure of the page that contains the page table allocation

```
typedef struct pt_desc {
    queue_chain_t pt_page;
    struct pmap *pmap;
    vm_offset_t va[PT_INDEX_MAX];
    ptd_info_t *ptd_info;
} pt_desc_t;
```

```
typedef struct {
    unsigned short refcnt;
    unsigned short wiredcnt;
} ptd_info_t;
```

Allocate Page Table (without Kcall)

- posix_memalign(&lvl2_addr, L2_BLOCK_SIZE, L2_BLOCK_SIZE)
  - Allocate the entire span of an L2 page table (0x2000000 bytes)
  - Faulting in one page in it will allocate the page table
  - We now have full control over this page table and can deallocate it at will
- Find pt_desc of page table
- Artificially bump pt_desc->ptd_info->refcnt using PPLRW
- free(lvl2_addr)
- Decrease reference count
- Page table is now still allocated and associated with process

Allocate Page Table (without Kcall)

- The allocated page table will be inside the memory that's covered by the VM layer
- We need to move it elsewhere to avoid collisions
- We also want to be able to give it to other processes
  - pt_desc->pmap = <target pmap>
  - pt_desc->va[0] = <target pmap va start>
- Setting these will prevent the system from panicing when the process exits
- Insert the allocated page table into the target pmap's ttep
- Can be used as both level 2 and level 3 page table

```
typedef struct {
    unsigned short refcnt;
    unsigned short wiredcnt;
} ptd_info_t;
```

```
typedef struct pt_desc {
    queue_chain_t pt_page;
    struct pmap *pmap;
    vm_offset_t va[PT_INDEX_MAX];
    ptd_info_t *ptd_info;
} pt_desc_t;
```

Full Kernel Mapping

- Read gPhysStart, gPhysSize to determine what pages need to be mapped
- Write logic that maps PA to VA by allocating page tables as needed
- Call said logic to map all pages between (gPhysStart) and (gPhysStart + gPhysSize) to between (gPhysStart + 0x7000000000) and (gPhysStart + gPhysSize + 0x7000000000)
- Phys R/W can now be done by simply adding 0x7000000000 to the physical pointer and dereferencing it!
- Fastest possible way of doing Kernel R/W

Full Kernel Mapping

03

section

Bypassing Code Signing

Code Signing (iOS >=15)

iOS 16+: Developer Mode

- In iOS 16, various entitlements are locked behind developer mode
  - E.g. task_for_pid-allow
- Stock flow to enable developer mode is complicated
  - Attempt to sideload an app
  - Enable toggle in settings
  - Reboot device, afterwards there will be a confirmation alert
- At runtime, the developer mode is stored in ppl_developer_mode_storage global bool
- Variable is PPL protected, using PPL R/W we can enable developer mode in a non persistent way by writing to it

Bypassing Code Signing

- Using PPL R/W, we can inject arbitrary CDHashes into a dynamic Trust Cache
- We first bundle a few binaries with the jailbreak and add them to trust cache
- This allows us to run code with arbitrary entitlements
- But in the end, we would like to modify the system to the point where *ANY* binary can run
- We need to automate adding the CDHash to Trust Cache for any binary that is attempted to be spawned and any library that is attempted to be loaded

Bypassing Code Signing

- All binaries are spawned via calling posix_spawn (and a few other functions) from userland
- launchd: Parent process of the entire userland
- Hook posix_spawn inside every process, add binary to Trust Cache, then call original implementation
- We can use a launchd task port to load a library into it (By creating a thread that calls dlopen)
- In this library we can hook posix_spawn
- But how do we extend this hook to the entire system?

Jailbreak Server (inside launchd)

- Inside launchd, hook XPC message handler to add additional logic to it
- After the app has ran the exploit chain and injected the library into launchd, the library will use XPC to communicate with the app to transfer PPL R/W to itself
- Once obtained, launchd can now provide the rest of the system with APIs to interact with the Kernel using PPL R/W
- For example, it can expose a call to add an executable to Trust Cache

System Wide Hook

- Idea: One library injects system wide and hooks relevant functions inside every single process
- If we hook posix_spawn, we can add custom environment variables to any processes spawned
- Using this, we can add DYLD_INSERT_LIBRARIES to reinsert the system hook library
- Unfortunately, dyld will ignore this environment variable under normal circumstances

Dyld: AMFI Flags

- Dyld calls out to Kernel to get “AMFI flags” for the process it runs in
- These flags dictate which environment variables are allowed
- DYLD_INSERT_LIBRARIES is only allowed if the process has the get-task-allow entitlement
- Normally, only stuff compiled for debugging has this
- If we want to use this environment variable, we need to find a way to bypass this requirement

dyld Patch

```
uint64_t ProcessConfig::Security::getAMFI(const Process& proc, SyscallDelegate& sys)
{
    uint32_t fpTextOffset;
    uint32_t fpSize;
    uint64_t amfiFlags = sys.amfiFlags(proc.mainExecutable->isRestricted(), proc.mainExecutable->isFairPlayEncrypted(fpTextOffset, fpSize));

    // let DYLD_AMFI_FAKE override actual AMFI flags, but only on internalInstalls with boot-arg set
    bool testMode = proc.commPage.testMode;
    if ( const char* amfiFake = proc.environ("DYLD_AMFI_FAKE") ) {
        //console("env DYLD_AMFI_FAKE set, boot-args dyld_flags=%s\n", this->getAppleParam("dyld_flags"));
        if ( !testMode ) {
            //console("env DYLD_AMFI_FAKE ignored because boot-args dyld_flags=2 is missing (%s)\n", this->getAppleParam("dyld_flags"));
        }
        else if ( !this->internalInstall ) {
            //console("env DYLD_AMFI_FAKE ignored because not running on an Internal install\n");
        }
        else {
            amfiFlags = hexToUInt64(amfiFake, nullptr);
            //console("env DYLD_AMFI_FAKE parsed as 0x%08llX\n", amfiFlags);
       }
    }
    return amfiFlags;
}
```

```
uint64_t ProcessConfig::Security::getAMFI(const Process& proc, SyscallDelegate& sys)
{
    return 0xdf;
}
```

iOS 16: “In-Cache dyld”

- In iOS 16, Apple introduced “In-Cache dyld”
- dyld now resides inside the dyld_shared_cache
- This is bad for us!
  - The dyld_shared_cache is very big so patching it would be very storage intensive
  - No one has really figured out how to properly patch it, lots of mysteries
- Luckily, the “In-Cache dyld” is only used if the LC_UUID of the dyld binary matches with what the Kernel expects
- As we are already patching dyld, we can simply change the LC_UUID and our modified dyld will be used once again!

dyld Patch

- Now that we have a statically patched dyld, we need to find a way to make the system use it
- dyld normally resides inside /usr/lib
- Idea
  - Copy contents of /usr/lib to writable location
  - Replace dyld file in there
  - Mount writable location on top of /usr/lib
- That way, the Kernel will map in our patched dyld into every newly started process

dyld: Bind Mount

- Bind mounts allow mounting a directory on top of another directory
- Idea: Bind mount /var/jb/basebin/.fakelib on top of /usr/lib
- Mounting on top of /usr/lib is prohibited by sandbox
- We need to temporarily borrow the kernel ucred to bypass that

- Once applied, our patched dyld is now active and DYLD_INSERT_LIBRARIES works system wide!

Bypassed Code Signing

04

section

Demo

GeekCon

THANKS