Putting it all together:

Building an iOS Jailbreak From Scratch


@umanghere

whoami

- 20
- Offensive security researcher.
- Primarily work on kernel and browser exploitation, occasionally release some of my research.
- Part of the Electra jailbreak team.
- Play CTFs with OpenToAll.

What's this talk about?

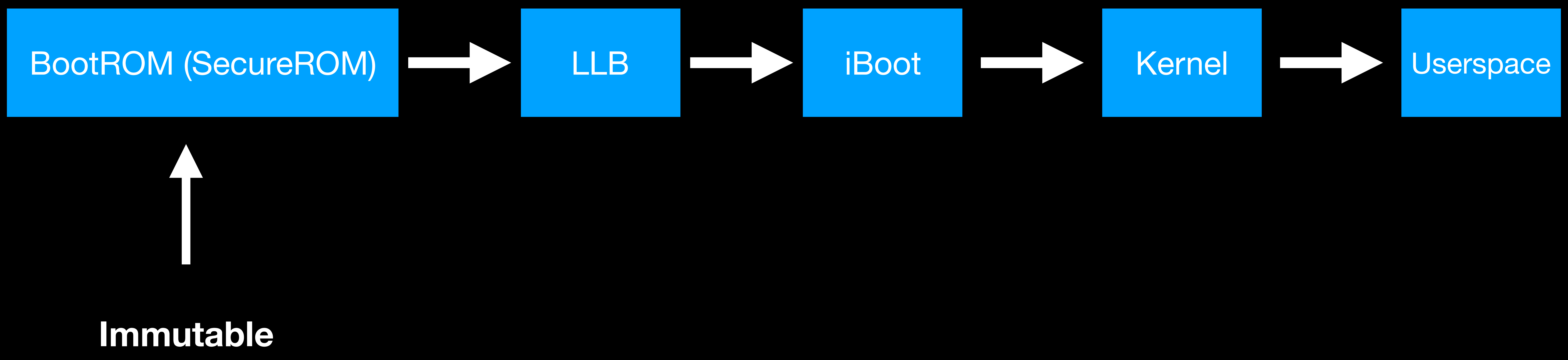
- A journey to run unsigned code on Apple's iOS devices, with the maximum privileges possible.
- A look at the mitigations that stand between us and our goal.
- Thoughts on breaking these mitigations, and understanding how they can be improved.

Where do we begin?

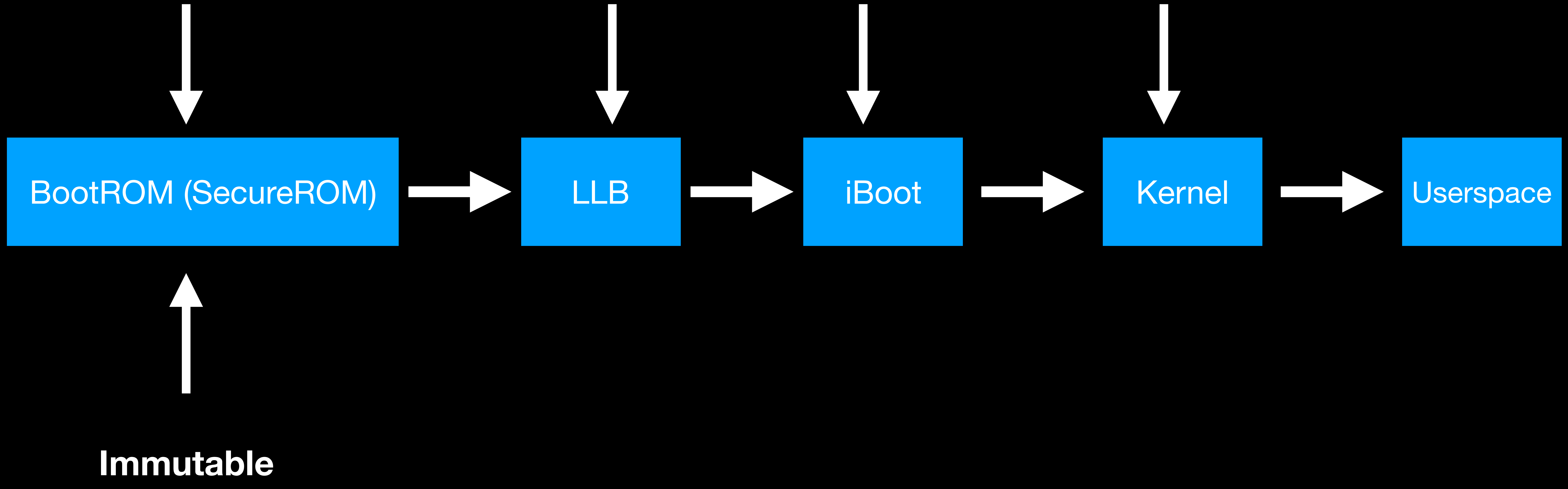
- We could target the iOS bootchain, and compromise the boot process of the device.
- Or, we could target the iOS kernel to escalate our privileges after the device has booted.

The iOS Bootchain

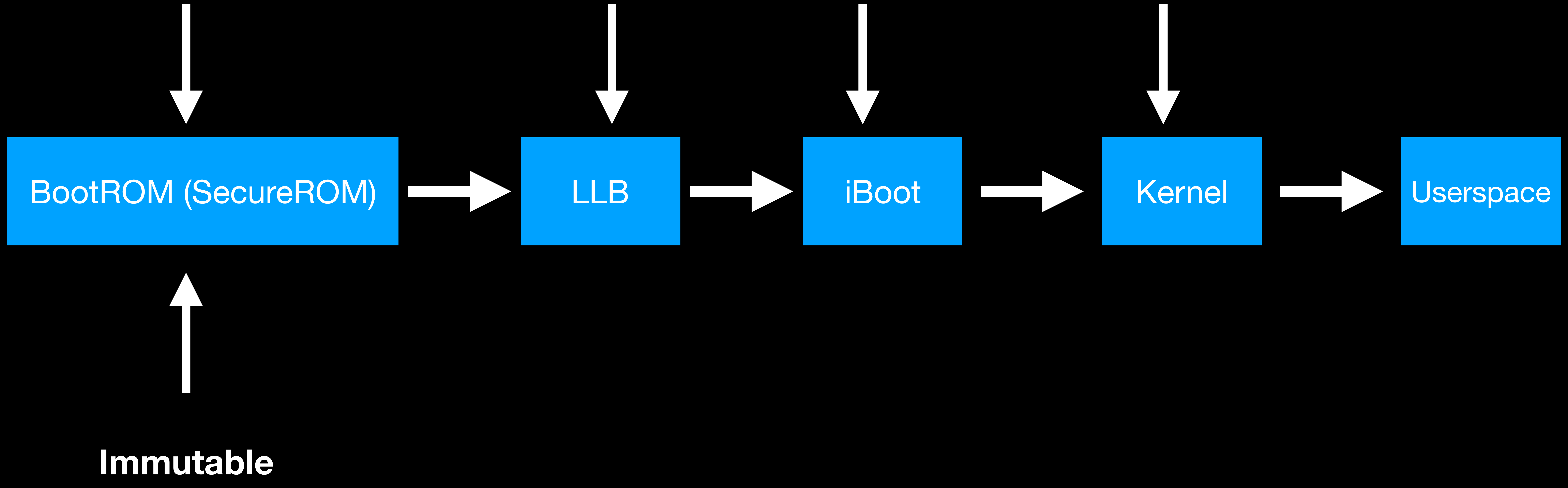
BootROM (SecureROM) → LLB → iBoot → Kernel → Userspace


BootROM (SecureROM)
LLB
iBoot
Kernel
Userspace
Immutable

Each stage verifies the next before switching to it



Attacking any stage allows you to control the next



Why target the bootchain?

- Processors start by executing code at the highest privilege level possible.

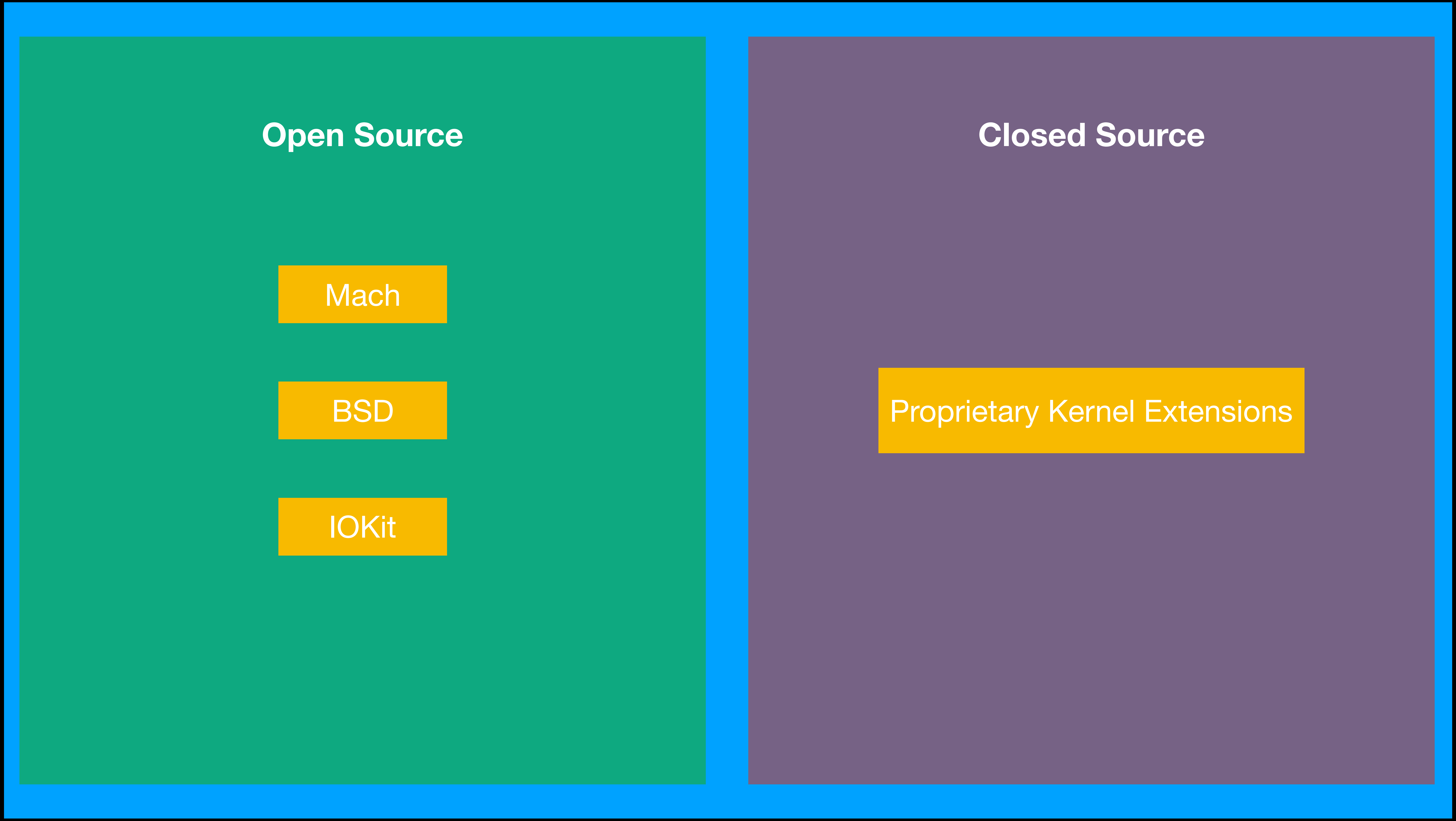
- Most exploit mitigations do not apply, or have not yet been initialised.

- If a vulnerability lies within the SecureROM, it cannot possibly be patched.

Why not
target the bootchain?

- Just a fraction of the complexity of the kernel and the userspace.
- Harder to find vulnerabilities due to reduced attack surface.
- Insufficient effort-to-reward ratio.

The iOS Kernel


Open Source
Mach
BSD
IOKit
Closed Source
Proprietary Kernel Extensions

The iOS Kernel

- Hybrid kernel, incorporates the Mach microkernel and BSD APIs.
- Some parts are released as open-source software.
- Device drivers are written with the IOKit framework, most of them are closed source.

Why target the iOS Kernel?

- Significantly more complex than the bootchain, vast attack surface.
- Closed-source kernel extensions are not audited enough.
- Successful compromise *should* allow code execution in EL1.
- Implicitly allows control over userspace processes.

Why not target the iOS Kernel?

- Open-source portions are heavily audited.
- Some of the attack surface is inaccessible from inside the sandbox.
- Interesting hardware, such as the crypto engine, are inaccessible.
- Significant vulnerability churn.

Attacking the iOS Kernel

What’s in a kernel exploit?

- We generalise a kernel exploit to have two primitives.
  - readKernelMemory(vm_addr_t, size_t)
  - writeKernelMemory(vm_addr_t, void *, size_t)

What’s in a kernel exploit?

- We generalise a kernel exploit to have two primitives.
  - readKernelMemory(vm_addr_t) ← **Read from the kernel’s address space**
  - writeKernelMemory(vm_addr_t, void *, size_t) ← **Write to the kernel’s address space**

What's in a kernel exploit?

- Most kernel exploits prefer to craft a send right to a fake Mach port corresponding to the kernel task.

- Reading and writing memory is just a matter of calling `mach_vm_{read|write}` on the send right.

- This isn't as straightforward as it sounds — being able to read kernel memory is often a prerequisite to craft this port.

oob_timestamp

- Kernel exploit targeting iOS 13.3 and below.
- Released by Brandon Azad for Google Project Zero.
- Out of bounds write of partially-controlled data in `AGXCommandQueue::processSegmentKernelCommand()`

oob_timestamp

- Kernel exploit targeting iOS 13.3 and below.
- Released by Brandon Azad for Google Project Zero.
- Out of bounds write of partially-controlled data in `AGXCommandQueue::processSegmentKernelCommand()`

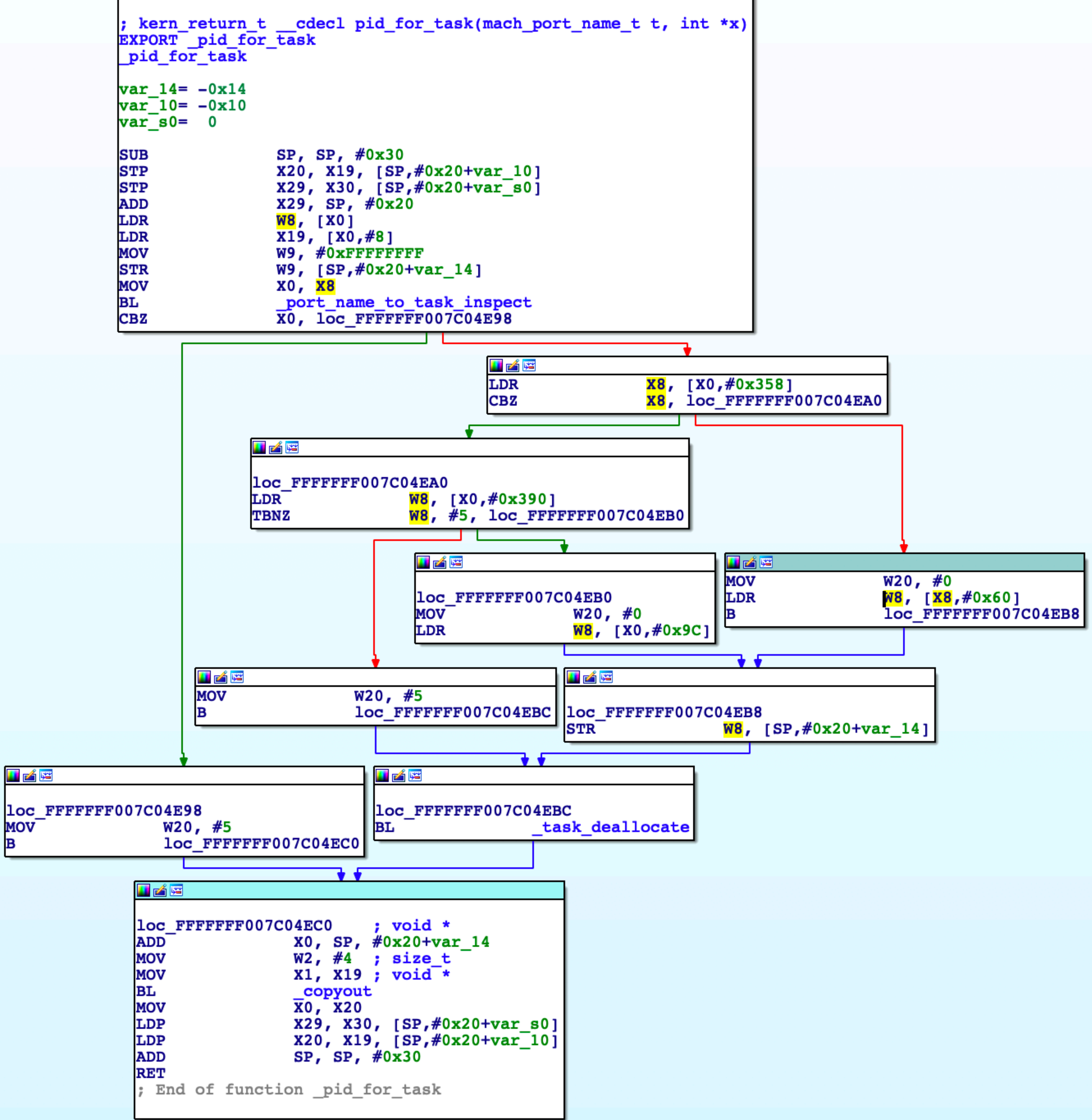
Closed Source Kernel Extension

oob_timestamp

- Sends a Mach message containing out-of-line ports.
- Uses a somewhat controlled out of bounds write to free some of these ports.
- Reallocates those ports by spraying data, leading to a Mach port with controlled contents waiting to be recieved.
- Receives the crafted Mach port.

early_readKernelMemory

- Given just control over a Mach port, how do we read kernel memory?
- Enter, `pid_for_task`.
    - In normal circumstances, returns the 32-bit process identifier of a task (to whose port you have a send right).


; kern_return_t __cdecl pid_for_task(mach_port_name_t t, int *x)
EXPORT _pid_for_task
_pid_for_task
var_14= -0x14
var_10= -0x10
var_s0= 0
SUB SP, SP, #0x30
STP X20, X19, [SP,#0x20+var_10]
STP X29, X30, [SP,#0x20+var_s0]
ADD X29, SP, #0x20
LDR W8, [X0]
LDR X19, [X0,#8]
MOV W9, #0xFFFFFFFF
STR W9, [SP,#0x20+var_14]
MOV X0, X8
BL _port_name_to_task_inspect
CBZ X0, loc_FFFFFF007C04E98
LDR X8, [X0,#0x358]
CBZ X8, loc_FFFFFF007C04EA0
loc_FFFFFF007C04EA0
LDR W8, [X0,#0x390]
TBNZ W8, #5, loc_FFFFFF007C04EB0
loc_FFFFFF007C04EB0
MOV W20, #0
LDR W8, [X0,#0x9C]
MOV W20, #0
LDR W8, [X8,#0x60]
B loc_FFFFFF007C04EB8
MOV W20, #5
B loc_FFFFFF007C04EBC
loc_FFFFFF007C04EB8
STR W8, [SP,#0x20+var_14]
loc_FFFFFF007C04E98
MOV W20, #5
B loc_FFFFFF007C04EC0
loc_FFFFFF007C04EBC
BL _task_deallocate
loc_FFFFFF007C04EC0 ; void *
ADD X0, SP, #0x20+var_14
MOV W2, #4 ; size_t
MOV X1, X19 ; void *
BL _copyout
MOV X0, X20
LDP X29, X30, [SP,#0x20+var_s0]
LDP X20, X19, [SP,#0x20+var_10]
ADD SP, SP, #0x30
RET
; End of function _pid_for_task

```
MOV     W9, #0xFFFFFFFF
STR     W9, [SP,#0x20+var_14]
MOV     X0, X8
BL      _port_name_to_task_inspect
CBZ     X0, loc_FFFFFF007C04E98
```

```
LDR     X8, [X0,#0x358]
CBZ     X8, loc_FFFFFF007C04EA0
```

```
MOV     W20, #0
LDR     W8, [X8,#0x60]
B       loc_FFFFFF007C04EB8
```

```
loc_FFFFFF007C04EC0     ; void *
ADD     X0, SP, #0x20+var_14
MOV     W2, #4  ; size_t
MOV     X1, X19 ; void *
BL      _copyout
MOV     X0, X20
LDP     X29, X30, [SP,#0x20+var_s0]
LDP     X20, X19, [SP,#0x20+var_10]
ADD     SP, SP, #0x30
RET
; End of function _pid_for_task
```

```
MOV     W9, #0xFFFFFFFF
STR     W9, [SP,#0x20+var_14]
MOV     X0, X8
BL      _port_name_to_task_inspect
CBZ     X0, loc_FFFFFF007C04E98
```

Get the task corresponding to the port

```
LDR     X8, [X0,#0x358]
CBZ     X8, loc_FFFFFF007C04EA0
```

```
MOV     W20, #0
LDR     W8, [X8,#0x60]
B       loc_FFFFFF007C04EB8
```

```
loc_FFFFFF007C04EC0     ; void *
ADD     X0, SP, #0x20+var_14
MOV     W2, #4   ; size_t
MOV     X1, X19  ; void *
BL      _copyout
MOV     X0, X20
LDP     X29, X30, [SP,#0x20+var_s0]
LDP     X20, X19, [SP,#0x20+var_10]
ADD     SP, SP, #0x30
RET
; End of function _pid_for_task
```

```
MOV     W9, #0xFFFFFFFF
STR     W9, [SP,#0x20+var_14]
MOV     X0, X8
BL      _port_name_to_task_inspect
CBZ     X0, loc_FFFFFFF007C04E98
```

Get the task corresponding to the port

```
LDR     X8, [X0,#0x358]
CBZ     X8, loc_FFFFFFF007C04EA0
```

Get the proc_t structure corresponding to the task

```
MOV     W20, #0
LDR     W8, [X8,#0x60]
B       loc_FFFFFFF007C04EB8
```

```
loc_FFFFFFF007C04EC0    ; void *
ADD     X0, SP, #0x20+var_14
MOV     W2, #4   ; size_t
MOV     X1, X19  ; void *
BL      _copyout
MOV     X0, X20
LDP     X29, X30, [SP,#0x20+var_s0]
LDP     X20, X19, [SP,#0x20+var_10]
ADD     SP, SP, #0x30
RET
; End of function _pid_for_task
```

```
MOV     W9, #0xFFFFFFFF
STR     W9, [SP,#0x20+var_14]
MOV     X0, X8
BL      _port_name_to_task_inspect
CBZ     X0, loc_FFFFFFF007C04E98
```

Get the task corresponding to the port

```
LDR     X8, [X0,#0x358]
CBZ     X8, loc_FFFFFFF007C04EA0
```

Get the proc_t structure corresponding to the task

```
MOV     W20, #0
LDR     W8, [X8,#0x60]
B       loc_FFFFFFF007C04EB8
```

Get the PID from the process structure

```
loc_FFFFFFF007C04EC0     ; void *
ADD     X0, SP, #0x20+var_14
MOV     W2, #4    ; size_t
MOV     X1, X19   ; void *
BL      _copyout
MOV     X0, X20
LDP     X29, X30, [SP,#0x20+var_s0]
LDP     X20, X19, [SP,#0x20+var_10]
ADD     SP, SP, #0x30
RET
; End of function _pid_for_task
```

```
MOV     W9, #0xFFFFFFFF
STR     W9, [SP,#0x20+var_14]
MOV     X0, X8
BL      _port_name_to_task_inspect
CBZ     X0, loc_FFFFFFF007C04E98
```

Get the task corresponding to the port

```
LDR     X8, [X0,#0x358]
CBZ     X8, loc_FFFFFFF007C04EA0
```

Get the proc_t structure corresponding to the task

```
MOV     W20, #0
LDR     W8, [X8,#0x60]
B       loc_FFFFFFF007C04EB8
```

Get the PID from the process structure

```
loc_FFFFFFF007C04EC0    ; void *
ADD     X0, SP, #0x20+var_14
MOV     W2, #4   ; size_t
MOV     X1, X19  ; void *
BL      _copyout
MOV     X0, X20
LDP     X29, X30, [SP,#0x20+var_s0]
LDP     X20, X19, [SP,#0x20+var_10]
ADD     SP, SP, #0x30
RET
; End of function _pid_for_task
```

Copy the PID out to userspace

early_readKernelMemory

- In effect, `pid_for_task` can be abused as a 4 byte read from a controlled address, given a controlled port.

- We combine two adjacent 4 byte reads into an 8 byte read.

early_readKernelMemory

- This would be enough to craft a task port that would behave like the actual kernel's task port, as we can now read *kernel_map* and *ipc_space_kernel*.

- Except we do not know where these values are located in kernel memory, due to Kernel Address Space Layout Randomisation.

KASLR

- Slides the kernel's virtual memory mapping by a random amount.
- The slide is generated by iBoot by hashing entropy, and changes every time the device reboots.
- In effect, we must know at least one pointer inside the kernel's image, direct or otherwise, to defeat KASLR.

KASLR

- Fortunately for us, *oob_timestamp* gives us a pointer to our IPC space.
- We can use this to leak the address of a Mach port corresponding to an IOSurface.
- This port has a pointer to the C++ object in *ip_kobject*.
- By reading the *vtable* pointer from this C++ object, we obtain a pointer within the kernel image.

{read,write}kernelMemory

- Using pid_for_task to read kernel memory is very inefficient.
- More importantly, we cannot write kernel memory with this technique.
- What do we do?

{read,write}kernelMemory

- Given that we now know the value of the *kernel_map, kernel_task* and *ipc_space_kernel,* we can now craft a fake task port that behaves exactly like the kernel task port.

- Having a send right to this task port grants us the ability to read and write memory in the kernel's address space by using `mach_vm_{read|write}`.

To Root and Beyond

- It is almost natural to escalate our privileges to the root user, so we locate our proc_t structure in memory and perform a few writes.
  - writeKernelMemory(proc->p_ucred.cr_posix.cr_uid, 0, 4);
- So we should now be able to do anything, right?

No!

To Root and Beyond

- Our application is running in the container sandbox profile, so we cannot perform several *interesting* operations.
- More importantly, we can't even successfully call some syscalls like *execve* or *fork*.
- What now?

To Root and Beyond

```
#if CONFIG_MACF
        /*
         * Determine if MAC policies applied to the process will allow
         * it to fork.  This is an advisory-only check.
         */
        err = mac_proc_check_fork(parent_proc);
        if (err != 0) {
                goto bad;
        }
#endif
```

bsd/kern/kern_fork.c

To Root and Beyond

```
int
mac_proc_check_fork(proc_t curp)
{
	kauth_cred_t cred;
	int error;

#if SECURITY_MAC_CHECK_ENFORCE
	/* 21167099 - only check if we allow write */
	if (!mac_proc_enforce) {
		return 0;
	}
#endif
	if (!mac_proc_check_enforce(curp)) {
		return 0;
	}

	cred = kauth_cred_proc_ref(curp);
	MAC_CHECK(proc_check_fork, cred, curp);
	kauth_cred_unref(&cred);

	return error;
}
```

security/mac_process.c

To Root and Beyond

```c
/*
 * In-kernel credential structure.
 *
 * Note that this structure should not be used outside the kernel, nor should
 * it or copies of it be exported outside.
 */
struct ucred {
        LIST_ENTRY(ucred)       cr_link; /* never modify this without KAUTH_CRED_HASH_LOCK */
#if defined(__STDC_VERSION__) && __STDC_VERSION__ >= 201112L && !defined(__STDC_NO_ATOMICS__)
        _Atomic u_long          cr_ref;  /* reference count */
#elif defined(__cplusplus) && __cplusplus >= 201103L
        _Atomic u_long          cr_ref;  /* reference count */
#else
        volatile u_long         cr_ref;  /* reference count */
#endif

        struct posix_cred {
                /*
                 * The credential hash depends on everything from this point on
                 * (see kauth_cred_get_hashkey)
                 */
                uid_t   cr_uid;         /* effective user id */
                uid_t   cr_ruid;        /* real user id */
                uid_t   cr_svuid;       /* saved user id */
                short   cr_ngroups;     /* number of groups in advisory list */
                gid_t   cr_groups[NGROUPS];/* advisory group list */
                gid_t   cr_rgid;        /* real group id */
                gid_t   cr_svgid;       /* saved group id */
                uid_t   cr_gmuid;       /* UID for group membership purposes */
                int     cr_flags;       /* flags on credential */
        } cr_posix;
        struct label    *cr_label;      /* MAC label */
        /*
         * NOTE: If anything else (besides the flags)
         * added after the label, you must change
         * kauth_cred_find().
         */
        struct au_session cr_audit;             /* user auditing data */
};
```

bsd/sys/ucred.h

To Root and Beyond

```c
/*
 * In-kernel credential structure.
 *
 * Note that this structure should not be used outside the kernel, nor should
 * it or copies of it be exported outside.
 */
struct ucred {
        LIST_ENTRY(ucred)       cr_link; /* never modify this without KAUTH_CRED_HASH_LOCK */
#if defined(__STDC_VERSION__) && __STDC_VERSION__ >= 201112L && !defined(__STDC_NO_ATOMICS__)
        _Atomic u_long          cr_ref;  /* reference count */
#elif defined(__cplusplus) && __cplusplus >= 201103L
        _Atomic u_long          cr_ref;  /* reference count */
#else
        volatile u_long         cr_ref;  /* reference count */
#endif

        struct posix_cred {
                /*
                 * The credential hash depends on everything from this point on
                 * (see kauth_cred_get_hashkey)
                 */
                uid_t   cr_uid;         /* effective user id */
                uid_t   cr_ruid;        /* real user id */
                uid_t   cr_svuid;       /* saved user id */
                short   cr_ngroups;     /* number of groups in advisory list */
                gid_t   cr_groups[NGROUPS];/* advisory group list */
                gid_t   cr_rgid;        /* real group id */
                gid_t   cr_svgid;       /* saved group id */
                uid_t   cr_gmuid;       /* UID for group membership purposes */
                int     cr_flags;       /* flags on credential */
        } cr_posix;
        struct label    *cr_label;      /* MAC label */
        /*
         * [illegible]
         * added after the label, you must change
         * kauth_cred_find().
         */
        struct au_session cr_audit;             /* user auditing data */
};
```

bsd/sys/ucred.h

To Root and Beyond

- The Mandatory Access Control framework is the foundation of the iOS sandbox.
- MAC uses `p_ucred.cr_label` to determine which policies to enforce.
- We could change it to a null pointer.
- Or we could change the process's `p_ucred` to the kernel's `p_ucred`, which bypasses almost all sandbox checks.

To Root and Beyond

```
kptr_t kern_ucred = readKernelMemory64(kernel_proc + OFF(proc,
p_ucred));

writeKernelMemory32(kern_ucred + OFF(ucred, cr_ref), 0xcdef);

writeKernelMemory64(my_proc + OFF(proc, p_ucred), kern_ucred);
```

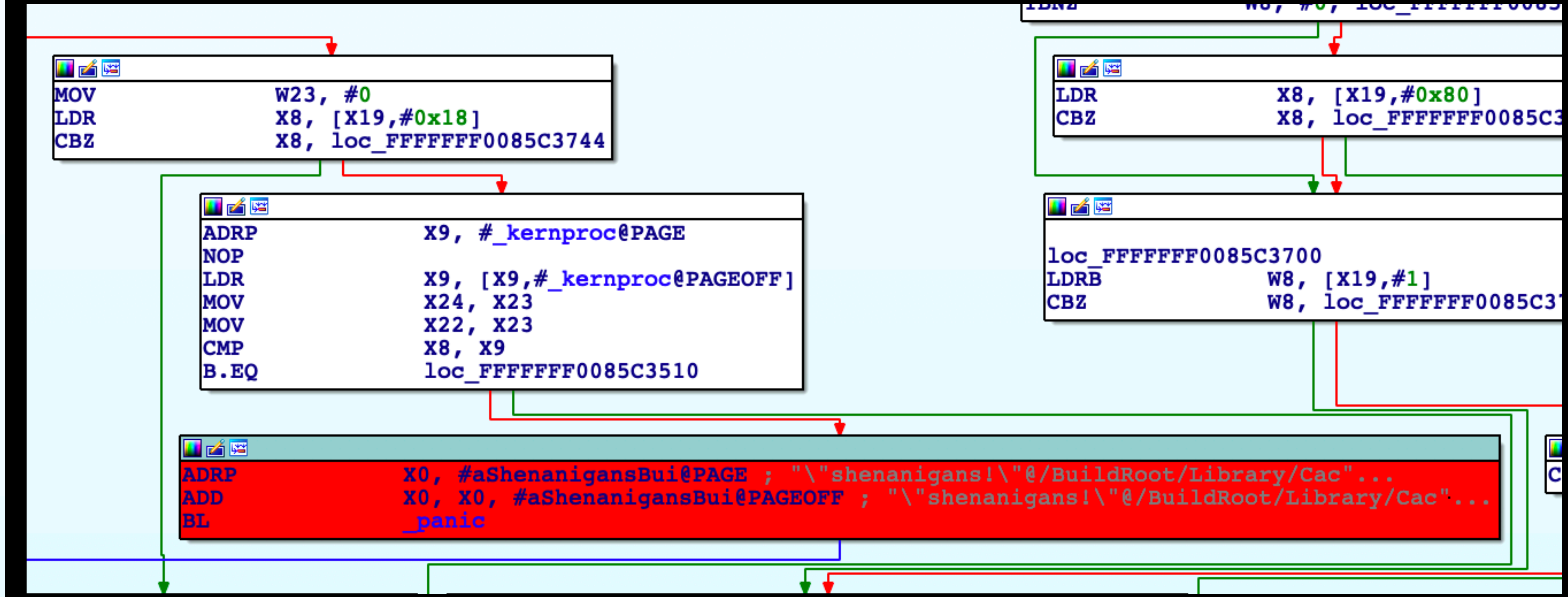
To Root and Beyond

- As soon as we try to do something useful, the kernel panics.

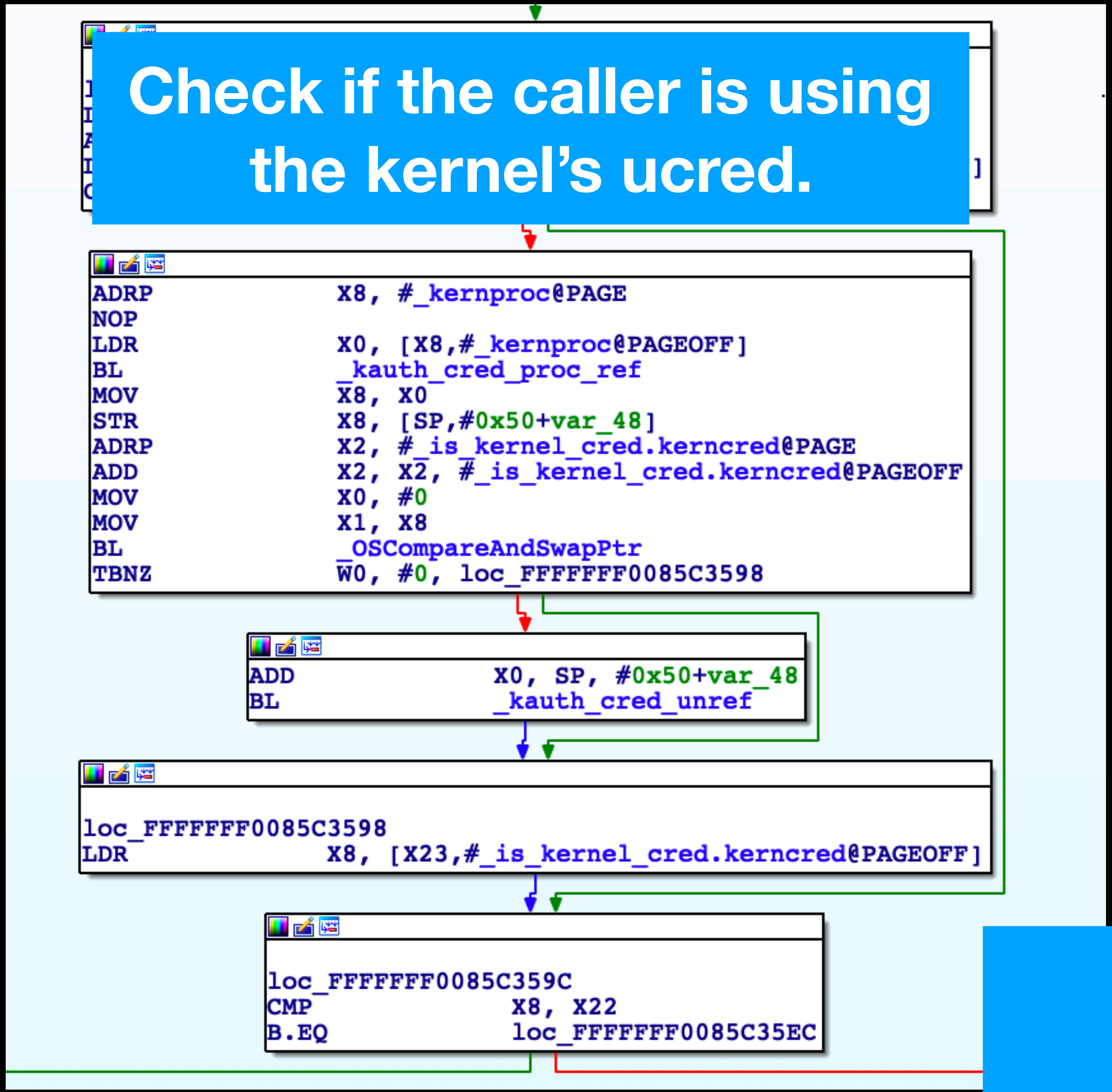
```
panic(cpu 0 caller 0xfffffff00a18b574): “shenanigans!"
```

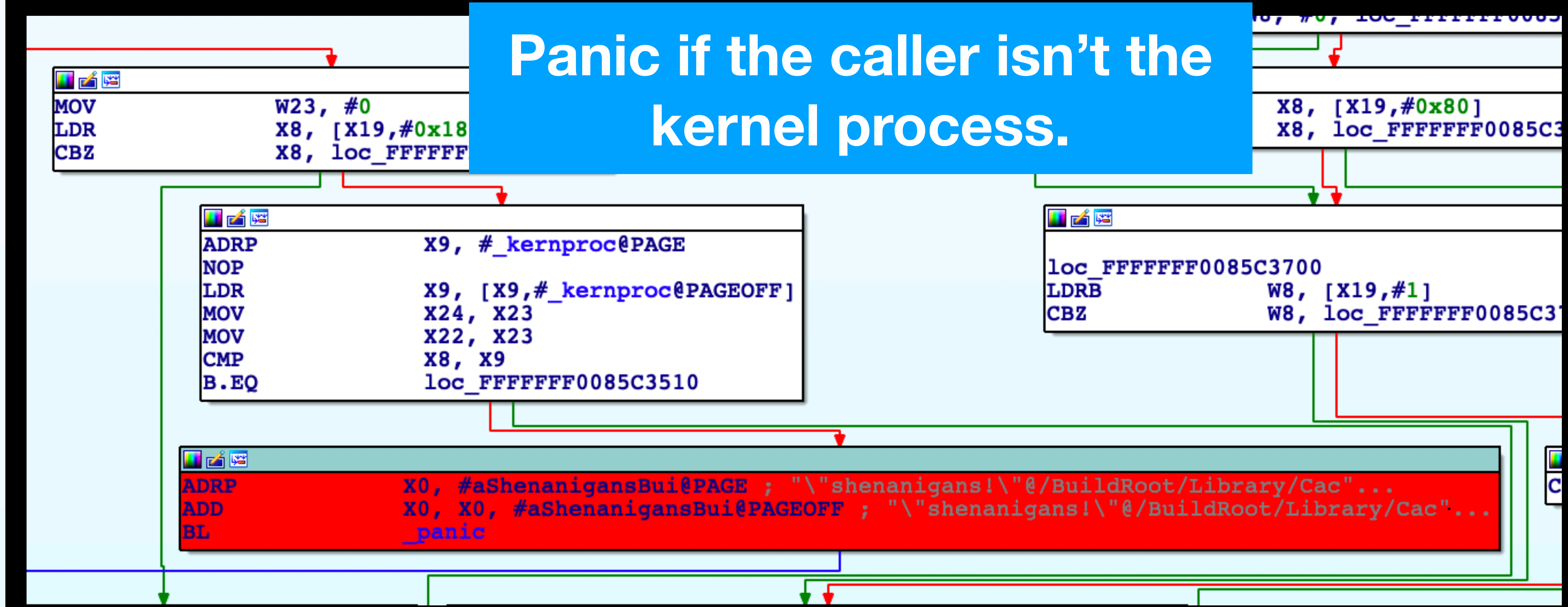
Shenanigans!

To Root and Beyond



To Root and Beyond





sb_evaluate

To Root and Beyond

- Good idea, terrible implementation.
- Caches the value of the kernel’s ucred.
- We can overwrite the cached value with garbage and always skip the check.

Remounting /

- The APFS filesystem at / is mounted read-only at boot, whereas /private/var is mounted as read-write.

- We'd like to write in /, so let's remount it.

Remounting /

- But we can’t! The kernel explicitly disallows remounting the filesystem at /.
- This is done by checking the MNT_ROOTFS flag on the root vnode.
- Let’s patch away the check in _hook_mount_check_remount.

Nope!

A Lesson in KTRR

- On the A10 SoCs and later, the MMU prevents writes to protected memory pages.
- These include the kernel code (__TEXT), kext code and all __const regions.
- Patching the kernel’s code is nontrivial, we have to make do by data-only post-exploitation.

Remounting /

- To remount the filesystem, let’s temporarily remove the MNT_ROOTFS flag from the root vnode.
- That worked, so let’s try to write something in /.

Panic!

Remounting /

- Starting iOS 11.2.6, an APFS snapshot is mounted at /.
- Snapshots are not designed to be written to, the filesystem driver panics.
- @SparkZheng and @bxl1989 released a temporary bypass, followed by which, I released a persistent one.

Remounting /

- Their solution: mount the root block device somewhere else and swap the filesystem-specific data.
- Somewhat unstable, the device eventually panics.

Remounting /

- My solution stops the snapshot from being mounted at / in the first place.
- This is because the kernel checks for the presence of a snapshot corresponding to the boot manifest hash and mounts it early in the boot process.
- Strangely enough, if a matching snapshot is not found, the kernel mounts the actual volume instead.
- We can rename the snapshot to anything we'd like by using the fs_snapshot_rename syscall, and force the actual filesystem to be mounted.

Remounting /

Additional recognition

APFS

We would like to acknowledge Umang Raghuvanshi for their assistance.

Entry added December 13, 2018

Remounting /

- After iOS 12, fs_snapshot_rename fails when called on the root volume.
- A flag inside the snapshot’s vnode is checked, and the syscall fails if it is set.
- Sneaky, sneaky!
- Didn’t last very long, I demoed a bypass just a few days after the first kernel exploit for iOS 12 became available.

Remounting /

- This change was a step in the right direction — attackers now needed the ability to call functions in EL1 if they wanted to locate the snapshot vnode.
- On A12 SoCs and above, this would require the ability to forge signed pointers in order to defeat ARMv8.3 Pointer Authentication.
- Here's what happens if you don't use a signed pointer when calling kernel functions:

Panic!

Remounting /

“It's hard to call something a PAC defeat without knowing what PAC is supposed to defend against, and it's hard to say that something like PAC "raises the bar" without knowing whether anyone really has to cross that bar anyway.”

— Ian Beer

Remounting /

- “Attackers now needed the ability to call functions in EL1 if they wanted to locate the snapshot vnode.”
  - Do they?
  - Is there absolutely no place where a pointer to the snapshot vnode might be saved?

Remounting /

- There is one — iterating through the namecache yields a pointer to “/.snaps/snapshot-name”.
- We can then set the flag and remount away.
- Since we only need to read and write kernel memory, bypassing pointer authentication is not required!

CoreTrust

- Introduced in iOS 12, with several rumours suggesting that it'd be a userspace version of KTRR.
- One of the most underwhelming security mitigations I have seen.
- Requires that every binary have a valid CMS blob in the code signature.

CoreTrust

- There are no checks if the certificate used to sign the CMS blob are still valid — the only requirement is that the certificate must have a chain of trust leading to Apple.
- While it can be bypassed entirely by a few well placed writes in the vnode cache, it’s far simpler to get an expired enterprise developer certificate and use it to sign binaries instead.

Takeaways

iOS and macOS pack a ton of pre-exploit and post-exploit mitigations.

Even after attacking the kernel, the amount of hoops an attacker has to jump through is fascinating.

The security architecture of iOS results in a semblance of normality even when the kernel has been successfully attacked.

Apple’s approach to security relies significantly on post-exploit mitigations, which are only set to increase in number.

The future for attackers is challenging — and, dare I say, exciting!

Thanks

We're standing on the shoulders of giants.

Thanks

- The Electra team (@CStar_OW, @jaimiebishop123 et. al.)
- Nullcon organisers.
- You

Questions?

@umanghere