Come for the Tradecraft, Stay for the Code

About Me

- Author of *Hacking and Securing iOS Applications*, *iPhone Open Application Development, iPhone Forensics, iPhone SDK Application Development*
- Original inventor of iOS forensics techniques, validated by NIJ / NIST
- Consult for federal government and military
- Teach iOS forensics worldwide
- Try to do a science every day

What is a Root Kit?

- An injection payload to persist access to a target system
- Can include additional components:
  - *Stealth components to hide its own existence*
  - *Monitoring components (keyboard, microphone, webcam, packet sniffers, etc.)*
  - *C2 component for remote access, special instructions, updates*
- Not the exploit itself
- Often paired with an exploit or other injection vector (0day, trojan, etc.)
- Persistence of the payload typically requires root access

Types of Root Kits: Userland Kit

- *Consist of userland programs (daemons, agents, startup programs)*
- *Typically trojanized binaries replacing otherwise trusted tools (ps, netstat, etc.)*
- *Can sometimes be a component of trojanized software (Transmission, Xcode, etc.)*

- Detectable by AV software, after signatures are added
- Detectable by behavioral analysis (Little Snitch, Little Flocker, etc.)
- Easy to remove once identified
- SIP in macOS protects system components at a kernel level

Types of Root Kits: Kernel Root Kit

- More difficult to detect; limited kit detection tools for macOS
- Can hide from userland processes by altering system state
- Userland programs rely on information from the kernel
  - *Process information, netstat information, etc. all come from kernel*
  - *Including userland antivirus software*
  - *Including userland root kit detection software*
  - *Detecting root kits in the kernel is a race*
- Kernel mode code runs at the highest privilege level
  - *Higher privileges than the user has in macOS*
- Kernel is open source! http://opensource.apple.com

macOS

- Recent protections in macOS:
  - *Translocation*
  - *Sandboxing*
  - *System Integrity Protection (SIP)*
  - *Kernel Signing Certificates* (now, root != kernel)*
  - ** otherwise, must turn off SIP to load an unsigned kext*

- Many exploits today are still kernel exploits from userland

XNU

- Based on BSD
- Fusion of Mach and BSD
- Open source - http://opensource.apple.com
- Kernel development fully supported in Xcode
- Private KPI, but with full access to kernel address space
- Supports kernel extensions (KEXT), dynamically loaded modules
- Similar to Linux Loadable Kernel Modules
- Supports device drivers, file systems, generic modules
- Code must be perfect, or entire system will become unstable

Common Classes of Vulnerabilities

- UAF (Use After Free)
  - *Pegasus*
  - *Most common in web browsing vulnerabilities*
- Copy from user bugs
- Uninitialized data
- Timing attacks
- Diffing the XNU sources are a great way to learn about kernel vulnerabilities

Use After Free: Pegasus

- Surveillance Toolkit for iOS / macOS
- Funded in part by nation states
- Was used to target a human rights activist
- Three different exploits, one using UAF (Use After Free) of OSUnserializeXML via OSUnserializeBinary
- Caused memory corruption, leading to remote code execution
- Patched in iOS 9.3.5

Pegasus: About OSUnserializeBinary

- Used to process OSObject data
- Exposed to user input via OSUnserializeXML
- Can be attacked using basic I/O Kit APIs
- Can be triggered from within the sandbox
- Source code: **libkern/c++/OSSerializeBinary.cpp**

Pegasus: The Vulnerability

- When an object is unserialized, it's added to a table of objects:

```
if (!isRef) {
    setAtIndex(objs, objsIdx, o);
    if (!ok)
        break;
    objsIdx++;
}
```

- But *setAtIndex* doesn't bump reference count

Pegasus: Vulnerable Code Path

```
if (dict) {
    if (!sym) sym = (OSSymbol *) o;
    else {
        str = sym;
        sym = OSDynamicCast(OSSymbol, sym);
        if (!sym && (str = OSDynamicCast(OSString, str))) {
            sym = (OSSymbol *) OSSymbol::withString(str);
            ok = (sym != 0);
            if (!ok) break;
        }
        if (o != dict) ok = dict->setObject(sym, o);
        if (sym && (sym != str))
            sym->release(); < releases object that was never retained!
        sym = 0;
    }
}
```

Pegasus

- OSSymbol and OSString dictionary keys were added in iOS 9
- Wasn't an old bug lying around for years
- Says a lot about the aggressiveness of APTs
- Demonstrates the importance of good coding practices
- They're reading the source code, so should you

Timing Attacks – Default memcmp

```
int memcmp(s1, s2, n)
CONST VOID *s1; /* First string. */
CONST VOID *s2; /* Second string. */
size_t n; /* Length to compare. */
{
    unsigned char u1, u2;
    for ( ; n-- ; s1++, s2++) {
        u1 = * (unsigned char *) s1;
        u2 = * (unsigned char *) s2;
        if ( u1 != u2) { return (u1-u2); } < not constant time!
     }
    return 0;
}
```

Constant Time Functions

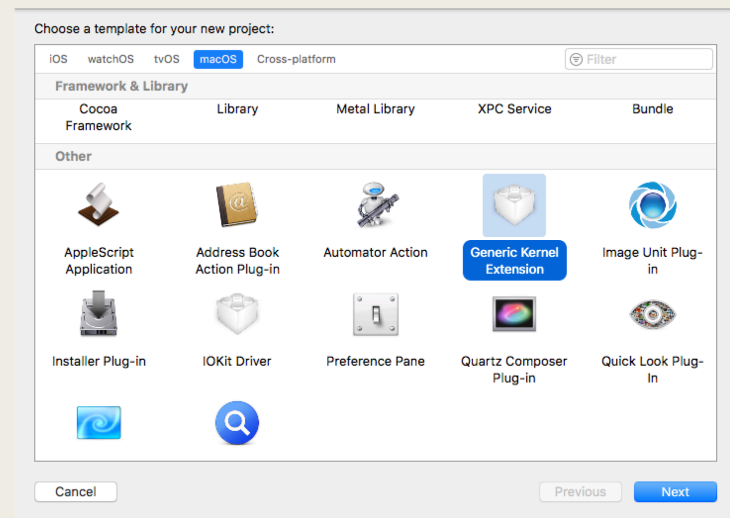
```
static int memcmp_s(const void *s1, const void *s2, size_t n)
{
    int ret = 0;
    if (n != 0) {
        const unsigned char *p1 = (unsigned char *)s1;
        const unsigned char *p2 = (unsigned char *)s2;
        do {
            if (*p1++ != *p2++ && !ret)
                ret = (*--p1 - *--p2);
        } while (--n != 0); < constant time!
    }
    return ret;
}
```

Copy From User Bugs

- I/O Kit doesn’t copy_from_user like Linux, it creates memory descriptors
- Memory is shared between user space and kernel space
- TOCTOU: Time of Check vs. Time of Use
- Dev failed to copy data in from user space
- Checked data while still connected to user space
- Userland later corrupts structures
- Profit!

Looking at Kernel Code

- XNU source: osfmk/mach/mach_types.h
- Xcode SDK /Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/System/Library/Frameworks/Kernel.framework/Headers/mach/mach_types.h
- Kernel development is literally this easy:



Skeleton KEXT

```
#include <mach/mach_types.h>

kern_return_t HelloWorld_start(kmod_info_t * ki, void *d);
kern_return_t HelloWorld_stop(kmod_info_t *ki, void *d);

kern_return_t HelloWorld_start(kmod_info_t * ki, void *d)
{
    return KERN_SUCCESS;
}

kern_return_t HelloWorld_stop(kmod_info_t *ki, void *d)
{
    return KERN_SUCCESS;
}
```

An Opaque Kernel

- Out of 20,000 symbols, only 3,200 are exposed through the KPI
- All others live in com.apple.kpi.private, can only be loaded by com.apple KEXTs
- All the fun stuff is hidden
- Kernel devs must do ugly things

1. Unslide ASLR
2. Walk through kernel base image
3. Perform symbol table lookups

Resolving Opaque Kernel Symbols

- Find ASLR Slide
  - *You could walk back memory to find 0xfeedfacf*
  - *...but Apple provides a function to unslide ASLR!*
- Read mach-o format from kernel base address
- Find __LINKEDIT segment and LC_SYMTAB command
- Perform symbol lookup in memory
- Profit!
- https://www.zdziarski.com/KernelResolver.c

./osfmk/vm/vm_kern.c

```
void
vm_kernel_unslide_or_perm_external(
        vm_offset_t addr,
        vm_offset_t *up_addr)
{
        if (VM_KERNEL_IS_SLID(addr)) {
                *up_addr = addr - vm_kernel_slide;
                return;
        }
        vm_kernel_addrperm_external(addr, up_addr);
        return;
}
```

./osfmk/mach/vm_param.h

```
#define VM_KERNEL_IS_SLID(_o)                                      \
  (((vm_offset_t)(_o) >= vm_kernel_slid_base) &&                 \
  ((vm_offset_t)(_o) <  vm_kernel_slid_top))
```

Deducing ASLR Slide

```
#define KERNEL_BASE 0xffffff8000200000

kern_return_t KernelResolver_start(kmod_info_t * ki, void *d)
{
    int64_t slide = 0;
    vm_offset_t slide_address = 0;

    vm_kernel_unslide_or_perm_external(KERNEL_BASE,
&slide_address);
    slide = KERNEL_BASE - slide_address;
    int64_t base_address = slide + KERNEL_BASE;

    DLOG("%s: aslr slide: %lld\n", __func__, slide);
    DLOG("%s: base address: %lld\n", __func__, base_address);
}
```

Find Segment and Load Command

- Load command starts immediately after header
- __LINKEDIT segment and LC_SYMTAB was hidden prior to Snow Leopard
  - *Now loaded into memory FTW!*
- Simple structures taken from XNU source
- Parse segments and load commands to find the ones we want

Walk the Symbol Table

```
int64_t strtab_addr = (int64_t)(linkedit->vmaddr - linkedit->fileoff) + symtab->stroff;

int64_t symtab_addr = (int64_t)(linkedit->vmaddr - linkedit->fileoff) + symtab->symoff;

strtab = (void *)strtab_addr;

for (i = 0, nl = (struct nlist_64 *)symtab_addr; i < symtab->nsyms; i++, nl = (struct nlist_64 *)((int64_t)nl + sizeof(struct nlist_64)))

{

    char *str = (char *)strtab + nl->n_un.n_strx;

    if (strcmp(str, name) == 0) {

        addr = (void *)nl->n_value;

    }

}
```

Use the Opaque Symbol

```
task_t (*_proc_task)(proc_t) = (task_t(*)(proc_t)) find_symbol(
    (struct mach_header_64 *)base_address,
    "_proc_task");

task_t task = _proc_task(proc);
```

I/O Kit

- Kernel and userland system for providing service to hardware
- Kernel mode architecture utilizing C++
- Hardware can be easily chained, functionality through inheritance
- Standardized systems for probing and matching equipment and drivers
- Support for generic providers (no hardware required)
- Kernel drivers can be easily created from Xcode templates
- User client class and provider class

I/O Kit: Benefits

- Early loading (before root mounted)
- Direct access to hardware drivers
- Object-oriented abstraction layer, less coding
- Calling conventions with built-in sanity checks for argument and structure sizes

```
#define super IOService
OSDefineMetaClassAndStructors(HelloWorld, IOService);

bool HelloWorld::init(OSDictionary *dict);
{ }

IOService * HelloWorld::probe(IOService *provider, SInt32* score);
{ }

bool HelloWorld::start(IOService *provider);
{ }

void HelloWorld::stop(IOService *provider);
{ }

void HelloWorld::free(void)
{ }
```

Userland Comms

- Kernel-mode drivers often paired with LaunchDaemon and LaunchAgent
- Drawbacks:
  - *No way to ensure daemon runs at startup*
  - *No way to ensure your daemon runs before malware (race)*
  - *No code signing functions exposed in kernel (only sha1)*
  - *Apple does not provide a means to pair KEXT-Daemon-Agent*
  - *Left up to the developer to verify the authenticity of all components*
  - *No guarantee daemon or agent will ever start, no way to start from KEXT*

Userland Comms

- IOConnectCallMethod
  - *Easy userland calling interface into KEXT*
  - *Performs sanity checks of argument and structure sizes*
  - *Facilities to write output and supply a return code*
  - *Uses memory descriptors to pass data, BEWARE of TOCTOU*

KEXT:

```
const IOExternalMethodDispatch
LittleFlockerDriverClient::sMethods[kLittleFlockerRequestMethodCount] =
{
    { &LittleFlockerDriverClient::sClearConfiguration, 0, SKEY_LEN, 0, 0 },
    { &LittleFlockerDriverClient::sStartFilter, 0, 0, 0, 0 },
    { &LittleFlockerDriverClient::sStopFilter, 0, SKEY_LEN, 0, 0 },
    { &LittleFlockerDriverClient::sSetDaemonPID, 1, SKEY_LEN, 0, 0 },

...

};

IOReturn LittleFlockerDriverClient::sSetDaemonPID(OSObject *target, void *reference,
IOExternalMethodArguments *args)
{
    LittleFlockerDriverClient *me = (LittleFlockerDriverClient *)target;
    me->m_driver->setDaemonPID(args->scalarInput[0], (unsigned char *)args-
>structureInput);
    return KERN_SUCCESS;
}
```

Userland:

```
kern_return_t kr = IOConnectCallMethod(
    connectionHandle,
    kRequestPerformMyOperation,
    input_args,
    input_arg_len,
    input_struct,
    input_struct_len,
    output_arg,
    output_arg_len,
    output_struct,
    output_struct_len);
```

Example:

```
        kern_return_t kr = IOConnectCallMethod(g_driverConnection,
kLittleFlockerRequestAssignDaemonPID, args, 1, g_skey, SKEY_LEN,
NULL, NULL, NULL, NULL);
```

Hiding KEXTs

- Kernel module list (kmod_info_t) is now deprecated
- Must patch I/O Kit in memory
  - *Crisis root kit patched array used by OSKext * OSKext::lookupKextWithLoadTag(uint32_t aTag)*
  - *Still there in macOS Sierra:*

```
$ nm kernel |grep __ZN6OSKext21lookupKextWithLoadTagEj
ffffff8000850590 T __ZN6OSKext21lookupKextWithLoadTagEj
```

- Simply manipulate the array to hide KEXT

./libkern/c++/OSKext.cpp:

```
OSKext *
OSKext::lookupKextWithLoadTag(uint32_t aTag)
{
    OSKext * foundKext = NULL;                  // returned
    uint32_t count, i;
    IORecursiveLockLock(sKextLock);
    count = sLoadedKexts->getCount();
    for (i = 0; i < count; i++) {
        OSKext * thisKext = OSDynamicCast(OSKext, sLoadedKexts->getObject(i));
        if (thisKext->getLoadTag() == aTag) {
            foundKext = thisKext;
            foundKext->retain();
            goto finish;
        }
    }
finish:
```

Hiding Files

- Manipulating *getdirentries*, *getdirentriesattr*, and *getdirentries64*
- Manipulate *struct direntry* based on filename comparison (*d_name*)
- Alternatively, use full path by reading from *proc* structure (first argument of every syscall)

```
struct proc {
    (...)
    struct filedesc *p_fd; /* open files structure. */
    (...)
}
struct filedesct {
    struct vnode *fd_cdir; /* current directory */
    struct vnode *fd_rdir; /* root directory */
}
```

Hiding Processes

- Intercept calls to *allproc* to obtain process list
- Remove proc from returned table, but keep proc structure intact
- Detectable
  - *rubilyn was detectable in this way*
- Better root kits attempt to delete the proc structures themselves

Hooking sysent

- sysent hooking is easy to detect
- Apple now attempts to hide the table, doesn't export its symbol
  - *unix_syscall, unix_syscall64, and unix_syscall_return reference it*
- Now loaded into read-only memory (even easier to detect changes)
- There are more clever ways to hook in a root kit

Mandatory Access Policy Framework

- MACF
- Also adopted from BSD
- Incorporates hooks for many system level operations
- Used extensively by sandboxd and tcc in macOS
- The ultimate in authoritarian kernel-mode system control
  - *Can prevent access to any file*
  - *Can prevent a process from being killed*
  - *Can prevent a KEXT from being loaded / unloaded*
  - *Ideal for maintaining persistence*

Define Policy Operations

```
mac_policy_handle_t policyHandle;
struct mac_policy_ops policyOps;
policyHandle = { 0 };
policyOps = {
            .mpo_vnode_notify_create   = _macf_vnode_notify_create_internal,
            .mpo_mount_check_mount     = _macf_mount_check_mount_internal,
            .mpo_iokit_check_nvram_get = _macf_iokit_check_nvram_get_internal,
            .mpo_iokit_check_nvram_set = _macf_iokit_check_nvram_set_internal,
            .mpo_iokit_check_nvram_delete = _macf_iokit_check_nvram_delete_internal,
            .mpo_kext_check_load          = _macf_kext_check_load_internal,
};
```

Define Policy Config

```
struct mac_policy_conf policyConf;
policyConf = {
            .mpc_name             = "LF File Monitor",
            .mpc_fullname         = "Little Flocker Kernel-Mode Monitor",
            .mpc_labelnames       = NULL,
            .mpc_labelname_count = 0,
            .mpc_ops              = &policyOps,
            .mpc_loadtime_flags  = MPC_LOADTIME_FLAG_UNLOADOK,
            .mpc_field_off        = NULL,
            .mpc_runtime_flags   = 0,
            .mpc_list             = NULL,
            .mpc_data             = NULL
        };
```

Register New Policy

```
#include <security/mac.h>

#include <security/mac_policy.h>

#include <security/mac_framework.h>


int _mac_policy_register_internal(struct mac_policy_conf *mpc,
mac_policy_handle_t *handlep) {

    return mac_policy_register(mpc, handlep, (void *)0);

}

int _mac_policy_unregister_internal(mac_policy_handle_t handlep) {

    return mac_policy_unregister(handlep);

}
```

Debugging Panic Logs

- Panic logs easier than debugging, often all you need
- KEXT load address and backtrace addresses included in panic

```
Kernel Extensions in backtrace:
com.zdziarski.LittleFlocker(1.4.8)[20D393DC-B937-35DF-9C0D-
10E3868808BA]@0xffffff7fa07f2000->0xffffff7fa07fbfff
com.metakine.handsoff.driver(3.1.4)[A7D68D9D-C470-3BE4-8037-
2E28E8D31F15]@0xffffff7fa089b000->0xffffff7fa08abfff
```

Debugging Panic Logs

```
Backtrace (CPU 1), Frame : Return Address
0xffffff8091cdde90 : 0xffffff801fef211c
0xffffff8091cddf10 : 0xffffff8020006a05
0xffffff8091cde070 : 0xffffff801fea3fff
0xffffff80b7ba03c0 : 0xffffff7fa07f3a51
0xffffff80b7ba0400 : 0xffffff8020528f3e
0xffffff80b7ba0440 : 0xffffff7fa08a2cbe
0xffffff80b7ba0cb0 : 0xffffff80201233ea
0xffffff80b7ba0d00 : 0xffffff8020142ee1
0xffffff80b7ba0db0 : 0xffffff802012086c
0xffffff80b7ba1130 : 0xffffff7fa07f58cf < within our address range!
…
```

Debugging Panic Logs

- return address – load address = code offset

0xffffff7fa07f58cf - 0xffffff7fa07f2000 = 0x38cf
0x38ca (call _vnode_open) crashed

```
00000000000038a8    call    _vfs_context_current
00000000000038ad    xor     r14d, r14d
00000000000038b0    lea     r8, qword [rbp+var_1D0]
00000000000038b7    mov     esi, 0x101
00000000000038bc    xor     edx, edx
00000000000038be    xor     ecx, ecx
00000000000038c0    mov     rdi, qword [rbp+var_240]
00000000000038c7    mov     r9, rax
00000000000038ca    call    _vnode_open
00000000000038cf    mov     r12d, eax
00000000000038d2    mov     edi, 0x5
00000000000038d7    mov     rsi, rbx
00000000000038da    call    _OSTestAndClear
00000000000038df    mov     rdi, qword [r13+0x2418]
00000000000038e6    call    _IORWLockWrite
```

Questions?