You can’t see me: A Mac OS X Rootkit uses the tricks you haven't known yet.

Ming-chieh Pan, Sung-ting Tsai. Team T5.

Contact: (nanika|tt)@teamt5.org

Abstract

Attacking Mac OS X has become a trend as we see more and more malware with advanced attack techniques on Mac OS X. In order to gain persistent control and avoid detection, malware have started to adopt rootkit tricks.

We will quickly review existing rootkit on Mac OS X, including both user and kernel mode, and approaches to detect them. In the major part of the presentation, we will disclose several new and advanced rootkit techniques by digging into more kernel objects and data structures. And we will demonstrate how to evade existing detection and memory forensics tools, such as Volatility.

Not only hiding things, tricks to gaining permission will also be discussed. It is not necessary to be root to get into kernel. And also, we will introduce techniques to start rootkit, special way to load kernel modules, and anti-tracing techniques.

The techniques we introduced have been tested on Mac OS X 10.9. There are new security features to verify 3rd party kernel modules in OS X 10.9, and we will tell you how do we bypass.

Table of Contents

1. Advanced Process Hiding

1.1 The rubilyn Rootkit

The rubilyn rootkit was released on full disclosure in 2012, and claimed the following capabilities:

- It works across multiple kernel versions (tested 11.0.0+)
- Give root privileges to pid.
- Hide files / folders
- Hide a process
- Hide a user from 'who'/'w'
- Hide a network port from netstat
- sysctl interface for userland control
- execute a binary with root privileges via magic ICMP ping

Although it is already 2 years old, it is still the most famous rootkit on Mac OS X.

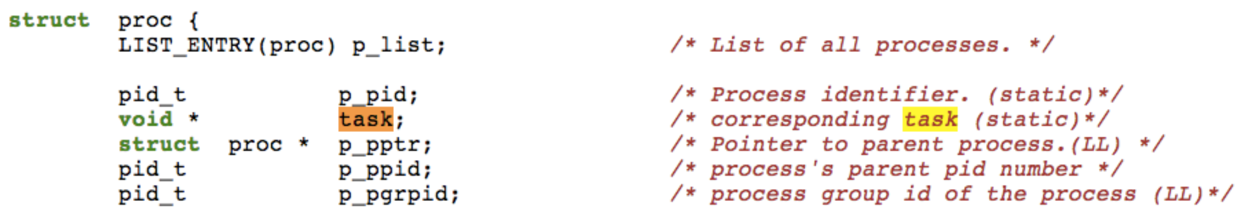
Here is the process structure in kernel:

```
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) */
        pid_t           p_ppid;                 /* process's parent pid number */
        pid_t           p_pgrpid;               /* process group id of the process (LL)*/

        lck_mtx_t       p_mlock;                /* mutex lock for proc */

        char            p_stat;                 /* S* process status. (PL)*/
        char            p_shutdownstate;
        char            p_kdebug;               /* P_KDEBUG eq (CC)*/
        char            p_btrace;               /* P_BTRACE eq (CC)*/

        LIST_ENTRY(proc) p_pglist;              /* List of processes in pgrp.(PGL) */
        LIST_ENTRY(proc) p_sibling;             /* List of sibling processes. (LL)*/
        LIST_HEAD(, proc) p_children;           /* Pointer to list of children. (LL)*/
        TAILQ_HEAD( , uthread) p_uthlist;       /* List of uthreads  (PL) */
```

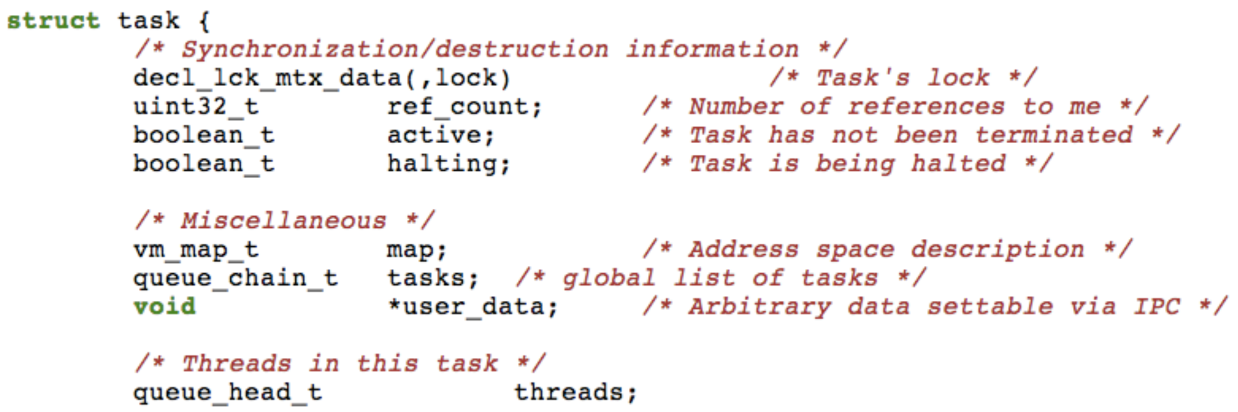
Rubilyn uses a simple DKOM (direct kernel object modification) to hide processes. It just unlinks p_list to hide process, so it is not difficult to detect rubilyn process hiding.

1.2 Detecting rubilyn Process Hiding

There is a corresponding task to each process, and tasks are also a linked-list like process list.



```
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) */
        pid_t           p_ppid;                 /* process's parent pid number */
        pid_t           p_pgrpid;               /* process group id of the process (LL)*/
```



```
struct task {
        /* Synchronization/destruction information */
        decl_lck_mtx_data(,lock)                /* Task's lock */
        uint32_t        ref_count;      /* Number of references to me */
        boolean_t       active;         /* Task has not been terminated */
        boolean_t       halting;        /* Task is being halted */

        /* Miscellaneous */
        vm_map_t        map;            /* Address space description */
        queue_chain_t   tasks;  /* global list of tasks */
        void            *user_data;     /* Arbitrary data settable via IPC */

        /* Threads in this task */
        queue_head_t            threads;
```

So we can easily detect rubilyn process hiding by listing tasks and comparing with process list. Actually rubilyn can only hide process from ‘ps’ command, however using Active Monitor can see process/task that hided by rubilyn.

1.3 Volatility and Bypass Volatility

Volatility is a well-know memory forensic tool. New version of Volatility can detect rubilyn rootkit.

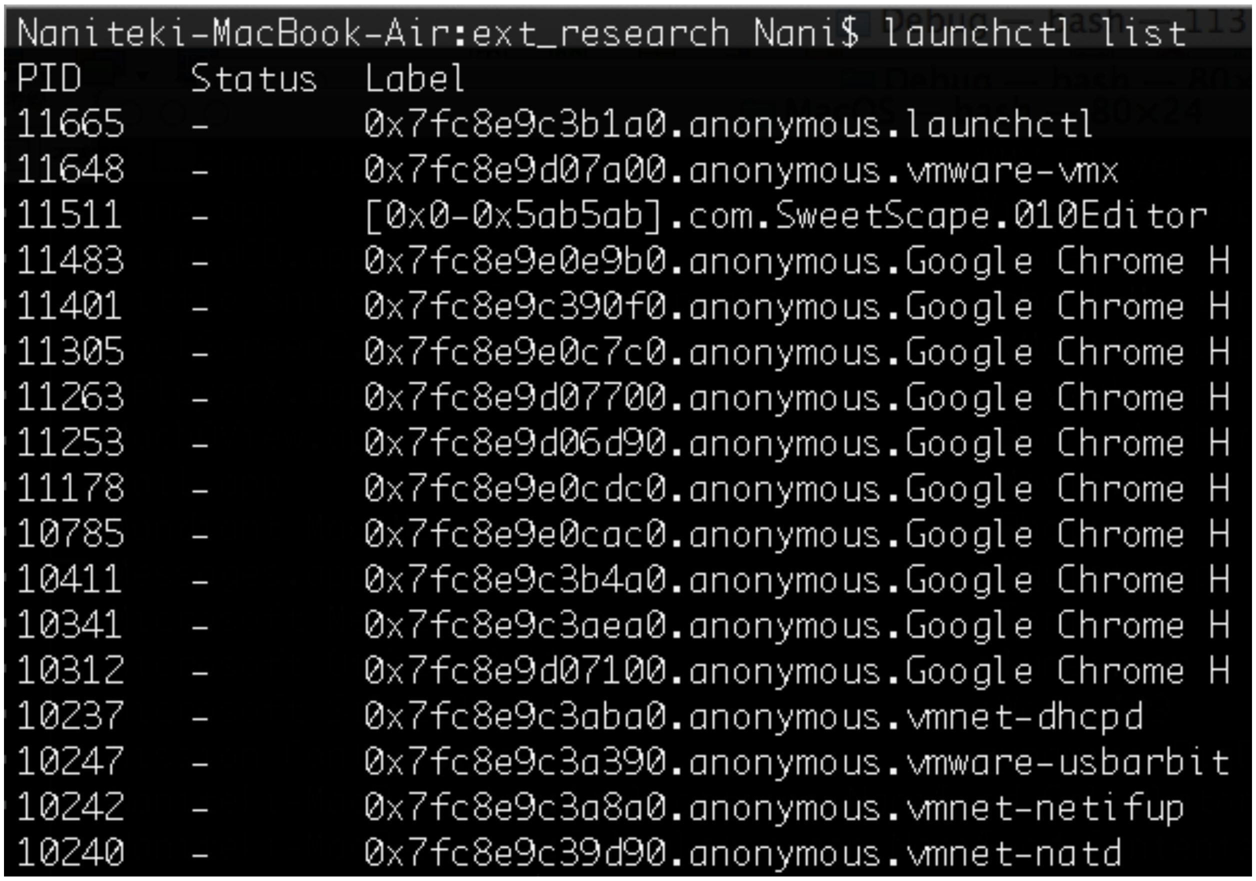
After some study on Volatility, we found that it checks p_list, p_hash, p_pglist, and task. So we can unlink p_list, p_hash, p_pglist, and task list, then Volatility cannot detect us.

Demonstration video: https://www.youtube.com/watch?v=_QD5YVSZz4U

1.4 Launchd Magic

In previous chapters, we did lots of hard works in kernel in order to hide process. However, there is a trick that we can easily find an invisible process from user mode!

Launchd is monitoring all process creation and termination. It maintains a job list in user mode. ‘launchctl’ is the tool to communicate with launchd. It can easily list jobs like this:



```
Naniteki-MacBook-Air:ext_research Nani$ launchctl list
PID     Status  Label
11665   -       0x7fc8e9c3b1a0.anonymous.launchctl
11648   -       0x7fc8e9d07a00.anonymous.vmware-vmx
11511   -       [0x0-0x5ab5ab].com.SweetScape.010Editor
11483   -       0x7fc8e9e0e9b0.anonymous.Google Chrome H
11401   -       0x7fc8e9c390f0.anonymous.Google Chrome H
11305   -       0x7fc8e9e0c7c0.anonymous.Google Chrome H
11263   -       0x7fc8e9d07700.anonymous.Google Chrome H
11253   -       0x7fc8e9d06d90.anonymous.Google Chrome H
11178   -       0x7fc8e9e0cdc0.anonymous.Google Chrome H
10785   -       0x7fc8e9e0cac0.anonymous.Google Chrome H
10411   -       0x7fc8e9c3b4a0.anonymous.Google Chrome H
10341   -       0x7fc8e9c3aea0.anonymous.Google Chrome H
10312   -       0x7fc8e9d07100.anonymous.Google Chrome H
10237   -       0x7fc8e9c3aba0.anonymous.vmnet-dhcpd
10247   -       0x7fc8e9c3a390.anonymous.vmware-usbarbit
10242   -       0x7fc8e9c3a8a0.anonymous.vmnet-netifup
10240   -       0x7fc8e9c39d90.anonymous.vmnet-natd
```

1.4.1 Unlink a job in Launchd

Here are the steps to unlink a job in launchd:

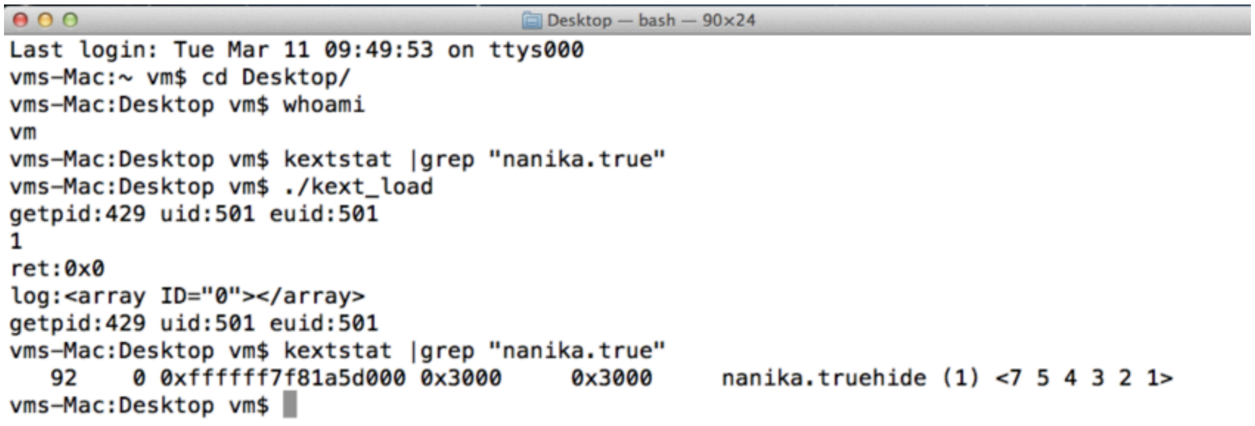
- Get root permission.
- Enumerate process launchd and get launchd task.
- Read launchd memory and find data section

- Find root_jobmgr
  - Check root_jobmgr->submgrs and submgrs->parentmgr
- Enumerate jobmgr and get job
- Enumerate job and find the target job
- Unlink the job

2. A Privileged Normal User

2.1 Running Privileged Tasks as a Normal User

Following picture shows that we can do privileged tasks as normal user:

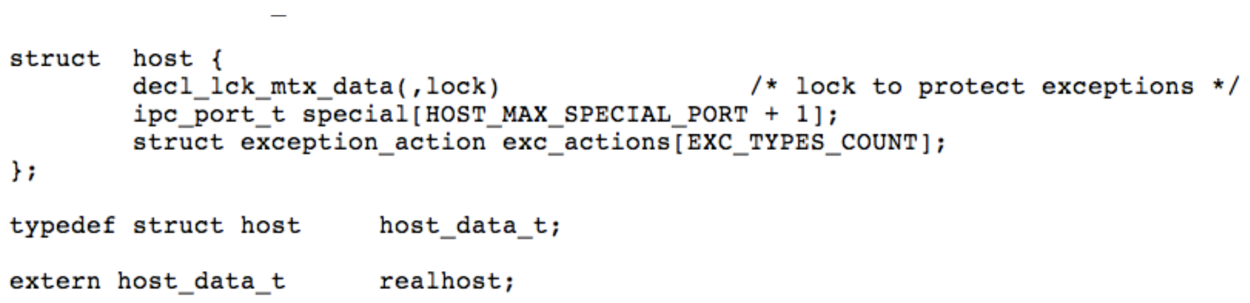


```
Last login: Tue Mar 11 09:49:53 on ttys000
vms-Mac:~ vm$ cd Desktop/
vms-Mac:Desktop vm$ whoami
vm
vms-Mac:Desktop vm$ kextstat |grep "nanika.true"
vms-Mac:Desktop vm$ ./kext_load
getpid:429 uid:501 euid:501
1
ret:0x0
log:<array ID="0"></array>
getpid:429 uid:501 euid:501
vms-Mac:Desktop vm$ kextstat |grep "nanika.true"
   92    0 0xffffff7f81a5d000 0x3000     0x3000     nanika.truehide (1) <7 5 4 3 2 1>
vms-Mac:Desktop vm$
```

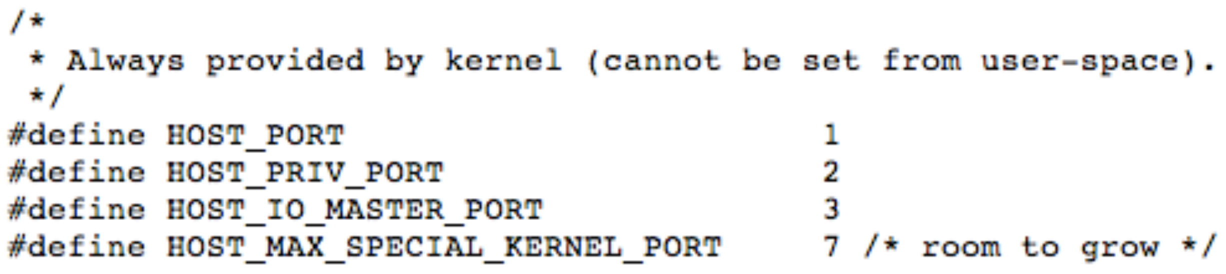
As a normal user vm (uid:501), we successfully loaded a kernel module ‘nanika.true’. How did we do this?

2.2 Host Privilege

In Mac OS X, when a process performing a task that requires permission, it doesn’t check uid of the process, instead, it checks if the task is granted the Host Privilege.

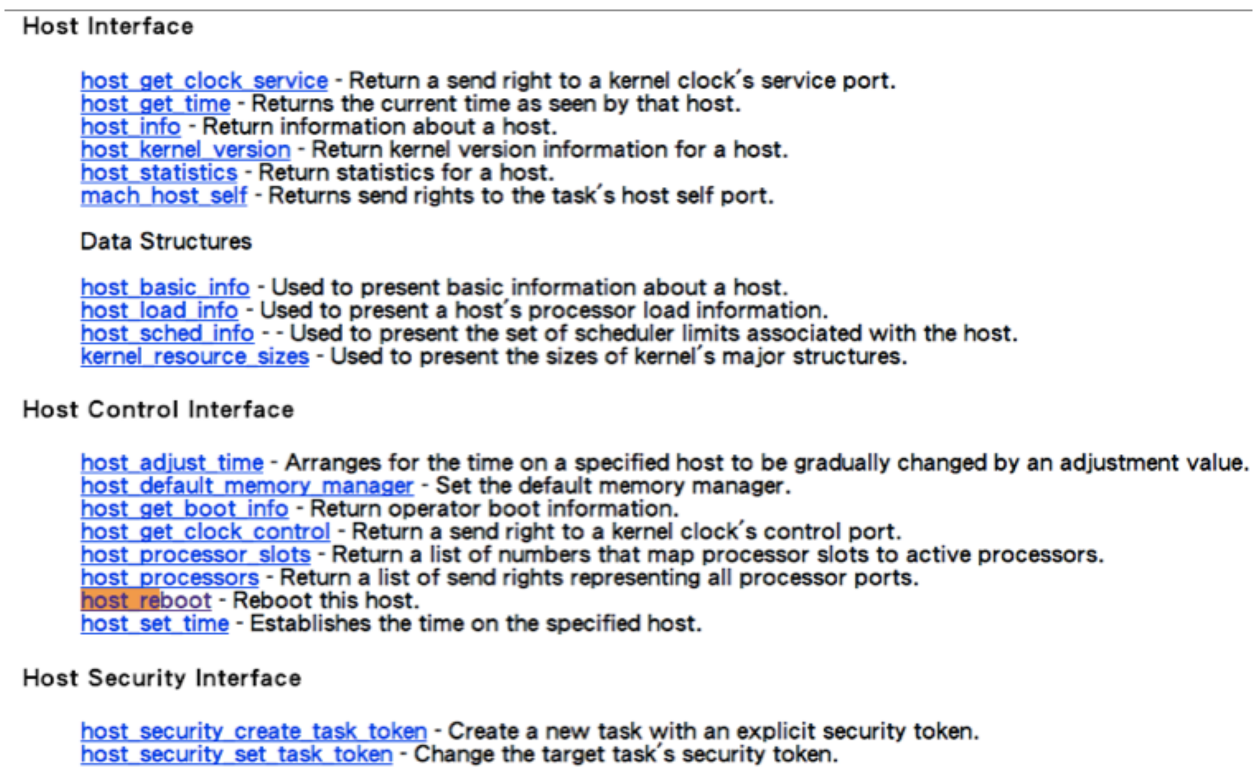


```
struct  host {
        decl_lck_mtx_data(,lock)                /* lock to protect exceptions */
        ipc_port_t special[HOST_MAX_SPECIAL_PORT + 1];
        struct exception_action exc_actions[EXC_TYPES_COUNT];
};

typedef struct host     host_data_t;

extern host_data_t      realhost;
```



```
/*
 * Always provided by kernel (cannot be set from user-space).
 */
#define HOST_PORT                       1
#define HOST_PRIV_PORT                  2
#define HOST_IO_MASTER_PORT             3
#define HOST_MAX_SPECIAL_KERNEL_PORT    7 /* room to grow */
```

Here is a list of the things we can do with host privilege:



Host Interface

host_get_clock_service - Return a send right to a kernel clock's service port.
host_get_time - Returns the current time as seen by that host.
host_info - Return information about a host.
host_kernel_version - Return kernel version information for a host.
host_statistics - Return statistics for a host.
mach_host_self - Returns send rights to the task's host self port.

Data Structures

host_basic_info - Used to present basic information about a host.
host_load_info - Used to present a host's processor load information.
host_sched_info - - Used to present the set of scheduler limits associated with the host.
kernel_resource_sizes - Used to present the sizes of kernel's major structures.

Host Control Interface

host_adjust_time - Arranges for the time on a specified host to be gradually changed by an adjustment value.
host_default_memory_manager - Set the default memory manager.
host_get_boot_info - Return operator boot information.
host_get_clock_control - Return a send right to a kernel clock's control port.
host_processor_slots - Return a list of numbers that map processor slots to active processors.
host_processors - Return a list of send rights representing all processor ports.
host_reboot - Reboot this host.
host_set_time - Establishes the time on the specified host.

Host Security Interface

host_security_create_task_token - Create a new task with an explicit security token.
host_security_set_task_token - Change the target task's security token.

And actually it can have permission to control a tasks via these API:

- processor_set_default
- host_processor_set_priv

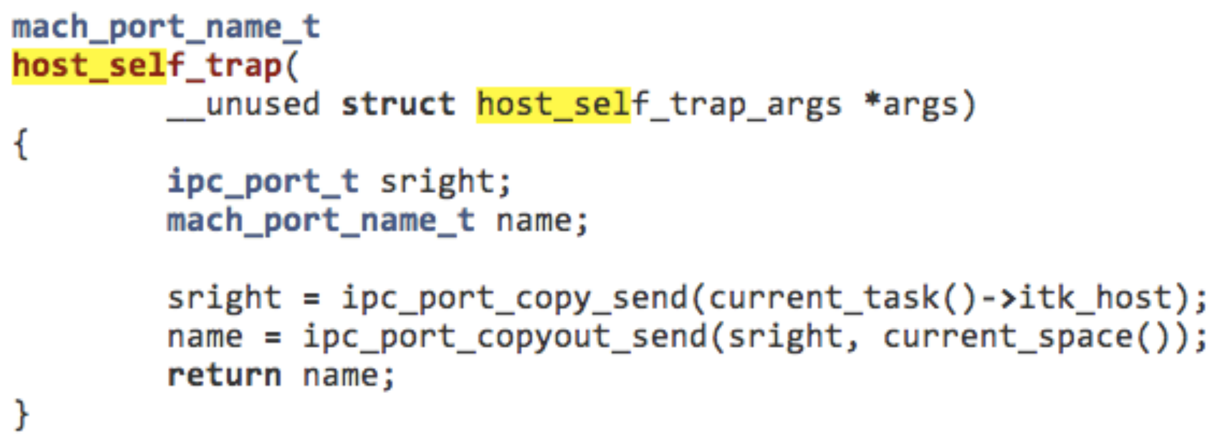
- processor_set_tasks

Host privilege gives a process power to do a lot of things. That's the reason why we can load a kernel module as a normal user.

2.3 How to Get Host Privilege

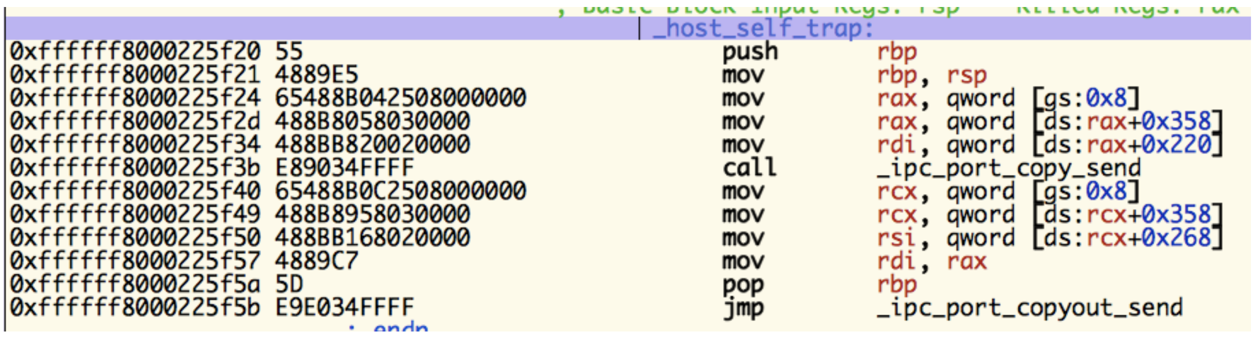
There are 3 ways to grant host privilege to a regular process:

- Assign host privilege to a task
  - Parse mach_kernel and find _realhost
  - Find task structure
  - Assign permission: task->itk_host = realhost->special[2]
  - Then the task/process can do privilege things.
- Hook system call (Global)
  - When process is retrieving the task information, make it return with host privilege.
  - Patch code (Global, good for rootkit)
  - Here is the code we are going to patch: (host_self_trap)



```
mach_port_name_t
host_self_trap(
        __unused struct host_self_trap_args *args)
{
        ipc_port_t sright;
        mach_port_name_t name;

        sright = ipc_port_copy_send(current_task()->itk_host);
        name = ipc_port_copyout_send(sright, current_space());
        return name;
}
```

  - Patch code:



```
                                             ; Basic Block Input Regs: rsp    Killed Regs: rax
                                             | _host_self_trap:
0xffffff8000225f20 55                           push     rbp
0xffffff8000225f21 4889E5                       mov      rbp, rsp
0xffffff8000225f24 65488B042508000000           mov      rax, qword [gs:0x8]
0xffffff8000225f2d 488B8058030000               mov      rax, qword [ds:rax+0x358]
0xffffff8000225f34 488BB820020000               mov      rdi, qword [ds:rax+0x220]
0xffffff8000225f3b E89034FFFF                   call     _ipc_port_copy_send
0xffffff8000225f40 65488B0C2508000000           mov      rcx, qword [gs:0x8]
0xffffff8000225f49 488B8958030000               mov      rcx, qword [ds:rcx+0x358]
0xffffff8000225f50 488BB168020000               mov      rsi, qword [ds:rcx+0x268]
0xffffff8000225f57 4889C7                       mov      rdi, rax
0xffffff8000225f5a 5D                           pop      rbp
0xffffff8000225f5b E9E034FFFF                   jmp      _ipc_port_copyout_send
                                  ; endp
```

call _host_self

mov rax, [rax+0x20]

mov rdi, rax

3. Direct Kernel Task Access (Read/Write)

3.1 Access Tasks Objects in Kernel from User Mode

Since Mac OS X 10.6, it restricted task access for kernel task. According to this report:

> "task_for_pid() is not supported on the kernel task, no matter your privilege level nor what API you use.
>
> … there is no legitimate use for inspecting kernel memory."

However, we discovered a way to direct access kernel task memory. We don't use task_for_pid(), instead we use processor_set_tasks().

- processor_set_tasks(p_default_set_control, &task_list, &task_count);
- then, **task_list[0]** is the kernel task!

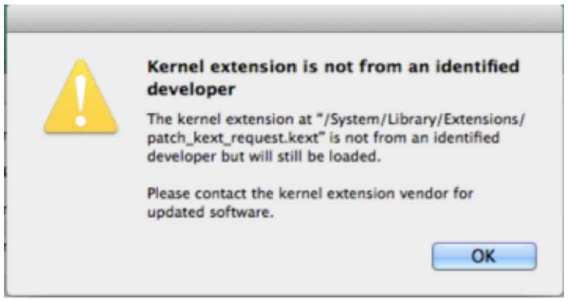
We can control all of tasks and read / write memory, even use thread_set_state() to inject dynamic libraries.

4. Bypass Kernel Module Verification in 10.9

4.1 Loading a Kernel Module

In Mac OS 10.9, if you want to load a kernel module you have to:

- Put the kernel module file into /System/Library/Extensions/
- Run kextload to load the file
- If the kernel module is not signed, OS will pop up a warning message.



Kernel extension is not from an identified developer

The kernel extension at "/System/Library/Extensions/patch_kext_request.kext" is not from an identified developer but will still be loaded.

Please contact the kernel extension vendor for updated software.

You can see there are many limitations.

Surprisingly, we found a way to break these limitations. We can:

- Load a kernel module from any path.
- Load a kernel module on the fly, from a memory buffer, etc. File is not required.
- Load a kernel module without verification. (no warning message)
- No need to patch kextd.

4.2 kext_request()

Using kext_reqest() to load kernel module, we can bypass many verifications. Following are steps to use kext_request():

- Get kext data ready. You need to know mkext

```
typedef struct mkext2_file_entry {
    uint32_t  compressed_size;  // if zero, file is not compressed
    uint32_t  full_size;        // full size of data w/o this struct
    uint8_t   data[0];          // data is inline to this struct
} mkext2_file_entry;

typedef struct mkext2_header {
    MKEXT_HEADER_CORE
    uint32_t plist_offset;
    uint32_t plist_compressed_size;
    uint32_t plist_full_size;
} mkext2_header;
```

- Get your host privilege. It checks the privilege.

```
    if (isMkext) {
#ifdef SECURE_KERNEL
        // xxx - something tells me if we have a secure kernel we don't even
        // xxx - want to log a message here. :-)
        *op_result = KERN_NOT_SUPPORTED;
        goto finish;
#else
        // xxx - can we find out if calling task is kextd?
        // xxx - can we find the name of the calling task?
        if (hostPriv == HOST_PRIV_NULL) {
            OSKextLog(/* kext */ NULL,
                kOSKextLogErrorLevel |
                kOSKextLogLoadFlag | kOSKextLogIPCFlag,
                "Attempt by non-root process to load a kext.");
            *op_result = kOSKextReturnNotPrivileged;
            goto finish;
        }

        *op_result = OSKext::loadFromMkext((OSKextLogSpec)clientLogSpec,
            request, requestLengthIn,
            &logData, &logDataLength);
```

- Call kext_request() to load the kernel module.
- Then you won't get any problems.

5. A Trick to Gain Root Permission

We mentioned many techniques that could be used in a rootkit. However, all of these tricks require the permission. We noticed a design problem that could be leveraged by malware to gain root permission.

Authorization rights are a core part of Mac OS X's security. Rights determine who can and cannot access specific functionality. This is controlled by securityd. It provides a mechanism for applications to gain root permission.

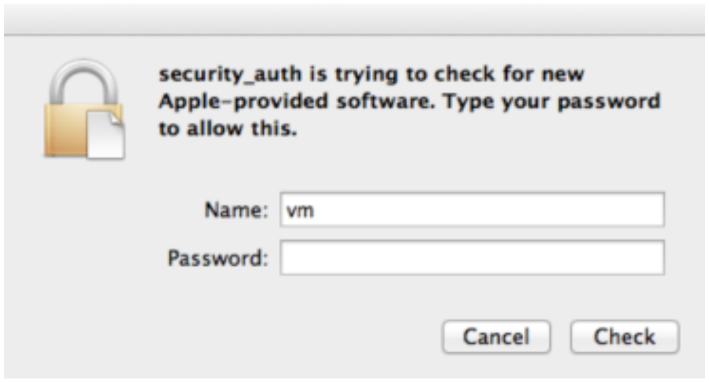
When an application requires root permission, it could send request to get specific right. For example:

- system.privilege.admin
- system.privilege.taskport
- com.apple.ServiceManagement.daemons.modify
- com.apple.ServiceManagement.blesshelper

Then user will see a pop up window and ask for password to confirm.

However, one of right is interesting: **com.apple.SoftwareUpdate.scan**

No matter who request this right, user will see a window like this:


security_auth is trying to check for new Apple-provided software. Type your password to allow this.

Name: vm

Password:

“security_auth is trying to check for new APPLE-PROVIDED software”. We think most of users will type the password and won’t feel anything wrong. After typed the password, we can gain root permission.

6. Conclusion

In this paper, we introduced several tricks that could be used by rootkit.

- Advanced Process Hiding: it could hide processes and bypass detection by all existing security software.

- A Privileged Normal User: rootkit can use this trick to create a ‘normal’ power user. It won’t be noticed easily.
- Direct Kernel Task Access: easier to access process memory.
- Loading Kernel Module Without Warnings: more flexible way to load rootkit modules.
- A Trick to Gain Root Permission: the trick might be used by malware to gain the 1st permission.