ADVANCED MAC OS X ROOTKITS

DINO A. DAI ZOVI
DDZ@THETA44.ORG

Abstract. The Mac OS X kernel (xnu) is a hybrid BSD and Mach kernel. While Unix-oriented rootkit techniques are pretty well known, Mach-based rootkit techniques have not been as thoroughly publicly explored. This paper covers a variety of rootkit techniques for both user-space and kernel-space rootkits using unique and poorly understood or documented Mac OS X and Mach features.

1. Introduction

Rootkit techniques affecting FreeBSD are well known and documented ([4]). Many of these techniques also apply directly to Mac OS X since it shares a good deal of code with FreeBSD. This research, however, focuses on rootkit techniques that use or abuse unique features of the Mach functionality in the Mac OS X kernel.

This paper is organized as follows. First, some background is given on Mach abstractions, IPC, and RPC. The body of the paper proceeds in describing injecting code into running processes, injecting new in-kernel RPC subsystems, transparent remote control of a compromised system via Mach IPC, and detecting hidden kernel rootkits. Finally, the paper concludes with some commentary on future directions and gives the reader instructions on how to obtain the proof-of-concept tools described in this paper.

2. Background

The Mac OS X kernel (xnu) is an operating system kernel of mixed lineage, combining the older research-oriented Mach microkernel with the more traditional and modern FreeBSD monolithic kernel. The Mach microkernel combines a powerful abstraction, Mach message-based interprocess communication (IPC) with a number of cooperating servers to form the core of an operating system. The Mach microkernel is responsible for managing separate tasks, each in their own separate address spaces and consisting of multiple threads. A number of default

Date: July 24, 2009.

servers also provide virtual memory paging, a system clock, and other miscellaneous low-level services.

The Mach microkernel alone, however, is not enough to implement a modern operating system. In particular, it has no notion of users, groups, or even files. In order to provide this functionality, the Mac OS X kernel includes a graft of the FreeBSD kernel. The top-half of the FreeBSD kernel (system call handlers, file systems, networking, etc) was ported to run on top of the Mach microkernel. In order to address the performance concerns with excessive IPC messaging between kernel components, both kernels exist in the same privileged address space. However, in most respects, the Mach API visible from kernel code is the same as the Mach API visible from user processes.

The Mach kernel deals with a number of core abstractions: the host, task, thread, and ports that represent them. The Mach *host* represents a computer system that is capable of running *tasks*. Each task encapsulates a number of resources, including a virtual memory address space, one or more *threads* of execution control, and an IPC space. The Mach kernel, tasks, and threads all communicate with each other through *ports*. A port is a unidirectional, sequenced, and structured data channel that is capable of transmitting *messages*. Each port in a task's IPC space is referred to by a task-specific *name*. Different tasks may refer to the same port via different names. Access to a port is governed by a task possessing defined *rights* on it. Multiple tasks may have rights to send messages on a port, however only one task may have rights to receive messages on it. In addition to sending data, tasks may transmit port rights through messages to other tasks. This system of ports and port rights forms Mach's capability-based security model.

Unidirectional IPC is only of limited use. In order to provide a more flexible system, Mach provides a Remote Procedure Call (RPC) system on top of Mach IPC. Tasks may use RPC to interact with other tasks on the same host. Tasks that primarily perform actions on behalf of other tasks are referred to as *servers*. Mach-based systems commonly provide MiG, the Mach Interface Generator to aid in constructing RPC clients and servers. MiG takes a definition file specifying the RPC interface (referred to as a *subsystem*) and creates both client and server stub functions to automatically handle the marshalling of data into and from Mach messages. MiG-generated client stubs are actually used in most of the Mach API to communicate with MiG-generated server stubs executing within the kernel.

3. Mach Injection

The Mach API supports a high-level of control over other tasks and threads. With access to another task's control port, a controlling task may allocate, deallocate, read, and write memory within the task. It may also create, suspend, and kill threads within the controlled task. These operations are sufficient to map new code into the remote task and create a new thread to execute that code. There are, however, some complications with this.

First, the injected code will not be able to call many system APIs. The code will be executing within a "naked" Mach thread and many system functions assume that all threads are POSIX threads. Second, the injected code may need to be linked in order to call system libraries. Purely injected code is essentially limited to the level of functionality of common exploit payloads used in memory corruption exploits.

In order to address these shortcomings, direct Mach thread injection is simply used to call functions loaded in the remote processes address space. A small bit of trampoline code is injected in order to promote the "naked" Mach thread into a full POSIX thread so that the full range of standard library functions may be called. In addition, we use a unique feature of Mach in order to obtain the return value of the function call in the remote address space. In addition to setting up the thread, the injected trampoline sets a bogus stack return address intended to make the thread crash in a deterministic way. Prior to starting the thread, the controlling task registers itself as an exception handler for that thread. By doing so, it is notified about exceptions in the thread before the owning task is. This allows the controlling task to handle the exception and terminate the thread without disturbing the rest of the task. When the injected thread crashes at the bogus return address, the return value from the function is obtained from the appropriate CPU register in the thread state. By combining these steps, the controlling task is able to call arbitrary functions in the target task with chosen arguments and obtain the returned value of the called function.

Using this technique, `dlopen()` is called in the remote process in order to properly load and link a bundle from disk. After loading, `dlsym()` is called to resolve the address of the `run()` symbol exported from the bundle. Finally, this function is called in order to run the injected bundle in the remote task.

For more complete details on this technique, see the author's *inject-bundle* tool (available as described in Section 7) and Chapter 11 of *The Mac Hacker's Handbook* [5].

4. Kernel RPC Subsystem Injection

A number of Mach RPC servers exist within the kernel. The Mach IPC abstractions allow RPC servers to run in the kernel or user-mode Mach tasks, often only requiring a recompile with different options to MiG. In Mac OS X Leopard, the Mach clock, clock_priv, host_priv, host_security, ledger, lock_set, mach_host, mach_port, mach_vm, processor, processor_set, security, task, thread_act, and vm_map subsystems are all implemented as kernel servers. Mac OS X adds a few new in-kernel Mach servers for the IOKit and UserNotification systems. Running these servers in the kernel improves performance as the delivery of Mach IPC messages requires less mode switches between user and kernel mode.

In the xnu kernel, IPC messages received by the kernel are routed to in-kernel RPC servers through the `mig_buckets` hash table. This data structure and the functions that operate on it are defined in the xnu kernel source file `osfmk/kern/ ipc_kobject.c`. Each routine in the MiG subsystem is inserted in the hash table by storing its routine identifier, function pointer, and maximum message size. Incoming messages have their `msgh_id` header field set to the routine identifier and the kernel function `ipc_kobject_server` uses this field to find the RPC server routine to handle the request.

While the `mig_buckets` hash table is statically initialized by the kernel, a rootkit can easily dynamically inject new subsystems into it, as described in [5] and shown in Figure 1. The injected or existing subsystems can also be removed as shown in Figure 2.

Just as easily as new subsystems are added or removed, existing subsystems can be modified. The subsystems in this hash table perform many of the critical functions of the Mach kernel, including task, thread, and memory management. A kernel-based rootkit could overwrite the routine pointers in this table in order to intercept messages to these servers. The request messages and their replies could be modified in transit in order to modify the behavior of the in- kernel Mach RPC server. In this respect, this technique is similar to how more traditional rootkits intercept system calls by patching the system call table on Unix-like operating systems or the System Service Table on Windows-based operating systems.

5. Machiavelli

Historical Mach-based operating systems often included the NetMessage Server [2]. The NetMessage Server transparently extended Mach IPC across the network to other hosts. Each node ran a NetName

```
int inject_subsystem(const struct mig_subsystem * mig)
{
  mach_msg_id_t h, i, r;

  // Insert each routine into mig_buckets hash table
  for (i = mig->start; i < mig->end; i++) {
    mig_hash_t* bucket;

    h = MIG_HASH(i);
    do {
      bucket = &mig_buckets[h % MAX_MIG_ENTRIES];
    } while (mig_buckets[h++%MAX_MIG_ENTRIES].num !=0 &&
             h < MIG_HASH(i)+MAX_MIG_ENTRIES);

    if (bucket->num == 0) {
      // We found a free spot
      r = mig->start - i;

      bucket->num = i;
      bucket->routine = mig->routine[r].stub_routine;
      if (mig->routine[r].max_reply_msg)
        bucket->size = mig->routine[r].max_reply_msg;
      else
        bucket->size = mig->maxsize;
      }
      else {
        // Table was full, return an error
        return -1;
      }
  }

  return 0;
}
```

LISTING 1. Inserting a new subsystem into the kernel server hash table

server that acted as a registry of ASCII string names to Mach server ports for servers running on that host. Clients on a given node could lookup ports registered in NetName servers on their local or remote hosts through their own local NetName server. The ports returned by the local NetName server would in fact be ports to the local NetMessage server. The NetMessage server would act as a proxy to the server running on the remote host by sending and receiving messages across the network. In Mac OS X, the functionality of the NetName server has been subsumed by the Bootstrap Server run within launchd [7]. This

```
int remove_subsystem(const struct mig_subsystem * mig)
{
  mach_msg_id_t h, i;

  // Remove each routine exhaustively from the
  // mig_buckets table
  for (i = mig->start; i < mig->end; i++) {
    for (h = 0; h < MAX_MIG_ENTRIES; h++) {
      if (mig_buckets[h].num == i) {
        bzero(&mig_buckets[h], sizeof(mig_buckets[h]));
      }
    }
  }

  return 0;
}
```

LISTING 2. Removing a subsystem from the kernel server hash table

Bootstrap Server only holds ports for servers running on the local host and provides no functionality analagous to the NetMessage Server.

The high level of abstraction and control provided by Mach IPC makes it an ideal facility for remote control of a Mach-based system. The author's proof-of-concept rootkit, *Machiavelli*, does just this by implementing a facility similar in functionality to the NetMessage Server, however with the spirit and goals of a covert rootkit. Mach IPC messages are also programming language and byte ordering neutral. While the current implementation uses the native MiG RPC client stub routines to marshal IPC messages, an alternate implementation could marshall IPC messages by hand in any programming language.

Machiavelli consists of a Mach proxy server on the local controlling host and a number of remote agent servers that run on remote compromised hosts. On the controlling host, rootkit management utilities obtain a proxy Mach port from the proxy server and use it just as a normal application would use a local Mach port. For example, MiG-generated RPC client routines may be used with the proxy port in order to execute the RPC request on the remote compromised host instead of the local host. The Machiavelli proxy server receives the Mach IPC message and transmits it over the network to the remote agent for actual processing by the destination RPC server.

5.1. **Machiavelli API.** From the client software's perspective, there is little difference in performing Mach RPC with local or remote servers. Normally, an application would obtain send rights to the local host,

```
#include <stdio.h>
#include <stdlib.h>
#include <mach/mach.h>
#include "machiavelli.h"

int main(int argc, char* argv[])
{
  kern_return_t kr;
  machiavelli_t m = machiavelli_init();
  mach_port_t port;
  vm_size_t page_size;

  machiavelli_connect_tcp(m, "192.168.13.37", "31337");

  port = machiavelli_get_port(m, HOST_PORT);

  if ((kr = _host_page_size(port, &page_size)) !=
       KERN_SUCCESS) {
    errx(EXIT_FAILURE, "_host_page_size:␣%s",
            mach_error_string(kr));
  }

  printf("Host␣page␣size:␣%d\n", page_size);

  return 0;
}
```

LISTING 3. Using Machiavelli to access a remote host port

task, or thread ports with `mach_host_self()`, `mach_task_self()`, or `mach_thread_self()` respectively. Alternatively, a privileged process may obtain send rights for another unrelated task through the `task_for_pid()` BSD system call. A Machiavelli utility on the other hand obtains a proxy port for a remote port through the `machiavelli_get_port()` function.

Consider the example in Figure 3. This small utility retrieves the page size through a remote Machiavelli agent. In line 9, the local Machiavelli proxy is initialized. On line 13, the local proxy is instructed to connect to a remote Machiavelli agent via TCP on IP 192.168.13.37 and port 31337. On line 15, the call to `machiavelli_get_port()` obtains a proxy port for the Mach host port on the remote system. This same function call can be used to obtain other special ports such as the host privileged port, IO master port, User Notification port, kernel task as well as task ports for a given process identifier. Finally on line 17, the utility calls `_host_page_size()`, which is a MiG generated RPC client

stub in the `mach_host` subsystem. Normally, the first argument is a host port obtained through `mach_host_self()`. In this case, we specify our Machiavelli proxy port so that the Mach RPC request is intercepted and handled by the RPC server on the remote system.

5.2. **Serializing and Deserializing Mach IPC Messages.** Serializing simple Mach messages to byte buffers suitable for transmission over a network is straight-forward. In effect, they are already suitable for transmission as-is with the simple step of translating remote port names to local proxy port names. In both the Machiavelli Proxy and Machiavelli Agents, proxy ports are used to intercept Mach messages that should be sent over the network. A local proxy port name with the same port rights is substituted for any port name that exists in the remote port namespace. When a message is received on a proxy port, the original remote proxy name is placed in the message header before it is transmitted to the remote system. This is always performed for the reply port that may be specified in the `msgh_local_port` field in the Mach message header and, as described below, may also be performed on additional transferred port rights.

Serializing complex Mach messages is a little more involved. Complex Mach messages include a number of descriptors for auxiliary out-of-line data, including additional Mach ports, large memory buffers, and port arrays. An additional port right transferred via a Mach message is handled identically as the implicitly transferred port right for the reply port specified in the message header. When out-of-line memory is attached to a Mach message, the descriptor contains the address of the beginning and the size of the memory buffer. On a local system, the entire memory pages containing this memory buffer are remapped into the task receiving the message. In order to transmit this memory to a remote system, we append the memory buffer to end of the Mach message. When the message is deserialized, this memory buffer is copied into freshly allocated memory pages. The buffer is not copied to the beginning of the memory pages, though. As per the original semantics of the out-of-line memory transfers, the sent memory must be placed at the same offset from the beginning of a memory page as it was found at in the sending task. Also, depending on flags in the out-of-line memory descriptor and as passed by the message receiver, the memory pages may need to be allocated at the specified virtual address. A complex Mach message may also include out-of-line port arrays. These arrays include both port names and port rights that have been transferred to the receiving task. In serialization, the contents of these arrays are appended to the message. In deserialization, the array

is restored and the port name are replaced with the port names of local proxy ports created with the port rights specified in the port array.

5.3. **Machiavelli Proxy.** The Machiavelli Proxy is the component of Machiavelli that runs locally on the controlling host. It is responsible for intercepting IPC messages on its proxy ports and transmitting them to the remote Machiavelli Agent, which relays them to the remote RPC server. A local application interacts with the Machiavelli Proxy by initializing it, connecting to the remote Machiavelli Agent, and requesting send rights for remote ports through the Machiavelli API. The Machiavelli Proxy returns send rights for proxy ports where it holds the receive rights. These ports are maintained in a port set that it receives messages on. When the Machiavelli Proxy receives a message on one of the ports in the set, it translates the message and transmits the message to the remote Machiavelli Agent.

While it currently runs as a background thread in any program that uses the Machiavelli API, it could also have been implemented as a background daemon with its server port registered in `lookupd`. This would allow multiple control utilities to share a single connection to the remote compromised system. This will be considered for future work.

5.4. **Machiavelli Agents.** The Mach API is largely identical for user-mode and in-kernel Mach servers. This allows us to easily implement both user-mode and in-kernel Machiavelli agents. In addition, the serialized Mach messages can be sent and received through any reliable data channel giving us a number of options for implementation of Machiavelli Agents.

The first proof-of-concept Machiavelli Agent is implemented as a user-mode process listening on a TCP socket. This is the simplest implementation that still allows full remote control of the system. If the agent is run as root, it will have access to the task ports for any process via the `task_for_pid()` system call. If `task_for_pid()` is called with a `pid` argument of 0, it returns the kernel task control port. This port allows remote direct memory operations on kernel memory. Combined with remote access to the privileged host port, this allows effective and high-level control of the remote kernel, including the ability to load or unload kernel extensions, read or write kernel memory, and create or suspend kernel threads.

The user-mode TCP Machiavelli Agent supports a simple protocol. Upon establishing the connection, the server sends the client the remote Mach port names for the host, task, and thread ports as well as a Mach

port name of the Machiavelli RPC server. Following this, the server and client communicate exclusively through serialized Mach RPC messages.

As described in [5], Mac OS X kernel rootkits may also take advantage of Network Kernel Extensions (NKE) APIs. The rootkit may use these to intercept network traffic at multiple levels within the kernel network stack, including ethernet frames directly from the network interface on through fully after IP defragmentation and TCP reassembly. Any number of these points may be used to implement a remote network covert channel for communication with an in-kernel Machiavelli Agent.

6. Unloaking Kernel Rootkits

While there have been few Mac OS X kernel rootkits observed in the wild, a number of proof-of-concept rootkits have been publicly released or presented. One of the first rootkits was nemo's WeaponX rootkit [6]. Other similar rootkit techniques have been written about in *Developing Mac OSX Rootkits* [8] and *The Mac Hacker's Handbook* [5]. These rootkits share a common technique to hide themselves from kernel extension listings by removing themselves from the kernel module linked list (`kmod`).

While a kernel module that is not present in the kmod linked list will not be able to be enumerated or removed, it will still be visible in memory. The memory allocated for the kernel module will also be listed in the kernel memory map. This memory map may be enumerated by gaining access to the kernel task control port by passing a `pid` argument of 0 to `task_for_pid()` and calling `mach_vm_region()` on that task port. This will enumerate the memory regions and their page protection permissions for the kernel memory address space.

The Mach host port also provides an interface to enumerate the loaded kernel modules, `kmod_get_info()`. This routine can be used to enumerate the loaded kernel modules, including the address and size of their loaded code. For proper usage of this function, see the source code for `kextstat` [1].

Detecting hidden rootkits is simply a matter of correlating Mach-O executable objects in the kernel address space with loaded code from either the kernel or kernel modules enumerated through `kmod_get_info()`. This is done by iterating across the memory regions allocated in the kernel and examining the beginning of them for a Mach-O header. The Mach-O headers will contain metadata describing its loaded code segment, which should represent the kernel or a loaded kernel module. Any Mach-O executables that cannot be substantiated back to the

kernel itself or a loaded kernel module should be considered suspicious. The suspicious Mach-O executable objects can then dumped to disk for further analysis.

For an example implementation of this technique, see the author's *Uncloak* tool, available as described in Section 7.

It should be noted, however, that this only identifies rootkits that have not also hidden themselves from the kernel memory maps. As described above in Section 4, a rootkit may intercept IPC messages to in-kernel servers and modify the replies. A rootkit that is aware of this detection technique could easily hide its memory regions from discovery by intercepting messages to the appropriate `mach_vm` subsystem routines.

7. Availability

In order to demonstrate these techniques, a number of proof-of-concept tools have been developed. These tools will be made available on the author's web site [3] shortly after the BlackHat USA 2009 conference.

- **Inject Bundle**: Inject a bundle loaded from disk into a running process
- **iChatSpy**: Swizzle iChat Objective-C methods in order to log IM messages
- **SSLSpy**: Hook SecureTransport SSL functions to log SSL traffic
- **iSightSpy**: Capture a single frame from the iSight camera
- **Machiavelli**: Remotely control a compromised system through remote Mach IPC
- **Uncloak**: Identify and dump kernel modules that have removed themselves from the `kmod` linked list.

References

1. Apple, *kextstat*, http://www.opensource.apple.com/source/kext_tools/kext_tools-117.4/kextstat_main.c.
2. Joseph Boykin, David Kirschen, Alan Langerman, and Susan LoVerso, *Programming under mach*, Addison-Wesley Professional, 1993.
3. Dino A. Dai Zovi, *Trail of bits*, http://trailofbits.com/.
4. Joseph Kong, *Designing bsd rootkits*, No Starch Press, 2007.
5. Charlie Miller and Dino A. Dai Zovi, *The mac hacker's handbook*, Wiley, 2009.
6. nemo, *Weaponx*, http://packetstormsecurity.org/UNIX/penetration/rootkits/wX.tar.gz.
7. Amit Singh, *Mac os x internals*, Addison-Wesley Professional, 2006.
8. wowie and ghalen, *Developing mac osx kernel rootkits*, Phrack **13** (2009), no. 66, 16.