Liar! Macs have no viruses!
Don’t take me too seriously, I fuzz the Human brain!
The capitalist "pig" degrees: Economics & MBA.
Worked for the evil banking system!
Security Researcher at COSEINC.
"Famous" blogger.
Wannabe rootkits book writer.
Love a secondary curvy road and a 911.
Today's subject

- OS X Kernel rootkits.
- Ideas to improve them.
- Sample applications.
- Raise awareness and interest in this area.
Prologue

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.
- Should be easier to find the necessary bugs.
- Less research = Less audits = More bugs.
- Main target is Mountain Lion.
- Also works with Mavericks (tested with DP1).
Current state of the “art”

- OS X rootkits are a rare "species".
- Interesting hardware rootkits (Dino's Vitriol and Snare’s EFI) but NO full code available 😞.
- Commercial rootkits: Crisis from Hacking Team and maybe FinFisher (iOS yes, OS X never saw).
- Crisis is particularly bad.
- Not many detection tools available.
IT’LL SIMPLIFY A LOT OF TASKS YOU NEVER HAD TO DO BEFORE
Problem #1

- Many interesting kernel symbols are not exported.
- Some are available in Unsupported & Private KPIs.
- Not acceptable for stable rootkits.
- Solving kernel symbols from a kernel extension is possible since Lion.
- Not in Snow Leopard and previous versions.
Simple Ideas

- __LINKEDIT segment contains the symbol info.
- Zeroed up to Snow Leopard.
- OS.X/Crisis solves the symbols in userland and sends them to the kernel rootkit.

```
.loc_E4EB:
  ; CODE XREF: _solveKernelSymbolsForKext+C71j
  mov [esp], edi ; mmap'ed kernel start address
  mov dword ptr [esp+4], 0DD2C36D6h ; symbol to solve
  call __findSymbolInFatBinary
  mov [ebp+var_98], 0DD2C36D6h
  mov [ebp+var_94], eax ; kmod address
  mov eax, ds:[Fcpu_filedecriptor - 0E067h][esi]
  lea ebx, [ebp+var_98]
  mov [esp+8], ebx
  mov [esp], eax ; int
  mov dword ptr [esp+4], 807AEEBFh ; send solved symbol request
  call __ioctl ... ; 0x807aeebf
```
Simple Ideas

- One easy solution is to read the kernel image from disk and process its symbols.
- The kernel does this every time we start a new process.
- Possible to implement with stable KPI functions.
- Kernel ASLR slide is easy to obtain in this scenario.
I need help on my homework. What's a pronoun?

A noun that lost its amateur status.

Maybe I can get a point for originality.
Idea #1

- Virtual File System – VFS.
- Read and write any file using VFS functions.
- Using only KPI symbols.
- Recipe for success:
  - Vnode.
  - VFS context.
  - Data buffer.
  - UIO structure/buffer.
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.

```c
vnode_t kernel_node = NULLVP;
int error = vnode_lookup("/mach_kernel", 0, &kernel_vnode, NULL);
```
Apple takes care of the ctx for us!

```c
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! :-)  
    }
}
```
Data buffer.

- Statically allocated.
- Dynamically, using one of the many kernel functions:
  - kalloc, kmem_alloc, OSMalloc, IOMalloc, MALLOC, _MALLOC.
- ___LINKEDIT size is around 1Mb.
- UIO buffer.
  - Use uio_create and uio_addiov.
  - Both are available in BSD KPI.

```c
char buffer[PAGE_SIZE_64];
uio_t uio = NULL;
uio = uio_create(1, 0, UIO_SYSSPACE, UIO_READ);
int error = uio_addiov(uio, CAST_USER_ADDR_T(buffer), PAGE_SIZE_64);
```
- Recipe for success:
  - vnode of /mach_kernel.
  - VFS context.
  - Data buffer.
  - UIO structure/buffer.
- 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-O header and extract some information:
  - __TEXT segment address (to find KASLR).
  - __LINKEDIT segment offset and size.
  - Symbols and strings tables offset and size from LC_SYMTAB command.
- Read `__LINKEDIT` into a buffer (~1Mb).
- Process it and solve immediately all the 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 Mach-O magic value backwards.
- Compute the __TEXT address difference to the value we extracted from disk image.
- Or use some other method you might have.
Checkpoint #1

- We are able to read and write any file.
- For now the kernel is the interesting target.
- We can solve any available symbol - function or variable, exported or not in KPIs.
- Compatible with all OS X versions.
Simple Ideas

Problem #2

- Many interesting functions & variables are static.
- Cross references not available (IDA spoils us!).
- Hex search is not very reliable.
- Internal kernel structures fields offsets, such as proc and task.
Idea #2

- Integrate a disassembler in the rootkit!
- Tested with diStorm, my personal favorite.
- Works great.
- Be careful with some inline data.
- One second to disassemble the kernel.
- In a single straightforward sweep.
Simple Ideas

Checkpoint #2

- Ability to search for static functions, variables, and structure fields.
- We still depend on patterns.
- These are more common between all versions.
- Possibility to hook calls by searching references and modifying the offsets.
- We can have full control of the kernel.
- Everything can be dynamic.
- Stable and future proof rootkits.

```c
/* 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;
};

/* system call table */
/* OS X Mavericks */
struct sysent {
  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;
};
```
Simple Ideas

- Can Apple close the VFS door?
- That would probably break legit products that use them.
- We still have the disassembler(s).
- Kernel anti-disassembly? 😊
- Imagination is the limit!
Simple Ideas

Practical applications

- Executing userland code.
- Playing with DTrace’s syscall provider & Volatility.
- Zombie rootkits.
- Additional applications in the SyScan slides and Phrack paper (whenever it comes out).
It can be useful to execute userland binaries from the rootkit or inject code into them.

Many different possibilities exist:
- Modify binary (at disk or runtime).
- Inject shellcode.
- Inject a library.
- Etc...

This particular one uses last year’s Boubou trick.

Not the most efficient but fun.
Idea!

- Kill a process controlled by launchd.
- Intercept the respawn.
- Inject a dynamic library into its Mach-O header.
- Dyld will load the library, solve symbols and execute the library’s constructor.
- Do whatever we want!
Requirements

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

- Easiest solution is to use `vm_map_write_user`.
- `vm_map_write_user(vm_map_t map, void *src_p, vm_map_address_t dst_addr, vm_size_t size);
- "Copy out data from a kernel space into space in the destination map. The space must already exist in the destination map."
- Write to userland memory from kernel.
  - Map parameter is the map field from the task structure.
  - proc and task structures are linked via void *.
  - Use proc_find(int pid) to retrieve proc struct.
  - Or proc_task(proc_t p).
  - Check kern_proc.c from XNU source.
Write to userland memory from kernel.

- The remaining parameters are buffer to write from, destination address, and buffer size.

```c
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);
```
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.
- Vnode information can also be used.
execve() -> __mac_execve()

exec_activate_image()

Read file

exec_mach_imgact() -> run dyld -> target entry point

load_machfile()

parse_machfile()  [maps the load commands into memory]

load_dylinker()  [sets image entrypoint to dyld]

(...)

Userland cmds
There's a function called proc_resetregister.

Located near the end so almost everything is ready to pass control to dyld.

Easy to rip and hook!

Have a look at Hydra (github.com/gdbinit/hydra).

```c
void proc_resetregister(proc_t p)
{
    proc_lock(p);
    p->p_flag &= ~P_LREGISTER;
    proc_unlock(p);
}
```
☑ Modified Mach-O header.

- Very easy to do.
- Check last year’s HiTCON slides.
- OS.X/Boubou source code
Userland cmds

```
<-- Fix this struct
struct mach_header {
...
uint32_t nccmds;    // 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;
};
```
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 needs to read the Mach-O header.
- If header is modified before dyld’s control we can inject a library (or change entrypoint and so on).
A dynamic library.

- Use Xcode's template.
- Add a constructor.

```c
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!
Problems with this technique:

- Requires library at disk (can be unpacked from rootkit and removed if we want).
- Needs to kill a process (but can be used to infect specific processes when started).
- Proc structure is not stable (get fields offset using the disassembler).
OS X is “instrumentation” rich:
- DTrace.
- FSEvents.
- kauth.
- kdebug.
- TrustedBSD.
- Auditing.
- Socket filters.
Let's focus on DTrace's syscall provider.

Nemo presented DTrace rootkits at Infiltrate.

Siliconblade with Volatility "detects" them.

But Volatility is vulnerable to an old trick.
Hide & seek

- Traces every syscall entry and exit.
- mach_trap is the mach equivalent provider.
- DTrace's philosophy of zero probe effect when disabled.
- Activation of this provider is equivalent to sysent hooking.
- Modifies the sy_call pointer inside sysent struct.
Before:
gdb$ print *(struct sysent*)(0xffffffff8025255840+5*sizeof(struct sysent))
$12 = {
    sy_narg = 0x3,
    sy_resv = 0x0,
    sy_flags = 0x0,
    sy_call = 0xffffffff8024cfc210,           <- open syscall, sysent[5]
    sy_arg_munge32 = 0xffffffff8024fe34f0,
    sy_arg_munge64 = 0,
    sy_return_type = 0x1,
    sy_arg_bytes = 0xc
}

dtrace_systrace_syscall is located at address 0xffffffff8024fde360.

After enabling a 'syscall::open:entry' probe:
gdb$ print *(struct sysent*)(0xffffffff8025255840+5*sizeof(struct sysent))
$13 = {
    sy_narg = 0x3,
    sy_resv = 0x0,
    sy_flags = 0x0,
    sy_call = 0xffffffff8024fde360,           <- now points to dtrace_systrace_syscall
    sy_arg_munge32 = 0xffffffff8024fe34f0,
    sy_arg_munge64 = 0,
    sy_return_type = 0x1,
    sy_arg_bytes = 0xc
}
- Not very useful to detect sysent hooking.
- fbt provider is better for detection (check SyScan slides).
- Nemo’s DTrace rootkit uses syscall provider.
- Can be detected by dumping the sysent table and verifying if _dtrace_systrace_syscall is present.
- False positives? Low probability.
```bash
$ python vol.py mac_check_syscalls --profile=Mac10_8_3_64bitx64 \
-f ~/Forensics/dtrace/Mac\ OS\ X\ 10.8\ 64-bit-12e6095b.vmem
Volatile Systems Volatility Framework 2.3_alpha
Table Name | Index | Address                      | Symbol
-------------|-------|------------------------------|-----------------------------
SyscallTable  | 0     | 0xffffffff80085755f0         | _nosys                      |
SyscallTable  | 1     | 0xffffffff8008555430         | _exit                       |
SyscallTable  | 2     | 0xffffffff8008559730         | _fork                       |
SyscallTable  | 3     | 0xffffffff8008575630         | _read                       |
SyscallTable  | 4     | 0xffffffff8008575de0         | _write                      |
SyscallTable  | 5     | 0xffffffff80085db440         | _dtrace_systrace_syscall     | syscall::open:entry probe |
SyscallTable  | 6     | 0xffffffff8008549f30         | _close                      |
SyscallTable  | 7     | 0xffffffff8008556660         | _wait4                      |
SyscallTable  | 8     | 0xffffffff80085755f0         | _nosys                      |
SyscallTable  | 9     | 0xffffffff80082fbc20         | _link                       |
SyscallTable  | 10    | 0xffffffff80082fc8c0         | _unlink                     |
SyscallTable  | 11    | 0xffffffff80085755f0         | _nosys                      |
```
HINDSIGHT HEROES

Captain Hindsight
With his sidekicks, Shoulda, Coulda, and Woulda
"Nemo's presentation has shown again that known tools can be used for subverting a system and won't be easy to spot by a novice investigator, but then again nothing can hide in memory ;)"

@ http://siliconblade.blogspot.com/2013/04/hunting-d-trace-rootkits-with.html
It’s rather easy to find what you know.
How about what you **don’t** know?
Sysent hooking is easily detected by memory forensics (assuming you can get memory dump!).
But fails at old sysent shadowing trick.
$ python vol.py mac_check_syscalls --profile=Mac10.8.3_64bitx64 \ 
-f ~/Forensics/dtrace/Mac\ OS\ X\ 10.8\ 64-bit-no\ hooking.vmem
Volatile Systems Volatility Framework 2.3_alpha
(...)
SyscallTable 339 0xffffffff800854a490 _fstat64
SyscallTable 340 0xffffffff80082fd620 _lstat64
SyscallTable 341 0xffffffff80082fd420 _stat64_extended
SyscallTable 342 0xffffffff80082fd6c0 _lstat64_extended
SyscallTable 343 0xffffffff800854a470 _fstat64_extended
SyscallTable 344 0xffffffff8008300c20 _getdirententries64
SyscallTable 345 0xffffffff80082f9c60 _statfs64
SyscallTable 346 0xffffffff80082f9e80 _fstatfs64
SyscallTable 347 0xffffffff80082fa2a0 _getfsstat64
SyscallTable 348 0xffffffff80082fa7c0 __pthread_chdir
SyscallTable 349 0xffffffff80082fa640 __pthread_fchdir
SyscallTable 350 0xffffffff8008535cb0 _audit
SyscallTable 351 0xffffffff8008535e20 _auditon
(...)

No hooking! Not fun 😞
Hide & seek

$ python vol.py mac_check_syscalls --profile=Mac10.8.3_64bitx64 \\n-f ~/Forensics/dtrace/Mac\ OS\ X\ 10.8\ 64-bit-hooking1.vmem
Volatile Systems Volatility Framework 2.3_alpha
(...)
SyscallTable 339 0xffffffff800854a490 _fstat64
SyscallTable 340 0xffffffff80082fd620 _lstat64
SyscallTable 341 0xffffffff80082fd420 _stat64_extended
SyscallTable 342 0xffffffff80082fd6c0 _lstat64_extended
SyscallTable 343 0xffffffff800854a470 _fstat64_extended
SyscallTable 344 0xffffffff7f89a2dce0 HOOKED <- getdirententries64 hooked
SyscallTable 345 0xffffffff80082f9c60 _fstatfs64
SyscallTable 346 0xffffffff80082f9e80 _fstatfs64
SyscallTable 347 0xffffffff80082fa2a0 _getfsstat64
SyscallTable 348 0xffffffff80082fa7c0 __pthread_chdir
SyscallTable 349 0xffffffff80082fa640 __pthread_fchdir
SyscallTable 350 0xffffffff8008535cb0 _audit
SyscallTable 351 0xffffffff8008535e20 _auditon
(...)

Sysent hooking, meh!
Hide & seek

```
$ python vol.py mac_check_syscalls --profile=Mac10.8.3_64bitx64 \\n-f ~/Forensics/dtrace/Mac\ OS\ X\ 10.8\ 64-bit-hooking2.vmem
Volatile Systems Volatility Framework 2.3_alpha
(...)
SyscallTable  339 0xffffffff800854a490  _fstat64
SyscallTable  340 0xffffffff80082fd620  _lstat64
SyscallTable  341 0xffffffff80082fd420  _stat64_extended
SyscallTable  342 0xffffffff80082fd6c0  _lstat64_extended
SyscallTable  343 0xffffffff800854a470  _fstat64_extended
SyscallTable  344 0xffffffff8008300c20  _getdirententries64
SyscallTable  345 0xffffffff80082f9c60  _stat+sb4
SyscallTable  346 0xffffffff80082f9e80  _fstatfs64
SyscallTable  347 0xffffffff80082fa2a0  _getfsstat64
SyscallTable  348 0xffffffff80082fa7c0  ___pthread_chdir
SyscallTable  349 0xffffffff80082fa640  ___pthread_fchdir
SyscallTable  350 0xffffffff8008535cb0  _audit
SyscallTable  351 0xffffffff8008535e20  _auditon
(...)
```
Volatility plugin can easily find sysent table modification(s).

But fails to detect a shadow sysent table.

Nothing new, extremely easy to implement with the kernel disassembler!

Hindsight is always easy!
How to do it in a few steps:

- Find syent table address via IDT and bruteforce, or some other technique.

**Warning:** Mavericks has a modified syent table.

- Use the address to find location in __got section.
- Disassemble kernel and find references to __got address.
- Allocate memory and copy original sysent table.
- Find space inside kernel to add a pointer (modifying __got is too noisy!).
- Install pointer to our sysent copy.
- Modify found references to __got pointer to our new pointer.
- Hook syscalls in the shadow table.
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.
- Be very careful with tool’s assumptions.
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.
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.
- Find rootkit’s Mach-O header and compute its size (___TEXT + ___DATA segments).
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.
- 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.
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.
- 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.
☑ 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.
☑ 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.
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!
Demo

(Dear Spooks: all code will be made public, don’t break my room! #kthxbay)
mountain-lion-64:~ reverser$ uname -an
Darwin mountain-lion-64.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
mountain-lion-64:~ reverser$ ls /
mountain-lion-64:~ reverser$ sudo sh
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# ls /
.DS_Store .fseventsd System home sbin
.DocumentRevisions-V100 .hotfiles.btree Users mach_kernel_new tmp
.Spotlight-V100 .vol bin mach_kernel_old usr
.Trashes .Applications cores net var
.VolumeIcon.icns .Library dev opt
.file Network etc private
sh-3.2#
uilt Aug 21 2012 21:49:26
memctl: Opening balloon
memctl: Instrumenting bug 151304...
memctl: offset 0: 72
memctl: offset 1: 16
memctl: offset 2: 56
memctl: offset 3: 64
memctl: offset 4: 76
memctl: Timer thread started.
[AppleBluetoothHCIControllerUSBTransport][start] -- completed
[IOBluetoothHCIController][staticBluetoothHCIControllerTransportShowsUp] -- Received Bluetooth Controller register service notification
Sandbox: sandboxd(105) deny mach-lookup com.apple.coresymbolicationd
**** [AppleBluetoothHCIControllerUSBTransport][configurePM] -- ERROR -- waited 30 seconds and still did not get the commandWakeup() notification -- this = 0xffffffff8006cfe800 ****
Bluetooth: Adaptive Frequency Hopping is not supported.
[IOBluetoothHCIController::setConfigState] calling registerService
[SendHCIRequestFormatted] ### ERROR: [0x0C3F] (Set AFH Host Channel Classification) -- Send request failed (err = 0x0001 (kBluetoothHCIErrorUnknownHCICommand))
sh-3.2#
Unstable internal structures!

- Proc structure is one of those.
- We just need a few fields.
- Find offsets by disassembling stable functions.
- Possible, you just need to spend some time grep'ing around XNU source code and IDA.
Problems

- Memory forensics.
  - A worthy rootkit enemy.
  - But with its own flaws.
  - In particular the acquisition process.
  - Some assumptions are weak.
  - Needs more features.
Problems

- And so many others.
- It's a cat & mouse game.
- Any mistake can be costly.
- When creating a rootkit, reduce the number of assumptions you have.
- Defenders face the unknown.
- Very hard game – abuse their assumptions.
Conclusions

Rubes By Leigh Rubin

THIS IS, WITHOUT A DOUBT, THE BIGGEST PIECE OF *@#!! I HAVE EVER READ!

In his own mind, Jerry quickly mastered the art.
Conclusions

- Improving the quality of OS X kernel rootkits is very easy.
- Stable and future-proof requires more work.
- 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.
Conclusions

- Attackers have better incentives to be creative.
- Defense will always lag and suffer from information asymmetry.
- Economics didn’t solve this problem and I doubt InfoSec will (because it’s connected to Economics aka MONEY).
- Always question assumptions. This presentation has a few ;-).

Practice makes perfection!
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, A. Ionescu, Igor from Hex-Rays, NSA & friends, and you for spending time of your life listening to me 😊.

Greets
We are hiring!

- Software Engineers.
- Based in Singapore.
- 2 years experience.
- You know C and Python better than me!
- Can communicate in English.
- $80000NT monthly salary.
- Housing provided.
- 2 Years contract.
http://reverse.put.as
http://github.com/gdbinit
reverser@put.as
pedro@coseinc.com
@osxreverser
#osxre @ irc.freenode.net
And iloverootkits.com maybe soon!
A day full of possibilities!

Let’s go exploring!