Rootkits

Stuff

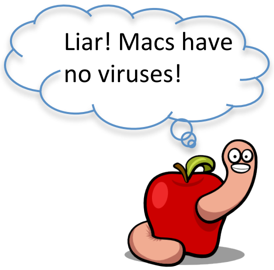
–[Revisiting Mac OS X Kernel Rootkits!]–

fG!

NoSuchCon, Paris

COSEINC®

Solid Security.Verified.



Who Am I

- Don't take me too seriously, I love the Human brain!
- The capitalist pig degrees: Economics & MBA.
- Worked for the evil banking system!
- Security Researcher at COSEINC (the PLA rumour so maybe I'm still on the evil side, damn!).
- Lousy coder.
- Co-authored a MISC article without speaking French!
- Passionate about 911s.

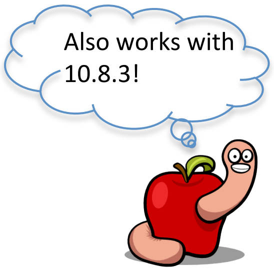
Today's subject

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

Assumptions

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

- Reaching to uid=0 is your problem!
- The same with startup and persistency aka APT.
- Probabilities should be favorable to you.
- Odays garage sale at SyScan '13 by Stefan Esser.
- You know how to create kernel extensions.
- Target is Mountain Lion 10.8.2, 64 bits.



Current state of the "art"

- Very few public developments since Leopard, besides EFI, and recently DTrace rootkits!
- Just lame Made in Italy rootkits (there goes the myth about Italian design!).
- Still, we must concede that they are "effective" and working in the "wild".
- The tools scene is even worse! No Such Tools...

Simple Ideas

Sophisticated!
Not simple.

Problem #1

- Many interesting kernel symbols are not exported.
- Some are available in Unsupported & Private KPIs.
- That's not good enough for stable rootkits.
- Solving kernel symbols from a kernel extension is possible in Lion and Mountain Lion.
- Not in Snow Leopard and previous versions.

Simple Ideas

- __LINKEDIT segment contains the symbol info.
- Zeroed up to Snow Leopard.
- Available in Lion and Mountain Lion.
- Not possible to have universal solution (Snow Leopard is still used by many people).
- OS.X/Crisis solves the symbols in userland and sends them to the kernel rootkit.

Simple Ideas

- One easy solution is to read the kernel image from disk and process its symbols.
- Some kind of "myth" that reading filesystem(s) from kernel is kind of hard to do.
- In fact it is very easy…
- Kernel ASLR is not a problem in this scenario.

Simple Ideas

Idea #1

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

Simple Ideas

- Let's explore the KPI symbols solution.
- Recipe for success:

- [ ] Vnode of mach_kernel.
- [ ] VFS context.
- [ ] Data buffer.
- [ ] UIO structure/buffer.

Simple Ideas

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

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

Simple Ideas

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

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

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

- vfs_context_current is available in Unsupported KPI.

Simple Ideas

- Alex Ionescu told me that this context might not be stable enough.
- If used very early in the boot process.
- You probably want to use the correct function.
- Or steal the context from somewhere else.

Simple Ideas

- Data buffer.
  - Statically allocated.
  - Or dynamically, using one of the many kernel functions:
  - kalloc, kmem_alloc, OSMalloc, IOMalloc, MALLOC, _MALLOC.

Simple Ideas

- UIO buffer.
  - Use uio_create and uio_addiov.
  - Both are available in BSD KPI.

```
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);
```

Simple Ideas

- Recipe for success:

☑ vnode of /mach_kernel.

☑ VFS context.

☑ Data buffer.

☑ UIO structure/buffer.

- Now we can finally read the kernel from disk...

Simple Ideas

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

Simple Ideas

- To solve the symbols we just need to read the Mach-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.

Simple Ideas

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

Simple Ideas

- To compute the KASLR value find out the base address of the running kernel.
- Using IDT or a kernel function address and then lookup 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 to any file.
- For now the kernel is the interesting target.
- We can solve any available symbol – function or variable, exported or not in KPIs.
- Compatible with all OS X versions.

Problem #2

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

Idea #2

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

Simple Ideas

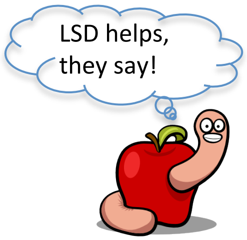
- The things you learn…
- There is already a disassembler in XNU kernel!
- DTrace has this function: dtrace_disx86.
- "Disassemble a single x86 or amd64 instruction."
- Unfortunately, strings output depends on DIS_TEXT, which is not active.
- Still, it's a fun thing to be found in the kernel.
- Thanks to espes for the tip ☺.

Checkpoint #2

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

Simple Ideas

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



Practical applications

- One way to execute userland code.
- Playing with DTrace's syscall provider.
- Zombie rootkits.
- Additional applications in the SyScan slides and Phrack paper (whenever it comes out).

Exec userland

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

Exec userland

Idea!

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

Requirements

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

Exec userland

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

Exec userland

- Write to userland memory from kernel.
  - vm_map_write_user(vm_map_t map, void *src_p, vm_map_address_t dst_addr, vm_size_t size);
  - 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.

Exec userland

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

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

Exec userland

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

Exec userland

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

Exec userland

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

```
execve() -> __mac_execve()
                 |
                 v
         exec_activate_image()
                 |
                 v
             Read file
                 |
                 v
  .----> exec_mach_imgact() -> run dyld -> target entry point
  |              |
  |              v
  |         load_machfile()
  |              |
  |              v
  |        parse_machfile()   [maps the load commands into memory]
  |              |
  |              v
  |         load_dylinker()   [sets image entrypoint to dyld]
  |              |
  |              v
  `--------- (...)
```

Exec userland

- Inside the "heart" there's a small function called proc_resetregister.
- Located near the end so almost everything is ready to pass control to dyld.
- Easy to rip and hook!
- Have a look at Hydra (github.com/gdbinit/hydra).

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

Exec userland

☑ Modified Mach-O header.

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

Exec userland

```
.---------------------.      .---------------------.
|       HEADER        |      |       HEADER        |<- Fix this struct
|---------------------|      |---------------------| struct mach_header {
|    Load Commands    |      |    Load Commands    |   ...
|   .-------------.   |      |   .-------------.   |   uint32_t  ncmds;      <- add +1
|   |  Command 1  |   |      |   |  Command 1  |   |   uint32_t  sizeofcmds; <- size of new cmd
|   |-------------|   |      |   |-------------|   |   ...
|   |  Command 2  |   |      |   |  Command 2  |   |   };
|   |-------------|   |      |   |-------------|   |
|   |     ...     |   |      |   |     ...     |   |
|   |-------------|   |      |   |-------------|   |
|   |  Command n  |   |      |   |  Command n  |   |
|   `-------------'   |      |   |-------------|   |
|                     |----->|   | Command n+1 |   |<- add new command here
|                     |----->|   `-------------'   | struct dylib_command {
|---------------------|----->|---------------------|   uint32_t        cmd;
|        Data         |----->|        Data         |   uint32_t        cmdsize;
|  .---------------.  |----->|  .---------------.  |   struct dylib    dylib;
|  |   | Section 1 |  |----->|  |   | Section 1 |  | };
|  | 1 |-----------|  |      |  | 1 |-----------|  |
|  |   | Section 2 |  |      |  |   | Section 2 |  |
|  `---------------'  |      |  `---------------'  |
|  .---------------.  |      |  .---------------.  |
|  |   | Section 1 |  |      |  |   | Section 1 |  |
|  | 2 |-----------|  |      |  | 2 |-----------|  |
|  |   | Section 2 |  |      |  |   | Section 2 |  |
|  `---------------'  |      |  `---------------'  |
|         ...         |      |         ...         |
`---------------------'      `---------------------'
```

Exec userland

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

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

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

Don't detect me

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

Don't detect me

- Let's focus on DTrace's syscall provider.
- Because nemo presented DTrace rootkits.
- Siliconblade with Volatility "detects" them.
- But Volatility is vulnerable to an old trick.

Don't detect me

- 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.

Don't detect me

```
Before:
gdb$ print *(struct sysent*)(0xffffff8025255840+5*sizeof(struct sysent))
$12 = {
  sy_narg = 0x3,
  sy_resv = 0x0,
  sy_flags = 0x0,
  sy_call = 0xffffff8024cfc210,          <- open syscall, sysent[5]
  sy_arg_munge32 = 0xffffff8024fe34f0,
  sy_arg_munge64 = 0,
  sy_return_type = 0x1,
  sy_arg_bytes = 0xc
}

dtrace_systrace_syscall is located at address 0xFFFFFF8024FDC630.

After enabling a 'syscall::open:entry' probe:
gdb$ print *(struct sysent*)(0xffffff8025255840+5*sizeof(struct sysent))
$13 = {
  sy_narg = 0x3,
  sy_resv = 0x0,
  sy_flags = 0x0,
  sy_call = 0xffffff8024fdc630,       <- now points to dtrace_systrace_syscall
  sy_arg_munge32 = 0xffffff8024fe34f0,
  sy_arg_munge64 = 0,
  sy_return_type = 0x1,
  sy_arg_bytes = 0xc
}
```

Don't detect me

- Not very useful provider to detect sysent hooking.
- DTrace doesn't care about original pointer.
- fbt provider is better for this task.
- Nemo's DTrace public rootkit uses this provider ;-).
- Can be detected by dumping the sysent table and verifying if _dtrace_systrace_syscall is present.
- Probability of false positives, although small?

Don't detect me

```
$ 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 0xffffff80085755f0 _nosys
SyscallTable         1 0xffffff8008555430 _exit
SyscallTable         2 0xffffff8008559730 _fork
SyscallTable         3 0xffffff8008575630 _read
SyscallTable         4 0xffffff8008575d00 _write
SyscallTable         5 0xffffff80085db440 _dtrace_systrace_syscall  <- syscall::open:entry probe
SyscallTable         6 0xffffff8008549f30 _close
SyscallTable         7 0xffffff8008556660 _wait4
SyscallTable         8 0xffffff80085755f0 _nosys
SyscallTable         9 0xffffff80082fbc20 _link
SyscallTable        10 0xffffff80082fc8c0 _unlink
SyscallTable        11 0xffffff80085755f0 _nosys
```

Don't detect me

- My goal is not to mock anyone, just fooling around!
- Famous last words:
- "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

Don't detect me

Don't detect me

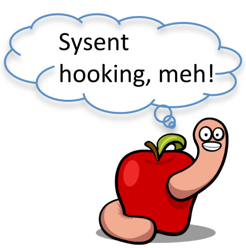
- It's rather easy to find what you know.
- How about what you don't know?
- syscall provider doesn't care about sysent hooking.
- But that is easily detected by memory forensics.
- What happens if we modify all the kernel references to sysent?
- AKA really old school sysent shadowing…

Don't detect me

```
$ 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 0xffffff800854a490 _fstat64
SyscallTable        340 0xffffff80082fd620 _lstat64
SyscallTable        341 0xffffff80082fd420 _stat64_extended
SyscallTable        342 0xffffff80082fd6c0 _lstat64_extended
SyscallTable        343 0xffffff800854a470 _fstat64_extended
SyscallTable        344 0xffffff8008300c20 _getdirentries64
SyscallTable        345 0xffffff80082f9c60 _statfs64
SyscallTable        346 0xffffff80082f9e80 _fstatfs64
SyscallTable        347 0xffffff80082fa2a0 _getfsstat64
SyscallTable        348 0xffffff80082fa7c0 ___pthread_chdir
SyscallTable        349 0xffffff80082fa640 ___pthread_fchdir
SyscallTable        350 0xffffff8008535cb0 _audit
SyscallTable        351 0xffffff8008535e20 _auditon
(...)
```

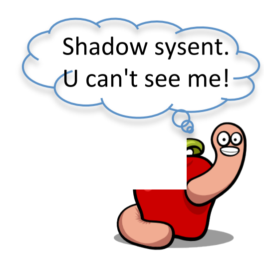
Don't detect me

```
$ python vol.py mac_check_syscalls --profile=Mac10_8_3_64bitx64 \
-f ~/Forensics/dtrace/Mac\ OS\ X\ 10.8\ 64-bit-hooking1.vmem
Volatile Systems Volatility Framework 2.3_alpha
(...)
SyscallTable        339 0xffffff800854a490 _fstat64
SyscallTable        340 0xffffff80082fd620 _lstat64
SyscallTable        341 0xffffff80082fd420 _stat64_extended
SyscallTable        342 0xffffff80082fd6c0 _lstat64_extended
SyscallTable        343 0xffffff800854a470 _fstat64_extended
SyscallTable        344 0xffffff7f89a2dce0 HOOKED  <- getdirentries64 hooked
SyscallTable        345 0xffffff80082f9c60 _statfs64
SyscallTable        346 0xffffff80082f9e80 _fstatfs64
SyscallTable        347 0xffffff80082fa2a0 _getfsstat64
SyscallTable        348 0xffffff80082fa7c0 ___pthread_chdir
SyscallTable        349 0xffffff80082fa640 ___pthread_fchdir
SyscallTable        350 0xffffff8008535cb0 _audit
SyscallTable        351 0xffffff8008535e20 _auditon
(...)
```



Don't detect me

```
$ python vol.py mac_check_syscalls --profile=Mac10_8_3_64bitx64 \
-f ~/Forensics/dtrace/Mac\ OS\ X\ 10.8\ 64-bit-hooking2.vmem
Volatile Systems Volatility Framework 2.3_alpha
(...)
SyscallTable        339 0xffffff800854a490 _fstat64
SyscallTable        340 0xffffff80082fd620 _lstat64
SyscallTable        341 0xffffff80082fd420 _stat64_extended
SyscallTable        342 0xffffff80082fd6c0 _lstat64_extended
SyscallTable        343 0xffffff800854a470 _fstat64_extended
SyscallTable        344 0xffffff8008300c20 _getdirentries64
SyscallTable        345 0xffffff80082f9c60 _statfs64
SyscallTable        346 0xffffff80082f9e80 _fstatfs64
SyscallTable        347 0xffffff80082fa2a0 _getfsstat64
SyscallTable        348 0xffffff80082fa7c0 ___pthread_chdir
SyscallTable        349 0xffffff80082fa640 ___pthread_fchdir
SyscallTable        350 0xffffff8008535cb0 _audit
SyscallTable        351 0xffffff8008535e20 _auditon
(...)
```



Don't detect me

- Volatility plugin can easily find sysent table modification(s).
- But fails to detect a simple shadow sysent table.
- Nothing new, extremely easy to implement with the kernel disassembler!
- Hindsight is always easy!
- Beware with the confidence levels you get from it.

Checkpoint

- Many instrumentation features available!
- Do not forget them if you are the evil rootkit coder.
- Helpful for a quick assessment if you are the potential victim.
- Friend or foe, use them!
- But don't trust too much in the tools ☺.

Zombies

Zombies

Idea!

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

Zombies

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

Zombies

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

Zombies

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

Zombies

☑ Fix symbols offsets.

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

Zombies

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

Zombies

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

Zombies

☑ Return KERN_FAILURE.

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

Advantages

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

Zombies

Demo

(Dear Spooks: you don't need to break in my room or computer, sample code will be made public! #kthxbay)

Marketing

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

Problems

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

Problems

❑ Memory forensics

- The "new" rootkit enemy.
- But with its own flaws.
- In particular the acquisition process.
- Which we can have a chance to play with.
- 29C3 had a presentation about Windows.
- Research-in-progress...

Problems

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

Conclusions

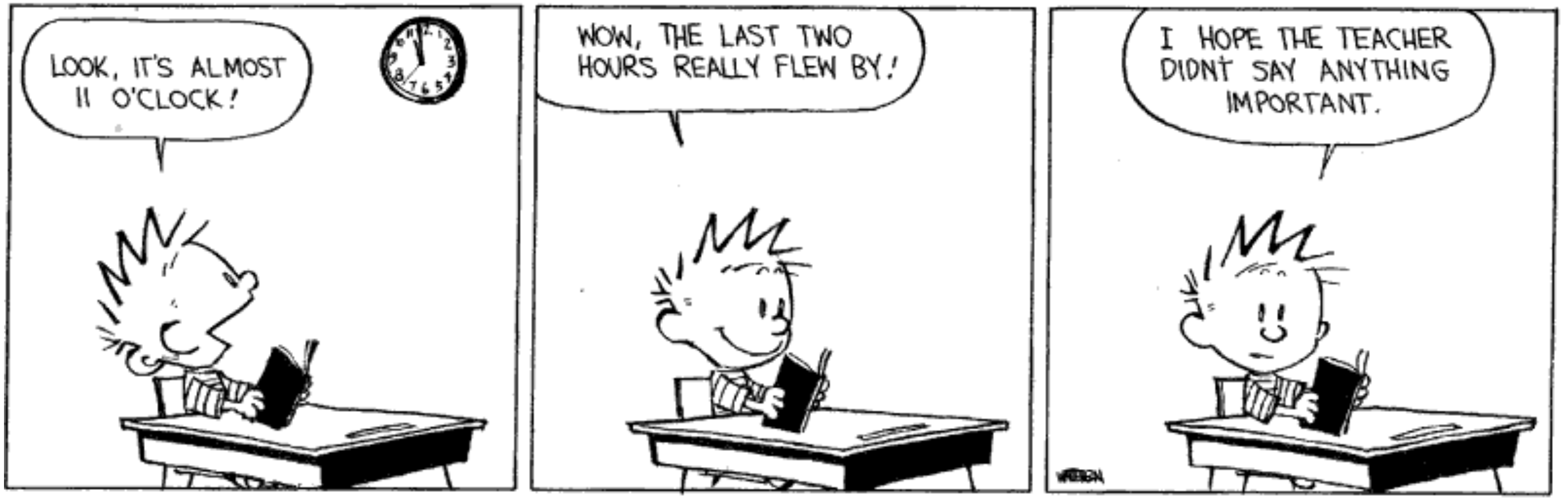
- Improving the quality of OS X kernel rootkits is very easy.
- Prevention and detection tools must be researched & developed.
- Kernel is sexy but don't forget userland.
- OS.X/Crisis userland rootkit is powerful!
- Easier to hide in userland from memory forensics.
- Read the paper, if you haven't already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

Conclusions

- WE don't know sh*t about OS X malware/rootkits.
- (AV) industry is generally lagging.
- Attackers have better incentives to be creative.
- Defense is very hard – information asymmetry.
- In particular because it's very easy to stick to a certain paradigm and hard to get out of it.
- That requires a lot of practice!

Greets

nemo, noar, snare, saure, od, emptydir, korn, g0sh, spico and all other put.as friends, everyone at COSEINC, thegrugq, diff-t, #osxre, Gil Dabah from diStorm, A. Ionescu, Igor from Hex-Rays, Shane (my assigned drone controller), and you for spending time of your life listening to me ☺.



Contacts

http://reverse.put.as

http://github.com/gdbinit

reverser@put.as

@osxreverser

#osxre @ irc.freenode.net


IF YOU COULD WISH FOR ANYTHING, WHAT WOULD IT BE?
A BIG SUNNY FIELD TO BE IN.
© 1986 Universal Press Syndicate
4·17
A STUPID FIELD?! YOU'VE GOT THAT NOW! THINK BIG! RICHES! POWER! PRETEND YOU COULD HAVE ANYTHING!
ACTUALLY, IT'S HARD TO ARGUE WITH SOMEONE WHO LOOKS SO HAPPY.
Z


Have fun!