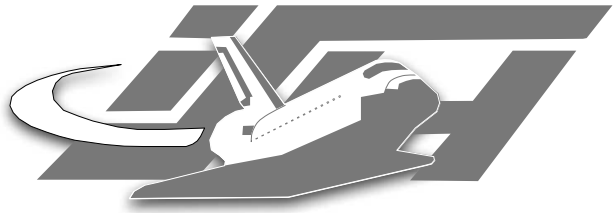
Anti-Honeypot Technology

Thorsten Holz

Laboratory for Dependable Distributed Systems

holz@i4.informatik.rwth-aachen.de



RWTH AACHEN

Overview

1. Brief introduction to honeypot technology
2. NoSEBrEaK
   - Workings of Sebek
   - Detecting & disabling Sebek
   - Kebes
   - Other anti-Sebek techniques
3. Detecting other honeypot architectures
   - VMware-based honeypots
   - UML-based honeypots
   - Others

Who we are

- Laboratory for Dependable Distributed Systems at RWTH Aachen University
- Main interests:
  - Theoretical considerations of security (safety / liveness / information flow properties, theoretical models of secure systems)
  - Threats in communication networks (honeypots, . . .)
  - Trusted Computing
- Summer School “Applied IT-security”
- “Hacker lab” & “Hacker seminar”

`http://www-i4.informatik.rwth-aachen.de/lufg`

Honeypot Technology

"Suppose," he [Winnie the Pooh] said to Piglet, "you wanted to catch me, how would you do it?"
"Well," said Piglet, "I should do it like this: I should make a trap, and I should put a jar of honey in the trap, and you would smell it, and you would go in after it, and . . . "

A. A. Milne: Winnie the Pooh

Honeypots?

- Electronic bait, i.e. network resources (e.g. computers, routers, switches, . . .) deployed to be probed, attacked and compromised
- "Learn the tools, tactics, and motives of the blackhat community and share these lessons learned"
- Monitoring software permanently collects data, helps in post-incident forensics
- Clifford Stoll: *The Cuckoo's Egg*, 1988
- **Honeynet Project**: Non-profit research organization of security professionals dedicated to information security

Global Honeynet Project

The Honeynet PROJECT™

- Development of tools, for example monitoring software like *Sebek* or software for data analysis
- Experiences up to now:
  - Capturing of exploits and tools, e.g. exploit for known vulnerability (`dtspcd`, 2002)
  - Typical approach of attackers
  - Monitoring of conversations over IRC Botnets, organized card fraud, . . .

Further information: **honeynet.org**

Building Blocks: Sebek

- Kernel-module on Linux & Solaris, patch on OpenBSD / NetBSD, device driver for Window$
- Tries to capture all activities of an attacker
- Hijacks `sys_read` (access to SSH sessions, `burneye`-protected programs, . . .)
- Direct communication to ethernet driver, therefore mostly stealth
- Unlinking from module list to hide its presence

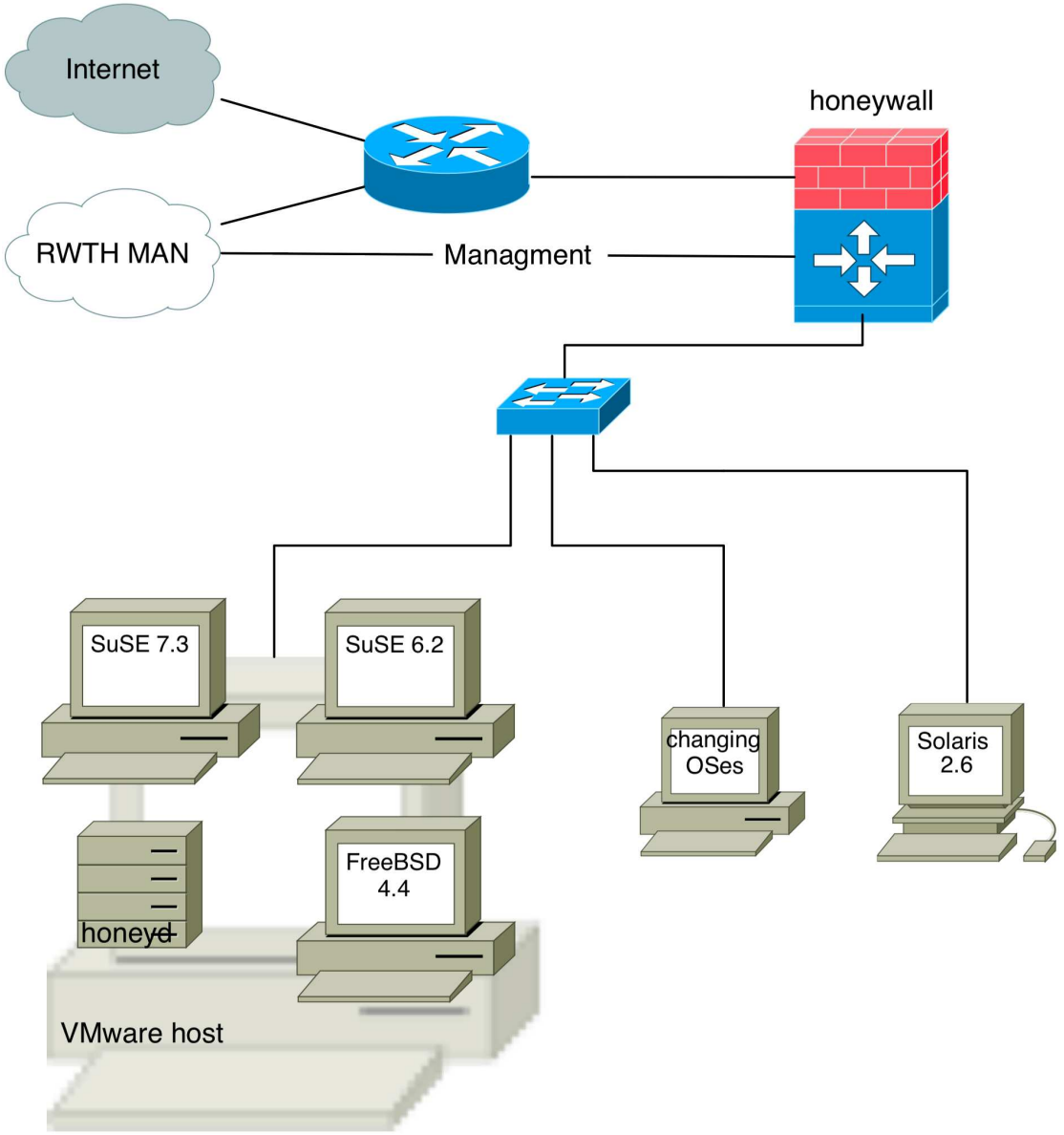
Building Blocks: Honeywall

- **Transparent bridge, used for data capture and data control**
- **IDS `snort` / IPS `snort_inline` (now part of `snort`)**

```
alert ip $HONEYNET any -> $EXTERNAL_NET any
(msg:"SHELLCODE x86 stealth NOOP"; rev:6; sid:651;
    content:"|EB 02 EB 02 EB 02|";
    replace:"|24 00 99 DE 6C 3E|";)
```

- **`netfilter/iptables` for traffic limiting**
- **Further monitoring**
  - **`monit` or `supervise`**
  - **`swatch`**

Setup at German Honeynet Project



Official website

NoSEBrEaK

NoSEBrEaK

- *We had no attacks on our honeynet, so . . .*
- Toolkit written in Python 2.3 to detect and remove Sebek from honeypot
- Work together with Maximillian Dornseif and Christian N. Klein
- Presented as academic paper at 5th IEEE Information Assurance Workshop, Westpoint Available at arXiv as **cs.CR/0406052**
- Get the source code at **md.hudora.de**
- Now: Short presentation of our results

Sebek

> [...] monitoring capability to all activity on the honeypot including, but not limited to, keystrokes. If a file is copied to the honeypot, Sebek will see and record the file, producing an identical copy. If the intruder fires up an IRC or mail client, Sebek will see those messages. [...] Sebek also provides the ability to monitor the internal workings of the honeypot in a glass-box manner, as compared to the previous black-box techniques. [...] intruders can detect and disable Sebek. Fortunately, by the time Sebek has been disabled, the code associated with the technique and a record of the disabling action has been sent to the collection server.

Know Your Enemy: Sebek

Workings of Sebek *in short*

Concentrate on Sebek version 2.1.7 for Linux, techniques are applicable for other Sebek versions

Basic mechanism of Sebek and interesting points for attack:

- **Hijack `sys_read()`**
- **Send data passing through `sys_read()` in covert manner over the network**
- **Overwrites part of the network stack (`packet_recvmsg`) to hide Sebek data passing on to the network**

Hiding of Sebek

- Sebek loads as a kernel module
- Afterwards `cleaner.o` (part of `adore`) is loaded which removes Sebek from modules list

From `cleaner.o`

```
if (__this_module.next)
    __this_module.next = __this_module.next->next;
```

This works because kernel maintains list of modules (`sys_create_module()`)

```
spin_lock_irqsave(&modlist_lock, flags);
mod->next = module_list;
module_list = mod;      /* link it in */
spin_unlock_irqrestore(&modlist_lock, flags);
```

Detecting Sebek

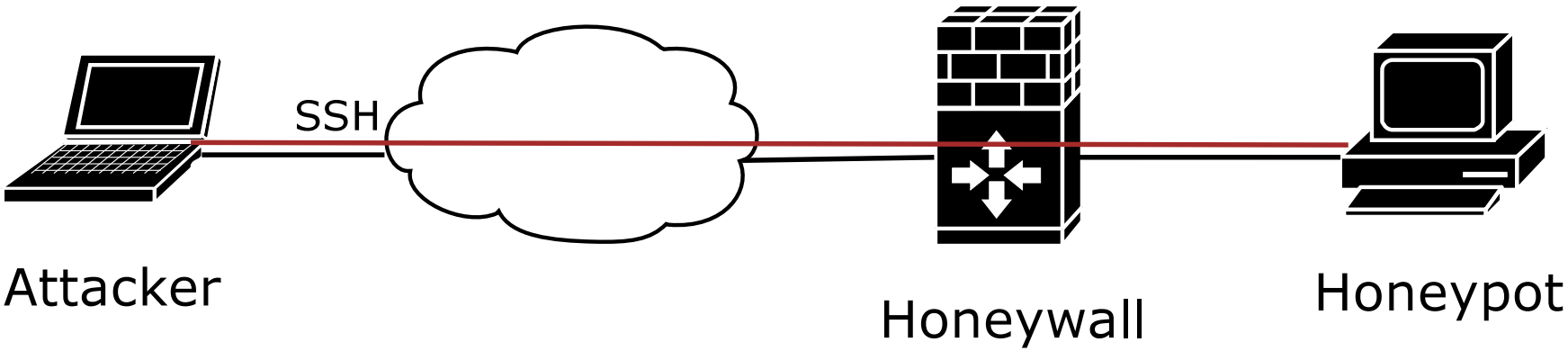
Several ways to detect Sebek come to mind:

- Latency
- Network traffic counters
- Modification of sys-call table
- Finding hidden module
- Other cruft in memory

Setup

Setup in movies:

- Attacker compromised one of the honeypots
- SSH-connection from attacker to honeypot (≈ 1 KB/s data)
- Movies show view of an attacker



Latency

First detection method we found during tests:

"*dd-attack*"

```
$ dd if=/dev/zero of=/dev/null bs=1
```

Just call `sys_read()` a couple of thousand times per second...

Movie: `dd.mov`

Network Traffic Counters

- **dd-attack / backward running counters**
  - ***Issue solved in Sebek 2.1.7, changed packet counter manipulation technique (take a look at `sprintf_stats`)***
- **`dev->get_stats->tx_bytes` or `dev->get_stats->tx_packets` vs. `/proc/net/dev` or `ifconfig` output**

Movie: `devchecker.mov`

4 GB traffic in 4 minutes?

RedTeam@RWTH

```
vampire:~/NoSEBrEaK/kebes# ifconfig  eth0
eth0      Link encap:Ethernet  HWaddr 00:02:3F:74:5E:3D
          inet addr:10.11.12.2  Bcast:10.255.255.255  Mask:255.0.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:254 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4294967295 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:25860 (25.2 KiB)  TX bytes:4294966268 (3.9 GiB)
          Interrupt:10 Base address:0x5800

vampire:~/NoSEBrEaK/kebes# uptime
 21:16:36 up 4 min,  2 users,  load average: 0.02, 0.07, 0.03
vampire:~/NoSEBrEaK/kebes#
```

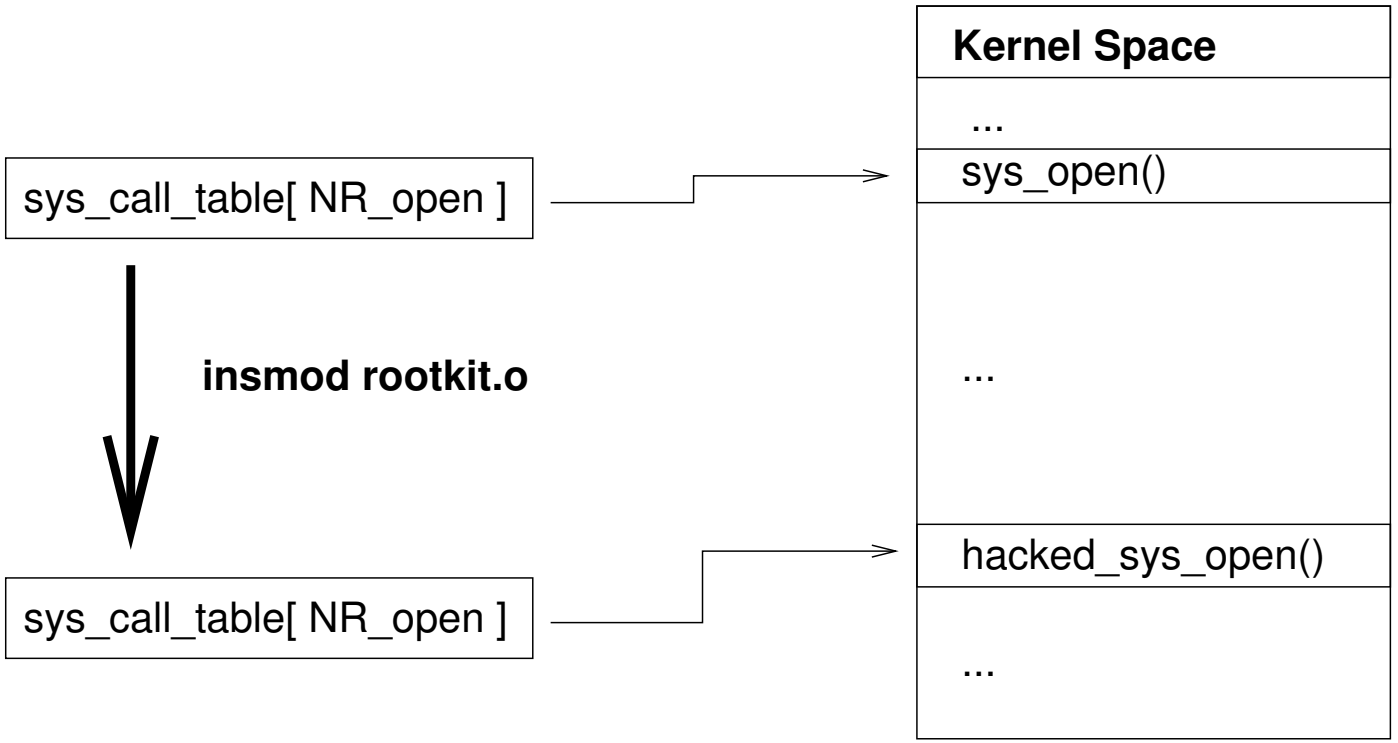
Excursus: System Calls

- User-land vs. kernel-land:
  - Upon `read()` in usermode, push parameter in register, call `0x80`
  - In kernelmode, search in Interrupt Descriptor Table (IDT) for interrupt handler
  - According to sys-call table, interrupt handler calls `sys_read()`
- Defined in `/usr/src/linux/include/asm/unistd.h`

```
#define __NR_exit                  1
#define __NR_fork                  2
#define __NR_read                  3
```

Excursus: Modifying it

- Sys-call-table stores pointers to function
- Modify these to control behaviour of sys-calls



- Some Linux 2.4 versions export it:

```
extern int sys_call_table[];
```

Excursus: Finding it

```
for (ptr = (unsigned long)&loops_per_jiffy;
     ptr < (unsigned long)&boot_cpu_data; ptr += sizeof(void *)){

  unsigned long *p;
  p = (unsigned long *)ptr;
  if (p[__NR_close] == (unsigned long) sys_close){
    sct = (unsigned long **)p;
    break;
  }
}

if (sct) {
  (unsigned long *) ord = sct[__NR_read];
  sct[__NR_read] = (unsigned long *) hacked_read;
}
```

Should work with recent 2.4.XX and 2.6.X kernels [1]

Modification of Sys-call Table

- Sebek modifies in current version `sys_read()`
- Modification can easily be detected – just take a look at the memory

- Before loading Sebek:

```
sys_read  = 0xc0132ecc
sys_write = 0xc0132fc8
```

- Afterwards:

```
sys_read  = 0xc884e748
sys_write = 0xc0132fc8
```

Detecting Sebek

Several ways to detect Sebek come to mind:

- Latency
- Network traffic counters
- Modification of sys-call table
- **Finding hidden module**
- **Other cruft in memory**

Interesting things in /usr/include/linux/module.h Kernel 2.4.X

```
struct module {
    unsigned long size_of_struct; /* == sizeof(module) *
    struct module *next;          // Pointer into kernel
    const char *name;             // Pointer into kernel

    struct module_symbol *syms; // Pointer into kernel
    struct module_ref *deps;    // Pointer into kernel
    struct module_ref *refs;    // Pointer into kernel
    int (*init)(void);          // Pointer into module
    void (*cleanup)(void);      // Pointer into module
}
```

(Note: Kernel 2.6 has different module.h)

Variables with only small range of “reasonable” values:

```
struct module {
    unsigned long size;

    union {
        atomic_t usecount;
        long pad;
    } uc;

    unsigned long flags;

    unsigned nsyms;
    unsigned ndeps;
}
```

Finding Modules

- Module header is allocated by kernel's `vmalloc`
- Function `vmalloc` aligns memory to page boundaries (4096 bytes on IA32)
- Memory allocated by `vmalloc` starts at `VMALLOC_START` and ends `VMALLOC_RESERVE` bytes later

```
for (p = VMALLOC_START;
     p <= VMALLOC_START + VMALLOC_RESERVE - PAGE_SIZE;
     p =+ PAGE_SIZE)
```

phrack issue 0x3d, phile #0x03 – `module_hunter.c`

Movie: `module_hunter.mov`

Retrieving Sebek's Variables

00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

- **Initial memory layout**

Retrieving Sebek's Variables

00000000	00000000	PORT	00000000	00000000	00000000	00000000	MAC5
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	MAC2	00000000	MAC1	00000000
00000000	MAGIC	00000000	00000000	00000000	00000000	00000000	00000000
MAC4	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	MAC0	00000000	00000000	00000000
00000000	MAC3	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	IP	00000000

- **Random positions of parameters** (`gen_fudge.pl`)

Retrieving Sebek's Variables

00000000	00000000	00007a69	00000000	00000000	00000000	00000000	000000d9
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	000000dc	00000000	0000000d	00000000
00000000	f001c0de	00000000	00000000	00000000	00000000	00000000	00000000
000000e5	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	0000003a	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	d5495b1d	00000000

- **Memory layout after random insertion of parameters**

Retrieving Sebek's Variables

00000000	00000000	00007a69	00000000	00000000	00000000	00000000	000000d9
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	000000dc	00000000	0000000d	00000000
00000000	f001c0de	00000000	00000000	00000000	00000000	00000000	00000000
000000e5	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	0000003a	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	d5495b1d	00000000

f001c0de = 240.1.192.222 (reserved address space)

- **Probably not the IP address**
- **But probably the magic number?**

Retrieving Sebek's Variables

00000000	00000000	00007a69	00000000	00000000	00000000	00000000	000000d9
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	000000dc	00000000	0000000d	00000000
00000000	f001c0de	00000000	00000000	00000000	00000000	00000000	00000000
000000e5	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	0000003a	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	d5495b1d	00000000

d5495b1d = 213.73.91.29

- Probably not the magic number
- But probably the IP address!

Retrieving Sebek's Variables

00000000	00000000	00007a69	00000000	00000000	00000000	00000000	000000d9
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	000000dc	00000000	0000000d	00000000
00000000	f001c0de	00000000	00000000	00000000	00000000	00000000	00000000
000000e5	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	0000003a	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	d5495b1d	00000000

00007a69 = 31337

- Is this perhaps the port number? Or magic?
- And are the other numbers part of the MAC address?

Movie: `NoSEBrEaKer.mov`

Disabling Sebek

- The easy way: Call `cleanup()`
  `kerneljumper.o` – jump to arbitrary memory location and execute code
- The obvious way: Reconstruct `sys_read()` pointer from the kernel and fix it in sys-call table
  Saved inside memory, so just patch memory
- The crazy way: Patch in your own, untainted version of `sys_read()`
  Untested, but should work

What can be logged?

- Unconditionally obtained by operator of honeypot
  - All network traffic ($\Rightarrow$ use encrypted communication / attack logging host (hard!))
  - All calls to `read()` ($\Rightarrow$ avoid `read()`)
- Possibly obtained after break-in
  - Forensic data obtained by disk analysis ($\Rightarrow$ keep most things in memory only)
  - Syslog-data ($\Rightarrow$ avoid it as best as possible)

Intercepting `read()`

- What kind of programs use `read()`?
  - Almost every interactive program uses `read(1)`
  - Many programs use `read()` for reading configuration files etc.
  - Network programs usually use `recv()` instead of `read()`
- Making `read()` unreliable
  - Read in as much data as possible

$\Rightarrow$ dd-attack (*not reliable, no control*)

Living without `read()`

- Surprisingly it is possible to avoid `read()` in many cases
- Use `mmap()` instead :-)
  - It is very hard to intercept
  - Drawback: It works only on regular files
  - Things you can not access:
    - `/dev/random` (useful for getting random seed for crypto stuff)
    - pipes (useful for communication)
    - All devices

Better living without `read()`

- Talk directly to network, execute commands without calling other programs wherever possible
- Nice bonus: `exec()` does not call `read()` (but importing libraries may do so...)

Other stuff

- Messing with the process name – just copy & rename the binary
- Name of the command calling `read()` is logged (max 12 bytes) – we can play with it
- Since filenames are not logged, we can give impression of reading certain files (makes forensic harder)

Kebes

- Proof of concept code
- Entirely written in Python 2.3 for portability with no external dependency
- Can do everything you can expect from a basic shell
- Highly dynamic, leaves not much traces at honeypot

Kebes : Networking

- Uses TCP-sockets for networking but could also be adopted to use `stdin`/`stdout` or anything else
- On top of that implements a crypto layer based on Diffie-Hellman / AES providing compression and random length padding
- Main problem: Getting entropy for DH
  - Use race-conditions and similar things to get entropy
- Python-specific “Kebes layer” using serialized objects to transfer commands and results back and forth

Kebes : “Kebes layer”

- Can work asynchronous and send multiple commands at once
  - *Asynchronous commands not implemented by the server at this time*
- Commands can usually work on several objects on the server at once
- Highly dynamic: Kebes layer initially knows only a single command; ADDCOMMAND

Kebes : “Kebes layer”

- Code for all additional commands is pushed by client into server at runtime as serialized Python objects

⇒ So most of NoSEBrEaK-code will only exist in the server’s RAM – makes forensic harder

- Implemented commands: Reading / writing files, secure deletion, direct execution, listing directories, . . .

Securing Sebek for Linux

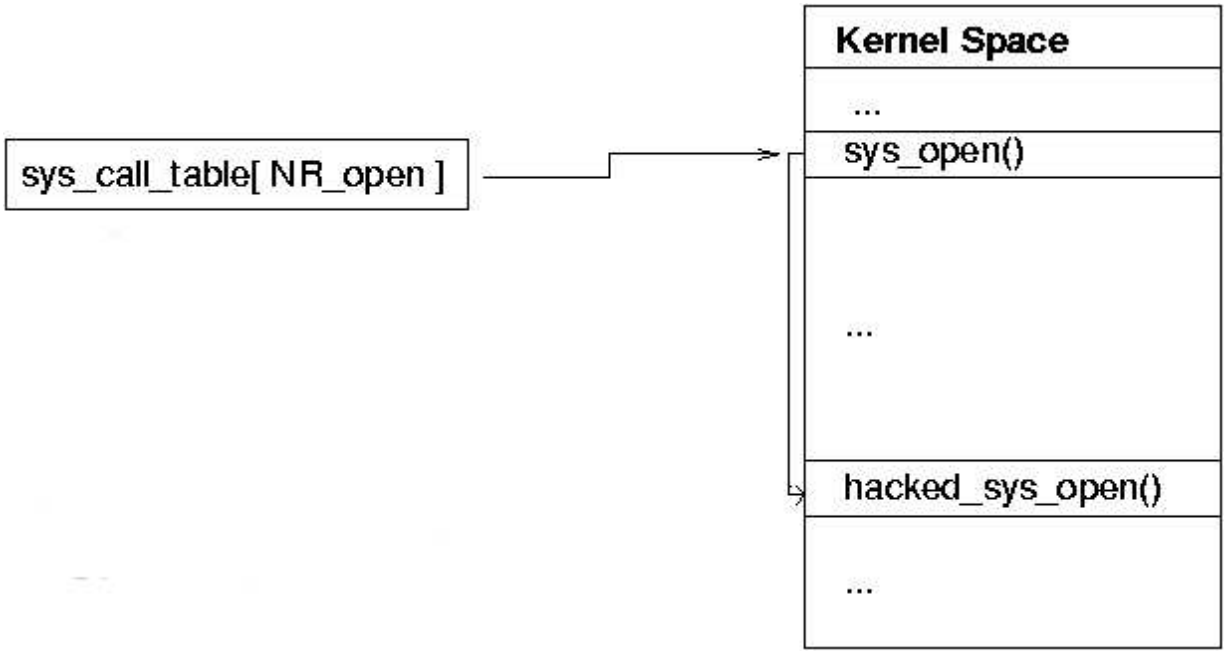
- Filling memory block with random data and not zeroing out everything
- Disable unloading of Sebek LKM via capabilities
- Rate limiting / threshold
- Filter expression to exclude things to log
- Presumably best solution: Kernel patch (currently in preparation, contact me if you want to help)

Anti-Sebek Techniques for Win32

- Similar techniques are also possible for Window$ version of Sebek:
  - Traverse PsLoadedModuleList (similar to module list in Linux)
  - Watch out for hooked APIs (similar to changed memory locations in sys-call table)
  - Disable Sebek through restoring of SDT ServiceTable (similar to reconstruction of sys-call table in Linux)
- Work by Tan Chew Keong ([1], [2])

Anti-Sebek Techniques for *BSD

- NetBSD LKM version of Sebek uses technique proposed by **Silvio Cesare**
  - Do not modify sys-call table directly
  - Instead, add `JUMP` (`0xE9`) at beginning of code and trojan `sys_read` in this way



Anti-Sebek Techniques for *BSD

- NetBSD LKM version of Sebek uses technique proposed by **Silvio Cesare**
- Also easy to detect:

  `unsebek-freebsd-lkm`

- OpenBSD version also detectable via `bpf` fingerprinting or searching through kernel file (disassemble `dofileread`)
- Article available by **Droids Corporation**

Detecting Other Honeypot Architectures

UML-based Honeypots

Easy to identify that you are within UML:

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : User Mode Linux
model name      : UML
mode            : tt
host            : Linux vampire 2.4.23 #1 Fri Dec 19 01:
bogomips        : 725.81
```

```
$ cat /proc/devices
[...]
Block devices:
[...]
 60 cow
 98 ubd
```

```
$ cat /proc/filesystems
[...]
nodev   hostfs
```

UML-based Honeypots

- Many other info about UML within `/proc`
  - `iomem`, `ioports`, `interrupts`, ... look suspicious
- If `hostfs` is available, you can have lots of fun

```
# mount -t hostfs /dev/hda1 /mnt/
```

⇒ You can access everything the user running UML can access

- For example: Retrieve information about processes

```
$ find /mnt/proc -name exe |xargs ls -l
```

UML-based Honeypots


```
$ egrep -i "uml|honey" ksyms
a02eb408 uml_physmem
a02ed688 honeypot
```

- `/proc` filesystem can be faked via `hppfs`
- But several other ways to fingerprint UML exist:
  - `/dev/cow`
  - Searching through `/dev/mem` (Requires CAP_SYS_RAWIO capability)
  - Info in `/var/log/messages`
  - In `tt`-mode UML kernel is present in address space of each of its processes

Securing UML

- Use `chroot`
- Directory (non-writeable) only contains UML binary and filesystem
- Run UML as user “nobody”
- UML binary non-writeable and immutable
- Filesystem non-executable
- `chown` everything to another user
- Use `skas`-mode (UML kernel runs in an entirely different host address space from its processes)

VMware-based Honeypots

- Characteristic fingerprints for VMware-based honeypots:
  - MAC address of NIC
  - Names of IDE & SCSI devices (HD & CDROM)
  - PCI vendor string and device ID of video adapter
  - I/O backdoor
  - `dmesg`
- Patch by Kostya Kortchinsky from FHP available

“Red Pill” by Joanna Rutkowska


```
int swallow_redpill () {
    unsigned char m[2+4],
        rpill[] = "\x0f\x01\x0d\x00\x00\x00\x00\xc3";
    *((unsigned*)&rpill[3]) = (unsigned)m;
    ((void(*)())&rpill)();
    return (m[5]>0xd0) ? 1 : 0;
}
```

- Get contents of the interrupt descriptor table register (IDTR)
- SIDT instruction (encoded as 0F010D[addr])
- Can be used in user-mode, but returns sensitive register
- On VMWare, relocated address of IDT is e.g. at `0xffXXXXXX`

Further things

- "Defeating Honeypots: Network Issues", written by Laurent Oudot and me, available at **securityfocus**
- "Defeating Honeypots: System Issues" currently in preparation, should be publised in January
- **PacSec.jp / core04** conference: Laurent Oudot – "Countering Attack Deception Techniques"

Further Questions?

- Thanks for your attention!
- Further information can be found on the links provided in the slides
- Greetings to Maximillian Dornseif, Christian N. Klein, Felix Gärtner, Laurent Oudot, the Droids, Joanna Rutkowska, Lutz Böhne, ...
- Mail: `holz@i4.informatik.rwth-aachen.de`