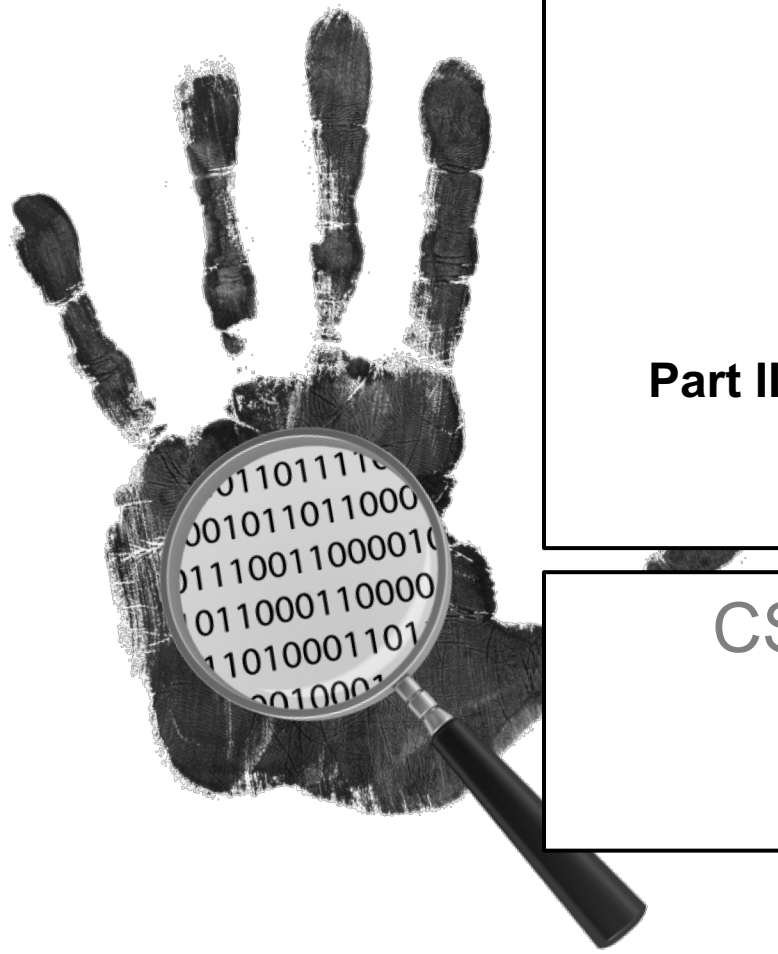
TÉCNICO LISBOA

Rootkits and Malware Analysis

Part III. Advanced Techniques and Tools for Digital Forensics



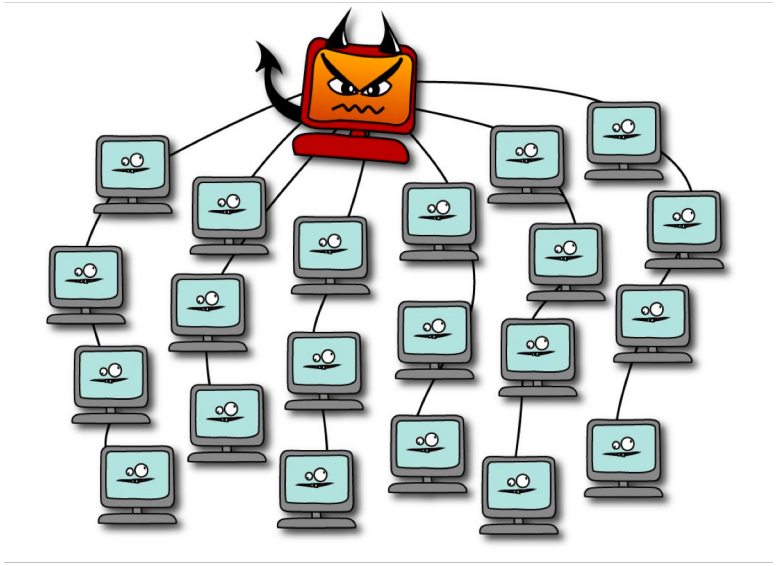
CSF: Forensics Cyber-Security

Fall 2019

Nuno Santos

Yesterday: Botnets

- The global threat of botnets
- Botnet operation and detection
- Investigating and taking down a botnet



Powerful tools for cybercrime

Anonymity systems

How criminals hide their IDs

Tools for cybercrime

Botnets

How to launch large scale attacks

Malware

Digital currency

How to make untraceable payments

Web

How to locate information and services

Class roadmap

- Malware and exploit kits
- Rootkits: hide and seek
- Introduction to malware analysis

TÉCNICO LISBOA

Malware and exploit kits

Malware

- Any software intentionally designed to cause damage to a computer, server or computer network
- Malware does the damage after it is implanted or introduced in some way into a target's computer
  - Can take the form of executable code, scripts, active content, and other software
- Some types of malware include:
  - Virus, trojans, worms, adware, spyware, ransomware

Exploits

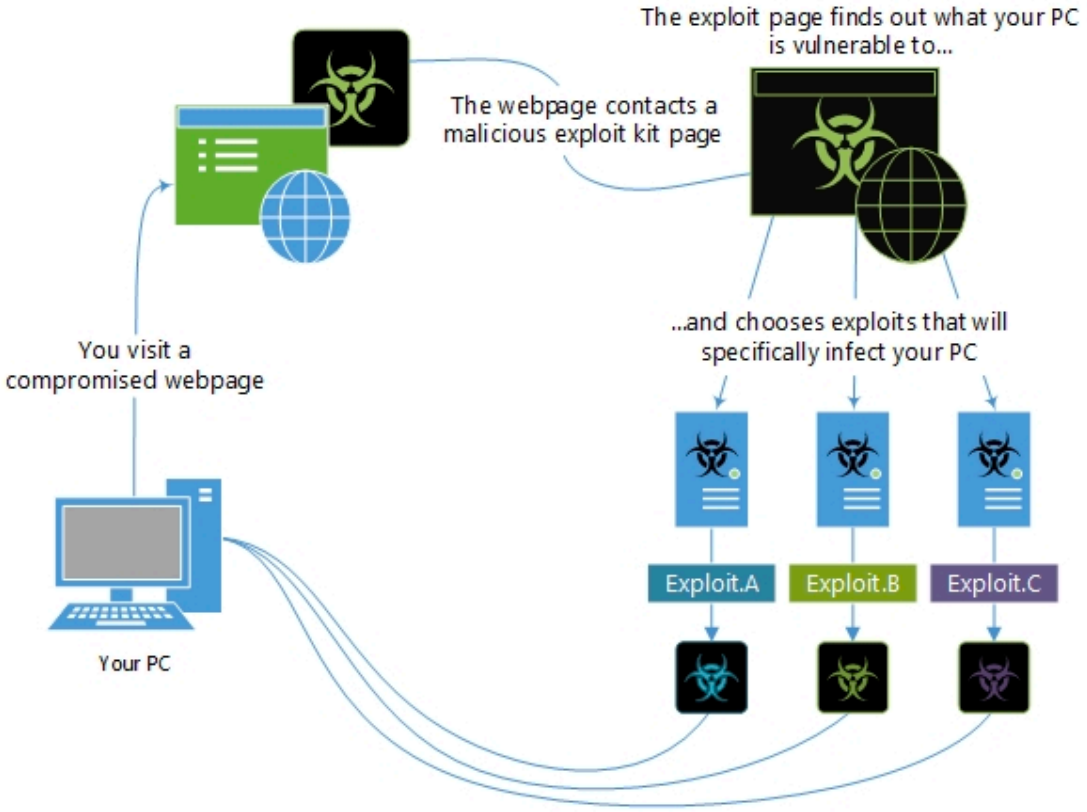
- Exploits are malicious programs that take advantage of application software or OS vulnerabilities

- Exploits typically target productivity applications such as Microsoft Office, Adobe applications, web browsers and operating systems, and they continue to pave the way for many malware-based attacks
  - Though not all exploits involve file-based malware (e.g.: null/default system password exploits, DDoS attacks)

Attacks through exploit kits

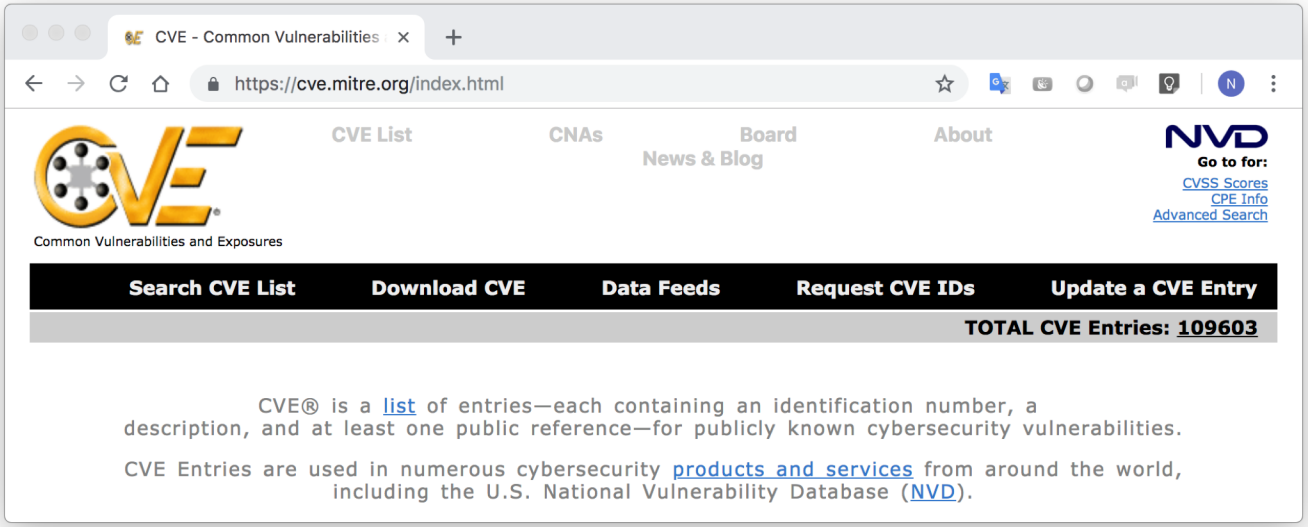
- The most common method to distribute exploits and exploit kits is through webpages, but exploits can also arrive in emails

- Exploit kits are more comprehensive tools that contain a collection of exploits



Exploits leverage exiting vulnerabilities

- A vulnerability is a mistake in software code that provides an attacker with direct access to a system or network
- **Common Vulnerabilities and Exposures (CVE)**
  - It is a program launched in 1999 by MITRE to identify and catalog vulnerabilities in software or firmware into a free "dictionary" for organizations to improve their security



Vulnerability reporting in CVEs

- The dictionary's main purpose is to standardize the way each known vulnerability or exposure is identified
- Example: **Shellshock** is a malware class that exploits **CVE-2014-6271** vulnerability reported in Bash
  - Allows to execute arbitrary code via the Unix Bash shell remotely

CVE-ID	
CVE-2014-6271	Learn more at National Vulnerability Database (NVD) • CVSS Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings • CPE Information

Description

GNU Bash through 4.3 processes trailing strings after function definitions in the values of environment variables, which allows remote attackers to execute arbitrary code via a crafted environment, as demonstrated by vectors involving the ForceCommand feature in OpenSSH sshd, the mod_cgi and mod_cgid modules in the Apache HTTP Server, scripts executed by unspecified DHCP clients, and other situations in which setting the environment occurs across a privilege boundary from Bash execution, aka "ShellShock." NOTE: the original fix for this issue was incorrect; CVE-2014-7169 has been assigned to cover the vulnerability that is still present after the incorrect fix.

References

Note: References are provided for the convenience of the reader to help distinguish between vulnerabilities. The list is not intended to be complete.

- BUGTRAQ:20141001 NEW VMSA-2014-0010 - VMware product updates address critical Bash security vulnerabilities
- URL:http://www.securityfocus.com/archive/1/533593/100/0/threaded
- EXPLOIT-DB:39918
- URL:https://www.exploit-db.com/exploits/39918/
- EXPLOIT-DB:40619
- URL:https://www.exploit-db.com/exploits/40619/

The Metasploit framework

- Metasploit Framework is a software platform for developing, testing, and executing exploits
  - It can be used to create security testing tools and exploit modules and as a penetration testing system
- Can incorporate new exploits in the form of modules (plug-ins)

https://www.exploit-db.com/exploits/39918/

IPFire - 'Shellshock' Bash Environment Variable Command Injection (Metasploit)

EDB-ID: 39918	**Author:** Metasploit	**Published:** 2016-06-10
CVE: CVE-2014-6271	**Type:** Remote	**Platform:** CGI
Aliases: N/A	**Advisory/Source:** N/A	**Tags:** Metasploit Framework (MSF)
E-DB Verified: ✔	**Exploit:** Download / View Raw	**Vulnerable App:** N/A

« Previous Exploit Next Exploit »

```
##
## This module requires Metasploit: http://metasploit.com/download
## Current source: https://github.com/rapid7/metasploit-framework
###

require 'msf/core'

class MetasploitModule < Msf::Exploit::Remote
  include Msf::Exploit::Remote::HttpClient

  def initialize(info = {})
    super(
      update_info(
        info,
        'Name'           => 'IPFire Bash Environment Variable Injection
(Shellshock)',
        'Description'    => %q(
          IPFire, a free linux based open source firewall
distribution,
          version <= 2.15 Update Core 82 contains an authenticated
```

TÉCNICO LISBOA

Rootkits: hide and seek

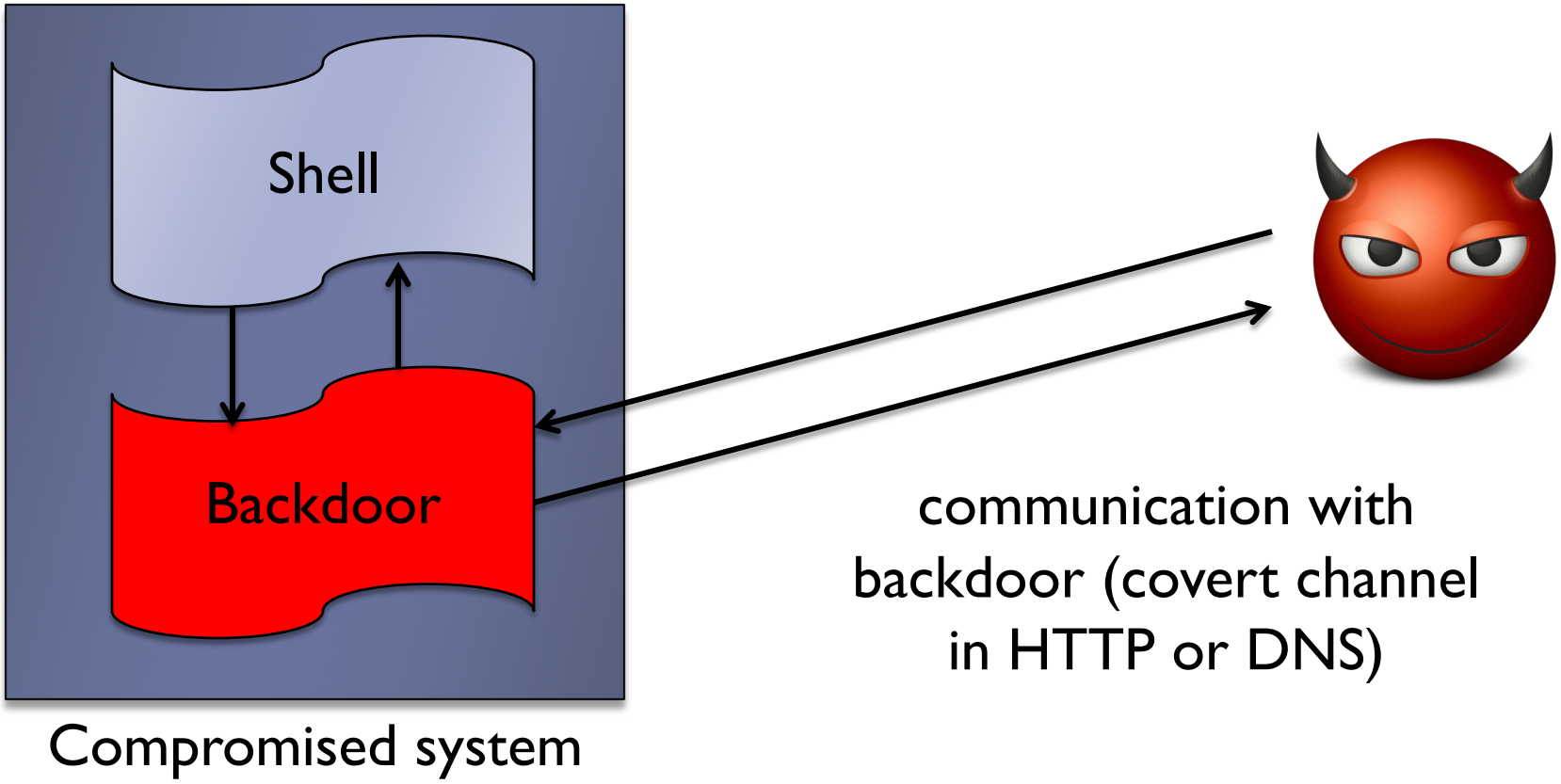
Rootkits

- The behavior of the operating system can be affected by the presence of **rootkits**
  - Enable access to a computer or areas of its software that is not otherwise allowed (for example, to an unauthorized user)

- Rootkits are a category of malware which has the ability to **hide itself** and cover up traces of activities

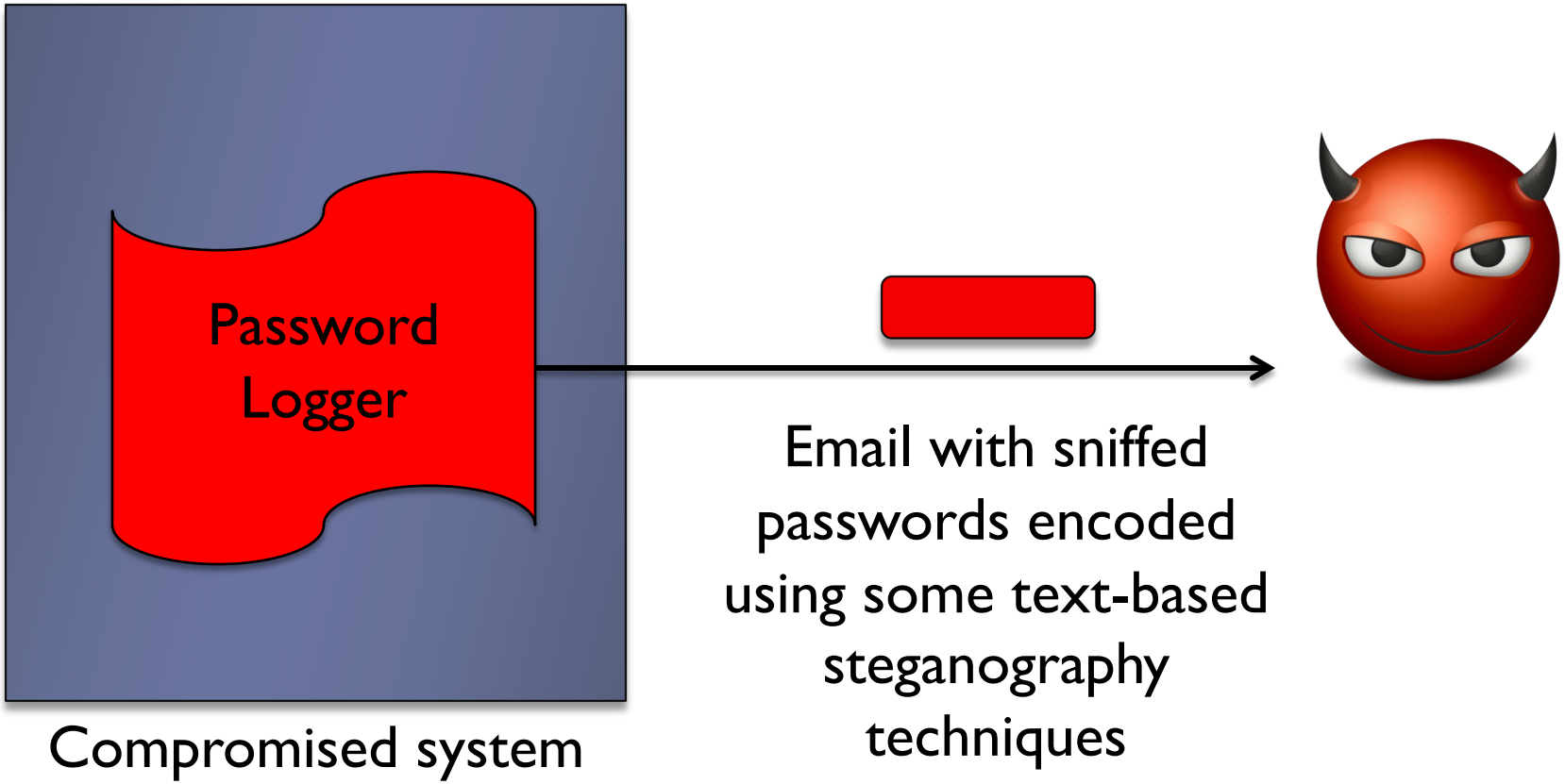
Examples

- Install a backdoor on the compromised system



Examples

- Run a password logger on a compromised system



Rootkit goals

1. Enable future access to system by attacker
2. Remove evidence of original attack and activity that led to rootkit installation
3. Hide future attacker activity (files, net connections, processes) and prevent it from being logged
4. Install tools to widen scope of penetration
5. Secure system so other attackers can't take control of system from original attacker

Rootkit tools: Backdoor programs

- Backdoor is an unauthorized way of gaining access to a program, online service or an entire computer system
  - Let attackers log in to the hacked system w/o using an exploit again

Examples	Description
Login Backdoor	Modify login.c to look backdoor password before stored password
Telnetd Backdoor	Trojaned the "in.telnetd" to allow attacker gain access with backdoor password
Services Backdoor	Replacing and manipulate services like "ftp", "rlogin", even "inetd" as backdoor to gain access
Cronjob backdoor	Backdoor could also be added in "crontjob" to run on specific time for example at 12 midnight to 1 am
Library backdoors	Shared libraries can be backdoor to do malicious activity including giving a root or administrator access
Kernel backdoors	This backdoor is basically exploiting the kernel

Rootkit tools: Sniffers and wipers

- Packet sniffers
  - Packet Sniffer is a program and/or device that monitor data traveling over a network, TCP/IP or other network protocol
  - Used to listen or to steal valuable information off a network; many services such as “ftp” and “telnet” transfer their password in plain text and it is easily capture by sniffer

- Log-wiping utilities
  - Log file are the lists actions that have occurred, e.g., in UNIX, wtmp logs time and date user log in into the system
  - Log file enable admins to monitor, review system performance and detect any suspicious activities
  - Deleting intrusion records helps prevent detection of the intrusion

Rootkit tools: Miscellaneous attacker tools

- DDOS program
  - To turn the compromised server into a DDOS client such as, trinoo
- IRC program
  - Connects to some remote server waiting for the attacker to issue a command (e.g., to trigger a distributed denial of service attack)
- System patch
  - Attacker may patch the system after successful attack; this will prevent other attacker to gain access into the system again
- Password cracker
- Vulnerability scanners
- Hiding utilities
  - Utilities to conceal the rootkit files on compromised system

Rootkit stealth techniques

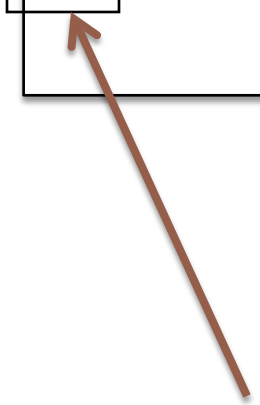
- File masquerading
- Hooking
- Direct Kernel Object Manipulation (DKOM)
- Virtualization

What's wrong with this picture?

```
[root@dobro bin]# ls -a
.               dd              igawk     nisdomainname  tar
..              df              ipcalc    pgawk          tcsh
..              dmesg           kbd_mode  ping           touch
alsaunmute      dnsdomainname   kill      ping6          tracepath
arch            doexec          ksh       ps             tracepath6
ash             domainname      link      pwd            traceroute
ash.static      dumpkeys        ln        red            traceroute6
aumix-minimal   echo            loadkeys  rm             true
awk             ed              login     rmdir          umount
```

Let's take a closer look

```
[root@dobro]# ls -a
.               dd              igawk     nisdomainname  tar
..              df              ipcalc    pgawk          tcsh
..              dmesg           kbd_mode  ping           touch
```



How can there be two ".." directories?...

How did this happen?

```
[root@dobro bin]# mkdir ..\
```

- This is actually:

```
mkdir <dot><dot><backslash><space><enter>
```

- It creates a directory named “dot-dot-space”

What's in this "mystery" directory?

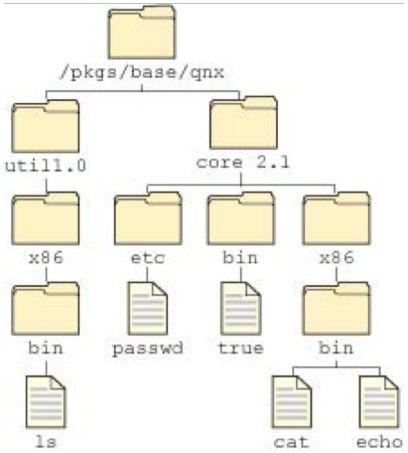
```
[root@dobro bin]# cd ..\
[root@dobro .. ]# ls -l
total 24
-rw-r--r--  1 root root 0 Dec 15 12:19 rootkit_file_01
-rw-r--r--  1 root root 0 Dec 15 12:19 rootkit_file_02
-rw-r--r--  1 root root 0 Dec 15 12:19 rootkit_file_03
-rw-r--r--  1 root root 0 Dec 15 12:19 rootkit_file_04
-rw-r--r--  1 root root 0 Dec 15 12:19 rootkit_file_05
-rw-r--r--  1 root root 0 Dec 15 12:19 rootkit_file_06
```

Here's a simple trick to hide malicious files in plain sight

1. File masquerading

- Replace system files (or directories) with malicious versions that shared the same name and services as the original
  - Or create files (or dirs) that resemble legitimate files (or dirs)

- Installation concealment:
  - Use spaces to make filenames look like expected dot files: “. “ and “.. “
  - Use dot files, which aren’t in ls output
  - Use a subdirectory of a busy system directory like /dev, /etc, /lib, or /usr/lib
  - Use filenames that system might use
    - /dev/hdd (if no 4th IDE disk exists)
    - /usr/lib/libX.a (libX11 is real Sun X-Windows)
  - Delete rootkit install directory once installation is complete



Change system commands

- Command-level rootkits hide malware by changing system commands
  - Based on principle: To suppress bad news, silence the messenger
  - Table shows examples of typical command-level rootkit modifications

Replaced Commands	Hidden Information
`du, find, ls`	Malware configuration files and network sniffer logs
`pidof, ps, top`	Network sniffer or back-door process
`netstat`	Network ports associated with malware
`ifconfig`	Network sniffing "enabled" status

2. Hooking

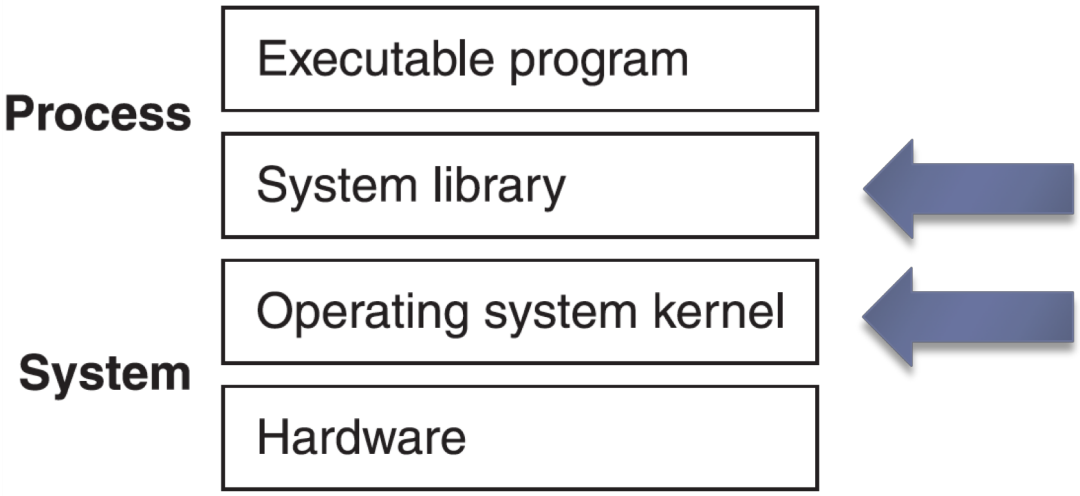
- The next step in evolution of rootkits was to redirect system calls to malicious code, a technique known as **hooking**

- Hooking is when **a given pointer** to a given resource or service **is redirected** to a different object
  - E.g., instead of replacing the file containing the "ls" command, a system call can be redirected to a custom "dir" command in memory space that filters out the malicious files and folders

- Basically, hooking achieves the same effect as file masquerading, but is more difficult to detect
  - Don't require changing executable files
  - Integrity checkers ineffective when validating executable files

Where hooking can be performed

- Hooking can be performed at several layers in the operating system, primarily libraries and kernel



Library-level hooking

- Instead of replacing system utilities, rootkits can hide their existence by making changes at the next level down in the system architecture: the **system run-time library**

- A good example is redirecting the **open()** and **stat()** calls
  - The purpose of these modifications is to fool file-integrity-checking software that examines executable file contents and attributes
  - By redirecting the open() and stat() calls to the original file, the rootkit makes it appear as if the file is still intact
  - However, execve() executes the subverted file

Example of library-level subversion

- Redirect specific open() system calls

```
#include <errno.h>
#include <syscall.h>
#include <real_syscall.h>

 /*
  * Define a real_open() function to invoke the SYS_open system call.
  */
static  real_syscall3(int, open, const char *, path,
                      int, flags, int, mode)

 /*
  * Intercept the open() library call and redirect attempts to open
  * the file /bin/ls to the unmodified file /dev/.hide/ls.
  */
int open(const char *path, int flags, int mode)
{
    if (strcmp(path, "/bin/ls") == 0)
        path = "/dev/.hide/ls";
    return (real_open(path, flags, mode));
}
```

real_syscall3() is a macro (not entirely shown) that modifies the standard _syscall13() macro

real_syscall3 is defines our real_open() function that invokes Sys_open system call

Kernel-level hooking

- Just like user-level rootkits, kernel-level rootkits are installed after a system's security has been breached
- Kernel-level rootkits compromise the kernel
  - Kernel runs in supervisor processor mode
  - Thus, the rootkit gains complete control over the machine
- Advantage: stealth, e.g.,
  - Runtime integrity checkers cannot see rootkit changes
  - All programs in the system can be affected by the rootkit
  - Open backdoors/sniff network without running processes

Methods to inject rootkit code into a kernel

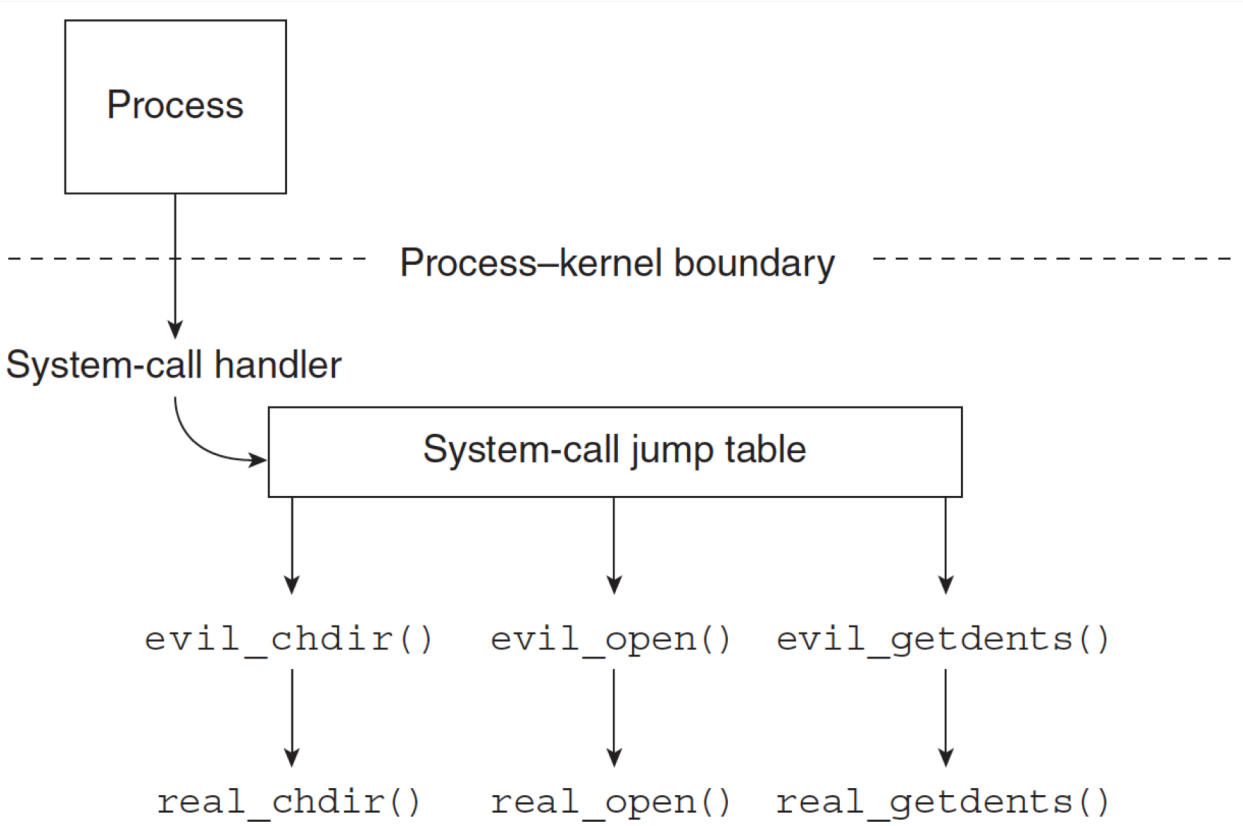
1. Loading kernel **module** into a running kernel
   - Uses official LKM interface, hence it's easier to use
   - Hide module names from external (/proc/ksyms) & internal tables
   - Might also intercept syscalls that report on status of kernel modules

2. Injecting code into the **memory** of a running kernel that has no support for module loading
   - Involves writing new code to unused kernel memory via /dev/kmem and activating the new code by redirecting, e.g., a system call

3. Injecting code into the kernel **file** or a kernel module file
   - These changes are persistent across boot, but require that the system is rebooted to activate the subverted code

Early kernel rootkit architecture

- Based on system-call interposition: Early kernel rootkits subvert syscalls close to the process-kernel boundary



Rootkit interposition code

- To prevent access to a hidden file, process, and so on, rootkits redirect specific system calls to wrapper code

```
evil_open(pathname, flags, mode)
    if (some_magical test succeeds)
        call real_open(pathname, flags, mode)
    else
        error: No such file or directory
```

- To prevent rootkit disclosure, syscalls that produce lists of files, etc., are intercepted to suppress info to be hidden

```
evil_getdents(handle, result)
    call real_getdents(handle, result)
    if (some_magical test fails)
        remove hidden objects from result
```

Routine patching

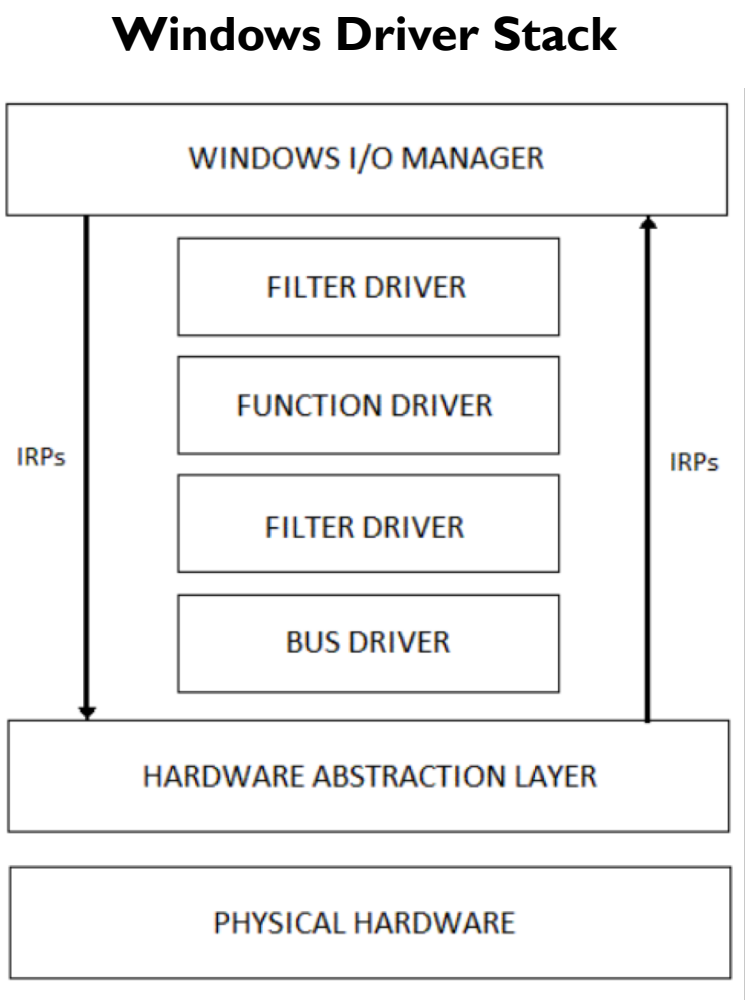
- Modify the source code of a system routine to cause the execution path to jump to malicious code which is resident either in memory or on disk

- Modern Windows-based rootkits may embed a JMP instruction within the system binary to redirect the execution path
  - This can be performed against the system binaries stored in the OS file system, or even against executing code loaded in memory

- Detection
  - If the modification was performed on the file system, this can be easily detected by file integrity monitoring systems
  - Run-time modification can be detected by applications such as Kernel Path Protection, which is provided by the 64-bit versions of Windows

Filter drivers

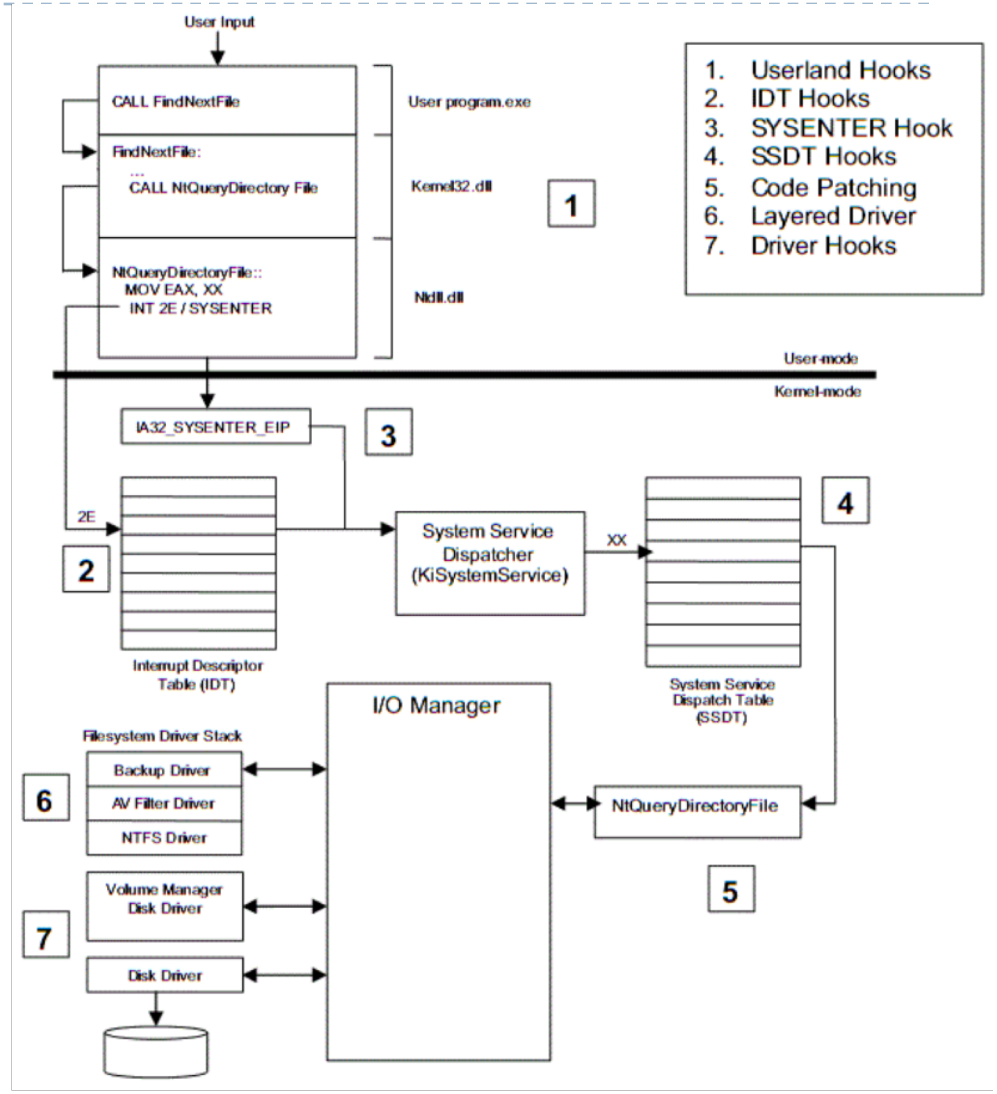
- The Windows driver stack architecture was designed in a layered manner
- This feature enables rootkit authors to inject their malicious code to interrupt the flow of I/O Request Packets and perform activities such as keystroke logging or filtering the results that are returned to anti-malware applications
- Rootkit authors can perform hooking of drivers, patch driver routines, or even create an entirely new driver and insert it into a driver stack



Potential hooking locations in Windows

- There are several different locations along the way that can be hooked to perform malicious activities

- These locations include:
  - Userland hooks in the Import Address Tables (IAT)
  - The Interrupt Descriptor Table (IDT)
  - The System Service Dispatch Table (SSDT)
  - Device drivers via I/O Request Packets



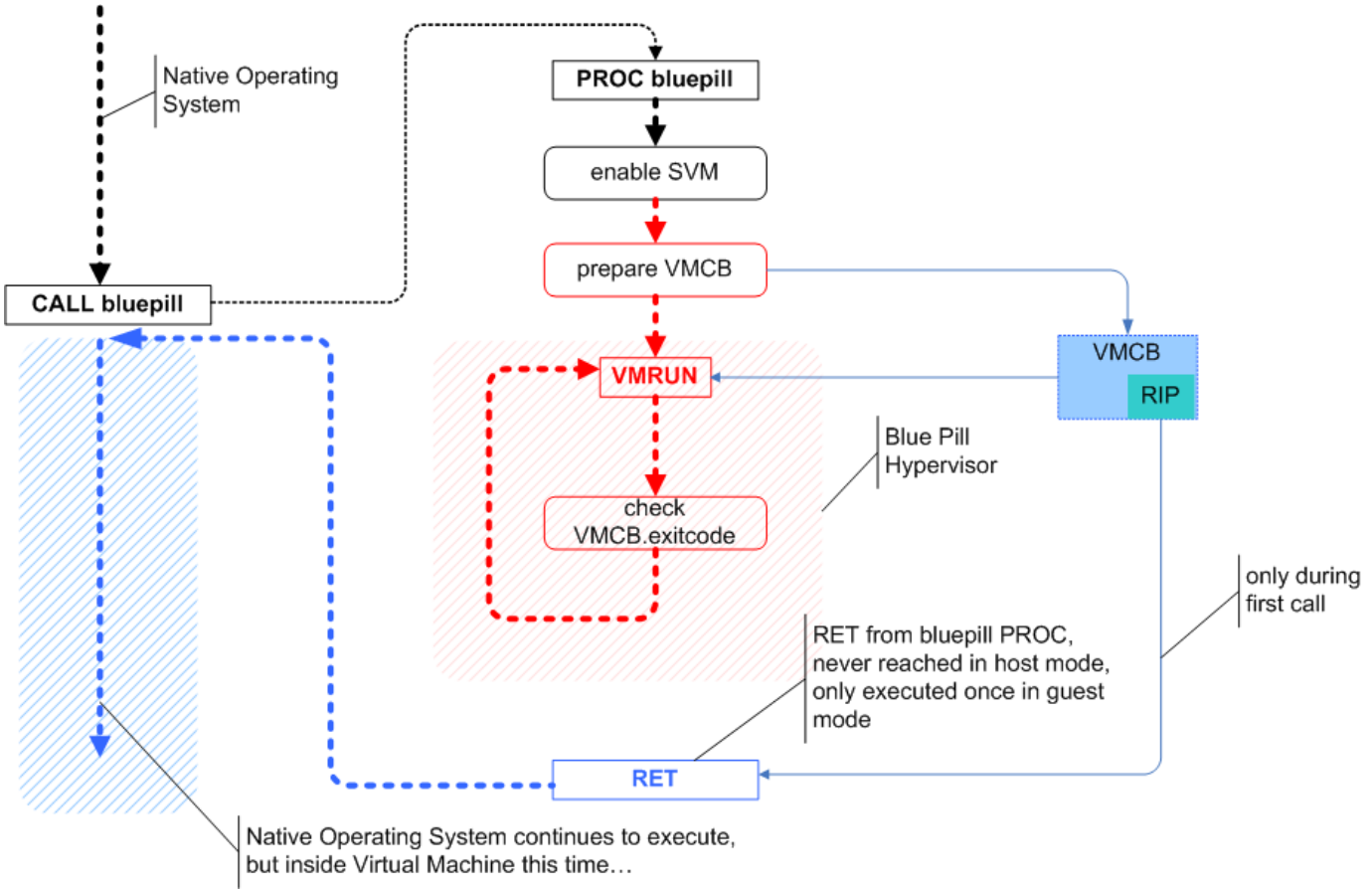
3. Direct Kernel Object Manipulation (DKOM)

- The third generation of rootkits used technique known as **Direct Kernel Object Manipulation** (**DKOM**)
- DKOM can manipulate kernel data structures to hide processes, change privileges, etc.
- The first known rootkit to perform DKOM was the FU rootkit, which modified the EPROCESS doubly linked list in Windows to “hide” the rootkit processes

4. Virtualization-based

- Leverage virtualization techniques to hide their presence "under" the native operating system

- Example: The Blue Pill rootkit
- Uses the SVM hw-virtualization instruction to install itself as a resident malicious hypervisor running in the computer



Approaches to detecting file masquerading

1. If a rootkit listens for connections, the network port will be visible to an external network port scanner
2. Some tools can reveal the names of all directory entries, including hidden or deleted files
3. Corrupted versions of ps and similar hide malware processes, but these can still be found using, e.g., the /proc file system
4. Deleted login/logout records in the wtmp file leave behind holes that can be detected using an appropriate tool
5. Ifconfig might report that a network interface is not in sniffer mode, but we can query the kernel for the interface status
6. CRC checksums reported by compromised cksum, can be detected using MD5 or SHA1
7. When examining a low-level copy of the file system on a trusted machine, all hidden files and modifications will be visible

Detection of kernel-level hooking

- Kernel rootkits may be exposed because they introduce **little inconsistencies** into a system
- Some may show up externally, in the results from system calls that manipulate processes, files, kernel modules, etc.
- Others show up only internally, in the contents of kernel data structures
  - E.g., hidden objects occupy some storage even though the storage does not appear in kernel symbol tables

Inconsistencies that may reveal kernel rootkits

- Output of tools that bypass the file system can reveal information that is hidden by compromised FS code
  - E.g., TSK
- Unexpected behavior of some system calls
  - E.g., when the Adore rootkit is installed, setuid() – change process privileges – will report success for some parameter value even though the user does not have sufficient privileges
  - E.g., when the Knark rootkit is installed, settimeofday() – set the system clock – will report success for some parameter values even though is should always fail when invoked by an unprivileged user
- Inconsistencies in the results from process-manipulating system calls and from the /proc file system
  - E.g., in reporting a process as "not found"

Inconsistencies that may reveal kernel rootkits

- Modifications to kernel tables, such as system call table or the virtual FS table
  - May be detected after the fact by reading kernel memory via /dev/kmem
  - Or by examining kernel memory from inside with a forensic kernel module such as Carbonite
- Modifications to kernel tables or kernel code may be detected using a kernel module that samples critical data structures periodically

Example of kernel toolkit detector tool

- Findrootkit can produce modification reports
  - Report example for a Solaris kernel

Interposed Vnode Operation	Interposing Function
specfs:ioctl	0xfe9f23c8
procfs:lookup	0xfe9f2080
procfs:readdir	0xfe9f22fc
ufs:setattr	0xfe9f1420
ufs:getattr	0xfe9f174c
ufs:lookup	0xfe9f1a08
ufs:readdir	0xfe9f1d50
ufs:remove	0xfe9f1e30
ufs:rename	0xfe9f1eec

(a)

Interposed System Call	Interposing Function
fork	0xfe9f2fb4
fork1	0xfe9f3058
kill	0xfe9f30fc
sigqueue	0xfe9f31a4
exec	0xfe9f324c
exece	0xfe9f3264

(b)

  - Changes to (a) the file system operations table, and (b) the system-call jump table

TÉCNICO LISBOA

Malware analysis

Why analyze malware?

- To assess damage
- To discover indicators of compromise
- To determine sophistication level of an intruder
- To identify a vulnerability
- To catch the “bad guy”®
- To answer questions…

To answer questions…

Business questions

- What is the purpose of the malware?
- How did it get here?
- Who is targeting us and how good are they?
- How can I get rid of it?
- What did they steal?
- How long has it been here?
- Does it spread on its own?
- How can I find it on other machines?
- How do I prevent this from happening in the future?

Technical questions

- Network indicators?
- Host-based indicators?
- Persistence mechanism?
- Date of compilation?
- Date of installation?

Static analysis techniques

- Hash the file
- Virus scan
  - Someone else may have already discovered and documented the program you are investigating
- List properties and type of file
  - E.g., file (in Linux)
- List strings inside the binary
  - E.g., strings (in Linux)
- Inspect raw bytes of the binary
  - E.g., hexdump (in Linux)
- List symbol info
  - E.g., nm (in Linux)
- View shared objects linked in at runtime
  - E.g., ldd (in Linux)
  - Listed in the .interp section Readelf, elfdump, objdump

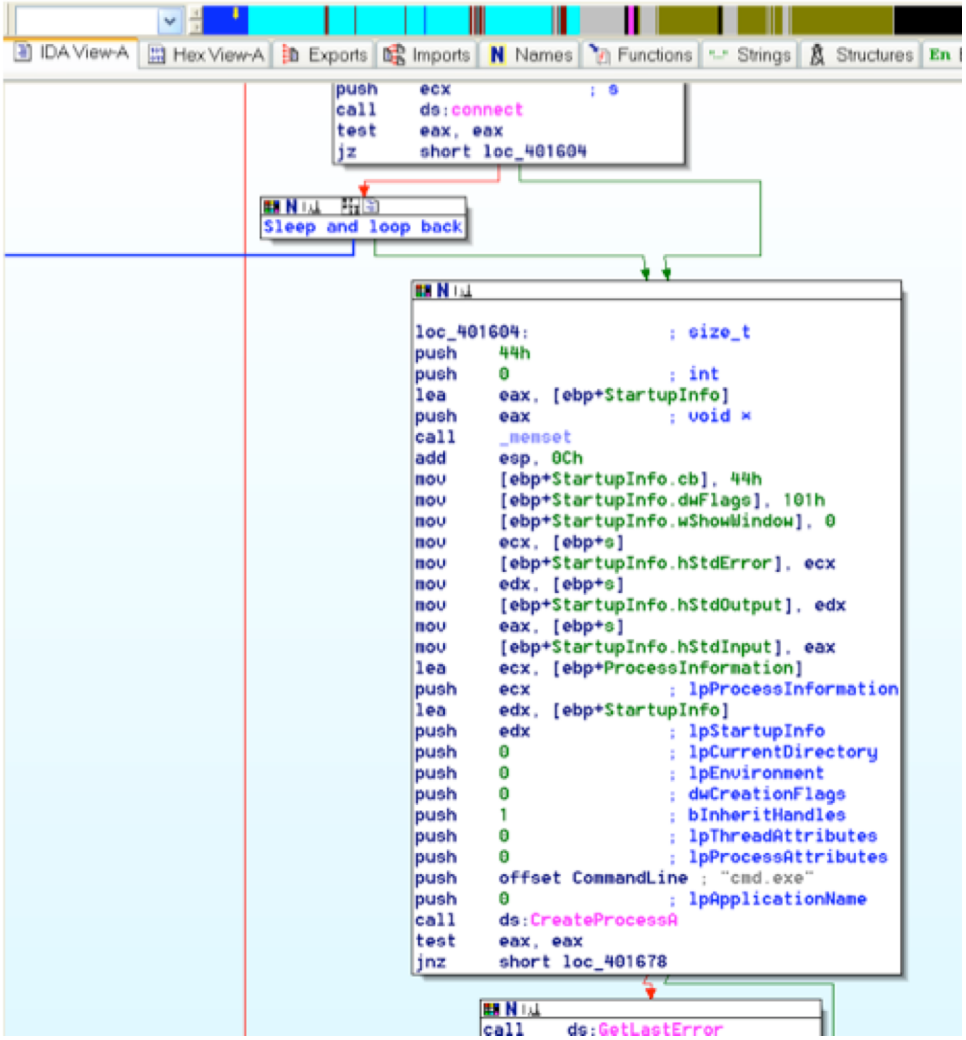
```
hjo@lnx:~/$ file winkill
winkill: ELF 32-bit LSB executable, Intel
80386, version 1 (SYSV), for GNU/Linux
2.0.0, dynamically linked (uses shared libs),
for GNU/Linux 2.0.0, not stripped
```

```
hjo@lnx:~/$ nm winkill
...
08048784 T parse_args
08049c78 D port
         U printf@@GLIBC_2.0
08048760 T usage
         U usleep@@GLIBC_2.0
…
D  The symbol is in the initialized .data section
T  The symbol is in the .text (code) section
U  The symbol is unknown
…
```

```
hjo@lnx:~/$ ldd winkill
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => /lib/tls/i686/cmov/libc.so.6
(0xb7e36000)
        /lib/ld-linux.so.2 (0xb7f70000)
```

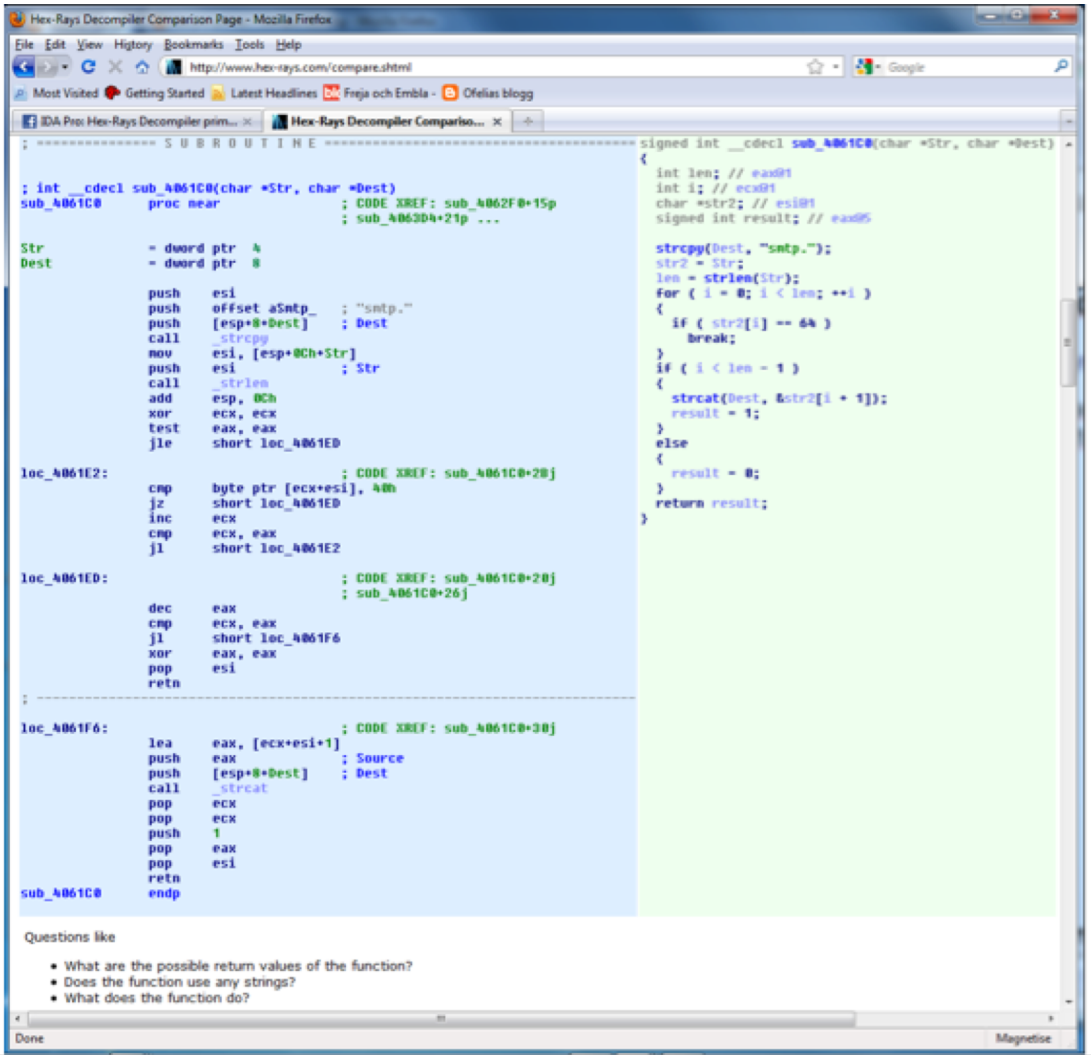
Static analysis techniques

- Disassembly: Automated disassemblers can take machine code and "reverse" it to a slightly higher-level
  - Many tools can disassemble x86 code
  - Objdump, Python w/ libdisassemble, IDA Pro (very popular)
- Manual examination of disassembly is painstaking, slow, and can be hard



Static analysis techniques

- Decompilation
  - Type of reverse engineering that takes an executable file as input, and attempts to create a high level source file which can be recompiled successfully
  - The reverse of a compiler



Dynamic analysis

- Static analysis will reveal some immediate information
- Exhaustive static analysis could theoretically answer any question, but it is slow and hard
- Usually you care more about “what” malware is doing than “how” it is being accomplished
- Dynamic analysis is conducted by observing and manipulating malware as it runs

Creating safe environment

- Do not run malware on your computer ☺
  - Create a safe environment for dynamic analysis!
- Old and busted:
  - Shove several PCs in a room on isolated network, create disk images, re-image a target machine to return to pristine state
- Better: Use virtualization to make things fast and safe
  - Xen, VMWare, VirtualBox, etc.
  - FLARE VM: VM for Windows malware analysis
    - https://github.com/fireeye/flare-vm

Creating safe environment

- It is easier to perform analysis if you allow the malware to "call home"...

- However:
  - The attacker might change his behavior
    - By allowing malware to connect to a controlling server, you may be entering a real-time battle with an actual human for control of your analysis (virtual) machine
  - Your IP might become the target for additional attacks
  - You may end up attacking other people

Creating safe environment

- Therefore, we usually do not allow malware to touch the real network
  - May entirely disable the networking capability
  - Or use the host-only networking feature of your virtualization platform

- More advanced
  - Establish real services (DNS, Web, etc) on your host OS or other virtual machines
  - Use netcat to create listening ports and interact with text-based client
  - Build custom controlling servers as required (usually in a high-level scripting language)

Dynamic analysis techniques

- System Call Trace (strace)
  - hjo@lnx:~/$ strace -d ./winkill
  - Library Call Trace (ltrace)
  - *trace got similar options

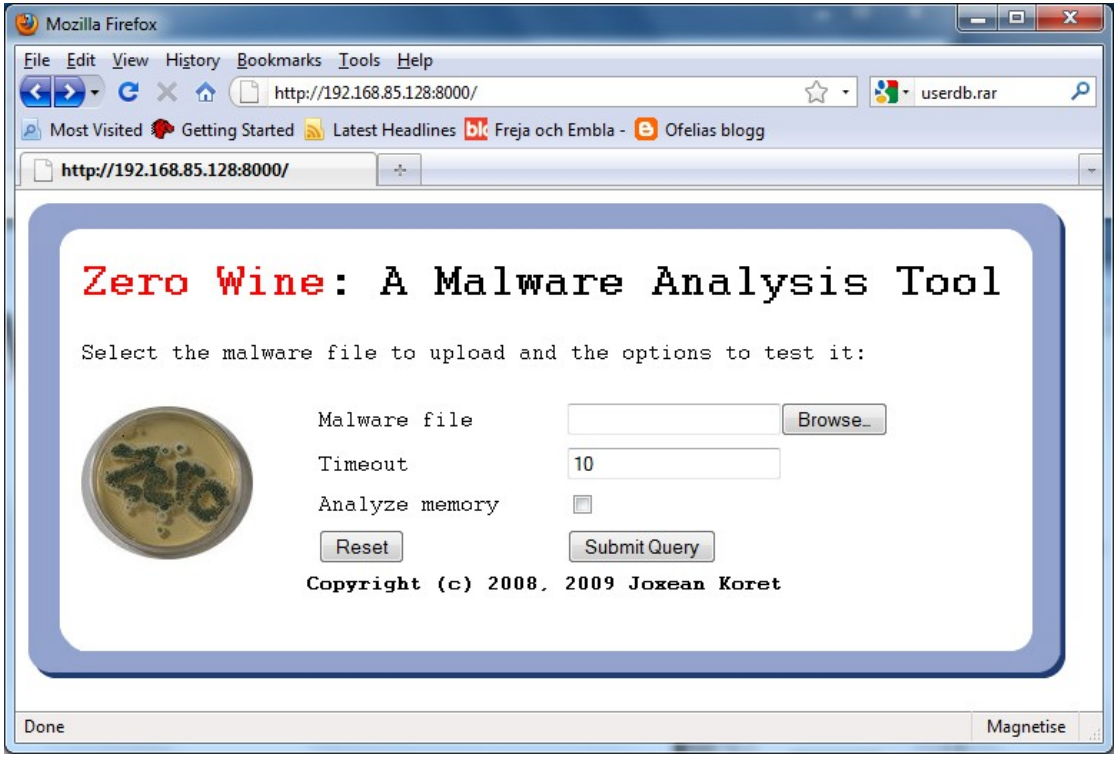
```
hjo@lnx:~/$ ltrace ./winkill
__libc_start_main(0x8048874, 1, 0xbfd3d314, 0x8048528,
0x8048b2c <unfinished ...>
__register_frame_info(0x8049c7c, 0x8049d90,
0xbfd3d298, 0x804854d, 0xb7faaff4)    = 0
printf("Usage: %s <host> -p port -t hits"...,
"./winkill"Usage: ./winkill <host> -p port -t hits
)                                     = 40
exit(1 <unfinished ...>
__deregister_frame_info(0x8049c7c, 0xbfd397a8,
0x8048b41, 0xb7faaff4, 0xbfd397c8) = 0
+++ exited (status 1) +++
```

- The GNU debugger
  - Huge subject: http://www.gnu.org/software/gdb/
    - Google on"gnu debugger gdb tutorial"
  - Stop program execution
  - Control program flow
  - Examine data structures
  - Disassemble etc. etc. etc. ...

- Dump the process' RAM and analyze it

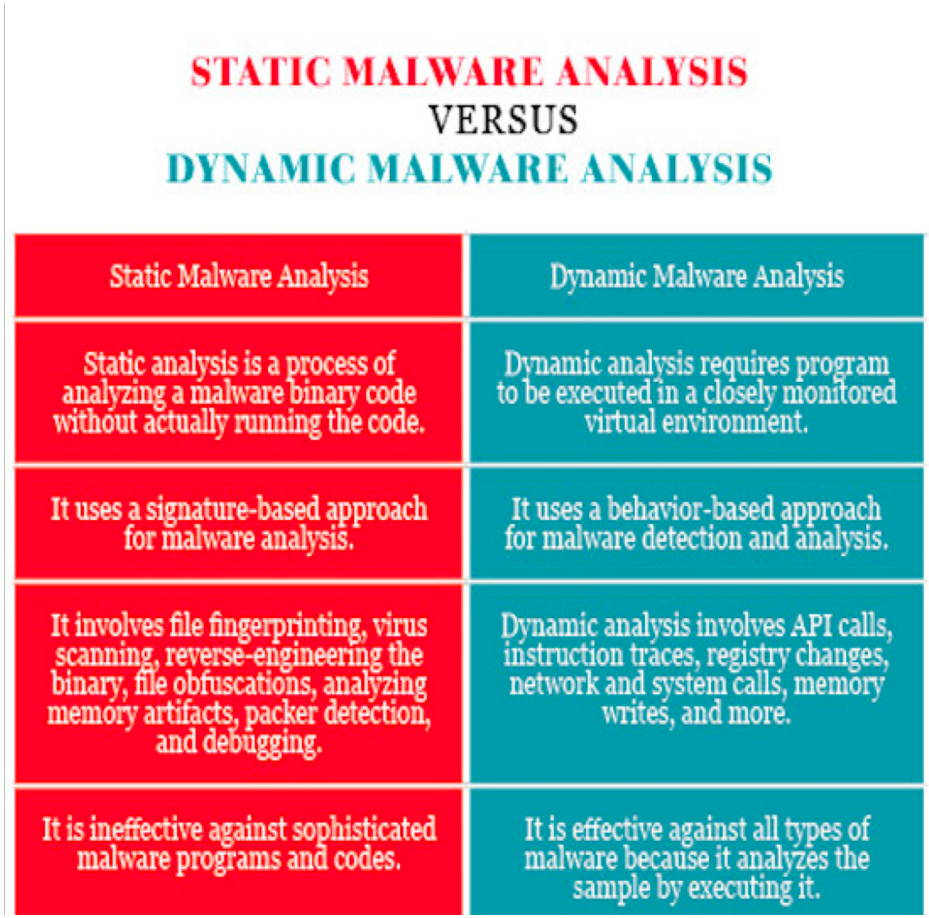
Zero Wine: Malware behavior analysis

- Upload malware perform static and dynamic analyze
  - http://zerowine.sourceforge.net/
- Virtual machine using Qemu or VMware and Linux/Wine
- Output:
  - Raw trace (Report)
  - Strings
  - PE headers
  - Signature (API calls)



Malware analysis methods

- **Static analysis**
  - No execution
  - Extensive search in the binary with various tools
- **Dynamic analysis**
  - Execution
  - Extensive monitoring
  - Alter the execution and program flow
- Static analysis is safer
  - Since we aren't actually running malicious code, we don't have to worry (as much) about creating a safe environment



Malware analysis template

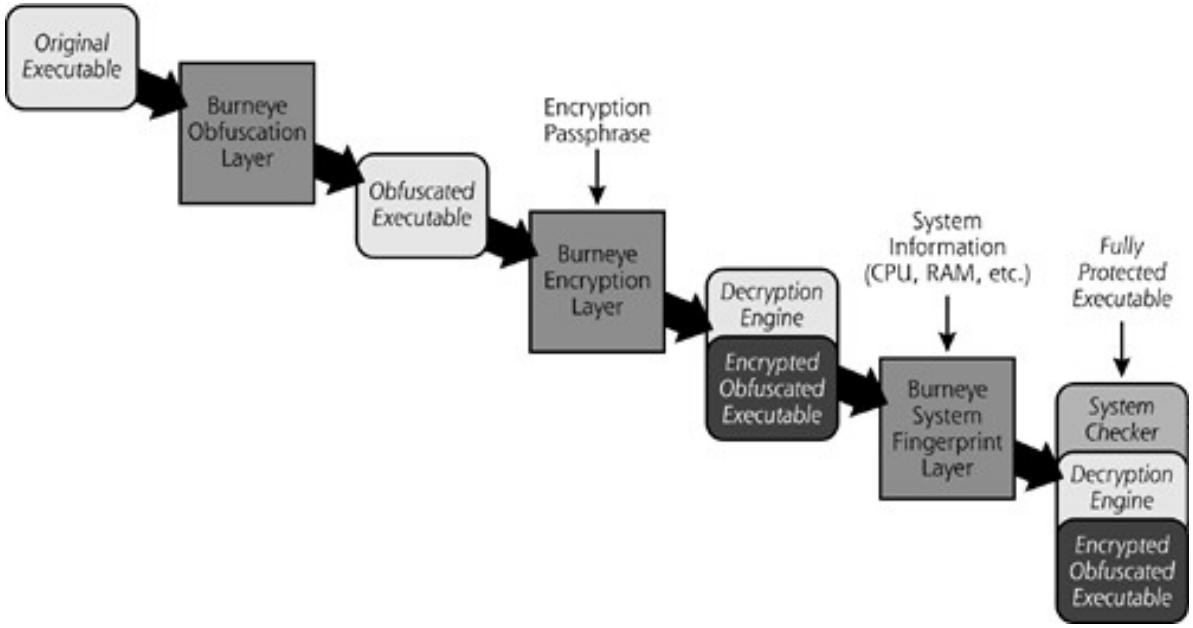
Static analysis

Dynamic analysis

Activity	Observed Results
Load specimen onto victim machine	
Run antivirus program	
Research antivirus results and file names	
Conduct strings analysis	
Look for scripts	
Conduct binary analysis	
Disassemble code	
Reverse-compile code	
Monitor file changes	
Monitor file integrity	
Monitor process activity	
Monitor local netwok activity	
Scan for open ports remotely	
Scan for vulnerabilities remotely	
Sniff network activity	
Check promiscuous mode locally	
Check promiscuous mode remotely	
Monitor registry activity	
Run code with debugger	

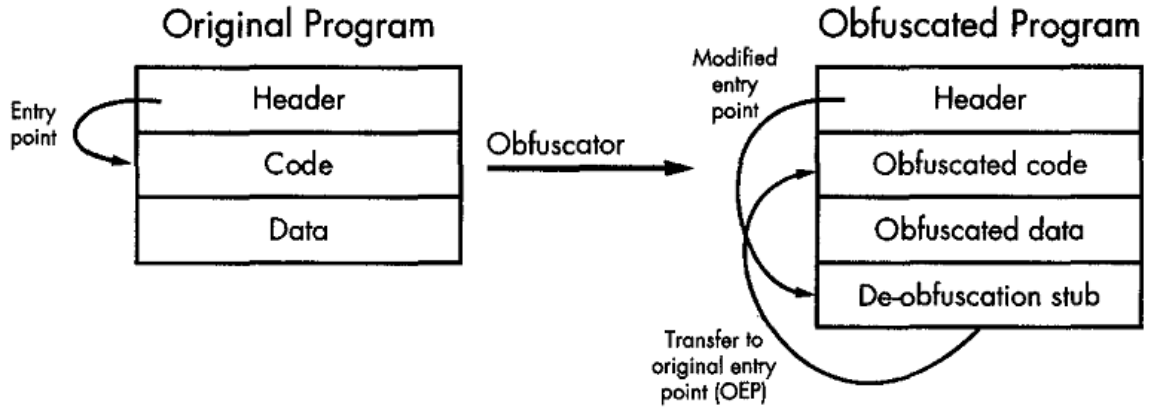
Analyzing malware binaries can be challenging

- BurnEye is a tool designed to protect binary files and is an example on how to protect malware
- It adds three protective layers to an executable file:
  - **Obfuscation:** scrambles the code in the executable thru obfuscated instructions
  - **Encryption** of the binary program
  - **Fingerprint layer:** will only run on certain computers



Anti-static analysis techniques

- Disassembly desynchronization
  - Prevent the disassembly from finding the correct starting address for one or more instructions. Forcing the disassembler to lose track of itself
- Dynamically computed target addresses
  - Address to which execution will flow is computed at run-time



Anti-static analysis techniques

- Opcode obfuscation
  - Encode or encrypt the actual instructions when the executable file is being created (self modification)
- Imported function obfuscation
  - In order to avoid leaking information about potential actions that a binary may perform, aimed at making it difficult for the static analysts to determine which shared libraries and library functions are used within an obfuscated binary
- Targeted attacks on analysis tools

Anti-dynamic analysis techniques

- Detecting virtualization
  - Detection of virtualization-specific software and hardware
  - Detection of virtual machine-specific behaviors
  - Detection of processor-specific behavioral changes (blue/red pill etc.)
- Detecting instrumentation (Sysinternals tools, WireShark etc.)
  - Check loaded drivers, scan active process list or windows title texts etc.
- Detecting debuggers
  - API functions such as the Windows IsDebuggerPresent(), NtQueryInformationProcess() or OutputDebugStringA()
  - Lower-level checks for memory or processor artifacts resulting from the use of a debugger
    - Detecting that a processor's trace (single step) Trap Flag (TF) is set.
  - SoftIce, a Windows kernel debugger, can be detected through the presence of the "\\.\NTICE" device (named pipe), which is used to communicate with the debugger

Conclusions

- Many attacks to operating systems are performed through rootkit software
- Depending on the rootkit, the forensic analyst needs to employ different rootkit-detection techniques
- Malware analysis allows for determining the behavior of malicious binaries and usually entails the adoption of static and dynamic analysis techniques

References

- Primary bibliography
  - Dan Farmer, Wietse Venema, *Forensic Discovery*, Chapter 5

Next class

- III.5 Cryptocoins