

CS 798: Digital Forensics and Incident Response

Lecture 15 - Stealthy Malware

Diogo Barradas

Winter 2025

University of Waterloo

Malware analysis

- It is an essential part of digital forensics because it can help:
 - Identify the source of an attack
 - Determine the extent of damage
 - Develop remediation strategies

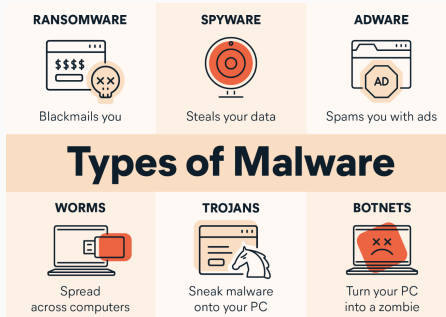


1. Malware and exploits
2. Rootkits
3. Malware Analysis

Malware and exploits

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



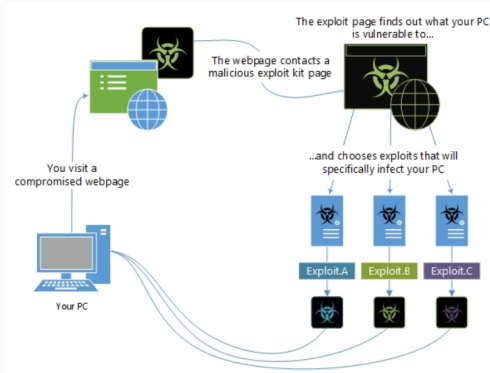
Exploits

- 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
 - Not all exploits involve file-based malware (e.g.: null/default system password exploits, DDoS attacks)



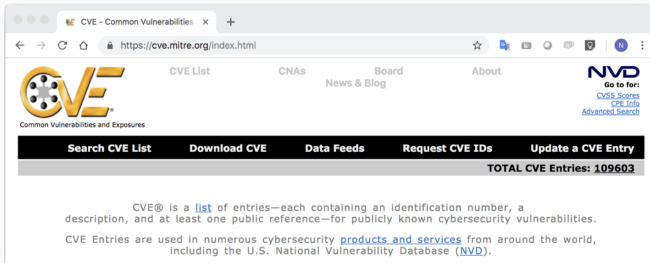
Attacks through exploit kits

- 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 existing 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)**
 - 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 the **CVE-2014-6271** vulnerability reported in Bash
 - Allows remote code execution via the Unix Bash shell

CVE-ID	
CVE-2014-6271	Learn more at National Vulnerability Database (NVD) <ul style="list-style-type: none">• 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](http://www.securityfocus.com/archive/1/533593/100/0/threaded)
- EXPLOIT-DB:39918
- [URL:https://www.exploit-db.com/exploits/39918/](https://www.exploit-db.com/exploits/39918/)
- EXPLOIT-DB:40619
- [URL:https://www.exploit-db.com/exploits/40619/](https://www.exploit-db.com/exploits/40619/)

The Metasploit framework

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

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 »

```
1  ##
2  ## This module requires Metasploit: http://metasploit.com/download
3  ## Current source: https://github.com/rapid7/metasploit-framework
4  ###
5
6  require 'msf/core'
7
8  class MetasploitModule < Msf::Exploit::Remote
9    include Msf::Exploit::Remote::HttpClient
10
11    def initialize(info = {})
12      super(
13        update_info(
14          info,
15          {
16            'Name' => 'IPFire Bash Environment Variable Injection
17              (Shellshock)',
18            'Description' => %q(
19              IPFire, a free linux based open source firewall
20              distribution,
21              version <= 2.15 Update Core 82 contains an authenticated
```

Rootkits

Hide and seek

- 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 (e.g., to an unauthorized user)



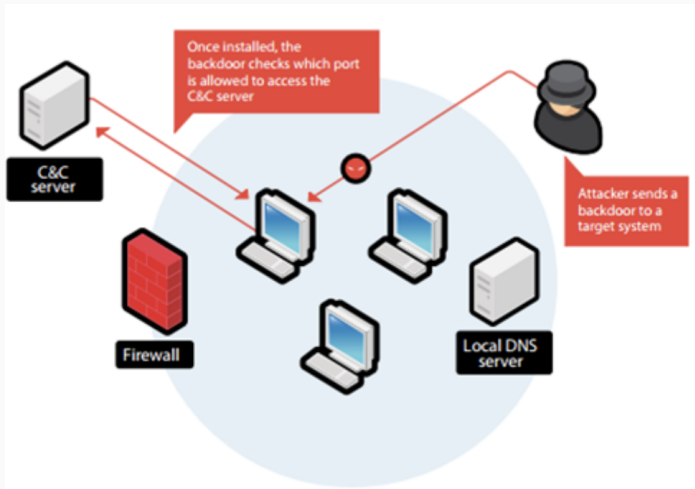
- Rootkits are a **category of malware** which has the ability to **hide itself** and cover up traces of activities

Rootkit goals

1. Enable attacker to access the system in the future
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

Example: Backdoor

- Install a backdoor on the compromised system
 - Communication may happen via a covert channel



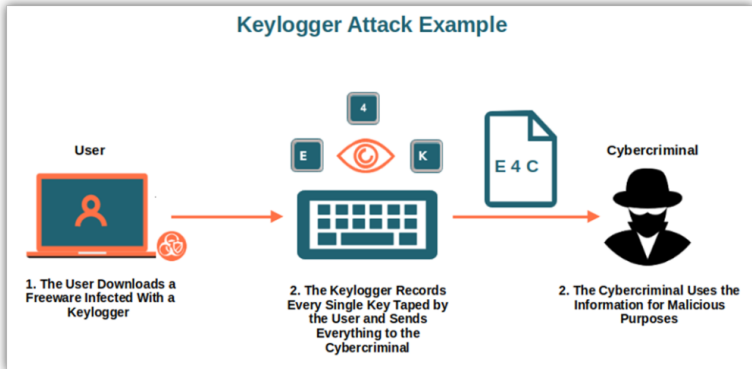
Backdoor programs

- A 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 again

Examples	Description
Login Backdoor	Modify <code>login.c</code> to look for a backdoor password before the stored password.
Telnetd Backdoor	Trojaned the <code>in.telnetd</code> to allow an attacker to gain access with a backdoor password.
Services Backdoor	Replacing and manipulating services like <code>ftp</code> , <code>rlogin</code> , and even <code>inetd</code> as backdoors to gain access.
Cronjob Backdoor	Backdoors could also be added in <code>crontab</code> to run at specific times, for example, from 12 midnight to 1 am.
Library Backdoors	Shared libraries can be used as backdoors to perform malicious activities, including providing root or administrator access.
Kernel Backdoors	These backdoors essentially exploit the kernel.

Example: Keylogger

- Run a password logger on a compromised system
 - Keystrokes may be exfiltrated using steganography



Rootkit tools: Sniffers and wipers

- **Packet sniffers**

- Programs and/or device that monitor data traveling over a network, TCP/IP or other network protocol
- Used to steal valuable information off a network; many services such as `ftp` and `telnet` transfer passwords in plaintext

- **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

Rootkit tools: Miscellaneous attacker tools

- **DDoS program**
 - To turn the compromised server into a DDoS client
- **IRC program**
 - Connects to some remote server waiting for the attacker to issue a command (e.g., to trigger a DDoS 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

1. File masquerading
2. Hooking
3. Direct Kernel Object Manipulation (DKOM)
4. 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	tracpath
arch	doexec	ksh	ps	tracpath6
ash	domainname	link	pwd	traceroute
ash.static	dumpkeys	ln	red	traceroute6
aumix-minimal	echo	loadkeys	rm	true
awk	ed	login	rmdir	umount

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
```

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”

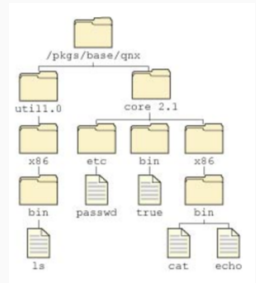
What's in this mysterious directory?

```
[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
```

- Nice 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, not shown in `ls` output
 - Use a subdirectory of a system dir. like `/dev`, `/etc`, `/lib`, or `/usr/lib`
 - Use names the system might use (e.g., `/dev/hdd` if no 4th IDE disk exists)
 - Delete rootkit install directory once installation is complete



1. File masquerading (cont.)

- **Change system commands:**
 - To suppress bad news, silence the messenger
 - The table shows examples of typical command-level rootkit modifications

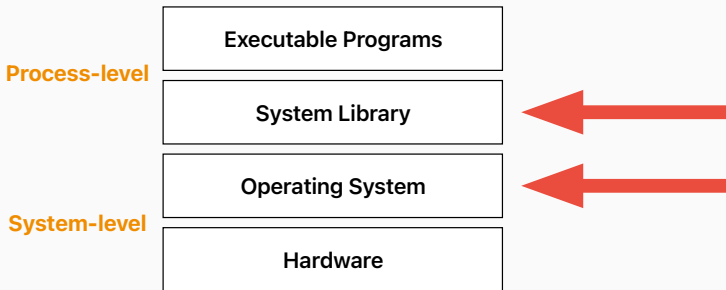
Replaced Commands	Hidden Information
<code>du, find, ls</code>	Malware configuration files and network sniffer logs
<code>pidof, ps, top</code>	Network sniffer or back-door process
<code>netstat</code>	Network ports associated with malware
<code>ifconfig</code>	Network sniffing “enabled” status

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**
 - Does not require changing executable files
 - Integrity checks are ineffective when validating executable files

Where does hooking happen

- 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
 - `real_syscall3()` is a macro (not entirely shown) that modifies the standard `_syscall13()` macro
 - `real_syscall3` defines our `real_open()` function that invokes the `Sys_open` system call

```
#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));
}
```

Kernel-level hooking

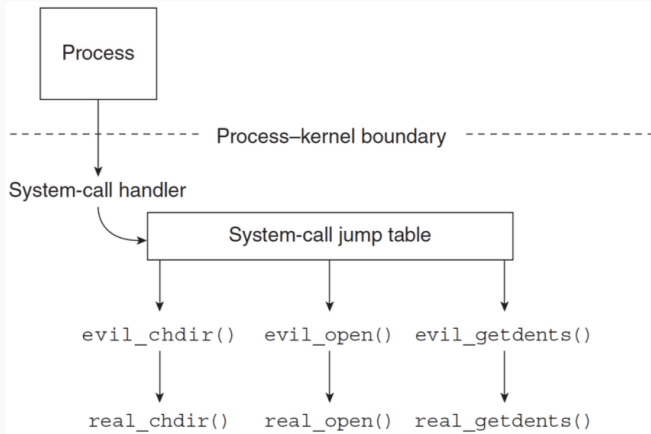
- Just like user-level rootkits, **kernel-level rootkits** are installed after a system's security has been breached
- Kernel-level rootkits compromise the kernel
 - Kernel runs in supervisor processor mode
 - Thus, the rootkit gains complete control over the machine
- Advantage: stealthiness
 - Runtime 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 a **kernel module** into a running kernel
 - Use official LKM interface, hence it's easier to use
 - Hide module names from external (`/proc/ksyms`)
 - 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
 - Write 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

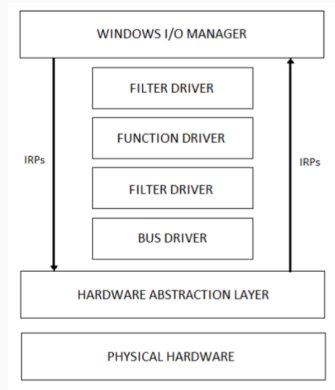
```
evil_getdents(handle, result)
    call_real_getdents(handle, result)
    if (some_magical_test_fails)
        remove_hidden_objects_from_result
```

Routine patching

- Modify the code of a system routine to **cause the execution path to jump** to malicious code which may live 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 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

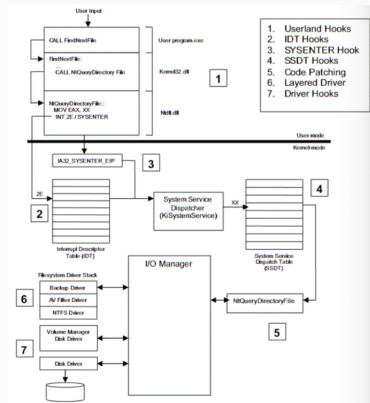
Filter drivers

- 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 Requests 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 create a 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

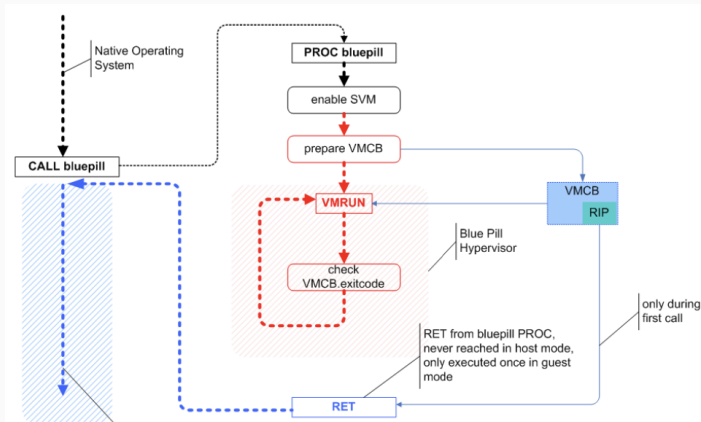


3. Direct Kernel Object Manipulation (DKOM)

- The third generation of rootkits used a 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
 - e.g., the (particularly evil) **Blue Pill** rootkit
 - Uses hw-virtualization to install itself as a resident malicious hypervisor and run the original OS as a VM



Are we doomed?

Detection of 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 its status
6. CRC checksums reported by compromised `cksum`, can be detected using MD5 or SHA1
7. Examining a low-level copy of the file system on a trusted machine reveals all hidden files and modifications

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 internally, within 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 it 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 (cont.)

- 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

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 criminals
- To answer a few more questions...



A few more questions...

Operational 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?

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 it
- 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 linked shared objects
 - e.g., ldd (in Linux)

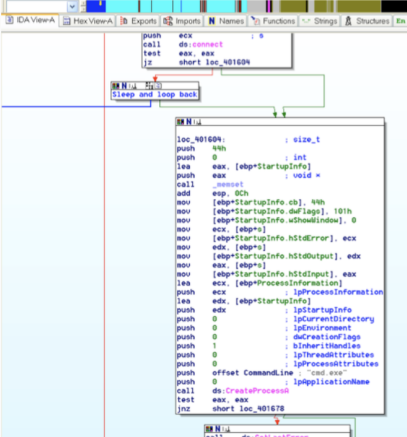
```
hjo@lnx:~/ $ file winkill
winkill: ELF 32-bit LSB executable, Intel
80386, version 1 (SYSV), for GNU/Linux
2.0.0, dynamically linked (uses shared libs),
for GNU/Linux 2.0.0, not stripped
```

```
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 => (0xffff0000)
libc.so.6 => /lib/tls/i686/cmov/libc.so.6
(0xb7e36000)
/lib/ld-linux.so.2 (0xb7f70000)
```

Static analysis techniques

- **Disassembly:** Take machine code and “reverse” it to a higher-level
 - Many tools can disassemble x86
 - e.g., Objdump, Python w/ libdisassemble, IDA Pro
- Manual examination of disassembled code can be quite hard



The screenshot shows the IDA Pro interface. The top menu bar includes 'IDA View-A', 'Hex View-A', 'Exports', 'Imports', 'Names', 'Functions', 'Strings', 'Structures', and 'En'. The main window displays assembly code for a function. A control flow graph is visible, showing a loop labeled 'Sleep and loop back'. The assembly code includes instructions like 'push ecx', 'call ds:connect', 'test eax, eax', 'jz short loc_401604', and a series of 'push' and 'mov' instructions for setting up a process creation structure. The code ends with 'call ds:CreateProcessA' and 'call ds:GetLastError'.

```
push    ecx                ; 6
call    ds:connect
test    eax, eax
jz      short loc_401604

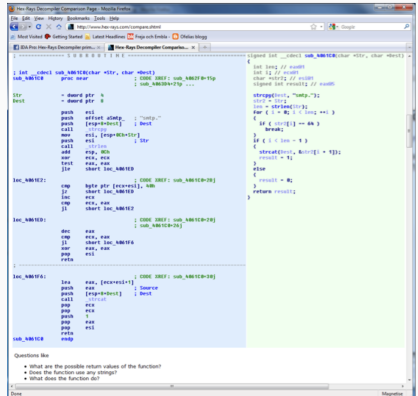
; Sleep and loop back

loc_401604:
push    44h                ; size_t
push    0                  ; int
lea     eax, [ebp+StartupInfo]
push    eax                ; void *
call    _memset
add     esp, 0Ch
mov     [ebp+StartupInfo.cb], 44h
mov     [ebp+StartupInfo.dwFlags], 101h
mov     [ebp+StartupInfo.hStdInput], 0
mov     ecx, [ebp+]
mov     [ebp+StartupInfo.hStdError], ecx
mov     edx, [ebp+]
mov     [ebp+StartupInfo.hStdOutput], edx
mov     eax, [ebp+]
mov     [ebp+StartupInfo.hStdInput], eax
lea     ecx, [ebp+ProcessInformation]
push    ecx                ; lpProcessInformation
lea     edx, [ebp+StartupInfo]
push    edx                ; lpStartupInfo
push    0                  ; lpCurrentDirectory
push    0                  ; lpEnvironment
push    0                  ; dwCreationFlags
push    1                  ; bInheritHandles
push    0                  ; lpThreadAttributes
push    0                  ; lpProcessAttributes
push    offset CommandLine ; "cmd.exe"
push    0                  ; lpApplicationName
call    ds:CreateProcessA
test    eax, eax
jnz     short loc_401678

call    ds:GetLastError
```


Static analysis techniques (cont.)

- **Decompilation:** Take an executable file and create a high-level source file
 - i.e., reverse of a compiler



Categorizing malware

- Investigators will identify unique patterns and strings within a given piece of malware
 - This allows for identifying the sample's malware family
- YARA helps investigators describe these patterns
 - Through rules that look for certain characteristics

```
rule silent_banker : banker
{
    meta:
        description = "This is just an example"
        threat_level = 3
        in_the_wild = true

    strings:
        $a = {6A 40 68 00 30 00 00 6A 14 8D 91}
        $b = {8D 4D B0 2B C1 83 C0 27 99 6A 4E 59 F7 F9}
        $c = "UVODFRYSIHLNWPEJXQZAKCBGMT"

    condition:
        $a or $b or $c
}
```

Dynamic analysis

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

Creating a safe environment for dynamic analysis

- **Rule of thumb:** Do not run malware on your computer!
 - Create a safe environment for dynamic analysis!
- Tried and tested way
 - 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.
 - Mandiant's FLARE VM: VM for Windows malware analysis



Creating a safe environment for dynamic analysis

- It is easier to perform analysis if you allow the malware to “call home”, however:
 - The attacker might **change their behavior** by allowing the malware to connect to a control server
 - Your IP might become the target for additional attacks
 - You may end up attacking other people
- Therefore, investigators usually do not allow malware to touch the real network, but may establish realistic services (DNS, Web, etc) on the host OS or other VMs

Virtualization considerations

- Our virtualization software is **not perfect**
- Malicious code can detect that it is running in a VM
 - It can then remain dormant, or worse...
 - It can use a 0-day exploit and escape the sandbox



Dynamic analysis techniques

- Call Traces

- e.g., strace, ltrace

```
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

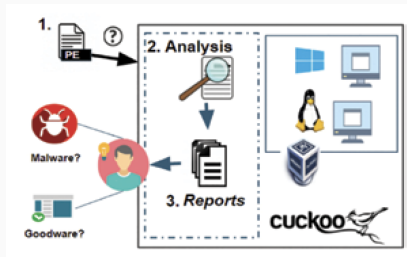
- Stop program execution
 - Control program flow
 - Examine data structures
 - Disassemble, etc...

- Memory analysis

- Dump the process' RAM and analyze it

A popular malware sandbox

- **Cuckoo Sandbox** is an advanced, modular, and automated malware analysis system, which can:
 - Analyze many different malicious files
 - Trace API calls and general behavior of the file
 - Dump and analyze network traffic
 - Perform advanced memory analysis of the infected virtualized system through Volatility and YARA



A recap on analysis methods

STATIC MALWARE VS DYNAMIC MALWARE ANALYSIS

Static Malware Analysis

vs

Dynamic Malware Analysis

Static analysis is a process of analyzing a malware binary code without actually running the code.

01

Dynamic analysis requires programs to be executed in a closely monitored virtual environment.

It uses a signature-based approach for malware analysis.

02

It uses a behavior-based approach for malware detection and analysis.

It involves file fingerprinting, virus scanning, reverse-engineering the binary, file obfuscation, analyzing memory artifacts, packer detection, & debugging.

03

Dynamic analysis involves API calls, instruction traces, registry changes, network, and system calls, memory writes, and more.

It is ineffective against sophisticated malware programs and codes.

04

It is effective against all types of malware because it analyzes the sample by executing it.

Malware analysis checklist

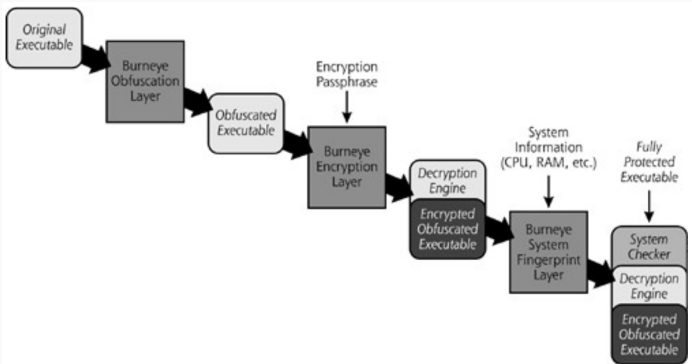
Static Analysis

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 network 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	

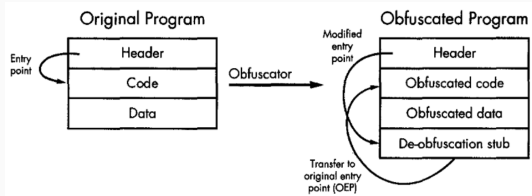
Challenges for malware analysis

- Some tools are designed to **protect binary files** and may help miscreants protect malware (e.g., BurnEye), via:
 - **Obfuscation:** scrambles the code in the executable
 - **Encryption:** encrypts the program's code
 - **Fingerprint matching:** only runs on certain computers
- Skype was known for applying some of these techniques



Anti-static analysis techniques

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



Anti-static analysis techniques (cont.)

- **Obfuscate opcodes:**
 - Encode or encrypt the actual instructions when the executable file is being created (self modification)
- **Obfuscate imported functions:**
 - To avoid leaking information about potential actions that a binary may perform, make it difficult for investigators 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
- **Detecting instrumentation** (Sysinternals tools, etc.)
 - Check loaded drivers, scan active processes or windows titles
- **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
 - 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

- **Preventing debugging**

- Intentionally generate various exceptions when a SEH (Structured Exception Handler) is set
- Confuse the debugger by introducing spurious breakpoints, clearing hardware breakpoints, hindering selection of breakpoint addresses or preventing the debugger from attaching to a process
- Calling `GetTickCount()` at regular intervals
- Suspend threads
- And many more...

Takeaways

- 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

- **Textbook:**
 - Casey – Chapter 13.5, Luttgens – Chapter 15
- **Other resources:**
 - Blue Pill
- **Acknowledgements:**
 - Slides adapted from Nuno Santos's Forensics Cyber-Security course at Técnico Lisbon