Forensics for System Administrators

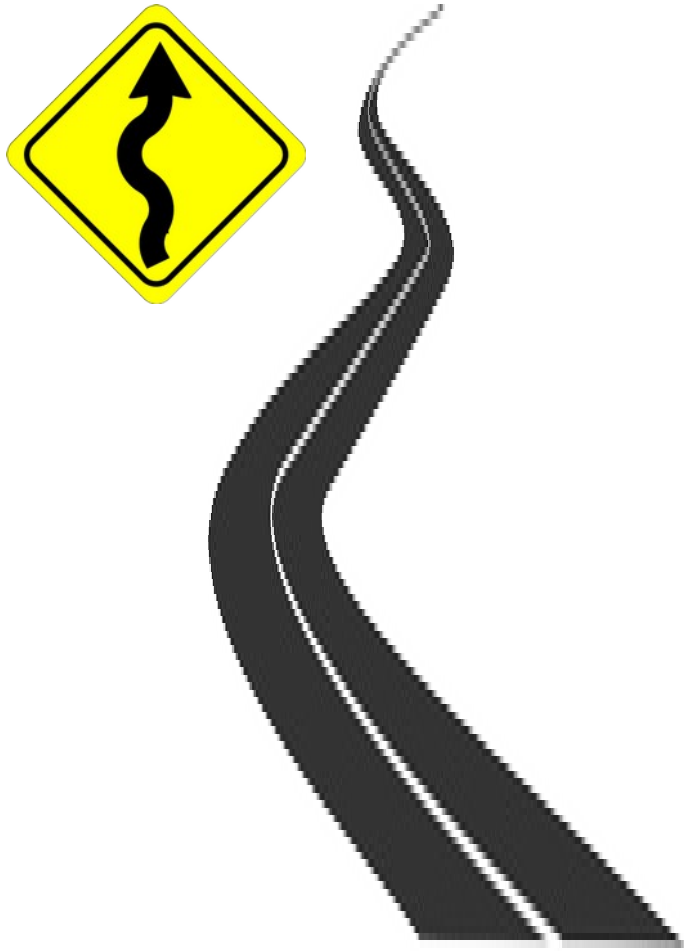
Advanced Memory Analysis - Dealing with Malicious Code

Klaus Möller
WP8-T1

Webinar, 12th of May 2022

www.geant.org

Agenda - Advanced Analysis



- Creating Timelines
- User Mode Rootkits
  - User mode rootkits
  - Injected libraries
- Kernel Mode rootkits
  - Kernel modules
  - Syscall & interrupt tables
- Malware/Code extraction
  - Searching for malware signatures
  - Extracting code from kernel
  - Extracting code from processes

Timeline Creation

www.geant.org

Timeline Creation

Live Demo

- Lots of timestamps in memory dumps
  - Process creation time (`pslist`) /termination time (`psscan`)
  - Thread creation time (`pslist`)
  - Driver/DLL/EXE compile time
  - Network socket creation time (`netstat, netscan, sockets,` etc.)
  - Memory resident registry key last write time (`printkey` et. al.)
  - Memory resident event log entry creation time
- Volatility 2 & 3: **`timeliner`** plug-in
  - Collects output from all plug-ins that produce timestamps and sorts them

```
> vol.py -f mem.img timeliner
```

GÉANT
Networks • Services • People

User Mode Rootkits

www.geant.org

User Mode Rootkit

- Rootkit - Software that hides adversary presence/activities, like
  - Processes, files, directories,
  - Network connections (sockets)
  - Free disk space, etc.
  - Interfaces in promiscuous mode (i. e. sniffers)
- User Mode - All manipulations are in user space
  - System programs: **`ps`**, **`ls`**, etc.
  - System libraries: **`libc`**, **`kernel32.dll`**, etc.
  - Code injected into (every) process
  - Does not manipulate kernel data
  - Adversary usually still has/needs root privileges
  - With clean tools/libraries, still a good chance to detect them

} Easy to find with persistent storage analysis

Manipulated System Libraries

- Rootkit functions are located in a shared library/executable
- Windows
  - Executable (**`.exe`**) or library (**`.dll`**) is injected (i.e. mapped into the address space) of the infected process
  - Overwrites OS functions/pointers or is executed in a separate threat
- Linux/Unix/MacOS
  - Setting of the **`LD_PRELOAD`** environment variable
  - Entries in **`/etc/ld.so.preload`**, **`/etc/ld.so.conf`**, or **`/etc/ld.so.conf.d/*`**
  - Direct manipulation of the **`ld.so`** code (the dynamic linker)
  - Malicious functions are found before regular OS functions, and linked instead of them

Searching for Libraries

- What to look for? (from Malware Analysts Cookbook)
  - Shared libraries/DLLs with suspicious names or names that haven't been seen before
  - Shared libraries/DLLs with common names that are loaded from a non-standard directory
    - For example C:\WINDOWS**sys**\kernel32.dll (correct: **system32**)
  - Shared libraries/DLLs that allow access to protected resources or otherwise alter system security
  - Legitimate Shared libraries/DLLs that are out of context
    - I. e. libraries that are not malicious, but don't belong into certain executables, like network or security libraries in a program without network or security functionality

Example: Linux bedevil (bdvl) Rootkit

- Linux LD_PRELOAD Rootkit
  - See: `https://github.com/Error996/bdvl`
  - Hides processes with certain GID
  - Hides network connection coming from certain ports
  - Writes into `/etc/ld.so.preload` and patches the dynamic linker `ld.so`

Live Demo

Example: Finding bedevil

- Regular `libapparmor` resides in `/usr/lib64`
- Nothing should be mapped from `/lib` on a 64 Bit system
- Besides, apparmor is not active on the system
- Volatility 2/3
  - Windows: **`dlllist / windows.dlllist`**
  - Linux: **`linux_proc_maps, linux_library_list, linux_ldrmodules`**
  - Mac OS: **`mac_ldrmodules`**

Live Demo

```
Task                Pid       Load Address        Path
------------------- -------- ------------------- ----
dbus-daemon             792 0x00007ffd42585000 linux-vdso.so.1
dbus-daemon             792 0x00007f2410986000 /usr/lib64/libdbus-1.so.3
…
dbus-daemon           94779 0x00007f1d6db2e000 /lib/apparmorg/libapparmorg.so.x86_64
…
```

Kernel Mode Rootkits

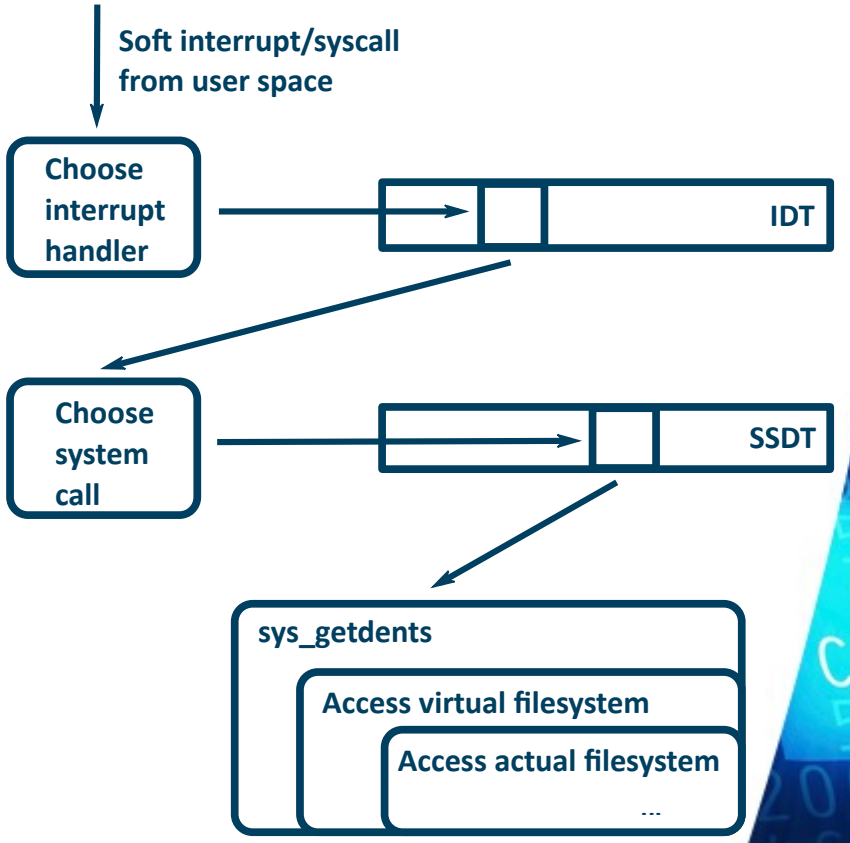
www.geant.org

Kernel Mode Rootkits

- Same functionality as user space rootkits
- But some/all code is now executed in kernel space
  - System calls return incomplete or false data
  - No way to detect kernel rootkits with clean tools/libraries
  - Not as long as the compromised kernel is running
- Memory analysis does not rely on the kernel
  - Instead, we examine the in-memory data structures directly
- Two common (among other) approaches of rootkit authors
  - Manipulating the system call or interrupt table
  - Loading a malicious kernel module/driver

System Call & Interrupt Table

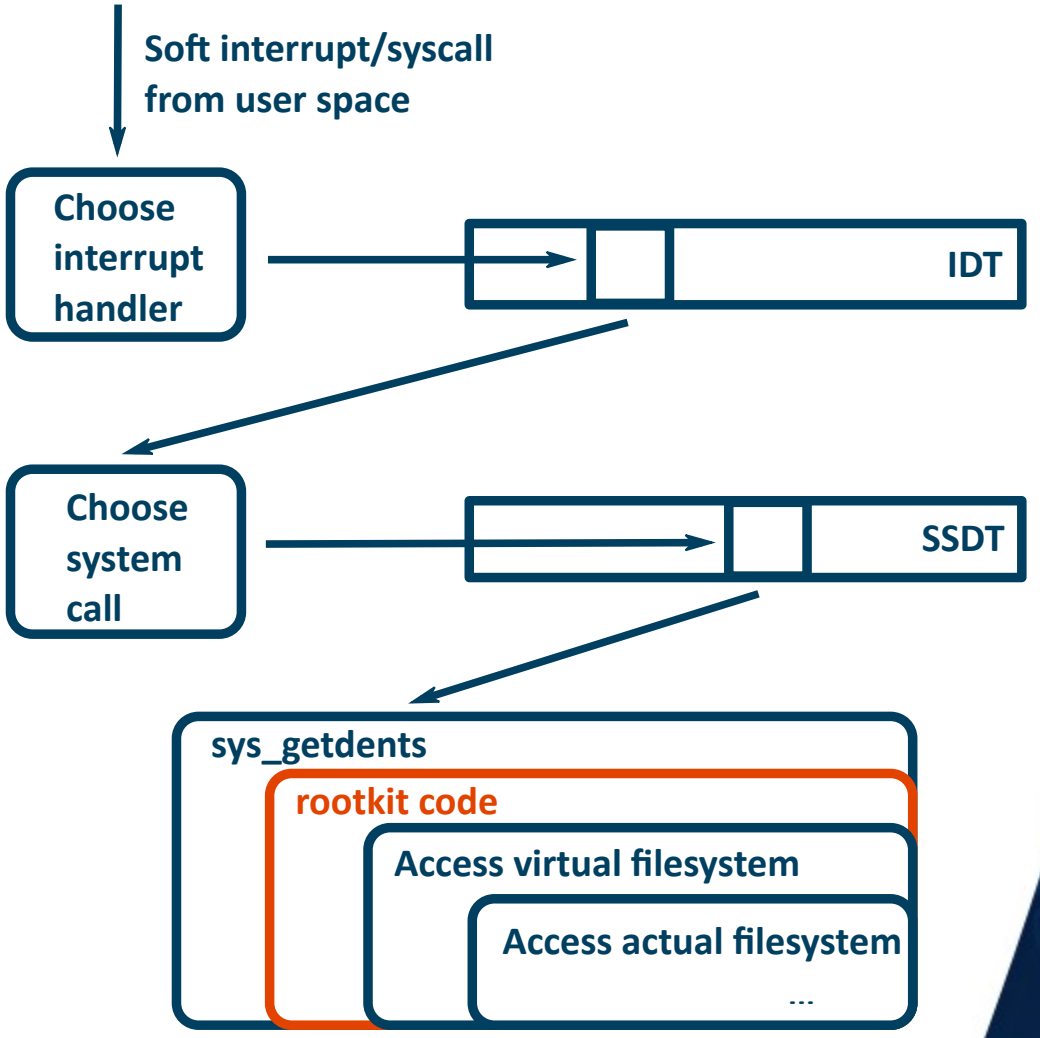
- Interrupt descriptor table (IDT) (Intel)
  - Table indexed by processor interrupt number
  - Entries are basically pointers to functions that are called when the interrupt is raised
- Service Descriptor (or System Call) Table (SSDT) (Windows/Linux)
  - Program code can raise interrupts, usually through special instruction
    - Called *soft interrupts* to differentiate from hardware generated ones (disks, keyboard, mouse, etc.)
  - Supplied with this interrupt is a number, used to look up in the SSDT
  - SSDT is indexed by this *system call number*
  - Entries are pointers to the system call functions
- Rootkits manipulate this lookup process to hide



Source: Andreas Bunten: *Unix und Linux Kernel-based Rootkits* (2004)

Direct manipulation of the system call code

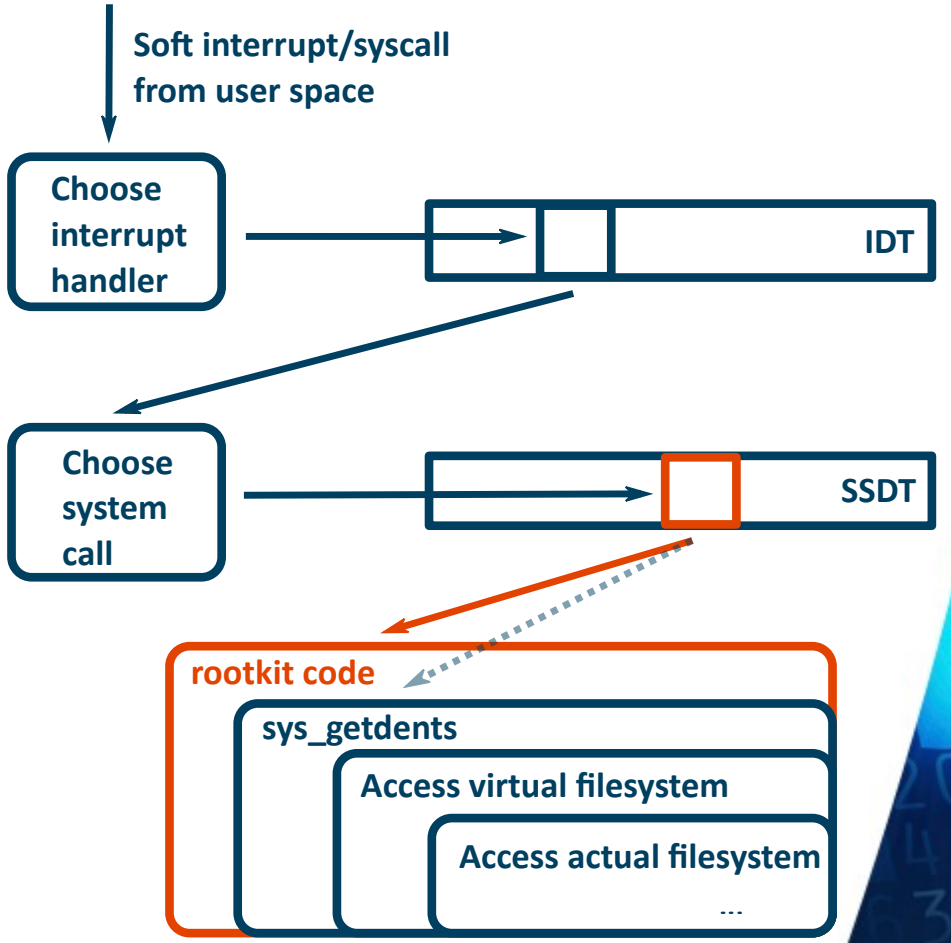
- IDT and SSDT entries and base addresses unchanged
- Syscall/Interrupt handler code unchanged
- Rootkit code is inserted into the actual syscall function



Source: Andreas Bunten: *Unix und Linux Kernel-based Rootkits* (2004)

Replace SSDT entry

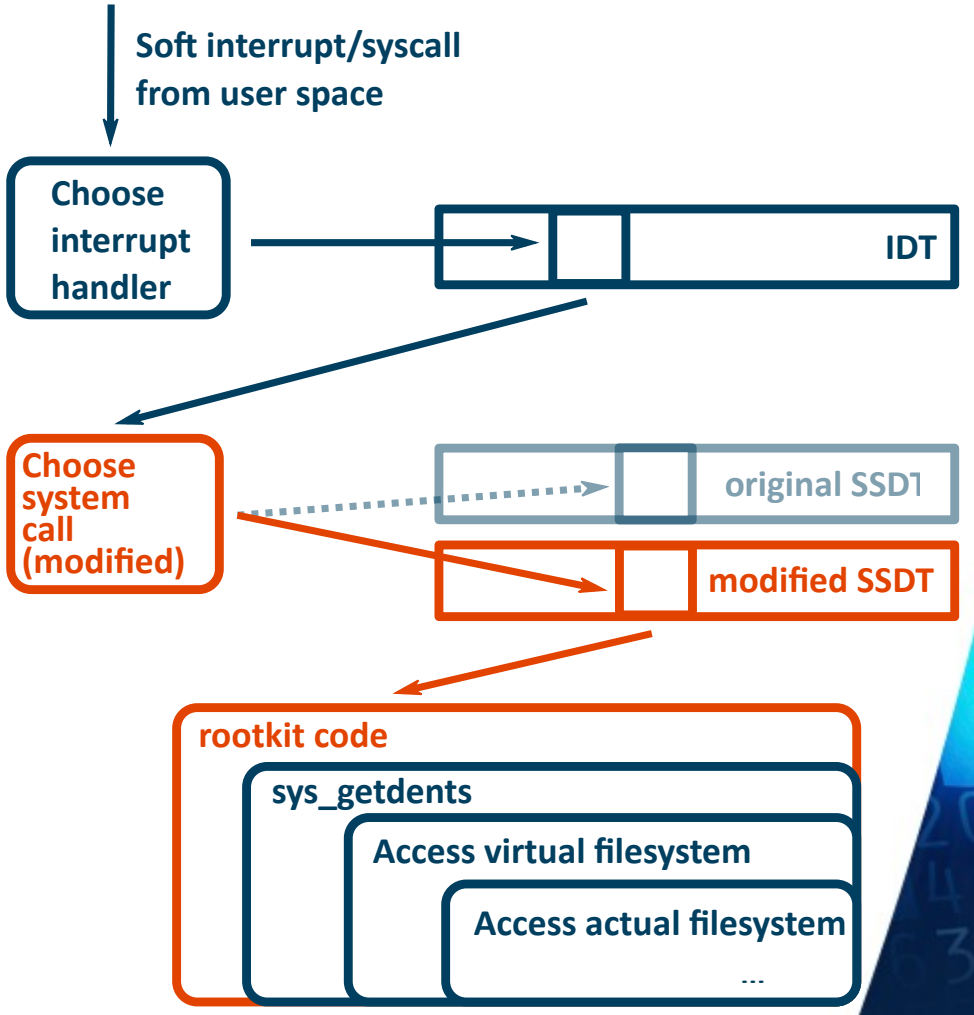
- Replace SSDT entry with a pointer to a function from the rootkit
  - May in turn call the original system call function, rootkit works than as a wrapper around the original system call code
  - Entry may stand out, when it points into another memory region
- IDT and soft interrupt handler unchanged
- Base SSDT address unchanged



Source: Andreas Bunten: *Unix und Linux Kernel-based Rootkits* (2004)

Modify Soft Interrupt Handler

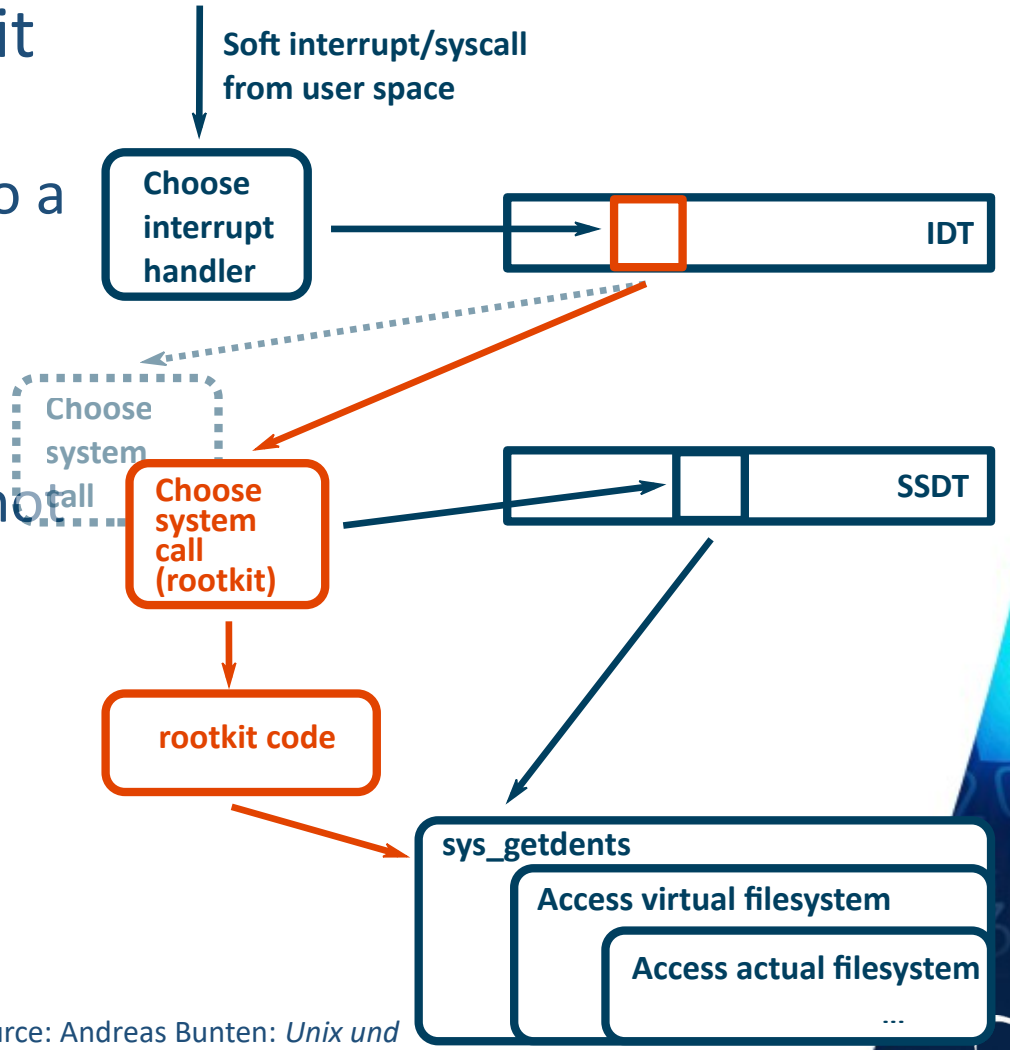
- Rootkit interrupt handler uses its own SSDT with manipulated functions
- Entries in the original SSDT are looking homogenous - in fact the SSDT itself is not changed
- Pointers from the rootkits SSDT still point to rootkit wrappers
- Search has to inspect the soft interrupt handler code to find rootkit
- Or to find the modified copy of the SSDT and compare to original



Source: Andreas Bunten: *Unix und Linux Kernel-based Rootkits* (2004)

Replace IDT Soft Interrupt Handler Entry

- Soft interrupt entry points to rootkit handler
  - Usually stands out as it may point into a different memory region than other interrupt handlers
- Rootkit handler uses original SSDT
  - Thus, SSDT and syscall functions are not manipulated (and look genuine)
- But bypasses SSDT for hooked functions
  - These rootkit functions may call the functions from the unmodified SSDT



Source: Andreas Bunten: *Unix und Linux Kernel-based Rootkits* (2004)

Volatility IDT/SSDT Plugins

- Plugins to inspect the IDT and SSDT (or syscall table)
  - Windows: **`idt, ssdt / windows.ssdt`**
  - Linux: **`linux_idt, linux_check_evt_arm, linux_syscall, linux_syscall_arm / linux.check_idt linux.check_syscall`**
  - Mac OS: **`mac_idt, mac_syscalls, mac_check_syscall_shadow / mac.check_syscall`**
- Critical functions (page faults, etc.) should always be handled by the core OS or standard modules
- Lots of more function pointer tables in the OS that can be used by rootkits
  - And more plugins to find them

Kernel Modules & Rootkits

- Kernel rootkit functions often bundled into a kernel module
  - Installation uses standard module loading mechanisms of the OS
- Often user space process to communicate with the kernel parts
  - Socket/handle between process and kernel module
  - User space process can be detected (sometimes not hidden)
- To list loaded kernel modules or drivers (Volatility 2/3)
  - Linux: **`linux_lsmod / linux.lsmod`**
  - Windows: **`modules, modscan, driverscan, driverirp / windows.modules, windows.modscan, windows.driverscan, windows.driverirp`**
  - Mac OS: **`mac_lsmod / mac.lsmod`**

Example: Windows Laqma rootkit

Live Demo

- Rootkit installs **`lanmandrv.sys`**
  - Path suspicious: `\windows\system32` instead of `\windows\system32\drivers`
  - Genuine LANMAN driver is called `lanman.drv`
  - Towards the end/beginning of the list, may be a sign of later loading of the driver/module

```
noob@v2os153:~/volatility> ./vol.py -f img/laqma.vmem modules
Volatility Foundation Volatility Framework 2.6.1
Offset(V)  Name                 Base             Size File
---------- -------------------- ---------- ---------- ----
0x810dbe68 ntoskrnl.exe         0x804d7000   0x1f6280 \WINDOWS\system32\ntkrnlpa.exe
0x810dbe00 hal.dll              0x806ce000    0x20380 \WINDOWS\system32\hal.dll
0x810dbd98 kdcom.dll            0xfc99b000     0x2000 \WINDOWS\system32\KDCOM.DLL
0x810dbd28 BOOTVID.dll          0xfc8ab000     0x3000 \WINDOWS\system32\BOOTVID.dll
0x810dbcc0 ACPI.sys             0xfc36c000    0x2e000 ACPI.sys
…
0xff381bd8 sysaudio.sys         0xf337d000     0xf000 \SystemRoot\system32\drivers\sysaudio.sys
0xff2837e8 Fastfat.SYS          0xf2ef5000    0x23000 \SystemRoot\System32\Drivers\Fastfat.SYS
0x80fae8b0 lanmandrv.sys        0xfca29000     0x2000 \??\C:\WINDOWS\System32\lanmandrv.sys
0xff147970 kmixer.sys           0xf2ecb000    0x2a000 \SystemRoot\system32\drivers\kmixer.sys
noob@v2os153:~/volatility>
```

Malware

www.geant.org

Searching for Malware Signatures

- VAD (Virtual Address Descriptor) are descriptor structures for memory regions in Windows
- **`Malfind`** plug-in searches the VAD blocks for code that may have been injected by malware
  - An **`MZ`** header in a block is almost always suspicious as it is normally not included
  - Maybe a whole executable was injected en-block
  - Header list can be limited to “suspicious” ones with -W
    - I. e. those containing “MZ”
- Suspicious blocks can be extracted with `-dump-dir=` (or `-D`)
- Files can than be reverse engineered or scanned for malicious code
  - I. e. AV-scanner or `virustotal.com`

Example: Zeus Banking Trojan (aka Zbot)

Live Demo

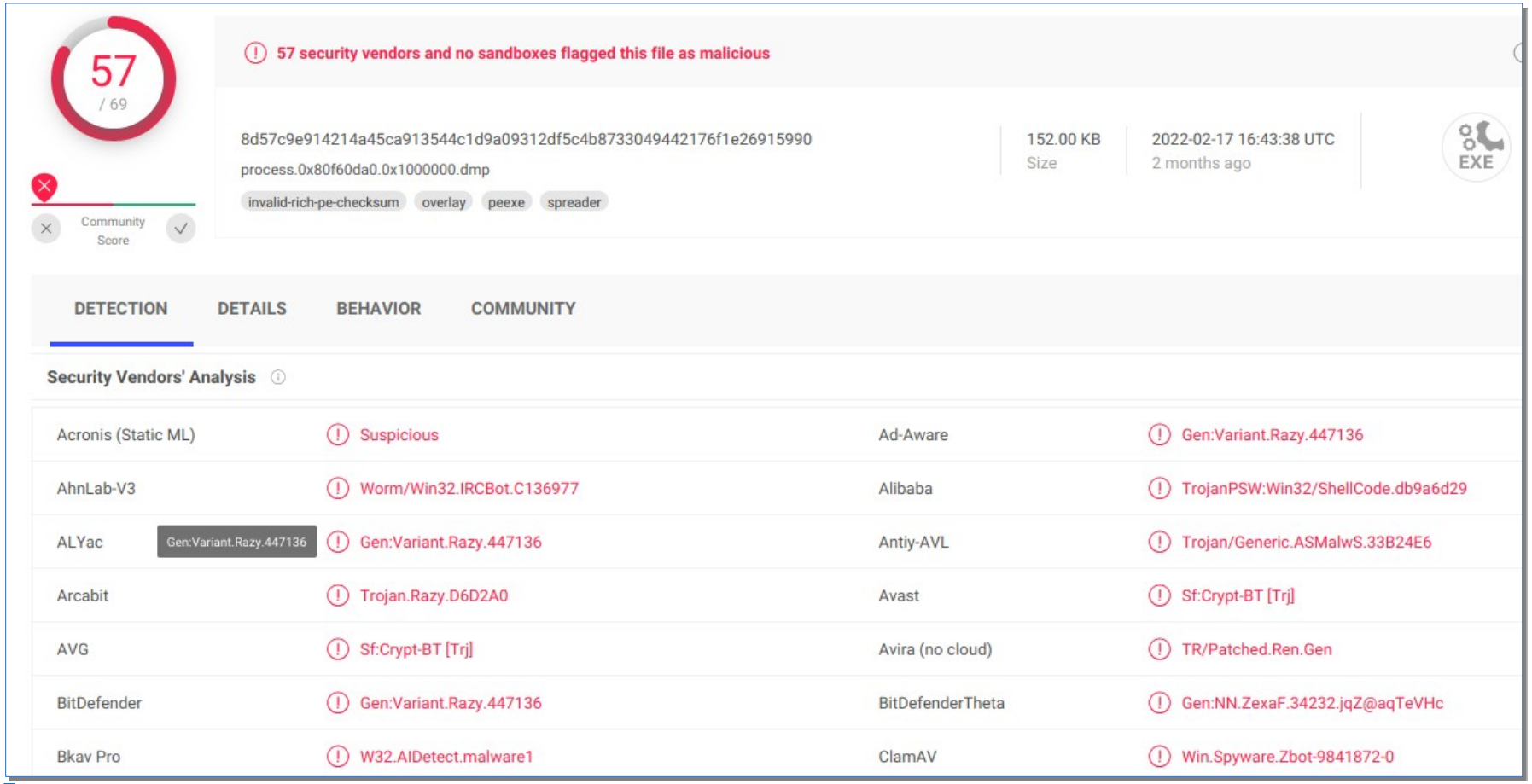
- Scanning (and dumping) the virtual memory image

```
> ./vol.py -f img/zeus.vmem malfind -W -D dumpdir
> cd dumpdir; ls -l
noob@v2os153:~/volatility/dumpdir> ls -l
insgesamt 3648
-rw-r--r-- 1 noob users 155648  9. Mai 20:34 process.0x80f60da0.0x1000000.dmp
-rw-r--r-- 1 noob users 155648  9. Mai 20:34 process.0x80f94588.0x12d0000.dmp
…
> sha1sum process.0x80f60da0.0x1000000.dmp
a2c13b59832d1cdb25ba2f6ba8f6cacd044c51a8  process.0x80f60da0.0x1000000.dmp
```

Example: Zeus Banking Trojan (virustotal upload)

Live Demo

- Surprise: It’s Zbot;)



Code Extraction from Kernel

Live Demo

- Volatility plug-ins to dump kernel modules (V2 only)
  - Windows: **moddump**
  - Linux: **linux_moddump**
  - Mac OS: **mac_moddump**
- Use base address from `modules` or `*_lsmod` plug-ins for `-b` option
  - Without, all modules will be dumped

```
> ./vol.py -f img/laqma.vmem moddump -b 0xfca29000 -D .
Volatility Foundation Volatility Framework 2.6.1
Module Base Module Name                  Result
---------- -------------------- ------
0x0fca29000 lanmandrv.sys        OK: driver.fca29000.sys
> sha1sum driver.fca29000.sys
01e53955f889fb317cd4981d03c622d8b60dc2f5  driver.fca29000.sys
```

Code Extraction from Processes

Live Demo

- Plugins to dump process memory to file(V2 only)
  - Windows: **procdump**
  - Linux: **linux_procdump**
  - Mac OS: **mac_procdump**

```
> ./vol.py -f img/laqma.vmem pslist | grep lanm
Volatility Foundation Volatility Framework 2.6.1
0xff3825f8 lanmanwrk.exe          1180   1060      2       75      0      0
2010-08-15 19:09:12 UTC+0000
> ./vol.py -f img/laqma.vmem procdump -p 1180 -D procdumpdir/
Volatility Foundation Volatility Framework 2.6.1
Process(V) ImageBase  Name                 Result
---------- ---------- -------------------- ------
0xff3825f8 0x00400000 lanmanwrk.exe        OK: executable.1180.exe
noob@v2os153:~/volatility> sha1sum procdumpdir/executable.1180.exe
d84f38096f040b167919bb8a639837305bf690bf  procdumpdir/executable.1180.exe
```

Wrapping Up

www.geant.org

What's Next?

- We've seen only the very beginning
  - Lots of more plug-ins for lots of more ways for adversaries to hide
  - Read the volatility blog, documentation, books, etc.
- Practice, practice, practice
  - Try analysing some of the sample images with real malware, or do some memory acquisition from your own systems
    - See references section
    - For production systems, ask for permission first!
- Don't forget: There are more ways/tools for forensic analysis
  - Persistent storage analysis (Autopsy, next webinars)
  - Network data analysis (Wireshark)
  - Reverse engineering (Ghidra, Radare 2)
  - And more, … even very simple ones (strings, objdump, …)

Finally

- The situation is hopeless - but not serious;)
- For every attack, there's a defence (and vice versa:(
- There will always be traces the adversary forgot, couldn't avoid or did not care about

Seek and ye shall find!

Thank you

Any questions?

Next Webinar: ***Persistent Storage Forensics I***

May 25th, 2022

www.geant.org

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement No. 731122 (GN4-2).