MALWARE EVASION

MALWARE ANALYSIS AND INCIDENT FORENSICS

M.Sc. in Cyber Security

MALWARE ANALYSIS

M.Sc. in Engineering in Computer Science

A.Y. 2025/2026

SAPIENZA
UNIVERSITÀ DI ROMA

CIS SAPIENZA
CYBER INTELLIGENCE AND INFORMATION SECURITY

MALWARE EVASION

- Staying undetected is vital for malware
- Dynamic analysis is necessary to face new samples surfacing daily
  - static analyses have limited capabilities
  - externally observable behavior is crucial for threat identification & classification
  - only few samples (e.g., new strains) undergo manual analysis
- If it can detect an execution or environmental **artifact**, malware can deceive even the most sophisticated analysis system

DISCREPANCIES

- What makes an analysis system different from a victim machine?
  - virtualization artifacts
  - hardware characteristics
  - Windows installation
  - applications
  - user artifacts
  - time overheads and memory footprint

VIRTUALIZATION ARTIFACTS

- A **red pill** is an instruction sequence that detects whether the code is running in a virtual machine
  - instructions that take longer to execute (mainly `cpuid`)
  - instruction errata (e.g., incomplete emulation)
  - information
    - CPU identification data from `cpuid`
    - system firmware tables (e.g., SMBIOS strings, ACPI tables)
    - contents and position in memory of specific structures (e.g., Interrupt Descriptor Table)
    - micro-architectural state of the machine
  - I/O ports

EXAMPLE: CPUID

- `cpuid` retrieves processor identification and feature information
  - writes an information record to {EAX, EBC, ECX, EDX}
  - record type chosen according to the value read in EAX
  - for EAX=0x1 it writes feature information in ECX

    This will reveal the presence of a hypervisor when the 31st bit is equal to 1
  - for EAX=0x40000000 it writes hypervisor brand information in {EBX, ECX, EDX} as ASCII bytes forming 12-byte strings like:
    - `"KVMKVMKVM\0\0\0"`
    - `"Microsoft Hv"`
    - `"VMwareVMware"`
    - `"XenVMMXenVMM"`
    - `"prl hyperv  "`
    - `"VboxVboxVbox"`

HYPERVISOR BIT

Virtualization check: set EAX=1 and inspect 31st bit of ECX

```
size_t eax, ebx, ecx, edx;
__asm__ volatile (
   "cpuid"
   : "=a"(eax), "=b"(ebx), "=c"(ecx), “=d"(edx)
   : "a"(1)
   : "memory" );
printf("%d\n", (int)(ecx >> 31));
```

HYPERVISOR BRAND

```
#include "stdafx.h"
#include <iostream>
#include <string>

using namespace std;


int main()
{
 bool IsUnderVM = false;

 __asm {
 xor     eax, eax
 mov     eax,0x40000000
 cpuid
 cmp ecx,0x4D566572
 jne NopInstr
 cmp edx,0x65726177
 jne NopInstr
 mov IsUnderVM, 0x1
 NopInstr:
 nop
 }
 cout << IsUnderVM;
 return 0;
}
```

“VMwareVMware”

https://rayanfam.com/topics/defeating-malware-anti-vm-techniques-cpuid-based-instructions/

EXECUTION TIME FOR CPUID

Under a hypervisor, cpuid causes a **VM exit** event and the virtual machine monitor component of the hypervisor kicks in. We can measure this latency:

```
int elapsed;
__asm__ volatile (
    "mov %%eax, 1;”
"rdtsc ;"
"mov %0, %%eax ;"
"cpuid ;"
"rdtsc ; "
"sub %0, %%eax ;"
"neg %0 ;"
: "=r"(elapsed) :
: "rax", "rbx", "rcx", "rdx");
```

AL-KHASER: CPUID LATENCY

```
/*
CPUID is an instruction which cauz a VM Exit to the VMM,
this little overhead can show the presence of a hypervisor
*/

BOOL rdtsc_diff_vmexit()
{
    ULONGLONG tsc1 = 0;
    ULONGLONG tsc2 = 0;
    ULONGLONG avg = 0;
    INT cpuInfo[4] = {};

    // Try this 10 times in case of small fluctuations
    for (INT i = 0; i < 10; i++)
    {
        tsc1 = __rdtsc();
        __cpuid(cpuInfo, 0);
        tsc2 = __rdtsc();

        // Get the delta of the two RDTSC
        avg += (tsc2 - tsc1);
    }

    // We repeated the process 10 times so we make sure our check is as much reliable as we can
    avg = avg / 10;
    return (avg < 1000 && avg > 0) ? FALSE : TRUE;
}
```

HARDWARE CHARACTERISTICS

- Some hardware details can reveal a VM
  - CPU model and cores
  - Number of network adapters, MAC address family
  - Disk size and serial number
  - Peripherals (e.g., audio capabilities)
  - Sensor measurements (e.g., fan speed, temperature)
- These features are obviously relevant for **evasive** malware
- **Targeted** malware may inspect some of the properties above to check whether it has reached an intended victim

EXAMPLE: MAC ADDRESS

- Some families of MAC addresses from hypervisors:
  - `"\x08\x00\x27"` (VirtualBox)
  - `"\x00\x05\x69"` (VMware)
  - `"\x00\x0C\x29"` (VMware)
  - `"\x00\x1C\x14"` (VMware)
  - `"\x00\x50\x56"` (VMware)
  - `"\x00\x1C\x42"` (Parallels)
  - `"\x00\x16\x3E"` (Xen)

EXAMPLE: SPEAKER CHECK

```
{
    wchar_t* filterName = L"random_name";

    IGraphBuilder *pGraph;
    CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC_SERVER, IID_IGraphBuilder, (void**)&pGraph);
    if (E_POINTER != pGraph->AddFilter(NULL, filterName))
        ExitProcess(-1);

    IBaseFilter* pBaseFilter;
    CoCreateInstance(CLSID_AudioRender, NULL, CLSCTX_INPROC_SERVER, IID_IBaseFilter, (void**)&pBaseFilter);

    // in case of no audio device, CoCreateInstance returns VFW_E_NO_AUDIO_HARDWARE 0x80040256
    // but the result is not checked and it fails later

    pGraph->AddFilter(pBaseFilter, filterName);

    IBaseFilter* pBaseFilter2;
    pGraph->FindFilterByName(filterName, &pBaseFilter2);
    if (NULL == pBaseFilter2)
        ExitProcess(1);

    FILTER_INFO info = { 0 };
    pBaseFilter2->QueryFilterInfo(&info);
    if (0 != wcscmp(info.achName, filterName))
        return;

    IReferenceClock* pClock;
    if (0 != pBaseFilter2->GetSyncSource(&pClock))
        return;

    if (0 != pClock)
        return;

    CLSID clsID;
    pBaseFilter2->GetClassID(&clsID);
    if (clsID.Data1 == 0)
        exit(1);

    if (NULL == pBaseFilter2)
        exit(-1);

    IEnumPins *pEnum = NULL;
    if (0 != pBaseFilter2->EnumPins(&pEnum))
        exit(-1);

    if (0 == pBaseFilter2->AddRef())
        exit(-1);
}
```

The TeslaCrypt ransomware makes a COM-based DirectShow audio check: not only for device presence, but also invokes APIs like AddFilter to expose inaccurate emulation of the device by a malware sandbox system.

Source: https://www.joesecurity.org/blog/6933341622592617830

WINDOWS INSTALLATION

- Context information:
  - timezone
  - language
  - uptime
  - install date
  - product keys
- Hypervisors and their guest additions also leave traces:
  - registry entries
  - processes
  - drivers

EXAMPLE: VBOX & REGISTRY

- Key values (HKEY_LOCAL_MACHINE):
  - `HARDWARE\DEVICEMAP\Scsi\Scsi Port 0\Scsi Bus 0\Target Id 0\Logical Unit Id 0 => Identifier = "VBOX"`
  - `HARDWARE\Description\System => SystemBiosVersion = "VBOX"`
  - `HARDWARE\Description\System => VideoBiosVersion = "VIRTUALBOX"`
  - `HARDWARE\Description\System => SystemBiosDate = "06/23/99"`
- Keys (HKEY_LOCAL_MACHINE):
  - `HARDWARE\ACPI\DSDT\VBOX__`
  - `HARDWARE\ACPI\FADT\VBOX__`
  - `HARDWARE\ACPI\RSDT\VBOX__`
  - `SOFTWARE\Oracle\VirtualBox Guest Additions`
  - `SYSTEM\ControlSet001\Services\VBoxGuest`
  - `SYSTEM\ControlSet001\Services\VBoxMouse`
  - `SYSTEM\ControlSet001\Services\VBoxService`
  - `SYSTEM\ControlSet001\Services\VBoxSF`
  - `SYSTEM\ControlSet001\Services\VBoxVideo`

AL-KHASER: VBOX REGISTRY KEYS

```
VOID vbox_reg_keys()
{
    /* Array of strings of blacklisted registry keys */
    const TCHAR* szKeys[] = {
        _T("HARDWARE\\ACPI\\DSDT\\VBOX__"),
        _T("HARDWARE\\ACPI\\FADT\\VBOX__"),
        _T("HARDWARE\\ACPI\\RSDT\\VBOX__"),
        _T("SOFTWARE\\Oracle\\VirtualBox Guest Additions"),
        _T("SYSTEM\\ControlSet001\\Services\\VBoxGuest"),
        _T("SYSTEM\\ControlSet001\\Services\\VBoxMouse"),
        _T("SYSTEM\\ControlSet001\\Services\\VBoxService"),
        _T("SYSTEM\\ControlSet001\\Services\\VBoxSF"),
        _T("SYSTEM\\ControlSet001\\Services\\VBoxVideo")
    };

    WORD dwlength = sizeof(szKeys) / sizeof(szKeys[0]);

    /* Check one by one */
    for (int i = 0; i < dwlength; i++)
    {
        TCHAR msg[256] = _T("");
        _stprintf_s(msg, sizeof(msg) / sizeof(TCHAR), _T("Checking reg key %s "), szKeys[i]);
        if (Is_RegKeyExists(HKEY_LOCAL_MACHINE, szKeys[i]))
            print_results(TRUE, msg);
        else
            print_results(FALSE, msg);
    }
}
```

APPLICATIONS

- The presence of a software program may be
    1. a necessary condition to trigger a payload
    2. an adversary to disarm (e.g., anti-virus products)
    3. a sufficient condition for evasion (e.g., an analysis tool like IDA is found)
- Analysts test malware using multiple virtual machine images, each containing different applications and versions

AL-KHASER: APPLICATIONS

```
const TCHAR *szProcesses[] = {
    _T("ollydbg.exe"),          // OllyDebug debugger
    _T("ProcessHacker.exe"),    // Process Hacker
    _T("tcpview.exe"),          // Part of Sysinternals Suite
    _T("autoruns.exe"),         // Part of Sysinternals Suite
    _T("autorunsc.exe"),        // Part of Sysinternals Suite
    _T("filemon.exe"),          // Part of Sysinternals Suite
    _T("procmon.exe"),          // Part of Sysinternals Suite
    _T("regmon.exe"),           // Part of Sysinternals Suite
    _T("procexp.exe"),          // Part of Sysinternals Suite
    _T("idaq.exe"),             // IDA Pro Interactive Disassembler
    _T("idaq64.exe"),           // IDA Pro Interactive Disassembler
    _T("ImmunityDebugger.exe"), // ImmunityDebugger
    _T("Wireshark.exe"),        // Wireshark packet sniffer
    _T("dumpcap.exe"),          // Network traffic dump tool
    _T("HookExplorer.exe"),     // Find various types of runtime hooks
    _T("ImportREC.exe"),        // Import Reconstructor
    _T("PETools.exe"),          // PE Tool
    _T("LordPE.exe"),           // LordPE
    _T("SysInspector.exe"),     // ESET SysInspector
    _T("proc_analyzer.exe"),    // Part of SysAnalyzer iDefense
    _T("sysAnalyzer.exe"),      // Part of SysAnalyzer iDefense
    _T("sniff_hit.exe"),        // Part of SysAnalyzer iDefense
    _T("windbg.exe"),           // Microsoft WinDbg
    _T("joeboxcontrol.exe"),    // Part of Joe Sandbox
    _T("joeboxserver.exe"),     // Part of Joe Sandbox
    _T("joeboxserver.exe"),     // Part of Joe Sandbox
    _T("ResourceHacker.exe"),   // Resource Hacker
    _T("x32dbg.exe"),           // x32dbg
    _T("x64dbg.exe"),           // x64dbg
    _T("Fiddler.exe"),          // Fiddler
    _T("httpdebugger.exe"),     // Http Debugger
};
```

Al-Khaser checks here for common analysis tools and debuggers

The **Furtim** malware first detects and bypasses a wide range of anti-virus products. Then, it searches for 29 tools that analysts launch manually: but instead of exiting immediately in their presence, it delays termination until a later stage of the fingerprinting process, intentionally causing frustration for analysts 😤

Reference: https://www.sentinelone.com/blog/sfg-furtims-parent/

USER ARTIFACTS

- Fresh Windows installations are suspicious
  - recently opened files
  - navigation history
  - applications and their install date
  - recently connected USB devices
  - number and type of files in Documents and Desktop
- VM/sandbox images shall exhibit realistic **wear-and-tear** state

WHAT COULD ONE MONITOR?

There is no such thing as a transparent sandbox

- We may learn from what imperfections samples look for, and either fix these issues or come up with designs unaffected by them
- How do you spot an evasive sequence?
  - special instructions: e.g., `cpuid`, `int`, `rdtsc`
  - library calls: files, registry, GUI events, hardware features, drivers, processes, pipes, DLL handling, network, IPC objects, time sources
  - system calls: sample "talks" to the kernel directly
  - WMI: queries to OS through Windows Management Instrumentation system
  - process environment: data structures like the PEB (Process Entry Block) may expose relevant information (e.g., CPU cores, presence of debuggers)

ANTI-DEBUGGING 101

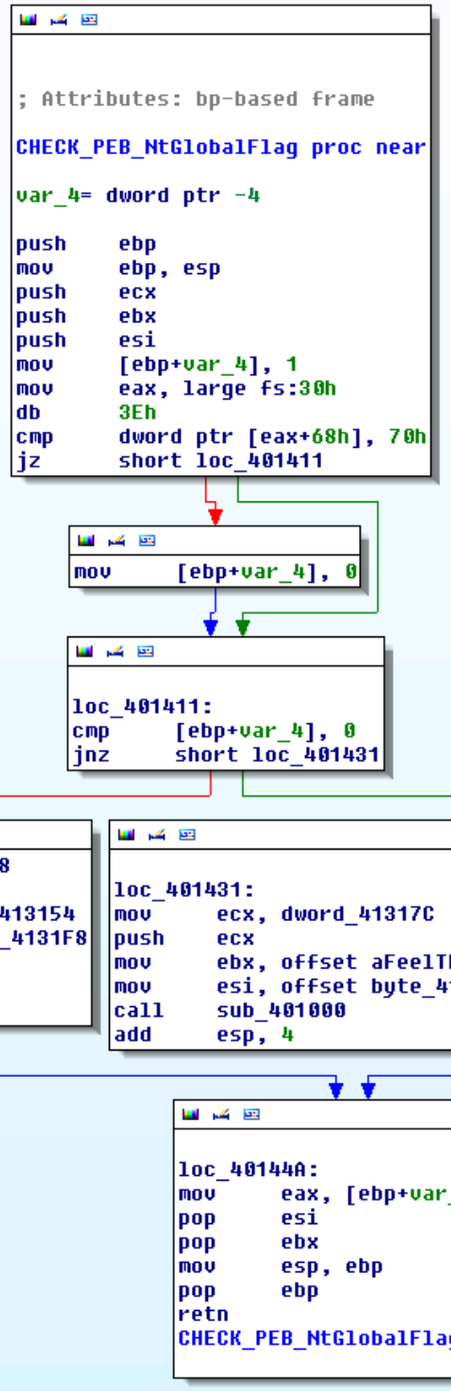
- The presence of a debugger alters the normal working of a process. The OS reflects such effect, both directly and indirectly
  - library functions: IsDebuggerPresent(), CheckRemoteDebuggerPresent()
  - system calls (e.g., NtQueryInformationProcess)
  - PEB flags
    - BeingDebugged (checked by IsDebuggerPresent)
    - NtGlobalFlag
- `int 2d` and other low-level tricks (e.g., `sti`)
- API semantics (e.g., CloseHandle on an invalid handle)
- Peter Ferrie wrote a guide with ~80 red pills to expose debuggers

NTGLOBALFLAG CHECK

```
#elif defined(ENV32BIT)
    /* NtGlobalFlags for real 32-bits OS */
    BYTE* _teb32 = (BYTE*)__readfsdword(0x18);
    DWORD _peb32 = *(DWORD*)(_teb32 + 0x30);
    pNtGlobalFlag = (PDWORD)(_peb32 + 0x68);

    if (IsWoW64())
    {
        /* In Wow64, there is a separate PEB for the 32-bit portion and the 64-bit portion
        which we can double-check */

        BYTE* _teb64 = (BYTE*)__readfsdword(0x18) - 0x2000;
        DWORD64 _peb64 = *(DWORD64*)(_teb64 + 0x60);
        pNtGlobalFlagWoW64 = (PDWORD)(_peb64 + 0xBC);
    }
#endif

    BOOL normalDetected = pNtGlobalFlag && *pNtGlobalFlag & 0x00000070;
    BOOL wow64Detected = pNtGlobalFlagWoW64 && *pNtGlobalFlagWoW64 & 0x00000070;

    if(normalDetected || wow64Detected)
        return TRUE;
    else
        return FALSE;
```

Al-Khaser: check NtGlobalFlag on 32-bit PEB (and also 64-bit PEB)

NTGLOBALFLAG CHECK



Taken from Challenge 7 in Flare On series 2014: the code checks NtGlobalFlag against 0x70.
Windows typically sets this value as a combination of multiple constants, and should be 0 when a debugger is not present. Compared to the C example from before, the code accesses the 32-bit PEB directly
(but does not check the Wow64 PEB)

Source: https://www.aldeid.com/wiki/PEB-Process-Environment-Block/NtGlobalFlag

AL-KHASER: CLOSEHANDLE

```
/*
APIs making user of the ZwClose syscall (such as CloseHandle, indirectly)
can be used to detect a debugger. When a process is debugged, calling ZwClose
with an invalid handle will generate a STATUS_INVALID_HANDLE (0xC0000008) exception.
As with all anti-debugs that rely on information made directly available.
*/
BOOL NtClose_InvalideHandle()
{
    auto NtClose_ = static_cast<pNtClose>(API::GetAPI(API_IDENTIFIER::API_NtClose));

    __try {
        NtClose_(reinterpret_cast<HANDLE>(0x99999999ULL));
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        return TRUE;
    }

    return FALSE;

}

BOOL CloseHandle_InvalideHandle()
{
    // Let's try first with user mode API: CloseHandle
    __try {
        CloseHandle(reinterpret_cast<HANDLE>(0x99999999ULL));
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        return TRUE;
    }

    // Direct call to NtClose to bypass user mode hooks
    if (NtClose_InvalideHandle())
        return TRUE;
    else
        return FALSE;
}
```

TIMING ATTACKS 101

- Playing with time is one of the oldest, yet still most effective ways to detect malware analysis systems
- Different options:
  - **time an instruction sequence** and compare it to some baseline
    - we already discussed red pills for hypervisors
    - the method can also detect binary instrumentation schemes
  - make the analysis run out of budget with **sleeps**
    - use multiple sleep actions (instead of just one with a large value)
    - check also if time has been accelerated (and different time sources being consistent)
    - Windows offers plenty of time sources
  - run **long computations** in the initial stages

WMI QUERIES

The Windows Management Instrumentation subsystem supports SQL-style queries to the OS about hardware and software

`SELECT * FROM` (below some examples of what can be checked)

`Win32_Bios` (SerialNumber)

`Win32_PnPEntity` (DeviceId)

`Win32_NetworkAdapterConfiguration` (MACAddress)

`Win32_Processor` (NumberOfCores, ProcessorId)

`Win32_LogicalDisk` (Size)

`Win32_ComputerSystem` (Model, Manufacturer)

`Win32_NTEventLogFile` (FileName - related to VBOX)

`MSACPI_ThermalZoneTemperature` (CurrentTemperature)

AL-KHASER: WMI WIN32_BIOS

```
// Get the value of the Name property
hRes = pclsObj->Get(_T("SerialNumber"), 0, &vtProp, 0, 0);
if (SUCCEEDED(hRes)) {
    if (vtProp.vt == VT_BSTR) {

        // Do our comparison
        if (
            (StrStrI(vtProp.bstrVal, _T("VMWare")) != 0) ||
            (wcscmp(vtProp.bstrVal, _T("0")) == 0) || // VBox (serial is just "0")
            (StrStrI(vtProp.bstrVal, _T("Xen")) != 0) ||
            (StrStrI(vtProp.bstrVal, _T("Virtual")) != 0) ||
            (StrStrI(vtProp.bstrVal, _T("A M I")) != 0)
            )
        {
            VariantClear(&vtProp);
            pclsObj->Release();
            bFound = TRUE;
            break;
        }
    }
    VariantClear(&vtProp);
}
```

Common BIOS serial numbers in hypervisors/emulators

IN-GUEST AGENTS

- We discussed two main attack surfaces: artifacts of the execution technology (imperfections, overheads) and of the software setup
- However, for sandboxes and general dynamic analysis systems, operating **inside** a virtualized guest is **already** a weakness
  - think of monitoring agents implemented in user space, often as an injected DLL
  - agents implemented in kernel space are more rare now (due to PatchGuard)
- We can always study fingerprinting techniques and fix/randomize the implementation accordingly but, in the end, these systems will always share the execution context with the sample under analysis

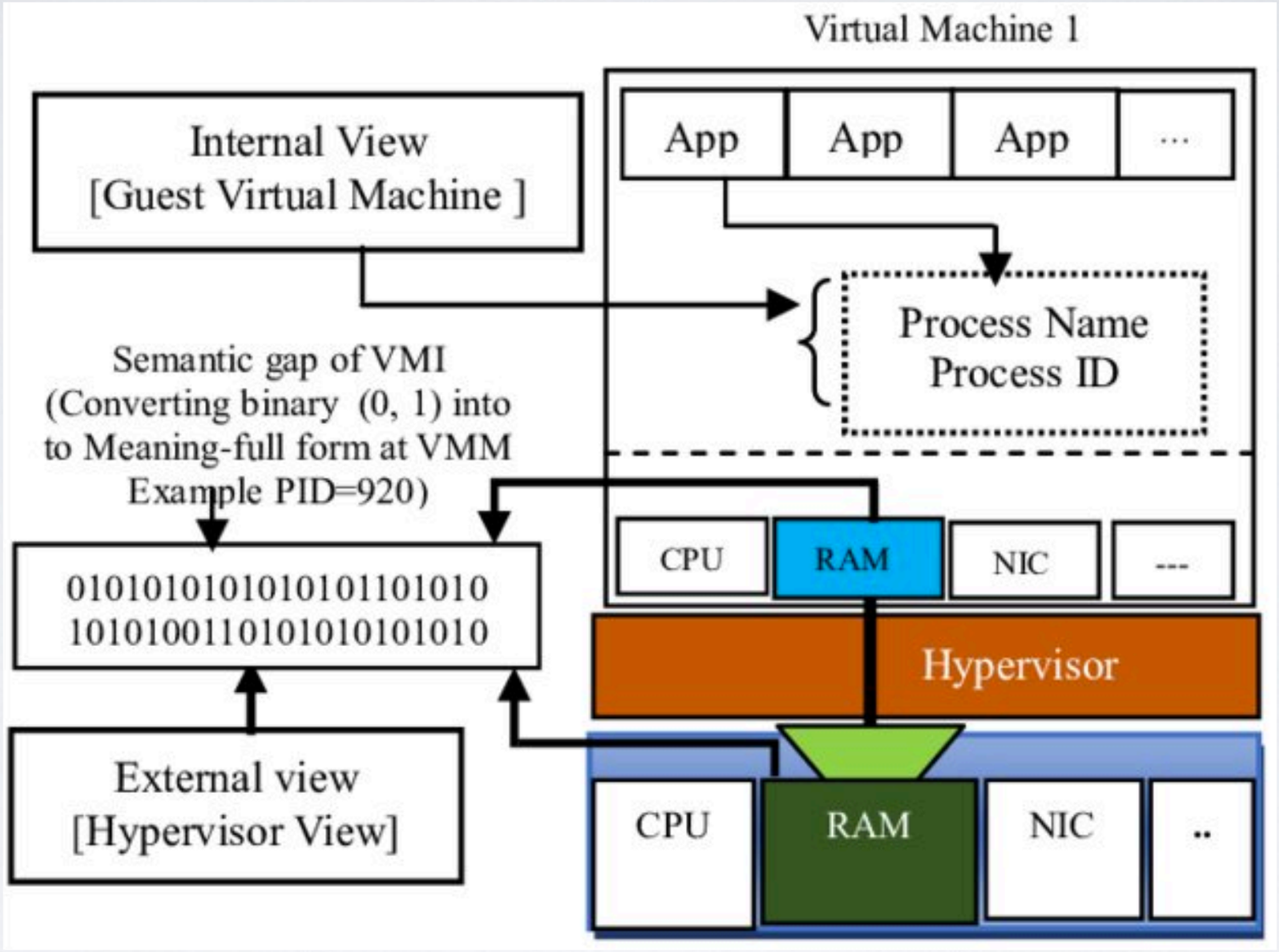
VIRTUAL MACHINE INTROSPECTION

- Idea: perform the analysis from outside a guest VM
  - pioneered in IDS systems for greater attack resistance while retaining visibility
  - requires cooperation from the hypervisor's VM monitor
- Three capabilities are crucial to support good visibility while offering resistance to evasion and subversion:
  - isolation: program cannot access code/state of the monitoring system
  - inspection: monitoring system has full access to the state of analyzed machine
  - interposition: monitoring system can interpose on specific operations, such as uses of privileged instructions (e.g., when issuing a system call)

ETHER [CCS'08]

- The Ether malware analyzer pioneered the use of the VT-x virtualization extensions to build a transparent analysis system for malware
- The analyzer resides in the VM monitor
- Its authors identified 5 requirements for a system that wants to hide memory and CPU changes inevitably caused by its very presence
  - higher privilege: analyzer runs at privilege level that analyzed code cannot reach
  - privileged access to side effects: if side effects are introduced, they should be visible only with a privilege level that analyzed code cannot reach
  - same basic instruction semantics: side effects are allowed only during exceptions
  - transparent exception handling: when an exception occurs, analyzer can reconstruct the expected context where needed
  - identical timing information: access to time sources are shepherded (to forge values)
- Design undermined by practical limitations: the imperfections of VT-x

THE SEMANTIC GAP PROBLEM



VMI methods incur a semantic gap when trying to inspect high-level concepts of the guest system such as API calls or threads *(image from Ajay Kumar et al., ICPADS 2015)*

AUTOMATIC AND MANUAL ANALYSIS

- VMI-based systems face higher implementation complexity than in-guest agents but offer better transparency
  - sandbox vendors have started adopting VMI-based or hybrid approaches
  - DRAKVUF is the reference system in academic research
- Automatic analysis is only one part of the story. When dissection is required, VMI is no lifeboat for your conspicuous manual work
  - analysts still work on their own laptops with classic virtualization products: so they have to dismantle evasions manually again
  - Latest trend: out-of-VM debugging of a process running in VM

REFERENCES

- Al-Khaser. https://github.com/LordNoteworthy/al-khaser/
- A fistful of red-pills: How to automatically generate procedures to detect CPU emulators. WOOT 2009. http://roberto.greyhats.it/pubs/woot09.pdf
- A Virtual Machine Introspection Based Architecture for Intrusion Detection. NDSS 2003. https://suif.stanford.edu/papers/vmi-ndss03.pdf
- Ether: Malware Analysis via Hardware Virtualization Extensions. ACM CCS 2008. http://ether.gtisc.gatech.edu/ether_ccs_2008.pdf
- Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System. ACM ACSAC 2014. https://www.sec.in.tum.de/i20/publications/scalability-fidelity-and-stealth-in-the-drakvuf-dynamic-malware-analysis-system/@@download/file/scalability-fidelity-stealth.pdf
- Spotless Sandboxes: Evading Malware Analysis Systems Using Wear-and-Tear Artifacts. IEEE S&P 2017. https://www.ieee-security.org/TC/SP2017/papers/429.pdf
- On the Dissection of Evasive Malware. IEEE Transactions on Information Forensics and Security, 2020. https://www.diag.uniroma1.it/~delia/papers/tifs20.pdf

REFERENCES

- BluePill: Neutralizing Anti-Analysis Behavior in Malware Dissection. Black Hat Europe 2019. https://www.blackhat.com/eu-19/briefings/schedule/index.html#bluepill-neutralizing-anti-analysis-behavior-in-malware-dissection-17685
- My Ticks Don't Lie: New Timing Attacks for Hypervisor Detection. Black Hat Europe 2020. https://www.blackhat.com/eu-20/briefings/schedule/#my-ticks-dont-lie-new-timing-attacks-for-hypervisor-detection-21520
- Evasion techniques from Checkpoint Research. https://evasions.checkpoint.com/ with PoC implementations available at https://github.com/CheckPointSW/Evasions
- Anti-debug tricks from Checkpoint Research. https://anti-debug.checkpoint.com/ with PoC implementations available at https://github.com/CheckPointSW/showstopper/
- Defeating evasive malware, VMRay whitepaper Winter 2020. https://www.vmray.com/wp-content/uploads/2020/03/Defeating-Evasive-Malware-VMRay-Whitepaper.pdf#new_tab
- PoW-How: An Enduring Timing Side-Channel to Evade Online Malware Sandboxes. ESORICS 2021. https://arxiv.org/pdf/2109.02979.pdf