Module 3

Understanding and countering malware's evasion and self-defence

https://github.com/hasherezade/malware_training_vol1

CC BY

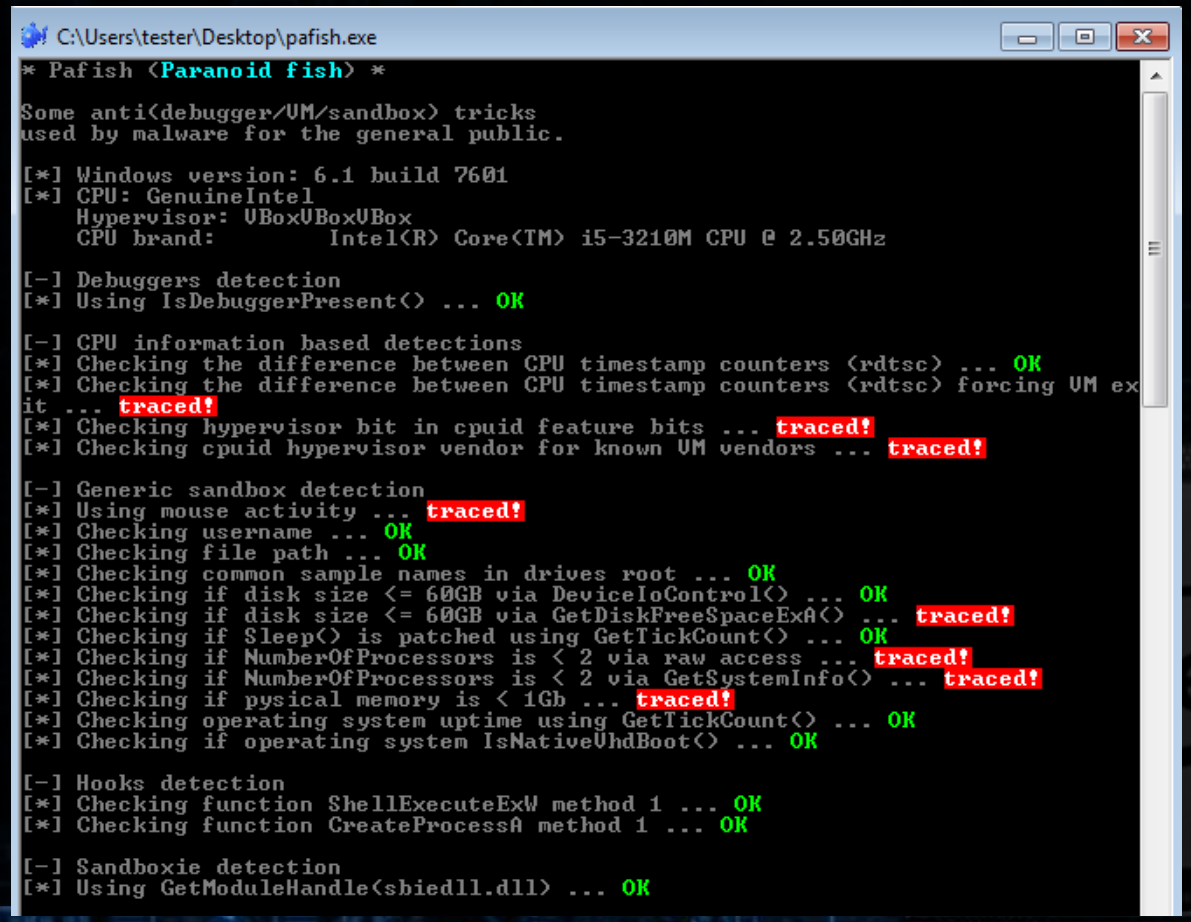
Fingerprinting for evasion

Fingerprinting for evasion

- Fingerprinting = gathering information about the environment where the executable was deployed
- It is used by malware to determine whether it is deployed in a controlled enviromnent, i.e. sandbox, analysis machine
- Open source projects with rich sets of techniques:
  - https://github.com/a0rtega/pafish
  - https://github.com/LordNoteworthy/al-khaser
  - https://www.aldeid.com/wiki/ScoopyNG
- Presented demos you can find at:
  - https://github.com/hasherezade/antianalysis_demos

Fingerprinting for evasion

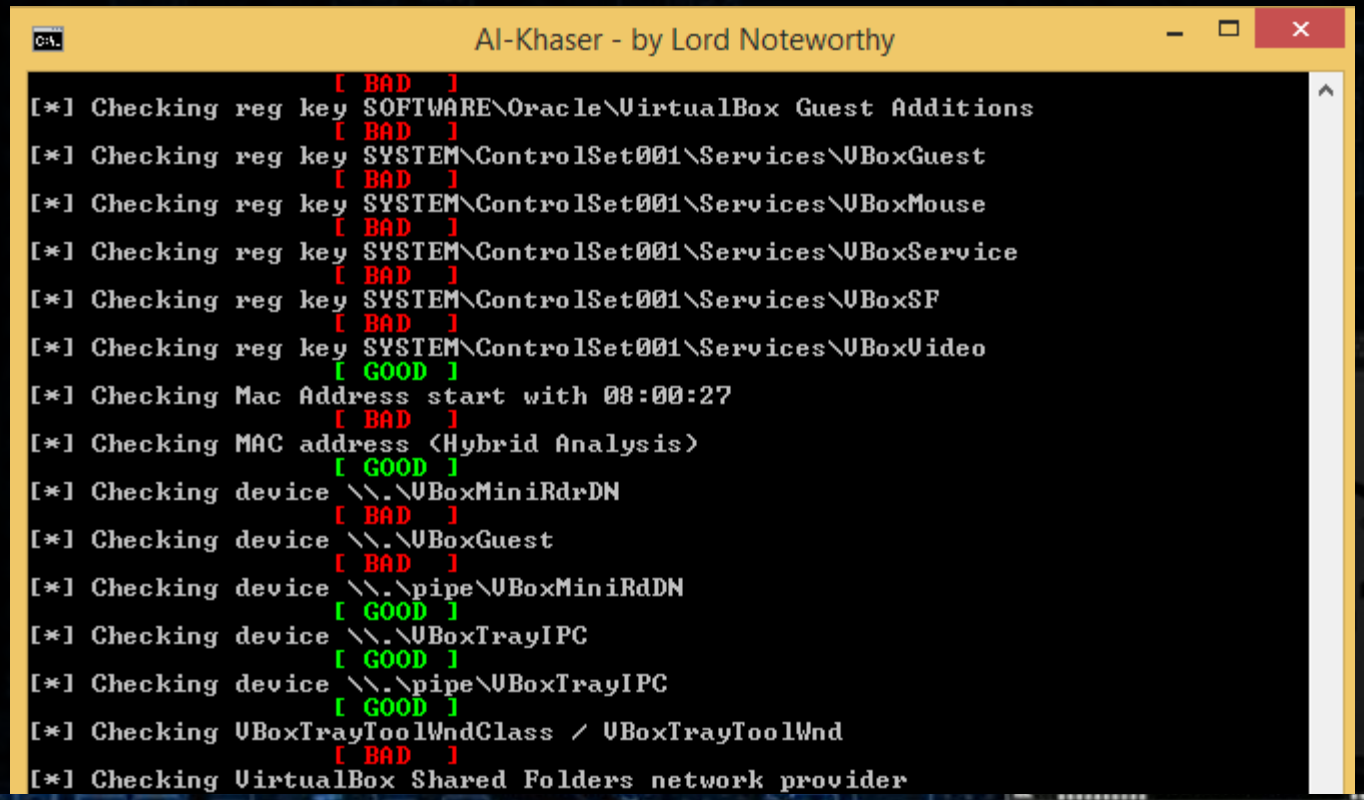
- PaFish in action:



https://github.com/a0rtega/pafish

Fingerprinting for evasion

- Al-Khaser in action:



https://github.com/LordNoteworthy/al-khaser

Fingerprinting for evasion

- Most of the malware stop their execution once they observe being analyzed – that's how they protect their real mission from being revealed. Common reactions:
  - ExitProcess
  - Infinite sleep loop
- Some malware are more tricky, and:
  - deploy a decoy (i.e. an old variant of Andromeda)
  - corrupt their execution (i.e. Kronos) to crash at further point

Classic debugger detection techniques

Anti-debugger: the classic set

- The fact that the application is being debugged leaves some artefacts in the execution environment
- Malware tries to pick them up, and terminate or alter execution on such event
- There is a list of classic, well-known techniques, that malware authors keep using from years, and probably will keep using in the future
- Let's take a look at them...

Anti-debugger: approaches

- Using flags in internal process structures: EPROCESS, PEB
  - Some of those checks can be invoked via APIs
- Breakpoint detection
- Reaction on exceptions
- Time checks
- Searching for the physical presence of the debugger in the system: checking running processes, windows names/classes, installation artifacts of a debugger

Detecting debugger: basic API

The most basic method, using: `IsDebuggerPresent` and/or `CheckRemoteDebuggerPresent`

```
bool is_debugger_api()
{
        if (IsDebuggerPresent()) return true;

        BOOL has_remote = FALSE;
        CheckRemoteDebuggerPresent(GetCurrentProcess(), &has_remote);

        return has_remote ? true: false;
}
```

Detecting debugger: basic API

The most basic method, using: `IsDebuggerPresent`

IsDebuggerPresent(32-bit ver.)

```
75621E2E <kernelbase.IsDebuggerPresent>
mov eax,dword ptr fs:[18]
mov eax,dword ptr ds:[eax+30]
movzx eax,byte ptr ds:[eax+2]
ret
```

1. Get TEB
2. Get PEB
3. Get: BeingDebugged Flag

PEB

```
lkd> dt nt!_PEB
   +0x000 InheritedAddressSpace : UChar
   +0x001 ReadImageFileExecOptions : UChar
   +0x002 BeingDebugged    : UChar
   +0x003 BitField         : UChar
   +0x003 ImageUsesLargePages : Pos 0, 1 Bit
   +0x003 IsProtectedProcess : Pos 1, 1 Bit
   +0x003 IsImageDynamicallyRelocated : Pos 2, 1 Bit
   +0x003 SkipPatchingUser32Forwarders : Pos 3, 1 Bit
   +0x003 IsPackagedProcess : Pos 4, 1 Bit
   +0x003 IsAppContainer   : Pos 5, 1 Bit
```

Anti-debugger: PEB

- PEB contains information about the environment where the process was executed, and as well contains a lot of information relevant to detecting a debugger...
- Using it is more stealthy then using API, and also easy to do in pure assembly (convenient for a shellcode)

Detecting debugger: PEB

The more stealthy variant of the previous method is getting the BeingDebugged flag via PEB

```
lkd> dt nt!_PEB
   +0x000 InheritedAddressSpace : UChar
   +0x001 ReadImageFileExecOptions : UChar
   +0x002 BeingDebugged    : UChar
   +0x003 BitField         : UChar
   +0x003 ImageUsesLargePages : Pos 0, 1 Bit
   +0x003 IsProtectedProcess : Pos 1, 1 Bit
   +0x003 IsImageDynamicallyRelocated : Pos 2, 1 Bit
   +0x003 SkipPatchingUser32Forwarders : Pos 3, 1 Bit
   +0x003 IsPackagedProcess : Pos 4, 1 Bit
   +0x003 IsAppContainer   : Pos 5, 1 Bit
```

Related API:

- `IsDebuggerPresent`

Detecting debugger: PEB

Another flag in PEB related to being debugged is NtGlobalFlag (more recent addition: NtGlobalFlag2)

Command

```
   +0x080 TlsBitmapBits     : [2] Uint4B
   +0x088 ReadOnlySharedMemoryBase : Ptr64 Void
   +0x090 SharedData        : Ptr64 Void
   +0x098 ReadOnlyStaticServerData : Ptr64 Ptr64 Void
   +0x0a0 AnsiCodePageData  : Ptr64 Void
   +0x0a8 OemCodePageData   : Ptr64 Void
   +0x0b0 UnicodeCaseTableData : Ptr64 Void
   +0x0b8 NumberOfProcessors : Uint4B
   +0x0bc NtGlobalFlag      : Uint4B
   +0x0c0 CriticalSectionTimeout : _LARGE_INTEGER
   +0x0c8 HeapSegmentReserve : Uint8B
   +0x0d0 HeapSegmentCommit : Uint8B
```

NTGlobalFlag is set when the stack of the application is being watched

Related API:

- RtlGetCurrentPeb()

PEB->NtGlobalFlag

PEB->NtGlobalFlag2

Detecting debugger: PEB

If the process is **not** being debugged: NtGlobalFlag == 0

Otherwise, the following flags are set (NtGlobalFlag == 0x70):

```
FLG_HEAP_ENABLE_TAIL_CHECK   0x10
FLG_HEAP_ENABLE_FREE_CHECK   0x20
FLG_HEAP_VALIDATE_PARAMETERS 0x40
```

Detecting debugger: PEB

PEB.ProcessHeap.Flags:

- If not degugged:

  HEAP_GROWABLE (0x2)

- Otherwise:

```
HEAP_GROWABLE                       0x2
HEAP_TAIL_CHECKING_ENABLED          0x20
HEAP_FREE_CHECKING_ENABLED          0x40
HEAP_SKIP_VALIDATION_CHECKS         0x10000000
HEAP_VALIDATE_PARAMETERS_ENABLED    0x40000000
```

Detecting debugger: PEB

PEB.ProcessHeap.ForceFlags:

- If not degugged: 0
- Otherwise: related to PEB.ProcessHeap.Flags:

```
PEB.ProcessHeapFlags & 0x60010070
```

Detecting debugger: basic API

The most basic method, using: `CheckRemoteDebuggerPresent`

CheckRemoteDebuggerPresent

```
kernel32.76663F94
 push 0 ; ReturnLength
 push 4 ; ProcessInformationLength
 lea eax,dword ptr ss:[ebp+8]
 push eax ; ProcessInformation
 push 7 ; ProcessInformationClass -> ProcessDebugPort
 push dword ptr ss:[ebp+8] ; ProcessHandle
 call dword ptr ds:[<&NtQueryInformationProcess>]
 test eax,eax
 jl kernel32.76669176
```

EPROCESS

```
lkd> dt nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x2d8 ProcessLock      : _EX_PUSH_LOCK
   +0x2e0 UniqueProcessId  : Ptr64 Void
   +0x2e8 ActiveProcessLinks : _LIST_ENTRY
   +0x2f8 RundownProtect   : _EX_RUNDOWN_REF
   +0x300 Flags2           : Uint4B
...
   +0x3f8 Peb              : Ptr64 _PEB
   +0x400 Session          : Ptr64 _MM_SESSION_SPACE
   +0x408 Spare1           : Ptr64 Void
   +0x410 QuotaBlock       : Ptr64 _EPROCESS_QUOTA_BLOCK
   +0x418 ObjectTable      : Ptr64 _HANDLE_TABLE
   +0x420 DebugPort        : Ptr64 Void
   +0x428 WoW64Process     : Ptr64 _EWOW64PROCESS
```

Detecting debugger: API

Some of the mentioned artifacts (and more) can be retrieved using `NtQueryInformationProcess`

Relevant parameters:

```
ProcessDebugPort                0x7 -> EPROCESS.DebugPort
ProcessDebugFlags              0x1F -> !(EPROCESS.NoDebugInherit)
ProcessDebugObjectHandle       0x1E -> returns DebugObject
ProcessBasicInformation         0x0 -> to get the parent process
```

https://ctf-wiki.github.io/ctf-wiki/reverse/windows/anti-debug/ntqueryinformationprocess/

Reaction on exceptions

If the debugger is present, it will try to handle the exception:

```
bool exception_is_dbg()
{
	__try {
		RaiseException(DBG_PRINTEXCEPTION_C, 0, 0, 0);
	} __except (EXCEPTION_EXECUTE_HANDLER) {
		return false;
	}

	return true;
}
```

Hardware breakpoints

- There are 4 Debug registrs that we can use for setting Hardware Breakpoints:
  - DR0-DR3

- Once we set the Hardware Breakpoint, the relevant address is filled in one of those registers. Example:

```
DR0 00413AB7      remcos.00413AB7
DR1 00000000
DR2 00000000
DR3 00000000
DR6 00000000
DR7 00000001
```

  - DR6 – flags indicating the Debug Register which's breakpoint got hit
  - DR7 – flags indicating which of the Debug Registers are set

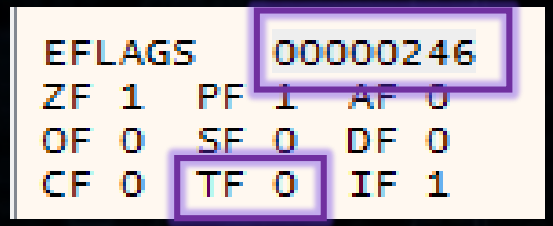
Hardware breakpoints

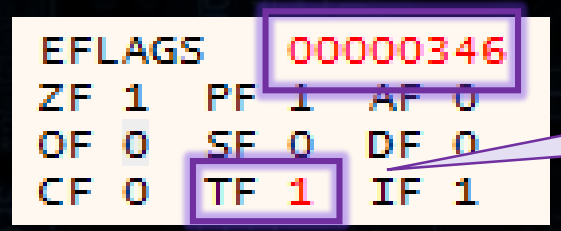
Checking if the Hardware Breakpoints have been set:

```
bool hardware_bp_is_dbg()
{
        CONTEXT ctx = { 0 };
        bool is_hardware_bp = false;

        HANDLE thread = OpenThread(THREAD_ALL_ACCESS, FALSE, GetCurrentThreadId());
        ctx.ContextFlags = CONTEXT_DEBUG_REGISTERS;
        if (GetThreadContext(thread, &ctx)) {
                is_hardware_bp = (ctx.Dr0 | ctx.Dr1 | ctx.Dr2 | ctx.Dr3) != 0;
        }
        CloseHandle(thread);

        return is_hardware_bp;
}
```

The Trap Flag: Single Stepping

- The Trap Flag is one of the Flags in the EFLAGS register





0x346 XOR 0x246 = 0x100 (TF)

- Setting the Trap Flag - allowing to step throught the code via INT 0x1: „Single Step" after each instruction (generates an exception)

The Trap Flag: Single Stepping

We cannot access EFLAGS directly - we need to do it via stack:

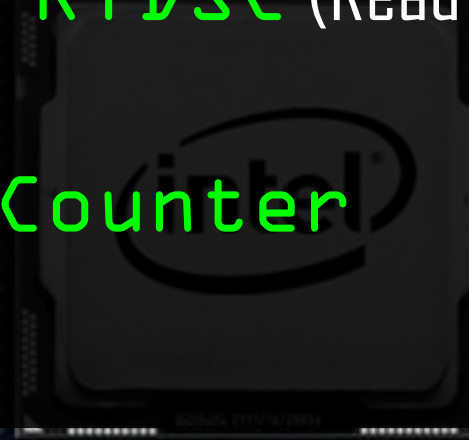
```
pushfd                          ; push all the flags
or dword ptr[esp], 0x100        ; the flags are now in [esp]
                                ; apply the mask to set the bit
                                ; 0x100, that means TF
popfd                           ; load the flags from the stack again
```

If we are single-stepping through the code, the debugger will handle the generated interrupt. Otherwise, setting of the Trap Flag will generates an exception.

The time check

- Debugging (also: emulation, or tracing the application by instrumentation tools) often slows down the execution
- The time check is a simple way to find out that the application may be under control of analysis tools
- The time check is often implemented with the help or `RTDSC` (Read Time-Stamp Counter) instruction

```
RTDCS -> EDX:EAX = TimeStampCounter
```



https://c9x.me/x86/html/file_module_x86_id_278.html

The time check

The time should be measured at least twice, and compared with a threshold. Example:

```
bool antidbg_timer_check()
{
    static ULONGLONG time = 0;
    if (time == 0) {
        time = __rdtsc();
        return false;
    }
    ULONGLONG second_time = __rdtsc();
    ULONGLONG diff = (second_time - time) >> 20;
    if (diff > 0x100) {
        time = second_time;
        return true;
    }
    return false;
}
```

Defense against anti-debug

- Debugger Plugins, i.e.
  - ScyllaHide (using user-mode hooking)
  - TitanHide (using kernel-mode hooking)
  - SharpOD
- OllyDbg plugins (older, classics):
  - OllyAdvanced
  - Phantom
  - StrongOD

https://github.com/fr0gger/awesome-ida-x64-olly-plugin

Classic anti-VM techniques

Anti-VM fingerprinting

- Virutal Machine emulates the real one to big extend, but still there are some artifacts in the environment that makes it distinguishable
- Depending which hypervisor do we use, those artifacts will differ
- It is quite common among malware to look for some of those artifacts in order to detect the Virtual Machine
- Some checks base on the presence of some particular names, related to the hypervisor, other – on some loosely related features (i.e. relatively weak parameters, one processor, etc)

Anti-VM: approaches

- Using presence/absence of some intstructions
- Identifiers returned by CPUID
- Memory-specific („The Red Pill" – IDT checking; GDT, LDT checks)
- Time checks
- Weaker hardware parameters (comparing to most modern physical machines)
- Searching for the physical presence of the VM-related artifacts: checking running processes, windows names/classes, registry keys, etc.

CPUID (1)

- One of the low-level anti-vm techniques, is a check using CPUID instruction
- Check for processof features:

```
mov is_bit_set, 0
mov eax, 1 ; the parameter given to CPUID
cpuid
bt  ecx, 0x1f; bit 31
jnc  finish
  mov is_bit_set, 1 ; if the bit is set, it is a VM
finish:
```

https://c9x.me/x86/html/file_module_x86_id_45.html

CPUID (0x40000000)

- One of the low-level anti-vm techniques, is a check using CPUID instruction
- Check for the hypervisor brand:

```
mov eax, 0x40000000; the parameter given to CPUID
cpuid
mov brand_id_0, ebx
mov brand_id_1, ecx
mov brand_id_2, edx
```

https://c9x.me/x86/html/file_module_x86_id_45.html

CPUID (0x40000000)

- One of the low-level anti-vm techniques, is a check using `CPUID` instruction

```
"KVMKVMKVM\0\0\0"; // KVM
"Microsoft Hv"; // MS Hyper-V or Virtual PC
"VMwareVMware"; // VMware
"XenVMMXenVMM"; // Xen
"prl hyperv  "; // Parallels
"VBoxVBoxVBox"; // VirtualBox
```

https://github.com/a0rtega/pafish/blob/master/pafish/cpu.c

CPUID - defense

- Fortunately, we often can overwrite the values returned by CPUID by our own
- Appropriate settings may force the VM to supply our custom values instead of the hardcodes ones...

https://www.vmray.com/cyber-security-blog/a-pafish-primer/

CPUID - defense

- In VMWare: settings can be changed in the .vmx file
- Anti bit-check - CPUID (1)

```
cpuid.1.ecx="0---:----:----:----:----:----:----:----"
```

- Anti brand-check (0x40000000)

```
cpuid.40000000.ecx="0000:0000:0000:0000:0000:0000:0000:0000"
cpuid.40000000.edx="0000:0000:0000:0000:0000:0000:0000:0000"
```

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

VMWare I/O port

- Trying to read the special I/O port, used by VMware to communicate with host, with the help of `IN` instruction
- On a physical machine, the exception will occur

```
mov eax, 'VMXh'
mov ebx,0
mov ecx, 10
Mov edx, 'VX'
in eax, dx
cmp ebx,'VMXh'
```

https://blog.malwarebytes.com/threat-analysis/2014/09/five-anti-debugging-tricks-that-sometimes-fool-analysts/

TODO...

- To be continued