Anti-Virtual Environments - Multiple Techniques

Introduction

Anti-virtualization was already introduced in an earlier module. This module will go through Anti-Virtual Environment (AVE) techniques.

Anti-Virtualization Via Hardware Specs

Generally speaking, virtualized environments do not have full access to the host machine's hardware. The lack of full access to the hardware can be used by malware to detect if it's being executed inside a virtual environment or sandbox. Keep in mind that there is no guarantee of complete accuracy because the machine could simply be running with low hardware specs. The hardware specs that will be checked are the following:

- CPU - Check if there are fewer than 2 processors.
- RAM - Check if there are less than 2 gigabytes.
- Number of USBs previously mounted - Check if there are fewer than 2 USBs.

CPU Check

The CPU check can be done using the GetSystemInfo WinAPI. This function returns an SYSTEM_INFO structure that contains information about the system, including the number of processors.

```
SYSTEM_INFO   SysInfo   = { 0 };

GetSystemInfo(&SysInfo);
if (SysInfo.dwNumberOfProcessors < 2){
  // possibly a virtualized environment
}
```

RAM Check

Checking the RAM storage can be done via the GlobalMemoryStatusEx WinAPI. This function returns a MEMORYSTATUSEX structure containing information about the current state of the physical and virtual memory in the system. The RAM storage can be found through the `ullTotalPhys` member. It contains the amount of current physical memory in bytes.

```
MEMORYSTATUSEX MemStatus = { .dwLength = sizeof(MEMORYSTATUSEX) };

if (!GlobalMemoryStatusEx(&MemStatus)) {
    printf("\n\t[!] GlobalMemoryStatusEx Failed With Error : %d \n",
GetLastError());
  }

  if ((DWORD)MemStatus.ullTotalPhys <= (DWORD)(2 * 1073741824)) {
     // Possibly a virtualized environment
  }
```

Note that `2 * 1073741824` is the size of two gigabytes in bytes.

Previously Mounted USBs Check

Lastly, the number of USBs previously mounted in the system can be checked via the `HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Enum\USBSTOR` registry key. Retrieving the registry key's value is done using the `RegOpenKeyExA` and `RegQueryInfoKeyA` WinAPIs.

```
  HKEY    hKey            = NULL;
  DWORD   dwUsbNumber     = NULL;
  DWORD   dwRegErr        = NULL;


  if ((dwRegErr = RegOpenKeyExA(HKEY_LOCAL_MACHINE,
"SYSTEM\\ControlSet001\\Enum\\USBSTOR", NULL, KEY_READ, &hKey)) !=
ERROR_SUCCESS) {
    printf("\n\t[!] RegOpenKeyExA Failed With Error : %d | 0x%0.8X \n",
dwRegErr, dwRegErr);
  }

  if ((dwRegErr = RegQueryInfoKeyA(hKey, NULL, NULL, NULL, &dwUsbNumber,
NULL, NULL, NULL, NULL, NULL, NULL, NULL)) != ERROR_SUCCESS) {
    printf("\n\t[!] RegQueryInfoKeyA Failed With Error : %d | 0x%0.8X \n",
dwRegErr, dwRegErr);
  }

  // Less than 2 USBs previously mounted
  if (dwUsbNumber < 2) {
    // possibly a virtualized environment
  }
```

Anti-Virtualization Via Hardware Specs Code

The previous code snippets are combined into one function, `IsVenvByHardwareCheck`. This function returns `TRUE` if it detects a virtualized environment.

```
BOOL IsVenvByHardwareCheck() {

        SYSTEM_INFO             SysInfo                 = { 0 };
        MEMORYSTATUSEX  MemStatus               = { .dwLength =
sizeof(MEMORYSTATUSEX) };
        HKEY                    hKey                    = NULL;
        DWORD                   dwUsbNumber             = NULL;
        DWORD                   dwRegErr                = NULL;

        // CPU CHECK
        GetSystemInfo(&SysInfo);

        // Less than 2 processors
        if (SysInfo.dwNumberOfProcessors < 2){
                return TRUE;
        }

        // RAM CHECK
        if (!GlobalMemoryStatusEx(&MemStatus)) {
                printf("\n\t[!] GlobalMemoryStatusEx Failed With Error : %d
\n", GetLastError());
                return FALSE;
        }

        // Less than 2 gb of ram
        if ((DWORD)MemStatus.ullTotalPhys < (DWORD)(2 * 1073741824)) {
                return TRUE;
        }



        // NUMBER OF USBs PREVIOUSLY MOUNTED
        if ((dwRegErr = RegOpenKeyExA(HKEY_LOCAL_MACHINE,
"SYSTEM\\ControlSet001\\Enum\\USBSTOR", NULL, KEY_READ, &hKey)) !=
ERROR_SUCCESS) {
                printf("\n\t[!] RegOpenKeyExA Failed With Error : %d |
0x%0.8X \n", dwRegErr, dwRegErr);
                return FALSE;
        }

        if ((dwRegErr = RegQueryInfoKeyA(hKey, NULL, NULL, NULL,
&dwUsbNumber, NULL, NULL, NULL, NULL, NULL, NULL, NULL)) != ERROR_SUCCESS)
{
                printf("\n\t[!] RegQueryInfoKeyA Failed With Error : %d |
0x%0.8X \n", dwRegErr, dwRegErr);
```

```
                return FALSE;
        }

        // Less than 2 usbs previously mounted
        if (dwUsbNumber < 2) {
                return TRUE;
        }

        RegCloseKey(hKey);

        return FALSE;
}
```

Anti-Virtualization Via Machine Resolution

In a sandbox environment, the resolution and display properties of the machine are often set to a standardized and consistent value, which can be different from the resolution and display properties of a real-world machine. Therefore, machines with low resolutions can be used as an indicator of a virtualized environment.

From a programming perspective, the first step will be to enumerate the display monitors of a system via the EnumDisplayMonitors WinAPI.

The `EnumDisplayMonitors` function requires a callback function to be executed for every display monitor it detects, in this callback function, the GetMonitorInfoW WinAPI must be called. This function retrieves the resolution of the display monitor.

The fetched information is returned as a MONITORINFO structure by `GetMonitorInfoW`, which is shown below.

```
typedef struct tagMONITORINFO {
  DWORD cbSize;                // The size of the structure
  RECT  rcMonitor;             // Display monitor rectangle, expressed in
virtual-screen coordinates
  RECT  rcWork;                // Work area rectangle of the display
monitor, expressed in virtual-screen coordinates
  DWORD dwFlags;                   // Represents attributes of the display
monitor
} MONITORINFO, *LPMONITORINFO;
```

The `rcMonitor` member contains the information that's needed. This member is also a structure of type RECT that defines a rectangle through the X and Y coordinates of its upper-left and lower-right corners.

After retrieving the values of the `RECT` structure, some calculations are made to determine the actual coordinates of the display:

1. `MONITORINFO.rcMonitor.right - MONITORINFO.rcMonitor.left` - This gives us the width (X value)
2. `MONITORINFO.rcMonitor.top - MONITORINFO.rcMonitor.bottom` - This gives us the height (Y value)

Anti-Virtualization Via Machine Resolution Code

The `CheckMachineResolution` function uses the described process in which the machine's resolution is calculated, by executing the `ResolutionCallback` callback.

```
// The callback function called whenever 'EnumDisplayMonitors' detects an
display
BOOL CALLBACK ResolutionCallback(HMONITOR hMonitor, HDC hdcMonitor, LPRECT
lpRect, LPARAM ldata) {

        int             X       = 0,
                        Y       = 0;
        MONITORINFO     MI      = { .cbSize = sizeof(MONITORINFO) };

        if (!GetMonitorInfoW(hMonitor, &MI)) {
                printf("\n\t[!] GetMonitorInfoW Failed With Error : %d \n",
GetLastError());
                return FALSE;
        }

        // Calculating the X coordinates of the desplay
        X = MI.rcMonitor.right - MI.rcMonitor.left;

        // Calculating the Y coordinates of the desplay
        Y = MI.rcMonitor.top - MI.rcMonitor.bottom;

        // If numbers are in negative value, reverse them
        if (X < 0)
                X = -X;
        if (Y < 0)
                Y = -Y;

        if ((X != 1920 && X != 2560 && X != 1440) || (Y != 1080 && Y !=
1200 && Y != 1600 && Y != 900))
                *((BOOL*)ldata) = TRUE; // sandbox is detected

        return TRUE;
}
```

```
BOOL CheckMachineResolution() {

        BOOL    SANDBOX         = FALSE;

        // SANDBOX will be set to TRUE by 'EnumDisplayMonitors' if a
sandbox is detected
        EnumDisplayMonitors(NULL, NULL,
(MONITORENUMPROC)ResolutionCallback, (LPARAM)(&SANDBOX));

        return SANDBOX;
}
```

Anti-Virtualization Via File Name

Sandboxes will often rename files as a method of classification (e.g. renaming it to its MD5 hash). This process generally results in an arbitrary file name with a mixture of letters and numbers.

The function `ExeDigitsInNameCheck` shown below is used to count the number of digits in the current filename. It uses GetModuleFileNameA to get the file name (which includes the path) and then PathFindFileNameA to separate the file name from the path.

Finally, the isdigit function is used to determine if the characters in the file name are digits. If more than 3 digits are in the file name, then `ExeDigitsInNameCheck` will assume it is in a sandbox and return `TRUE`.

```
BOOL ExeDigitsInNameCheck() {

        CHAR    Path                    [MAX_PATH * 3];
        CHAR    cName                   [MAX_PATH];
        DWORD   dwNumberOfDigits        = NULL;

        // Getting the current filename (with the full path)
        if (!GetModuleFileNameA(NULL, Path, MAX_PATH * 3)) {
                printf("\n\t[!] GetModuleFileNameA Failed With Error : %d
\n", GetLastError());
                return FALSE;
        }

        // Prevent a buffer overflow - getting the filename from the full
path
        if (lstrlenA(PathFindFileNameA(Path)) < MAX_PATH)
                lstrcpyA(cName, PathFindFileNameA(Path));

        // Counting number of digits
        for (int i = 0; i < lstrlenA(cName); i++){
                if (isdigit(cName[i]))
                        dwNumberOfDigits++;
        }

        // Max digits allowed: 3
        if (dwNumberOfDigits > 3){
                return TRUE;
        }

        return FALSE;
}
```

Anti-Virtualization Via Number Of Running Processes

Another way of detecting a virtualized environment is by checking the number of running processes on the system. Sandboxes will generally not have many applications installed and therefore will have fewer processes running. Similarly to the previous methods, this is not a silver bullet that will guarantee the system to be a sandbox. A Windows system should have at least 60-70 processes running.

The processes will be enumerated using the `EnumProcesses` technique. The `CheckMachineProcesses` function returns `TRUE` if it detects a sandbox which is if the system is running less than 50 processes.

```
BOOL CheckMachineProcesses() {

        DWORD           adwProcesses    [1024];
        DWORD           dwReturnLen             = NULL,
                            dwNmbrOfPids                = NULL;

        if (!EnumProcesses(adwProcesses, sizeof(adwProcesses),
&dwReturnLen)) {
                printf("\n\t[!] EnumProcesses Failed With Error : %d \n",
GetLastError());
                return FALSE;
        }

        dwNmbrOfPids = dwReturnLen / sizeof(DWORD);

        // If less than 50 process, it's possibly a sandbox
        if (dwNmbrOfPids < 50)
                return TRUE;

        return FALSE;
}
```

Anti-Virtualization Via User Interaction

Sandboxes often run in a headless environment, meaning that there is no display or peripherals, such as a keyboard and mouse. Headless environments are also typically automated and triggered by scripts or other tools. The lack of user interaction can be an indicator of a possible sandbox environment. For example, the malware can check if an environment does not receive any mouse clicks or keystrokes over a certain period.

Recall the *API Hooking - Using Windows APIs* module where the `SetWindowsHookExW` and `CallNextHookEx` WinAPIs were used to track mouse clicks. The same technique is applied in the function below, `MouseClicksLogger`. If it does not receive more than 5 mouse clicks over a period of 20 seconds then it will assume it's inside a sandboxed environment.

```
// Monitor mouse clicks for 20 seconds
#define MONITOR_TIME   20000

// Global hook handle variable
HHOOK g_hMouseHook      = NULL;
// Global mouse clicks counter
DWORD g_dwMouseClicks   = NULL;

// The callback function that will be executed whenever the user clicked a
mouse button
LRESULT CALLBACK HookEvent(int nCode, WPARAM wParam, LPARAM lParam){

    // WM_RBUTTONDOWN :         "Right Mouse Click"
    // WM_LBUTTONDOWN :         "Left Mouse Click"
    // WM_MBUTTONDOWN :         "Middle Mouse Click"

    if (wParam == WM_LBUTTONDOWN || wParam == WM_RBUTTONDOWN || wParam ==
WM_MBUTTONDOWN) {
        printf("[+] Mouse Click Recorded \n");
        g_dwMouseClicks++;
    }

    return CallNextHookEx(g_hMouseHook, nCode, wParam, lParam);
}



BOOL MouseClicksLogger(){

    MSG         Msg         = { 0 };

    // Installing hook
    g_hMouseHook = SetWindowsHookExW(
        WH_MOUSE_LL,
        (HOOKPROC)HookEvent,
        NULL,
        NULL
    );
    if (!g_hMouseHook) {
        printf("[!] SetWindowsHookExW Failed With Error : %d \n",
GetLastError());
    }

    // Process unhandled events
    while (GetMessageW(&Msg, NULL, NULL, NULL)) {
        DefWindowProcW(Msg.hwnd, Msg.message, Msg.wParam, Msg.lParam);
    }

    return TRUE;
}



int main() {

    HANDLE  hThread         = NULL;
    DWORD   dwThreadId      = NULL;

    // running the hooking function in a seperate thread for 'MONITOR_TIME'
ms
    hThread = CreateThread(NULL, NULL,
(LPTHREAD_START_ROUTINE)MouseClicksLogger, NULL, NULL, &dwThreadId);
    if (hThread) {
        printf("\t\t<<>> Thread %d Is Created To Monitor Mouse Clicks For
%d Seconds <<>>\n\n", dwThreadId, (MONITOR_TIME / 1000));
        WaitForSingleObject(hThread, MONITOR_TIME);
    }

    // unhooking
    if (g_hMouseHook && !UnhookWindowsHookEx(g_hMouseHook)) {
        printf("[!] UnhookWindowsHookEx Failed With Error : %d \n",
GetLastError());
    }

    // the test
    printf("[i] Monitored User's Mouse Clicks : %d ... ", g_dwMouseClicks);
    // if less than 5 clicks - its a sandbox
    if (g_dwMouseClicks > 5)
        printf("[+] Passed The Test \n");
    else
        printf("[-] Posssibly A Virtual Environment \n");


    printf("[#] Press <Enter> To Quit ... ");
    getchar();

    return 0;
}
```