Anti Anti Forensics - Daniel Ayzenshteyn

The goal: Flag process which uses timing based anti forensics techniques.
Subgoal: Avoid false positives

Anti Forensics techniques - Timing based

```
"When a process is traced in a debugger, there is a huge delay between
instructions and execution. The "native" delay between some parts of
code can be measured and compared with the actual delay using several
approaches."
```

The attacker has two main parameters to play with:

1. The tool to measure time (Different api functions)
2. The native delay to exploit (delay from VM/Sandbox/Debugger)

For experimentation purposes suppose there is a native delay of more than 10 ms in the time it takes to sleep under debugger/vm/sandbox.

In my testing sample I setted up 6 different api calls to measure time, and a sleep operation for 100 ms - The attacker expects that the sleep operation will take between 90-110 ms and will suspect being "under a microscope" if it takes less than 90 ms or more than 110 ms.

For example here is an attacker written time based anti forensics function:

```
bool checkGetTickCountDiscrepancy() {
    DWORD startTick = GetTickCount();
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    DWORD endTick = GetTickCount();
    DWORD elapsedTime = endTick - startTick;
    return (elapsedTime < 90 || elapsedTime > 110);
}
```

We call GetTickCount(), and do some action that behaves differently on a VM / Sandbox / Debugger versus a real machine. That action could be an IO bound task that will have an overhead because of the use of debugger, virtualization or hooking solutions in sandboxes.

The attacker should set his expected time for the action, and measure the time with the GetTickCount() call for the second time. After the second call he calculates the elapsed time between those two time measuring calls and decides whether or not he is "under a microscope".

In my example the attacker chose to use sleep for 100ms - then he checks whether the action took some time between 90 ms to 110 ms.
Note: The attacker can theoretically chose to call two different time measuring functions to avoid getting flag by rules such:

```
rule:
  meta:
    name: check for time delay via GetTickCount
    namespace: anti-analysis/anti-debugging/debugger-detection
    author: michael.hunhoff@fireeye.com
    scope: function
    mbc:
      - Anti-Behavioral Analysis::Debugger Detection::Timing/Delay Check
GetTickCount [B0001.032]
    examples:
      - Practical Malware Analysis Lab 16-03.exe_:0x4013d0
  features:
    - and:
      - count(api(kernel32.GetTickCount)): 2 or more
```

This rule flags every program which has 2 or more GetTickCount calls.

The attacker can evade this rule via calling GetTickCount and the second time calling GetTickCount64.
We can think of other interesting ways to measure time with different combinations of different api calls - For example calling GetTickCount and later getting the time from other sources such as the RDTSC instruction or GetSystemTime.

Anti - Anti Forensics

We should note that we can't flag every API call for time - too much false positives. However we can try to catch those API calls which are too close to each other. (Sign that they measure small discrepancies in time that distinguish between a VM/Sandbox/Debugger and a regular system)

For detection of the technique mentioned above I have decided on two things:

1. We will assume that the attacker has a way to measure time using multiple different api calls (He is smart).
2. We will set our "Suspicious close time frame" to be less than 150ms.

- Meaning: My solution will flag any program which requests time from the system less than 150ms apart from each other.
- Recall/Precision: We can set the 150ms value to other value. If we choose higher number we will get more false positives, if we choose lower value the malware won't be flagged.

Implementation

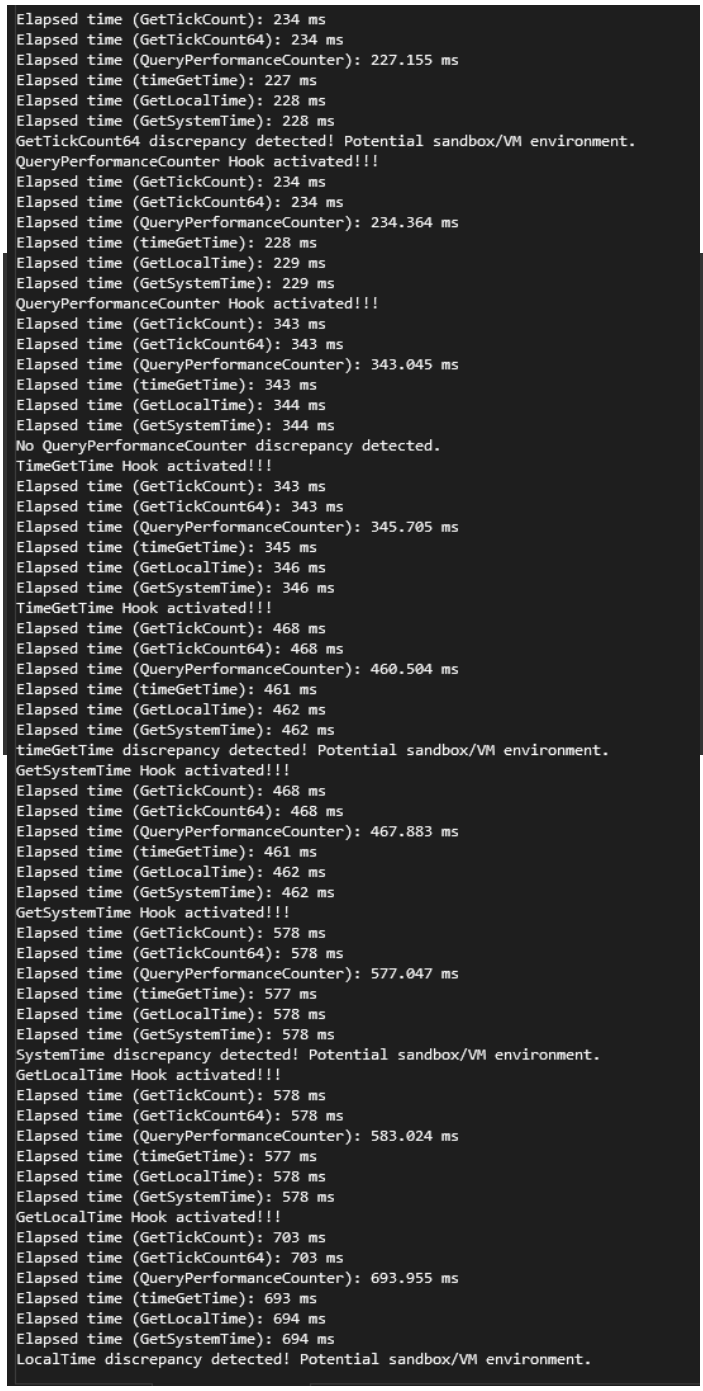
We will use IAT hooking and process creation in suspend mode to detect the time based anti-forensics technique.
The general flow is as follows:

1. The detector (AntiForensicsDetector) is launching the sample (AntiForensicsSample) in suspended mode.

2. The detector injects a dll (TimeBasedDetectorDLL.dll) to the sample.
3. The detector passes a log file path to the dll via named pipe - in which the analysis results will be saved.
4. The dll hooks 6 common time measuring functions.
5. The detector resumes the execution of the sample.

I hooked the most common measuring time api calls via IAT hooking:
`GetLocalTime()`
`GetSystemTime()`
`GetTickCount()`
`GetTickCount64()`
`QueryPerformanceCounter()`
`timeGetTime()`

We see that they all measure the time with an more or less same accuracy (+-10ms)



```
Elapsed time (GetTickCount): 234 ms
Elapsed time (GetTickCount64): 234 ms
Elapsed time (QueryPerformanceCounter): 227.155 ms
Elapsed time (timeGetTime): 227 ms
Elapsed time (GetLocalTime): 228 ms
Elapsed time (GetSystemTime): 228 ms
GetTickCount64 discrepancy detected! Potential sandbox/VM environment.
QueryPerformanceCounter Hook activated!!!
Elapsed time (GetTickCount): 234 ms
Elapsed time (GetTickCount64): 234 ms
Elapsed time (QueryPerformanceCounter): 234.364 ms
Elapsed time (timeGetTime): 228 ms
Elapsed time (GetLocalTime): 229 ms
Elapsed time (GetSystemTime): 229 ms
QueryPerformanceCounter Hook activated!!!
Elapsed time (GetTickCount): 343 ms
Elapsed time (GetTickCount64): 343 ms
Elapsed time (QueryPerformanceCounter): 343.045 ms
Elapsed time (timeGetTime): 343 ms
Elapsed time (GetLocalTime): 344 ms
Elapsed time (GetSystemTime): 344 ms
No QueryPerformanceCounter discrepancy detected.
TimeGetTime Hook activated!!!
Elapsed time (GetTickCount): 343 ms
Elapsed time (GetTickCount64): 343 ms
Elapsed time (QueryPerformanceCounter): 345.705 ms
Elapsed time (timeGetTime): 345 ms
Elapsed time (GetLocalTime): 346 ms
Elapsed time (GetSystemTime): 346 ms
TimeGetTime Hook activated!!!
Elapsed time (GetTickCount): 468 ms
Elapsed time (GetTickCount64): 468 ms
Elapsed time (QueryPerformanceCounter): 460.504 ms
Elapsed time (timeGetTime): 461 ms
Elapsed time (GetLocalTime): 462 ms
Elapsed time (GetSystemTime): 462 ms
timeGetTime discrepancy detected! Potential sandbox/VM environment.
GetSystemTime Hook activated!!!
Elapsed time (GetTickCount): 468 ms
Elapsed time (GetTickCount64): 468 ms
Elapsed time (QueryPerformanceCounter): 467.883 ms
Elapsed time (timeGetTime): 461 ms
Elapsed time (GetLocalTime): 462 ms
Elapsed time (GetSystemTime): 462 ms
GetSystemTime Hook activated!!!
Elapsed time (GetTickCount): 578 ms
Elapsed time (GetTickCount64): 578 ms
Elapsed time (QueryPerformanceCounter): 577.047 ms
Elapsed time (timeGetTime): 577 ms
Elapsed time (GetLocalTime): 578 ms
Elapsed time (GetSystemTime): 578 ms
SystemTime discrepancy detected! Potential sandbox/VM environment.
GetLocalTime Hook activated!!!
Elapsed time (GetTickCount): 578 ms
Elapsed time (GetTickCount64): 578 ms
Elapsed time (QueryPerformanceCounter): 583.024 ms
Elapsed time (timeGetTime): 577 ms
Elapsed time (GetLocalTime): 578 ms
Elapsed time (GetSystemTime): 578 ms
GetLocalTime Hook activated!!!
Elapsed time (GetTickCount): 703 ms
Elapsed time (GetTickCount64): 703 ms
Elapsed time (QueryPerformanceCounter): 693.955 ms
Elapsed time (timeGetTime): 693 ms
Elapsed time (GetLocalTime): 694 ms
Elapsed time (GetSystemTime): 694 ms
LocalTime discrepancy detected! Potential sandbox/VM environment.
```

We can measure the time for each function independently, for this I implemented checkTimes, captureInitialTimes and calculateElapsedTime (inside the TimeBasedDetectorDLL).

```
// This code snippet will
track all time functions
independently when placed
inside all the hook functions

if (!initialValuesCaptured) {
    captureInitialTimes();
}
else {
    checkTimes();
    captureInitialTimes();
}
```

As stated above for my final solution I will measure time between all different time calls - via GetTickCount64() - We will flag every time call function which is called in less than 150ms apart from the previous time call function. (in the code - globalTime)
** **Small Note**: Winmm.lib should be specified in both the Sample and the DLL as Additional Dependencies.