Evade Sandboxes With a Single Bit – the Trap Flag

unit42.paloaltonetworks.com/single-bit-trap-flag-intel-cpu/

Mark Lim July 19, 2021

By Mark Lim

July 19, 2021 at 3:30 PM

Category: Application Advisory/Analysis, Malware, Unit 42

Tags: Cortex, Intel, Lampion, Sandbox, threat prevention, Trap Flag, WildFire

This post is also available in: 日本語 (Japanese)

Executive Summary

Unit 42 has discovered a specific single bit (Trap Flag) in the Intel CPU register that can be abused by malware to evade sandbox detection in general purposes. Malware can detect whether it is executing in a physical or virtual machine (VM) by monitoring the response of the CPU after setting this single bit.

Sandboxing is a popular technique used to detect whether a sample is malicious. A sandbox analyzes the behaviors of the binary as it executes inside a controlled environment. To overcome the challenge of analyzing a large number of binaries with limited computing resources, virtual machines are used to build sandboxes. To evade detection, malware will

try to determine whether it is executing in a physical or virtual machine. When the malware finds out it is executing in a virtual machine, it will terminate its execution or provide fake outputs to hide its real intentions.

Some of the most common evasion techniques involve malware conducting various system checks against the environment it is executing in. For example, malware will often look for abnormal screen resolution, hard disk and physical memory size. Sandboxes can build countermeasures, such as returning fake information to the malware during these checks.

This blog documents how malware can detect the differences in CPU behaviors in a virtual or physical machine with only a single bit in the CPU register.

Palo Alto Networks customers are protected from malware families using similar sandbox evasion techniques with Cortex XDR or the Next-Generation Firewall with WildFire and Threat Prevention security subscriptions.

Single-Step Mode With a Single Bit — the Trap Flag (TF)

To detect the use of a VM in a sandbox, malware could check the behavior of the CPU after the trap flag is enabled.

The trap flag (TF) is the 8th single bit in the EFLAGs register of the Intel x86 CPU architecture. If the TF is enabled before a single instruction is executed, the CPU will raise an exception (single-step mode) after the instruction is completed. This exception stops the CPU execution to allow the contents of the registers and memory location to be examined by the exception handler. Before allowing code execution to continue, the CPU also has to clear the TF.

To determine whether a VM is used, malware can check whether the single-step exception was delivered to the correct CPU instruction, after executing specific instructions (e.g. CPUID, RDTSC, IN) that cause the VM to exit with the TF enabled. During VM exits, the hypervisor – also known as Virtual Machine Monitor (VMM) – will emulate the effects of the physical CPU it encounters.

The following sequence of instructions explains the CPU's behavior after enabling the TF in a physical machine.

```
.text:00401068          pushf
.text:00401069          or      dword ptr [esp], 100h
.text:00401070          popf
.text:00401071          rdtsc
.text:00401073          nop
.text:00401074          nop
```

Figure 1. CPU instructions to enable TF.

The first three instructions enable the TF bit in the EFLAGs register of the CPU. RDTSC is executed with the TF enabled. In a physical machine, the exception would be delivered to the first no operation (NOP) instruction (0x00401073). Take note that the exception occurred on the instruction immediately after the execution of the instruction with TF enabled.

```
.text:00401068          pushf
.text:00401069          or      dword ptr [esp], 100h
.text:00401070          popf
.text:00401071          rdtsc       ; TF Enabled
.text:00401073          nop         ; exception
.text:00401074          nop
```

Figure 2. Execution in a physical machine.

Executing the same sequence of instructions in a VM would have a different effect. In a VM, executing RDTSC would result in a VM exit. The hypervisor will carry out its usual tasks of emulating the behaviors of the RDTSC instruction. However, an implementation of the hypervisor with incorrect emulation of the TF would result in the TF being ignored and the code execution will continue to the first NOP instruction. During the execution of the first NOP instruction, the TF is still enabled as the TF was not handled by the hypervisor. This results in an exception occurring on the second NOP instruction (0x00401073). The correct implementation will require the hypervisor to inject a debug exception after emulating the instruction that caused the VM exit and clearing the TF.

```
.text:00401068          pushf
.text:00401069          or      dword ptr [esp], 100h
.text:00401070          popf
.text:00401071          rdtsc       ; TF Enabled
.text:00401073          nop         ; TF Enabled
.text:00401074          nop         ; exception
```

Figure 3. Execution in a virtual machine.

As a sandbox evasion technique, malware will use an exception handler in addition to the above instruction sequence to examine which instruction the exception occurred on. The next section describes a real-world example of a malware family that made use of this technique to evade sandboxes.

Real-World Example

Lampion is a malware family that was targeting users in Portugal. Lampion employed multiple system checks to evade sandbox detection. One of the techniques is making use of the single-step mode with TF, as discussed in the previous section.

Lampion implemented all its system checks with x86 assembly instructions and minimal Windows API calls. This allowed the Lampion samples to conceal their behavior from the sandboxes. The Lampion samples would terminate if the malware determined it was

executing inside a VM. The system checks are also intertwined with multiple anti-reverse engineering techniques to hide from human analysts.

The following screenshot shows a snippet of instructions hidden in the Lampion sample that conducts the system check.

```
007F0E0C                db 'HeapCreate',0
007F0E17 word_7F0E17    dw 0                    ; DATA XREF: .text:007F3FD5↓o
007F0E19                db 'GetLocalTime',0
007F0E26 word_7F0E26    dw 0                    ; DATA XREF: .text:007F4009↓o
007F0E28                db 'CreateDirectoryW',0
007F0E39 word_7F0E39    dw 0                    ; DATA XREF: .text:007F3D8D↓o
007F0E3B                db 'RegDeleteValueW',0
007F0E4B ; ---------------------------------------------------------------------------
007F0E4B                popf                    ; TF enabled!
007F0E4C                rdtsc                   ; Privileged instruction
007F0E4E                nop
007F0E4F                pushf
007F0E50                pushf
007F0E51                pusha
007F0E52                lea     esp, [esp+28h]
007F0E56                jnp     loc_7EE5F2
007F0E5C                push    6600F72Fh
007F0E61                pusha
007F0E62                jmp     loc_7F8CD7
007F0E62 ; ---------------------------------------------------------------------------
007F0E67 word_7F0E67    dw 0                    ; DATA XREF: .text:007F3EC5↓o
007F0E69                db 'OpenProcess',0
007F0E75
007F0E75 ; =============== S U B R O U T I N E =======================================
```

Figure 4. Instructions in Lampion used to evade sandboxes.

The following is pseudocode to demonstrate how Lampion carries out one of its sandbox system checks by enabling TF on an instruction that causes the VM to exit.

```
Anti_sandbox_Check()
{
    try
    {
                    pushfd
                    or dword ptr[esp], 0×100
                    popfd
                    rdtsc
                    nop
                    pushf
                    pushf
    }
    catch ()
    {
        dwEIP = ExceptionInfo→ContextRecord→EIP;
        bByte = ReadByte(dwEIP);
        if bByte ≠ 0×90 ExitProcess();
    }
}
```

Figure 5. Pseudocode of Lampion carrying out anti sandbox check using TF.

The instruction right after the instruction RDTSC is NOP. The byte code for the NOP instruction is 0x90. The exception handler would traverse the ContextRecord structure to locate the address of the instruction in the Extended Instruction Pointer register (EIP) when the exception occurred. The instruction is then compared against the 0x90 byte and the malware will exit if the check fails.

The following screenshot shows the EIP=0x7F0E4E when the exception happened.

Figure 6. Address of the instruction where the exception occurred.

Malware vs Sandbox Authors

For many years, there has been an ongoing cat and mouse game between malware authors crafting evasion techniques to prevent effective analysis, and sandbox authors who research novel ways to defeat those evasions.

This is one of the main drivers that led us at Palo Alto Networks to build our own custom hypervisor for malware analysis. Since we have full control over the software stack, including the virtualization layer, we can react to new and emerging threats. In this particular case, once we had identified the issue with the incorrect emulation of the trap flag, our hypervisor team was able to test and deploy a fix. This evasion issue has since been resolved for any malware sample using this technique.

Palo Alto Networks customers are further protected from malware families using similar sandbox evasion techniques with Cortex XDR or the Next-Generation Firewall with WildFire and Threat Prevention security subscriptions. AutoFocus customers can track the malware discussed here using the Lampion tag. Other similar sandbox evasion techniques that rely on abusing Intel CPU instructions or registers will not work against WildFire.

Indicators of Compromise

Lampion Sample

EB3F2BE571BB6B93EE2E0B6180C419E9FEBFDB65759244EA04488BE7C6F5C4E2

Get updates from Palo Alto

Networks!

Sign up to receive the latest news, cyber threat intelligence and research from us

By submitting this form, you agree to our Terms of Use and acknowledge our Privacy Statement.