research.checkpoint.com /2022/invisible-cuckoo-cape-sandbox-evasion/

Invisible Sandbox Evasion - Check Point Research

⋮ 2/7/2022

February 7, 2022

Research By: Alexey Bukhteyev

Malware uses sandbox evasion techniques to avoid exposing its malicious behavior inside a sandbox and thus prevent detection.

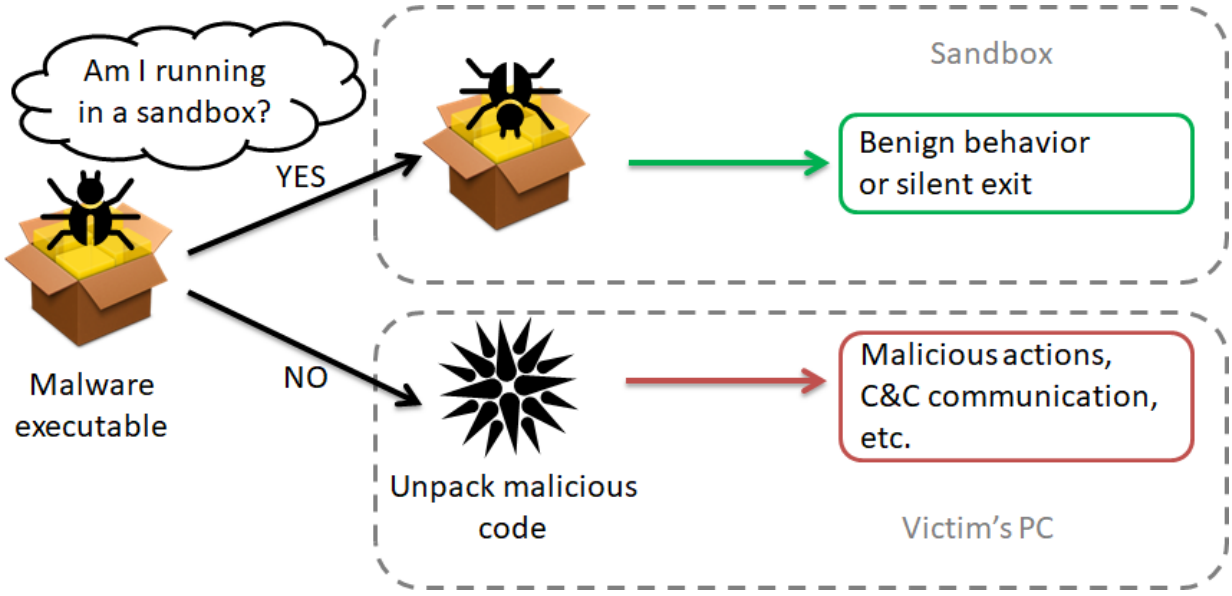

***Figure 1** – Sandbox evasion techniques.*

Common evasion techniques include the use of specific assembly instructions, and looking for specific registry keys or file names. Such evasion techniques can be easily discovered by an experienced analyst, or even detected by a sandbox using signatures.

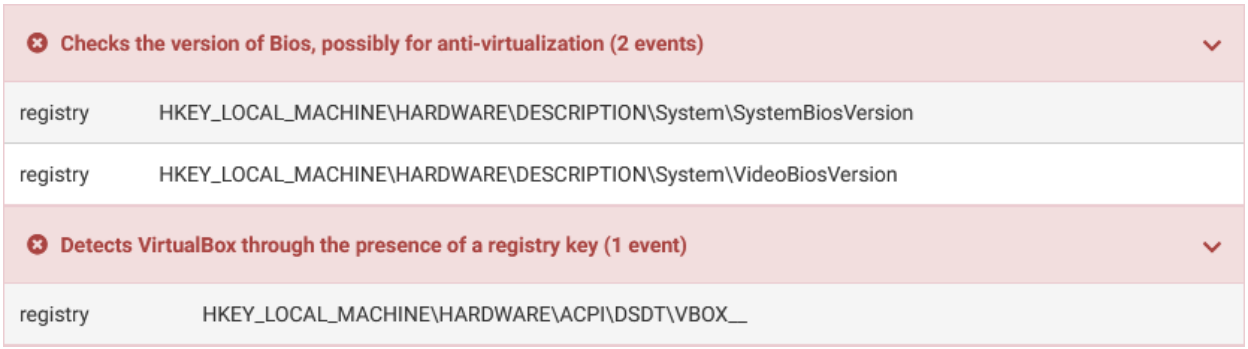

***Figure 2** – Cuckoo Sandbox signatures detected evasion techniques.*

Are there any evasion techniques that completely look like regular code and can't be easily detected? Due to issues we found in the Cuckoo monitor, it is indeed possible. CAPE sandbox is affected as well. We should emphasize that Cuckoo Sandbox and CAPE are the common open source sandbox

environments used in the research and protection area. Therefore, the discovered evasion techniques may affect many researchers and companies. Although we've only tested in Cuckoo and CAPE, the described techniques may be applicable to other sandboxes as well.

Check Point Research reported these issues to CAPE developers, who immediately implemented fixes to mitigate sandbox evasion techniques described in this article.

Cuckoo Sandbox evasion in one Windows API function call

There are more than 400 Native API functions (or Nt-functions) in **ntdll.dll** that are usually hooked in sandboxes. In such a large list, there is enough space for different kinds of mistakes. We checked the hooked Nt-functions and found several issues.

One of them is a discrepancy in the number of arguments in the hooked and the original **NtLoadKeyEx** function. If a function is hooked incorrectly, in kernel mode this may lead an operating system to crash. Incorrect user-mode hooks are not as critical. However, they may lead an analyzed application to crash or can be easily detected.

Let's look at the **NtLoadKeyEx** function. This function was first introduced in Windows Server 2003 and had only 4 arguments:

```
; Exported entry 235. NtLoadKeyEx
; Exported entry 1072. ZwLoadKeyEx
; __stdcall NtLoadKeyEx(x, x, x, x)
public [email protected]
```

Later on, this function changed significantly. Starting from Windows Vista up to the latest version of Windows 10, it has 8 arguments:

```
; Exported entry 318. NtLoadKeyEx
; Exported entry 1450. ZwLoadKeyEx
; __stdcall NtLoadKeyEx(x, x, x, x, x, x, x, x)
public [email protected]
```

However, in the Cuckoo monitor, the **NtLoadKeyEx** declaration still has only 4 arguments:

```
*  POBJECT_ATTRIBUTES TargetKey
*  POBJECT_ATTRIBUTES SourceFile
** ULONG Flags flags
** HANDLE TrustClassKey trust_class_key
```

We found this legacy prototype used in other sources as well. For example, CAPE monitor has the same issue:

```
extern HOOKDEF(NTSTATUS, WINAPI, NtLoadKeyEx,
    __in      POBJECT_ATTRIBUTES TargetKey,
    __in      POBJECT_ATTRIBUTES SourceFile,
    __in      ULONG Flags,
```

```
    __in_opt  HANDLE TrustClassKey
);
```

Therefore, if a sandbox uses any recent Windows OS, this function is hooked incorrectly. After the call to the incorrectly hooked function, the stack pointer value becomes invalid. Therefore, a totally “legitimate” call to the **RegLoadAppKeyW** function, which calls **NtLoadKeyEx**, leads to an exception. This fact can be used to evade **Cuckoo** and **CAPE** sandbox with just a single call to this function.

The evasion technique is quite straightforward. If we want to hide some code when running in a sandbox, we should call the **RegLoadAppKeyW** with valid arguments before this code. In a sandbox, this code will not be reached due to the exception.

```
RegLoadAppKeyW(L"storage.dat", &hKey, KEY_ALL_ACCESS, 0, 0);
// If the application is running in a sandbox an exception will occur
// and the code below will not be executed.

// Some legitimate code that works with hKey to distract attention goes here
// ...
RegCloseKey(hKey);
// Malicious code goes here
// ...
printf("Some malicious code");
```

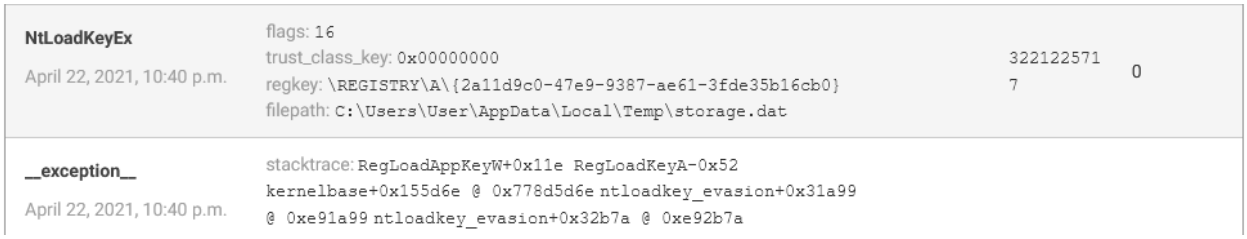

Figure 3 *– Cuckoo Sandbox behavioral analysis report.*

Instead of using **RegLoadAppKeyW,** we can call the **NtLoadKeyEx** function directly and check the **ESP** value after the call. If we want to prevent the application from crashing in the case of an exception inside of the hooked **NtLoadKeyEx** function, the exception handling can also be added.

```
__try
{
        _asm mov old_esp, esp
        NtLoadKeyEx(&TargetKey, &SourceFile, 0, 0, 0, KEY_ALL_ACCESS, &hKey,
&ioStatus);
        _asm mov new_esp, esp
        _asm mov esp, old_esp
        if (old_esp != new_esp)
                printf("Sandbox detected!");
}
__except (EXCEPTION_EXECUTE_HANDLER)
{
```

```
        printf("Sandbox detected!");
}
```

Now that you know what to look for, you won't be tripped up by this evasion technique.

Lack of necessary checks for arguments in a hooked function

Another issue we found in Cuckoo Sandbox and CAPE is a lack of the necessary checks for all arguments in hooked functions.

For example, let's look at a very frequently used function, **NtDelayExecution**, which is called every time you call the **Sleep** function.

```
NTSTATUS
NTAPI
NtDelayExecution(
        IN BOOLEAN              Alertable,
        IN PLARGE_INTEGER       DelayInterval);
```

The second argument of the **NtDelayExecution** function is a pointer to the delay interval value. In the kernel-mode, the **NtDelayExecution** function validates this pointer and can also return the following values:

- **STATUS_ACCESS_VIOLATION** – If the pointer value is not a valid user-mode address.
- **STATUS_DATATYPE_MISALIGNMENT** – If the address is not aligned (*DelayInterval & 3 != 0*).

In a sandbox, the input arguments for **NtDelayExection** and similar functions might not be handled correctly. If we call **NtDelayExecution** with an unaligned pointer for **DelayInterval**, normally it returns the **STATUS_DATATYPE_MISALIGNMENT**. However, in a sandbox, the value for **DelayInterval** may be copied to a new variable without the appropriate checks. In this case, a delay is performed and the returned value will be **STATUS_SUCCESS**. This can be used to detect a sandbox:

```
__declspec(align(4)) BYTE aligned_bytes[sizeof(LARGE_INTEGER) * 2];
DWORD Timeout = 10000; //10 seconds
PLARGE_INTEGER DelayInterval = (PLARGE_INTEGER)(aligned_bytes + 1);
//unaligned

DelayInterval->QuadPart = Timeout * (-10000LL);
if (NtDelayExecution(TRUE, DelayInterval) != STATUS_DATATYPE_MISALIGNMENT)
        printf("Sandbox detected");
```

On the other hand, if an inaccessible address is set for **DelayInterval**, the return code should be **STATUS_ACCESS_VIOLATION**. This can be used to detect a sandbox as well:

```
if (NtDelayExecution(FALSE, (PLARGE_INTEGER)0) != STATUS_ACCESS_VIOLATION)
        printf("Sandbox detected");
```

If the **DelayInterval** argument is not verified before it is accessed, this may lead to an exception in the case of using an invalid pointer. For example, the next code leads the Cuckoo monitor to crash:

```
NtDelayExecution(FALSE, (PLARGE_INTEGER)0xFFDF0000);
```

As stated earlier, normally this call should return **STATUS_ACCESS_VIOLATION** without causing an exception.

These and many other techniques are described in our updated Malware Evasion Encyclopedia. You can also check if your sandbox is affected by sandbox evasion techniques using the InviZzzible tool.

Check Point's Threat Emulation protects against the evasion.