Trường Đại học Công nghệ Thông tin (UIT)

EXERCISE 02 REPORT

Subject: The mechanism of operation of malware

Topic: Advanced Virus Techniques

1. GENERAL INFORMATION:

Class: NT230.N22.ATCL

Num	Họ và tên	Student ID	Email
1	Nguyen Dinh Kha	20520562	20520562@gm.uit.edu.vn
2	Tran Duc Minh	20521617	20521617@gm.uit.edu.vn
3	Nguyen Thi Truong An	19521184	19521184@gm.uit.edu.vn

2. IMPLEMENTATION CONTENT:

Num	Request	Self-assessment
1	Request	100%

The section below of this report is a detailed report document executed by the team.
Implementation requirements:
Write a program to infect executable files (Windows PE 32-bit executable files) with a simple functionality (for educational demo purposes) as requested below. Regarding functionality, the purpose of the payload (reusing the basic virus part from exercise 01):

Display a message on the screen through a "pop-up" window with the window title "Infection by NT230" and the message structure "MSSV01_MSSV02_MSSV03" (student IDs of team members). Note: no quotation marks.
Restore the original functionality of the infected program (without destroying the host program's functionality).
Regarding the ability to evade detection:
a) Research the principle of sandbox detection (such as Cuckoo Sandbox, etc.).
Implement anti-analysis malware (equipped additionally for the original payload) with environmental sensitivity:

Running in a virtual environment,

Running in a sandbox environment.

Ability to detect being debugged.
Once it recognises it's placed in an analysis environment, it will not perform its behavior, not reveal its nature (e.g., the payload doesn't execute the given target code, stops the program, etc.).
Student teams choose 2 different methods in each technique of antidebugging, anti-VM, and anti-sandbox to implement the feature above.

b) Implement an encrypted virus using XOR technique.

a) Hiện thực virus mã hóa (encrypted virus) dùng kỹ thuật XOR.

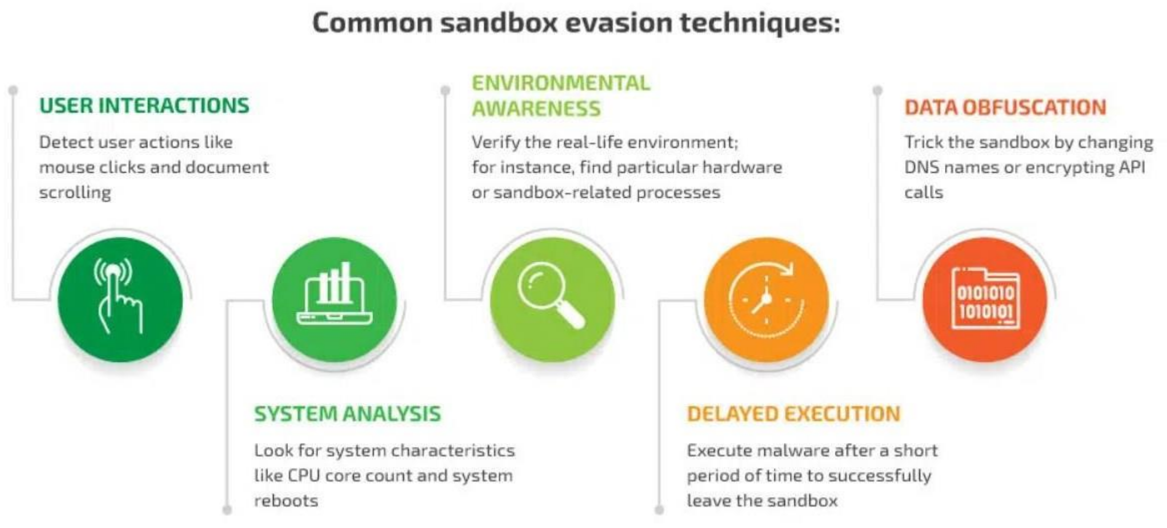
DETAILED REPORT

Steps/Methodology/Learning Content (Screenshot with explanation)

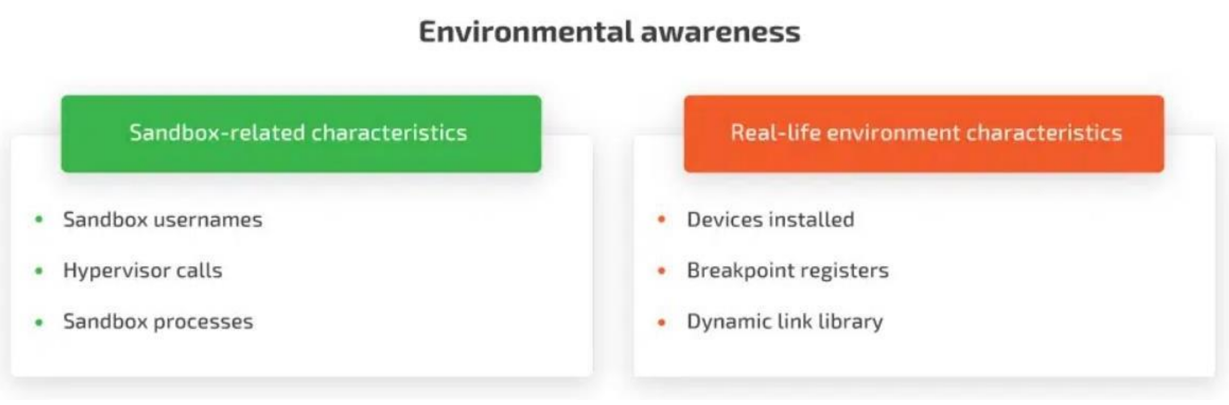
Sandbox Detection Principles (such as Cuckoo Sandbox, etc.):

Sandbox-evading malware is software that can recognise if it's inside a sandbox or virtual environment. These malicious programs don't execute their harmful code until they're outside the controlled environment.

Some typical examples of Sandbox-evading malware:
- Locky ransomware is an example of a virus that bypasses sandboxes. Locky spreads through infected JavaScript code in encrypted DLL files. This malware requires the use of run32dll.exe to execute the DLL. However, run32dll.exe isn't available in sandbox environments, therefore Locky remains undetected.
- HAWKBALL backdoor is used by hackers to target government sectors in Central Asia. This malware exploits Microsoft Office vulnerabilities to distribute payloads and gather system information. It can execute Microsoft's root commands, survey servers, and even verify whether the process it's using is being debugged.
- To avoid detection, malware utilises special sandbox evasion techniques primarily based on detecting user or system interactions or gathering environment awareness. Common sandbox evasion techniques:
  -


Common sandbox evasion techniques:
USER INTERACTIONS
Detect user actions like mouse clicks and document scrolling
ENVIRONMENTAL AWARENESS
Verify the real-life environment; for instance, find particular hardware or sandbox-related processes
DATA OBFUSCATION
Trick the sandbox by changing DNS names or encrypting API calls
0101010 1010101
SYSTEM ANALYSIS
Look for system characteristics like CPU core count and system reboots
DELAYED EXECUTION
Execute malware after a short period of time to successfully leave the sandbox

However, we'll focus on the technique of environmental awareness:



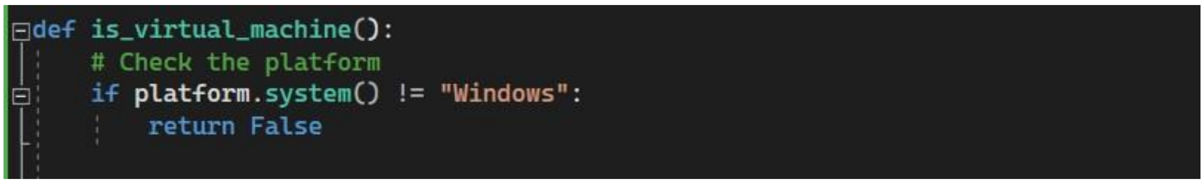
Evironmental Awareness is the name given to a set of advanced techniques used by attackers to attempt to detect sandbox environments, virtual machines, or the presence of forensic tools. To inspect its environment, malware can be programmed to detect devices installed on the infected system or look for indicators unique to virtual environments, such as hypervisor calls, certain file names, and typical sandbox processes.

Furthermore, such malware can detect a sandbox when it's named "sample" or "malware." Additionally, the virus can realise it's in a virtual environment when it finds processes like vmusrvc.exe, boxservice.exe, or vmtoolsd.exe.

Malware can determine if a system is running on a virtual machine (virtual environment) or sandbox when it scans hard drives and registries. It detects the installation of VMware tools, running processes, and registry entries during the scan. Authors of malware programs frequently work to reverse the malware detection techniques built into Sandbox systems. This dedicated malware attack for VMware is becoming more sophisticated. Through detection techniques, malware chains can now avoid sandboxes and breach detection systems.

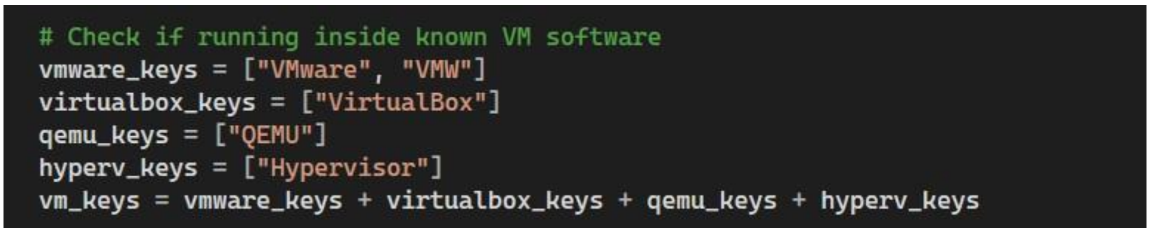
<u>Regarding the concept::</u>

- Virtual Machine Detection Function:



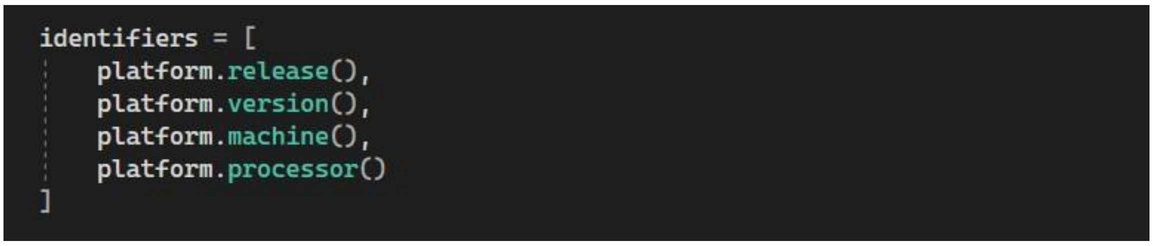
```
def is_virtual_machine():
    # Check the platform
    if platform.system() != "Windows":
        return False
```

First, we check the platform on which the code is running using the platform.system() function. If the platform is not Windows, it immediately returns False, assuming that the virtual machine check is only relevant to Windows.



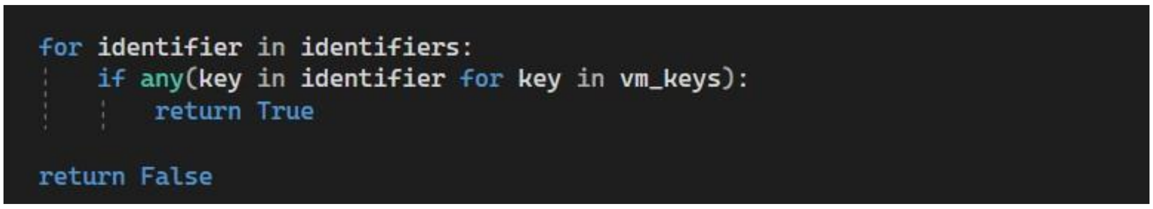
```python
# Check if running inside known VM software
vmware_keys = ["VMware", "VMW"]
virtualbox_keys = ["VirtualBox"]
qemu_keys = ["QEMU"]
hyperv_keys = ["Hypervisor"]
vm_keys = vmware_keys + virtualbox_keys + qemu_keys + hyperv_keys
```

Next, it defines a list (vmware_keys, virtualbox_keys, qemu_keys, hyperv_keys, vm_keys) containing keywords associated with various virtual machine software.



```python
identifiers = [
    platform.release(),
    platform.version(),
    platform.machine(),
    platform.processor()
]
```

The function then creates a list called identifiers containing specific system information obtained using platform.release(), platform.version(), platform.machine(), and platform.processor(). These values are checked for the presence of any virtual machine-related keywords.

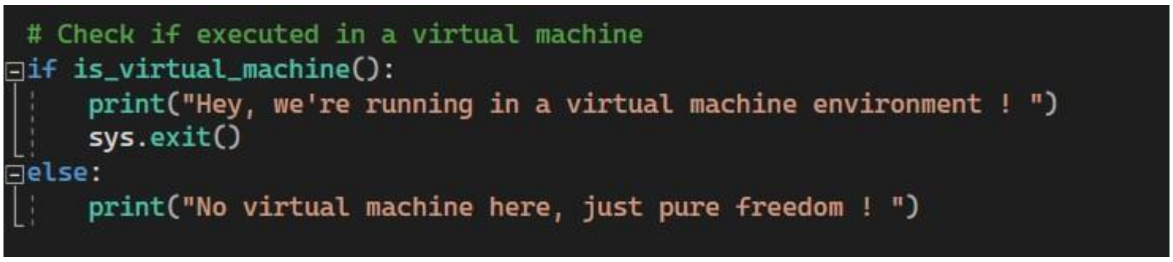


```python
for identifier in identifiers:
    if any(key in identifier for key in vm_keys):
        return True

return False
```

The program proceeds to enter a for loop iterating over each identifier in the list of identifiers. In the loop, it uses the any() function combined with a generator expression to check if any virtual machine-related keywords (vm_keys) are present in the current identifier. If any keyword is found, it returns True, indicating that the code is running on a virtual machine.

If no virtual machine-related keywords are found in any of the identifiers, the function returns False, meaning our malware-infected program is not running on a virtual machine.

- If detected inside a virtual machine:

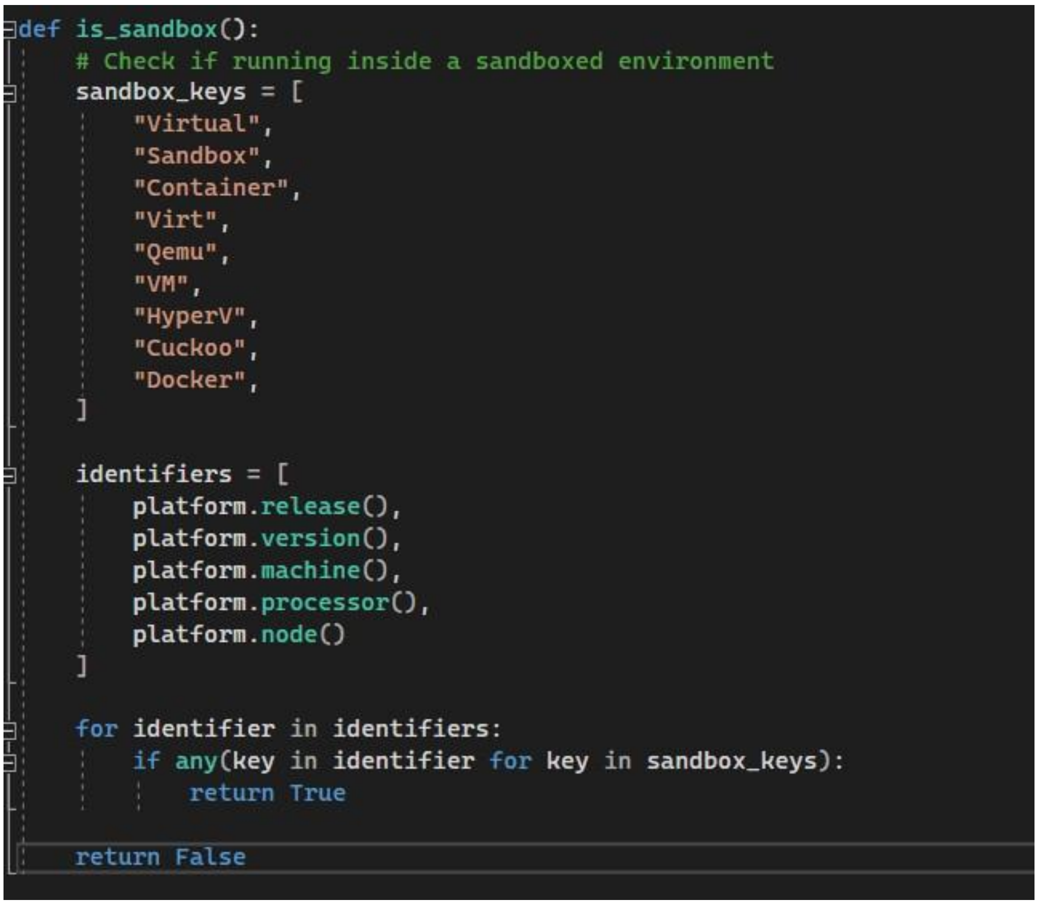


```python
# Check if executed in a virtual machine
if is_virtual_machine():
    print("Hey, we're running in a virtual machine environment ! ")
    sys.exit()
else:
    print("No virtual machine here, just pure freedom ! ")
```

Next, check if it's running on a virtual machine by calling the is_virtual_machine() function. If the function returns True, it prints the message 'Hey, we're running in a virtual machine environment!' and exits the program using sys.exit().

Otherwise, if the function returns False, it prints the message 'No virtual machine here, just pure freedom!' meaning the program is not running on a virtual machine, and it will continue its infecting behaviour.

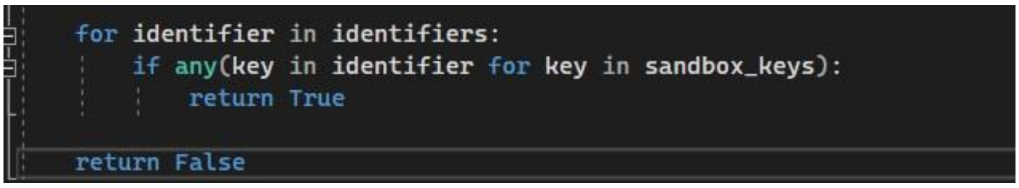
- Sandbox Environment Detection Function:



```python
def is_sandbox():
    # Check if running inside a sandboxed environment
    sandbox_keys = [
        "Virtual",
        "Sandbox",
        "Container",
        "Virt",
        "Qemu",
        "VM",
        "HyperV",
        "Cuckoo",
        "Docker",
    ]

    identifiers = [
        platform.release(),
        platform.version(),
        platform.machine(),
        platform.processor(),
        platform.node()
    ]

    for identifier in identifiers:
        if any(key in identifier for key in sandbox_keys):
            return True

    return False
```

Similarly for the sandbox environment detection function. First, the program identifies a list called sandbox_keys containing keywords associated with various sandbox environments. These keywords are often found in system information.

The function then creates a list called identifiers containing specific system information obtained using platform.release(), platform.version(), platform.machine(), platform.processor(), and platform.node(). These values are checked for the presence of any sandbox-related keywords.

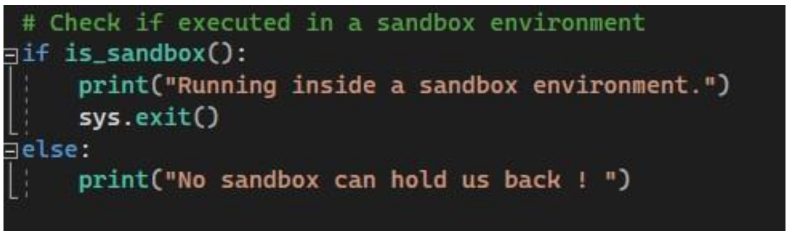


```
    for identifier in identifiers:
        if any(key in identifier for key in sandbox_keys):
            return True

    return False
```

Next, the program enters a for loop iterating over each identifier in the list of identifiers. In the loop, it uses the any() function combined with a generator expression to check if any sandbox-related keywords (sandbox_keys) are present in the current identifier. If any keyword is found, it returns True, indicating that the code is running in a sandbox environment.

Otherwise, if no sandbox-related keywords are found in any of the identifiers, the function returns False, indicating that the code is not running in a sandbox environment.

- If a sandbox environment is detected:

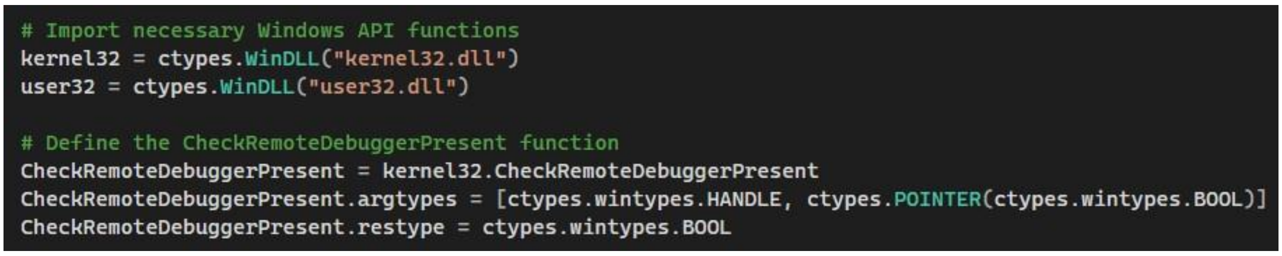


```
# Check if executed in a sandbox environment
if is_sandbox():
    print("Running inside a sandbox environment.")
    sys.exit()
else:
    print("No sandbox can hold us back ! ")
```

We check if the program is running in a sandbox environment by calling the function. If the function returns True, it prints the message 'Running inside a sandbox environment.' and exits the program using sys.exit().

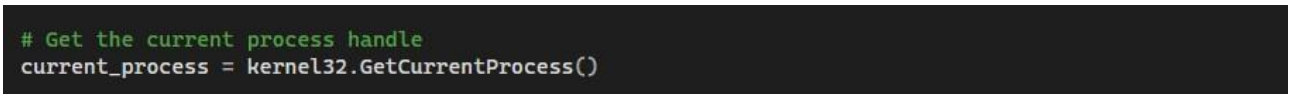
If the function returns False, it prints the message 'No sandbox can hold us back!'

- Debug Detection Function:



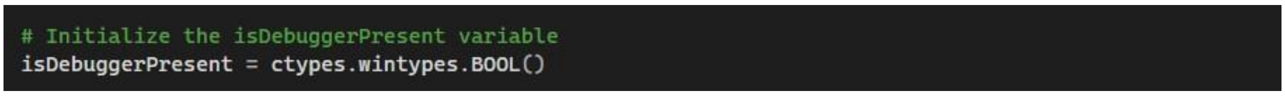
```
# Import necessary Windows API functions
kernel32 = ctypes.WinDLL("kernel32.dll")
user32 = ctypes.WinDLL("user32.dll")

# Define the CheckRemoteDebuggerPresent function
CheckRemoteDebuggerPresent = kernel32.CheckRemoteDebuggerPresent
CheckRemoteDebuggerPresent.argtypes = [ctypes.wintypes.HANDLE, ctypes.POINTER(ctypes.wintypes.BOOL)]
CheckRemoteDebuggerPresent.restype = ctypes.wintypes.BOOL
```

First, we import necessary Windows API functions from the 'kernel32.dll' and 'user32.dll' libraries using ctypes.WinDLL. Next, we define the CheckRemoteDebuggerPresent function using the imported kernel32. The CheckRemoteDebuggerPresent function. It specifies the argument types and return type of the function using the argtypes and restype properties respectively. This step ensures the appropriate data types when calling the function.



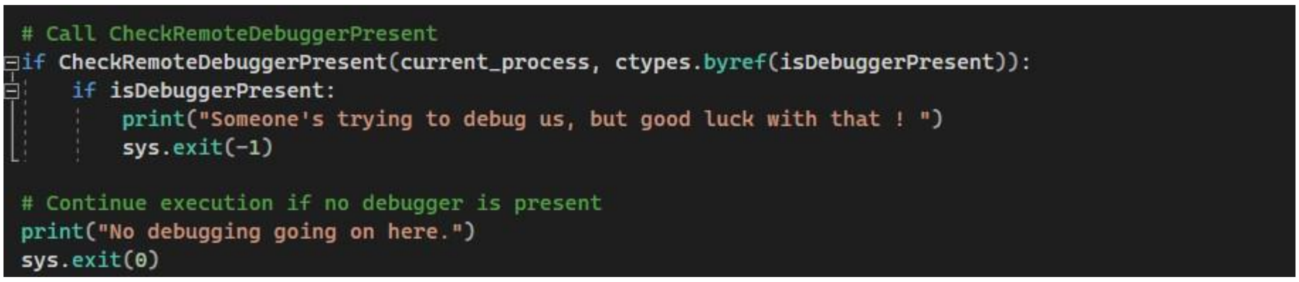
```
# Get the current process handle
current_process = kernel32.GetCurrentProcess()
```

It obtains the current process's handle using kernel32.GetCurrentProcess() and assigns it to the variable current_process. The process handle represents the currently running process.



```
# Initialize the isDebuggerPresent variable
isDebuggerPresent = ctypes.wintypes.BOOL()
```

Next, initialize the variable isDebuggerPresent of type ctypes.wintypes.BOOL (a boolean type for Windows) to store the debugger presence check result.



```
# Call CheckRemoteDebuggerPresent
if CheckRemoteDebuggerPresent(current_process, ctypes.byref(isDebuggerPresent)):
    if isDebuggerPresent:
        print("Someone's trying to debug us, but good luck with that ! ")
        sys.exit(-1)

# Continue execution if no debugger is present
print("No debugging going on here.")
sys.exit(0)
```

We call the CheckRemoteDebuggerPresent function using the current process handle and refer to the variable isDebuggerPresent as the argument. This function checks for the presence of a remote debugger and sets the isDebuggerPresent variable accordingly. We call the CheckRemoteDebuggerPresent function using the current process handle and refer to the isDebuggerPresent variable as the argument. This function checks for the presence of a remote debugger and sets the isDebuggerPresent variable accordingly.

If the debugger check succeeds (no errors occur), it checks the value of isDebuggerPresent. If it is True, it means a debugger is present, and it prints the message 'Someone's trying to debug us, but good luck with that!'. Then, it exits the program using sys.exit(-1).

If no debugger is detected (the debugger check fails or isDebuggerPresent is False), it prints the message 'No debugging going on here.'"---

END