“Filesnfer” Tool (C#, Python) – One Night in Norfolk

Published: 2019-05-07 · Archived: 2026-04-06 01:21:08 UTC

On 6 May 2019, Symantec published reporting on a series of tools possibly used by APT3 (or a broader China-based espionage apparatus), including a previously publicly unreported backdoor dubbed “Filesnfer.”* Several hashes were made available for this malware, including one for a variant written in C++, one for a variant written in Python (compiled via Py2Exe), and one purportedly written in PowerShell.

The hash for the PowerShell file is unavailable on VirusTotal; however, analysis of the Python code can be used to identify a different file uploaded to the Hybrid Analysis platform that is delivered via a PowerShell loader, written in C#, and contains significant code-level and unique-string overlaps with the Python variant. This file was *also* not made available for download on the platform, but the strings for the loaded C# code in this sandbox run are enough to find an additional sample of the entire decompiled code on VirusTotal.

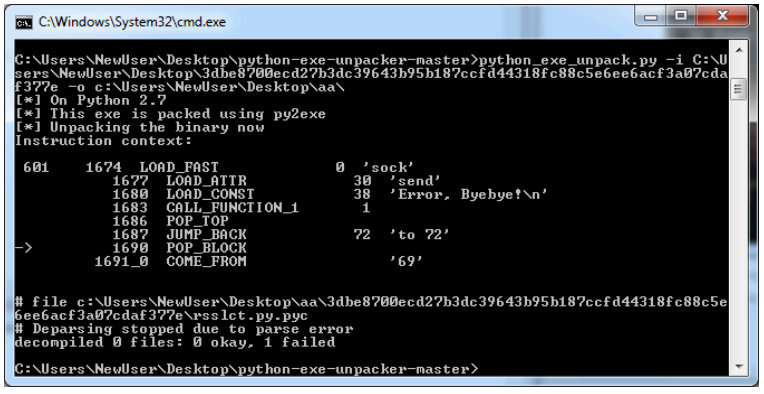
This blog contains a brief technical overview of each of these variants, and the pivoting method described. If you’re just here for the C# (“PowerShell”) hash: 8de3b2eac3fa25e2cf9042d1b952f0d9. For analysis of these files, keep reading.

* (Symantec notes that the connection between this backdoor and APT3 was provided to them through collaboration with another vendor).

Py2Exe Filesnfer

MD5: a3932533efc04ac3fe89fb5b3d60128a
SHA1: 2a01d103b2bb66cba2cdb201f09933fee2055db3
SHA256: 3dbe8700ecd27b3dc39643b95b187ccfd44318fc88c5e6ee6acf3a07cdaf377e

As Symantec notes, this file is a compiled Python executable built using Py2Exe. The python-exe-unpacker available on Github can be used to partially decompile the code, although it did encounter an error during this analysis that prevented a full dump from being created:



Decompilation error of the Py2Exe-compiled file

Despite this limitation, it's still possible to infer some of the functionality and, as described above, identify a complete dump of the C# version through additional pivoting. The Python code defines several classes including:

– DirItem
– TransArgs
– LoadCert
– Monitor
– Timer

In addition, the following functions are defined:

– print_log(xtype, pstr, btime=True)
– Recv(sock, size, timeout=3, bLoop=True)
– SendPacket(sock, buf)
– RecvPacket(sock, size=0)
– GenFileName(strFile)
– UnitConv(fSize)
– PutFileData(pSocket, sourceFile, Length, types=0, lLen=0, Speeds=10485760, bZip=True)
– GetFileData(gSocket, destFile, Length, types, lLen, bZip=True)
– SendDirList(sock, filepath)
– ServerLoopPro
– ServerX(host, port)
– handler(signum, frame)

Of these, ServerLoopPro failed to decompile. The rest of the available code can be used to determine the intent and functionality of this section, however. On execution, the "main" block of the code defines variables that call the Monitor, Timer, and LoadCert classes, which set a backconnect Boolean to "True," initiates a "timer" interval, and defines a certificate and key respectively.

"Main" initiating the Monitor and Timer classes

The malware then performs a series of checks on any supplied parameters to make sure that they are properly formatted. The code then calls the "ServerX" function if a host and port have been specified.

ServerX and ServerLoopPro

The ServerX function operates as the second “parent” routine within the decompiled code. ServerX will read a specified host and port and attempt to open a socket connection to this location (using the certificate and key defined earlier). From here, it creates a thread for the ServerLoopPro function.

ServerLoopPro did not properly decompile during this analysis, but the code provided offers insight into the functionality. The code references several “orders” that are compared to integer constants. If these conditions are not met, the code jumps to a different location, often performing a similar comparison against a different value. In addition, there are several references to the functions defined above, such as PutFileData and GetFileData:

```
506      455  LOAD_GLOBAL           24  'PutFileData'
         458  LOAD_FAST              0  'sock'
         461  LOAD_FAST             13  'filePath'
         464  LOAD_CONST            11  ''
         467  LOAD_CONST            13  1
         470  LOAD_FAST             16  'localLen'
         473  LOAD_FAST              9  'iSpeed'
         476  LOAD_FAST             10  'bZip'
         479  CALL_FUNCTION_7        7
         482  POP_TOP

507      483  CONTINUE              72  'to 72'

508      486  LOAD_FAST              8  'Order'
         489  LOAD_CONST            14  202
         492  COMPARE_OP             2  '=='
         495  POP_JUMP_IF_FALSE    705  'to 705'

509      498  LOAD_GLOBAL           15  'print_log'
         501  LOAD_GLOBAL           16  'ltype'
         504  LOAD_ATTR             17  'warning'
         507  LOAD_CONST            15  "%s:%d upload file '%s'.\n"
         510  LOAD_FAST              4  'host'
         513  LOAD_FAST              5  'port'
         516  LOAD_FAST             13  'filePath'
```

References to “PutFileData” and “Order”

A likely inferences is that these functions are part of a command structure contained within the ServerLoopPro routine. Thus, analyzing these functions lends insight into the likely commands available to the attackers. PutFileData and GetFileData read and write content to and from the infected device. SendDirList enumerates the contents of a specified directory back to the attacker:

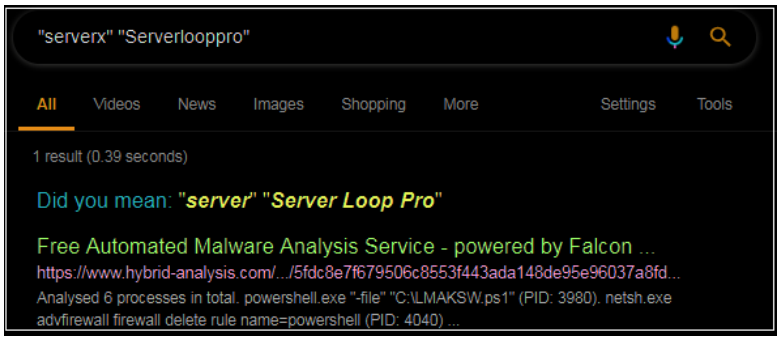
```
if len(matchstr) > 0:
    pattern = re.compile(matchstr.replace('.', '\\.').replace('*', '.*').replace('?', '.?') + '$', re.IGNORECASE)
dlist = os.listdir(findpath)
for fs in dlist:
    if pattern:
        if not re.match(pattern, fs):
            continue
    item = DirItem()
    path = os.path.abspath(findpath + Separator + fs)
    if os.path.isdir(path):
        item.m_isDir = True
    stat = os.lstat(path)
    item.m_mtime = stat[8]
    item.m_filelen = stat[6]
    item.m_filename = fs
    SendPacket(sock, str(item))

item = DirItem()
item.m_State = True
SendPacket(sock, str(item))
return
```

Snipper of "SendDirList"

PowerShell/C# Filesnfer

MD5: 8de3b2eac3fa25e2cf9042d1b952f0d9
SHA1: 23b1c6b81fd7d4d6ea0bc81109ce886a45967180
SHA256: 6972ba198ed0d30de9f66be5777ecdba2d657078f138325ee6db225c20b29e6e

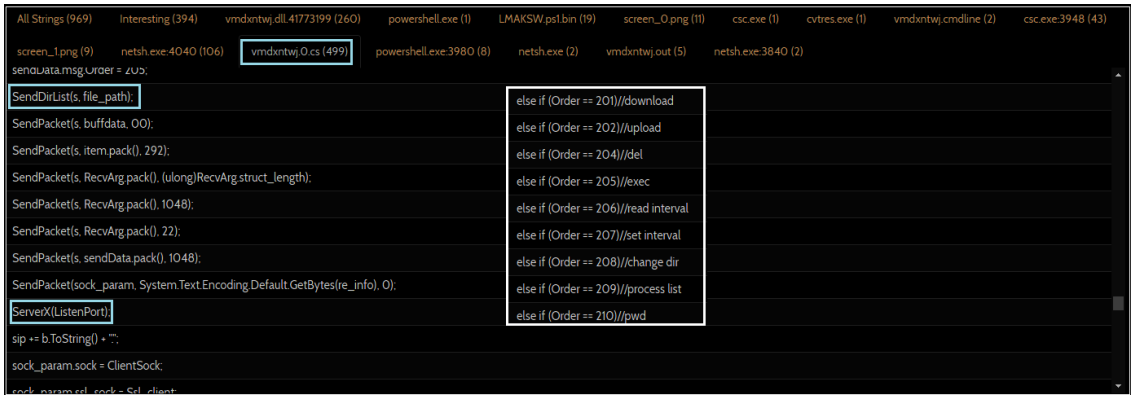
As noted above, the hash Symantec provided for the "PowerShell" version of this tool is not available on VirusTotal; however, searching for the unique function strings ServerLoopPro and ServerX leads to a Hybrid Analysis sandbox report containing a likely variant.



Pivoting results from unique strings in Python file

Although the file is not available for download, analysis of the strings in memory suggests an encrypted payload delivered via a PowerShell file (that also opens a firewall exception). In addition, the malware launches "csc.exe," the legitimate Microsoft C# compiler, and appears to compile and run a file with this tool. Analysis of the C# code in the sandbox report reveals the similar function names as well as the suspected "order" commands:

Encrypted contents of the PowerShell file



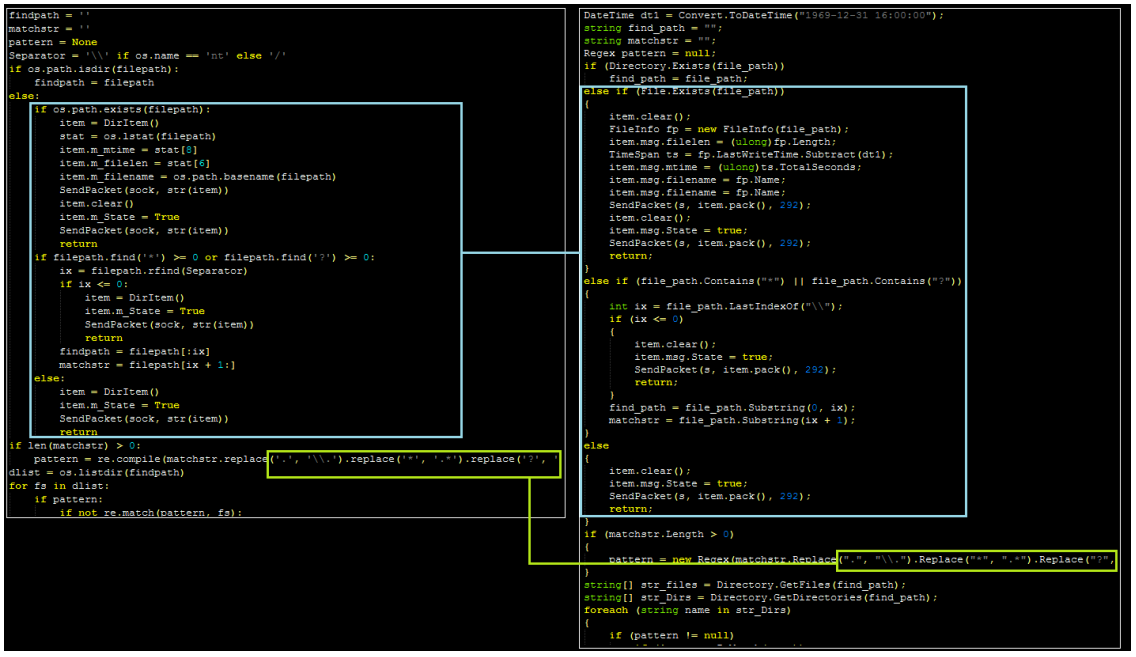
Identical function names within the C# code as well as "order" references

A VirusTotal content search on these unique strings reveals a nearly identical C# file (8de3b2eac3fa25e2cf9042d1b952f0d9). Much like the Python variant, the C# variant defines several similar structures and classes at the start:

```
public struct DirItem
{
    public Boolean State;
    public ulong mtime;
    public Boolean isDir;
    public ulong filelen;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 260)]
    public string filename;
}

public struct TransItem
{
    public int Order;
    public int Speed;
    public Boolean Bzip;
    public ulong Length;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 1024)]
    public string m_path;
}
class TransArgs
{
    public Socket sockClient;
    public IPAddress ip;
    public int port;
    public byte[] prebuf;
}
class CDir
```

Class and structure definitions within the C# variant of Filesnfer

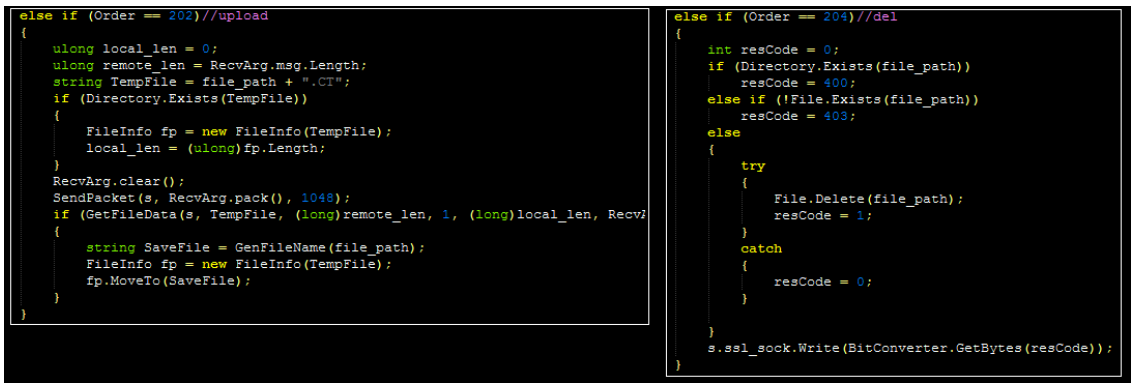
In some cases, the code overlaps are almost identical:



Directory listing function in Python variant (left) compared to C# variant (right)

As a whole, the C# variant functions in largely the same fashion, with some slight differences. The “Main” routine only requires a listening port (in this case, port 47000) to pass to ServerX (interestingly, the entire file is also held within a class named “xserver”). ServerX can allow the malware to act as a proxy in addition to calling ServerLoopPro; in turn, ServerLoopPro can receive “order” (just as inferred in the Python sample). Several of these orders are commented by the malware author, and include the following (largely self-explanatory) options:

– list
– download
– upload
– del
– exec
– read interval
– set interval
– change dir
– process list



Example of command structure within the C# variant

Concluding Thoughts

Symantec's article notes that the Filesnfer backdoor was observed by a undisclosed additional vendor, and that corroborating analysis tying it to the APT3 threat actor is unavailable at this time; as such, this blog post is only intended to highlight the functionality of this backdoor (and identify a publicly available variant of the C#/PowerShell version). Regardless of the operator, the backdoor is capable of providing several basic but important capabilities, including file transfers, command execution, and basic reconnaissance.

Post navigation
