Threat Analysis: Carbon Black Threat Research Dissects PNG Dropper | Carbon Black

Published: 2017-08-18 · Archived: 2026-11-28 02:18:01 UTC

UPDATE (8/14/17: After posting the original analysis, the Carbon Black Threat Research team received numerous requests for the tools to extract the second stage payload from the initial PNG_dropper file. As a result, the source code and compiled binaries are being made public and are posted to the Threat Research external GIT repository (https://github.com/carbonblack/threat-research-tools/tree/master/png_extract).

This report and the associated tools, scripts, rules, and YARA signatures are being provided by the Carbon Black Threat Research Team to assist other researchers and practitioners investigating a dropper family that has been used in recent targeted attacks.

The samples analyzed in this report were collected from Carbon Black's binary pipeline, incident responders, and in publicly available malware repositories. The tools, as well as the Carbon Black specific rules and queries developed to assist in analysis and detection, can be located in the Carbon Black User Exchange.

The dropper family, referred to internally as PNG_dropper, was observed being used as a second stage tool in different targeted attacks. One of final payloads that is created by this dropper is an Uroburos variant used by the Turla group, which traditionally operates out of Russia. This technique is being used to allow the attackers to conceal their secondary payloads, bypassing different AV products. Attackers, regardless of their skills and motives, often attempt to wrap malicious code in a way that will seem innocuous to practitioners and security products.

The PNG_dropper technique does a good job of achieving this goal, as the malicious payload is split across several files, concealing the data using the standard PNG encoding. Portable Network Graphic (or PNG) files were created over 20 years ago for the purpose of transferring compressed graphic files across the internet, and benign PNG files are present on computers in every environment and network. Developing a tool like this to deliver malicious payloads demonstrates the attackers' level of sophistication, creativity, and desire to deliver targeted attacks and remain undetected for as long as possible.

This malware is command-line driven and used to establish secondary or ad-hoc Command and Control (C2) channels as attackers move laterally through an infected network. The PNG_dropper utilizes modified programs whose source code is publicly available (source code links are listed in the technical details section).

Primarily, an altered version of JPEGview, which is an highly configurable open-source tool for viewing and editing image files, was used to call an attacker added code. That malicious function loads several embedded PNG resources, portions of which are decoded using standard API image libraries resulting in sections of a malicious payload.

During our research we observed two different second-stage payloads, an Uroburos loader and a Crypto-loader (the latter requires additional command line arguments to execute properly). The Uroburos loader, and associated files, have been documented publicly from numerous different sources, subsequently this report will focus on the PNG_dropper and Crypto-loader variants. It should be noted that any second stage payload could be used in conjunction with the PNG_Dropper, as this is a technique to conceal a malicious secondary payload in a file that appears to be innocuous.

Technical Details

PNG_Dropper

The PNG_dropper family primarily uses a modified version of the publicly available tool JPEGView.exe (version 1.0.32.1 – both x86 and x64 bit versions). Carbon Black Threat Research also observed where PNG_dropper malware was seen compiled into a modified version of the 7-Zip File Manager Utility (version 9.36.0.0 – x64 bit).

In both cases the executable still retains much of the original code from the legitimate binary, as well as the original icon. The size of the modified binary is much larger than the original due to the PNG resources that have been placed into the malicious versions. While researching this malware dropper we observed where the file was commonly named as rpcepu~1.exe, variations of ~DFxxxx.tmp (ex. ~DF4a20.tmp), or 7zsetup.exe. The file has observed being run directly, while other variants (compiled as Dlls) are executed using rundll32.exe.

Once executed the modified binary will call a function added by the attacker. The below images display the additional functions. In each image the left hand portion, highlighted in green is the legitimate binary, and the portion on the right side (highlighted in red) is the malicious function.



7Zip Comparison



JPEGView Comparison

The PNG_dropper family embeds a portion of the malicious payload into each PNG resource, which are compiled into the dropper. The images below show how the PNG resources would appear in the dropper itself. The image on the far left shows that in this particular sample there are a total of 8 PNG type resources. The middle image shows the data of one of the PNG resources itself. The final image shows how one of the PNG resources would be rendered by a image viewer. It should be noted that this is a legitimate PNG file, and is not a scenario where data is simply appended to the end of a file.



PNG Resource Overview

The attacker added code, in the legitimate JPEGViewer, will load each resource, and then use the GDI+ class API to read the the data from each PNG resource into a buffer (highlighted in red in the image below). Several GDI+ methods are then called to gather information about the PNG image file, as well as a call to allocate a memory buffer (highlighted in blue).

A call to the Bitmap.LockBits method is then called with a pointer to the loaded PNG buffer along with a pixel format of 16bppARGB1555 (highlighted in in green in the image below). Once the buffer is decoded the data is copied into a secondary memory location, where the additional decoded portions will be appended. It should be noted that many of the arguments and calls were renamed to assist readers in reviewing the code. The text in green is additional comments added by the analyst.



PE extraction Functions

The malicious code will then locate the PE headers for the decoded data (which is a binary) taking the data offset and injecting the payload back into the current process' memory space (highlighted in green in the image below). The address entry point for the decoded payload along with the arguments 0xDEADBEAF is then called directly by the function. These commands are highlighted in blue in the image below. At this point PNG_dropper has completed its primary objective and the secondary payload begins to execute. It should be noted that that this hex string has been used in other families used in targeted attacks, namely Uroburos.



Code Injection and Payload execution

The below image displays a narrower view of the above function, where the arguments and PE header information is being pushed on the stack before the entry point address is being called.



Process Setup and PE header calculation

Second Stage Payload

Uroburos Variant

The two images below show the overall execution flow of the PNG_dropper and the two observed second stage payloads. The first image highlights the variant using a Uroburos Dropper as a second stage variant. The main PNG_dropper will load the PNG resources, decoding them and them injecting the PE back into the current process. From there the second stage Uroburos implant will take over. Due to the visibility and previous analysis of the Uroburos family this second stage will not be analyzed further in this report.



PNG_dropper Uroburos Dropper Payload

CryptoWrapper Variant

The second image below shows the overall execution flow of the PNG_dropper with the second stage CryptoWrapper implant. This image depicts how the PNG_dropper loads and decodes the PE file from the PNG resources, and then injects the code back into the current process. The CryptoWrapper implant will then begin to execute. From a high level the CryptoWrapper will perform a basic check, and then load a binary resource compiled into its code. This resource is encrypted, once decrypted the third stage implant will be injected back into the current process and will begin to execute.



PNG_dropper CryptoWrapper Dropper Payload

Once the the CryptoWrapper implant executes it will initially check to see if two of the arguments passed on the stack are the values 0xDEADBEAF (this process is highlighted in red in the image below). The code will then parse the command line of the current process (which is the original command line run with the PNG_dropper (this is due to the fact that the CryptoWrapper code was copied into the original process). This process is highlighted in green in the image below. One of the final pieces of the CryptoWrapper code is to decrypt the compiled binary resource and then inject that code into the current process before calling the final third stage implant.



CryptoWrapper Initial Check

As referenced earlier the CryptoWrapper code will parse the command line of the original process (whose context it is running under). It expects at least two strings to be passed as arguments (however there is additional code that will copy the third and fourth arguments into a memory location for later use). The first two arguments are used as a base for the key and IV that will be used to decrypt the binary resource compiled into the Cryptowrapper executable.

The image below illustrates how the string arguments are modified and used in the decryption process. After a call to retrieve the command line, the first argument will ultimately be used as the input passed to the bcrypt SHA-256 API, the result is then hashed again by the same function. The final result will be used as the key to decrypt the binary resource using the AES algorithm. The second argument is prepended with the value 0x31 ("1"), and then the first 8 bytes of that string are used as the IV to decrypt the binary resource. It should be noted that these strings are treated as wide characters. There is no mandatory length for the first argument however the second argument must have a length of seven or greater or the code will error and terminate.



CryptoWrapper Argument logic

The below image depicts the actual code responsible for gathering the command line arguments, and checking the overall number of arguments passed upon the initial execution (highlighted in red). The code, highlighted in green, shows where the first argument is passed to the bcrypt function which will create a SHA-256 hash of the original string, and then pass the result back to the same function creating the final hash object used as a key.



CryptoWrapper argument parsing and hashing

The CryptorWrapper code will then load the embedded binary resource into memory (highlighted in red in the below image). The code, highlighted in green, is responsible for prepending the value 0x31 to the second

argument. The decryption portion will then be called (highlighted in blue). The decryption function uses the BCrypt method to decrypt the binary resource using the supplied key and IV as parameters to the AES algorithm. It should be noted that the function contains code to both encrypt and decrypt, however an argument passed to the function will always result in the data being decrypted.



CryptoWrapper IV alteration and decryption

The CryptoWrapper code expects that the decrypted data is a PE file. The code will look for the same type of PE structures and offsets that were observed in the original PNG_dropper before it injected the second payload back into the current process. If the payload does not decrypt correctly (due to the wrong key and IV being provided) the code will error out and terminate. The ultimate payload is being researched and will be detailed in future post.

IOCs

Hashes

MD5
f84aa30676d2c05ed290b43c4c1e2d4c
ae2ec6d8e455c674d5486ce198d4d46e
7a1a174dd24d3f88454615102a074600
SHA1
6459858057805106700924 69b7627a23803eefd1

17941a20d86c9518c168c7f765785095a57246a3
ba221b85c1923866ce2ec3cd0824970216052c82
SHA256
eeb7784b77d86627bac32e4db20da382cb4643ff8eb86ab1abaebaa56a650158
69389f0d35d003ec3c9506243fd264afefe099d99fcc0e7d977007a12290a290
3a5918c69b6ee801ab8bfc4fc872ac32cc96a47b53c3525723cc27f150e0bfa3

Yara Signature

```
rule PNG_dropper:RU TR APT

{

  meta:

    author = “CarbonBlack Threat Research”

    date = “2017-June-11”

    description = “Dropper tool that extracts payload from PNG resources”

    yara_version = “3.5.0”

    exemplar_hashes = “3a5918c69b6ee801ab8bfc4fc872ac32cc96a47b53c3525723cc27f150e0bfa3,
69389f0d35d003ec3c9506243fd264afefe099d99fcc0e7d977007a12290a290,
eeb7784b77d86627bac32e4db20da382cb4643ff8eb86ab1abaebaa56a650158 “

  strings:

$s1 = “GdipGetImageWidth”

$s2 = “GdipGetImageHeight”

$s3 = “GdipCreateBitmapFromStream”

$s4 = “GdipCreateBitmapFromStreamICM”
```

```
$s5 = “GdipBitmapLockBits”

$s6 = “GdipBitmapUnlockBits”

$s7 = “LockResource”

$s8 = “LoadResource”

$s9 = “ExpandEnvironmentStringsW”

$s10 = “SetFileTime”

$s11 = “memcmp”

$s12 = “strlen”

$s13 = “memcpy”

$s14 = “memchr”

$s15 = “memmove”

$s16 = “ZwQueryValueKey”

$s17 = “ZwQueryInformationProcess”

$s18 = “FindNextFile”

$s19 = “GetModuleHandle”

$s20 = “VirtualFree”

$PNG1 = {89 50 4E 47 [8] 49 48 44 52} //PNG Header

$bin32_bit1 = {50 68 07 10 06 00 6A 07 8?} //BitmapLockBits_x86

$bin64_bit1 = {41 B? 07 10 06 00} //BitmapLockBits_x64

$bin64_bit2 = {41 B? 07 00 00 00}//BitmapLockBits_x64

$bin32_virt1 = {6A 40 68 00 10 00 00 50 53} //VirtualAlloc_x86

$bin64_virt1 = {40 41 B? 00 10 00 00}//VirtualAlloc_x64

    condition:

    uint16(0) == 0x5A4D and// MZ header check

    filesize < 6MB and

    18 of ($s*) and
```

```
    (#PNG1 > 7) and
//checks for multiple PNG headers
        ((#bin32_bit1 > 1 and $bin32_virt1) or
//More than 1 of $bin32_bit and $bi32_virt1
        (for 1 of ($bin64_bit*) : (# > 2) and $bin64_virt1))
//1 of $bin64_bit – present more that 2 times and $bin64_Virt1
}
```

Source: https://web.archive.org/web/20190921183557if_/https://www.carbonblack.com/2017/08/18/threat-analysis-carbon-black-threat-research-dissects-png-dropper/