Threat Hunter Playbook

Search this book...

KNOWLEDGE LIBRARY

PRE-HUNT ACTIVITIES

GUIDED HUNTS

Contents

DLL Process Injection via CreateRemoteThread and LoadLibrary

Hypothesis

Adversaries might be injecting a dll to another process to execute code via CreateRemoteThread and LoadLibrary functions.

Technical Context

Get Handle to Target Processs

The malware first needs to target a process for injection (e.g. svchost.exe). This is usually done by searching through processes by calling a trio of Application Program Interfaces (APIs) > CreateToolhelp32Snapshot, Process32First, and Process32Next. After finding the target process, the malware gets the handle of the target process by calling OpenProcess.

There are two processes involved in this attack > your DLLInjector process (Process A), and the remote process you want to inject with a DLL (Process B). To interact with the remote process, Process A must call OpenProcess() while passing the remote process process ID as an argument. OpenProcess will then return to Process A a Handle to Process B. Having a Handle to the remote process allows Process A to interact with it in powerful ways. Process A can allocate memory, write memory, and create an execution thread in Process B by calling functions like VirtualAllocEx, WriteProcessMemory, and CreateRemoteThread and passing the Handle to Process B as an argument to those functions.

Get address of the LoadLibraryA function

Kernel32.dll is loaded into every Windows process, and within it is a useful function called LoadLibrary. When LoadLibrary is called in a certain process, it maps a DLL into that process. LoadLibrary needs to know what DLL to load, so you need to provide it the path to the DLL on your system. LoadLibrary will then find the DLL at that path and load that DLL into memory for you. Note > LoadLibraryA is the function name. “A” means you provide the DLL path as an ASCII string.

Allocate Memory for DLL

Why do we write the DLL path to Process B using VirtualAllocEx and then WriteRemoteMemory? This is because LoadLibrary needs to know what DLL you want to inject. The string it accepts as a parameter needs to be present in Process B memory. The malware calls VirtualAllocEx to have a space to write the path to its DLL. The malware then calls WriteProcessMemory to write the path in the allocated memory.

Execute Code

Finally, to have the code executed in another process, the malware calls APIs such as CreateRemoteThread, NtCreateThreadEx, or RtlCreateUserThread. The latter two are undocumented. However, the general idea is to pass the address of LoadLibrary to one of these APIs so that a remote process has to execute the DLL on behalf of the malware. The CreateRemoteThread function creates a thread in the virtual address space of an arbitrary process.

Use CreateRemoteThread to create a remote thread starting at the memory address (which means this will execute LoadLibrary in the remote process). Besides the memory address of the remote function you want to call, CreateRemoteThread also allows you to provide an argument for the function if it requires one. LoadLibrary wants the memory address of where you wrote that DLL path from earlier, so provide CreateRemoteThread that address as well.

Offensive Tradecraft

This technique is one of the most common techniques used to inject malware into another process. The malware writes the path to its malicious dynamic-link library (DLL) in the virtual address space of another process, and ensures the remote process loads it by creating a remote thread in the target process.

Pre-Recorded Security Datasets

Metadata	Value
docs	https://securitydatasets.com/notebooks/atomic/windows/defense_evasion/SDWIN-190518221344.html
link	https://raw.githubusercontent.com/OTRF/Security-Datasets/master/datasets/atomic/windows/defense_evasion/host/empire_dllinjection_LoadLibrary_CreateRemoteThrea

Download Dataset

```
import requests
from zipfile import ZipFile
from io import BytesIO

url = 'https://raw.githubusercontent.com/OTRF/Security-Datasets/master/datasets
zipFileRequest = requests.get(url)
zipFile = ZipFile(BytesIO(zipFileRequest.content))
datasetJSONPath = zipFile.extract(zipFile.namelist()[0])
```

Read Dataset

```
import pandas as pd
from pandas.io import json

df = json.read_json(path_or_buf=datasetJSONPath, lines=True)
```

Analytics

A few initial ideas to explore your data and validate your detection logic:

Analytic I

Look for any use of the CreateRemoteThread function to create a remote thread starting at the memory address (which means this will execute LoadLibrary in the remote process).

Data source	Event Provider	Relationship	Event
Process	Microsoft-Windows-Sysmon/Operational	Process wrote to Process	8

Logic

```
SELECT `@timestamp`, Hostname, SourceImage, TargetImage
FROM sparkTable
WHERE Channel = "Microsoft-Windows-Sysmon/Operational"
    AND EventID = 8
    AND lower(StartModule) LIKE "%kernel32.dll"
    AND StartFunction = "LoadLibraryA"
```

Pandas Query

```
(
df[['@timestamp', 'Hostname', 'SourceImage', 'TargetImage']]

[(df['Channel'] == 'Microsoft-Windows-Sysmon/Operational')
    & (df['EventID'] == 8)
    & (df['StartModule'].str.lower().str.contains('.*kernel32.dll', regex=True)
    & (df['StartFunction'] == 'LoadLibraryA')
]
.head()
)
```

Analytic II

You can look for the same file being created and loaded. The process that creates the file and loads the file are not the same.

Data source	Event Provider	Relationship	Event
Module	Microsoft-Windows-Sysmon/Operational	Process loaded Dll	7
File	Microsoft-Windows-Sysmon/Operational	Process created File	11

Logic

```
SELECT f.`@timestamp` AS file_date, m.`@timestamp` AS module_date, f.Hostname,
FROM sparkTable f
INNER JOIN (
    SELECT `@timestamp`,Hostname,Image,ImageLoaded
    FROM sparkTable
    WHERE Channel = "Microsoft-Windows-Sysmon/Operational"
        AND EventID = 7
    ) m
ON f.TargetFilename = m.ImageLoaded
WHERE f.Channel = "Microsoft-Windows-Sysmon/Operational"
    AND f.EventID = 11
    AND f.Hostname = m.Hostname
```

Pandas Query

```
imageLoadDf = (
df[['@timestamp', 'Hostname', 'Image', 'ImageLoaded']]

[(df['Channel'] == 'Microsoft-Windows-Sysmon/Operational')
    & (df['EventID'] == 7)
]
)

fileCreateDf = (
df[['@timestamp', 'Hostname', 'Image', 'TargetFilename']]

[(df['Channel'] == 'Microsoft-Windows-Sysmon/Operational')
    & (df['EventID'] == 11)
]
)

(
pd.merge(imageLoadDf, fileCreateDf,
    left_on = ['ImageLoaded','Hostname'], right_on = ['TargetFilename','Hostnam
)
```

Known Bypasses

Idea	Playbook
Instead of passing the address of the LoadLibrary, adversaries can copy the malicious code into an existing open process and cause it to execute (either via a small shellcode, or by calling CreateRemoteThread) via a technique known as PE injection.	
The advantage of this is that the adversary does not have to drop a malicious DLL on the disk.	
Similar to the basic dll injection technique, the malware allocates memory in a host process (e.g. VirtualAllocEx), and instead of writing a "DLL path" it writes its malicious code by calling WriteProcessMemory.	

False Positives

Hunter Notes

- Looking for CreateRemoteThread APIs with LoadLibrary functions might return several entries in your environment. I recommend to stack the values of the source and target processes or user to baseline your environmennt.
- Look for processes loading files that have just been created on disk (i.e 1min time window). Stack the values of the processes and files involved. You can tag the files as signed or unsigned depending on the information provided in the security events.

Hunt Output

Type	Link

Sigma Rule	https://github.com/SigmaHQ/sigma/blob/master/rules/windows/create_remote_thread/sysmon_createremotethread_loadli

References

- https://www.endgame.com/blog/technical-blog/ten-process-injection-techniques-technical-survey-common-and-trending-process
- https://resources.infosecinstitute.com/using-createremotethread-for-dll-injection-on-windows/
- https://arvanaghi.com/blog/dll-injection-using-loadlibrary-in-C/
- https://github.com/EmpireProject/Empire/blob/master/data/module_source/code_execution/Invoke-DllInjection.ps1#L249
- https://github.com/EmpireProject/Empire/blob/master/data/module_source/code_execution/Invoke-DllInjection.ps1#L291
- https://github.com/EmpireProject/Empire/blob/master/data/module_source/code_execution/Invoke-DllInjection.ps1#L295
- https://github.com/EmpireProject/Empire/blob/master/data/module_source/code_execution/Invoke-DllInjection.ps1#L303
- https://github.com/EmpireProject/Empire/blob/master/data/module_source/code_execution/Invoke-DllInjection.ps1#L307
- https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibrarya

‹ Previous
LSASS Memory Read Access

Next
Active Directory Object Access via Replication Services ›

By Roberto Rodriguez @Cyb3rWard0g

© Copyright 2022.