Analyzing a “multilayer” Maldoc: A Beginner’s Guide

By Didier Stevens

Published: 2022-04-06 · Archived: 2026-04-06 00:18:43 UTC

In this blog post, we will not only analyze an interesting malicious document, but we will also demonstrate the steps required to get you up and running with the necessary analysis tools. There is also a howto video for this blog post.

I was asked to help with the analysis of a PDF document containing a DOCX file.

The PDF is REMMITANCE INVOICE.pdf, and can be found on VirusTotal, MalwareBazaar and Malshare (you don’t need a subscription to download from MalwareBazaar or Malshare, so everybody that wants to, can follow along).

The sample is interesting for analysis, because it involves 3 different types of malicious documents.
And this blog post will also be different from other maldoc analysis blog posts we have written, because we show how to do the analysis on a machine with a pristine OS and without any preinstalled analysis tools.

To follow along, you just need to be familiar with operating systems and their command-line interface.
We start with a Ubuntu LTS 20.0 virtual machine (make sure that it is up-to-date by issuing the “sudo apt update” and “sudo apt upgrade” commands). We create a folder for the analysis: /home/testuser1/Malware (we usually create a folder per sample, with the current date in the filename, like this: 20220324_twitter_pdf). testuser1 is the account we use, you will have another account name.

Inside that folder, we copy the malicious sample. To clearly mark the sample as (potentially) malicious, we give it the extension .vir. This also prevents accidental launching/execution of the sample. If you want to know more about handling malware samples, take a look at this SANS ISC diary entry.

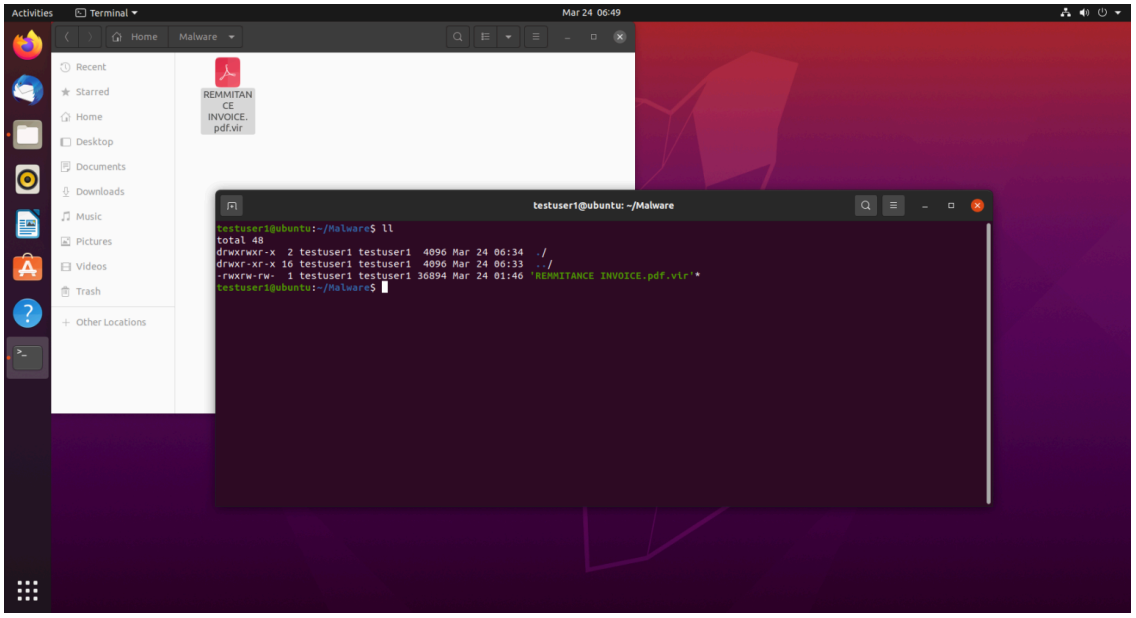


Figure 1: The analysis machine with the PDF sample

The original name of the PDF document is REMMITANCE INVOICE.pdf, and we renamed it to REMMITANCE INVOICE.pdf.vir.
To conduct the analysis, we need tools that I develop and maintain. These are free, open-source tools, designed for static analysis of malware. Most of them are written in Python (a free, open-source programming language).
These tools can be found here and on GitHub.

PDF Analysis

To analyze a malicious PDF document like this one, we are not opening the PDF document with a PDF reader like Adobe Reader. In stead, we are using dedicated tools to dissect the document and find malicious code. This is known as static analysis.
Opening the malicious PDF document with a reader, and observing its behavior, is known as dynamic analysis.

Both are popular analysis techniques, and they are often combined. In this blog post, we are performing static analysis.

To install the tools from GitHub on our machine, we issue the following “git clone” command:

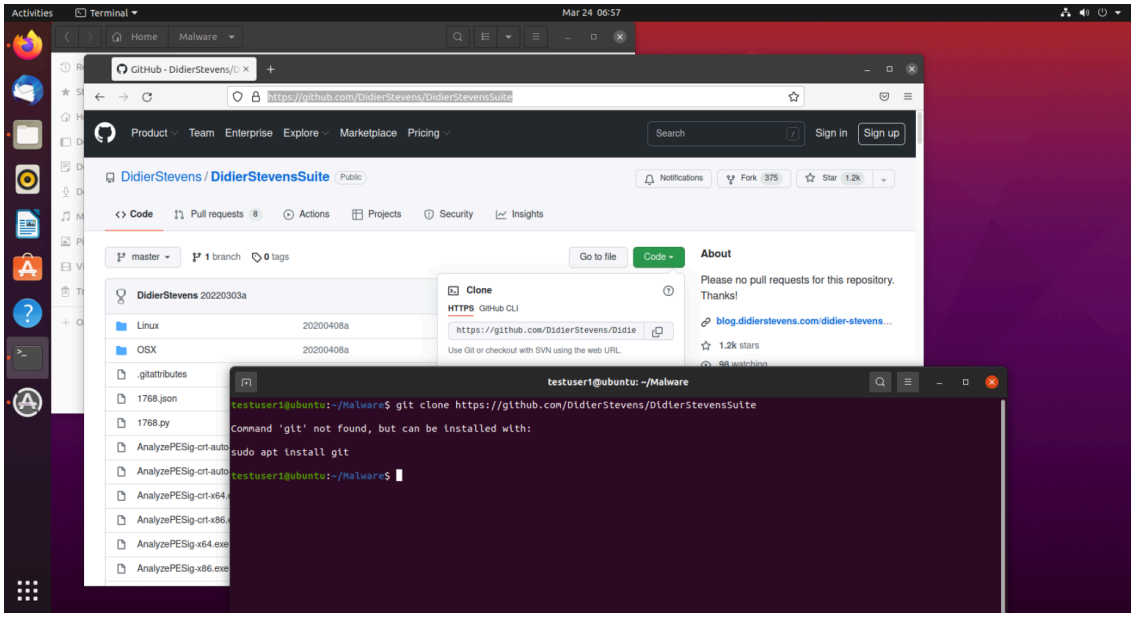


Figure 2: The “git clone” command fails to execute

As can be seen, this command fails, because on our pristine machine, git is not yet installed. Ubuntu is helpful and suggest the command to execute to install git:

sudo apt install git

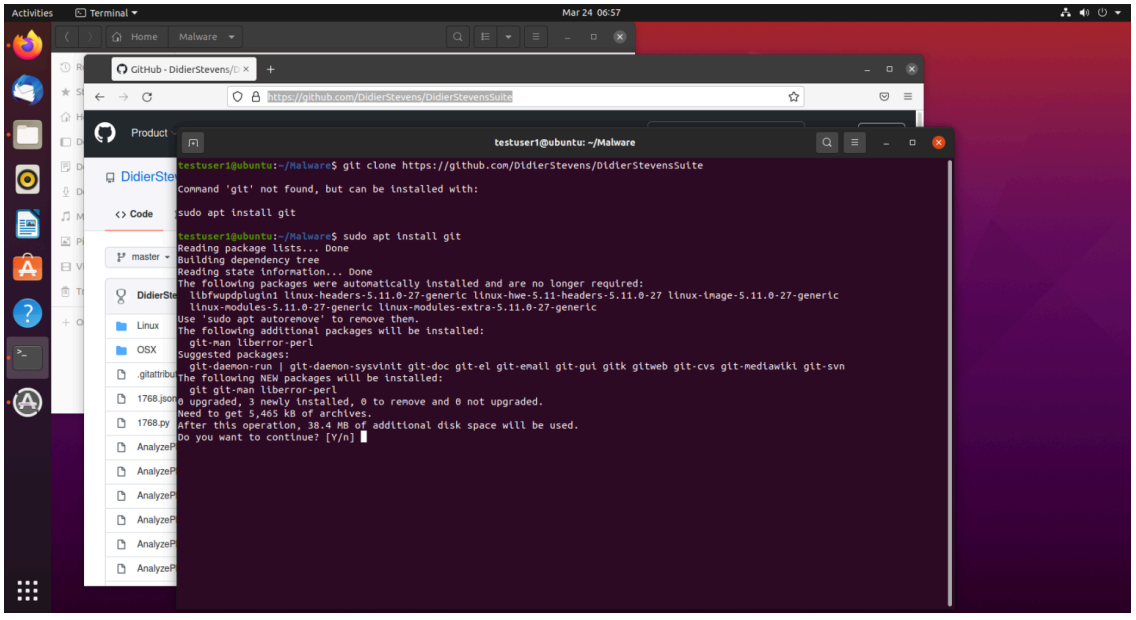


Figure 3: Installing git

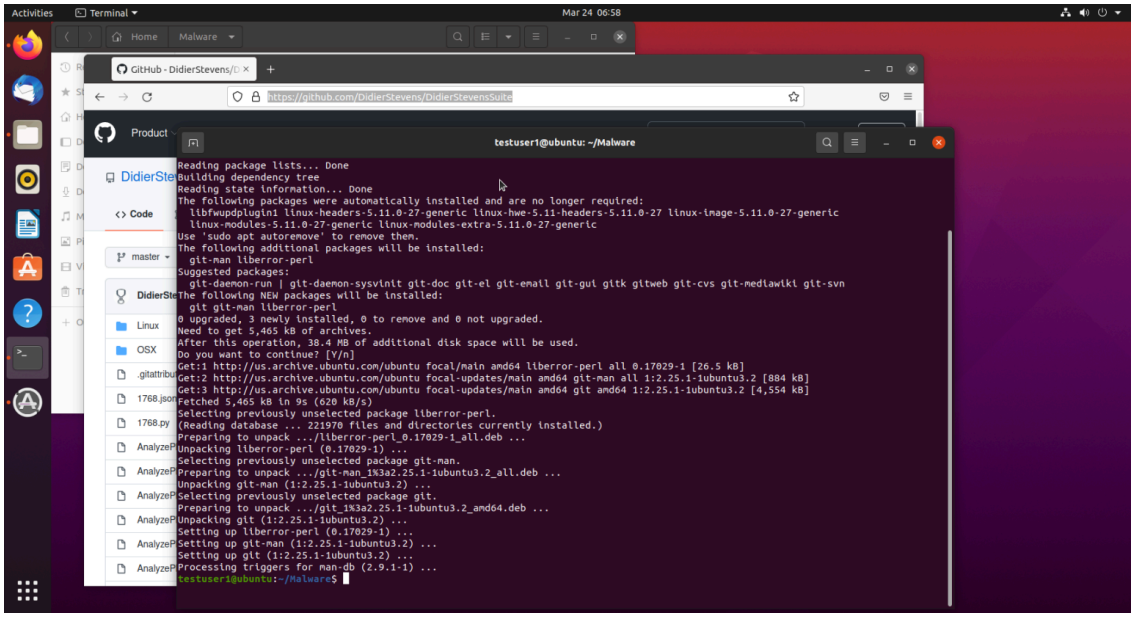


Figure 4: Installing git

When the DidierStevensSuite repository has been cloned, we will find a folder DidierStevensSuite in our working folder:

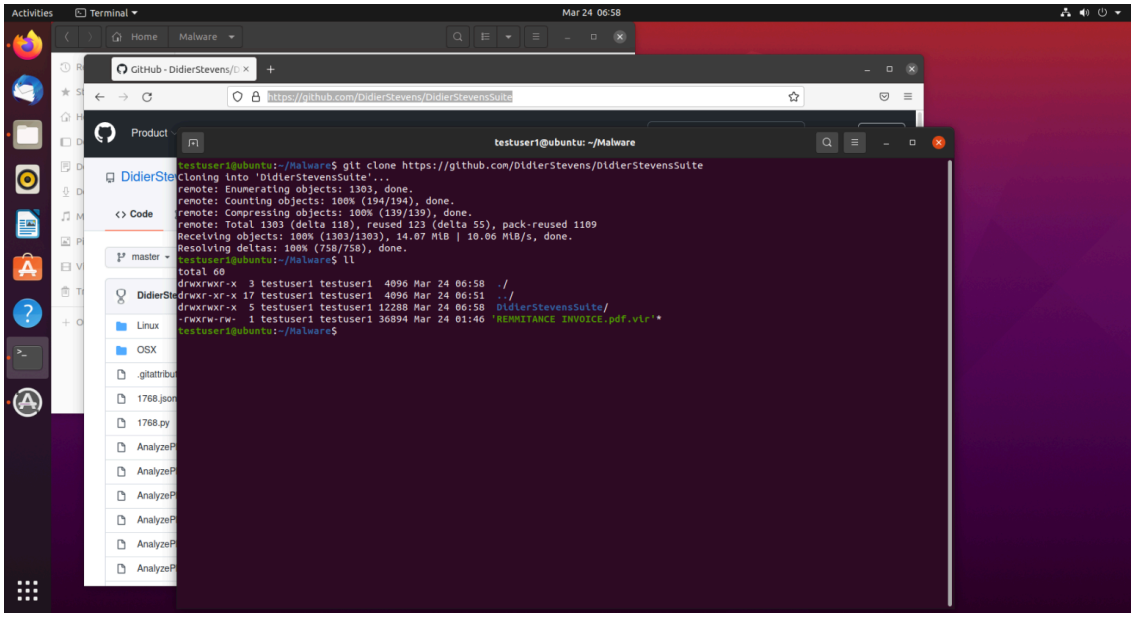


Figure 5: Folder DidierStevensSuite is the result of the clone command

With this repository of tools, we have different maldoc analysis tools at our disposal. Like PDF analysis tools. pdfid.py and pdf-parser.py are two PDF analysis tools found in Didier Stevens' Suite. pdfid is a simple triage tool, that looks for known keywords inside the PDF file, that are regularly associated with malicious activity. pdf-parser.py is able to parse a PDF file and identify basic building blocks of the PDF language, like objects.

To run pdfid.py on our Ubuntu machine, we can start the Python interpreter (python3), and give it the pdfid.py program as first parameter, followed by options and parameters specific for pdfid. The first parameter we provide for pdfid, is the name of the PDF document to analyze. Like this:

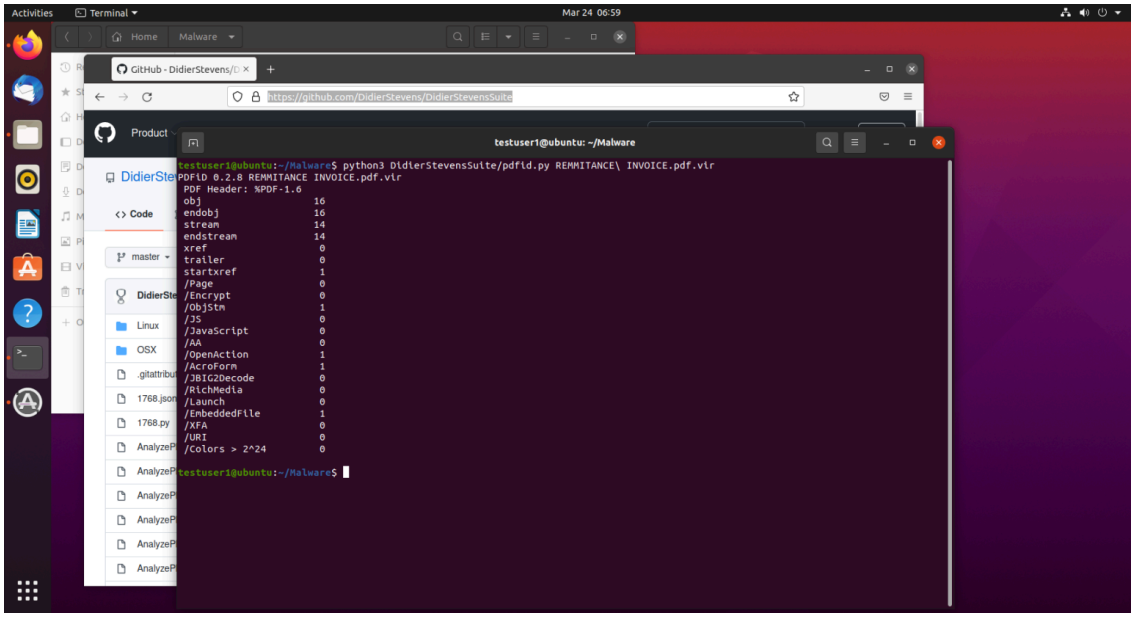


Figure 6: pdfid's analysis report

In the report provided as output by pdfid, we see a bunch of keywords (first column) and a counter (second column). This counter simply indicates the frequency of the keyword: how many times does it appear in the analyzed PDF document?

As you can see, many counters are zero: keywords with zero counter do not appear in the analyzed PDF document. To make the report shorter, we can use option -n. This option excludes zero counters (n = no zeroes) from the report, like this:

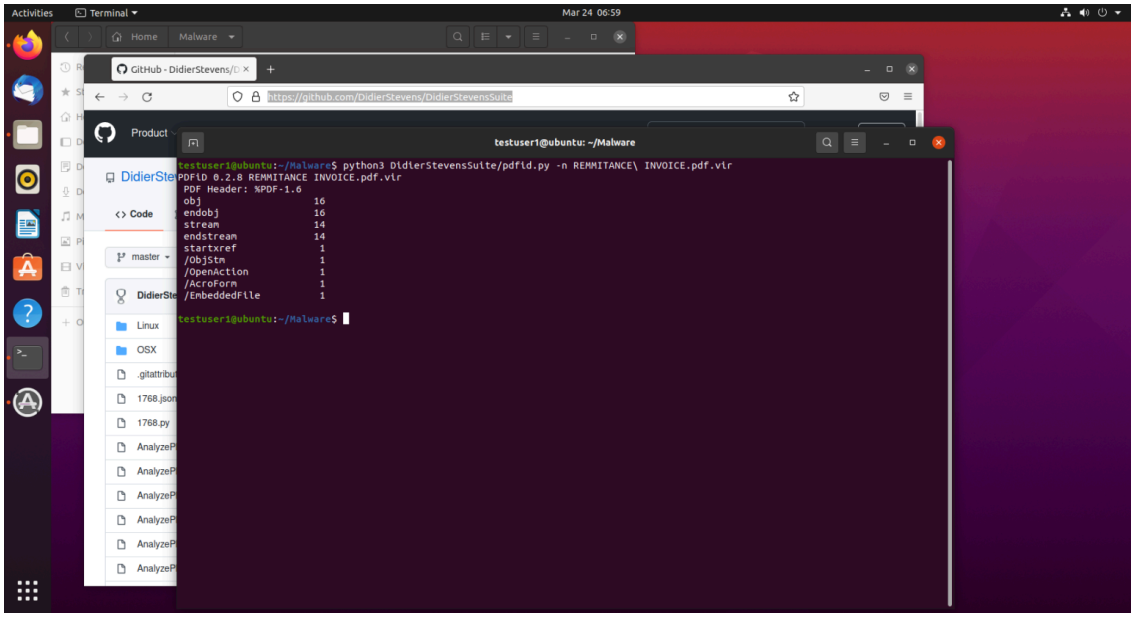


Figure 7: pdfid's condensed analysis report

The keywords that interest us the most, are the ones after the /Page keyword.
Keyword /EmbeddedFile means that the PDF contains an embedded file. This feature can be used for benign and malicious purposes. So we need to look into it.
Keyword /OpenAction means that the PDF reader should do something automatically, when the document is

opened. Like launching a script.
Keyword /ObjStm means that there are stream objects inside the PDF document. Stream objects are special objects, that contain other objects. These contained objects are compressed. pdfid is in nature a simple tool, that is not able to recognize and handle compressed data. This has to be done with pdf-parser.py. Whenever you see stream objects in pdfid's report (e.g., /ObjStm with counter greater than zero), you have to realize that pdfid is unable to give you a complete report, and that you need to use pdf-parser to get the full picture. This is what we do with the following command:

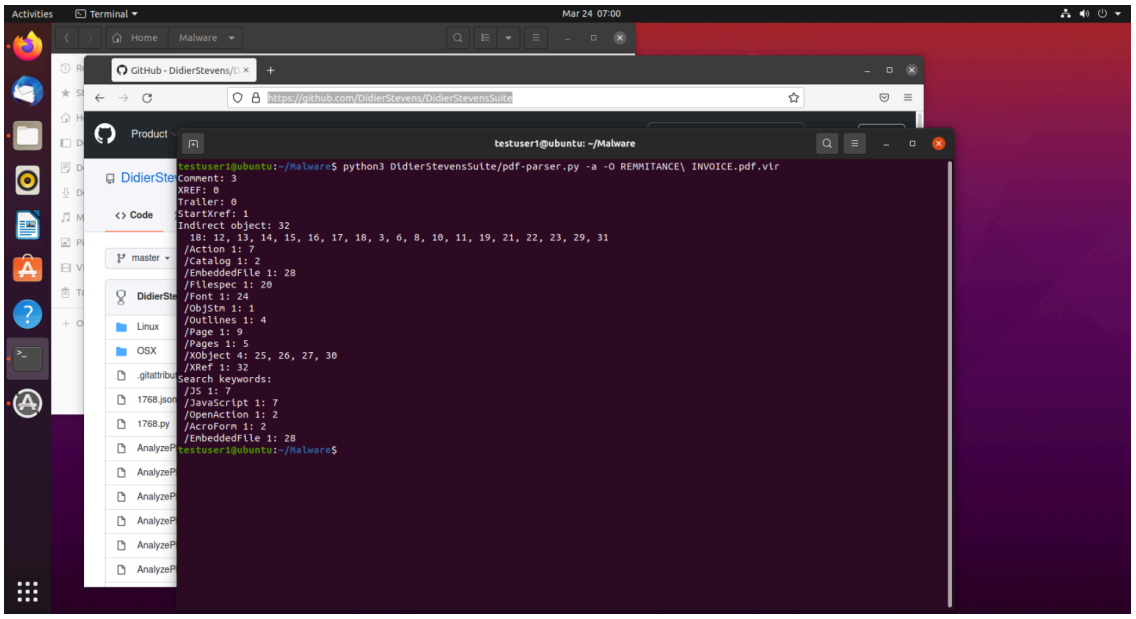


Figure 8: pdf-parser's statistical report

Option -a is used to have pdf-parser.py produce a report of all the different elements found inside the PDf document, together with keywords like pdfid.py produces.
Option -O is used to instruct pdf-parser to decompress stream objects (/ObjStm) and include the contained objects into the statistical report. If this option is omitted, then pdf-parser's report will be similar to pdfid's report. To know more about this subject, we recommend this blog post.

In this report, we see again keywords like /EmbeddedFile. 1 is the counter (e.g., there is one embedded file) and 28 is the index of the PDF object for this embedded file.
New keywords that did appear, are /JS and /JavaScript. They indicate the presence of scripts (code) in the PDF document. The objects that represent these scripts, are found (compressed) inside the stream objects (/ObjStm). That is why they did not appear in pdfid's report, and why they do in pdf-parser's report (when option -O is used).
JavaScript inside a PDF document is restricted in its interactions with the operating system resources: it can not access the file system, the registry, … .
Nevertheless, the included JavaScript can be malicious code (a legitimate reason for the inclusion of JavaScript in a PDF document, is input validation for PDF forms).
But we will first take a look at the embedded file. We to this by searching for the /EmbeddedFile keyword, like this:

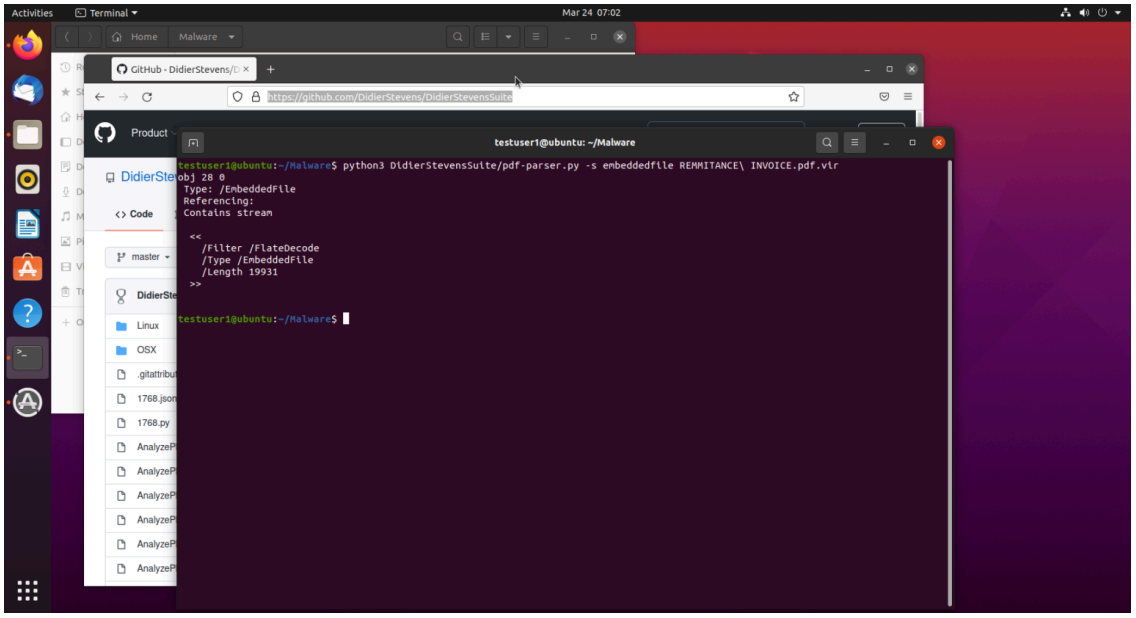


Figure 9: Searching for embedded files

Notice that the search option -s is not case sensitive, and that you do not need to include the leading slash (/). pdf-parser found one object that represents an embedded file: the object with index 28.

Notice the keywords /Filter /Flatedecode: this means that the embedded file is not included into the PDF document as-is, but that it has been “filtered” first (e.g., transformed). /FlateDecode indicates which transformation was applied: “deflation”, e.g., zlib compression.

To obtain the embedded file in its original form, we need to decompress the contained data (stream), by applying the necessary filters. This is done with option -f:

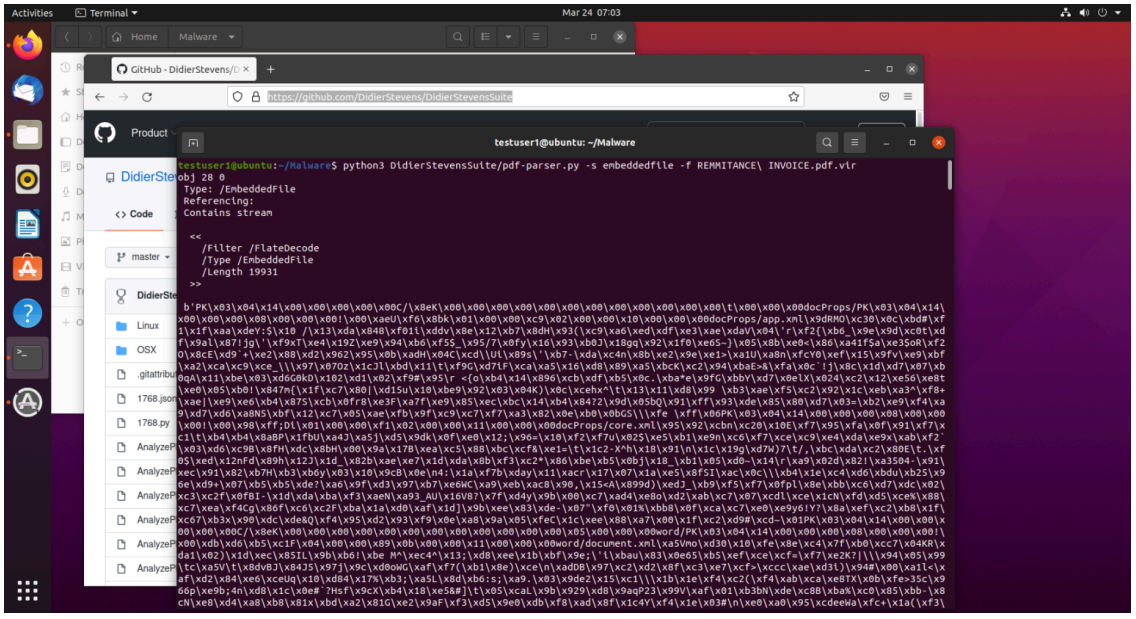


Figure 10: Decompressing the embedded file

The long string of data (it looks random) produced by pdf-parser when option -f is used, is the decompressed stream data in Python’s byte string representation. Notice that this data starts with PK: this is a strong indication that the embedded file is a ZIP container.

We will now use option -d to dump (write) the contained file to disk. Since it is (potentially) malicious, we use again extension .vir.

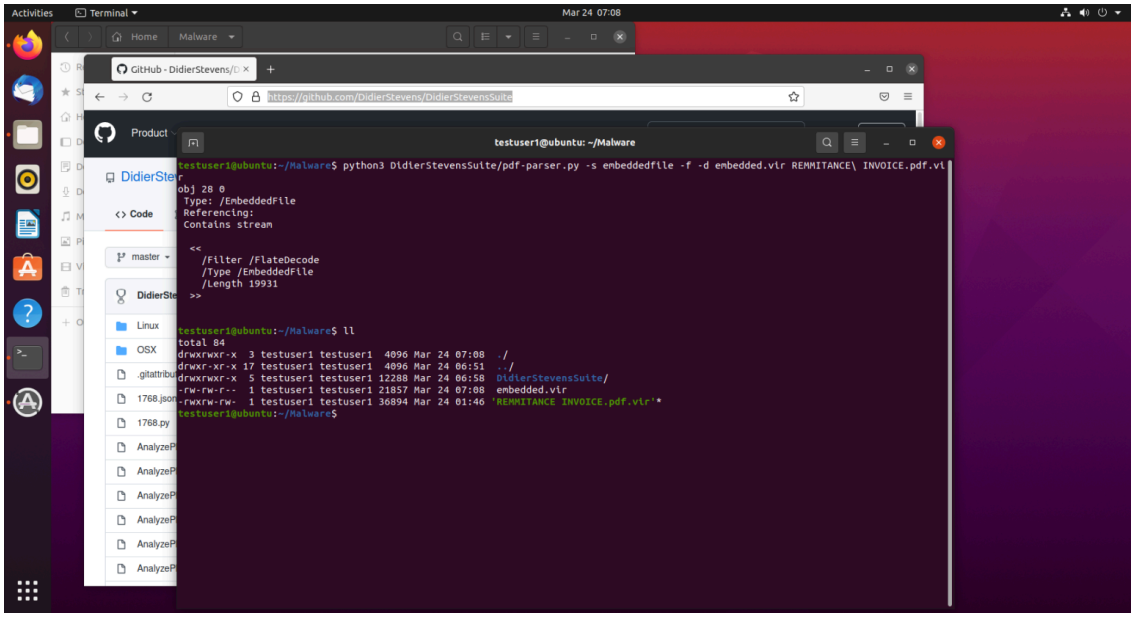


Figure 11: Extracting the embedded file to disk

File embedded.vir is the embedded file.

Office document analysis

Since I was told that the embedded file is an Office document, we use a tool I developed for Office documents: oledump.py
But if you would not know what type the embedded file is, you would first want to determine this. We will actually have to do that later, with a downloaded file.

Now we run oledump.py on the embedded file we extracted: embedded.vir

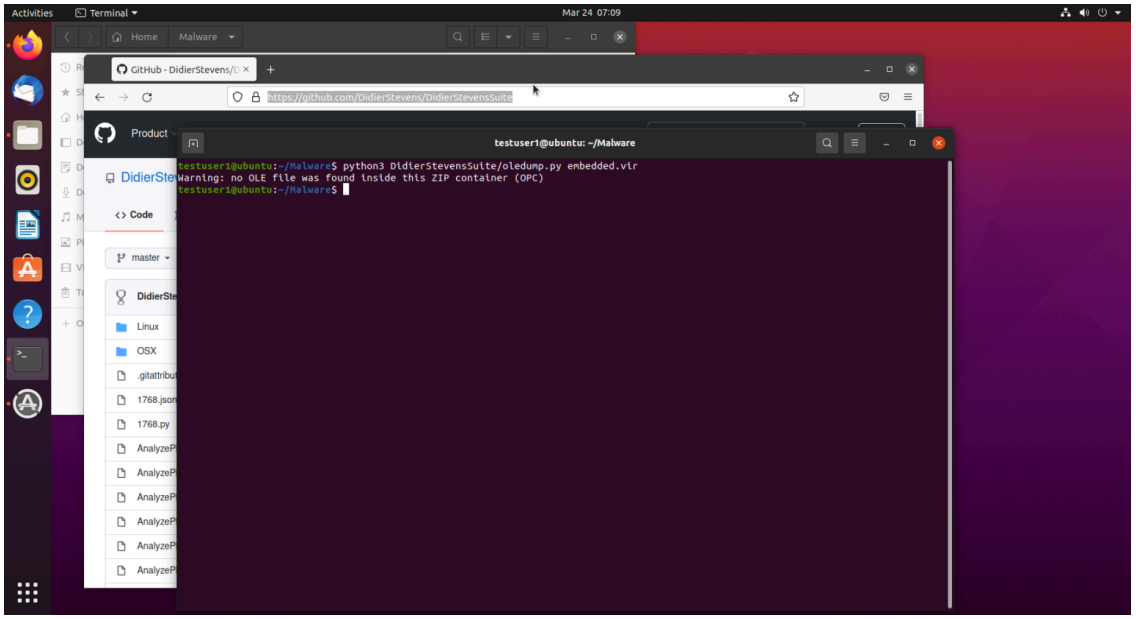


Figure 12: No ole file was found

The output of oledump here is a warning: no ole file was found.
A bit of background can help understand what is happening here. Microsoft Office document files come in 2 major formats: ole files and OOXML files.
Ole files (official name: Compound File Binary Format) are the “old” file format: the binary format that was default until Office 2007 was released. Documents using this internal format have extensions like .doc, .xls, .ppt, …
OOXML files (Office Open XML) are the “new” file format. It’s the default since Office 2007. Its internal format is a ZIP container containing mostly XML files. Other contained file types that can appear are pictures (.png, .jpeg, …) and ole (for VBA macros for example). OOXML files have extensions like .docx, .xlsx, .docm, .xlsm, …
OOXML is based on another format: OPC.
oledump.py is a tool to analyze ole files. Most malicious Office documents nowadays use VBA macros. VBA macros are always stored inside ole files, even with the “new” format OOXML. OOXML documents that contain macros (like .docm), have one ole file inside the ZIP container (often named vbaProject.bin) that contains the actual VBA macros.
Now, let’s get back to the analysis of our embedded file: oledump tells us that it found no ole file inside the ZIP container (OPC).
This tells us 1) that the file is a ZIP container, and more precisely, an OPC file (thus most likely an OOXML file) and 2) that it does not contain VBA macros.
If the Office document contains no VBA macros, we need to look at the files that are present inside the ZIP container. This can be done with a dedicated tool for the analysis of ZIP files: zipdump.py
We just need to pass the embedded file as parameter to zipdump, like this:

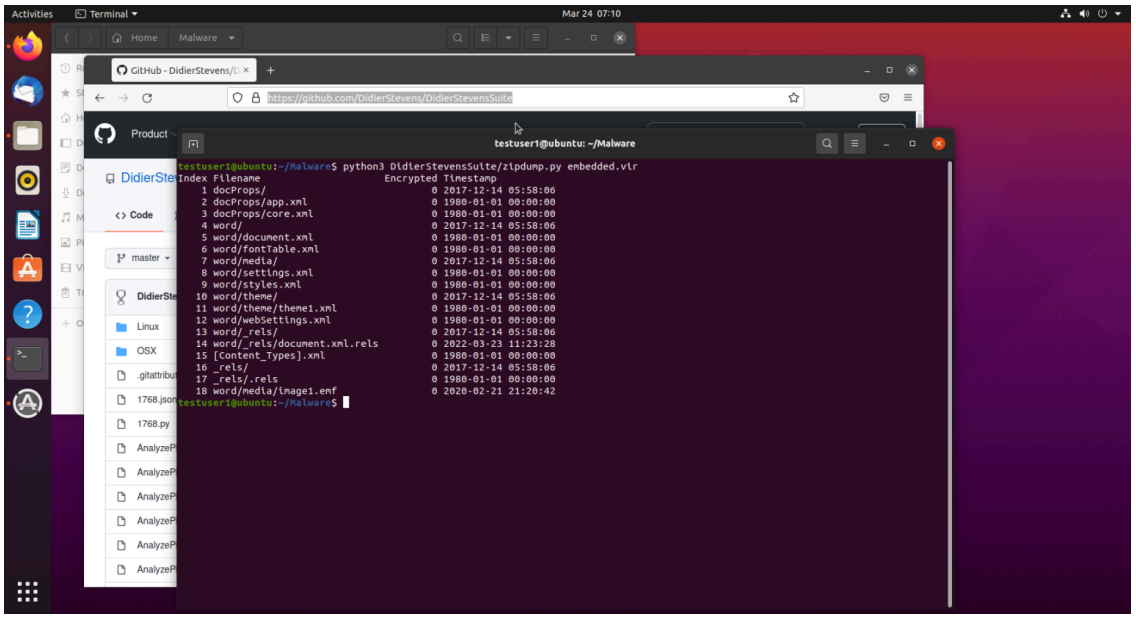


Figure 13: Looking inside the ZIP container

Every line of output produced by zipdump, represents a contained file.
The presence of folder “word” tells us that this is a Word file, thus extension .docx (because it does not contain VBA macros).
When an OOXML file is created/modified with Microsoft Office, the timestamp of the contained files will always

be 1980-01-01.

In the result we see here, there are many files that have a different timestamp: this tells us, that this .docx file has been altered with a ZIP tool (like WinZip, 7zip, …) after it was saved with Office.

This is often an indicator of malicious intend.

If we are presented with an Office document that has been altered, it is recommended to take a look at the contained files that were most recently changed, as this is likely the file that has been tampered for malicious purposed.

In our extracted sample, that contained file is the file with timestamp 2022-03-23 (that's just a day ago, time of writing): file document.xml.rels.

We can use zipdump.py to take a closer look at this file. We do not need to type its full name to select it, we can just use its index: 14 (this index is produced by zipdump, it is not metadata).

Using option -s, we can select a particular file for analysis, and with option -a, we can produce a hexadecimal/ascii dump of the file content. We start with this type of dump, so that we can first inspect the data and assure us that the file is indeed XML (it should be pure XML, but since it has been altered, we must be careful).

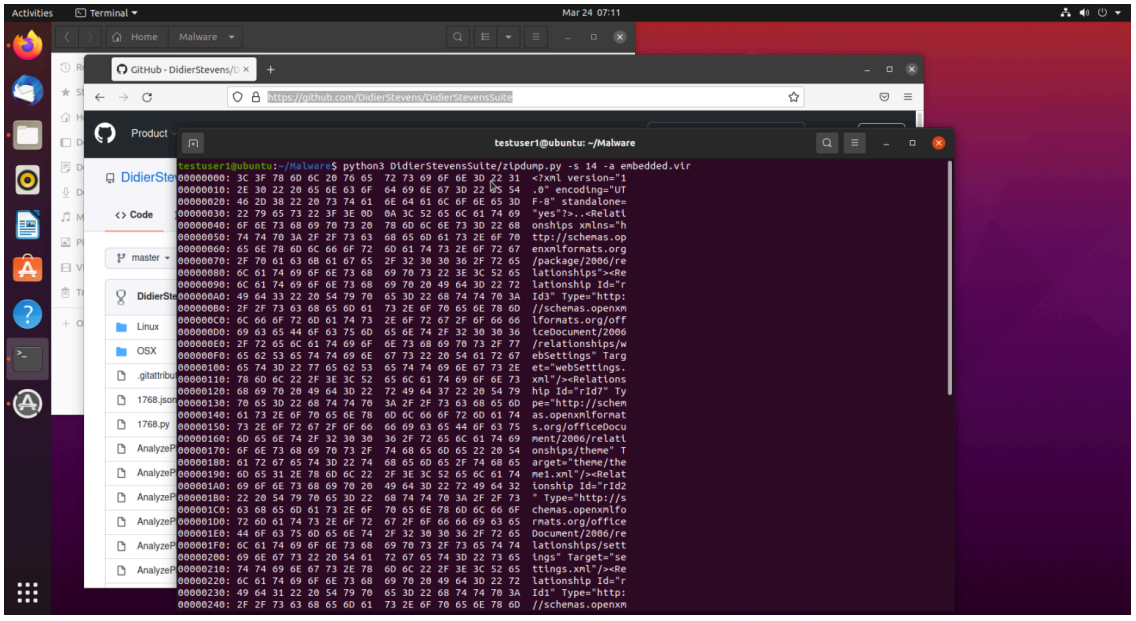


Figure 14: Hexadecimal/ascii dump of file document.xml.rels

This does indeed look like XML: thus we can use option -d to dump the file to the console (stdout):

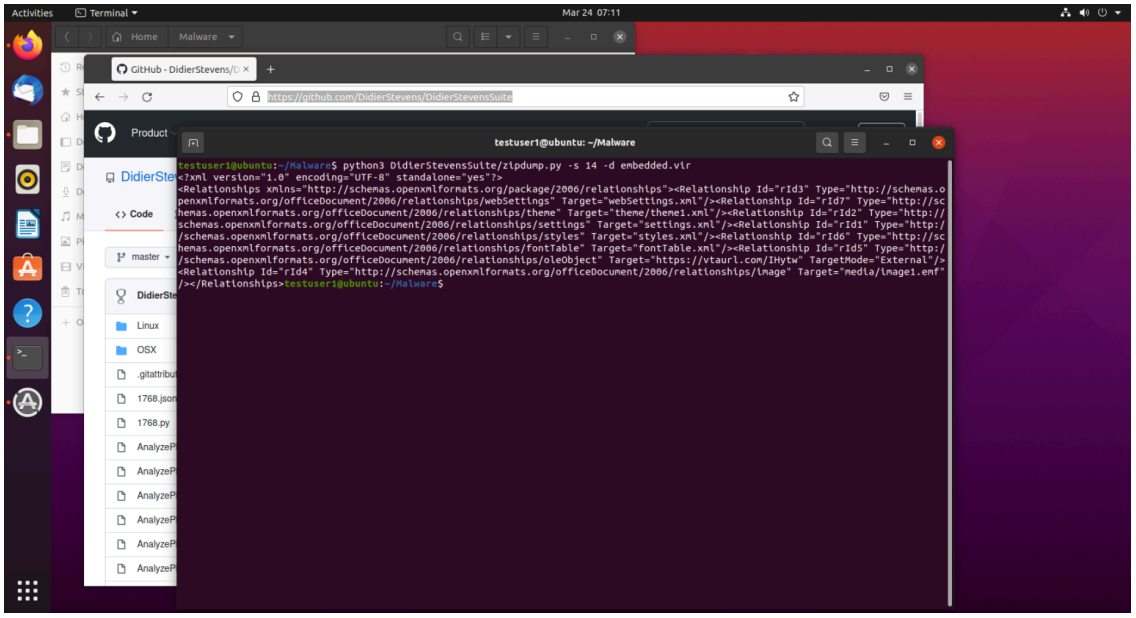


Figure 15: Using option -d to dump the file content

There are many URLs in this output, and XML is readable to us humans, so we can search for suspicious URLs. But since this is XML without any newlines, it's not easy to read. We might easily miss one URL.

Therefor, we will use a tool to help us extract the URLs: re-search.py

re-search.py is a tool that uses regular expressions to search through text files. And it comes with a small embedded library of regular expressions, for URLs, email addresses, …

If we want to use the embedded regular expression for URLs, we use option -n url.

Like this:

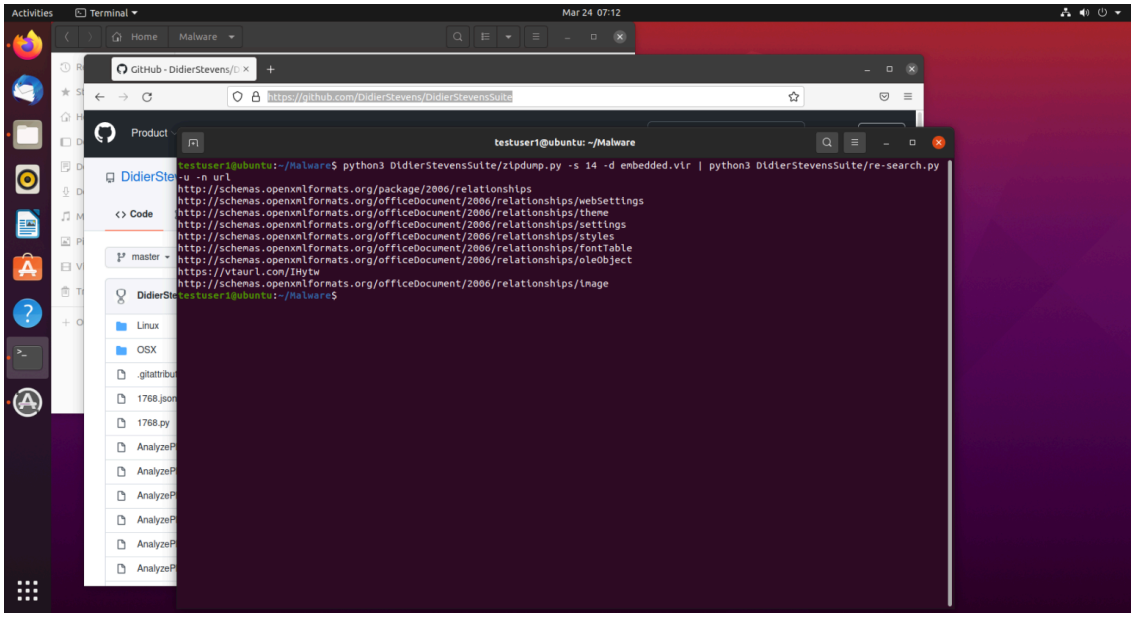


Figure 16: Extracting URLs

Notice that we use option -u to produce a list of unique URLs (remove duplicates from the output) and that we are piping 2 commands together. The output of command zipdump is provided as input to command re-search by using a pipe (|).

Many tools in Didier Stevens' Suite accept input from stdin and produce output to stdout: this allows them to be

piped together.

Most URLs in the output of re-search have schemas.openxmlformats.org as FQDN: these are normal URLs, to be expected in OOXML files. To help filtering out URLs that are expected to be found in OOXML files, re-search has an option to filter out these URLs. This is option -F with value officeurls.

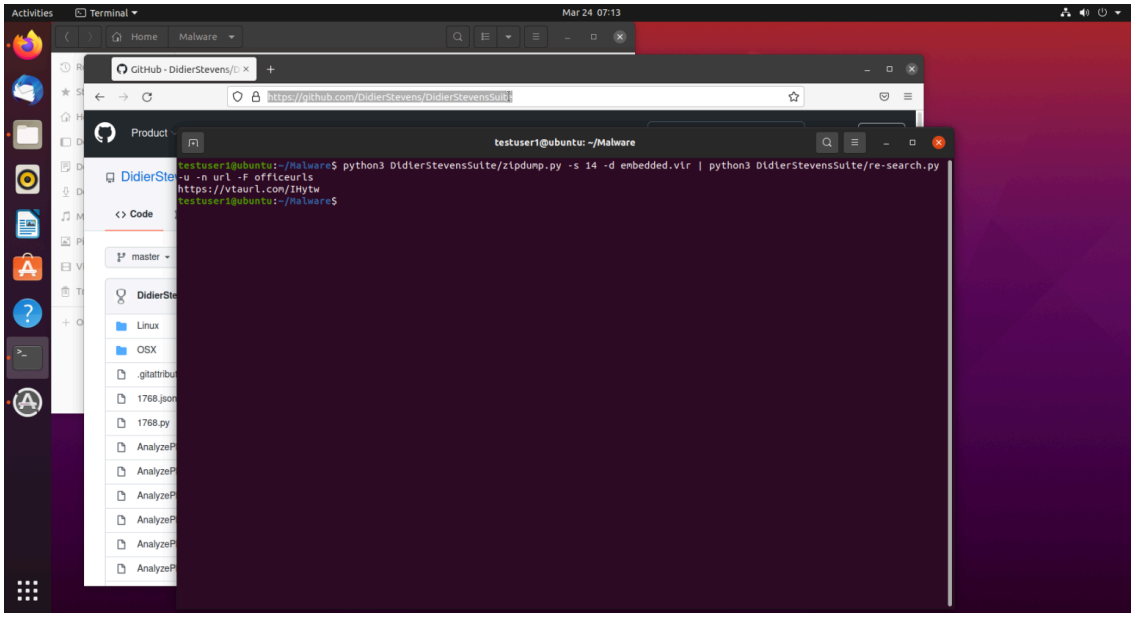


Figure 17: Filtered URLs

One URL remains: this is suspicious, and we should try to download the file for that URL.

Before we do that, we want to introduce another tool that can be helpful with the analysis of XML files: xmldump.py. xmldump parses XML files with Python's built-in XML parser, and can represent the parsed output in different formats. One format is "pretty printing": this makes the XML file more readable, by adding newlines and indentations. Pretty printing is achieved by passing parameter pretty to tool xmldump.py, like this:

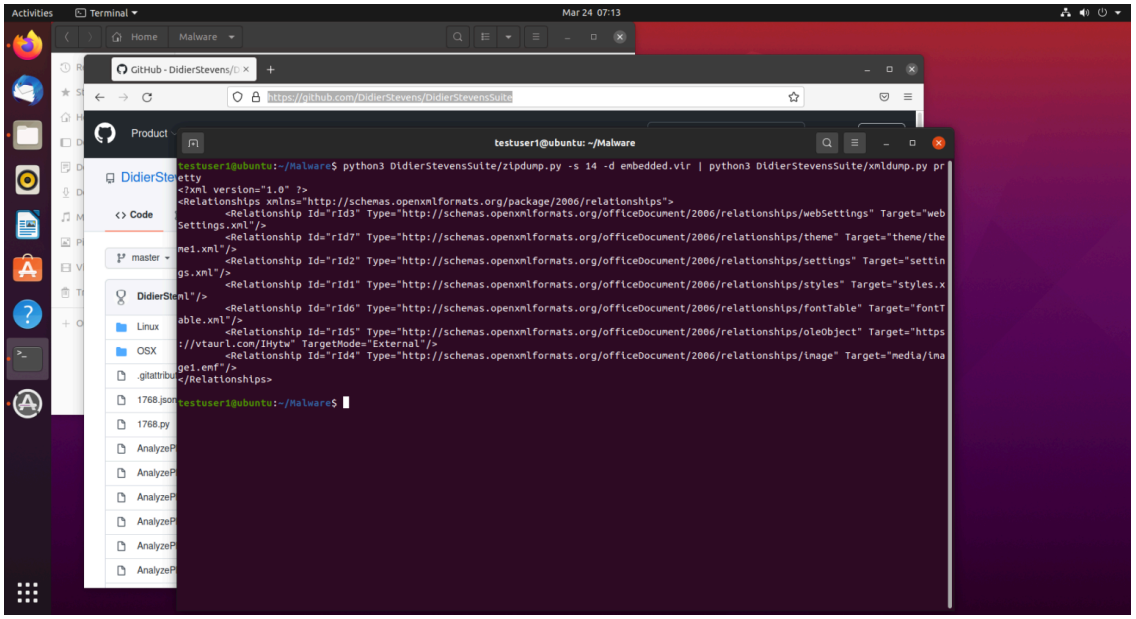


Figure 18: Pretty print of file document.xml.rels

Notice that the <Relationship> element with the suspicious URL, is the only one with attribute TargetMode=”External”.
This is an indication that this is an external template, that is loaded from the suspicious URL when the Office document is opened.
It is therefore important to retrieve this file.

Downloading a malicious file

We will download the file with curl. Curl is a very flexible tool to perform all kinds of web requests.
By default, curl is not installed in Ubuntu:

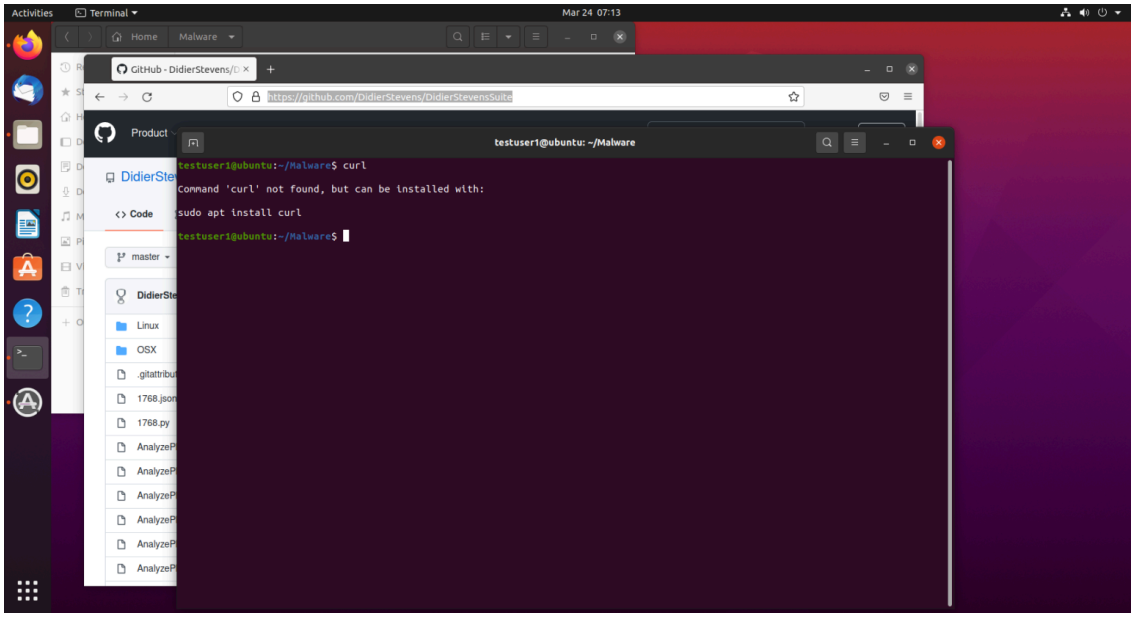


Figure 19: Curl is missing

But it can of course be installed:

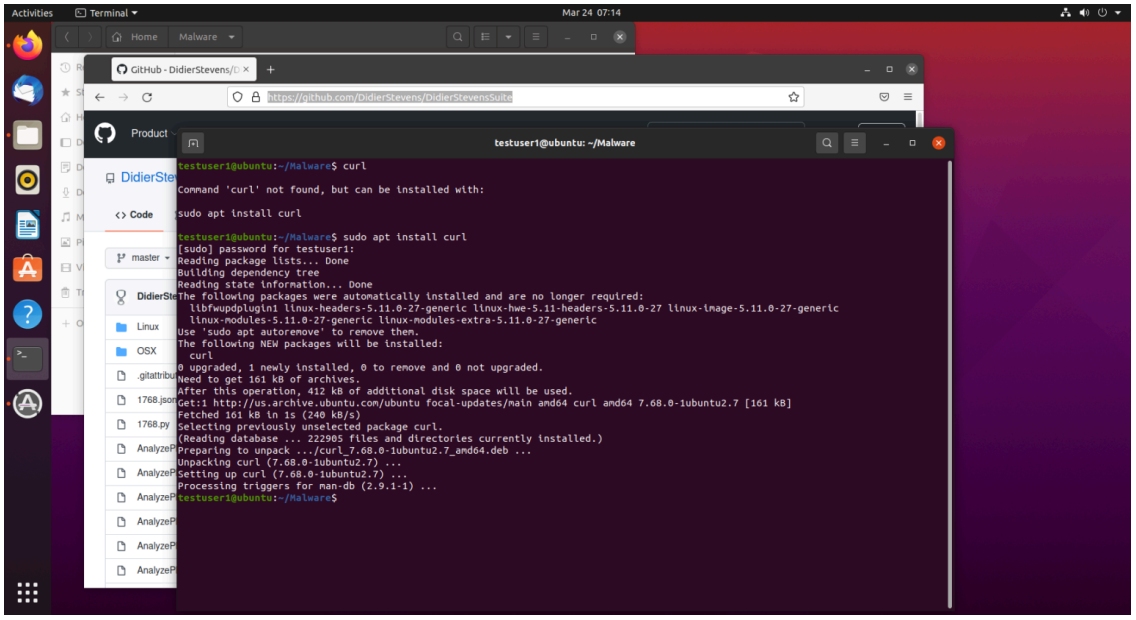


Figure 20: Installing curl

And then we can use it to try to download the template. Often, we do not want to download that file using an IP address that can be linked to us or our organisation. We often use the Tor network to hide behind. We use option -x 127.0.0.1:9050 to direct curl to use a proxy, namely the Tor service running on our machine. And then we like to use option -D to save the headers to disk, and option -o to save the downloaded file to disk with a name of our choosing and extension .vir.
Notice that we also number the header and download files, as we know from experience, that often several attempts will be necessary to download the file, and that we want to keep the data of all attempts.

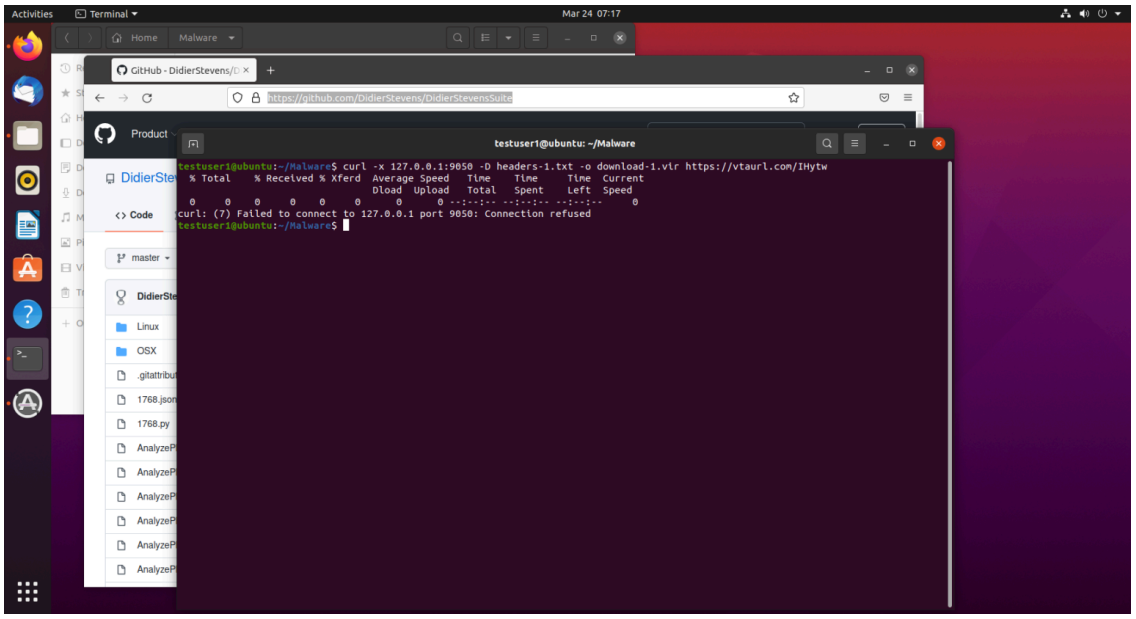


Figure 21: Downloading with curl over Tor fails

This fails: the connection is refused. That's because port 9050 is not open: the Tor service is not installed. We need to install it first:

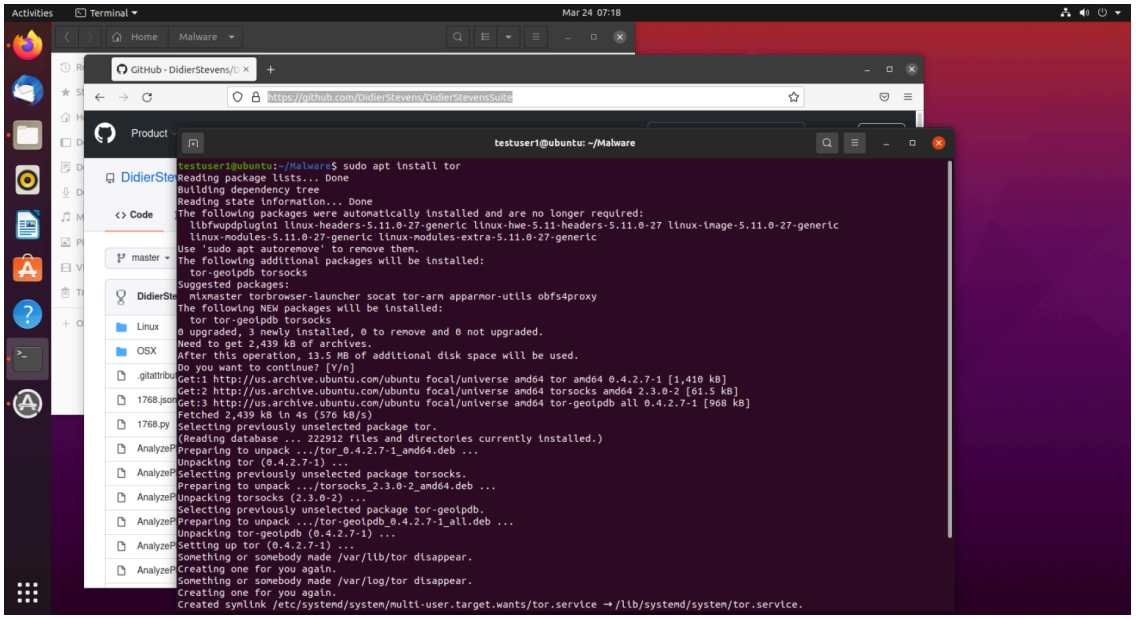


Figure 22: Installing Tor

Next, we try again to download over Tor:

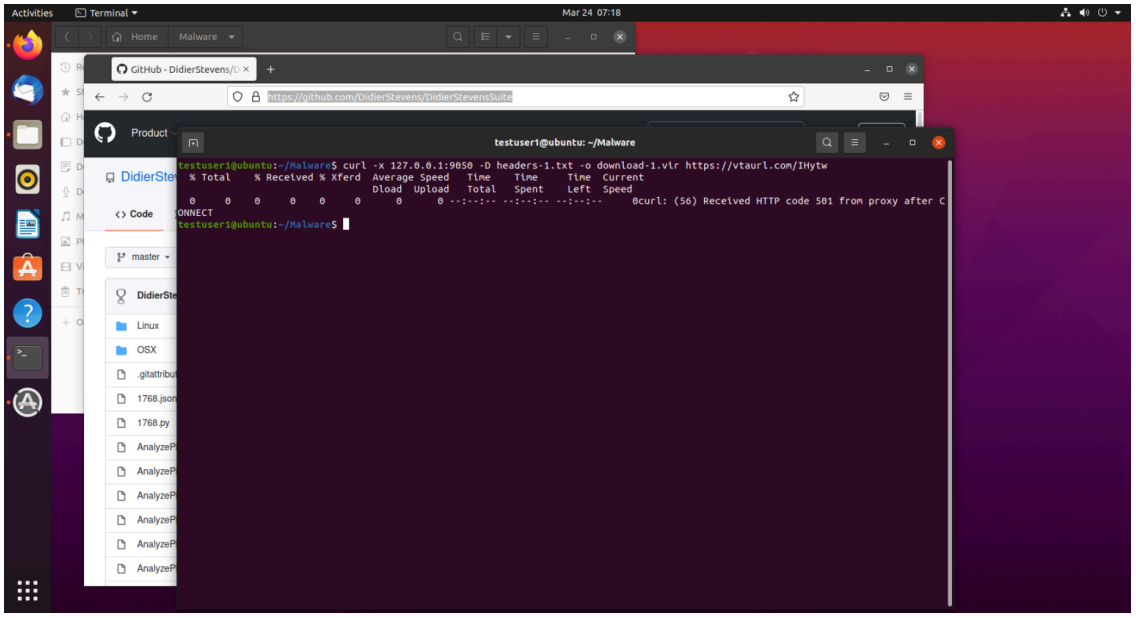


Figure 23: The download still fails

The download still fails, but with another error. The CONNECT keyword tells us that curl is trying to use an HTTP proxy, and Tor uses a SOCKS5 proxy. I used the wrong option: in stead of option -x, I should be using option –socks5 (-x is for HTTP proxies).

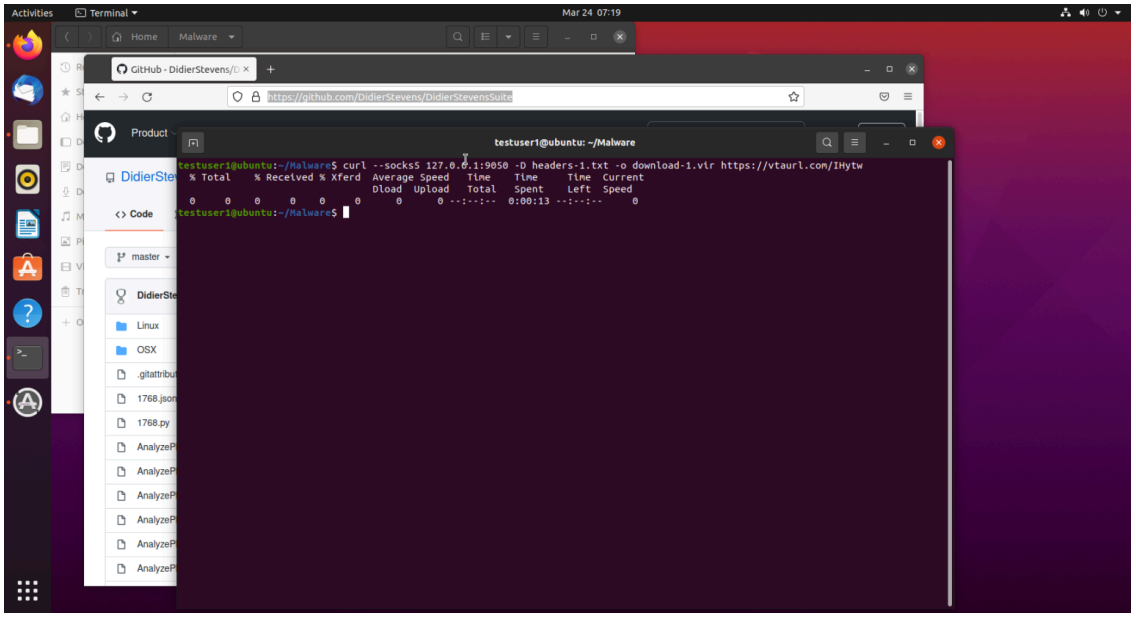


Figure 24: The download seems to succeed

But taking a closer look at the downloaded file, we see that it is empty:

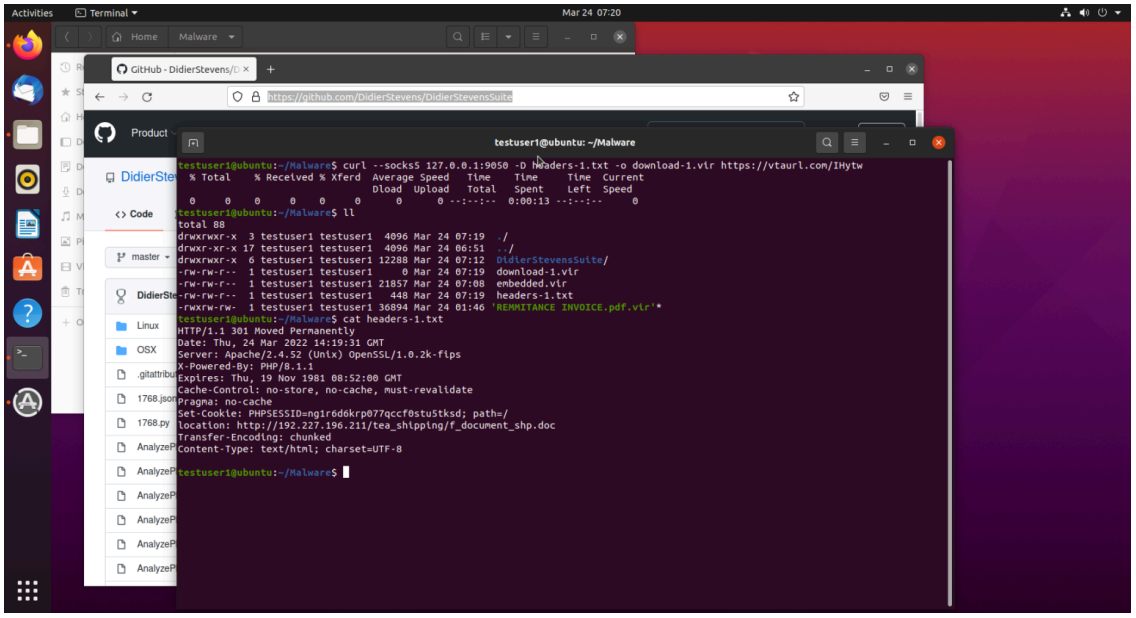


Figure 25: The downloaded file is empty, and the headers indicate status 301

The content of the headers file indicates status 301: the file was permanently moved.
Curl will not automatically follow redirections. This has to be enabled with option -L, let's try again:

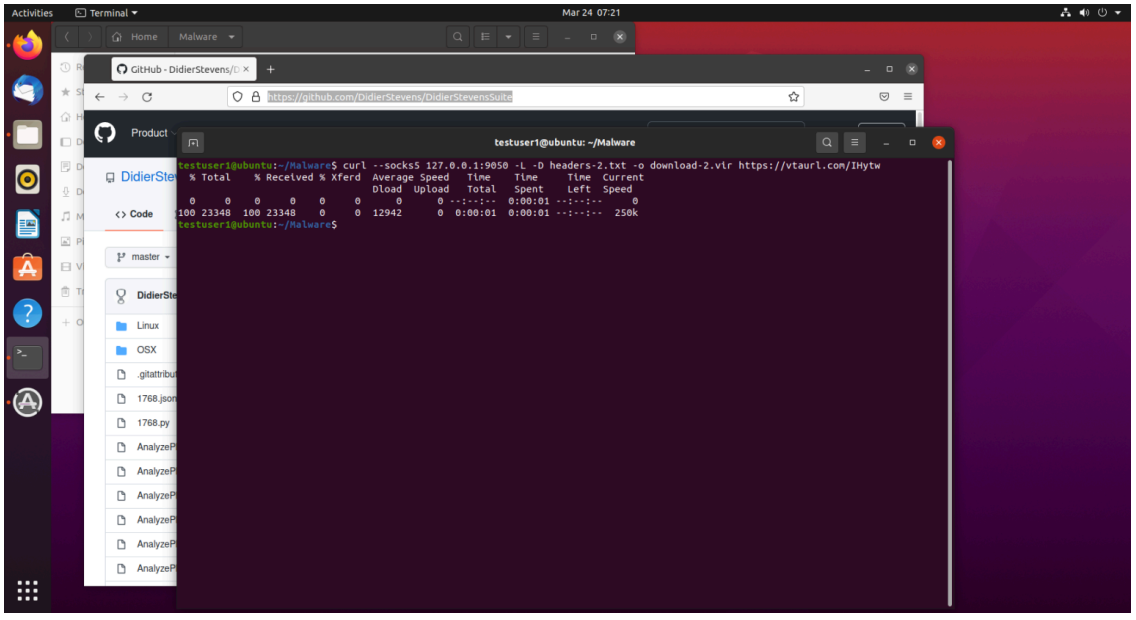


Figure 26: Using option -L

And now we have indeed downloaded a file:

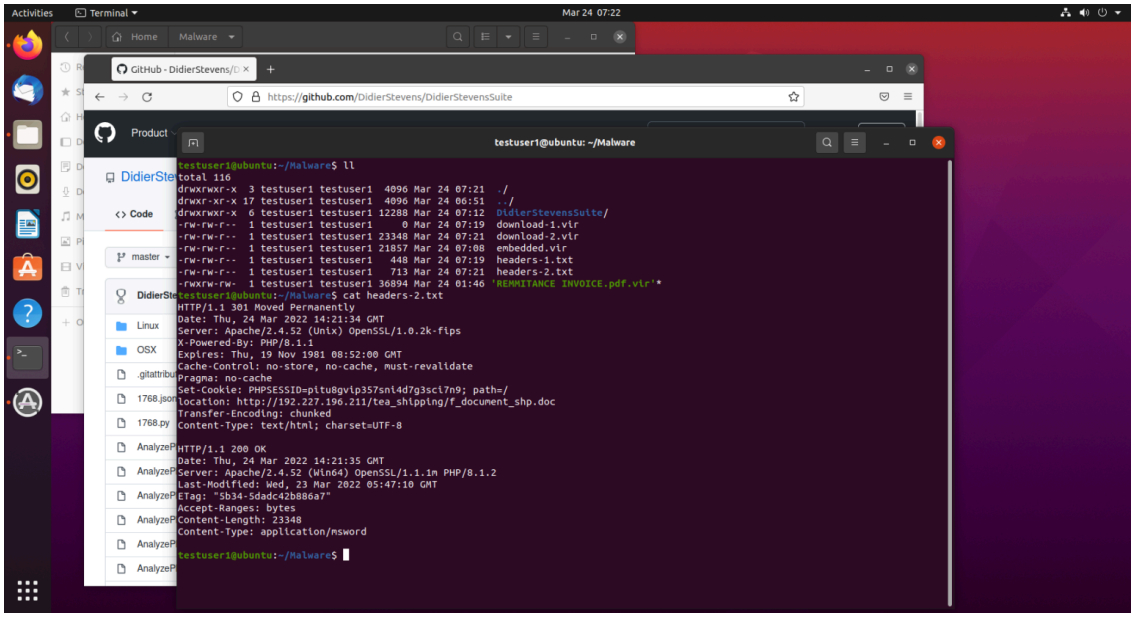


Figure 27: Download result

Notice that we are using index 2 for the downloaded files, as to not overwrite the first downloaded files.
Downloading over Tor will not always work: some servers will refuse to serve the file to Tor clients.
And downloading with Curl can also fail, because of the User Agent String. The User Agent String is a header that Curl includes whenever it performs a request: this header indicates that the request was done by curl. Some servers are configured to only serve files to clients with the “proper” User Agent String, like the ones used by Office or common web browsers.
If you suspect that this is the case, you can use option -A to provide an appropriate User Agent String.

As the downloaded file is a template, we expect it is an Office document, and we use oledump.py to analyze it:

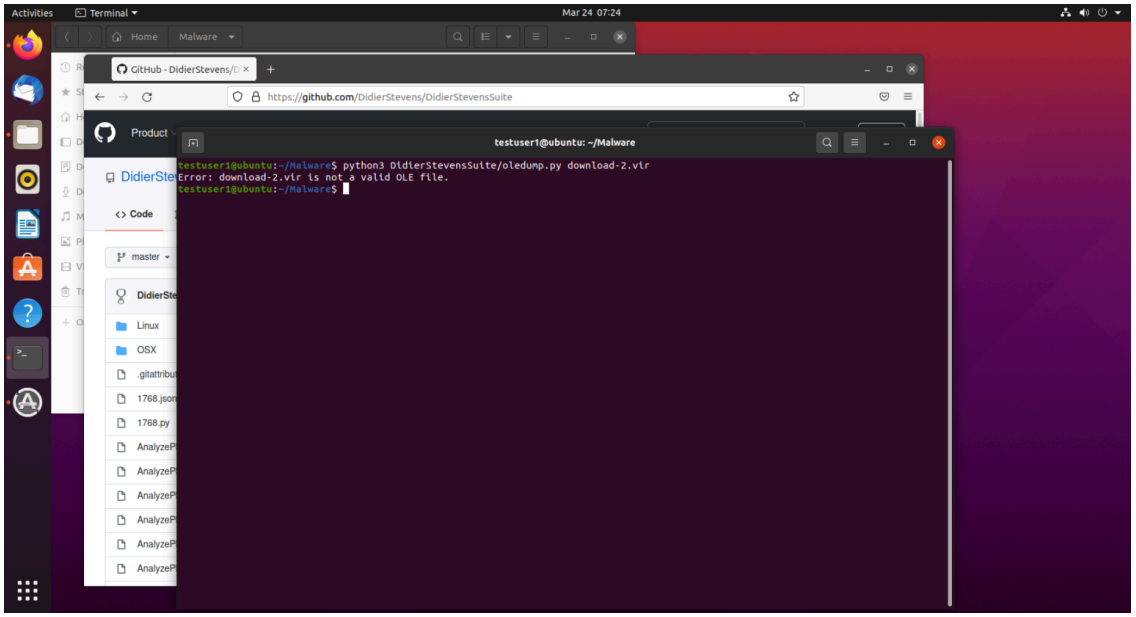
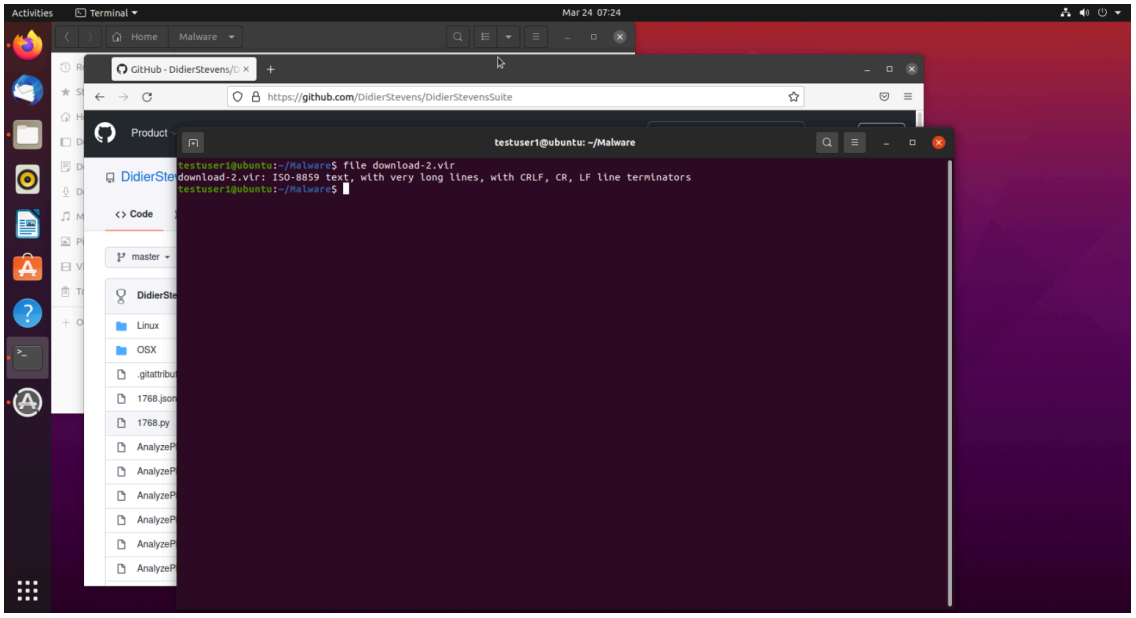


Figure 28: Analyzing the downloaded file with oledump fails

But this fails. Oledump does not recognize the file type: the file is not an ole file or an OOXML file.
We can use Linux command file to try to identify the file type based on its content:



Fgiure 29: Command file tells us this is pure text

If we are to believe this output, the file is a pure text file.
Let's do a hexadecimal/ascii dump with command xxd. Since this will produce many pages of output, we pipe the output to the head command, to limit the output to the first 10 lines:

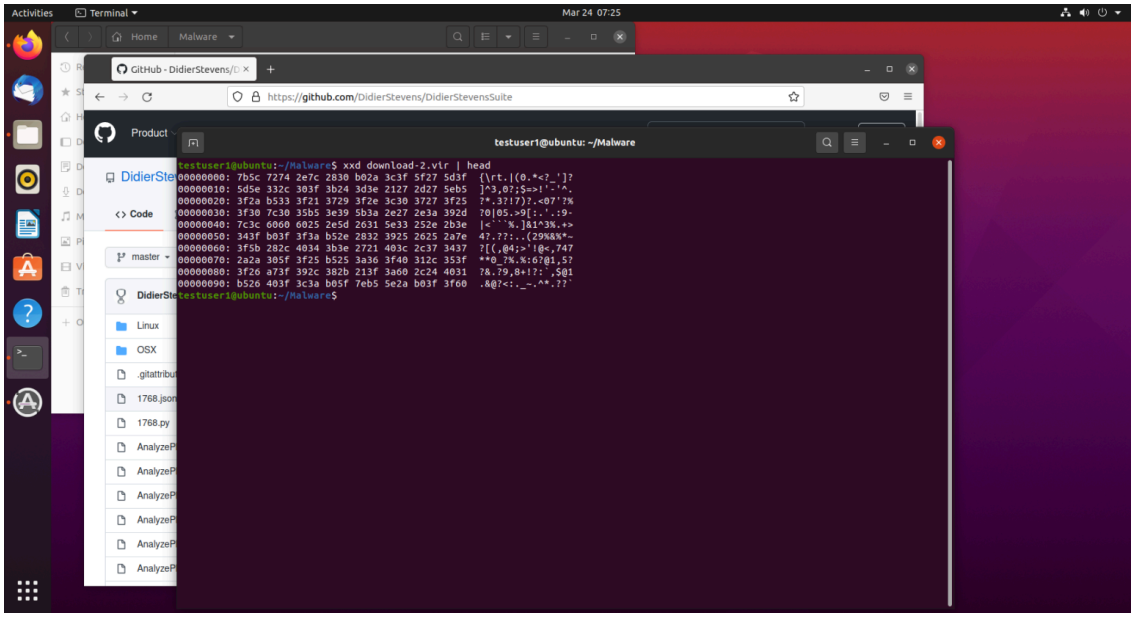


Figure 30: Hexadecimal/ascii dump of the downloaded file

RTF document analysis

The file starts with {\rt : this is a deliberately malformed RTF file. Richt Text Format is a file format for Word documents, that is pure text. The format does not support VBA macros. Most of the time, malicious RTF files perform malicious actions through exploits.
Proper RTF files should start with {\rtf1. The fact that this file starts with {\rt. is a clear indication that the file has been tampered with (or generated with a maldoc generator): Word will not produce files like this. However, Word's RTF parser is forgiving enough to accept files like this.

Didier Stevens' Suite contains a tool to analyze RTF files: rtfdump.py
By default, running rtfdump.py on an RTF file produces a lot of output:

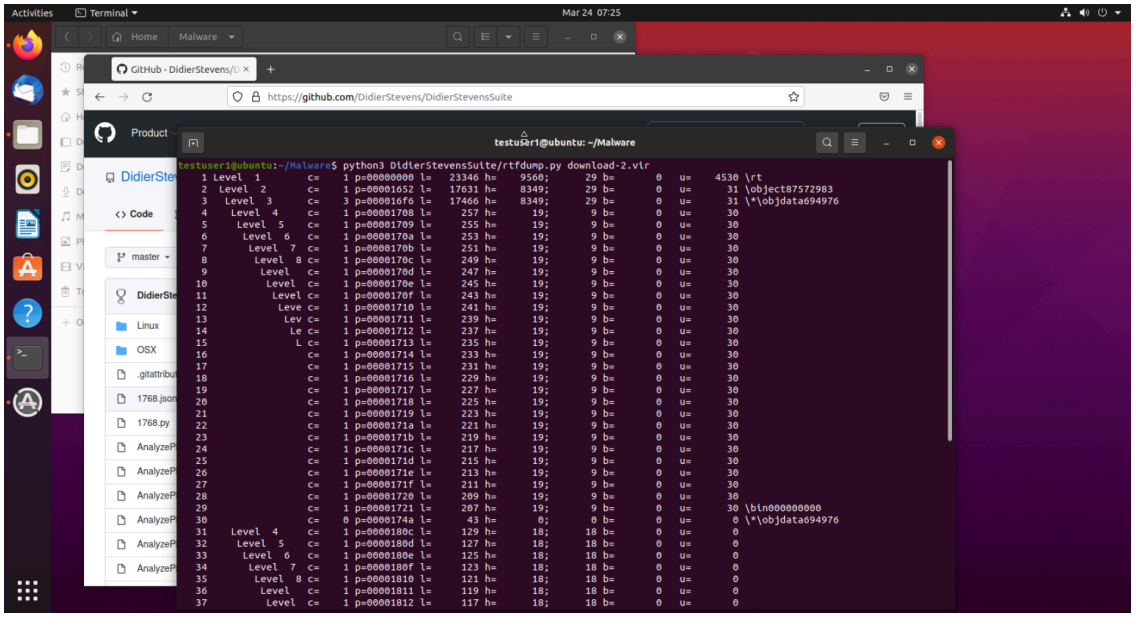


Figure 31: Parsing the RTF file

The most important fact we know from this output, is that this is indeed an RTF file, since rtfdmp was able to parse it.
As RTF files often contain exploits, they often use embedded objects. Filtering rtfdump's output for embedded objects can be done with option -O:

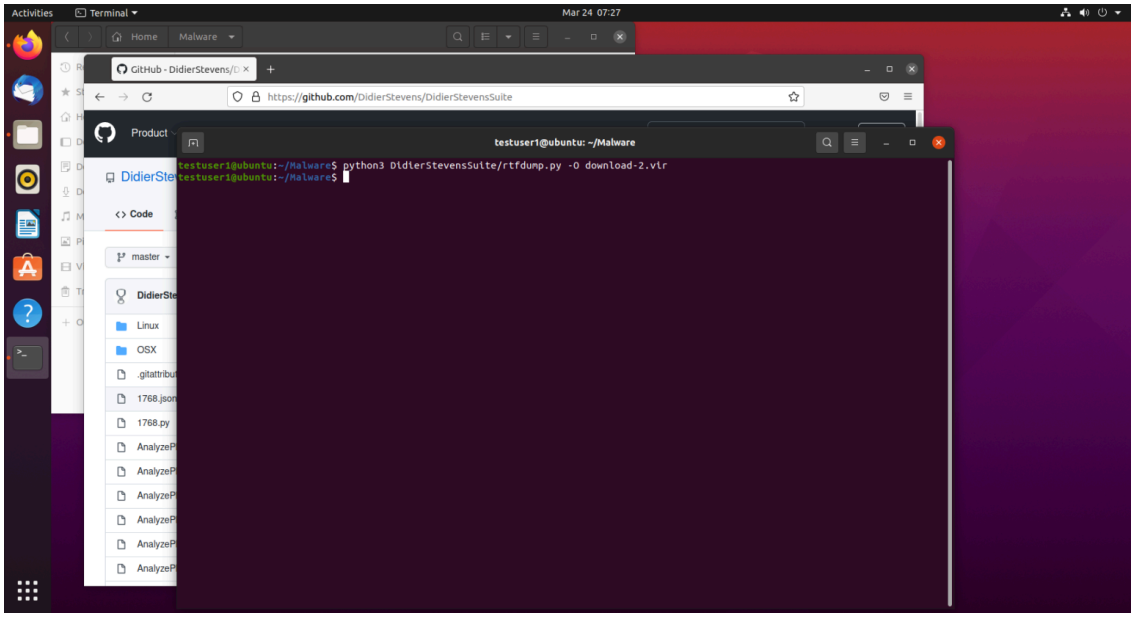


Figure 32: There are no embedded objects

No embedded objects were found. Then we need to look at the hexadecimal data: since RTF is a text format, binary data is encoded with hexadecimal digits. Looking back at figure 30, we see that the second entry (number 2) contains 8349 hexadecimal digits (h=8349). That's the first entry we will inspect further.
Notice that 8349 is an uneven number, and that encoding a single byte requires 2 hexadecimal digits. This is an

indication that the RTF file is obfuscated, to thwart analysis.
Using option -s, we can select entry 2:

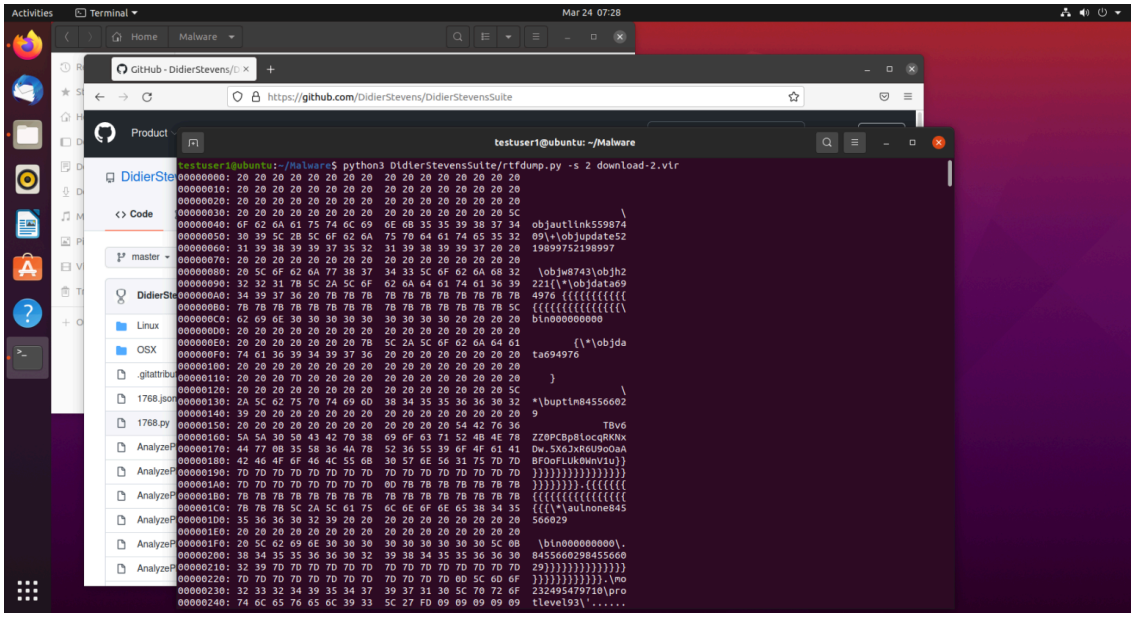


Figure 33: Selecting the second entry

If you are familiar with the internals of RTF files, you would notice that the long, uninterrupted sequences of curly braces are suspicious: it’s another sign of obfuscation.
Let’s try to decode the hexadecimal data inside entry 2, by using option -H

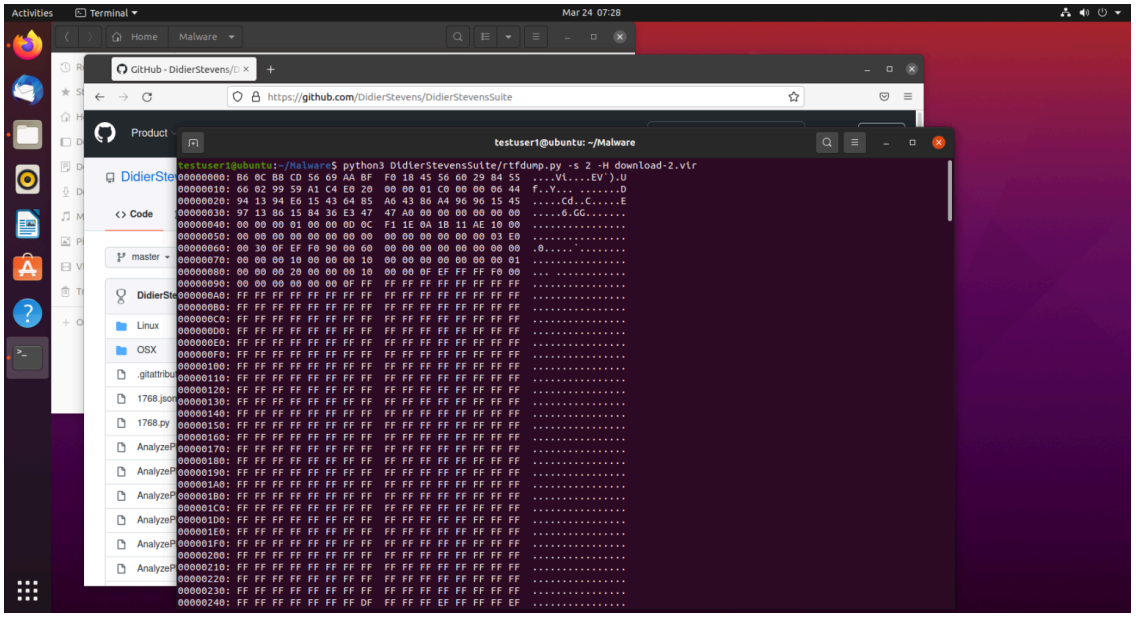


Figure 34: Hexadecimal decoding

After some randomly looking bytes and a series of NULL bytes, we see a lot of FF bytes. This is typical of ole files. Ole files start with a specific set of bytes, known as a magic header: D0 CF 11 E0 A1 B1 1A E1.
We can not find this sequence in the data, however we find a sequence that looks similar: 0D 0C F1 1E 0A 1B 11 AE 10 (starting at position 0x46)
This is almost the same as the magic header, but shifted by one hexadecimal digit. This means that the RTF file is

obfuscated with a method that has not been foreseen in the deobfuscation routines of rtfdump. Remember that the number of hexadecimal digits is uneven: this is the result. Should rtfdump be able to properly deobfuscate this RTF file, then the number would be even.
But that is not a problem: I've foreseen this, and there is an option in rtfdump to shift all hexadecimal strings with one digit. This is option -S:

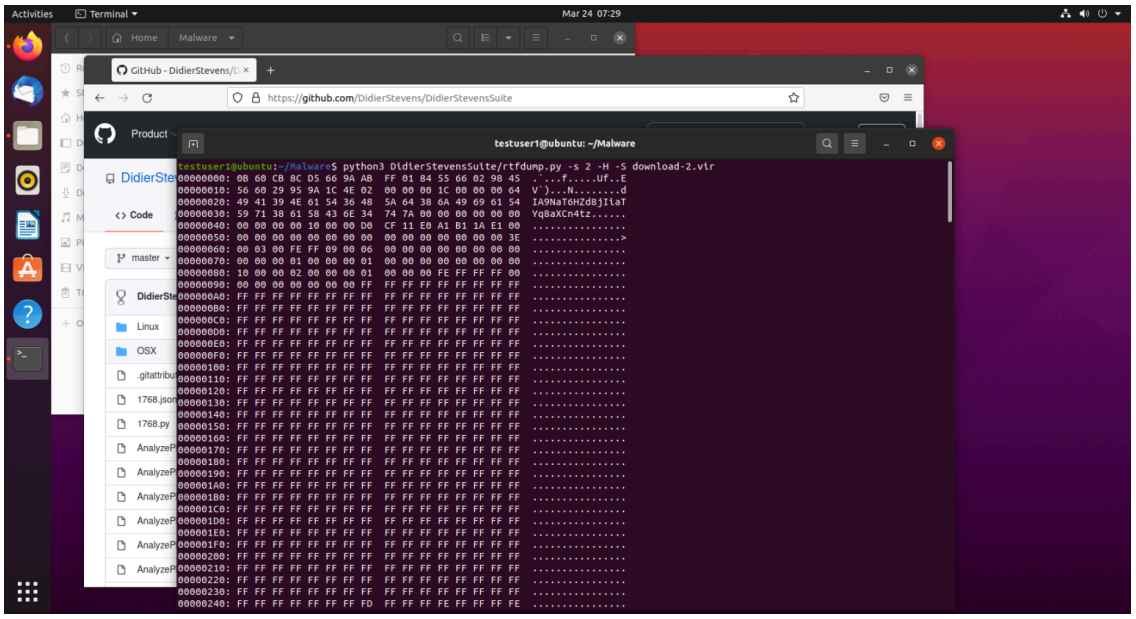


Figure 35: Using option -S to manually deobfuscate the file

We have different output now. Starting at position 0x47, we now see the correct magic header: D0 CF 11 E0 A1 B1 1A E1
And scrolling down, we see the following:

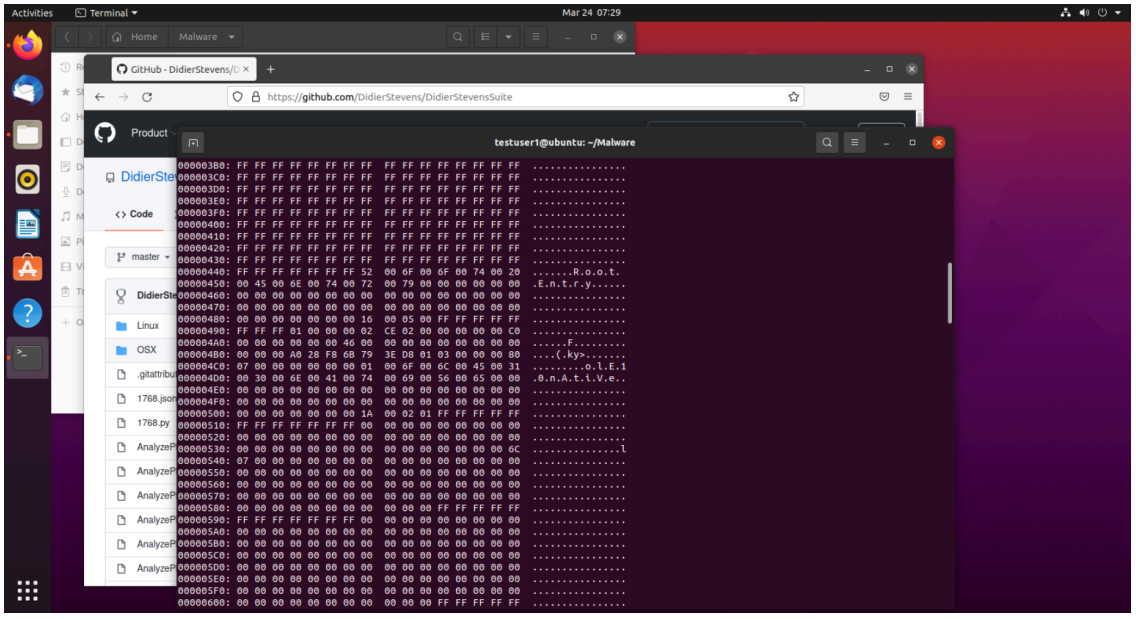


Figure 36: ole file directory entries (UNICODE)

We see UNICODE strings RootEntry and ole10nAtiVE.
Every ole file contains a RootEntry.

And ole10native is an entry for embedded data. It should all be lower case: the mixing of uppercase and lowercase is another indicator for malicious intend.

As we have now managed to direct rtfdump to properly decode this embedded olefile, we can use option -i to help with the extraction:

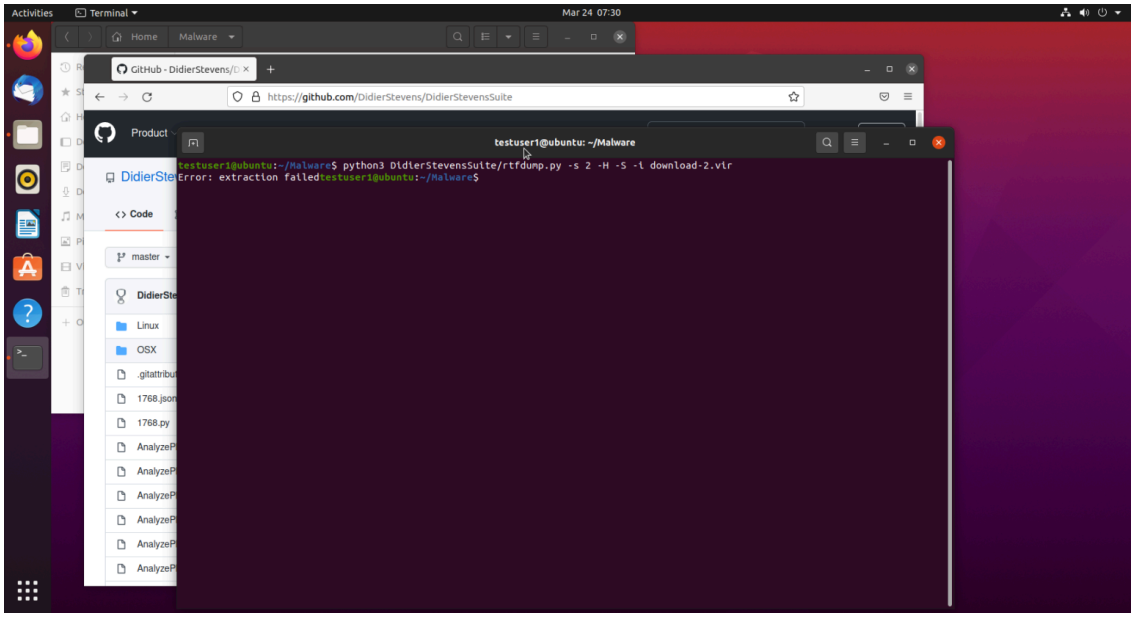


Figure 37: Extraction of the olefile fails

Unfortunately, this fails: there is still some unresolved obfuscation. But that is not a problem, we can perform the extraction manually. For that, we locate the start of the ole file (position 0x47) and use option -c to “cut” it out of the decoded data, like this:

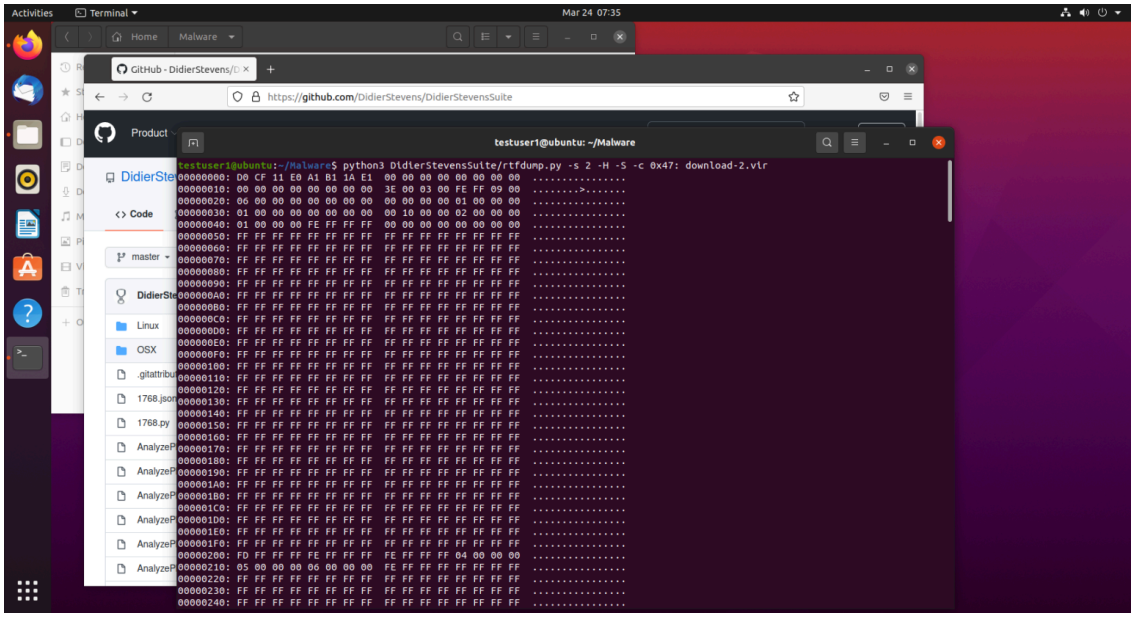


Figure 38: Hexadecimal/ascii dump of the embedded ole file

With option -d, we can perform a dump (binary data) of the ole file and write it to disk:

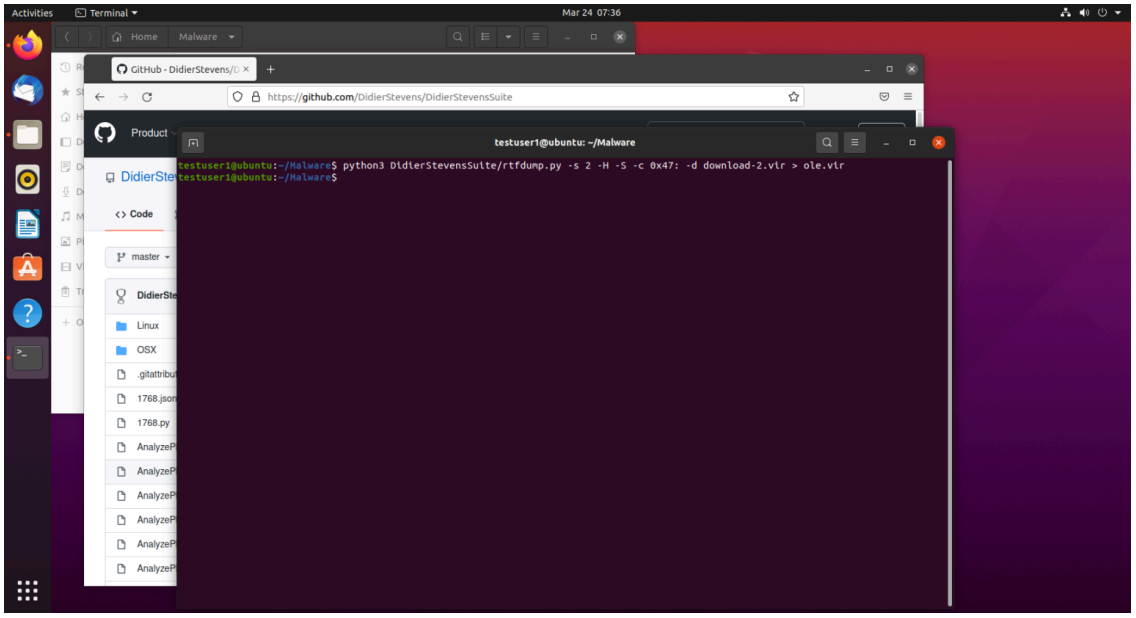


Figure 39: Writing the embedded ole file to disk

We use oledump to analyze the extracted ole file (ole.vir):

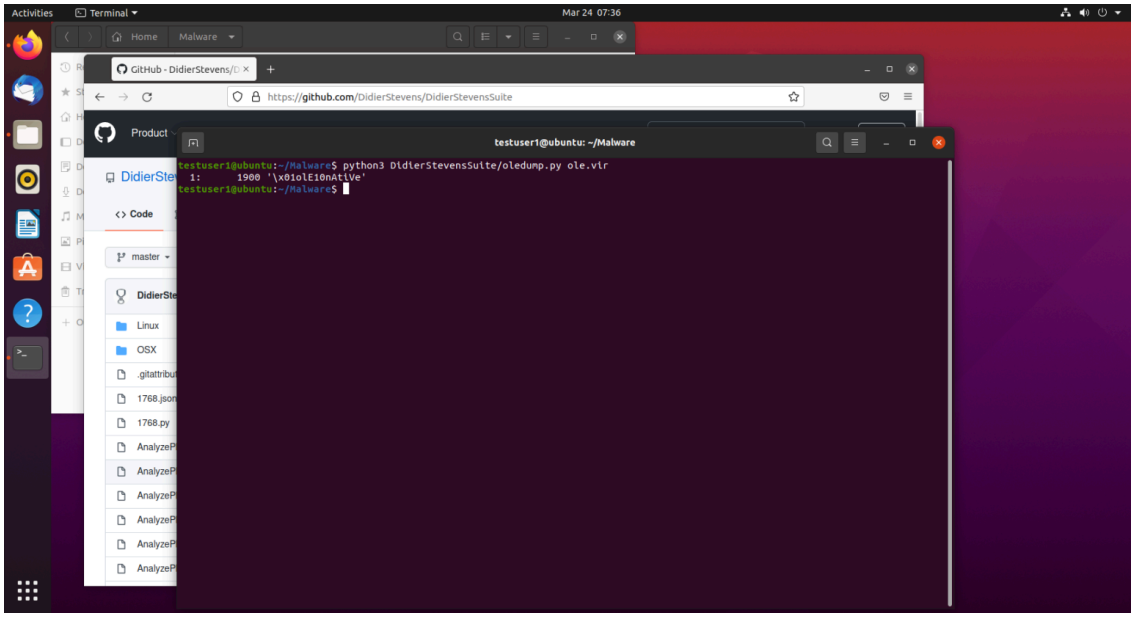


Figure 40: Analysis of the extracted ole file

It succeeds: it contains one stream.

Let’s select it for further analysis:

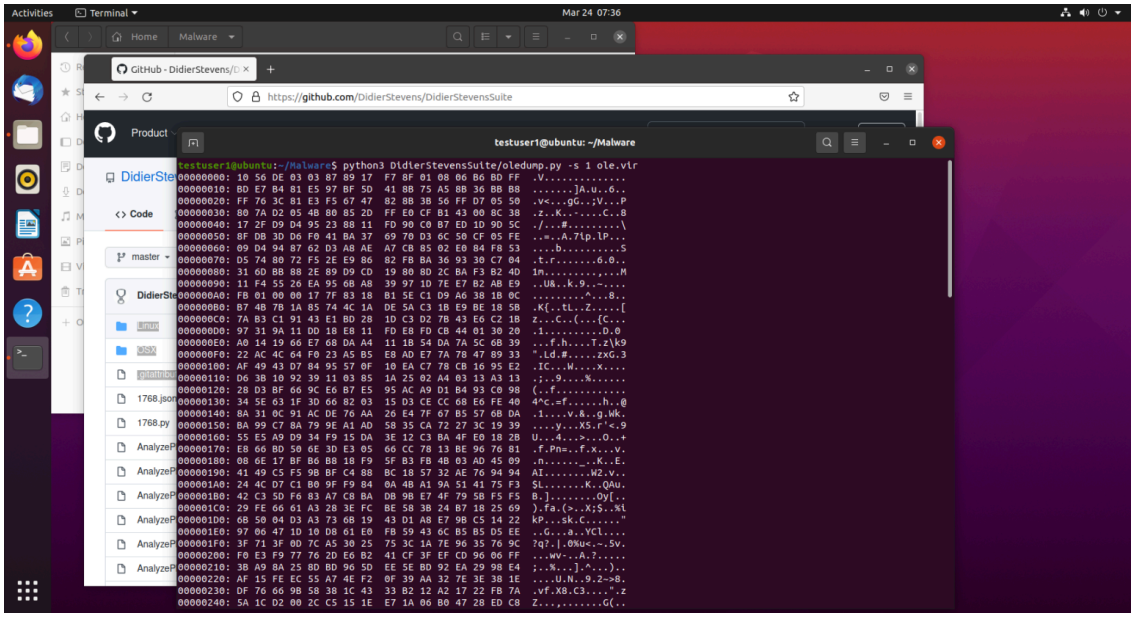


Figure 41: Content of the stream

This binary data looks random.
Let's use option -S to extract strings (this option is like the strings command) from this binary data:

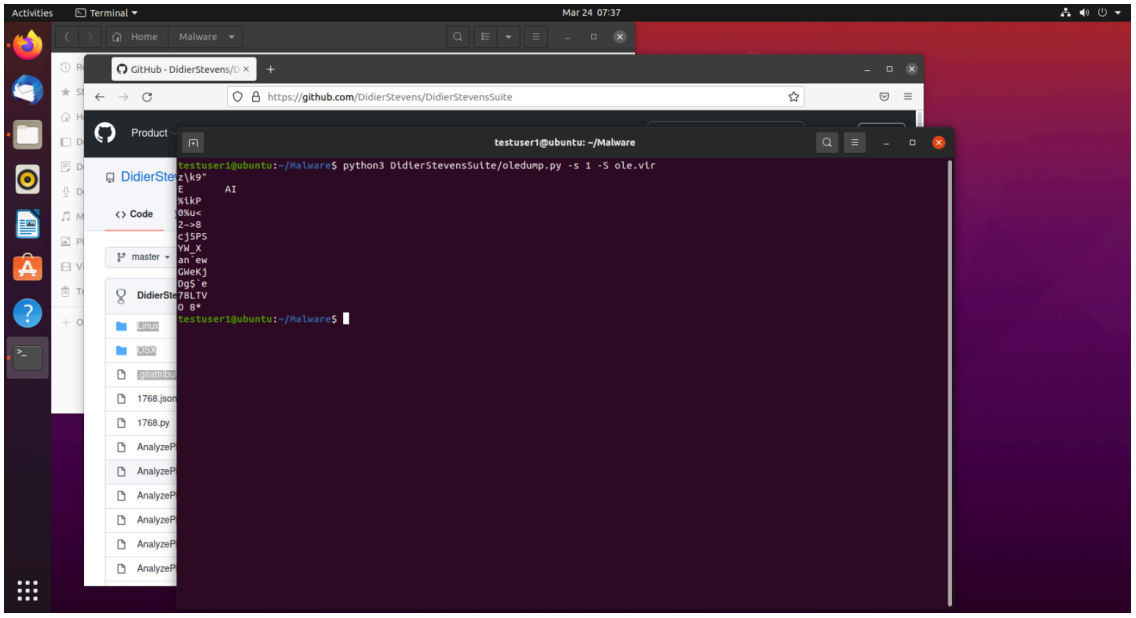


Figure 42: Extracting strings

There's nothing recognizable here.

Let's summarize where we are: we extracted an ole file from an RTF file that was downloaded by a .docx file embedded in a PDF file. When we say it like this, we can only think that this is malicious.

Shellcode analysis

Remember that malicious RTF files very often contain exploits? Exploits often use shellcode. Let's see if we can find shellcode.

To achieve this, we are going to use scdbg, a shellcode emulator developed by David Zimmer.
First we are going to write the content of the stream to a file:

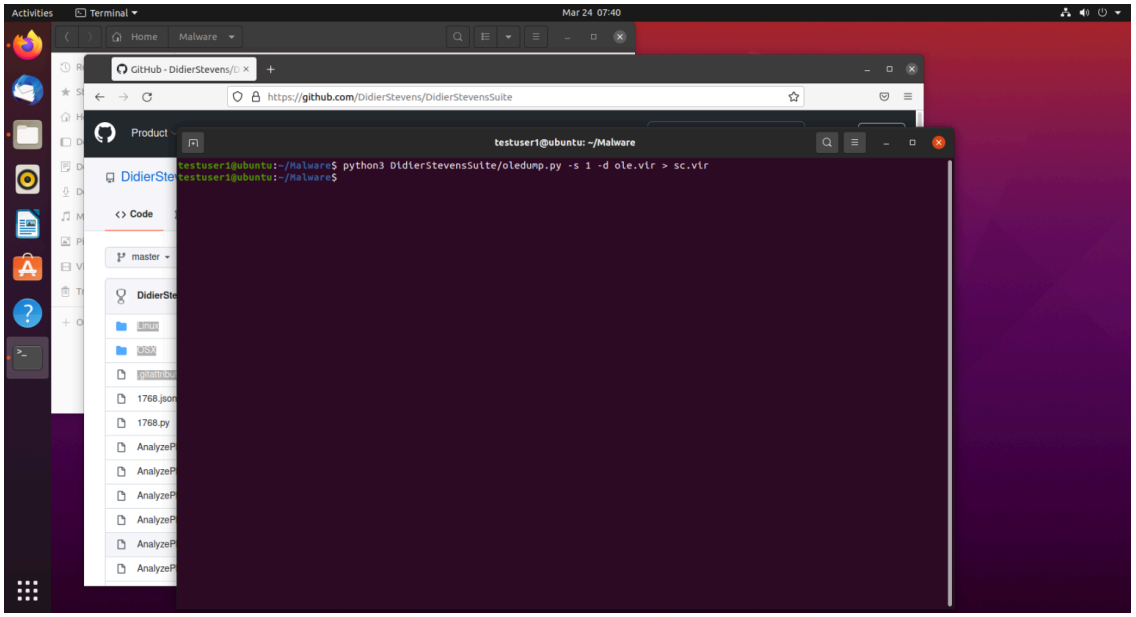


Figure 43: Writing the (potential) shellcode to disk

scdbg is an free, open source tool that emulates 32-bit shellcode designed to run on the Windows operating system. Started as a project running on Windows and Linux, it is now further developed for Windows only.

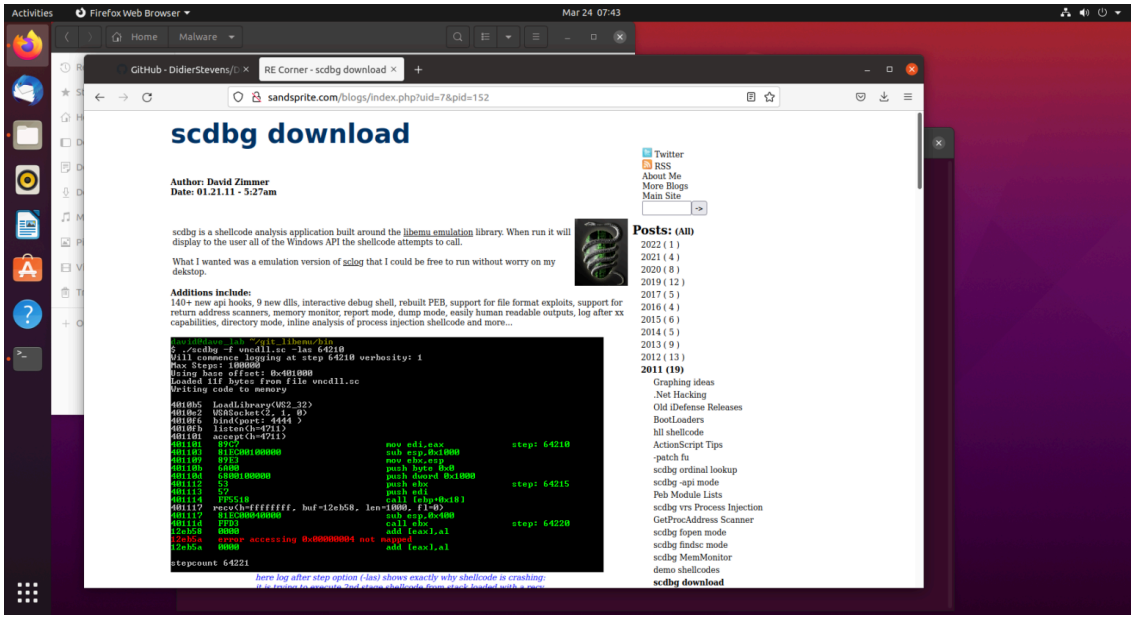


Figure 44: Scdbg

We download Windows binaries for scdbg:

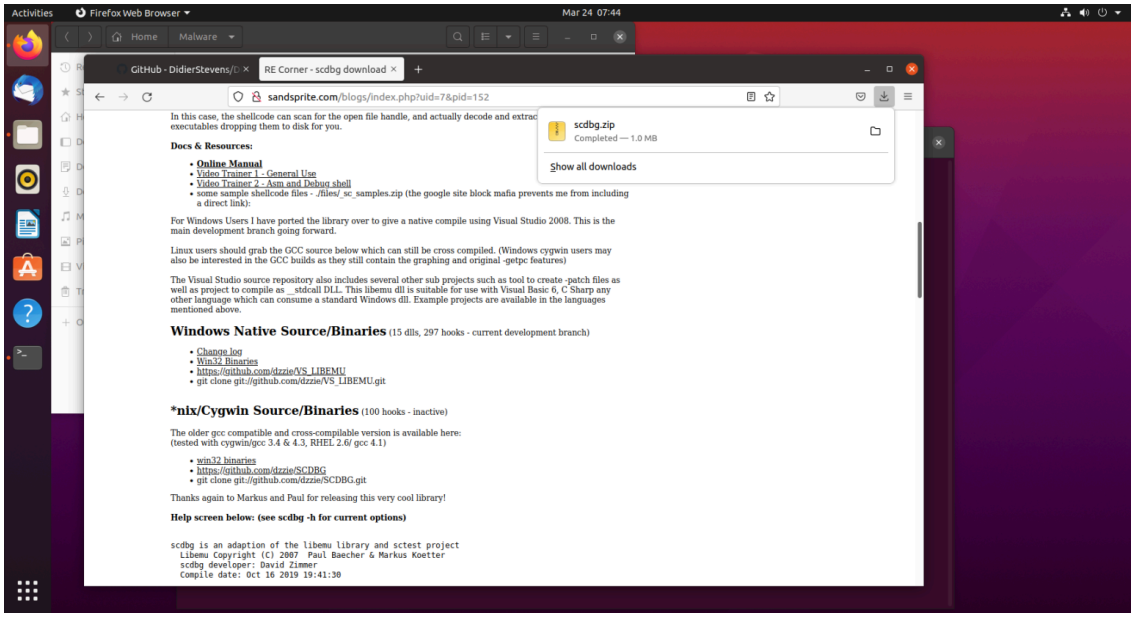


Figure 45: Scdbg binary files

And extract executable scdbg.exe to our working directory:

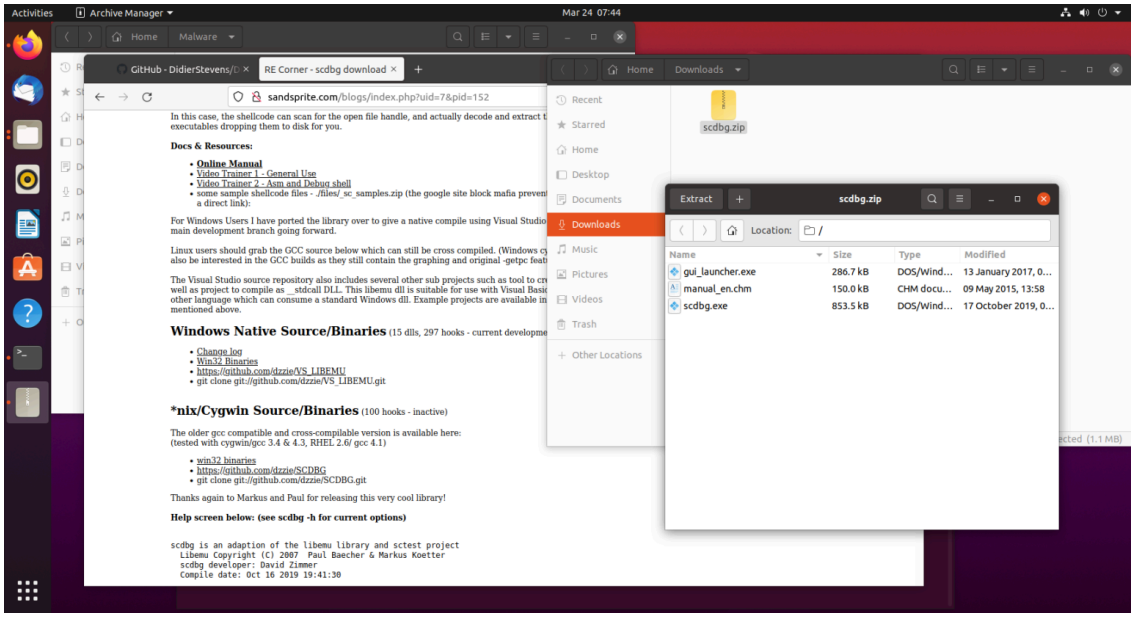


Figure 46: Extracting scdbg.exe

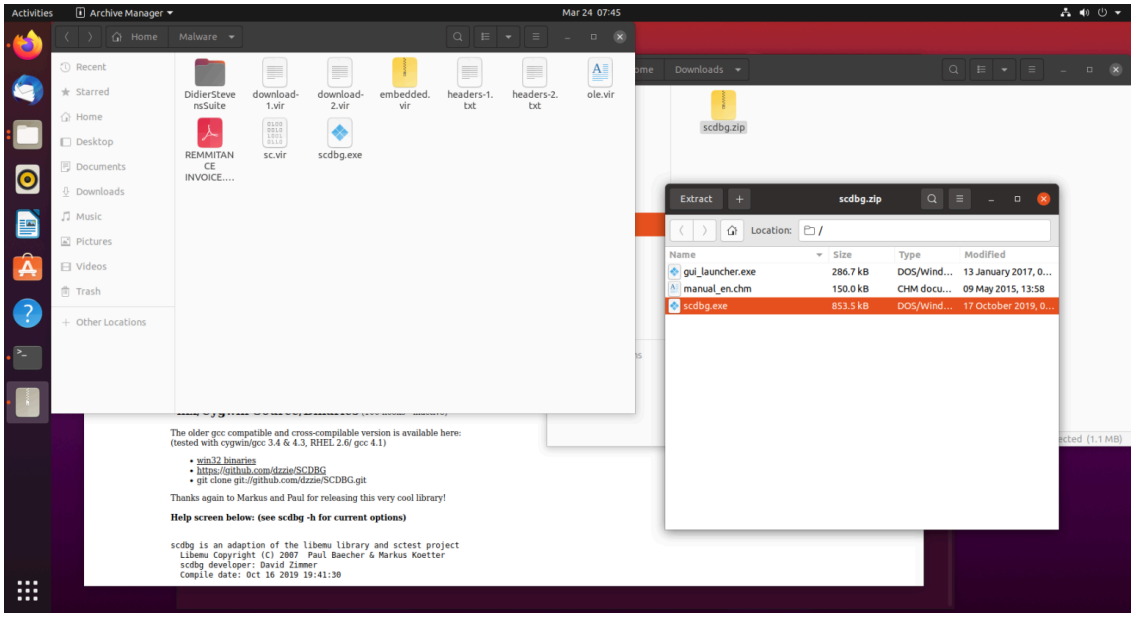


Figure 47: Extracting scdbg.exe

Although scdbg.exe is a Windows executable, we can run it on Ubuntu via Wine:

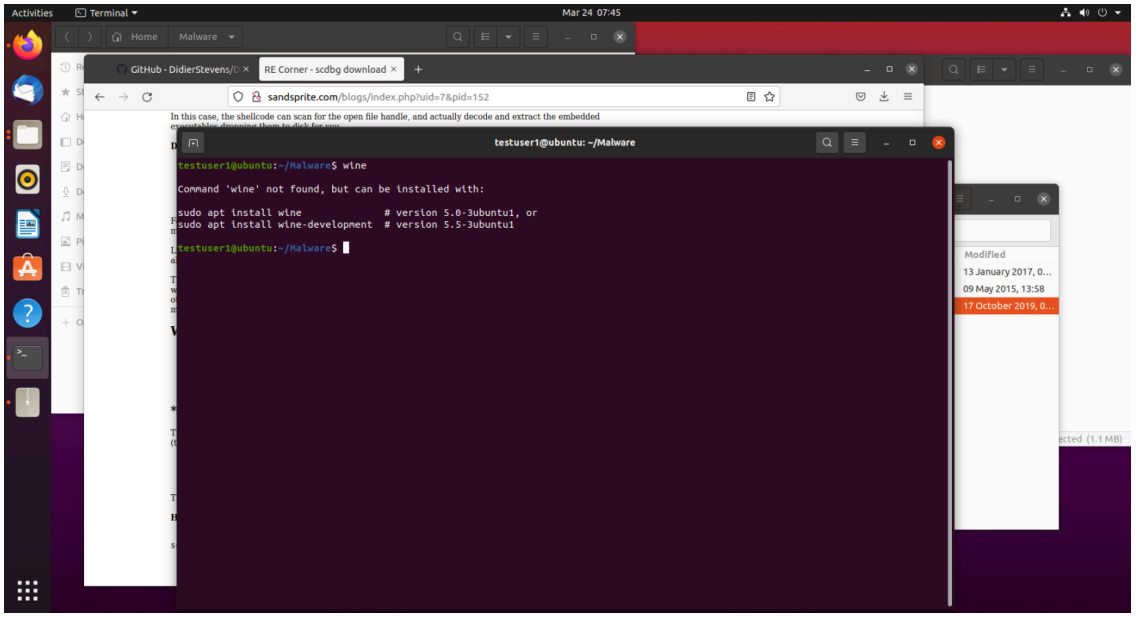


Figure 48: Trying to use wine

Wine is not installed, but by now, we know how to install tools like this:

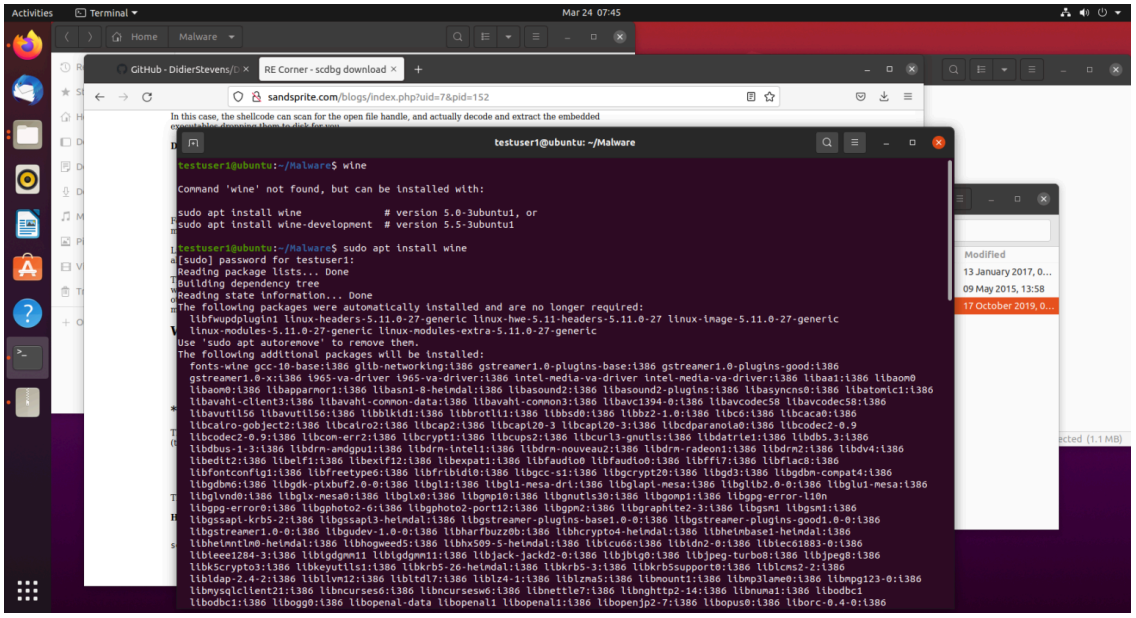


Figure 49: Installing wine

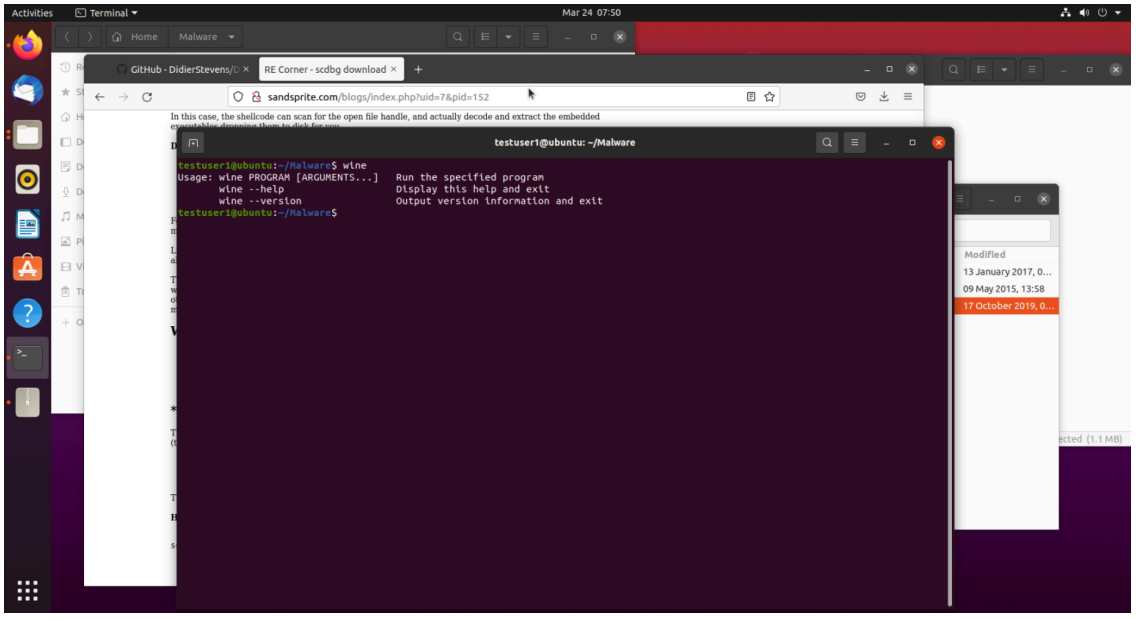


Figure 50: Tasting wine 😊

We can now run scdbg.exe like this:

wine scdbg.exe

scdbg requires some options: -f sc.vir to provide it with the file to analyze

Shellcode has an entry point: the address from where it starts to execute. By default, scdbg starts to emulate from address 0. Since this is an exploit (we have not yet recognized which exploit, but that does not prevent us from trying to analyze the shellcode), its entry point will not be address 0. At address 0, we should find a data structure (that we have not identified) that is exploited.
To summarize: we don't know the entry point, but it's important to know it.

Solution: scdbg.exe has an option to try out all possible entry points. Option -findsc.
And we add one more option to produce a report: -r.

Let's try this:

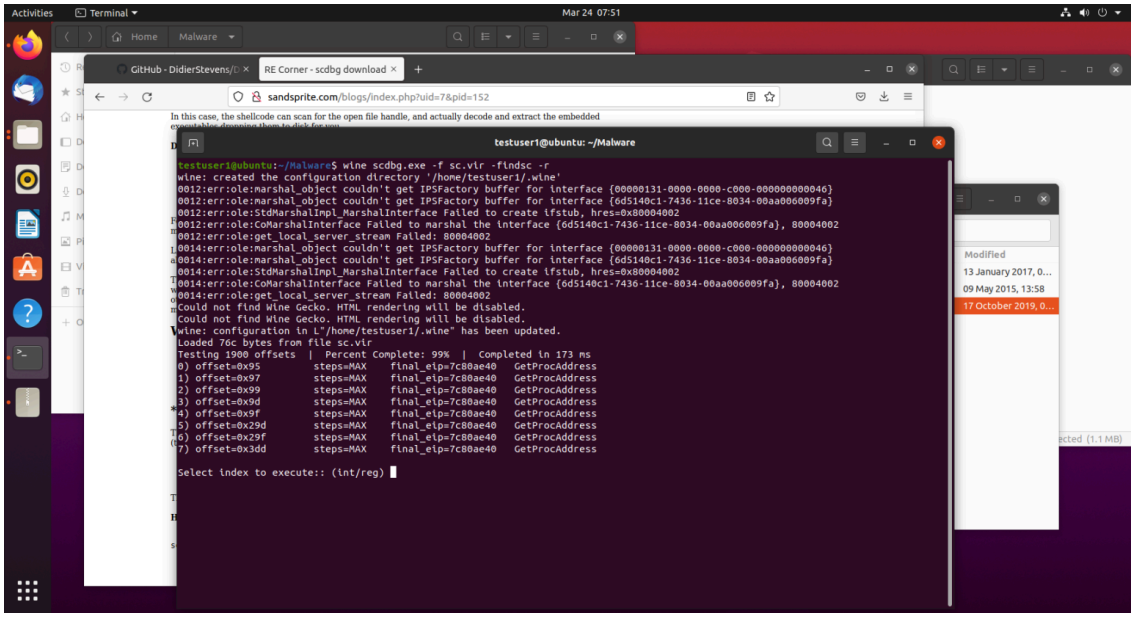


Figure 51: Running scdbg via wine

This looks good: after a bunch of messages and warnings from Wine that we can ignore, scdbg proposes us with 8 (0 through 7) possible entry points. We select the first one: 0

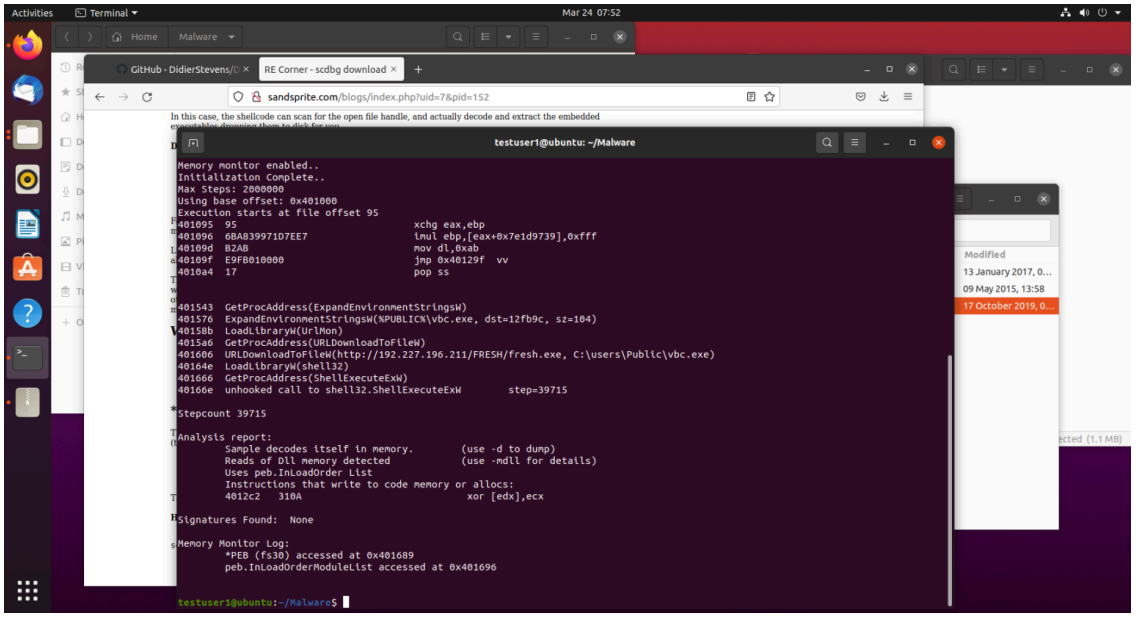


Figure 52: Trying entry point 0 (address 0x95)

And we are successful: scdbg.exe was able to emulate the shellcode, and show the different Windows API calls performed by the shellcode. The most important one for us analysts, is URLDownloadToFile. This tells us that the shellcode downloads a file and writes it to disk (name vbc.exe).

Notice that scdbg did emulate the shellcode: it did not actually execute the API calls, no files were downloaded or written to disk.

Although we don't know which exploit we are dealing with, scdbg was able to find the shellcode and emulate it, providing us with an overview of the actions executed by the shellcode.
The shellcode is obfuscated: that is why we did not see strings like the URL and filename when extracting the strings (see figure 42). But by emulating the shellcode, scdbg also deobfuscates it.

We can now use curl again to try to download the file:

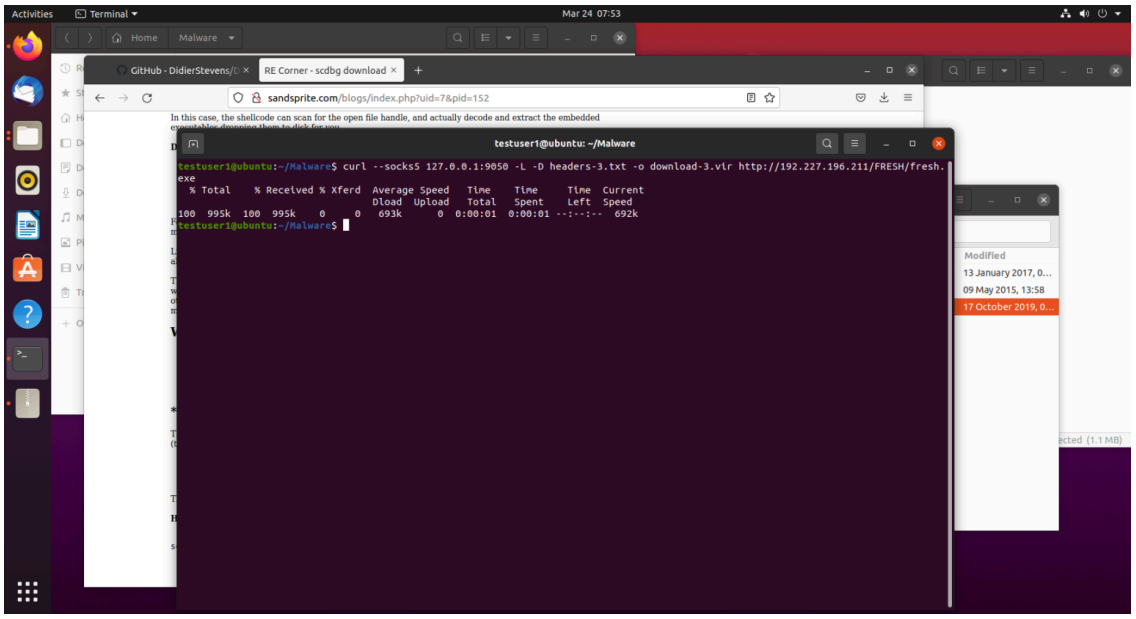


Figure 53: Downloading the executable

And it is indeed a Windows executable (.NET):

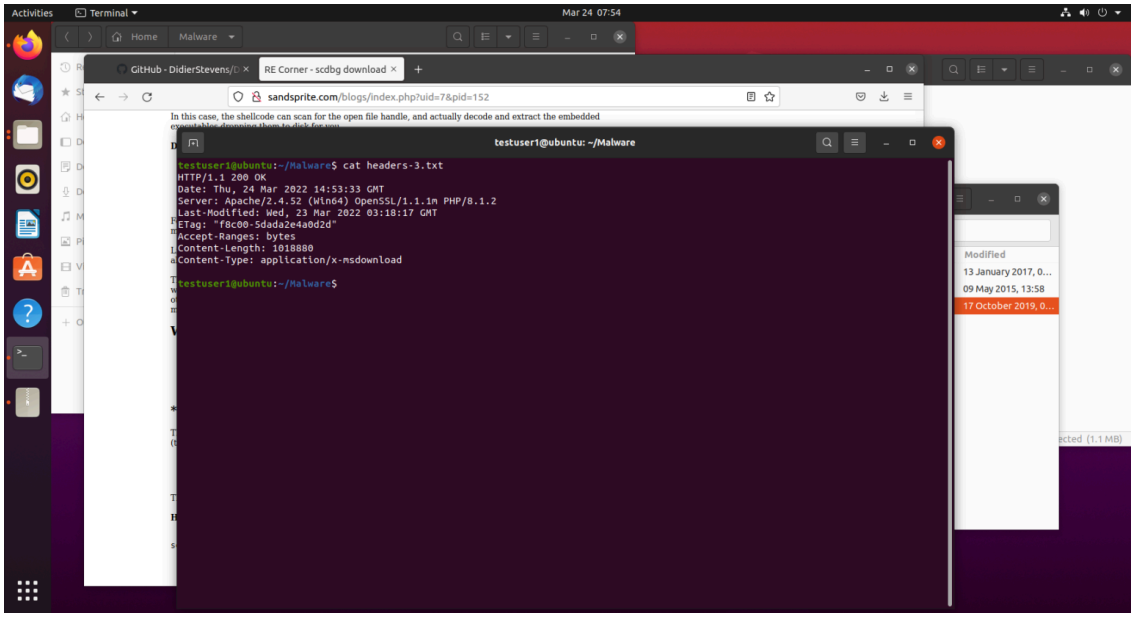


Figure 54: Headers

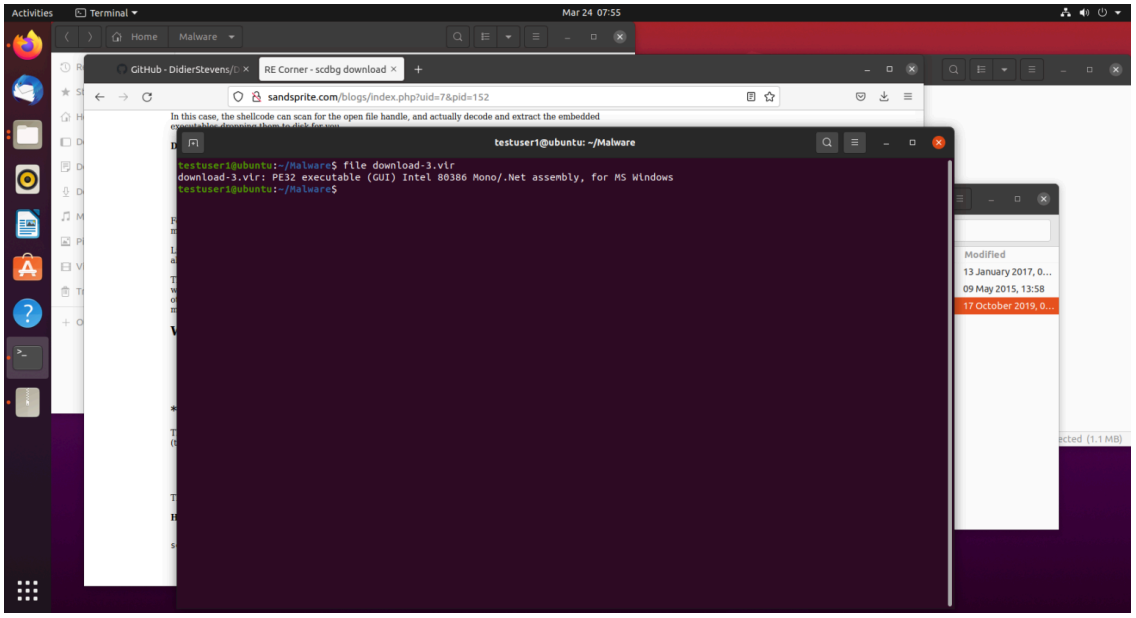


Figure 55: Running command file on the downloaded file

To determine what we are dealing with, we try to look it up on VirusTotal.
First we calculate its hash:

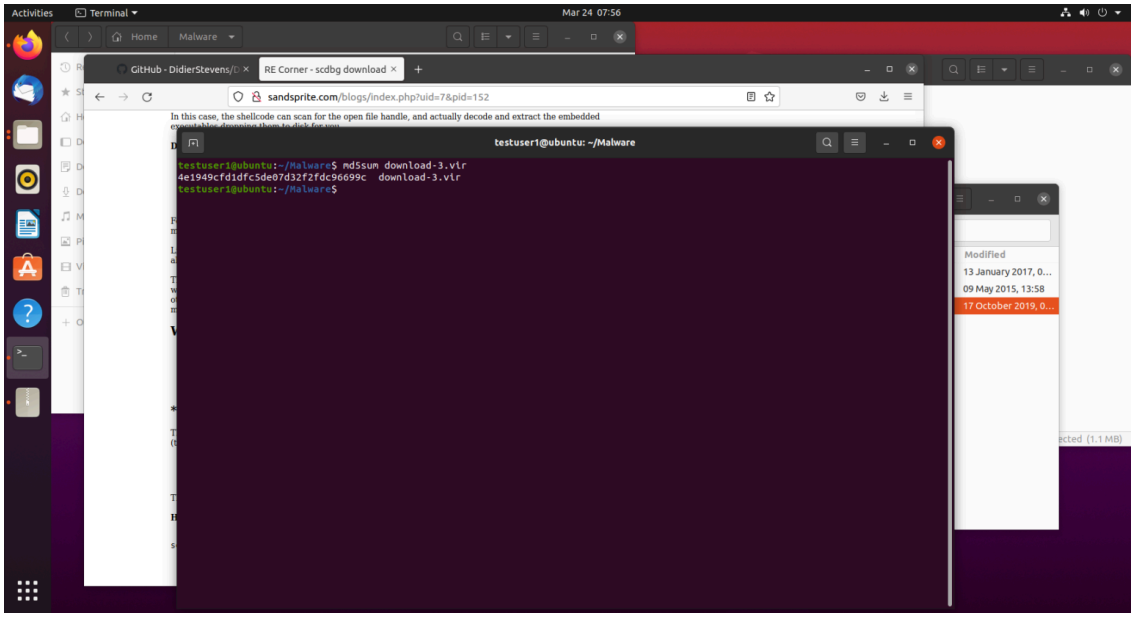


Figure 56: Calculating the MD5 hash

And then we look it up through its hash on VirusTotal:

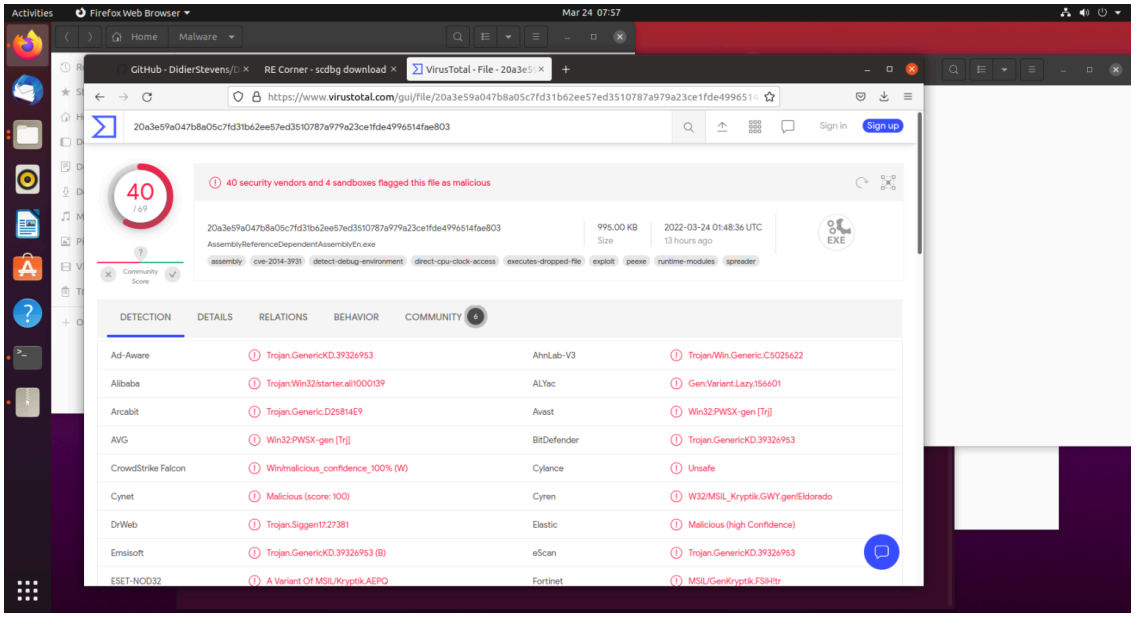


Figure 57: VirusTotal report

From this report, we conclude that the executable is Snake Keylogger.

If the file would not be present on VirusTotal, we could upload it for analysis, provided we accept the fact that we can potentially alert the criminals that we have discovered their malware.

In the video for this blog post, there's a small bonus at the end, where we identify the exploit: CVE-2017-11882.

Conclusion

This is a long blog post, not only because of the different layers of malware in this sample. But also because in this blog post, we provide more context and explanations than usual.
We explained how to install the different tools that we used.
We explained why we chose each tool, and why we execute each command.
There are many possible variations of this analysis, and other tools that can be used to achieve similar results. I for example, would pipe more commands together.
The important aspect to static analysis like this one, is to use dedicated tools. Don't use a PDF reader to open the PDF, don't use Office to open the Word document, … Because if you do, you might execute the malicious code. We have seen malicious documents like this before, and written blog post for them like this one. The sample we analyzed here, has more "layers" than these older maldocs, making the analysis more challenging.

In that blog post, we also explain how this kind of malicious document "works", by also showing the JavaScript and by opening the document inside a sandbox.

IOCs

Type	Value
PDF	sha256: 05dc0792a89e18f5485d9127d2063b343cfd2a5d497c9b5df91dc687f9a1341d
RTF	sha256: 165305d6744591b745661e93dc9feaea73ee0a8ce4dbe93fde8f76d0fc2f8c3f

EXE	sha256: 20a3e59a047b8a05c7fd31b62ee57ed3510787a979a23ce1fde4996514fae803
URL	hxxps://vtaurl[.]com/IHytw
URL	hxxp://192[.]227[.]196[.]211/FRESH/fresh[.]exe

These files can be found on VirusTotal, MalwareBazaar and Malshare.

About the authors

Didier Stevens is a malware expert working for NVISO. Didier is a SANS Internet Storm Center senior handler and Microsoft MVP, and has developed numerous popular tools to assist with malware analysis. You can find Didier on Twitter and LinkedIn.

You can follow NVISO Labs on Twitter to stay up to date on all our future research and publications.
