Linux IR - AI-Assisted Malware Analysis

Taz Wake | Halkyn Consulting Ltd | Originally published 2024-06-06

Introduction

Incident response often has to be fast. We are chasing an active attacker and trying to get control of a situation before anything gets worse. It is not often that we have the luxury of time.

This means that *sometimes* we have to find "quick and dirty" ways to resolve issues. In turn, this often carries some sacrifices - we give up a deep understanding and, instead, rely on a high-level review to make decisions.

Where this becomes most apparent on Linux intrusions, is dealing with malware. Even if we have the right skills and knowledge to take apart complex ELF files, it is rare we can put aside enough time to get value.

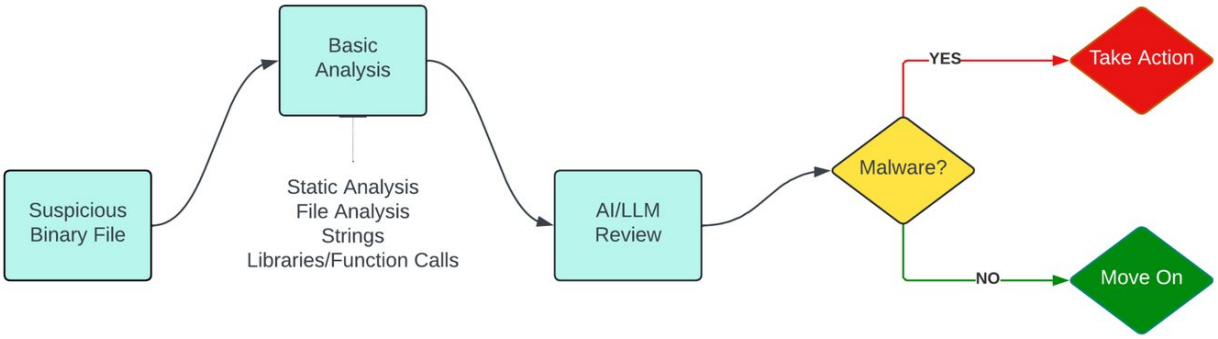
In this article, I will look at ways you can get a superficial assessment by running some basic commands and sharing the data with ChatGPT (or any AI/LLM platform).

Note 1: None of this is a replacement for having trained, capable, and experienced reverse engineers available. The "quick and dirty" approach is just to help you deal with the early stage of the incident. You absolutely need a deeper understanding at some point.

Note 2: I am going to use ChatGPT here but you need to consider your own situation. You need to make sure you have appropriate approvals to upload data to public platforms and you need to understand enough that you can identify where the platform is making a mistake (or just plain old lying to you). This is 100% not a replacement for skilled staff, it is just a way to improve your efficiency.

AI platforms have limits. They can be great for quick assessments but rarely stand up to scrutiny...

Overview



High-level workflow

The high-level workflow basically has four steps:

- Discover a suspicious file during an incident.
- Run basic analysis commands, save the output as a text file.
- Share output with your AI/LLM model and provide a prompt to support your analysis.
- Review output and act on any new knowledge.

The good news is that a lot of this is definitely scriptable and you could easily build this into your SOAR application with API calls to the AI/LLM platform. In this article we will focus on the basic commands to run and some suggested prompts.

A lot of this process will rely on you keeping good notes during your IR work. It should go without saying that this is critically important!

Basic Analysis Commands

In this article, I will use malware.elf as a placeholder for the file you are analysing.

Basic Information

- Note the filename, where it was discovered and any immediately relevant information or clues about its behaviour.
- Collect file metadata

This is an example of what the output should look like, although in use this would be redirected to a text file.

```
root@siftworkstation:~/malware# file malware.elf
malware.elf: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
missing section headers at 105168
```

```
root@siftworkstation:~/malware# sha1sum malware.elf
7f59204b53cd7fab4ce40e73a97115130e7382ee  malware.elf
```

Static Analysis

readelf

The readelf command is used to extract information from an ELF file. It provides details on things such as the section headers, program headers, symbols etc. It has a range of command line arguments which allow you specify what specific items you are interested in, but for our purposes we will use the **-a** argument to collect all headers.

The output which is redirected into the text file should look something like this:

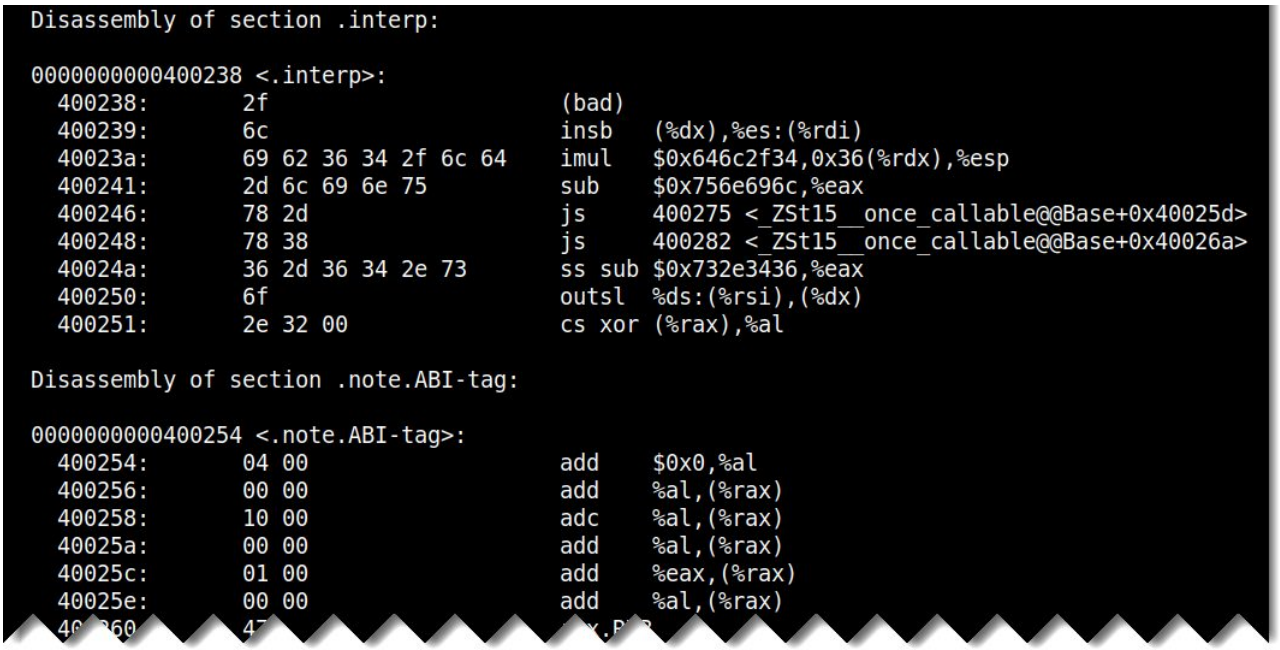
```
root@siftworkstation:~/malware# readelf -a malware.elf
readelf: Error: ELF Header:
Reading 1792 bytes extends past end of file for section headers
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
readelf: Error:   Data:                              2's complement, little endian
Section headers are not available!
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x401530
  Start of program headers:          64 (bytes into file)
  Start of section headers:          103440 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         28
  Section header string table index: 27

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001f8 0x00000000000001f8  R E    0x8
  INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
```

Example readelf output

objdump

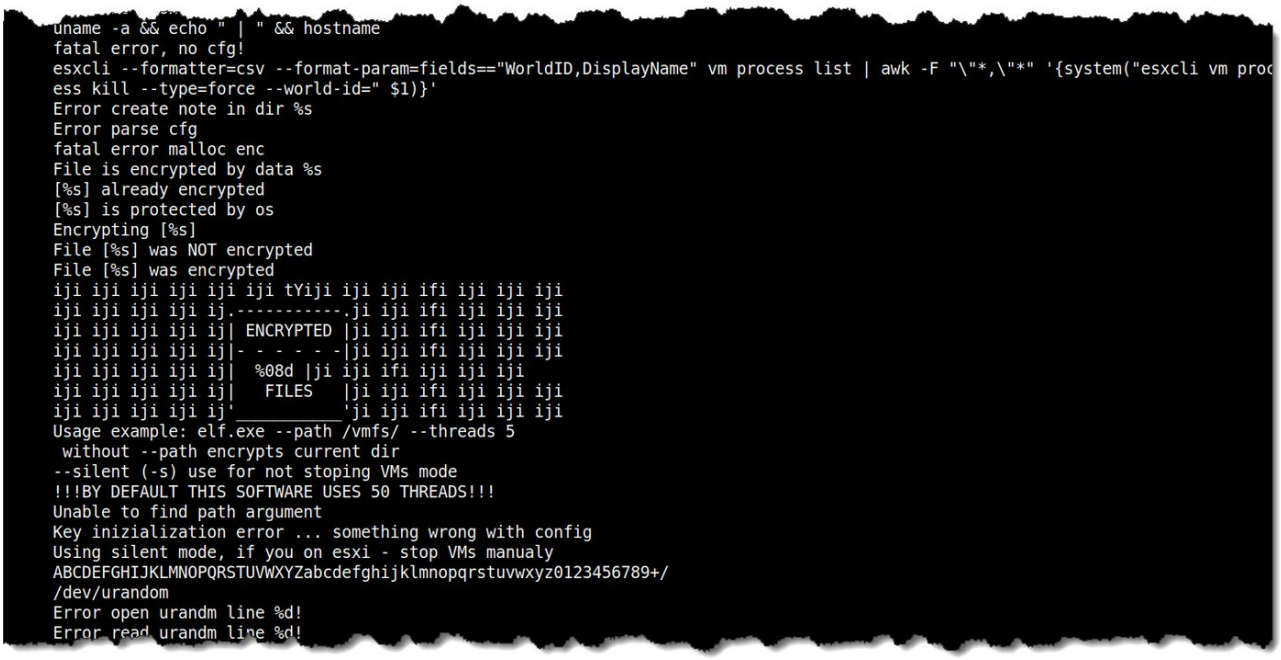
Objdump is similar but different. Again, there are a range of options we can use to specify which objects we want, but for speed and simplicity, we will use the -D argument to disassemble all the sections, not just the ones expected to contain instructions. Please be aware that this is very noisy and might result in several hundred thousand lines of output. It might also result in nothing depending on how the malware is compiled.



Example Objdump output

strings

Often overlooked, strings is an excellent way of identifying human-readable content in a file. You can specify the minimum length string and, as a general guide, I find starting with -n8 is a good approach.



Example output from running strings on a malware sample

ldd

Finally a quick check of any libraries the binary calls with the ldd command.

This varies in effectiveness, with lots of malware families (especially ransomware) coming up empty here.

```
root@siftworkstation:~/malware# ldd suspicious
        linux-vdso.so.1 (0x00007ffe8e8c3000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f1b0a889000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f1b0a7a2000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f1b0a57a000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f1b0a8a8000)
```

ldd output from a SIDEWALK malware sample

Dynamic Analysis

Next is checking how the application works. This does present an element of risk as for strace and ltrace, the binary has to be made executable. **Always approach any analysis with caution**.

gdb

You can use gdb (the "Gnu Debugger") to get a rapid assessment of some key components of the file. You can do this manually, or for speed, we can use some command line arguments to extract commonly analysed elements.

For our purposes this command will be useful, *most* of the time.

This is an example of how the output might look.

```
(No debugging symbols found in suspicious)
Symbols from "/root/malware/suspicious".
Local exec file:
        `/root/malware/suspicious', file type elf64-x86-64.
        Entry point: 0x45acb4
        0x0000000000400238 - 0x0000000000400254 is .interp
        0x0000000000400254 - 0x0000000000400274 is .note.ABI-tag
        0x0000000000400278 - 0x00000000000408bf4 is .hash
        0x00000000000408bf8 - 0x0000000000424468 is .dynsym
        0x0000000000424468 - 0x00000000004546a8 is .dynstr
        0x00000000004546a8 - 0x0000000000456b5c is .gnu.version
        0x0000000000456b60 - 0x0000000000456c30 is .gnu.version_r
        0x0000000000456c30 - 0x00000000004587d8 is .rela.dyn
        0x00000000004587d8 - 0x0000000000459c00 is .rela.plt
        0x0000000000459c00 - 0x0000000000459c18 is .init
```

Example output from a SIDEWALK malware sample

For some malware samples this won't work - especially if they have a lot of protection mechanisms built in, so don't be too surprised if the output is effectively blank. A good reverse engineer will work around this but that is outside the scope of our rapid assessment.

```
No symbol table is loaded.  Use the "file" command.
All defined functions:
All defined variables:
No stack.
```

Example from a Sodinokibi ransomware sample

While gdb is generally safe to run, the next two commands do significantly increase the risks. **The file must be executable and the binary is running on the system**. If this is a malware sample, it is likely to be able to run its payloads.

Only carry out these steps if you have an isolated analysis environment or sandbox where the impact from (for example) deploying ransomware would be limited.

strace

This command attaches a debugger to the running process and records system calls. At the most basic use, it will monitor the file until it exits which can be a considerable time. It is often a good idea to use the timeout command to avoid any system hangs.

In this example, strace will write to an output file (-o), follow child processes (-f) and add a timestamp with millisecond precision to each entry. It will trace all system calls (-e trace=all) and capture up to 256 characters of strings.

In use, you have to specify the path correctly so either use full paths or strings to point to objects in the same folder.

Alternatively, you can run the command with a timeout. This makes it much faster to run but can miss malicious activity if there are delays built in (a common anti-analysis technique).

This example will kill the process after 10 seconds (timeout 10) and will summarise the trace (-c). The summary causes the output to lose millisecond precision timestamps.

```
6257  18:02:32.389475 execve("./suspicious", ["./suspicious"], 0x7fff49f79780 /* 18 vars */) = 0
6257  18:02:32.390105 brk(NULL)         = 0xc5b000
6257  18:02:32.391243 arch_prctl(0x3001 /* ARCH_??? */, 0x7fffb1fdf4b0) = -1 EINVAL (Invalid argument)
6257  18:02:32.392694 mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f993b31e000
6257  18:02:32.394321 access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
6257  18:02:32.394612 openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
6257  18:02:32.394939 newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=94263, ...}, AT_EMPTY_PATH) = 0
6257  18:02:32.395212 mmap(NULL, 94263, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f993b306000
6257  18:02:32.395501 close(3)          = 0
6257  18:02:32.395694 openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libpthread.so.0", O_RDONLY|O_CLOEXEC) = 3
6257  18:02:32.395871 read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0\0@\0\0\0\0\0\0\0\210J\0\0\0\0\0\0
\0\0\0@\08\0\v\0@\0%\0$\0\1\0\0\0\4\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\250\n\0\0\0\0\0\0\250\n\0\0\0\0\0\0
\20\0\0\0\0\0\0\1\0\0\0\5\0\0\0\0\20\0\0\0\0\0\0\0\20\0\0\0\0\0\0\0\20\0\0\0\0\0\0\25\1\0\0\0\0\0\0\25\1\0\0\0\0\0\0\0\20\0\0
\0\0\0\1\0\0\0\4\0\0\0\0 \0\0\0\0\0\0\0 \0\0\0\0\0\0\0 \0\0\0\0\0\0\210\1\0\0\0\0\0\0\210\1\0\0\0\0\0\0\0\20\0\0\0\0\0\0\1\0\0
0\6\0\0\0\0.\0\0\0\0\0\0\0>\0\0\0\0\0\0"..., 832) = 832
6257  18:02:32.396172 newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=21448, ...}, AT_EMPTY_PATH) = 0
6257  18:02:32.396857 mmap(NULL, 16424, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f993b301000
6257  18:02:32.397188 mmap(0x7f993b302000, 4096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1000) = 0x7f99
b302000
6257  18:02:32.397476 mmap(0x7f993b303000, 4096, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7f993b303000
6257  18:02:32.397730 mmap(0x7f993b304000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7f9
3b304000
6257  18:02:32.397971 close(3)          = 0
6257  18:02:32.398540 openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libm.so.6", O_RDONLY|O_CLOEXEC) = 3
```

Example of strace output against a SIDEWALK malware sample with millisecond timestamps

ltrace

ltrace is similar to strace in that it runs the file, however, this time it intercepts and records dynamic library calls.

In this example, the commands are:

-o output file.

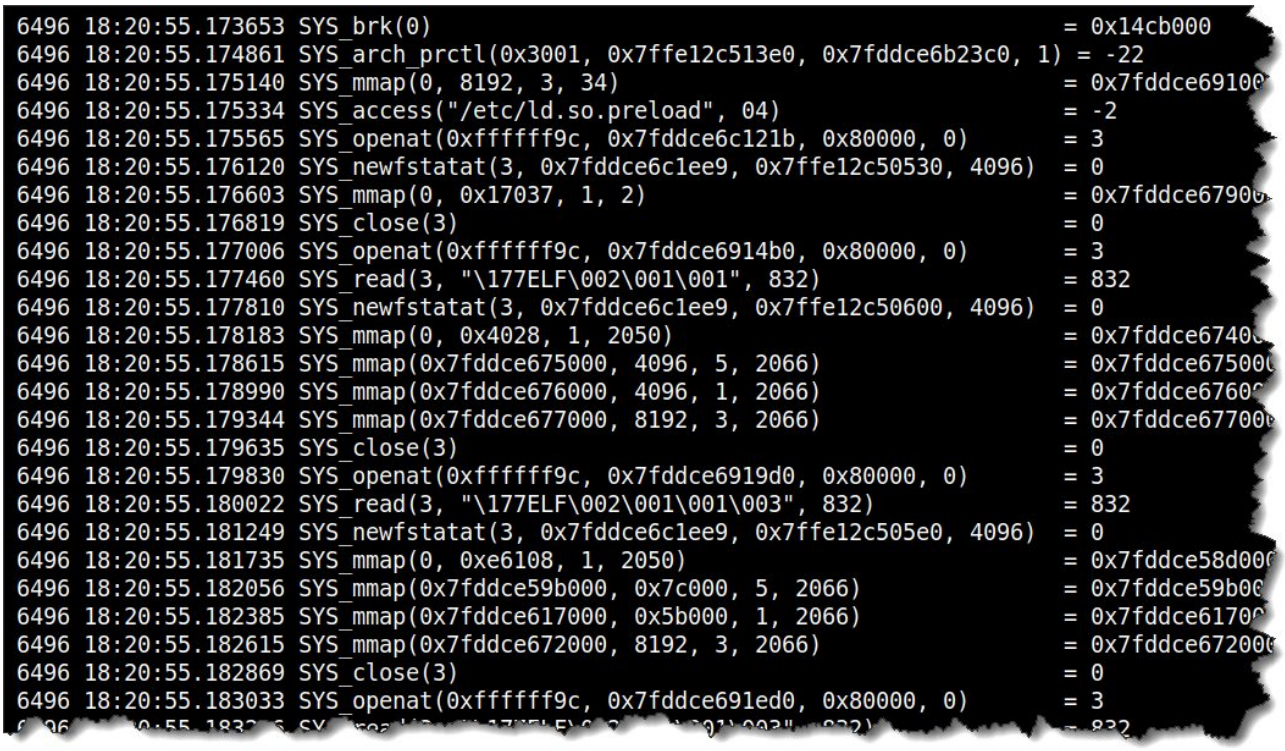
-f follow child processes

-S show system calls

-C translate C++ symbols to make the output more readable

-tt microsecond timestamps

This is also a good candidate for using the timeout command to prevent it hanging for long periods of time.



Example ltrace output from a SIDEWALK malware sample

Scripting it

Now, when we have a situation where we run the same set of commands each time, then it is crying out to be scripted. This is no exception.

A starter example using a bash script to run the commands above is available at https://for577.com/analysisscript

This is not likely to be perfect in every environment, instead, it should be seen as a starting point to build your own. If you have access to an AI/LLM with an API gateway you could even look to automate the submission and response parts.

Working with "AI"

Once you have carried out the basic analysis, it is time to ask your AI platform for help. The exact syntax and process here will depend on your investigation (and the platform) but, in general, it works best to provide the evidence and ask for assistance in analysing the files.

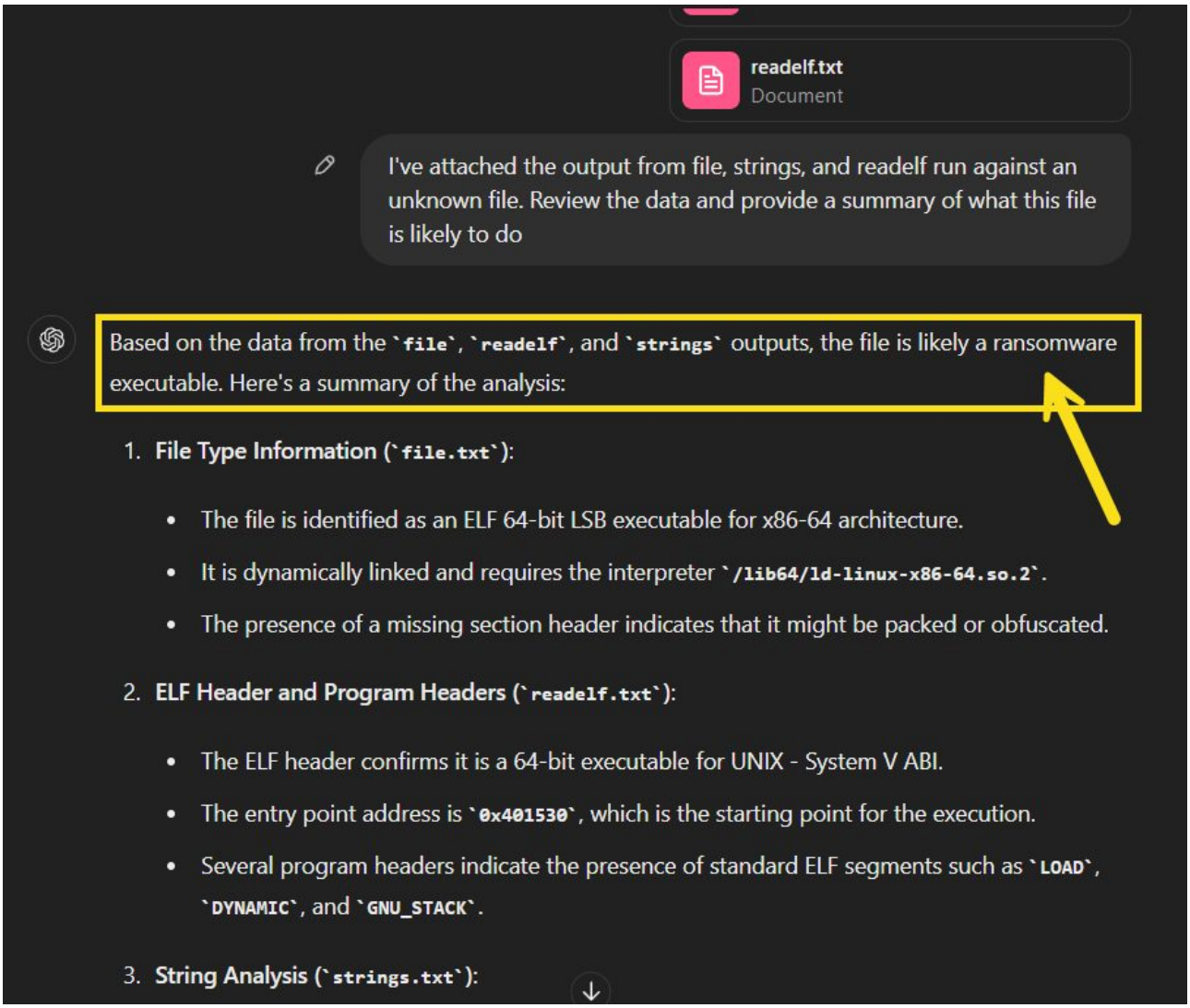
Most LLM tools will struggle to make a definite "malware/not malware" assessment and you need to be wary of their answers. Instead, it is better to have the tool explain what the software does and then make the determination yourself.

Some example prompts which have worked well for me are:

> *I've attached the output from file, strings, and readelf run against an unknown file. Review the data and provide a summary of what this file is likely to do*

(This was for a suspected ransomware sample and no dynamic analysis took place)

With this information, ChatGPT was able to quickly identify the file as probably ransomware. In this case, it was correct and the sample was Sodinokibi (https://www.virustotal.com/gui/file/a322b230a3451fd11dcfe72af4da1df07183d6aaf1ab9e062f0e6b14cf6d23cd)



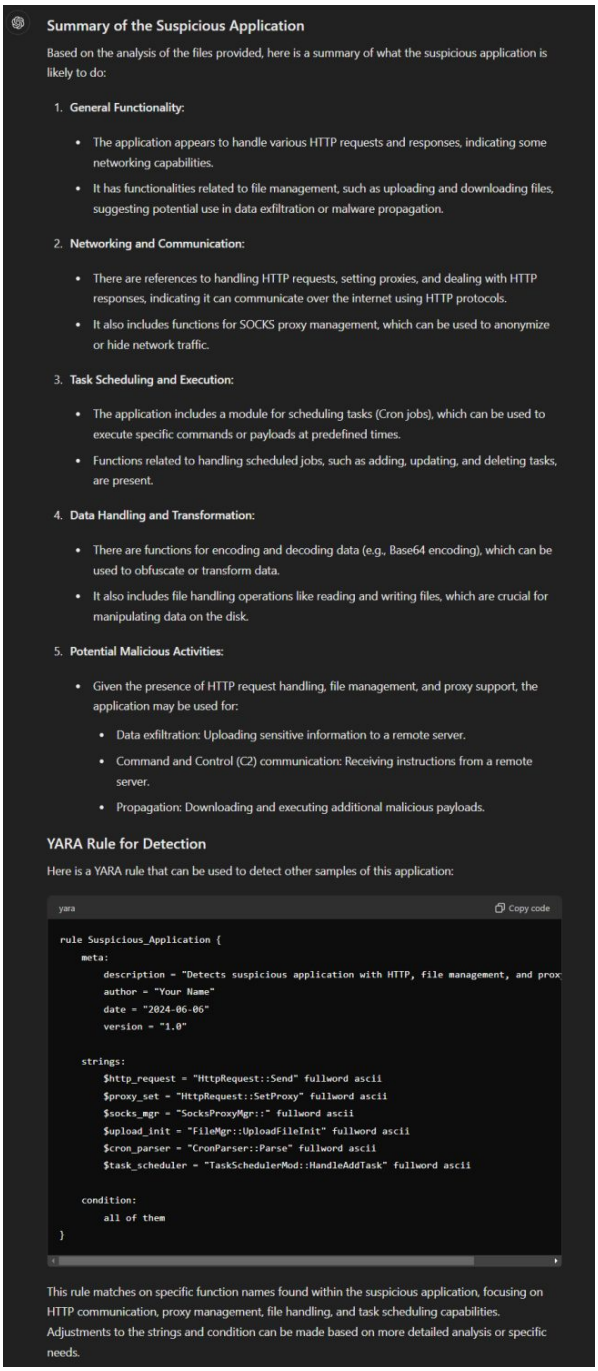
Correctly identified Sodinokibi

It is worth noting that this detection was almost entirely down to the strings data.

Another example is:

> *Attached is the output from gdb, strings, file, ldd, readelf, strace and ltrace for a suspicious file. Read them and provide a summary of what this application is likely to do. Also provide a yara rule to hunt for other samples.*

With this sample, objdump was 15mb in size and rejected by ChatGPT. However, the other files were sufficient for a good determination. Although the response was verbose, it did correctly identify that this was a C2 implant.



Summary of the Suspicious Application

Based on the analysis of the files provided, here is a summary of what the suspicious application is likely to do:

1. **General Functionality:**
   - The application appears to handle various HTTP requests and responses, indicating some networking capabilities.
   - It has functionalities related to file management, such as uploading and downloading files, suggesting potential use in data exfiltration or malware propagation.
2. **Networking and Communication:**
   - There are references to handling HTTP requests, setting proxies, and dealing with HTTP responses, indicating it can communicate over the internet using HTTP protocols.
   - It also includes functions for SOCKS proxy management, which can be used to anonymize or hide network traffic.
3. **Task Scheduling and Execution:**
   - The application includes a module for scheduling tasks (Cron jobs), which can be used to execute specific commands or payloads at predefined times.
   - Functions related to handling scheduled jobs, such as adding, updating, and deleting tasks, are present.
4. **Data Handling and Transformation:**
   - There are functions for encoding and decoding data (e.g., Base64 encoding), which can be used to obfuscate or transform data.
   - It also includes file handling operations like reading and writing files, which are crucial for manipulating data on the disk.
5. **Potential Malicious Activities:**
   - Given the presence of HTTP request handling, file management, and proxy support, the application may be used for:
     - Data exfiltration: Uploading sensitive information to a remote server.
     - Command and Control (C2) communication: Receiving instructions from a remote server.
     - Propagation: Downloading and executing additional malicious payloads.

YARA Rule for Detection

Here is a YARA rule that can be used to detect other samples of this application:

yara Copy code

```
rule Suspicious_Application {
    meta:
        description = "Detects suspicious application with HTTP, file management, and prox
        author = "Your Name"
        date = "2024-06-06"
        version = "1.0"

    strings:
        $http_request = "HttpRequest::Send" fullword ascii
        $proxy_set = "HttpRequest::SetProxy" fullword ascii
        $socks_mgr = "SocksProxyMgr::" fullword ascii
        $upload_init = "FileMgr::UploadFileInit" fullword ascii
        $cron_parser = "CronParser::Parse" fullword ascii
        $task_scheduler = "TaskSchedulerMod::HandleAddTask" fullword ascii

    condition:
        all of them
}
```

This rule matches on specific function names found within the suspicious application, focusing on HTTP communication, proxy management, file handling, and task scheduling capabilities. Adjustments to the strings and condition can be made based on more detailed analysis or specific needs.

ChatGPT output based on analysis of a SIDEWALK implant.

The Yara rules might need some tweaking but definitely provide a good starting point for incident responders.

Conclusion

Using an AI/LLM can, in the right hands, speed up the incident response cycle and help free up DFIR staff for other tasks. It is important to remember that it does need skilled, knowledgeable staff to get good results.

In the examples here, it took under four minutes to collect the data on the Sodinokbi sample and have a determination, without using API calls. The SIDEWALK took slightly longer (six minutes) but did generate more useful data with dynamic analysis.

It is definitely worth adding this to your DFIR tool box if your organisation allows you to submit data to AI platforms, or if you have an internal AI tool.

About Halkyn Consulting

Halkyn Consulting Ltd is a UK-based security and risk management consultancy providing 24/7 incident response, digital forensics, threat hunting, virtual security leadership (vCISO) and security compliance support to organisations worldwide. We deliver clear, practical, vendor-independent advice based on real-world threats — no hype, no upsell.

More free security resources are available at www.halkynconsulting.co.uk/security-resources.

Web: www.halkynconsulting.co.uk | **Email:** info@halkynconsulting.co.uk | **Tel:** +44 1244 940858