**“zeek -v” for version check**

**For zeek to run always do sudo**

Here we can manage the Zeek service and view the status of the service. Primary management of the Zeek service is done with three commands; "status", "start", and "stop".

“Zeekctl” to open ZeekControl Module

“Start” start the zeek

“Status” view status

“Stop” stop zeek

OR type this

* **zeekctl status**
* **zeekctl start**
* **zeekctl stop**

**zeek -C -r sample.pcap**

* **zeek**: This is the command-line interface for Zeek.
* **-C**: This flag specifies that Zeek should only perform packet capture analysis and should not attempt to perform signature-based detection (like IDS/IPS).
* **-r sample.pcap**: This flag specifies that Zeek should analyze the packet capture file named "sample.pcap". Zeek will read the packets from this file and analyze them according to the specified configuration.

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| **-r** | Reading option, read/process a pcap file. |
| **-C** | Ignoring checksum errors. |
| **-v** | Version information. |
| **zeekctl** | ZeekControl module. |

**Zeek Logs**

Zeek generates log files according to the traffic data. You will have logs for every connection in the wire, including the application level protocols and fields. Zeek is capable of identifying 50+ logs and categorising them into seven categories. Zeek logs are well structured and tab-separated ASCII files, so reading and processing them is easy but requires effort. You should be familiar with networking and protocols to correlate the logs in an investigation, know where to focus, and find a specific piece of evidence.

Each log output consists of multiple fields, and each field holds a different part of the traffic data. Correlation is done through a unique value called "UID". The "UID" represents the unique identifier assigned to each session.

|  |  |  |
| --- | --- | --- |
| **Category** | **Description** | **Log Files** |
| **Network** | Network protocol logs. | *conn.log, dce\_rpc.log, dhcp.log, dnp3.log,*  *dns.log, ftp.log, http.log, irc.log,*  *kerberos.log, modbus.log,*  *modbus\_register\_change.log,*  *mysql.log, ntlm.log, ntp.log, radius.log,*  *rdp.log, rfb.log,*  *sip.log, smb\_cmd.log, smb\_files.log, smb\_mapping.log, smtp.log, snmp.log, socks.log, ssh.log, ssl.log, syslog.log, tunnel.log.* |
| **Files** | File analysis result logs. | *files.log, ocsp.log, pe.log, x509.log.* |
| **NetControl** | Network control and flow logs. | *netcontrol.log, netcontrol\_drop.log,*  *netcontrol\_shunt.log,*  *netcontrol\_catch\_release.log,*  *openflow.log.* |
| **Detection** | Detection and possible indicator logs. | *intel.log, notice.log, notice\_alarm.log,*  *signatures.log, traceroute.log.* |
| **Network Observations** | Network flow logs. | *known\_certs.log, known\_hosts.log,*  *known\_modbus.log,*  *known\_services.log, software.log.* |
| **Miscellaneous** | Additional logs cover external alerts, inputs  and failures. | *barnyard2.log, dpd.log, unified2.log*  *, unknown\_protocols.log,*  *weird.log, weird\_stats.log.* |
| **Zeek Diagnostic** | Zeek diagnostic logs cover system messages, actions and some statistics. | *broker.log, capture\_loss.log,*  *cluster.log, config.log,*  *loaded\_scripts.log,*  *packet\_filter.log, print.log,*  *prof.log, reporter.log,*  *stats.log, stderr.log, stdout.log.* |

Zeek official Documentation “https://docs.zeek.org/en/current/script-reference/log-files.html “

|  |  |  |
| --- | --- | --- |
| **Update Frequency** | **Log Name** | **Description** |
| **Daily** | *known\_hosts.log* | List of hosts that completed  TCP handshakes. |
| **Daily** | *known\_services.log* | List of services used by hosts. |
| **Daily** | *known\_certs.log* | List of SSL certificates. |
| **Daily** | *software.log* | List of software used on the  network. |
| **Per Session** | *notice.log* | Anomalies detected by Zeek. |
| **Per Session** | *intel.log* | Traffic contains malicious  patterns/indicators. |
| **Per Session** | *signatures.log* | List of triggered signatures. |

**Brief log usage primer table;**

|  |  |  |  |
| --- | --- | --- | --- |
| **Overall Info** | **Protocol Based** | **Detection** | **Observation** |
| *conn.log* | *http.log* | *notice.log* | *known\_host.log* |
| *files.log* | *dns.log* | *signatures.log* | *known\_services.log* |
| *intel.log* | *ftp.log* | *pe.log* | *software.log* |
| *loaded\_scripts.log* | *ssh.log* | *traceroute.log* | *weird.log* |

You can categorise the logs before starting an investigation. Thus, finding the evidence/anomaly you are looking for will be easier. The given table is a brief example of using multiple log files. You can create your working model or customise the given one. Make sure you read each log description and understand the purpose to know what to expect from the corresponding log file. Note thatthese are not the only ones to focus on. Investigated logs are highly associated with the investigation case type and hypothesis, so do not just rely only on the logs given in the example table!

The table shows us how to use multiple logs to identify anomalies and run an investigation by correlating across the available logs.

* **Overall Info:**The aim is to review the overall connections, shared files, loaded scripts and indicators at once. This is the first step of the investigation.
* **Protocol Based:**Once you review the overall traffic and find suspicious indicators or want to conduct a more in-depth investigation, you focus on a specific protocol.
* **Detection:**Use the prebuild or custom scripts and signature outcomes to support your findings by having additional indicators or linked actions.
* **Observation:**The summary of the hosts, services, software, and unexpected activity statistics will help you discover possible missing points and conclude the investigation.

|  |  |
| --- | --- |
| **Tool/Auxilary Name** | **Purpose** |
| **Zeek-cut** | Cut specific columns from zeek logs. |

**root@ubuntu$ cat conn.log | zeek-cut uid proto id.orig\_h id.orig\_p id.resp\_h id.resp\_p CTMFXm1AcIsSnq2Ric udp 192.168.121.2 51153 192.168.120.22 53 CLsSsA3HLB2N6uJwW udp 192.168.121.10 50080 192.168.120.10 514**

Qus: Investigate the **sample.pcap** file. Investigate the **dhcp.log** file. What is the available hostname?

Ans: cat dhcp.log

Qus: Investigate the **dns.log** file. What is the number of unique DNS queries?

Ans: cat dns.log | zeek-cut query

Qus: Investigate the **conn.log** file. What is the longest connection duration?

Ans: *cat conn.log | zeek-cut id.orig\_h id.resq\_h duration | sort -rn*

|  |  |  |  |
| --- | --- | --- | --- |
| **Category** | **Command Purpose and Usage** | **Category** | **Command Purpose and Usage** |
| **Basics** | View the command history: **ubuntu@ubuntu$ history**  Execute the 10th command in history: **ubuntu@ubuntu$ !10**  Execute the previous command: **ubuntu@ubuntu$ !!** | **Read File** | Read sample.txt file: **ubuntu@ubuntu$ cat sample.txt**  Read the first 10 lines of the file: **ubuntu@ubuntu$ head sample.txt**  Read the last 10 lines of the file: **ubuntu@ubuntu$ tail sample.txt** |
| **Find & Filter** | Cut the 1st field: **ubuntu@ubuntu$ cat test.txt | cut -f 1**  Cut the 1st column: **ubuntu@ubuntu$ cat test.txt | cut -c1**  Filter specific keywords: **ubuntu@ubuntu$ cat test.txt | grep 'keywords'**  Sort outputs alphabetically: **ubuntu@ubuntu$ cat test.txt | sort**  Sort outputs numerically: **ubuntu@ubuntu$ cat test.txt | sort -n**  Eliminate duplicate lines: **ubuntu@ubuntu$ cat test.txt | uniq**  Count line numbers: **ubuntu@ubuntu$ cat test.txt | wc -l**  Show line numbers **ubuntu@ubuntu$ cat test.txt | nl** | **Advanced** | Print line 11: **ubuntu@ubuntu$ cat test.txt | sed -n '11p'**  Print lines between 10-15: **ubuntu@ubuntu$ cat test.txt | sed -n '10,15p'**  Print lines below 11: **ubuntu@ubuntu$ cat test.txt | awk 'NR < 11 {print $0}'**  Print line 11: **ubuntu@ubuntu$ cat test** |

|  |  |
| --- | --- |
| **Special** | Filter specific fields of Zeek logs:  **ubuntu@ubuntu$ cat signatures.log | zeek-cut uid src\_addr dst\_addr** |

|  |  |
| --- | --- |
| **Use Case** | **Description** |
| **sort | uniq** | Remove duplicate values. |
| **sort | uniq -c** | Remove duplicates and count the number of occurrences for each  value. |
| **sort -nr** | Sort values numerically and recursively. |
| **rev** | Reverse string characters. |
| **cut -f 1** | Cut field 1. |
| **cut -d '.' -f 1-2** | Split the string on every dot and print keep the first two fields. |
| **grep -v 'test'** | Display lines that  don't match the "test" string. |
| **grep -v -e 'test1' -e 'test2'** | Display lines that don't match one or both "test1" and "test2" strings. |
| **file** | View file information. |
| **grep -rin Testvalue1 \* | column -t | less -S** | Search the "Testvalue1" string everywhere,  organise column spaces and view the output with less. |

**Zeek Signatures**

Zeek supports signatures to have rules and event correlations to find noteworthy activities on the network. Zeek signatures use low-level pattern matching and cover conditions similar to Snort rules. Unlike Snort rules, Zeek rules are not the primary event detection point. Zeek has a scripting language and can chain multiple events to find an event of interest. We focus on the signatures in this task, and then we will focus on Zeek scripting in the following tasks.

Zeek signatures are composed of three logical paths; signature id, conditions and action. The signature breakdown is shown in the table below;

|  |  |
| --- | --- |
| **Signature id** | **Unique** signature name. |
| **Conditions** | **Header:**Filtering the packet headers for specific source and destination addresses, protocol and port numbers.  **Content:**Filtering the packet payload for specific value/pattern. |
| **Action** | **Default action:**Create the "signatures.log" file in case of a signature match.  **Additional action:**Trigger a Zeek script. |

Now let's dig more into the Zeek signatures. The below table provides the most common conditions and filters for the Zeek signatures.

|  |  |
| --- | --- |
| **Condition Field** | **Available Filters** |
| **Header** | **src-ip:**Source IP.  **dst-ip:**Destination IP.  **src-port:**Source port.  **dst-port:**Destination port.  **ip-proto:** Target protocol. Supported protocols; TCP, UDP, ICMP, ICMP6, IP, IP6 |
| **Content** | **payload:** Packet payload. **http-request:**Decoded HTTP requests. **http-request-header:** Client-side HTTP headers. **http-request-body:** Client-side HTTP request bodys. **http-reply-header:** Server-side HTTP headers. **http-reply-body:** Server-side HTTP request bodys. **ftp:** Command line input of FTP sessions. |
| **Context** | **same-ip:** Filtering the source and destination addresses for duplication. |
| **Action** | **event:**Signature match message. |
| **Comparison Operators** | **==**, **!=**, **<**, **<=**, **>**, **>=** |
| **NOTE!** | Filters accept string, numeric and regex values. |

**ubuntu@ubuntu$ zeek -C -r sample.pcap -s sample.sig**

|  |
| --- |
| Zeek signatures use the ".sig" extension. |
| **-C:** Ignore checksum errors.  **-r:** Read pcap file.  **-s:** Use signature file. |

Remember, Zeek signatures support regex. Regex ".\*" matches any character zero or more times. The rule will match when a "password" phrase is detected in the packet payload. Once the match occurs, Zeek will generate an alert and create additional log files (signatures.log and notice.log).

**ubuntu@ubuntu$ zeek -C -r http.pcap -s http-password.sig ubuntu@ubuntu$ ls clear-logs.sh conn.log files.log http-password.sig http.log http.pcap notice.log packet\_filter.log signatures.log ubuntu@ubuntu$ cat notice.log | zeek-cut id.orig\_h id.resp\_h msg 10.10.57.178 44.228.249.3 10.10.57.178: Cleartext Password Found! 10.10.57.178 44.228.249.3 10.10.57.178: Cleartext Password Found! ubuntu@ubuntu$ cat signatures.log | zeek-cut src\_addr dest\_addr sig\_id event\_msg 10.10.57.178 http-password 10.10.57.178: Cleartext Password Found! 10.10.57.178 http-password 10.10.57.178: Cleartext Password Found!**

As shown in the above terminal output, the signatures.log and notice.log provide basic details and the signature message. Both of the logs also have the application banner field. So it is possible to know where the signature match occurs. Let's look at the application banner!

**ubuntu@ubuntu$ cat signatures.log | zeek-cut sub\_msg POST /userinfo.php HTTP/1.1\x0d\x0aHost: testphp.vulnweb.com\x0d\x0aUser-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86\_64; rv:98.0) Gecko/20100101 Firefox/... ubuntu@ubuntu$ cat notice.log | zeek-cut sub POST /userinfo.php HTTP/1.1\x0d\x0aHost: testphp.vulnweb.com\x0d\x0aUser-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86\_64; rv:98.0) Gecko/20100101 Firefox/...**

**Example | FTP Brute-force**

Let's create another rule to filter FTP traffic. This time, we will use the FTP content filter to investigate command-line inputs of the FTP traffic. The aim is to detect FTP "admin" login attempts. This basic signature will help us identify the admin login attempts and have an idea of possible admin account abuse or compromise events.

**ubuntu@ubuntu$ zeek -C -r ftp.pcap -s ftp-admin.sig ubuntu@ubuntu$ cat signatures.log | zeek-cut src\_addr dst\_addr event\_msg sub\_msg | sort -r| uniq 10.234.125.254 10.121.70.151 10.234.125.254: FTP Admin Login Attempt! USER administrator 10.234.125.254 10.121.70.151 10.234.125.254: FTP Admin Login Attempt! USER admin**

Our rule shows us that there are multiple logging attempts with account names containing the "admin" phrase. The output gives us great information to notice if there is a brute-force attempt for an admin account.

This signature can be considered a case signature. While it is accurate and works fine, we need global signatures to detect the "known threats/anomalies". We will need those case-based signatures for significant and sophistical anomalies like zero-days and insider attacks in the real-life environment. Having individual rules for each case will create dozens of logs and alerts and cause missing the real anomaly. The critical point is logging logically, not logging everything.

We can improve our signature by not limiting the focus only to an admin account. In that case, we need to know how the FTP protocol works and the default response codes. If you don't know these details, please refer to [RFC documentation](https://datatracker.ietf.org/doc/html/rfc765).

**Let's optimise our rule and make it detect all possible FTP brute-force attempts.**

This signature will create logs for each event containing "FTP 530 response", which allows us to track the login failure events regardless of username.

**signature ftp-username {**

**ip-proto == tcp**

**ftp /.*\*USER.\**/**

**event "FTP Username Input Found!"**

**}**

**signature ftp-brute { ip-proto == tcp payload /.*\*530.\**Login.*\*incorrect.\**/**

**event "FTP Brute-force Attempt!" }**

Let's merge both of the signatures in a single file. We will have two different signatures, and they will generate alerts according to match status. The result will show us how we benefit from this action. Again, we will need the "CLI Kung-Fu" skills to extract the event of interest.

This rule should show us two types of alerts and help us to correlate the events by having "FTP Username Input" and "FTP Brute-force Attempt" event messages. Let's investigate the logs. We're grepping the logs in range 1001-1004 to demonstrate that the first rule matches two different accounts (admin and administrator).

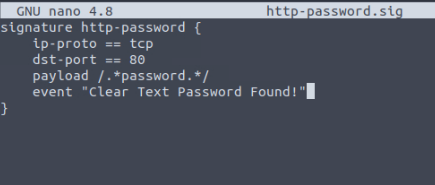
**ubuntu@ubuntu$ zeek -C -r ftp.pcap -s ftp-admin.sig ubuntu@ubuntu$ cat notice.log | zeek-cut uid id.orig\_h id.resp\_h msg sub | sort -r| nl | uniq | sed -n '1001,1004p' 1001 CeMYiaHA6AkfhSnd 10.234.125.254 10.121.70.151 10.234.125.254: FTP Username Input Found! USER admin 1002 CeMYiaHA6AkfhSnd 10.234.125.254 10.121.70.151 10.121.70.151: FTP Brute-force Attempt! 530 Login incorrect. 1003 CeDTDZ2erDNF5w7dyf 10.234.125.254 10.121.70.151 10.234.125.254: FTP Username Input Found! USER administrator 1004 CeDTDZ2erDNF5w7dyf 10.234.125.254 10.121.70.151 10.121.70.151: FTP Brute-force Attempt! 530 Login incorrect.**

*Investigate the http.pcap file. Create the HTTP signature shown in the task and investigate the pcap. What is the source IP of the first event?*

*cd Desktop/Exercise-Files/TASK-5/http*

*nano http-password.sig*

HTTP signature:



Command: Save and exit

CRTL+S  
CRTL+x

Run zeek command:

*zeek -C -r http.pcap -s http-password.sig*

*Investigate the http.pcap file. Create the HTTP signature shown in the task and investigate the pcap. What is the source IP of the first event?*

*cat signatures.log | zeek-cut src\_addr*

10.10.57.178

*What is the source port of the second event?*

*cat signatures.log | zeek-cut src\_port*

38712

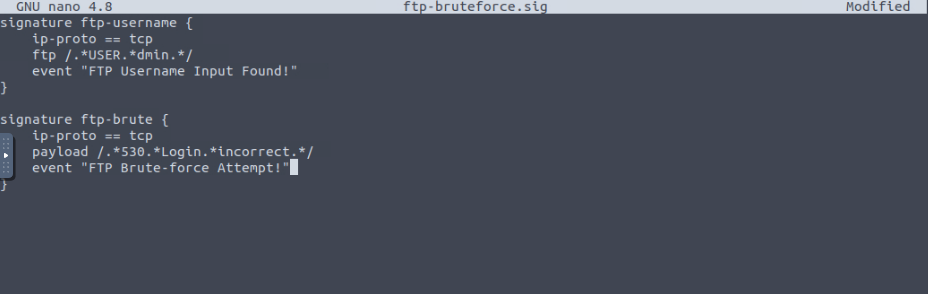
*Investigate the****conn.log****.  
What is the total number of the sent and received packets from source port 38706?*

*cat conn.log | zeek-cut id.orig\_p id.resp\_h id.resp\_p proto service orig\_pkts orig\_ip\_bytes resp\_pkts*

20

*Create the global rule shown in the task and investigate the ftp.pcap file.Investigate the****notice.log****. What is the number of unique events?*

Rule: ftp-bruteforce.sig:



*zeek -C -r ftp.pcap -s ftp-bruteforce.sig*

*I*

*Investigate the****notice.log****. What is the number of unique events?*

*cat notice.log | zeek-cut uid | sort | uniq | wc -l*

1413

*What is the number of****ftp-brute****signature matches?*

*cat signatures.log | grep “ftp-brute” | wc -l*

1410

**Tip**: to see the top of the file run:

**cat notice.log | head -10**

**Zeek Scripts**

Zeek has its own event-driven scripting language, which is as powerful as high-level languages and allows us to investigate and correlate the detected events. Since it is as capable as high-level programming languages, you will need to spend time on Zeek scripting language in order to become proficient. In this room, we will cover the basics of Zeek scripting to help you understand, modify and create basic scripts. Note that scripts can be used to apply a policy and in this case, they are called policy scripts.

Zeek has base scripts installed by default, and these are not intended to be modified.

These scripts are located in  
 **"/opt/zeek/share/zeek/base"**.

User-generated or modified scripts should be located in a specific path.

These scripts are located in  
**"/opt/zeek/share/zeek/site".**

Policy scripts are located in a specific path.

These scripts are located in  
 **"/opt/zeek/share/zeek/policy".**

Like Snort, to automatically load/use a script in live sniffing mode, you must identify the script in the Zeek configuration file. You can also use a script for a single run, just like the signatures.

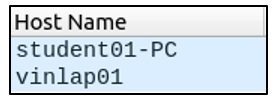
**The configuration file is located in "/opt/zeek/share/zeek/site/local.zeek".**

* Zeek scripts use the ".zeek" extension.
* Do not modify anything under the "zeek/base" directory. User-generated and modified scripts should be in the "zeek/site" directory.
* You can call scripts in live monitoring mode by loading them with the command **load @/script/path** or **load @script-name** in local.zeek file.
* Zeek is event-oriented, not packet-oriented! We need to use/write scripts to handle the event of interest.

ubuntu@ubuntu$ zeek -C -r sample.pcap -s sample.sig

**GUI vs Scripts**

Have you ever thought about automating tasks in Wireshark, tshark or tcpdump? Zeek provides that chance to us with its scripting power. Let's say we need to extract all available DHCP hostnames from a pcap file. In that case, we have several options like using tcpdump, Wireshark, tshark or Zeek.

Let's see Wireshark on the stage first. You can have the same information with Wireshark. However, while this information can be extracted using Wireshark is not easy to transfer the data to another tool for processing. Tcpdump and tshark are command-line tools, and it is easy to extract and transfer the data to another tool for processing and correlating.

**ubuntu@ubuntu$ sudo tcpdump -ntr smallFlows.pcap port 67 or port 68 -e -vv | grep 'Hostname Option' | awk -F: '{print $2}' | sort -nr | uniq | nl 1 "vinlap01" 2 "student01-PC" ubuntu@ubuntu$ tshark -V -r smallFlows.pcap -Y "udp.port==67 or udp.port==68" -T fields -e dhcp.option.hostname | nl | awk NF 1 student01-PC 2 vinlap01**

Now let's see Zeek scripts in action. First, let's look at the components of the Zeek script. Here the first, second and fourth lines are the predefined syntaxes of the scripting language. The only part we created is the third line which tells Zeek to extract DHCP hostnames. Now compare this automation ease with the rest of the methods. Obviously, this four-line script is easier to create and use. While tcpdump and tshark can provide similar results, transferring uncontrolled data through multiple pipelines is not much preferred.

**event dhcp\_message (c: connection, is\_orig: bool, msg: DHCP::Msg, options: DHCP::Options) { print options$host\_name; }**

**ubuntu@ubuntu$ zeek -C -r smallFlows.pcap dhcp-hostname.zeek student01-PC vinlap01**

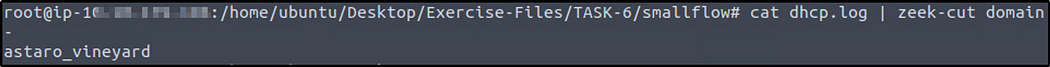
|  |
| --- |
| **Customized script locations** |
| /opt/zeek/share/zeek/base/bif  /opt/zeek/share/zeek/base/bif/plugins  /opt/zeek/share/zeek/base/protocols |

**Investigate the smallFlows.pcap file. Investigate the dhcp.log file. What is the domain value of the “vinlap01” host?**

**Ans: astaro\_vineyard**

zeek -C -r smallFlows.pcap  
cat dhcp.log  
cat dhcp.log | zeek-cut domain





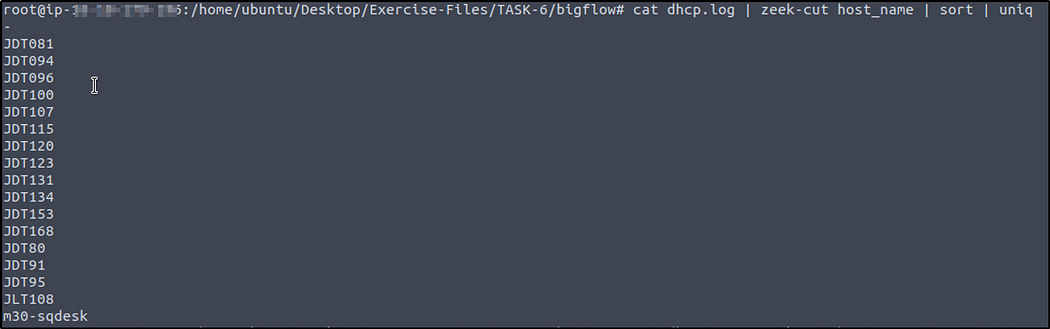
**Investigate the bigFlows.pcap file. Investigate the dhcp.log file. What is the number of identified unique hostnames?**

**Ans: 17**

zeek -C -r bigFlows.pcap  
cat dhcp.log | zeek-cut host\_name | sort | uniq  
cat dhcp.log | zeek-cut host\_name | sort | uniq | wc -l







When we include the command to count the lines, the result is 18. But in the image above the first line is empty so the correct answer is 17.

**Investigate the dhcp.log file. What is the identified domain value?**

**Ans: jaalam.net**

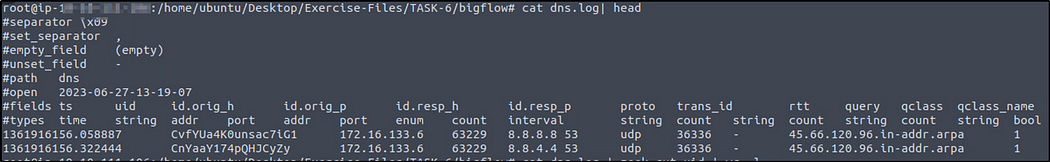
cat dhcp.log | zeek-cut domain | sort | uniq



**Investigate the dns.log file. What is the number of unique queries?**

**Ans: 1109**

cat dns.log| head



Recall from CLI Kung-fu the command grep -v -e 'test1' -e 'test2', which display lines that don’t match one or both “test1” and “test2” strings. The hint also provided us this, “grep -v -e ‘\*’ -e ‘-’ “.

The hint says that there are two values that we should not include in our result. But what if in other situations there could be more values?

So to determine what are the special characters from a file or logs, we can use this command. We will read from the “dns.log” for example.

cat dns.log | zeek-cut query | grep -oP "^[^\w\s]+$" | sort -u

This command includes a grep command that only output (-o) the matched special character and ‘-P’ option to enable Perl-compatible regular expresssions. The regex expression matches any individual special character. “sort -u” sorts the output in alphabetical order (sort) and ‘-u’ option ensures that duplicate characters are removed, so each special character appears only once in the output.



So we know now what special characters not to include in our output.

cat dns.log | zeek-cut query |grep -v -e '\*' -e '-' | sort | uniq| wc -l



**Scripts 101 | Write Basic Scripts**

Scripts contain operators, types, attributes, declarations and statements, and directives. Let's look at a simple example event called "zeek\_init" and "zeek\_done". These events work once the Zeek process starts and stops. Note that these events don't have parameters, and some events will require parameters.

**event zeek\_init() { print ("Started Zeek!"); } event zeek\_done() { print ("Stopped Zeek!"); }**

**ubuntu@ubuntu$ zeek -C -r sample.pcap 101.zeek Started Zeek! Stopped Zeek!**

Let's print the packet data to the terminal and see the raw data. In this script, we are requesting details of a connection and extracting them without any filtering or sorting of the data. To accomplish this, we are using the "new\_connection" event. This event is automatically generated for each new connection. This script provides bulk information on the terminal. We need to get familiar with Zeek's data structure to reduce the amount of information and focus on the event of interest. To do so, we need to investigate the bulk data.

**event new\_connection(c: connection) { print c; }**

**ubuntu@ubuntu$ zeek -C -r sample.pcap 102.zeek [id=[orig\_h=192.168.121.40, orig\_p=123/udp, resp\_h=212.227.54.68, resp\_p=123/udp], orig=[size=48, state=1, num\_pkts=0, num\_bytes\_ip=0, flow\_label=0, l2\_addr=00:16:47:df:e7:c1], resp=[size=0, state=0, num\_pkts=0, num\_bytes\_ip=0, flow\_label=0, l2\_addr=00:00:0c:9f:f0:79], start\_time=1488571365.706238, duration=0 secs, service={}, history=D, uid=CajwDY2vSUtLkztAc, tunnel=, vlan=121, inner\_vlan=, dpd=, dpd\_state=, removal\_hooks=, conn=, extract\_orig=F, extract\_resp=F, thresholds=, dce\_rpc=, dce\_rpc\_state=, dce\_rpc\_backing=, dhcp=, dnp3=, dns=, dns\_state=, ftp=, ftp\_data\_reuse=F, ssl=, http=, http\_state=, irc=, krb=, modbus=, mysql=, ntlm=, ntp=, radius=, rdp=, rfb=, sip=, sip\_state=, snmp=, smb\_state=, smtp=, smtp\_state=, socks=, ssh=, syslog=]**

The above terminal provides bulk data for each connection. This style is not the best usage, and in real life, we will need to filter the information for specific purposes. If you look closely at the output, you can see an ID and field value for each part.

To filter the event of interest, we will use the primary tag (in this case, it is c --comes from "c: connection"--), id value (id=), and field name. You should notice that the fields are the same as the fields in the log files.

**event new\_connection(c: connection) { print ("###########################################################"); print (""); print ("New Connection Found!"); print (""); print fmt ("Source Host: %s # %s --->", c$id$orig\_h, c$id$orig\_p); print fmt ("Destination Host: resp: %s # %s <---", c$id$resp\_h, c$id$resp\_p); print (""); } # %s: Identifies string output for the source. # c$id: Source reference field for the identifier**

**ubuntu@ubuntu$ zeek -C -r sample.pcap 103.zeek ########################################################### New Connection Found! Source Host: 192.168.121.2 # 58304/udp ---> Destination Host: resp: 192.168.120.22 # 53/udp <--- ###########################################################**

**Scripts 202 | Load Local Scripts**

**Load all local scripts**

We mentioned that Zeek has base scripts located in "/opt/zeek/share/zeek/base". You can load all local scripts identified in your "local.zeek" file. Note that base scripts cover multiple framework functionalities. You can load all base scripts by easily running the **local** command.

**ubuntu@ubuntu$ zeek -C -r ftp.pcap local ubuntu@ubuntu$ ls 101.zeek 103.zeek clear-logs.sh ftp.pcap packet\_filter.log stats.log 102.zeek capture\_loss.log conn.log loaded\_scripts.log sample.pcap weird.log**

The above output demonstrates how to run all base scripts using the "local" command. Look at the above terminal output; Zeek provided additional log files this time. Loaded scripts generated loaded\_scripts.log, capture\_loss.log, notice.log, stats.log files. Note that, in our instance, 465 scripts loaded and used by using the "local" command. However, Zeek doesn't provide log files for the scripts doesn't have hits or results.

**Load Specific Scripts**

Another way to load scripts is by identifying the script path. In that case, you have the opportunity of loading a specific script or framework. Let's go back to FTP brute-forcing case. We created a script that detects multiple admin login failures in previous steps. Zeek has an FTP brute-force detection script as well. Now let's use the default script and identify the differences.

**ubuntu@ubuntu$ zeek -C -r ftp.pcap /opt/zeek/share/zeek/policy/protocols/ftp/detect-bruteforcing.zeek ubuntu@ubuntu$ cat notice.log | zeek-cut ts note msg 1024380732.223481 FTP::Bruteforcing 10.234.125.254 had 20 failed logins on 1 FTP server in 0m1s**

The above output shows how to load a specific script. This script provides much more information than the one we created. It provides one single line output and a connection summary for the suspicious incident. You can find and read more on the prebuilt scripts and frameworks by visiting Zeek's online book [here](https://docs.zeek.org/en/master/frameworks/index.html).

**Investigate the sample.pcap file with 103.zeek script. Investigate the terminal output. What is the number of the detected new connections?**

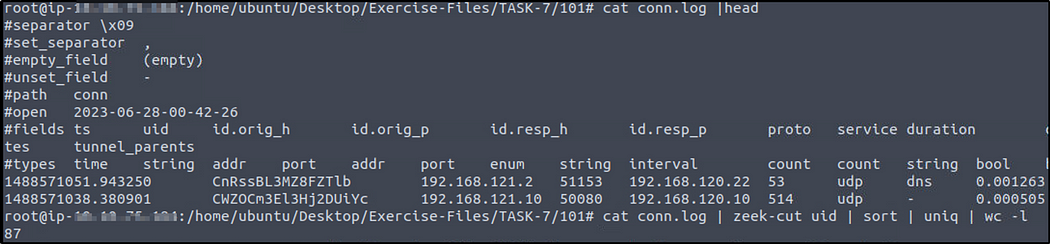
**Ans: 87**

let’s run the sample.pcap with the script.

zeek -C -r sample.pcap 103.zeek

We will “cat” the “conn.log” then select the “uid” field, sort the results, and pipe with “uniq” to avoid duplication, and then finally count the lines.

cat conn.log | zeek-cut uid | sort | uniq | wc -l



**Go to folder TASK-7/201.  
Investigate the ftp.pcap file with ftp-admin.sig signature and 201.zeek script. Investigate the signatures.log file. What is the number of signature hits?**

**Ans: 1401**

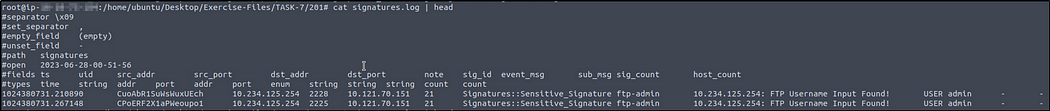
zeek -C -r ftp.pcap 201.zeek -s ftp-admin.sig | wc -l



**Investigate the signatures.log file. What is the total number of “administrator” username detections?**

**Ans: 731**

cat signatures.log | zeek-cut sub\_msg | grep "USER administrator" | wc -l





**Investigate the ftp.pcap file with all local scripts, and investigate the loaded\_scripts.log file. What is the total number of loaded scripts?**

**Ans: 498**

zeek -C -r ftp.pcap 201.zeek local



Don’t worry if you get a warning.

cat loaded\_scripts.log | zeek-cut name | wc -l

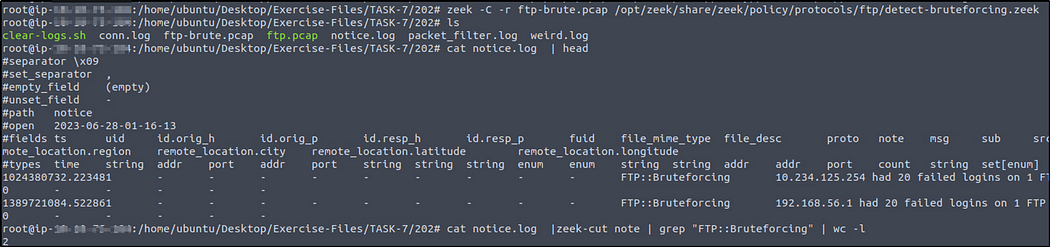


**Go to folder TASK-7/202.  
Investigate the ftp-brute.pcap file with “/opt/zeek/share/zeek/policy/protocols/ftp/detect-bruteforcing.zeek” script. Investigate the notice.log file. What is the total number of brute-force detections?**

**Ans: 2**

zeek -C -r ftp-brute.pcap /opt/zeek/share/zeek/policy/protocols/ftp/detect-bruteforcing.zeek

cat notice.log |zeek-cut note | grep "FTP::Bruteforcing" | wc -l



**Scripts 203 | Load Frameworks**

Zeek has 15+ frameworks that help analysts to discover the different events of interest. In this task, we will cover the common frameworks and functions. You can find and read more on the prebuilt scripts and frameworks by visiting Zeek's online book [here](https://docs.zeek.org/en/master/frameworks/index.html).

**File Framework | Hashes**

Not all framework functionalities are intended to be used in CLI mode. The majority of them are used in scripting. You can easily see the usage of frameworks in scripts by calling a specific framework as **load @ $PATH/base/frameworks/framework-name**. Now, let's use a prebuilt function of the file framework and have MD5, SHA1 and SHA256 hashes of the detected files. We will call the "File Analysis" framework's "hash-all-files" script to accomplish this. Before loading the scripts, let's look at how it works.

**ubuntu@ubuntu$ cat hash-demo.zeek # Enable MD5, SHA1 and SHA256 hashing for all files. @load /opt/zeek/share/zeek/policy/frameworks/files/hash-all-files.zeek**

The above output shows how frameworks are loaded. In earlier tasks, we mentioned that Zeek highly relies on scripts, and the frameworks depend on scripts. Let's have a closer look at the file hash framework and see the script behind it.

**ubuntu@ubuntu$ cat /opt/zeek/share/zeek/policy/frameworks/files/hash-all-files.zeek # Enable MD5, SHA1 and SHA256 hashing for all files. @load base/files/hash event file\_new(f: fa\_file) { Files::add\_analyzer(f, Files::ANALYZER\_MD5); Files::add\_analyzer(f, Files::ANALYZER\_SHA1); Files::add\_analyzer(f, Files::ANALYZER\_SHA256); }**

Now let's execute the script and investigate the log file.

**ubuntu@ubuntu$ zeek -C -r case1.pcap hash-demo.zeek ubuntu@ubuntu$ zeek -C -r case1.pcap /opt/zeek/share/zeek/policy/frameworks/files/hash-all-files.zeek ubuntu@ubuntu$ cat files.log | zeek-cut md5 sha1 sha256 cd5a4d3fdd5bffc16bf959ef75cf37bc 33bf88d5b82df3723d5863c7d23445e345828904 6137f8db2192e638e13610f75e73b9247c05f4706f0afd1fdb132d86de6b4012 b5243ec1df7d1d5304189e7db2744128 a66bd2557016377dfb95a87c21180e52b23d2e4e f808229aa516ba134889f81cd699b8d246d46d796b55e13bee87435889a054fb cc28e40b46237ab6d5282199ef78c464 0d5c820002cf93384016bd4a2628dcc5101211f4 749e161661290e8a2d190b1a66469744127bc25bf46e5d0c6f2e835f4b92db18**

Look at the above terminal outputs. Both of the scripts provided the same result. Here the preference is up to the user. Both of the usage formats are true. Prebuilt frameworks are commonly used in scriptings with the "@load" method. Specific scripts are used as practical scripts for particular use cases.

**File Framework | Extract Files**

The file framework can extract the files transferred. Let's see this feature in action!

**ubuntu@ubuntu$ zeek -C -r case1.pcap /opt/zeek/share/zeek/policy/frameworks/files/extract-all-files.zeek ubuntu@ubuntu$ ls 101.zeek 102.zeek 103.zeek case1.pcap clear-logs.sh conn.log dhcp.log dns.log extract\_files files.log ftp.pcap http.log packet\_filter.log pe.log**

We successfully extracted files from the pcap. A new folder called "extract\_files" is automatically created, and all detected files are located in it. First, we will list the contents of the folder, and then we will use the **file** command to determine the file type of the extracted files.

**ubuntu@ubuntu$ ls extract\_files | nl 1 extract-1561667874.743959-HTTP-Fpgan59p6uvNzLFja 2 extract-1561667889.703239-HTTP-FB5o2Hcauv7vpQ8y3 3 extract-1561667899.060086-HTTP-FOghls3WpIjKpvXaEl ubuntu@ubuntu$ cd extract\_files ubuntu@ubuntu$ file \*| nl 1 extract-1561667874.743959-HTTP-Fpgan59p6uvNzLFja: ASCII text, with no line terminators 2 extract-1561667889.703239-HTTP-FB5o2Hcauv7vpQ8y3: Composite Document File V2 Document, Little Endian, Os: Windows, Version 6.3, Code page: 1252, Template: Normal.dotm, Last Saved By: Administrator, Revision Number: 2, Name of Creating Application: Microsoft Office Word, Create Time/Date: Thu Jun 27 18:24:00 2019, Last Saved Time/Date: Thu Jun 27 18:24:00 2019, Number of Pages: 1, Number of Words: 0, Number of Characters: 1, Security: 0 3 extract-1561667899.060086-HTTP-FOghls3WpIjKpvXaEl: PE32 executable (GUI) Intel 80386, for MS Windows**

Zeek extracted three files. The "file" command shows us one .txt file, one .doc/.docx file and one .exe file. Zeek renames extracted files. The name format consists of four values that come from conn.log and files.log files; default "extract" keyword, timestamp value (ts), protocol (source), and connection id (conn\_uids). Let's look at the files.log to understand possible anomalies better and verify the findings. Look at the below output; files.log provides the same results with additional details. Let's focus on the .exe and correlate this finding by searching its connection id (conn\_uids).

The given terminal output shows us that there are three files extracted from the traffic capture. Let's look at the file.log and correlate the findings with the rest of the log files.

**ubuntu@ubuntu$ cat files.log | zeek-cut fuid conn\_uids tx\_hosts rx\_hosts mime\_type extracted | nl 1 Fpgan59p6uvNzLFja CaeNgL1QzYGxxZPwpk 23.63.254.163 10.6.27.102 text/plain extract-1561667874.743959-HTTP-Fpgan59p6uvNzLFja 2 FB5o2Hcauv7vpQ8y3 CCwdoX1SU0fF3BGBCe 107.180.50.162 10.6.27.102 application/msword extract-1561667889.703239-HTTP-FB5o2Hcauv7vpQ8y3 3 FOghls3WpIjKpvXaEl CZruIO2cqspVhLuAO9 107.180.50.162 10.6.27.102 application/x-dosexec extract-1561667899.060086-HTTP-FOghls3WpIjKpvXaEl ubuntu@ubuntu$ grep -rin CZruIO2cqspVhLuAO9 \* | column -t | nl | less -S #NOTE: The full output is not shown here!. Redo the same actions in the attached VM! 1 conn.log:43:1561667898.852600 CZruIO2cqspVhLuAO9 10.6.27.102 49162 107.180.50.162 80 tcp http 2 files.log:11:1561667899.060086 FOghls3WpIjKpvXaEl 107.180.50.162 10.6.27.102 CZruIO2cqspVhLuAO9 HTTP 0 EXTRACT,PE 3 http.log:11:1561667898.911759 CZruIO2cqspVhLuAO9 10.6.27.102 49162 107.180.50.162 80 1 GET**

The "grep" tool helps us investigate the particular value across all available logs. The above terminal output shows us that the connection id linked with .exe appears in conn.log, files.log, and http.log files. Given example demonstrates how to filter some fields and correlate the findings with the rest of the logs. We've listed the source and destination addresses, file and connection id numbers, MIME types, and file names. Up to now, provided outputs and findings show us that record number three is a .exe file, and other log files provide additional information.

**Notice Framework | Intelligence**

The intelligence framework can work with data feeds to process and correlate events and identify anomalies. The intelligence framework requires a feed to match and create alerts from the network traffic. Let's demonstrate a single user-generated threat intel file and let Zeek use it as the primary intelligence source.

Intelligence source location: **/opt/zeek/intel/zeek\_intel.txt**

There are two critical points you should never forget. First, the source file has to be tab-delimited. Second, you can manually update the source and adding extra lines doesn't require any re-deployment. However, if you delete a line from the file, you will need to re-deploy the Zeek instance.

Let's add the suspicious URL gathered from the case1.pcap file as a source intel and see this feature in action! Before executing the script, let's look at the intelligence file and the script contents.

**ubuntu@ubuntu$ cat /opt/zeek/intel/zeek\_intel.txt #fields indicator indicator\_type meta.source meta.desc smart-fax.com Intel::DOMAIN zeek-intel-test Zeek-Intelligence-Framework-Test ubuntu@ubuntu$ cat intelligence-demo.zeek # Load intelligence framework! @load policy/frameworks/intel/seen @load policy/frameworks/intel/do\_notice redef Intel::read\_files += { "/opt/zeek/intel/zeek\_intel.txt" };**

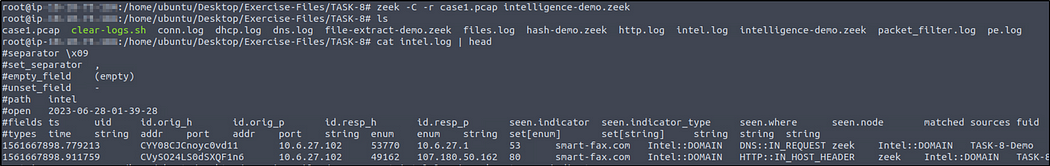
The above output shows the contents of the intel file and script contents. There is one intelligence input, and it is focused on a domain name, so when this domain name appears in the network traffic, Zeek will create the "intel.log" file and provide the available details.

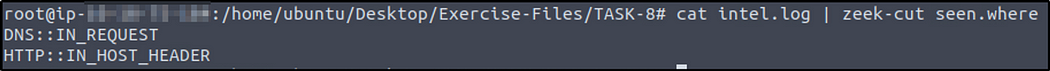
**ubuntu@ubuntu$ zeek -C -r case1.pcap intelligence-demo.zeek ubuntu@ubuntu$ cat intel.log | zeek-cut uid id.orig\_h id.resp\_h seen.indicator matched CZ1jLe2nHENdGQX377 10.6.27.102 10.6.27.1 smart-fax.com Intel::DOMAIN C044Ot1OxBt8qCk7f2 10.6.27.102 107.180.50.162 smart-fax.com Intel::DOMAIN**

**Investigate the case1.pcap file with intelligence-demo.zeek script. Investigate the intel.log file. Look at the second finding, where was the intel info found?**

**Ans: IN\_HOST\_HEADER**

zeek -C -r case1.pcap intelligence-demo.zeek  
cat intel.log | head  
cat intel.log | zeek-cut seen.where



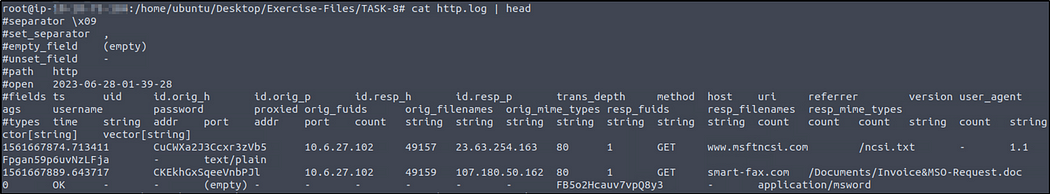


**Investigate the http.log file. What is the name of the downloaded .exe file?**

**Ans: knr.exe**

cat intel.log | head  
cat http.log | zeek-cut uri | grep '\.exe$'

“grep ‘\.exe$’” searches for lines that contain the “.exe” extension at the end of the line. The backslash (\) before the dot (.) is used to escape it, so that it matches a literal dot. The “$” matches the end of the line.

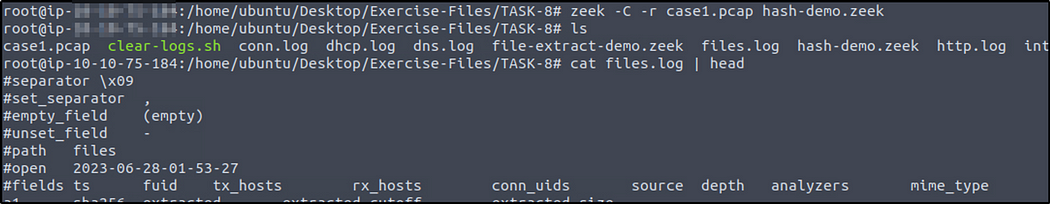




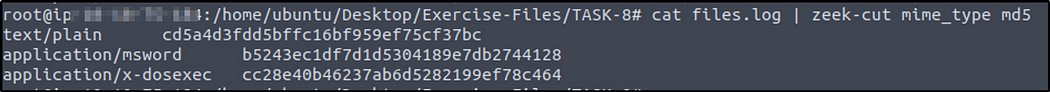
**Investigate the case1.pcap file with hash-demo.zeek script. Investigate the files.log file. What is the MD5 hash of the downloaded .exe file?**

**Ans: cc28e40b46237ab6d5282199ef78c464**

zeek -C -r case1.pcap hash-demo.zeek  
cat files.log | head  
cat files.log | zeek-cut mime\_type md5



We know that it is an executable file so it should be the third md5 value.



We can also find the correlation of the “.exe” file with the other log files.

First we need a common value of the “.exe” file.

cat files.log | zeek-cut fuid conn\_uids tx\_hosts rx\_hosts mime\_type extracted | nl



We will choose the value of the field “conn\_uids”.

So we got the “conn\_uids” value of the “.exe” file. Now we will extract all values in the current directory that correlates to the “.exe” file.

grep -rin CVsnuagu2ZhLnXy91 \* | column -t | nl | less -S

* grep -rin CVsnuagu2ZhLnXy91 \*: This searches for the pattern "CVsnuagu2ZhLnXy91" recursively (-r) in all files (\*) within the current directory. The -i option is used for case-insensitive matching, and the -n option displays line numbers.
* column -t: This formats the output into multiple columns for better readability. It assumes tabular data with whitespace as the delimiter.
* nl: This adds line numbers to the output.
* less -S: This command opens the output in the less pager, which allows scrolling through the content. The -S option disables line wrapping for better readability.

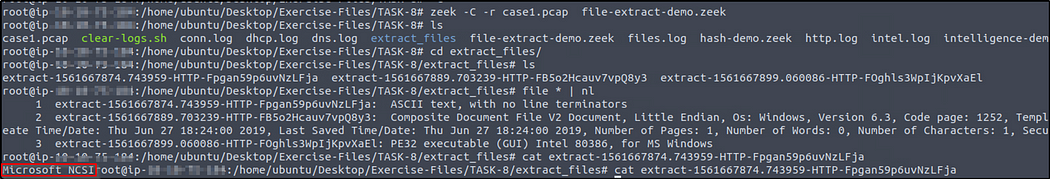
From the result, the “.exe” file is found in three logs. We see other information that relates to the file, and in “files.log” we see its MD5 value.



**Investigate the case1.pcap file with file-extract-demo.zeek script. Investigate the “extract\_files” folder. Review the contents of the text file. What is written in the file?**

**Ans: Microsoft NCSI**

zeek -C -r case1.pcap file-extract-demo.zeek  
file \* | nl  
cat extract-1561667874.743959-HTTP-Fpgan59p6uvNzLFja



**Scripts 204 | Package Manager**

Zeek Package Manager helps users install third-party scripts and plugins to extend Zeek functionalities with ease. The package manager is installed with Zeek and available with the **zkg** command. Users can install, load, remove, update and create packages with the "zkg" tool. You can read more on and view available packages [here](https://packages.zeek.org/) and [here](https://github.com/zeek/packages). Please note that you need root privileges to use the "zkg" tool.

**Basic usage of zkg;**

|  |  |
| --- | --- |
| **Command** | **Description** |
| **zkg install package\_path** | Install a package. Example (zkg install zeek/j-gras/zeek-af\_packet-plugin). |
| **zkg install git\_url** | Install package. Example (zkg install https://github.com/corelight/ztest). |
| **zkg list** | List installed package. |
| **zkg remove** | Remove installed package. |
| **zkg refresh** | Check version updates for installed packages. |
| **zkg upgrade** | Update installed packages. |

There are multiple ways of using packages. The first approach is using them as frameworks and calling specific package path/directory per usage. The second and most common approach is calling packages from a script with the "@load" method. The third and final approach to using packages is calling their package names; note that this method works only for packages installed with the "zkg" install method.

**Packages | Cleartext Submission of Password**

Let's install a package first and then demonstrate the usage in different approaches.   
**Note:**The package is installed in the given VM.

**ubuntu@ubuntu$ zkg install zeek/cybera/zeek-sniffpass The following packages will be INSTALLED: zeek/cybera/zeek-sniffpass (master) Proceed? [Y/n] Y Installing "zeek/cybera/zeek-sniffpass" Installed "zeek/cybera/zeek-sniffpass" (master) Loaded "zeek/cybera/zeek-sniffpass" ubuntu@ubuntu$ zkg list zeek/cybera/zeek-sniffpass (installed: master) - Sniffpass will alert on cleartext passwords discovered in HTTP POST requests**

The above output shows how to install and list the installed packages. Now we successfully installed a package. As the description mentions on the above terminal, this package creates alerts for cleartext passwords found in HTTP traffic. Let's use this package in three different ways!

**### Calling with script ubuntu@ubuntu$ zeek -Cr http.pcap sniff-demo.zeek ### View script contents ubuntu@ubuntu$ cat sniff-demo.zeek @load /opt/zeek/share/zeek/site/zeek-sniffpass ### Calling from path ubuntu@ubuntu$ zeek -Cr http.pcap /opt/zeek/share/zeek/site/zeek-sniffpass ### Calling with package name ubuntu@ubuntu$ zeek -Cr http.pcap zeek-sniffpass**

The above output demonstrates how to execute/load packages against a pcap. You can use the best one for your case. The "zeek-sniffpass" package provides additional information in the notice.log file. Now let's review the logs and discover the obtained data using the specific package.

**ubuntu@ubuntu$ cat notice.log | zeek-cut id.orig\_h id.resp\_h proto note msg 10.10.57.178 44.228.249.3 tcp SNIFFPASS::HTTP\_POST\_Password\_Seen Password found for user BroZeek 10.10.57.178 44.228.249.3 tcp SNIFFPASS::HTTP\_POST\_Password\_Seen Password found for user ZeekBro**

The above output shows that the package found cleartext password submissions, provided notice, and grabbed the usernames. Remember, in **TASK-5** we created a signature to do the same action. Now we can do the same activity without using a signature file. This is a simple demonstration of the benefit and flexibility of the Zeek scripts.

**Packages | Geolocation Data**

Let's use another helpful package called "geoip-conn". This package provides geolocation information for the IP addresses in the conn.log file. It depends on "GeoLite2-City.mmdb" database created by MaxMind. This package provides location information for only matched IP addresses from the internal database.

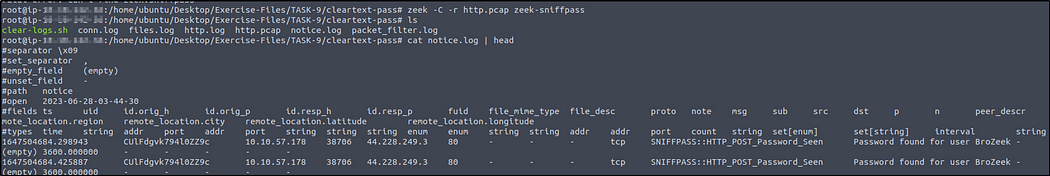
**ubuntu@ubuntu$ zeek -Cr case1.pcap geoip-conn ubuntu@ubuntu$ cat conn.log | zeek-cut uid id.orig\_h id.resp\_h geo.orig.country\_code geo.orig.region geo.orig.city geo.orig.latitude geo.orig.longitude geo.resp.country\_code geo.resp.region geo.resp.city Cbk46G2zXi2i73FOU6 10.6.27.102 23.63.254.163 - - - - - US CA Los Angeles**

Up to now, we've covered what the Zeek packages are and how to use them. There are much more packages and scripts available for Zeek in the wild. You can try ready or third party packages and scripts or learn Zeek scripting language and create new ones.

**Investigate the http.pcap file with the zeek-sniffpass module. Investigate the notice.log file. Which username has more module hits?**

**Ans: BroZeek**

zeek -C -r http.pcap zeek-sniffpass  
cat notice.log | head  
cat notice.log | zeek-cut msg | uniq -c   
# "-c"count the number of occurences for each unique value





**Investigate the case2.pcap file with geoip-conn module. Investigate the conn.log file. What is the name of the identified City?**

**Ans: Chicago**

zeek -C -r case2.pcap geoip-conn  
cat conn.log |head  
cat conn.log |zeek-cut id.resp\_h geo.resp.city | grep -v -e "-" | uniq -c





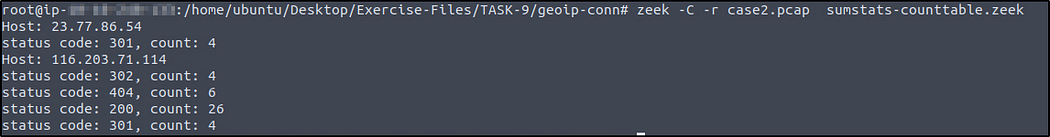
**Which IP address is associated with the identified City?**

**Ans: 23.77.86.54**

**Investigate the case2.pcap file with sumstats-counttable.zeek script. How many types of status codes are there in the given traffic capture?**

**Ans: 4**

zeek -C -r case2.pcap sumstats-counttable.zeek



Here is a modified command.

zeek -C -r case2.pcap sumstats-counttable.zeek | awk '{print $3}' | grep -v -e '^$' | sort | uniq | wc -l

In this command, the -v option inverts the matching logic, causing grep to exclude lines that match the specified pattern. The pattern ^$ matches empty lines because ^ represents the start of a line, and $ represents the end of a line. Thus, ^$ matches lines that contain no characters between the start and end.

