The bash script structure you've outlined is a good starting point. It's important to remember that the functions in the script are placeholders and you'll need to fill in the details based on your specific requirements.

Here are some key points to consider while developing your script:

- **Appropriate Formatting and Layout**: Your script should be well-structured and easy to read. Use comments to explain what each part of the script does.

- **Clear Definition**: The script should clearly define its authorship, purpose, inputs, and outputs. This information can be included in the comments at the top of the script.

- **Validation and Error Checking**: Your script should include error checking to handle any issues that might arise during its execution. This could involve checking the exit status of commands, validating input, etc.

- **Feedback and Reporting**: The script should provide feedback about its activities as it runs. This could be done through echo statements or by writing to a log file.

- **Output Formatting**: All output must be plain text and formatted suitably to be passed as direct input to the mail command. Avoid using fancy colors or terminal scripted text as this can cause problems with automated parsing.

- **User/Installation Guide**: Along with the script, provide a simple but complete user/installation guide. This guide should be sufficient for a junior administrator to follow and successfully install and configure the script.

Remember, the script will be executed in a virtual environment and the output validated against a known set of changes made on the examination system. Therefore, it's crucial to test your script thoroughly in a similar environment before submission.

For the setup/configuration instructions and an overview of the logic and flow of the script, you should create a separate PDF document. This document should include:

1. Instructions on how to set up and configure the script on a system.
2. Appropriate entries relating to system configuration to ensure the script runs in an automated manner at a specified time.
3. An overview of the logic and flow of the script, explaining how each function works and how they interact with each other.

these instructions should be brief but complete, and sufficient for a junior administrator to follow and successfully install and configure the script. The overview of the script should help users understand how the script works and how to troubleshoot any issues that may arise. Good luck with your project!

**check\_iocs():** This function reads the IoC file line by line, ignores comments, and checks if each line starts with “IOC” or “STR”. If the line starts with “IOC”, it gets the hash and directory and checks if the hash appears in the specified directory. If the line starts with “STR”, it gets the string and directory and checks if the string occurs in files in the specified directory. You’ll need to fill in the code for these checks based on your specific requirements.

For the validation tools, since they are just statically linked versions of sha256sum and grep respectively, you can use these commands in your script to validate the integrity of the IoC file and check for IoCs. Remember to access them via the full path as they will not be in the system search for commands defined by $PATH in the shell. Good luck with your project

Having test data and sample input available before the submission date will indeed be helpful. It’s important to use this data to validate your implementation and ensure it works as expected.

Remember, the test data provided is only indicative of the types of data to be provided, and the actual assessment will use different data conforming to the same formats. Therefore, it’s crucial to write your script in a way that it can handle a variety of inputs and not just the sample data.

Also, it’s worth noting that scripts which merely emulate the output of the sample data will be considered incorrect. Your script should be designed to process any data that conforms to the specified format, not just the sample data.

It's crucial to follow the submission guide accurately to ensure your work is evaluated correctly. Here's a summary of the key points:

1. Package your submission as a single .tgz (tar-gzip) file.

2. Include the following files in the root of the .tgz file:

- Your Hunting Script, named `tth-xxxx.sh`.

- A detached pgp/gpg signature file for the script, named `tth-xxxx.sh.sig`.

- A PDF file containing installation guides/instructions, named `xxxx.pdf`.

3. Replace `xxxx` with the portion of your email address before `@stud.noroff.no`.

4. Name your .tgz file as `CND-XXXX-HUNTER.tgz`.

5. Test your work in a clean environment before submission.

6. Ensure your scripts can execute and take parameters as specified.

7. Retain a complete copy of your work until the grading process is completed.

Remember, incorrectly named files or scripts that fail to execute will not be evaluated further. All submissions will be tested for AI and similarity. If further clarification of your script is needed, you may be required to attend an oral interview during the marking period. Good luck with your submission!

The marking and assessment process for your assignment is quite comprehensive. Here’s a summary:

1. **Submission Processing**: The submitted .tgz files are downloaded from Moodle and processed based on a class list of eligible students.
2. **File Extraction and Verification**: The .tgz file is copied to a clean instance of the test server, extracted, and the signatures are verified for the script based on your provided keys. If the verification is successful, testing will proceed.
3. **Script Execution**: The script is executed on the test environment with the provided parameters.
4. **Environment Configurations**: The script is run against several environment configurations to check error handling and cases as described in the specifications. The output of the script to STDOUT is captured. Commands called on a remote server are also captured along with output.
5. **Output Recording**: The recorded output is named appropriately for each case and student, and copied to a central system.
6. **Output Review**: The review will compare the provided output against the expected output/actions for the test case(s). This will be done using a combination of programmatic and manual inspection.
7. **Manual Review and Grading**: On successful completion of testing, output reports and the submitted PDF are manually reviewed and graded.
8. **Penalty for Damaging Test Environment**: Any attempt to damage or adversely impact the test execution environment will result in a zero mark.

Remember, this assignment is designed to assess the depth of your work, not just the surface level. So, it’s important to pay attention to detail and ensure your work is thorough and complete. Good luck!

Remember, satisfying the requirements accounts for a maximum of 75% of the assessment grade. The remaining 25% is based on the appropriate formatting of your output, error checking, and readability of your code and the accompanying PDF. Good luck with your assignment!

Sure, here's a basic example of how you might implement the `` function in bash. This function uses `wget` to download the IoC file from the URL specified on the command line:

**download\_ioc:** In this function, `ioc\_url` is the first command line parameter passed to the script. The function checks if the URL starts with `https`, downloads the IoC file using `wget`, and checks if the download was successful. If any of these steps fail, the function prints an error message and exits the script.

You would call this function in your main script execution like this:

```bash

# Main script execution

download\_ioc "$1"

```

Here, `"$1"` is the first command line parameter passed to the script, which should be the URL of the IoC file. Remember to replace `"$IOC\_FILE"` with the actual path where you want to download the IoC file.

Please note that this is a basic implementation and you might need to modify it based on your specific requirements and environment. For example, you might need to add more error checking or handle additional edge cases. Also, remember to test your script thoroughly to ensure it works as expected. Good luck with your project! If you have any more questions or need further clarification, feel free to ask.

Sure, here's a basic example of how you might implement the `validate\_ioc` function in bash. This function uses `sha256sum` to validate the integrity of the IoC file:

**validate\_ioc** : In this function, `expected\_hash` is the expected SHA-256 hash of the IoC file. The function calculates the actual SHA-256 hash of the IoC file using `sha256sum`, and checks if the actual hash matches the expected hash. If they don't match, the function prints an error message and exits the script.

You would call this function in your main script execution like this:

```bash

# Main script execution

download\_ioc "$1"

validate\_ioc "$2"

```

Here, `"$1"` is the first command line parameter passed to the script, which should be the URL of the IoC file, and `"$2"` is the second command line parameter passed to the script, which should be the expected SHA-256 hash of the IoC file.

Please note that this is a basic implementation and you might need to modify it based on your specific requirements and environment. For example, you might need to add more error checking or handle additional edge cases. Also, remember to test your script thoroughly to ensure it works as expected. Good luck with your project! If you have any more questions or need further clarification, feel free to ask.

Sure, here's a basic example of how you might implement the `check\_iocs` function in bash. This function processes the IoC file and checks for IoCs in the specified directories:

**check\_iocs :** In this function, the IoC file is read line by line, comments are ignored, and each line is checked to see if it starts with "IOC" or "STR". If the line starts with "IOC", the function gets the hash and directory, checks if the hash appears in the specified directory using `find` and `sha256sum`, and prints a warning message if the hash is found. If the line starts with "STR", the function gets the string and directory, checks if the string occurs in files in the specified directory using `grep`, and prints a warning message if the string is found.

Please note that this is a basic implementation and you might need to modify it based on your specific requirements and environment. For example, you might need to add more error checking or handle additional edge cases. Also, remember to test your script thoroughly to ensure it works as expected. Good luck with your project! If you have any more questions or need further clarification, feel free to ask.

Sure, here's a basic example of how you might implement the `collect\_sys\_info` function in bash. This function collects some information about the local system:

**collect\_sys\_info :** In this function, `listening\_ports` gets the currently listening ports using `netstat`, `firewall\_rules` gets the current firewall rules using `iptables`, `new\_files` finds files in `/var/www` (and subdirectories) that have been created in the last 48 hours using `find`, and `suid\_gid\_files` finds any SUID/GID files in the same path regardless of modification time using `find`.

You'll need to fill in the code for validating that all files installed in `/sbin`, `bin`, `/usr/sbin`, `/usr/bin` and `/usr/lib` match the valid hashes in the system package database, and ensuring that file systems mounted on `/var/www/images` and `/var/www/uploads` are set as non-executable.

Please note that this is a basic implementation and you might need to modify it based on your specific requirements and environment. For example, you might need to add more error checking or handle additional edge cases. Also, remember to test your script thoroughly to ensure it works as expected. Good luck with your project! If you have any more questions or need further clarification, feel free to ask.

[user](#message|)

can you write the code of the function generate\_report in a good way?

[assistant'](#message|)

Sure, here's a basic example of how you might implement the `generate\_report` function in bash. This function generates a report based on the information collected by the `collect\_sys\_info` function:

**generate\_report :** In this function, `collect\_sys\_info` is called to collect system information. The collected information is then written to the report file. Each piece of information is preceded by a heading, and the information itself is appended to the report file.

Please note that this is a basic implementation and you might need to modify it based on your specific requirements and environment. For example, you might need to add more error checking or handle additional edge cases. Also, remember to test your script thoroughly to ensure it works as expected. Good luck with your project! If you have any more questions or need further clarification, feel free to ask.

[user'](#message|)

can you write the code of the function upload\_outputs in a good way?

[assistant'](#message|)

Sure, here's a basic example of how you might implement the `upload\_outputs` function in bash. This function uses `rsync` to upload the outputs to a central server:

**upload\_outputs :** In this function, `remote\_server` is the remote server, `user\_identity` is the user identity, and `destination\_dir` is the destination directory on the remote server. The function uses `rsync` to upload the report file to the central server using the specified SSH identity file. If the upload fails, the function prints an error message and exits the script.

You would call this function in your main script execution like this:

# Main script execution

download\_ioc "$1"

validate\_ioc "$2"

check\_iocs

collect\_sys\_info

generate\_report

upload\_outputs "$3" "$4"

Here, `"$1"` is the first command line parameter passed to the script, which should be the URL of the IoC file, `"$2"` is the second command line parameter passed to the script, which should be the expected SHA-256 hash of the IoC file, `"$3"` is the third command line parameter passed to the script, which should be the remote server, and `"$4"` is the fourth command line parameter passed to the script, which should be the user identity.

Please note that this is a basic implementation and you might need to modify it based on your specific requirements and environment. For example, you might need to add more error checking or handle additional edge cases. Also, remember to test your script thoroughly to ensure it works as expected. Good luck with your project! If you have any more questions or need further clarification, feel free to ask.

[user'](#message|)

can you write the code of the function validate\_backup in a good way?

[assistant'](#message|)

Sure, here's a basic example of how you might implement the `validate\_backup` function in bash. This function uses `gpg` to validate the backup on the remote server:

**validate\_backup :** In this function, `remote\_server` is the remote server, `user\_identity` is the user identity, and `destination\_dir` is the destination directory on the remote server. The function uses `ssh` to connect to the remote server and `gpg` to validate the backup using the specified key ID. If the validation fails, the function prints an error message and exits the script.

You would call this function in your main script execution like this:

# Main script execution

download\_ioc "$1"

validate\_ioc "$2"

check\_iocs

collect\_sys\_info

generate\_report

upload\_outputs "$3" "$4"

validate\_backup "$3" "$4"

Here, `"$1"` is the first command line parameter passed to the script, which should be the URL of the IoC file, `"$2"` is the second command line parameter passed to the script, which should be the expected SHA-256 hash of the IoC file, `"$3"` is the third command line parameter passed to the script, which should be the remote server, and `"$4"` is the fourth command line parameter passed to the script, which should be the user identity.

Please note that this is a basic implementation and you might need to modify it based on your specific requirements and environment. For example, you might need to add more error checking or handle additional edge cases. Also, remember to test your script thoroughly to ensure it works as expected. Good luck with your project! If you have any more questions or need further clarification, feel free to ask.

The `clean\_up` function should restore the environment to the state it was in before the script was executed. This could involve deleting any temporary files that were created during the script's execution, closing any network connections that were opened, and releasing any resources that were allocated.

Here's a basic implementation of the `clean\_up` function:

# Function to clean up after the script

clean\_up() {

# Delete the IoC file

rm -f "$IOC\_FILE"

# Delete the log file

rm -f "$LOG\_FILE"

# Delete the report file

rm -f "$REPORT\_FILE"

}

This function will delete the IoC file, the log file, and the report file. Please modify this function as needed to suit your specific requirements.

Remember, this is just a basic implementation. Depending on what your script does, you might need to add more commands to this function to fully clean up after your script. For example, if your script opens a network connection, you might need to add a command to close that connection. If your script allocates any resources (like memory or file descriptors), you might need to add commands to release those resources.

I hope this helps! If you have any other questions or need further clarification, feel free to ask. 😊