Middle East Technical University

Department of Computer Engineering

CENG 435

Data Communications and Networking

Programming Assignment - Phase 2

Due Date: 29 December 2024 (Sunday) - 23:59

1 Introduction

In this assignment, you are going to implement covert channel communication in a container environment. You will use Python programming language with a specific version (Python 3.10.12). You do not have to create an environment for this assignment, instead, it was designed and shared with you as a docker environment as in the first phase. However, the docker environment and git repository have been updated considering the requirements of this phase. Therefore, you need to fork and clone the new version of the repository again for the second phase.

As you worked as a group in the first phase, you will continue with the same group in this phase. With this assignment, you will also gain essential information for various important computer engineering skills.

In this assignment, you are going to

- fork the new version of the same public repository as in the first phase to your own account and clone it to your local to work on it (only one of the team members should do it, the other member should be added as a collaborator),
- create two containers using the 'docker-compose.yaml' file which generates the images using the relevant Dockerfile,
- choose the covert channel type you want to implement in this assignment with the choice item in ODTUClass (each group must choose one unique covert channel type, and you can change your decision during the assignment),
- implement the chosen covert channel (sender and receiver) using the shared base covert channel class (with scapy) and following the restrictions,
- edit the 'config.json' file and set the parameters to run the covert channel sender and receiver,
- perform covert channel communication successfully,
- measure the covert channel capacity (bits per second),
- create a documentation using an automatic documentation system (Sphinx) as in the first phase,
- push your work to the forked repository,
- upload the required files to ODTUClass.

Important Note: In this assignment, you must edit and explain your work within the 'README.md' file in the app folder (or in the code folder for your local environment) instead of writing a report file. Additionally, you must share the link to your repository by editing the 'index.rst' in the docs folder.

2 What is a Covert Channel?

A **covert channel** is a method of communication that bypasses normal security controls and is used to transmit information in ways that were not originally intended or authorized. Unlike legitimate communication channels, covert channels exploit side effects or unused portions of a system, such as timing delays, hidden data fields, or unconventional use of protocols, to encode and transfer information between parties. These channels often operate secretly, making them difficult to detect and prevent, and are commonly studied in cybersecurity to understand how sensitive information might be leaked or exfiltrated without being noticed.

Covert channels can be analyzed in two groups, covert timing channels and covert storage channels:

- **Covert Timing Channels:** A covert timing channel manipulates the timing or ordering of events, such as inter-packet delays, or idle times between packet bursts to encode information and transfer it secretly. It depends on using time-based variations in the packet-sending behavior. These timing variations are well-designed and observable only by the intended receiver. It makes them challenging to detect or differentiate from legitimate traffic. In this assignment, two exploitation techniques for covert timing channels are mentioned:
  - **Packet Inter-Arrival Times:** In this technique, the sender encodes data in the time intervals between consecutive packets. For example, a short delay between packets may represent a '0', and a long delay may represent a '1'. The receiver measures these inter-arrival times and decodes the hidden information. This method is effective because timing variations are typically hard to monitor or control in real-time.
  - **Idle Period Between Packet Bursts:** This technique encodes information by varying the duration of inactivity or idle time between bursts of packets. For example, a short idle period might represent a '0', and a long idle period might represent a '1'. The sender controls the idle times between groups of packets, and the receiver decodes the data by analyzing these gaps. This approach is also hard to monitor and leverages periods of inactivity that cannot be recognized often in normal traffic analysis.
- **Covert Storage Channels:** A covert storage channel embeds information within some fields of a protocol or system, such as headers or metadata, to transmit data covertly. **Without changing the system's behavior**, it takes advantage of legitimate storage locations in the packets and transfers the secret data to the receiver.

  In this assignment three exploitation techniques of covert storage channels are mentioned:
  - **Packet Bursting:** In this technique, the sender encodes information by varying the number of packets sent in a burst. For example, sending five packets in a burst could represent a '1', while sending a single packet could represent a '0'. The receiver decodes the message by counting the number of packets in each burst. This method uses packet quantities as the information-hiding technique.
  - **Packet Size Variation:** The sender encodes data by manipulating the size of the packets. For instance, a small packet might represent a '0', while a larger packet might represent a '1'. The receiver observes and interprets the packet sizes to extract the hidden information. This technique is effective because packet sizes often vary naturally, making it difficult to distinguish covert data from legitimate traffic.
  - **Protocol Field Manipulation:** This technique involves embedding data in specific fields of a network protocol header, such as the IP identification field, TCP sequence number, or unused flag bits. The sender modifies these fields to encode information, and the receiver extracts the data by analyzing the manipulated fields. This approach leverages legitimate but often missed protocol features to transmit covert information.

In a covert channel communication, the actual transmitted data within the packet is not important and in this case, the receiver does not care about the actual packet data. Instead, it analyzes the transmission behavior of packets or specific fields in the received packets to get the actual transmitted message.

A covert channel sender and a receiver are fundamental requirements for covert channel communication. This is not a regular sender and receiver pair in a regular communication; instead, these are customized and well-designed to perform covert channel communication. The covert channel sender should transmit the message by using covert timing or storage channel techniques while the covert channel receiver should convert the transmitted packets to a meaningful message that was sent by the sender.

3 Setup & Preparation

3.1 Cloning GitHub Repository

First of all, you should open a GitHub account to fork the existing repository (covertovert) as in the first phase. Forking is the process of copying an existing repository to a new repository in your own account. To fork an existing repository, click the **fork** on the upper right in the 'covertovert' repository and follow the instructions.

Important Note: The repository has been edited for the second phase and you must fork the new version and create a new repository for this phase.

Afterward, you should open a new directory for the assignment on your local machine to work on it and clone the forked repository that is in your GitHub account. To clone the repository, you should click the **code** on the upper right, copy the command, and run the copied command in a terminal window which is in your newly created folder.

Note: According to the operating system you use, you may need to install Git if it is not automatically installed. To install, you can follow the instructions in the link.

3.2 Container Creation

This step was explained in the README.md file of the GitHub project. Please follow the instructions to create your container environment.

Docker containers are isolated environments that can be personalized for your own purposes. Therefore, they are very helpful tools for environment-dependent projects.

Important Note: In this assignment, it is important that you write your codes using specific versions of the tools in the environment. Therefore, you **must** use the shared container environment while developing. Your codes will be tested in the shared container environment.

Important Note: As mentioned in the 'README.md' file in the GitHub repository, the 'code' folder in the host machine is mounted to the 'app' folder in both containers. These folders are the same, and any change that is made in any container or the host machine will directly affect the folder. Therefore, use this folder to avoid losing your codes, and be careful while editing the folder.

3.3 Choosing the Covert Channel Type

In ODTUClass, there is a choice item that helps you to choose the covert channel type you prefer to implement. For every project group, there is a unique covert channel type, and you can choose which one to implement. When the choice item is available to you, the selection will be done in an FCFS manner. You can change your choice during the assignment deadline; however, your assignment will be evaluated and graded according to your last choice. General explanations of the covert channel types are explained in the 'What is a Covert Channel?' section, you can follow the explanations for the chosen covert channel type.

4 Implementation of the Covert Channel

The files in the app folder (or in the code folder for your local environment) are explained one by one in this section.

In your implementation, you should create a consensus between the sender and receiver, which means the packets should be sent in a way that the receiver should convert them to a meaningful message. It is important that you should **maximize** the covert channel capacity and send more information as much as possible in a specific time interval. However, you must **not** send the entire data with one packet; instead, you are expected to send one bit or two bits with a transmitted packet.

Note: If you propose a **very smart and undetectable technique** to transfer the message in a covert channel communication, you are free to transfer more than one bit or two bits. **We are not setting a limit for your implementation, you are very welcome to use your imagination to design your own covert channel communication.** If you explain your smart technique in 'README.md', you might be rewarded with **bonus points**.

- **'Covert Timing Channels'** that exploit **'Packet Inter-Arrival Times'**: You can encode the message by exploiting the time intervals between the consecutive messages, which means you should wait intentionally between the consecutive packets before sending. You should also consider the network delays as it limits the covert channel capacity.

  Example: If the inter-arrival time between the packets is less than 10 ms, it can be decoded as 1; otherwise 0.

- **'Covert Timing Channels'** that exploit **'Idle Period Between Packet Bursts'**: You can encode the message by exploiting the time intervals between packet bursts. Packet bursting means sending more than one packet without waiting between the consecutive packets. However, the number of the packets in the bursts should not be static, which means **you are not allowed to send the same number of packets in each burst**. You should consider network delays while determining the idle time between the bursts as a network delay might be interpreted as idle time.

  Example: If the inter-arrival time between the bursts is less than 20 ms and more than 10 ms, it can be decoded as 1 (Hint: Because you do not know the number of packets in a burst, there should be a lower time limit also.). Otherwise, if it is more than 20 ms, it can be decoded as 0.

- **'Covert Storage Channels'** that exploit **'Packet Bursting'**: You can use the packet quantities in this technique to encode and decode the message. The number of packets in a burst should be decoded as 0 or 1. However, there should also be an idle time between bursts to detect the number of packets and you should consider network delays while determining the idle time between the bursts.

  Example: While transmitting 2 packets in a burst can be decoded as 1, transmitting 3 packets in a burst can be decoded as 0.

- **'Covert Storage Channels'** that exploit **'Packet Size Variation'**: The packet size can be customized by exploiting the optional data and payload fields of the protocols. However, there is a lower limit for the packet size naturally because of the headers of the packets. Hint: You can use 'Raw()' class of scapy to add a payload and manipulate the packet size.

  Example: If the size of a transmitted packet is more than 100, it can be decoded as 1, otherwise 0.

- **'Covert Storage Channels'** that exploit **'Protocol Field Manipulation'**: The chosen protocol-specific field, whose size may vary, can be exploited to encode the data. Each field shared with you in ODTUClass has been tested to determine whether it can be edited or not. As the most important part, **you cannot directly transmit one bit or all bits of the binary message** with these exploited fields. Instead, you must design a consensus between the sender and receiver. In other words, the transmitted message should carry information that can be decoded as 0 or 1 on the receiver side; however, it should not directly transmit 0 or 1 bit of the binary message. As mentioned previously, you can decode and send at most one bit or two bits of the binary message with each transmission. **If you directly transmit one or more bits of the binary message without encoding, you will lose the entire point of the covert channel implementation part.** Additionally, be careful about exploiting the fields as they might need to carry

some specific values in some cases, and you should determine the possible values by reading the protocol manuals. In these cases, you can encode your information by using only the possible values for these fields.

Example: Let's have a 4-bit field to exploit. If the decimal representation of the number is less than 8, it can be decoded as 0, otherwise 1.

Unlike the first phase, the sender and receiver docker containers have static IPs. You do not need IP information if you prefer a protocol whose layer is lower than the network layer. Otherwise, you can find the IPs of the containers in the 'docker-compose.yaml' file.

Important Note: While grading, 'CovertChannelBase', 'Makefile' and 'run.py' files will be replaced with the original copies; therefore, any change made in these files will **not** be considered in the evaluation of your codes.

Important Note: You are not allowed to create any files and folders or edit the file and folder hierarchy in this phase. You **must** use the 'MyCovertChannel' class for your implementation, and it is enough to implement the sender and receiver of a covert channel communication.

4.1 MyCovertChannel.py

In this assignment, there are sender and receiver containers to implement a covert channel sender and receiver pair. Different from the first phase, you do not have different folders and files to implement sender and receiver; instead, you have one class 'MyCovertChannel' and in this class you are expected to implement the covert channel sender and receiver pair. The 'send' and 'receive' functions in the 'MyCovertChannel' class are defined for that purpose and the covert channel communication will be triggered by calling these functions.

- **send():** In this function, you are expected to create a **random message (you must use function/s in 'CovertChannelBase')**, and send it to the receiver container by using the 'send' function given in the 'CovertChannelBase' class.

  Important Note: As you analyze the 'CovertChannelBase' class, you can see that the dot character is not used while creating random messages. **The dot character is the stopping character for the covert channel communication.** When the dot character is received, the entire covert channel communication should finish, and **the receiver should stop capturing the new packets**.

- **receive():** In this function, you are expected to receive and decode the transferred message. Because there are many types of covert channels, the receiver implementation depends on the chosen covert channel type, and you can call the scapy's 'sniff' function without any constraint.

After the implementation of the 'send' and 'receive' functions, you must rewrite the comment part to explain your code basically for each functions.

Important Note: Your entire implementation in the 'MyCovertChannel' class should be parametric, which means there should not be any constant values that you have determined in your algorithm. All of the values should be taken as a parameter in the 'send' and 'receive' functions.

Important Note: You are not allowed to change the file or class name in 'MyCovertChannel.py'. You can edit the class in any way you want (e.g. adding helper functions); however, there must be a 'send' and a 'receive' function, remember that the covert channel communication will be triggered by calling these functions.

4.2 CovertChannelBase.py

This is the base class for 'MyCovertChannel' and you can use the given functions in your implementation. However, you are not allowed to change the 'CovertChannelBase' class and the 'CovertChannelBase.py' file, please make your implementation in the 'MyCovertChannel' class. The detailed explanations for each function is given in the file, you should read them before using the functions.

Important Note: You **must** use the given 'send' function in the 'CovertChannelBase' class to send a packet that was created according to your covert channel communication. You are not allowed to use the scapy's send function directly, the given 'send' function calls it already. The reason is that we may change or analyze the created packets by your implementation during the evaluation of your codes.

4.3 config.json, Makefile and run.py

As the name suggests, it is the configuration file to run and test your implementation. Different from the first phase, you will not call your 'send' and 'receive' functions directly. Instead, there are 'Makefile' and 'run.py' files to test the sending and receiving functionality you implemented by commanding only 'make send' and 'make receive' in a terminal window.

For any parameter that you have defined in the 'send' and 'receive' functions, you should add them to the 'config.json' file with the same parameter name and determine a value for each of them. When you trigger the send or receive operations using the 'Makefile', the 'run.py' file reads the 'config.json' file with the parameter names and their values defined for the 'send' and 'receive' functions. After reading, it calls your 'send' and 'receive' functions implemented in the 'MyCovertChannel' class with the read parameters and values. Therefore, for each parameter defined in the 'send' and 'receive' functions, there should be a **corresponding field with the same parameter name** in the 'config.json' file. Additionally, please complete the 'covert_channel_code' field in the 'config.json' file, according to the covert channel type you have chosen via the ODTUClass covert channel type choice item (the code can be seen in the parentheses).

You **must log** the sent and received message strings using the logging functionality given in the 'CovertChannelBase' class to compare whether the message is transferred successfully or not. The 'log_file_name' fields are defined for the 'send' and 'receive' functions separately in the 'config.json' file. While logging, you should use these file names, which are given as parameters in the 'send' and 'receive' functions. In the end, the generated random string before sending and the received string should be logged within the files whose names are determined in the 'config.json' file as 'log_file_name'. To check the difference between the sent and received logs, you can use the 'make compare' command defined in the 'Makefile'. This command reads and compares the files whose names are determined with the 'log_file_name' field in the send and receive parameters. Therefore, you are not allowed to change the name of the 'log_file_name' field and are expected to determine the log file name as you wish.

A step-by-step explanation about how you can run and test the 'send' and 'receive' functions is:

1. Implement the 'send' and 'receive' functions in the 'MyCovertChannel' class according to the covert channel you have chosen.
2. Everything in your implementation should be parametric; therefore, define the required parameters for the 'send' and 'receive' functions.
3. Add the corresponding fields to the 'config.json' file with the same parameter names.
4. Run 'make receive' at first to capture the received packets, and 'make send' to send the covert channel packets.
5. After the message transfer is terminated (if you see 'Sender is finished!' and 'Receiver is finished!' in the containers, it implies that the message transfer is finished.), run 'make compare' to check whether the message transfer is successfully completed or not.

While debugging your implementation, you might need to analyze the packet/s by capturing the live network traffic. For example, you may want to capture the outgoing traffic from the 'sender' or incoming traffic to the 'receiver' to analyze all of the header and data information of the lively transmitted packets. For that purpose, there is a command line tool (TShark) which lets you capture packet data from a live network, or read packets from a previously saved capture file, either printing a decoded form of those packets to the standard output or writing the packets to a file. TShark was installed on the containers and you can follow the (link) for more usage information.

4.4 README.md

In this assignment, you will not write a report file different from the first phase; instead, you should prepare a README.md file. You are expected to explain your study in detail as you share your work with the community in

a public repository. Anyone should understand your project when reading it without having previous information about the assignment.

Additionally, there might be some upper or lower limitations for the parameters defined in your implementations, such as the covert timing channel's minimum threshold limits. You should explain these limitations clearly in the 'README.md' file so that while grading we can consider them.

As mentioned, you should maximize the covert channel capacity and send information as much as possible. You are expected to present your covert channel capacity (bits per second) in your 'README.md' file and if the capacity is too low than expected, there might be a point deduction. You can follow the steps below to measure covert channel capacity:

1. Create a binary message whose length is 128 (contains 16 character, and you can set both the min_length and max_length as 16).
2. Start the timer just before sending the first packet.
3. Finish the timer, just after sending the last packet.
4. Find the difference in seconds.
5. Divide 128 by the calculated time in seconds.
6. You can find your covert channel capacity in bits per second.
7. Report the covert channel capacity in bits per second in the 'README.md' file.

4.5 Documentation Using Sphinx

As in the first phase, you will use Sphinx, which is a documentation generator for writing technical documentation, and the configuration files are also provided for you. Different from the first phase, you should use the 'Makefile' provided in the 'app' folder to create the documentation for your project. Running 'make documentation' in the 'app' folder on one of the containers (**not** in the 'code' folder on the host machine because Sphinx was installed in the containers) is enough to create the documentation. After a successful run, you should check the '_build/html' folder, and you will see the automatically generated documentation of your codes.

Important Note: As you will see when you open the automatically generated documentation files ('index.html'), you have to change the 'index.rst' file by replacing the text with your names, group ID, and **your link to the forked public repository (different from the first phase)**. **Be careful, if you do not share the public repository with us, we cannot reach your work and evaluate your codes.** After changing, you can safely rerun the 'make documentation' command.

5 Submission & Grading

You should compress the following files and folders as <groupID>.tar.gz and upload it to ODTUClass (it is enough to upload by one of the group members):

- '_build' folder including documentation files and folders created by Sphinx
- 'MyCovertChannel.py' file
- 'config.json' file
- 'README.md' file in the 'app' folder (or in the code folder for your local environment)

The testing process will include two steps, black-box testing and white-box testing. If your code does not pass the black-box test, you will get a 0 from the coding part. However, if your code is passed, a white-box test will also be applied to check whether you have followed the restrictions.

For the black-box testing, the following steps will be applied:

1. ‘make receiver’ command will be run at first to capture the received packets, and ‘make send’ to send the covert channel packets.
2. ‘make compare’ command will be run to check whether the message transfer is successfully completed or not.

The grading of the assignment will be conducted using the rules shown below.

- covert channel implementation (with presenting its capacity) that passes black-box and white-box testing (80 pts)
- successfully created Sphinx documentation (10 pts)
- the completed repository with a ‘README.md’ file (10 pts)