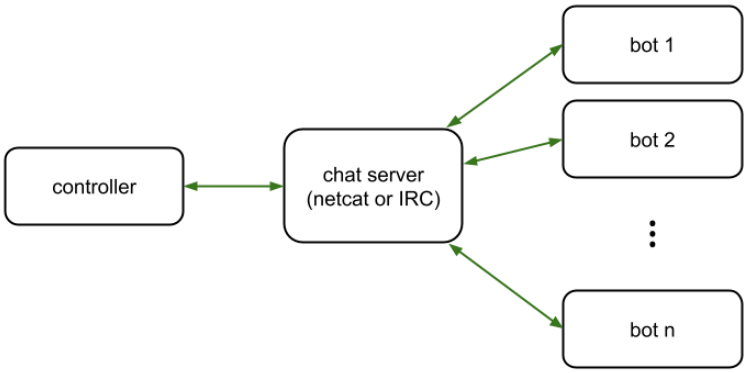
CPSC 526 Winter 2025

Assignment 4

Due date: posted on D2L
Weight: 20% of your final grade.
Group work is allowed but not required. Maximum group size: 2 students.

For this assignment you will design and implement your own botnet. A "botnet" is a network of connected computers ("bots") that an attacker can control remotely via a network ("net"). Botnets are typically used for malicious purposes, such as launching Distributed Denial-of-Service attacks to overwhelm a victim's server with traffic, sending spam emails, mining cryptocurrencies, and many other malicious purposes. The attacker manages the botnet by issuing commands through a command-and-control system, which then sends the commands to be executed by each bot. There are many different mechanisms that can be used to deliver such commands to each bot, such as IRC (Internet Relay Chat).

You are going to write both the controller program and the bot program. The controller and the bots will communicate with each other through a chat server. The controller program will accept commands from the user, send them to the bots, and report results back to the user.



You will write two versions of the botnet, one using a very simple communication mechanism, and the other one using a more complicated communication mechanism. Tasks 1 and 2 ask you to write the botnet using a simple ncat-based broker server. In Tasks 3 and 4, you will improve your simple chatbot from Tasks 1 and 2 by making it support IRC-based communication.

Task 1 – Implement `ncbot.py`

Write a bot called `ncbot.py` that connects to an ncat-based broker server. After connecting to the server, the bot will wait for the ncat server to send commands, authenticate them and execute them, and optionally send results of the commands back to the ncat server. See the appendix for information on how to start the ncat broker server. You should be able to fully debug your `ncbot.py` program using `ncat` in client mode (yes, the same `ncat` command can be used to start a server in one terminal, and then to connect to it from a different terminal).

Task 1.a – Connecting the bot to the server

Your bot must accept the following 4 command line arguments:

```
$ ./ncbot.py <hostname>:<port> <nick> <secret>
```

The `<hostname>` and `<port>` specify the location where ncat server is listening for connections. Your bot should immediately connect to the given hostname at the given port. If the connection cannot be made, it should keep retrying forever, with a 5s sleep between retries. If the bot unexpectedly loses connection to the ncat server at any time, it should try to reconnect forever, with 5s sleep between retries. The bot should print an appropriate diagnostic message on standard output after each connection attempt and after each unexpected disconnect. Here is a possible output of your bot:

```
$ ./ncbot.py csx1:12345 bot1 green
Failed to connect.
Failed to connect.
Connected.
Disconnected.
Failed to connect.
…
```

The output above could result by starting the bot first, then starting the ncat broker, then killing the ncat broker while the bot continues to run.

Every time your bot connects or re-connects to the server, it should send the server a single line containing the message "`-joined <nick>`", where `<nick>` is the nickname given to the bot on the command line. The bot should remember its nickname, as it will be used by some commands. For example, if you start the Ncat broker on csx1, your bot should eventually connect to it:

```
(csx1)$ nc --broker -l 12345
```

For debugging purposes, in a different terminal you can start an additional Ncat client to display all traffic sent to the server. And if you start it quickly enough after you start the server, you might even see the connection from your bot (as shown below):

```
(cxs2)$ nc csx1 12345
-joined bot1
```

Your bot must remember the `<secret>` from the command line arguments because it will be used to authenticate the commands from the server.

Task 1.b – Implement authentication of commands

After connecting to the server, your bot will listen for commands and then authenticate each command. If the authentication is successful, your bot will execute the command. Each command will be specified on its own line, and will have the following format (the parts in angle brackets will be separated by 1 or more spaces):

`<nonce> <mac> <command> <argument1> <argument2> …`

The `<nonce>` and `<mac>` will be used to prove the command came from someone that knows the `<secret>` that was given to the bot on command line when it was started. The command will be authenticated if:

- The `<mac>` matches the first 8 hexadecimal digits of the message authentication code computed using SHA-256 from concatenation of the `<nonce>` with the `<secret>`; and
- The bot has never seen `<nonce>` before.

If the command is authenticated, your bot needs to remember the `<nonce>` and never authenticate another command with that nonce. Pseudocode for authenticating commands:

1.) *if <nonce> is in seen_nonces: ignore command*
2.) *compute mac2 = sha-256(<nonce>+<secret>)*
3.) *if mac2 != <mac>: ignore command*
4.) *insert <nonce> into seen_nonces*
5.) *execute command*

For example, if "green" is the secret, and nonce=123, then the MAC could be computed on command line like this:

```
$ echo -n "123green" | sha256sum | cut -b1-8
0dae83fd
```

A command using this MAC would look like this:

```
123 0dae83fd status
```

If the bot received this command, and it has never seen 123 as a nonce, it would execute the command "status". If it has seen the nonce 123 before, it would ignore the command.

Please note that this type of authentication is not secure. I designed it to make it simple to debug. A more secure bot could use the entire output of HMAC-SHA256 computed over the entire message. An even better solution would involve digital signatures, to avoid hard-coding secrets into bots. But for this assignment, to make your code easier to debug, we'll use 8-digit MACs.

Task 1.c – Implement command `status`

The `status` command does not have any arguments. When your bot receives the `status` command, it will reply with the text "`-status <nick> <n>`", where `<nick>` is the bot's nickname, and `<n>` represents the number of authenticated commands this bot has executed so far, including the current status command. For example, assume an Ncat server was started on `csx1.ucalgary.ca` server like this:

```
$ nc --broker -l 12312
```

And 4 bots were started like this, from csx2.ucalgary.ca server:

```
$ ./ncbot.py csx1:12312 bot-n green
Connected.
```

Where “bot-n” were nicks given to the bots, e.g. “bot-1”, “bot-2”, “bot-3” and “bot-4”. You connect to the ncat server using ncat in client mode and send a status command to your 4 bots like this:

```
$ nc csx1 12312
1 497404d2 status
-status bot-3 1
-status bot-1 1
-status bot-2 1
-status bot-4 1
^C
```

You can end the communication with the server using <ctrl-c>. Note that bots that joined the server at a later time would report different (lower) counts.

Task 1.d – Implement command `shutdown`

The `shutdown` command does not have any arguments. When your bot receives the `shutdown` command, it will reply with the text “`-shutdown <nick>`”, where `<nick>` is the nickname of the bot, and then terminate. Please flush the socket before terminating, so that the server receives the reply. Example of possible output in server chat:

```
$ nc csx1 12312
2 896c03ea shutdown
-shutdown bot-1
-shutdown bot-2
```

Task 1.e – Implement command `attack <hostname>:<port>`

The `attack` command will make your bot perform a network “attack” on a remote target computer. The attack will consist of connecting to the given `<hostname>` at the provided `<port>` and then sending a single line message to the target. The message will contain two entries: `<nick>` and `<nonce>`, where `<nick>` will be the bot’s nickname, and `<nonce>` will be the nonce used in the authentication part of the command. If your bot is unable to connect to the target computer in 3s, it should give up trying.

After sending the line to the target, your bot should close its connection to the target. Once the attack is successfully completed, the bot will send back to the server a message of the format “`-attack <nick> OK`”, where `<nick>` is the bot’s nickname. If the attack was unsuccessful, the message sent to the server will have the format: “`-attack <nick> FAIL <error-message>`”, where the `<error-message>` should contain some information about why the attack failed.

Example of possible output in server chat:

```
$ nc csx1 12312
3 6b366bb9 attack localhost:333
-attack bot-1 FAIL connection refused
-attack bot-2 FAIL timeout
-attack bot-3 OK
```

To test your attacks, you can use ncat with the ‘-k’ option. This will make Ncat accept multiple connections:

```
$ nc -k -l 8963
```

Task 1.f – Implement command `move <hostname>:<port>`

The move command will result in:

1.) Your bot will replay with a message “`-move <nick>`” to the ncat server.
2.) Your bot will then disconnect from the current ncat server.
3.) Your bot will attempt to connect to the new server located at `<hostname>:<port>`, using the same algorithm as described in Task 1a, including the same diagnostic messages printed to standard output.
4.) When your bot connects to the new server, it will send to the server the same message as described in Task 1a.

Example of a possible output in server chat:

```
$ nc csx1 12312
4 b8416fc6 move csx2:20001
-move bot2
```

```
-move bot3
-move bot2
```

Task 2 – nccontroller.py

While it is possible to control your botnet just by using ncat in client mode, it is cumbersome. It is inconvenient to have to manually compute the authentication information for each command, and it could be tedious to have to interpret the outputs from many bots. To make the interactions with your army of bots more enjoyable, you will write a bot controller program called `nccontroller.py`.

The command line arguments for the controller are similar to the bot's command line arguments:

```
$ ./nccontroller.py <hostname>:<port> <secret-phrase>
```

Your controller should connect to the given server, print out a message stating so, and then accept commands from standard input, one command per line. If the controller cannot connect to the server, it should print out an error message and quit. If the controller gets disconnected from the server, it should print out an error message and quit. If a command is typed in that the controller does not understand, the controller should print out an appropriate error message and otherwise ignore the command.

The controller should support all commands that your bot understands. Upon receiving the command from standard input, the controller should calculate an appropriate nonce and MAC, and then send the command with the added nonce and MAC to the server. The controller then waits for any responses from the bots, collects them and prints a summary of the responses to standard output.

The controller should allow some small amount of time for the bots to send back their responses. You should experiment to determine what would be a good timeout value for this, but do not make it longer than 5 seconds. During this time the controller should collect the responses from the bots. Once the controller is ready to accept a new command, it should prompt the user to do so.

Task 2.a – Implement command `status`

The controller will send the "`status`" command to the ncat server and wait for bots to reply. The controller will then print out the list of bots (their nicks & command counts), and their total number. Example:

```
$ ./nccontroller.py csx1:12312 green
cmd> status
  Waiting 5s to gather replies.
  Result: 5 bots replied.
    bot-3 (2), bot-1 (2), bot-2 (2), bot-4 (2), bot-5 (1)
```

Task 2.b – Implement command `shutdown`

The controller will send the "`shutdown`" command to the ncat server, then wait for bots to send their replies, and then print out the summary of their replies. The summary should include number of bots and their nicks. Example:

```
cmd> shutdown
  Waiting 5s to gather replies.
  Result: 4 bots shut down.
    bot-3, bot-1, bot-2, bot-4
```

Task 2.c – Implement command `attack <host-name>:<port>`

The controller will send the "`attack`" command to the ncat server. It will wait for bots to post their responses and then print out the summary of their replies. The summary should include the bots that succeeded in their attacks (their number and their nicks). The summary should then list all bots that attempted the attack, but failed, together with the error messages.

Example:

```
cmd> attack localhost:4545
  Waiting 5s to gather replies.
  Result: 2 bots attacked successfully:
    bot1, bot5
  2 bots failed to attack:
    badrobot: timeout
    yellowrobot: connection refused
```

Task 2.d – Implement command `move <host-name>:<port>`

The controller will send the "`move`" command to the server. It will wait for bots to post their responses, and then print out the number of bots and their nicks that moved. The controller should remain connected to the current ncat server. Example:

```
cmd> move csx2:20001
  Waiting 5s to gather replies.
  Result: 3 bots moved.
    redbot, greenbot, bluebot
cmd> status
  Waiting 5s to gather replies.
  Result: 0 bots replied.
```

Task 2.e – Implement command `quit`

The controller will disconnect from the server and terminate. The bots are unaffected. Example:

```
cmd> quit
Disconnected.
$
```

Example controller session

Suppose an ncat server was started on csx1 on port 3001 like this:

```
(csx1)$ nc --broker -l 3001
```

and 4 bots with nicks "bot1", "bot2", "bot3" and "bot4" connected to it, using the secret "redonionsarepurple". The following is a possible session using the controller:

```
$ ./nccontroller.py csx1:3001 redonionsarepurple
cmd> status
  Waiting 5s to gather replies.
  Result: 4 bots discovered.
    bot3 (1), bot1 (1), bot4 (1), bot2 (1)
cmd> attack csx3.ucalgary.ca:8899
  Waiting 5s to gather replies.
  Result: 3 bots attacked successfully:
    bot3, bot1, badrobot                        # badrobot added to server
  2 bots failed to attack:
    bot2: timeout
    bot4: connection refused
cmd> attack blah.blah.blah:8899
  Waiting 5s to gather replies.
  Result: 0 bots attacked successfully:
  3 bots failed to attack:                      # badrobot and bot4 left
    bot1: no such hostname
    bot2: no such hostname
    bot3: no such hostname
cmd> shutdown
  Waiting 5s to gather replies.
  Result: 4 bots shut down:
    bot1, bot2, bot4, bot1                      # bot4 came back
cmd> shutdown
  Waiting 5s to gather replies.
  Result: 0 bots shut down:
cmd> status
  Waiting 5s to gather replies.
  Result: 0 bots discovered.
cmd> attack csx3.cpsc.ucalgary.ca:8899
  Waiting 5s to gather replies.
  Result: 0 bots attacked successfully:
  0 bots failed to attack:
cmd> status
```

```
  Waiting 5s to gather replies.
  Result: 1 bots discovered.
    goodrobot (1)                                # goodrobot added to server
cmd> quit
Bye.
$
```

Task 3 – Implement `ircbot.py`

Reimplement `ncbot.py` from Task 1, so that it connects to an IRC server, instead of an ncat server. The command line arguments are little bit different:

```
$ ./ircbot.py <hostname>:<port> <channel> <secret>
```

Instead of accepting a nickname for the bot, the bot will accept the name of the `<channel>` that the bot needs to join. The connection sequence and actions should be the same as with Ncat-based bot, although the bot will need to pick a random nickname.

I give you a bit of flexibility and leave room for some creativity for this task. The main purpose of this task is to exploit an existing IRC server to utilize as the communication mechanism for you to connect to your botnet. For example, it is up to you to decide how you want to communicate results back – whether using channel-wide messages or using direct user-to-user messages (IRC supports both). However you decided to implement your IRC bot, it should be possible to use a standard IRC client to connect to the same server/channel and control your botnet by typing in commands.

Task 4 – Implement `irccontroller.py`

Reimplement `nccontroller.py` from Task 2 to be able to control `ircbot.py` bots via an IRC server, but otherwise your IRC-based controller should behave exactly like the Ncat-based bot controller. The command line arguments are little bit different – instead of accepting a nickname for the bot, the user specifies the channel that the bot needs to join:

```
$ ./nccontroller.py <hostname>:<port> <channel> <secret-phrase>
```

The controller should issue its commands to the bots by posting public messages to the channel.

IRC related information

I will try to deploy an IRC server for you to test your code on. Information will be posted on the assignment page on D2L and/or on Ed Discussion. However, you may decide to run your own IRC server for testing as well. Here is a link to a simple IRC server written in python:

https://github.com/jrosdahl/miniircd

You can implement the IRC communication from scratch. There are many resources available on the web related to the IRC protocol, e.g. a nice compact description of the IRC protocol can be found here:

- https://chi.cs.uchicago.edu/chirc/irc.html
- https://chi.cs.uchicago.edu/chirc/irc_examples.html

If you prefer, you may use an existing library to help you with the IRC protocol, such as:

https://pypi.python.org/pypi/irc

Two command-line IRC clients are installed on our cslinuxlab machines: `weechat` and `irssi`.

Additional notes

- You may implement your program in Python, C or C++.
- Your program must run on our cslinuxlab machines.
- You may not call external programs.
- Your programs should not crash under any circumstances.
- Invalid commands passed to the bots should be ignored.
- Invalid commands given to the controller should report some type of error message to `stdout`.

Group work & demo

You will need to demo your assignment to your TA. The time for your demo will be arranged by your TA. During the demo, you may be asked to adjust some functionality of your code to show you are familiar with the code you submitted.

You are allowed to work on this assignment with another student (max. group size is 2). If you decide to group up, make sure that both of you understand all of your code. If you are not able to explain any part of your code during the demo, you may receive a reduced grade.

Submissions

Make sure all your code works on cslinuxlab machines.

All group members must submit 5 files on D2L:

`ncbot.py`	Implementation of task 1
`nccontroller.py`	Implementation of task 2
`ircbot.py`	Implementation of task 3
`irccontroller.py`	Implementation of task 4
`readme.[txt|pdf|docx]`	Include names and IDs of all group members at the top.

You must submit the above to D2L to receive any marks for this assignment.

General information about all assignments

- All assignments are due on the date listed on D2L. Late penalties will be applied as described in the official course outline and as discussed during the first week of classes.
- Extensions may be granted only by the course instructor.
- Most assignments will be marked by your TAs. If you have questions about assignment marking, contact your TA first. If you still have questions after you have talked to your TA, then you can contact your instructor.
- All programs you submit must run on departmental Linux machines. If we are unable to run your code on the departmental Linux machines, you will receive 0 marks.
- Assignments must reflect your own, or your group's individual work. Here are some examples of what you are not allowed to do for assignments: you are not allowed to copy code or written answers (in part, or in whole) from anyone else; you are not allowed to collaborate with anyone; you are not allowed to share your solutions (including code or pseudocode) with anyone; you are not allowed to sell or purchase a solution; you are not allowed to make your code available publicly, you are not allowed to AI tools to generate your code nor answers. This list is not exclusive. For further information on plagiarism, cheating and other academic misconduct, check the information at this link: http://www.ucalgary.ca/pubs/calendar/current/k.html .
- We may use automated similarity detection software to check for plagiarism. Your submission will be compared to other students (current and previous), as well as to any known online sources. Any cases of detected plagiarism or any other academic misconduct will be investigated and reported.

Marking

`ncbot.py` **functionality**	**50 marks**

Note: all functionalities must be accessible using `nc` client:
- Basic connection to server (able to connect at least once)
- Automatic reconnection to server
- Command authentication (correct hash verification)
- Command authentication (refusing re-used nonces)
- Command `status`
- Command `shutdown`
- Command `attack` basic
  o With timeout support
- Command `move` basic
  o With auto-reconnect support
	
	5
	5
	5
	5
	5
	5
	5
	5
	5
	5

`nccontroller.py` **functionality**	**30 marks**
- Authenticated commands - sending correct hashes - Authenticated commands - sending unique nonces - Basic support for 5 commands `status shutdown attack move quit` - Supporting `status` with summary - Supporting `shutdown` with summary - Supporting `attack` with summary - Supporting `move` with summary	4 4 5 (1 for each) 4 4 4 4
`ircbot.py` **functionality**	**10 marks**
note: all functionalities must be accessible using IRC chat client: - any 1 command working - `move` command working - 3 commands working - 4 commands working - 5 commands working	 5 2 1 1 1
`irccontroller.py` **functionality**	**10 marks**
- per command	2 per command
Penalties	
Crashes	10 per crash, up to 30
Ugly/undocumented source code	up to 20

Appendix A – Ncat-broker server

Information taken from https://nmap.org/ncat/guide/ncat-broker.html

One of Ncat's most useful and unique abilities is called connection brokering. A listening Ncat in broker mode accepts connections from multiple clients. Anything received from one of the clients is sent back out to all the others. In this way an Ncat broker acts like a network hub, broadcasting all traffic to everyone connected. Activate broker mode with the `--broker` option, which must be combined with `--listen`.

For example, start ncat broker in terminal 1. Then, connect to it from 3 different terminals:

```
$ ssh csx1
$ ncat --broker -l 12345
```

```
$ ssh csx2
$ nc csx1 12345
```

```
$ ssh csx3
$ nc csx1 12345
```

```
$ ssh cslinuxlab
$ nc csx1 12345
```

Now any message you type in any of the terminals 2-4:

```
$ ssh csx1
$ ncat --broker -l 12345
```

```
$ ssh csx2
$ nc csx1 12345
Hi there
```

```
$ ssh csx3
$ nc csx1 12345
```

```
$ ssh cslinuxlab
$ nc csx1 12345
```

will be sent to the other 2 terminals:

```
$ ssh csx1
$ ncat --broker -l 12345
```

```
$ ssh csx2
$ nc csx1 12345
Hi there
```

```
$ ssh csx3
$ nc csx1 12345
Hi there
```

```
$ ssh cslinuxlab
$ nc csx1 12345
Hi there
```

Appendix B - Connection timeout example

The following code tries to connect to a server 10.0.0.0:1234 but will time out after 3 seconds.

```
import socket
HOST = '10.0.0.0'
PORT = 1234
with socket.create_connection((HOST,PORT), timeout=3) as s:
    s.sendall(b'Hello, world')
```

If the connection takes longer than 3s to establish, the code will raise a `TimeoutError` exception.