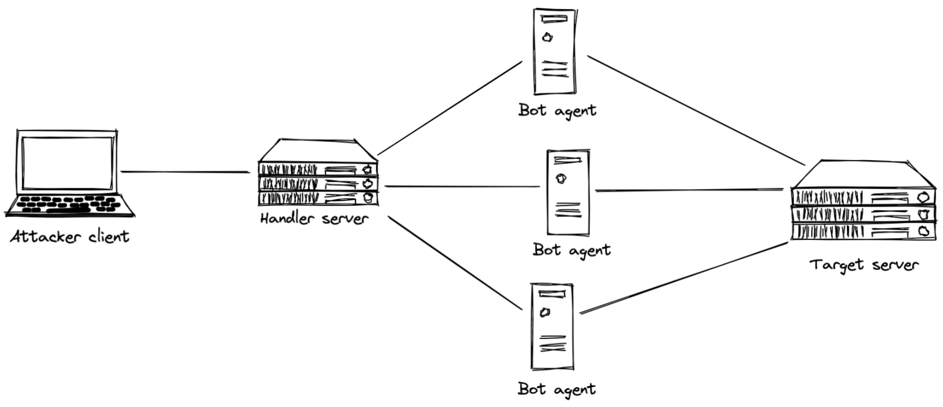
CPSC 329 Unessay Writeup

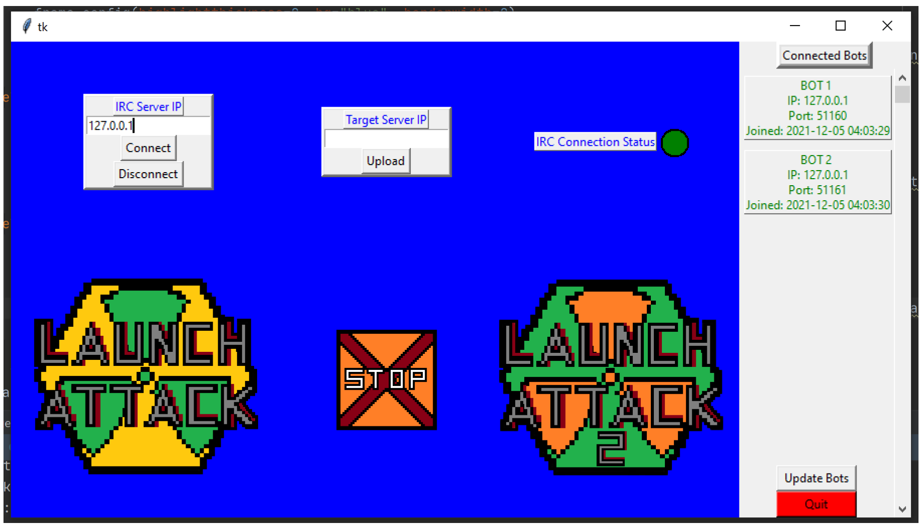
Group 12:
Cole Cathcart, Brody Long, Justin Flores, Khaled Mograbee,
Marilyn Bulanda, Ryan Smit

General Outline:

For our unessay project, we created software for and performed a basic DDoS attack. This was a fun project that allowed us to learn about and create a real-world example of a cyber-security threat that wasn't covered extensively in class, as well as develop skills highly applicable to the field of cybersecurity. To facilitate this attack, we had to create multiple pieces of code and combine them to create a program capable of directing "bots" to target and overwhelm a given IP address with the goal of impacting or outright terminating that IP's services.



The project consists of a client-side application (the "attacker") with a GUI that connects to an IRC server along with several bots. These "bots" are simply other computers on which a bot script is running that allows them to receive orders from the master-client and perform attacks based on said orders. For our project we simply ran this bot-script onto several of our own computers, but in a real-world scenario these scripts could be delivered to hundreds of unsuspecting users through malware such as a trojan, allowing a single attacker to take command of many devices to use for the attack. The IRC server allows the client to communicate with the bots and send them commands such as an IP address to attack, which attack to use and a start/stop attack. The master-client runs in a command-line interface as

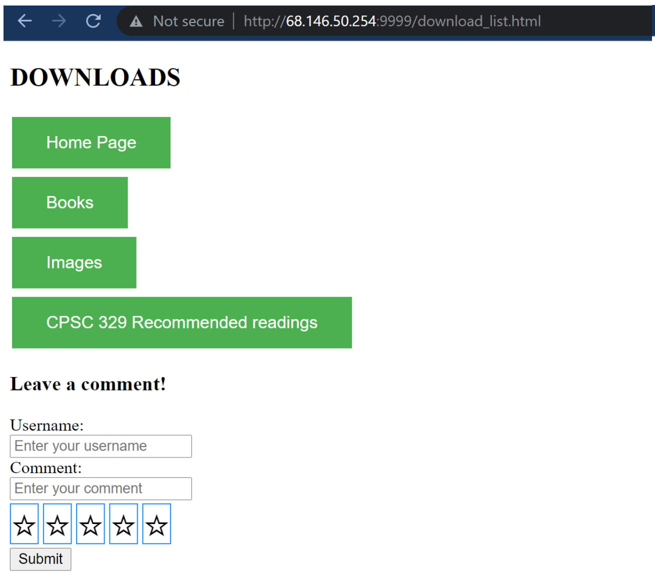


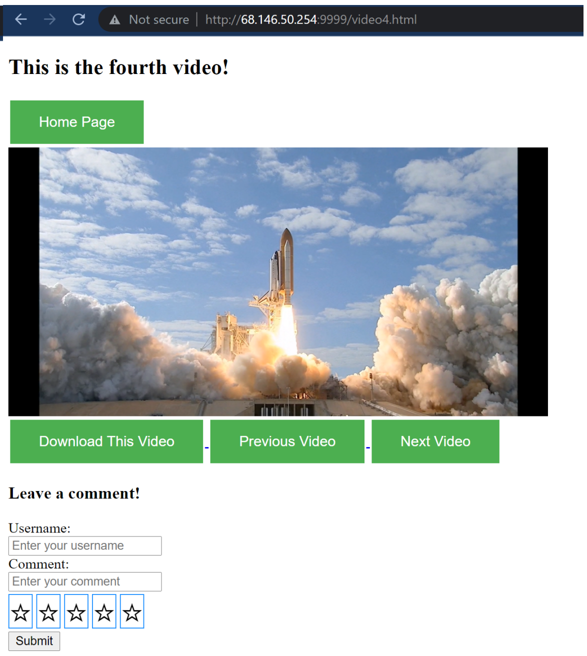
BOT 1
IP: 127.0.0.1
Port: 51160
Joined: 2021-12-05 04:03:29

BOT 2
IP: 127.0.0.1
Port: 51161
Joined: 2021-12-05 04:03:30

well as a GUI, and allows the attacker to connect to a given IRC server, send commands, and see the number of bots connected with their IP, port and join time.

Finally, we created a victim for these bots to attack, a target server hosted on a raspberry pi. The server has a webpage with numerous functions such as file download/upload and video streaming in order to simulate the real-time impact of a bot attack on the performance of these common website features.





Bot Scripts:

As per our proposal, the bot script was written in Python. We used threading to create several instances so that one bot could deliver a more devastating attack. The bot script utilizes the handler server as a "middle man" to receive commands for state changes from the master client via the handler server. The bot script is connected to the handler server with Python's socket library. The handler server sends the bot script commands from the master client to change the target server, choose what attack is to be run, when to start running said attack, and when to stop. We successfully implemented two types of DDoS attacks.

The first attack is a request attack which is a type of HTTP flood attack. This attack overwhelms the target server with HTTP requests, both GET and POST, which the target server is eventually unable to parse. This attack eats up the CPU of the target server and makes the video playback on the website stop. This attack is implemented in a loop to constantly send either GET or POST requests chosen at random to the target server via sockets. The requests are sent with randomly chosen user agents from randomly chosen websites every time. Potential defences against this attack would be for the target server to include adding a web application firewall (WAF), or issuing a requirement like a JavaScript computational challenge to the requesting machine to test if it was a bot[1].

The second attack is a Slowloris attack which tries to keep as many socket connections open to the target server as possible by only sending partial requests and keeping them open for as long as possible. It is DDoSing the server by using up all available connections the target server has. This attack is implemented in a loop to constantly (every 10 seconds) send packets with random headers to the target server, while keeping all of the sockets open, until the target server is overwhelmed. Potential defences against this attack would be for the target server to include increasing the maximum number of clients the server will allow, limiting the number of connections a single IP address is allowed to attempt, placing restrictions on the minimum transfer speed a connection is allowed, and constraining the amount of time a client is permitted to stay connected[2].

1. https://www.netscout.com/what-is-ddos/http-flood-attacks
2. https://www.netscout.com/what-is-ddos/slowloris-attacks

Master-client Handler and GUI:

The Handler

The master-client runs on the command line. It starts by creating a socket and creating a TCP connection with the handler server, then it instantiates a thread for sending messages and a thread for receiving messages. It's written in Python and makes use of the threading and socket libraries. The program takes the host address of the handler-server as the first command line argument and the port (default 8080) as the second. There is a third optional command line argument to identify if the client is a master or bot, which is simply for debugging purposes. The client connects to the handler-server and identifies itself as the master client. If the client doesn't do this, authentication fails and the connection is refused. Once connected, the client displays a list of commands that the user may input, and a prompt saying the client is ready to take input. Commands are in the form {command_type}:{data}. So for authenticating the client as master, the command would be iam:master. The client encodes the user input in utf-8 and sends it to the handler-server, and prints any feedback or confirmation that the command was received. Relevant error handling for requests that can't be parsed and connection errors is present. The client will receive and display the list of connected bots if the listbot: command is sent or any time a new bot connects to the handler-server. The available commands are:

- iam:{client_type}
- listbot:
- changeip:{target_ip}
- changeattk:{attack_type}
- startattk:
- stopattk:
- disconnect:

The client will refuse to start the attack if the target and attack type is not set, and the client will refuse to stop the attack if an attack is not currently running. Upon disconnect, whether manually or due to error, the client will print a log containing each event with a unix timestamp. In our testing, only one error was encountered: whenever the client is closed, it throws a threading related error. It still closes everything as intended and writes the log file, so the error affects nothing. Due to the nature of the error being related to threading, I didn't have the time to solve the issue. In the future, I think the command-line client could be more polished in terms of display, logging, and error handling. Of course, the entire set of programs should also have far more robust security and authentication features, as simply passing the phrase 'iam:master' is sufficient authentication right now. Although these security features would be relevant to the course, our focus was on creating a functional attack, but I'm sure it would be an interesting secondary project to build a strong security system on top of what we have.

The GUI

The GUI interface simply applies the same functions from the master-client command-line interface and implements them into a window with clickable buttons and fillable fields to improve useability. As per our proposal, the GUI was developed using Python's tkinter library, which allows for simple yet functional GUIs with some customization options. The final product runs in a scalable window and features a field to input the IP of the IRC server, a button to connect to it, and an indicator light to tell when one has successfully connected to and been authorized by the IRC. Once the client is connected, all the currently connected bots are displayed on the right-hand side with their IP, port and join time. This list can be refreshed by a button at any time to update the list. A second field allows the user to send the IP of the target server to the bots, and 3 buttons below allow the user to tell the bots to commence one of two different attacks against the target, or to stop a running attack. Finally, a dedicated quit button allows the application to close all processes smoothly. Whenever one of the features on the GUI are interacted with, it simply calls a corresponding function stored within the separate master-client handler file.

Handler Server:

The handler server was essential in performing the DDoS attack architecture that the group proposed at the start of the project. It acted as the middle man between the master server and the connected bot agents. Without the server, attacking the target server with multiple bot agents would not be possible. From the simulations that the group performed, the server was able to handle ten bot agents and one master server without issues, which was more than what the group needed to attack the target server.

As was proposed, the server is implemented using Python and the sockets library. Clients are connected to the server through TCP since the architecture needs a reliable connection between the server and clients. For DDoS attacks to be effective, the system needs as many bot agents attacking simultaneously, which is only possible if the attack details reach all agents. It is possible with TCP since it guarantees data arrives at the destination router, unlike UDP that is best-effort. The server uses a single-threaded approach using the select library to handle multiple clients. There are two reasons for using this approach over the more conventional multithreading approach. The first one is the difficulty of implementing and debugging a multithreaded server. The second one is that it does not provide any performance gains since there are no blocking operations on the server.

The server was the middle man between the master client and bot agents. It means that the server is responsible for ensuring that connected entities are on the same page. The server does this by keeping track of the system's current state and transmitting it to all clients. The state consists of the list of bot agents, which client is the master, the target server IP address, the type of attack to execute, and if the system is executing an attack. Clients can change the state of the system by sending a request, and it is the server's responsibility to calculate the new state and transmit it to all clients. The server is also responsible for identifying the type of each client so that it can handle each request appropriately. It is done by asking clients to identify themselves as master client or bot when they connect to the server. After that, the server will route requests from each client to the appropriate handlers based on their type. In the case of this system, only the master client is allowed to send a request that can modify the state.

The master server has all the features required for the scope of this project, however, the team would like to see how the server would perform with over a hundred clients. In the current state of the server, having that amount of agents connected will not work because of the instance hosting the server. The instance is an AWS EC2 t2.micro which only has 1vCPU and 1GiB of memory. The team would need an instance with more resources to handle that many bots. Another idea would be to have multiple handler servers hosted on different instances, where each server handles separate groups of bot agents. This would mean that the master client would need to connect to more than one handler server. The last improvement would be to write the server code in a lower-level language like C++ for some performance gains.

<u>Target Server:</u>

The target server is implemented using the built in http.server library, and runs on a Raspberry Pi. It begins by first parsing the command line arguments for the IP address and PORT number the server will run on, then it continues to listen for new requests indefinitely until a keyboard interrupt is received. In the case where no arguments are provided, it defaults to running at “localhost:8080”. The webserver utilizes threads for every user that tries to access the website. The webserver is set up to handle basic GET and POST requests. In the event of a GET request the server tries to open the specified file as a bytes object, if successful the file is then sent via HTTP back to the user. If the requested file cannot be opened or found, then a 404 response is instead sent back to the user. POST requests are treated as requests to add a new comment to the comments.json file, so when a POST request is received the webserver gets the length of the POST body and proceeds to decode the POST body into a UTF-8 string. From here it can be parsed to learn the given name of the user, followed by their comment, and then a rating out of 5. This comment data is then added to comments.json in the form:

{Username: [comment, rating, path to the webpage comment was submitted on, IP:PORT, timestamp]}

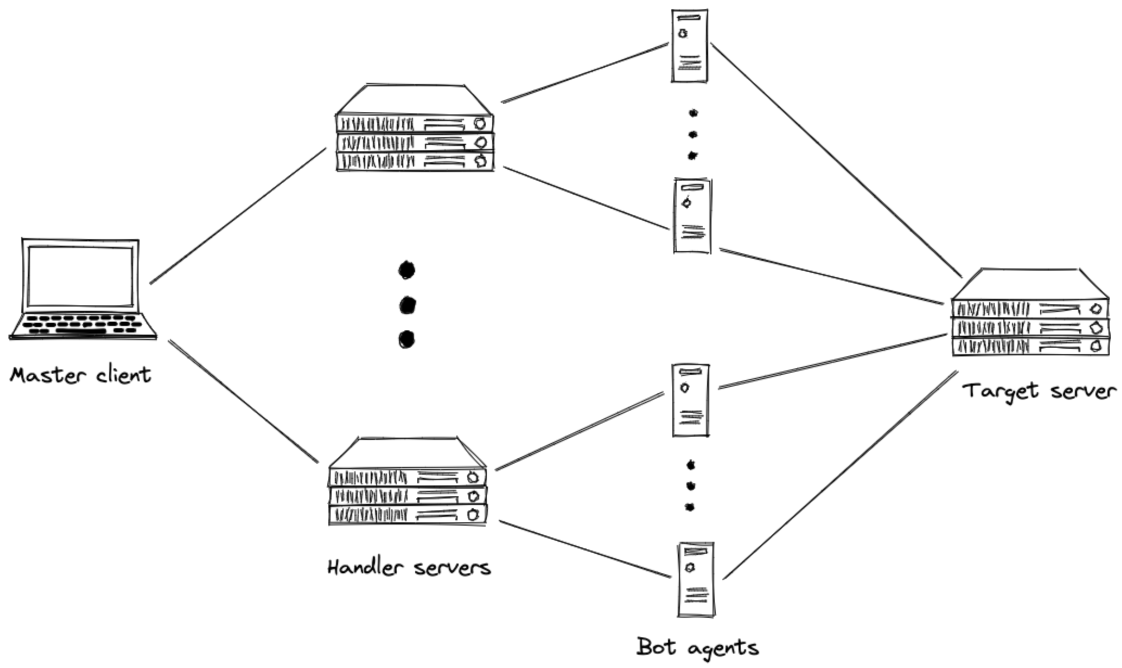
Each time a new request is received and processed by the webserver; the request is logged in requests.log. Each entry in the log is in the form:

IP - TIME - REQUEST - PATH – HTTP VERSION - (optional) CPU USAGE WARNING

The optional CPU usage warning is added to the end of the log whenever CPU usage is above 65%, to get a better idea of what is happening to the server at the time of an attack. All comments and request logs are stored in the respective files located in the server_logs folder in the webserver directory.

The website itself consists of some HTML files used to navigate to different files/videos, initialize basic video playback, submit comments, and download files. The video-js library was used to implement the video player for the videos. An HTML form was used to allow users to input comments and send them to the webserver. As an added feature, I attempted to get adaptive bitrate video streaming using MPEG-DASH encoded videos to work with the webserver, but due to time constraints I was unable to get it working properly, as it would require modifying the http.server library or using some other python webserver framework.

Improved architecture



Outcome:

Repo link: https://github.com/justinf34/CPSC329-Unessay

The goal of a DDoS attack is to interrupt the services of a target server. Using the project we built, we were able to significantly reduce or outright suspend the services of our target in two different ways using only a handful of bots (see the demo video for more info). These attacks were able to be carried out by the bots via remote-control from the master client, thanks to their connection to the handler-server. This successfully satisfies the goals laid out in our rubric, and as such we consider our project to be a complete success.

The learning outcome of this project was to develop knowledge of how to implement basic DDoS attacks to a target server. This is valuable because it gave us a better understanding of how these types of attacks function, and this knowledge allowed us to gain a unique insight into the various ways a DDoS can be prevented/protected against in real life. The creation and interaction with the IRC and target servers from the bots and client provided an opportunity to learn many security and web-programming skills that can be applied elsewhere, such as threading and socket programming. This project was quite large for the scope of this class, and if we had more time, there are many more features/improvements that we would have liked to add. Some of these include:

- Scaling our attacks to include more bots so that we could take down larger servers.

- Adding GUI functionality to see the real-time effect of an attack on the target server from the attacker's end
- Include more varied types of DDoS attacks to attack servers at different ports/through different protocols
- Implementing so-called "amplification" DDoS attacks to more effectively and efficiently take down even larger servers
- Adding better forms of authentication between the master, IRC server, and bots for added security