Developer manual

This is the developer manual of Network Defense SImulator. In this manual, we explain some crucial functions and classes of our simulator. We also mention a few ways to improve our simulator.

Globals

Globals stores important information about the simulation. A few examples are the number of simulations and the run time. Most of the information in globals can be changed by the user in the GUI. Those changeable parameters are at the top of the file.

The parameters that cannot be changed are the access levels, the possible attacks and hardenings, and some of the properties of the hosts in the network. There are three access levels:

- 0, no access
- 1, user access
- 2, admin access

We never deviated from those three access levels, but more could be added in globals. The Host class already has a property access_for_score that indicates at which level the attacker steals the information of the host. This property is currently not being used, but could be useful if more access levels are added.

The possible attacks and hardenings are stored in the arrays atts_h, atts_e, hard_h, and hard_e. Atts_h contains the privilege escalations. Atts_e contains exploits. Hard_h contains the host hardenings. Hard_e contains the edge hardenings. More possible actions can be added to the simulation by adding the action to the corresponding array. Adding another exploit to atts_e will result in the simulator having another exploit.

The arrays hardware, os, services, and processes are used when generating the network. The first entries of hardware and os are used as the hardware and os of all the hosts in the network. The first or first two entries of services and processes are used as services and processes in the network. We never implemented any action that targets the hardware or os, thus changing those parameters does not change the results of the simulator.

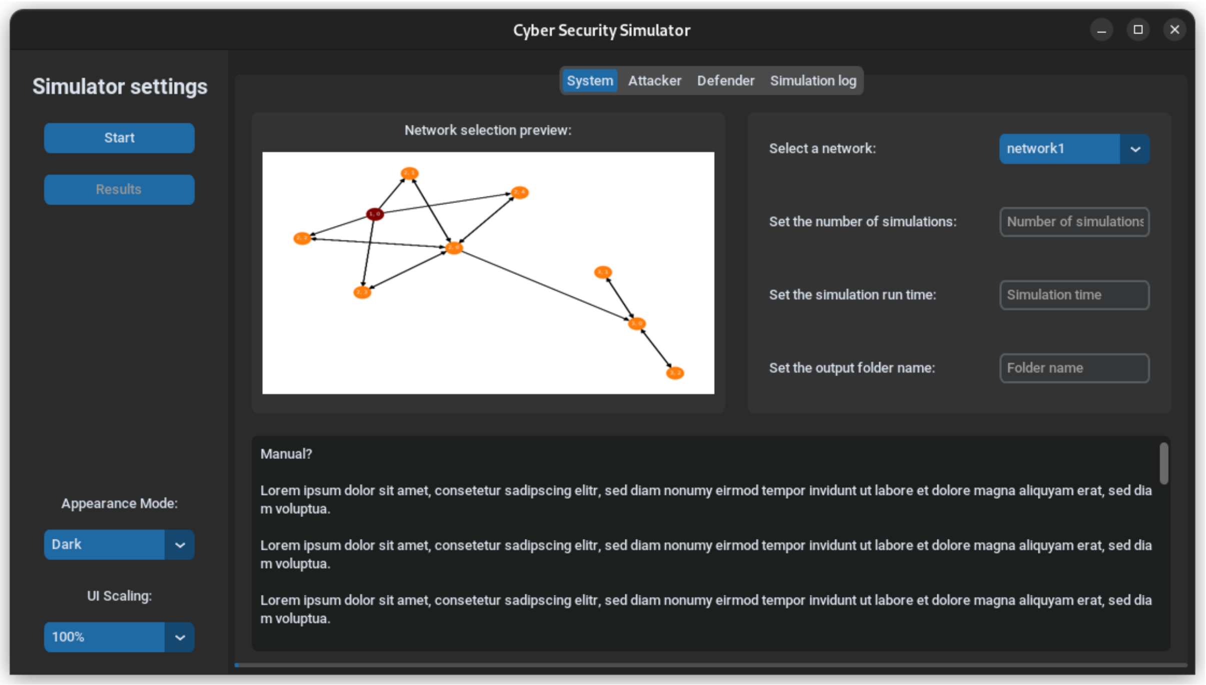
Graphical User Interface

The graphical user interface is made with Custom Tkinter and has its own in-detail documentation of how to use it: https://customtkinter.tomschimansky.com/documentation/ The other Ctk Messagebox which is being used also provides its own documentation for the use which can be found here: https://github.com/Akascape/CTkMessagebox

In the following section, I will talk about how the GUI is set up and how to add new elements. First, the GUI is created in the ‘simulator.py’ file. This file consists of a class App and a class ResultsWindow.

The App class

This is where the main window of the GUI is set up. All frames within the GUI are set up using a grid. How all the elements work and how to add those are mentioned in the Custom Tkinter documentation.



Cyber Security Simulator

Simulator settings

Network selection preview:

Select a network: network1

Set the number of simulations: Number of simulations

Set the simulation run time: Simulation time

Set the output folder name: Folder name

Manual?

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.

Appearance Mode: Dark

UI Scaling: 100%

However, there are some functions within the App class which need explanation:

- **update_network_entry()**
  This is the callback function for the network selector. Whenever the chosen network changes this function will be called and loads the corresponding image from the "basic_networks/" directory.

- **reset_results()**
  This is the function which gets called before starting a new simulation, this way all the existing data will be wiped and a clean simulation can be run.

- **start_event()**
  This function starts the simulation. First, it will set up the logs and check for edge cases. Then it will run the simulation and after that, it will finish with writing the output to the logs.

- **set_attackers()**
  This function will set up the number of attackers which have been filled in the GUI. The function uses a for loop to create frames for each attacker and within that all the options it provides. Those options are then put in a list and appended to: “glob.attacker_list”.

- **check_edge_cases()**
  This function is made to check if the user has filled the GUI with the correct values. If this is not the case, then return True and the values are not stored. If the user does provide the valid input values get stored and the simulation can be run.

The ResultsWindow class

The results window is a top window displaying the results after simulation. This window does not use any new techniques. However, there is one function:

- **is_mult_runs():**
  This function will show whether the average is taken for the results or not.

Event handler

The event handler is made with the Simpy library and has its own in-detail documentation of how to use it: https://simpy.readthedocs.io/en/latest/

After the graphical user interface, we move towards the event handler as the start button of the simulator runs the event handler function called **start_simulation()**. This main function consists of multiple parts which I will elaborate on.

First, we have the **generate_network()** function which will get the chosen network from the 'globals.py' file and then generate the network according to that value.

Followed by the network generation we have the creation of a Simpy environment.

```
>  env = simpy.Environment()
```

This environment is one of the core components of an event-based simulation. The environment is also the place where agents such as the attacker(s) and the defender, also known as processes, will interact with each other through events. The passing of time is simulated by stepping from event to event, meanwhile, the environment keeps track of the current simulation time which starts at 0, which is set by default.

When the environment is set up we will follow by **generating the attackers**. I have done this by retrieving the attacker settings list from the 'globals.py' file and then looping over it for each item in the list and creating an object of the Attacker class for each iteration, when creating an attacker we need to give it an environment, network, attacker settings, and id as its parameters. This attacker object is then added to the attacker list in the 'globals.py' file. When this is done the attacker object is called as a process in the environment we created before. This is a crucial step as it ensures the attacker will run as a process in the environment and are able to communicate with the defender process which will be generated next.

For the **defender generation**, the procedure does not differ much from the attacker generation, however, there is only one defender in the environment. This defender is an object from the Defender class and when creating this object it is called with an environment, network, and defender settings.

Finally, the **event handler will request all the scores** and costs from the network, defender, and attacker(s). This way these values can be stored and viewed by the user after the simulation. This section is divided into two parts one section will handle a single run and the other will handle multiple runs. The difference between the two is that the part for multiple runs will add the values of each run onto the values which are already stored in the global variables, so when shown in the results window the average of the number of simulations can be taken. Whereas in the single run, it only stores them once. After that when the simulation is ending the plots are drawn and saved to the results folder, so the GUI can access them for the results window.

The attacker

The **Attacker class** starts with initializing all the attributes. When all the necessary attributes have been initialized we move on to the **run()** function. This function is the generator function of the Attacker class, this means that this function will remain running throughout the simulation and generate events for the attacker.

However, before we enter the infinite loop the attacker will **load the attacks** that have been made available for the attacker in the GUI. This is done by the load actions() function. This function will read the values from the attacker strategy and attacker settings which will tell whether to load the stored actions and data from the global variables in globals.py or not. The loading part is done by simply appending the actions in the actions dictionary of the attacker.

When the attacker's actions are loaded the infinite while loop begins, in this while loop the attacker checks which attack strategy he should follow and then starts the process for that strategy. The three strategies I have implemented are a random strategy, a zero-day exploit strategy, and an advanced persistent threat (APT) strategy.

Subnet scan

First, we have the **subnetscan()** this function will scan for hosts which are connected directly to the host the attacker is currently positioned on. The attacker keeps track of which node he is on by saving the address in the start attribute. To start the scan the attacker first needs to wait for the appointed duration of the subnet scan action which can be found in the self.actions dictionary. After the waiting duration is completed the attacker can add the host addresses which were found by the subnet scan to its attribute list called scanned hosts. The process of finding the hosts which are directly connected can be done by calling the **reachable_hosts()** function which is imported from the Network class.

Exploit

The next action is the **exploit()** action, this action enables the attacker to exploit a vulnerability and then grant access to another host. In other words, when an attacker wants to move over an edge to another host the attacker needs to use an exploit action to achieve that.

Inside the exploit() function we begin with selecting an exploit from the possible exploits which are stored in the actions dictionary. The way I select the exploit is by looking for the exploit with the lowest cost from all the available options. The available options are found by calling the possible exploits function on the edge we are trying to travel through to access the next target host.

This operation has a duration of 5 seconds and after waiting for that long the attacker will try to use the exploit, but it is possible for the defender to mitigate against the exploit in the meantime and therefore render it useless. This is a possibility due to the separated actions of the defender and attacker which then can cause a state to alter if finished before the other. When this scenario does not happen we move on to the part where I check if the exploit is still effective and if the probability of said exploit is high enough to work. If both requirements are met then the attacker moves to the target host and appends the address of that host with its access level (that is set to 0 by default) to the compromised host's list in the following tuple form: (host address, access level).

On the other hand, when the requirements are not met, the exploit will fail and leave a trail of a failed exploit in the network. This is done by adding the edge to the list of failed att edges that are found in the network. All these steps are then followed by a log message containing the success or failure of the exploit and the address of the edge, the exploit was run on

Privilege escalation

At last, we have the **privilege_escalation()** action, which is used to escalate the access level of the attacker in the given host. The start of this process is somewhat similar to the exploit, to begin we look for the privilege escalation with the lowest cost between all the available options.

After that, the attacker will try to apply the privilege escalation on the selected host. For the privilege escalation to succeed, there should not be a mitigation against it by the defender and the probability should be high enough to succeed. When the privilege escalation succeeds it will change the tuple in the compromised host's list by increasing the access level by 1 level, but the access level is capped at the root level. However, if the privilege escalation fails the process is exactly the same as with the exploit action I explained before. To end the privilege escalation, a log message containing the success or failure of the exploit and the address of the host will be sent to the logger.

Attacker actions

Every action inherits from the base class: **Action class**, which defines some common attributes and functions. Different types of actions are implemented as subclasses of the Action class.

Attacker action types implemented:

- :class:`Exploit`
- :class:`PrivilegeEscalation`
- :class:`SubnetScan`
- :class:`OSScan`
- :class:`HardwareScan`
- :class:`ProcessScan`
- :class:`ServiceScan`

The base action class for Attack actions is made with the following attributes. To add another attribute you can add it here if you want all actions to have this attribute as well.

Attributes
----------

- name: str
  The name of the action
- target : (int, int)
  The (subnet, host) address of the target of the action. The target of the action could be the address of a host that the action is being used against or could be the host that the action is being executed.
- cost: float
  The cost of performing the action.
- duration: float
  The time it takes for the action to finish.
- prob: float
  The success probability of the action. This is the probability that the action works given that its preconditions are met.
- req_access: AccessLevel,
  The required access level to perform an action.

If you want to add attributes only used for one action you can add it in the class of that action with the initialisation.

Network

The network is a graph with hosts and the edges. The network keeps track of all the properties of all the hosts and edges in the network.

Host

The Host class stores all information about the host. It has a lot of get functions that return a property of the host. Besides that, Host also has a harden function that adds a hardening to the host. The other interesting function is possible_attacks(). This function determines which privilege escalations can work on the host. This is done by looking at the process of the host and then comparing them with the targets of all the privilege escalations. Only the privilege escalations that the host is not hardened against are returned. It is not realistic that the defender or the network always knows all possible privilege escalations. A solution to this could be to add a separate array in globals that contains the privilege escalations the defender knows.

Access_for_score, hardware, services, and os are properties of Host that are currently not being used. Access_for_score indicates the access level needed to get the important information and thus the score of the host. We decided that the information is always obtained at the admin level, the admin level is level 2. We thus do not use access_for_score anymore. Hardware, services, and os are properties that were intended as possible targets for privilege escalations. The privilege escalations now only target processes. Hardware, services, and os are thus not used.

Edge

The Edge class stores all information about the edge. It is similar to Host, but with services allowed instead of processes and exploits that target it instead of privilege escalations. Edge has a lot of get functions that return a property of the edge. Besides that, Edge also has a harden function that adds a hardening to the edge. The other interesting function is possible_exploits(). This function determines which exploits can work on the edge. This is done by looking at the services allowed on the edge and then comparing them with the targets of all the exploits. Only the exploits that the edge is not hardened against are returned. It is not realistic that the defender or the network always knows all possible exploits. A solution to this could be to add a separate array in globals that contains the exploits the defender knows.

The firewalls are the services_allowed property of the edge. The services allowed indicate which services the destination host accepts via this edge. All the other services are blocked by the firewall of the destination host. Realistically edges do not have firewalls or hardenings. The host has firewalls and hardenings, but we moved them from the host to the edges since different incoming edges can have different firewalls and hardenings.

Network

The Network class has a few properties. The first two are hosts and host_map. Hosts is an array filled with objects of the Host class. Host_map is a dictionary that takes (subnet address, host address) as a key and outputs the place in the array hosts of the host with the corresponding subnet and host address. This is useful, because the following two properties of Network work with the place of the host in the array hosts: edges and adjacency matrix.

Edges is a dictionary with objects of the Edge class. The key is (place of source in hosts, place of destination in hosts). The adjacency matrix is used to quickly see which edges exist. The matrix is filled with 0’s and 1’s. The 0’s indicate that the edge does not exist, while a 1 means that the edge does exist. We use a directed adjacency matrix. An edge from host 1 to host 2 does not mean there is an edge from host 2 to host 1. The rows of the matrix show all the outgoing edges of the hosts. The first row indicates which outgoing edges the first host in hosts has. The columns of the matrix show the incoming edges of the hosts.

In the following adjacency matrix, it can be seen that host 1 has incoming and outgoing edges to both host 2 and 3. There is also an edge from host 3 to host 2.

$$\begin{array}{cc} & \text{Destination} \\ \text{Source} & \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix} \end{array}$$

Sensitive_hosts is an array with the addresses of the most important hosts. These are the hosts that contain the most sensitive information and thus have the highest score. The simulation stops if the attacker manages to compromise all the hosts in Sensitive_hosts. The simulation stops otherwise if the run time is up. Sensitive_hosts is also used in some defensive strategies to prioritize the important hosts.

Failed_att_hosts and failed_att_edges are empty arrays that will be filled when attacks fail. The target of the failed attack will be put in one of these arrays, depending on whether a host or an edge was attacked. The defender can periodically check these arrays to see if any failed attacks happened. The defender takes action after a failed attack is noticed. The defender also empties failed_att_hosts and failed_att_edges if any action was taken.

Adding to the network

Hosts and edges can be added to the network. When adding a host, the host is added to the array hosts in the network. The place of this new host is stored in the dictionary host_map with the address of the host as the key. The adjacency matrix is increased from a NxN matrix to a N+1xN+1 matrix. The new host does not have any edges. An edge is added by changing the correct value of the adjacency matrix from a zero to a one. The edge is also added to the dictionary edges of the network.

We do not check if an edge already exists and we also do not check if the address of a host is unique. It is thus important to make sure as the programmer that this is actually the case when adding hosts or edges.

Removing from the network

The network does not support removing a host or edge. Removing an edge could be done by setting the corresponding value in the adjacency matrix from 1 to 0. The edge must also be removed from the edges dictionary. Removing a host is more complicated. The host must be removed from the array hosts. The host must be removed from host_map. The corresponding row and column of the adjacency matrix must be removed. All edges with the removed host as source or destination must be removed as well. The removal of a host in the array hosts means that all hosts further in the array now have a different place. Their places have decreased by 1. This means that the values of the dictionary host_map and the keys of the dictionary edges must be updated.

Creating a network

The first step in creating a network is to create a Network class object. This object starts with one host. This host has the address (1, 0) and represents the internet. Then, other hosts and edges can be added to the network. These hosts and edges represent the network that is attacked in the simulation. The sensitive hosts are then selected. The score and processes of a host are selected during the creation of that host. The same applies to the services of an edge.

We use the networkx library to generate graphs. We then use these graphs to create a network based on the topology of the generated graph. The main problem is that the generated graph does not have directions, does not have scores and is not connected to the internet host. We make all the edges generated multidirectional to solve the directions problem. We use a seed when generating the network to always get the same graph with the same parameters. This allows us to distribute scores and connect the internet to the network in a way that is logical. This is done in the creat_small_world() and create_power_law() functions.

Drawing the network

A network can be drawn with the draw_network() function. This function uses networkx to generate a directed graph based on the adjacency matrix of the network. The hosts of the network are then drawn multiple times based on their properties. The order is important, since the last color is the one that is seen. We first draw all hosts in orange, the color of normal hosts. Then the hardened hosts are drawn in blue. The hardened edges are colored blue as well. Afterwards, the hosts that are compromised at the user level are drawn in red. Lastly, the hosts that are compromised at the admin level are colored maroon.

Defender

One of the properties of the defender is the network. The defender makes changes in the network to prevent the attacker(s) from stealing important information. Another property of the defender is the strategy. The strategy determines which actions the defender will use to defend the network. The cost is the total cost of all the actions taken by the defender. The cost is used in the evaluation on how well the defender performed. The env is the Simpy Environment and the connection to the time. All functions that take time in the simulation need to be called with **yield self.env.process(*function*)**. This allows function to use **yield self.env.timeout(*time*)** to have the defender wait for a time in order to simulate the time that an action takes.

Failed_att_hosts and failed_att_edges are arrays that start empty. Failed attacks on hosts and edges will be stored in these arrays over the course of the simulation. The information of which hosts or edges were under attack can be used to estimate the position of the attacker(s). Targeted defenses can be set up if the general location is known. We do not have any strategy that takes advantage of this information. This is a subject for future research.

The other properties of defender end with _allowed. Those properties indicate which options the user selected in the GUI. At least one of harden_host and harden_edge must be allowed. The current strategies do not work otherwise. The actions scan, update host, and update firewall are not being used. Thus scan_host_allowed, update_firewall_allowed and update_host_allowed are never used as well.

The run() function is the place where actions are taken based on the strategy of the defender. This is done with infinite while loops. The infinite while loops are not a problem in this case, since the simulation stops when the time is up. New strategies can also be added in the run() function.

Random hardening

A few strategies make use of hardening a random host or edge. This is done in the random_defense() function. We achieve this by picking a random host or edge and then fully hardening that host or edge. It could be that the selected host or edge is already fully hardened. In that case, a random host or edge is selected again. This process is repeated until a target is hardened. It could be that all hosts and edges are already fully hardened. This way of hardening a random host or edge will then result in an infinite loop. This loop does not include any **self.env.timeout(*time*)**, since nothing is ever hardened and thus no time is taken. The simulation never ends in this case. To prevent this, a small time out is done if fully hardening a host or edge fails. The time outs can be found in the fully_harden_host() and fully_harden_edge() functions. These time outs ensure that the time keeps going, even if nothing can be hardened anymore.

An alternative to our way of random hardening is to keep track of which hardenings are not done yet. The random hardening is then picked from among those hardenings. It would then also be clear when no further hardenings can be done. Thus the small time outs become unnecessary.

Defender strategies

The first strategy is **random**. This means that the defender randomly picks a host or edge and hardens that host or edge as much as possible. The probability of hardening a host or edge are equal. The user can choose to not harden hosts. In that case, the random strategy only hardens edges. In the same way, only hosts are hardened if edge hardening is not allowed.

The second strategy is **last layer defense**. The network has an array with the most sensitive hosts. These are the hosts that have the highest score. This strategy prevents the attackers from getting the score of the most important hosts. The first step is to use host hardening on these hosts to prevent privilege escalation. There could be privilege escalations that the defender cannot defend against. The attacker could then still use those privilege escalations to get admin access and thus the score. To prevent that, all edges that go towards the sensitive hosts are fully hardened at the second step of this strategy. This makes it harder to actually get to those hosts. The order in which the sensitive hosts are hardened is based on score. The host with the highest score is hardened first. The edges of the host with the highest score are hardened first as well. The random strategy is used after hardening all the sensitive hosts and their incoming edges.

The third strategy is **minimum defense**. This defense only takes action when an attack fails. The network then puts the targeted host or edge in a special array to indicate that an attack happened. This defense strategy periodically checks that array to see if any attacks failed. An attack can fail if the probability of the attack is not one. An attack can also fail if the targeted host or edge was hardened during the attack. When a failed attack is noticed, it is most likely that an attacker is trying to get access to something that is not yet compromised. Realistically it could be a false alarm, but I assume that every failed attack actually is a failed attack. The defender fully hardens the host or edge that was the target of the failed attack. It then waits until another attack fails.

The fourth strategy is **reactive and random** and is a combination of the first and third strategies. It does random hardenings until a failed attack is noticed. It then fully hardens the target of the attack. It goes back to random hardening afterwards. The checking for failed attacks is only done in between hardenings. This strategy is thus a bit slower to react to a failed attack.

The last strategy is **highest degree neighbour**. This strategy first picks a random host in the network. It then takes the neighbour of this random host that has the highest degree. The degree is calculated by adding the number of incoming and outgoing edges together. An edge between host A and B that goes both ways is thus counted twice. The same process is then repeated on the neighbour with the highest degree. The degrees of all its neighbours are calculated and the best is chosen. There are now two best hosts as a result from the two rounds. Either one of the two hosts will be fully hardened or an edge between the two hosts will be fully hardened. This depends on whether host and edge hardening are allowed. It is chosen randomly if both are allowed. A completely random host or edge is fully hardened if the target chosen by this method is already fully hardened.

Actions_def

Actions_def.py contains 6 classes. One base class Action_def and five classes that use that base class as their base. The five classes are the five actions that the attacker can do. They are Harden_host, Harden_edge, Scan_host, Update_host, and Update_firewall. We use classes for each action because there could be multiple types of each action. There could be six harden_host actions for example. Each with its own name, cost, duration, and attack_type. Classes are a good way to store that information.

The classes Harden_host and Harden_edge have the exact same properties and functions. The same is true for the classes Update_host and Update_firewall. It is thus possible to merge them together into one class. We did not do this, because the classes might diverge with new features. It turns out we never added any features that caused the classes to be different, but we kept them separated anyway.

The Scan_host, Update_host, and Update_firewall classes are never used, since no strategy uses any of those actions.

Plot_log

The draw_plot() function generates the figure that shows the scores of the defender and attacker(s) over time. This is done based on the data in score_log.txt. The log_scores() function in the event handler periodically writes the scores of the defender and attackers to the score_log.txt. This is an example of possible lines in score_log.txt:

```
2023-06-14 15:23:17,427 154 Defender damage 120 actions cost 15
2023-06-14 15:23:17,428 154 Attacker0 score 10 actions cost 12
```

The lines follow the format:
Real time, **run time**, **role**, “damage” or ”score”, **score**, “action cost”, **cost**

The total score of the defender is calculated as: total score = -1 * (score + cost)
The total score of an attacker is calculated as: total score = score - cost

All the lines of score_log.txt are read and processed. The total scores and the run time are both stored in a separate dictionary. The key used for both dictionaries is the role.

The results are then averaged if multiple simulations were done. Lastly, the figure is created and saved in the chosen output folder under the name Plot_fig.png.

Future work network and defender

Adding more characteristics to the network can be done by adding properties and functions to the Host and Edge classes.

Adding a strategy to the defender can be done by adding a function for that strategy to the Defender class. The new function must also be added to the if-statements in the run() function of the defender class. The GUI must also be changed to make it possible to pick the new strategy.

More host and edge hardenings can be added in globals. Add the new host hardenings to the array hard_h and the new edge hardenings to the array hard_e. More privilege escalations and exploits can be added in a similar way with attas_h and atts_e.

Complete new defensive actions can be added by creating a new class in action_def.py. The new class needs to hold all the necessary information to perform the action. A new function is then written in the Defender class to actually perform that action based on the information of the newly created class.

Al lot of things need to be added to introduce DoS into the simulator. The Host class needs more properties to indicate if it is under a DoS attack and what the loss is for the defender when the host is under a DoS attack. All calculations for both the attacker(s) and defender need to be changed to include the score gained or lost by DoS attacks. New actions are needed to prevent or solve DoS attacks. The current strategies need to be changed or new strategies must be added that take DoS attacks into account.

Events can be created in the log_score() function of the event handler. This function periodically writes data to a file. It thus uses a while loop with a stable interval. An event can be created by putting an if-statement inside the while loop. The if-statement will be correct at a certain time and the event will trigger at that time. An example can be found in the comments of the log_scores() function in the event handler.