SecureInT Visualizer Implementation & API (backend) documentation

21/07/2017

By Saket Khandelwal

NUS school of Computing

1. Introduction

1.1 Scope

SecureInT is a discrete-event simulation model to investigate, analyze and secure insider threats. In order for users to use this model we need a middleware. Thus, a web visualizer was proposed to visualize and to utilize the properties of the simulation model in which users will be able to interact with the model through this interface.

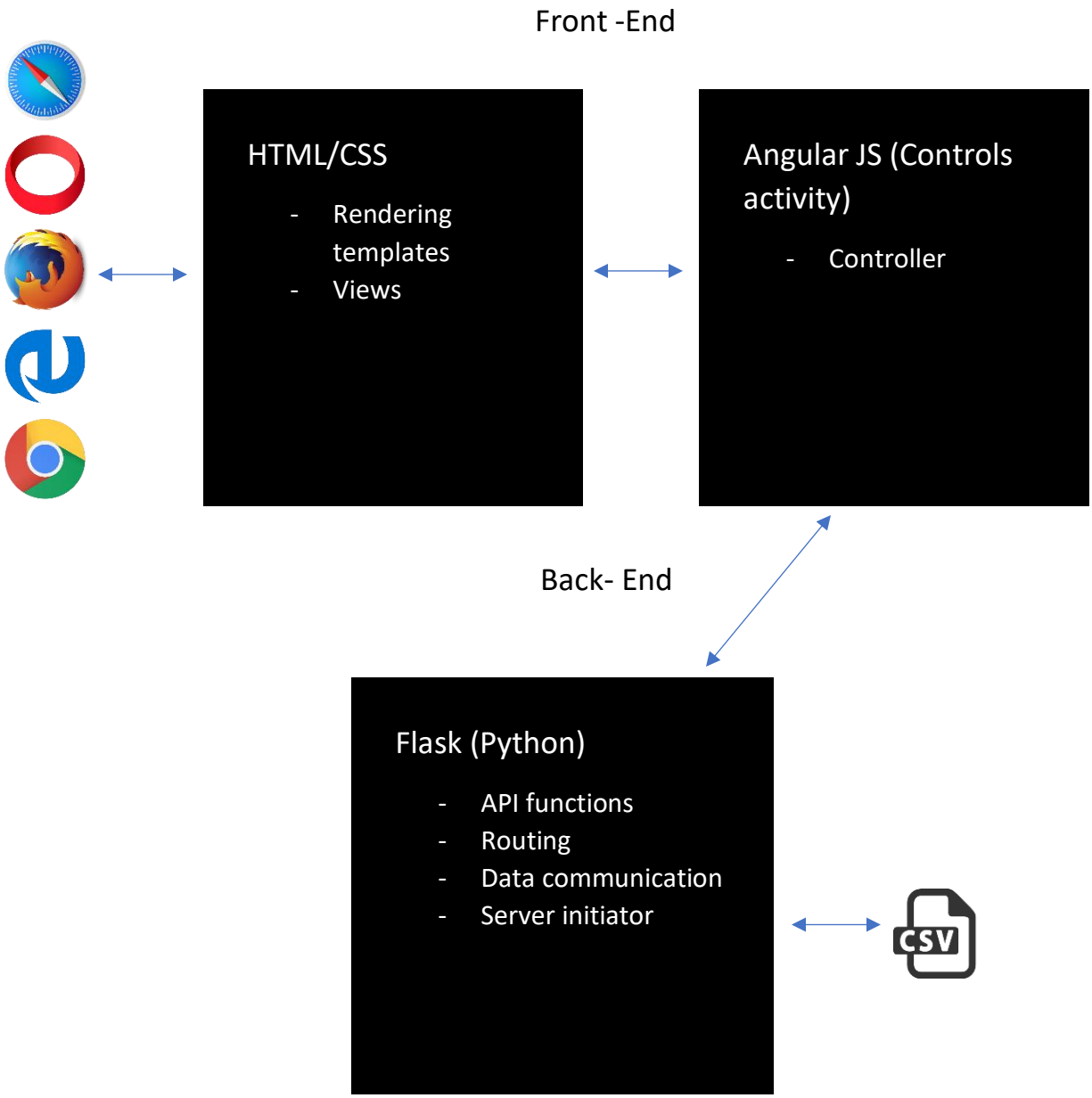
The final scope is to design an interface such that a user can visualize the network, the users and the interactions. The user should be able to tweak parameters like vulnerabilities for the hosts and connections between users. Furthermore, the user should be able to "play" the simulator and view metrics like the vulnerability for the overall system. The frontend should be designed in such a way that it is easy to plug in any simulator (API for backend in Python).

1.2 Overview

The visualizer is developed as a web interface in which user can interact with the interface from anywhere. The interface is implemented using web programming tools and python framework as a server-side handler. This project is implemented using HTML, CSS, JavaScript and Python. Flask is the python framework used in order to maintain and join the back-end with the front-end. Angular JavaScript is used to implement routing and scope to bind the controller with the view (HTML).

2. High Level Description

2.1 Architecture Diagram



2.2 Architectural Diagram Description

2.2.1 Front-End

The views are implemented using HTML and CSS which render the templates. The controller is implemented using JS and Angular JS binds the view and the controller. As the user input is analyzed by the control and then views are manipulated by the controller. The front-end takes care of the visualization part as the network graph and simulation graph are created using D3 library (JavaScript).

2.2.2 Back-End

The back-end or server-side is implemented using Python on the Flask framework. The back-end does the routing of the pages, creates API functions for external simulators and manipulates the data stored. The controller uses JS to trigger API functions which call the python functions in the back-end ergo flask is consumed by the controller.

3. Regulations and Instructions (Input, Usage and API)

3.1 Input Explanation and Instructions

The input to the system is to be log file of the network. This input is then visualized in the network panel where users can edit and add new nodes or links. When the simulator is triggered it is passed to the simulator as the input.

The input has to be a JSON type log file of the network and follow the format as described below.

1. The file has to be distributed as nodes and links differentiated from each other.

Ex.

{“nodes”: [],
“links”: []}

2. There are two types of nodes (user and machines) each should be defined in this format.

Machine Node

```
{
    "id": "172.16.40.24",
    "label": "work-A",
    "properties": {
        "type": "machine",
        "vulnerability": 0.9,
        "CVE" : "1-CA"
    }
},
```

User Node

```
{
    "id": "u1",
    "label": "Alice",
    "properties": {
        "type": "user",
        "vulnerability": 0.2,
        "hasacc": "work-A"
    }
},
```

3. There are two types of links (users and machines) as well and should be defined in the following format for the system to be able to read.

Machine Link

```
{
    "source": "172.16.40.24",
    "target": "172.16.40.34",
    "properties": {
        "protocol": "HTTP",
        "port": 80
    }
},
```

User Link

```
{
    "source": "u1",
    "target": "u2",
    "properties": {
        "leakage" : 0.2
    }
}
```

3.2 Usage Instructions and Explanation

The main webpage is divided two pages as home (main simulator) and about. The Home page has three sub panels.

1. Upload Panel: User to upload log file.
2. Network Panel: User to visualize graph and edit properties of the graph.
3. Simulation Panel: Users to initiate the simulation and analyze the results.

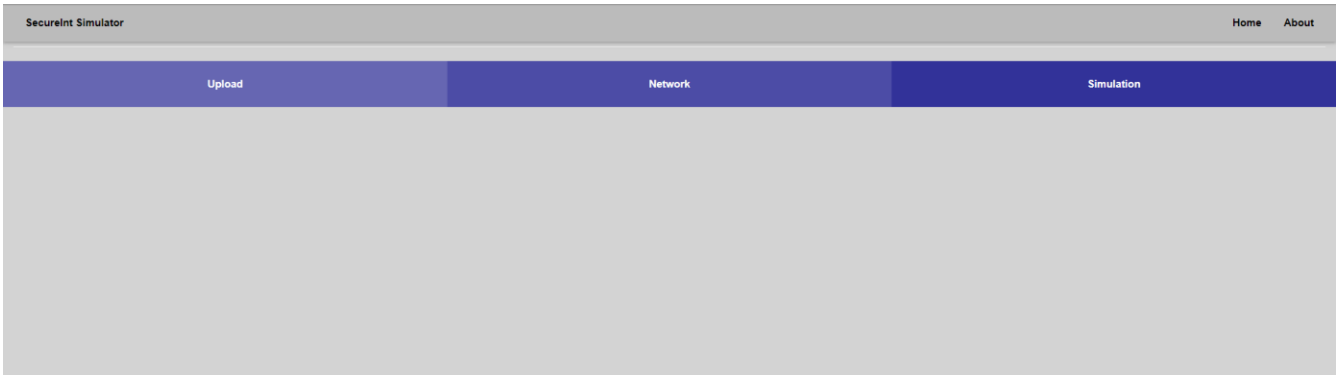


Figure 1 Home Page

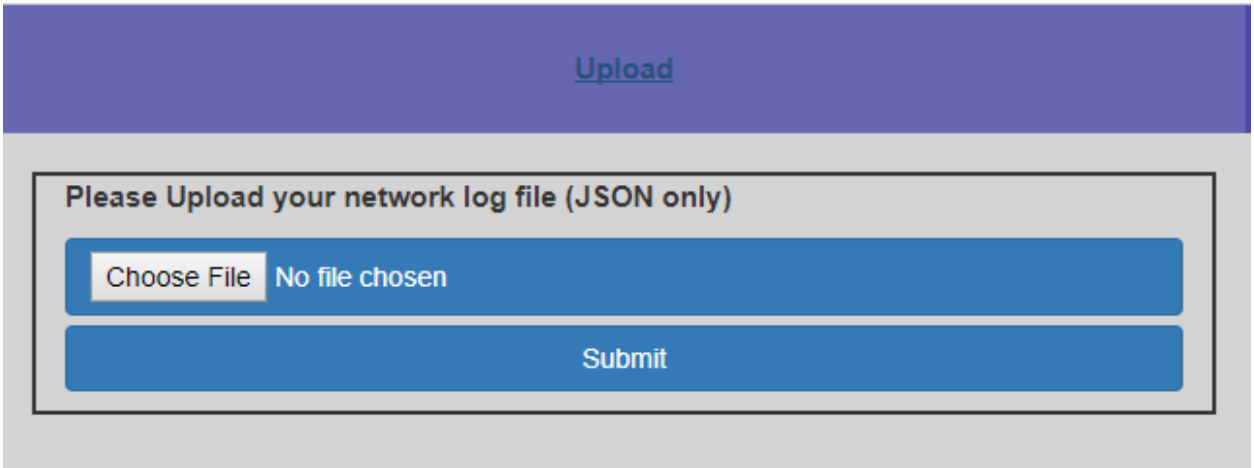


Figure 2 Upload Panel

As you can see in the figure 2 above this is the upload panel where users should upload a JSON network log file in the format provided in the Input section.

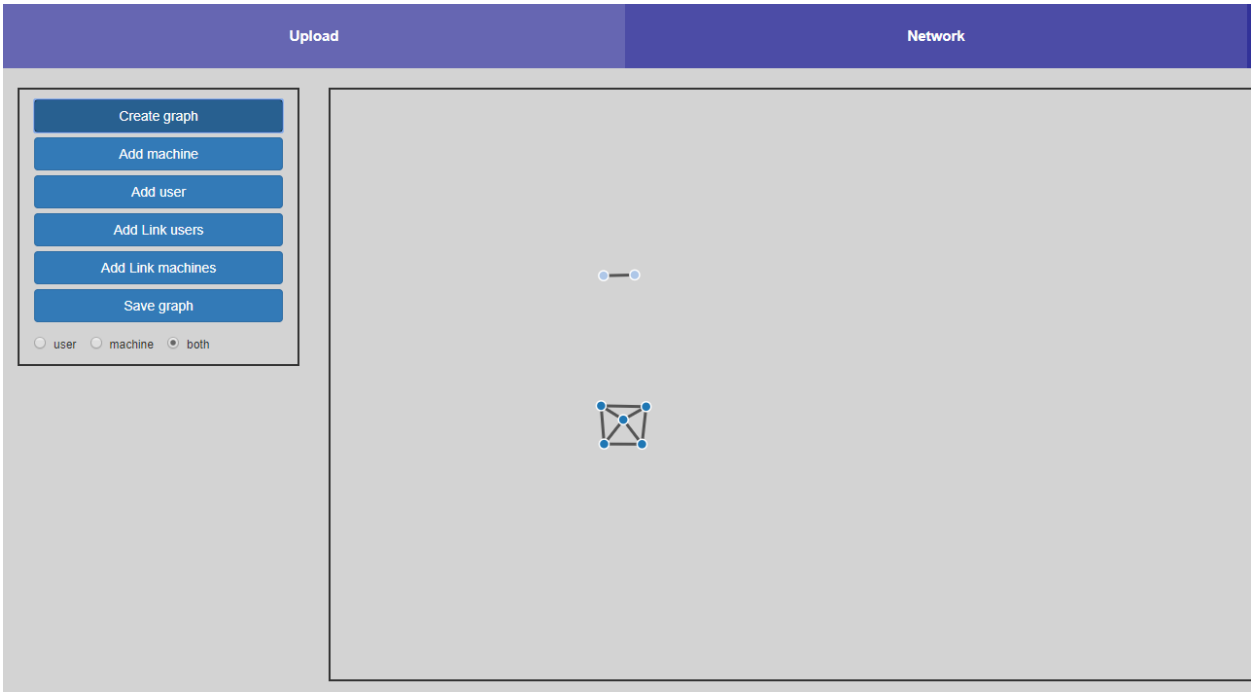


Figure 3 Network Panel

The panel above (Figure 3) is the Network panel where user can initiate and view the network log as a network graph when pressing the button (Create graph).

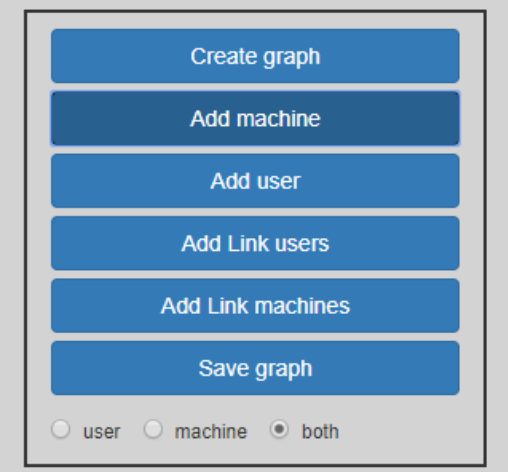


Figure 4 Network Panel Menu

This is the network panel menu (Figure 4) in which user can choose as per their requirements to edit their network graph. The buttons are explained below.

1. **Create graph**: Initialize and draw the input log file as a network graph.
2. **Add Machine**: Add a machine node to the graph (form with properties to input).
3. **Add User**: Add a user node to the graph (form with properties to input).
4. **Add Link User**: Add a link between two users (form with properties to input).
5. **Add Link Machine**: Add a link between two machine nodes.
6. **Save Graph**: Save the changes added or edited to the graph as a final file for the simulation.
7. **Radio buttons**: To filter the network graph depending on the node type.

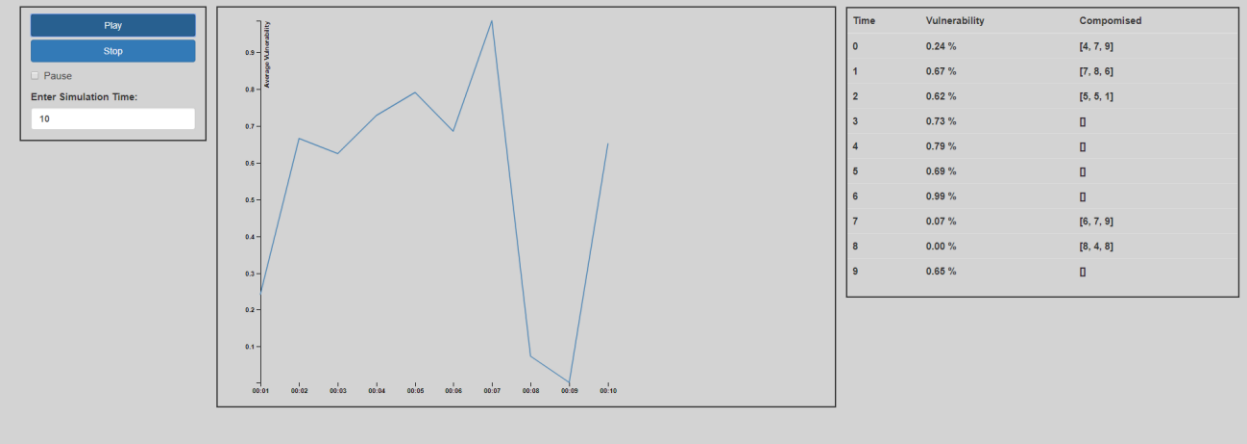


Figure 5 Simulation Panel

The figure above is of the simulation panel. The user initiates the simulation by clicking the Play button. The input file will be passed to the simulator in the backend which will then return the values for the graph. User can edit the simulation time and can pause the simulation by ticking the Pause radio button.

3.3 API instructions and Usage (Back-end Python)

The API functions are designed so any python simulator can be integrated to the backend by using these functions and connect to the front-end functions. The API functions implemented are as explained below. **The simulator function should be imported into simulationStub.py file and be added to the code for passing and using the functions.**

1. **Input File**: This is the file which user has uploaded (network log file). Simulator will use this as an input.
2. **Simulation Time:** This is the function for passing the simulation time input by the user to the simulation.
3. **Start Simulation:** Function for toggling the simulation to running state.
4. **Pause Simulation:** Function for toggling the simulation to pause state.
5. **Stop Simulation:** Function for toggling the simulation to stop state.
6. **Return Data:** This is the API function for returning data from the simulator to the system.

The functions usage instructions will be described in the following page.

1. **Input File**

```
# Add stub api for input file pass to simulator
with open("./static/raw.json") as json_data:
    jsonGraphData = json.load(json_data)
    print jsonGraphData
```

Figure 6 Input File API

This is a stub function which shows which file to pass as the input to the simulator. When the user uploads the log file the file is parsed and read and saved as raw.json which is passed as shown above. When passing the file to the simulation this can be passed.

2. **Simulation Time**

```
def start(maxTick):
```

Figure 7 Simulation Time API

This function is defined in the starting and the simulation time which is received as input from user is stored in the variable called “maxTick” as shown in figure 7. So, this variable needs to be passed to the simulator.

3. **Start Simulation**

```
def start(maxTick):

    # Add stub api for input file pass to simulator
    with open("./static/raw.json") as json_data:
        jsonGraphData = json.load(json_data)
        print jsonGraphData


    # toggle to running state
    with open('./static/state.json', 'w') as fp:
        json.dump({"state":"running"}, fp)
```

Figure 8 Start Simulation API

The def start functions is triggered when the play button is pressed. The simulator function can be called here where it passes the input file and toggle the state of state.json to running.

4. **Pause and Stop Simulation**

```
for i in range(maxTick):

  # handle current state
  with open("./static/state.json") as json_data:
    state = json.load(json_data)
    print data

    if state["state"] == "done":
      exit()

    if state["state"] == "pause":
      while True:
        isPause = True
        time.sleep(1)
        with open("./static/state.json") as json_data:
          state = json.load(json_data)
          if state["state"] != "pause":
            isPause = False
        if (isPause == False):
          break
```

Figure 9 Pause and Stop Simulation API

As the state of the simulation is handled in the file called state.json the file should be called in order to read and store the state of the simulation. For the pause in between the simulation, the simulator function to stop the simulation can be added after the '*state["state"] == "pause"* 'where the function to stop the external simulator can be triggered and the function to unpause the simulator can be triggered after the *isPause = false*.

When the stop simulation is triggered for adding the simulator function to stop the simulation it can be added after '*if state["state"] == "done" '*.

5. **Return Data**

```
11 def getVuln(i):
12     value = random.random()
13     print value
14     print i
15
16     customCompromised = []
17     for _ in range(0,random.choice(range(4))):
18       customCompromised.append(random.choice([1,2,3,4,5,6,7,8,9]))
19
20     return {"vuln": value, "compromised": customCompromised}
21
```

Figure 10 Return Data API

This function is a stub function for writing data to a file in order for visualizing as a live graph. This function in figure 10 has an argument passed which is the current time of simulation in which the function passes the vulnerability value and compromised array from the function. In order to integrate the simulator data to write the simulator function should be called here and return that data out of this function.

4. Non-Functional Requirements

4.1 Operating Constraints

This project is built using the Flask framework. Operating requirements are listed below.

Python: 2.7.x
Flask: 0.12.2
OS: any that supports flask.
Browsers: any

5. Future Development and Future Work

This project can be developed further as per the requirement of the simulator plugged in. As scope is used for the implementation of this project it is scalable.

Future Work:

- Usable for both python versions.
- Live attack graph on the network panel with the compromised entities.

6. Conclusion

SecureInT Simulator is a web interface where the user can upload their network log file and visualize and read their network through a network graph. Users can also choose to edit the properties and add changes to their network. Users can play the simulation in order obtain and view the results of the simulation on their network.