Abridged Cybersecurity Process White Paper With Full Code Listing

Miles Higman

June 2024

Abstract

This whitepaper presents a process and a code implementation of a way to weigh and prioritize cybersecurity attacks by treating the attacks as nodes and connecting attack nodes with edges with variable weights, allowing cybersecurity professionals to prioritize attack patterns depending on their system.

1 Definitions

1.1 Defining a node

In this context, a node is a data structure representing a specific attack and the properties of the specific attack. Here is a simple list of the attributes of each node:

- Name - The name of each node is the name of the general class of attack that node represents. These names can be from any type of threat modeling framework. However, this process and generalized program was made with the MITRE ATT&CK framework in mind
- Layer - The layer of a threat is an optional attribute representing where a certain threat is in a layered threat model and its index relative to the first layer. Giving MITRE ATT&CK as an example, "Reconnaissance" would be layer 0. This attribute constrains what a given node can connect to, as nodes on the same layer shouldn't be connected (after all, they are independent methods of achieving a similar goal).
- Likelihood - This attribute is the likelihood this event will occur, assuming the attack reaches a node connecting to this node. There is one precondition to this variable. However, this variable has an extra caveat on a layered model. In general, if two or more independent nodes complete similar tasks, the sum of their likelihoods should be 1.

- Protection - This attribute is simply the probability that a network's protection will stop the attack if a node's associated attack happens. This variable is always approximated, as certain threats like 0-day vulnerabilities can't be known to measure protection accurately. Also, many attacks may not get to certain nodes, making the value of this variable more uncertain. However, it is important to give some estimates via defense evaluations.
- Severity - This attribute is a quantity bounded between 0 and 1 that quantitatively measures the repercussions where the attack each node describes were to occur. This data can be collected from statistical modeling or some type of ML program that measures some sort of "impact coefficient" that can be bounded.

1.2 Defining an edge

An edge is a data structure connecting two nodes that has three attributes:

- Start - The starting node of the edge.
- End - The ending node of the edge.
- Weight - The "importance" of an edge. This value is calculated by using the properties of both the start and end nodes.

2 Customizing Edge Weights

In addition to visualizing attack graphs, we can customize the visualization to highlight important features of the graph. One way to do this is by customizing the weights of the edges between nodes. The weight of an edge can represent the severity of the attack path it represents, the likelihood of the attack being successful, or a combination of both.

2.1 Importance of Edge Weights

Customizing edge weights is crucial for prioritizing cybersecurity efforts and mitigating potential risks effectively. By assigning weights to the connections between nodes in the attack graph, cybersecurity professionals can identify critical attack paths that pose the highest risk to the system. This allows organizations to allocate resources more efficiently and focus on addressing the most pressing security vulnerabilities.

2.2 Equation for Calculating Edge Weights

The weight of an edge between two nodes can be calculated using a custom equation that considers various factors such as severity, likelihood, and protection level. The equation provides a quantitative measure of the risk associated

with each attack path, allowing cybersecurity professionals to prioritize their response accordingly.

$$
\begin{aligned}
w = f(s, l, p) = p_{\text{part}}(1 - \sigma(\frac{\partial p}{\partial t}))[\\
& l_{\text{part}} s_{\text{part}}(1 - \sigma(\frac{\partial s}{\partial t}))(1 - \sigma(\frac{\partial l}{\partial t}))+ \\
& l_{\text{part}}(1 - \sigma(\frac{\partial l}{\partial t}))\sigma(\frac{\partial s}{\partial t})+ \\
& s_{\text{part}}(1 - \sigma(\frac{\partial s}{\partial t}))\sigma(\frac{\partial l}{\partial t})+ \\
& \sigma(\frac{\partial s}{\partial t})\sigma(\frac{\partial l}{\partial t})]
\end{aligned} \tag{1}
$$

where:

$$
\begin{aligned}
s_{part} &= \begin{cases} s + (1 - s)\mu_s & 0 < s \leq 1 \\ 0 & s = 0 \end{cases} \\
l_{part} &= \begin{cases} l + (1 - l)\mu_l & 0 < l \leq 1 \\ 0 & l = 0 \end{cases} \\
p_{part} &= (1 - p)
\end{aligned} \tag{2}
$$

and:

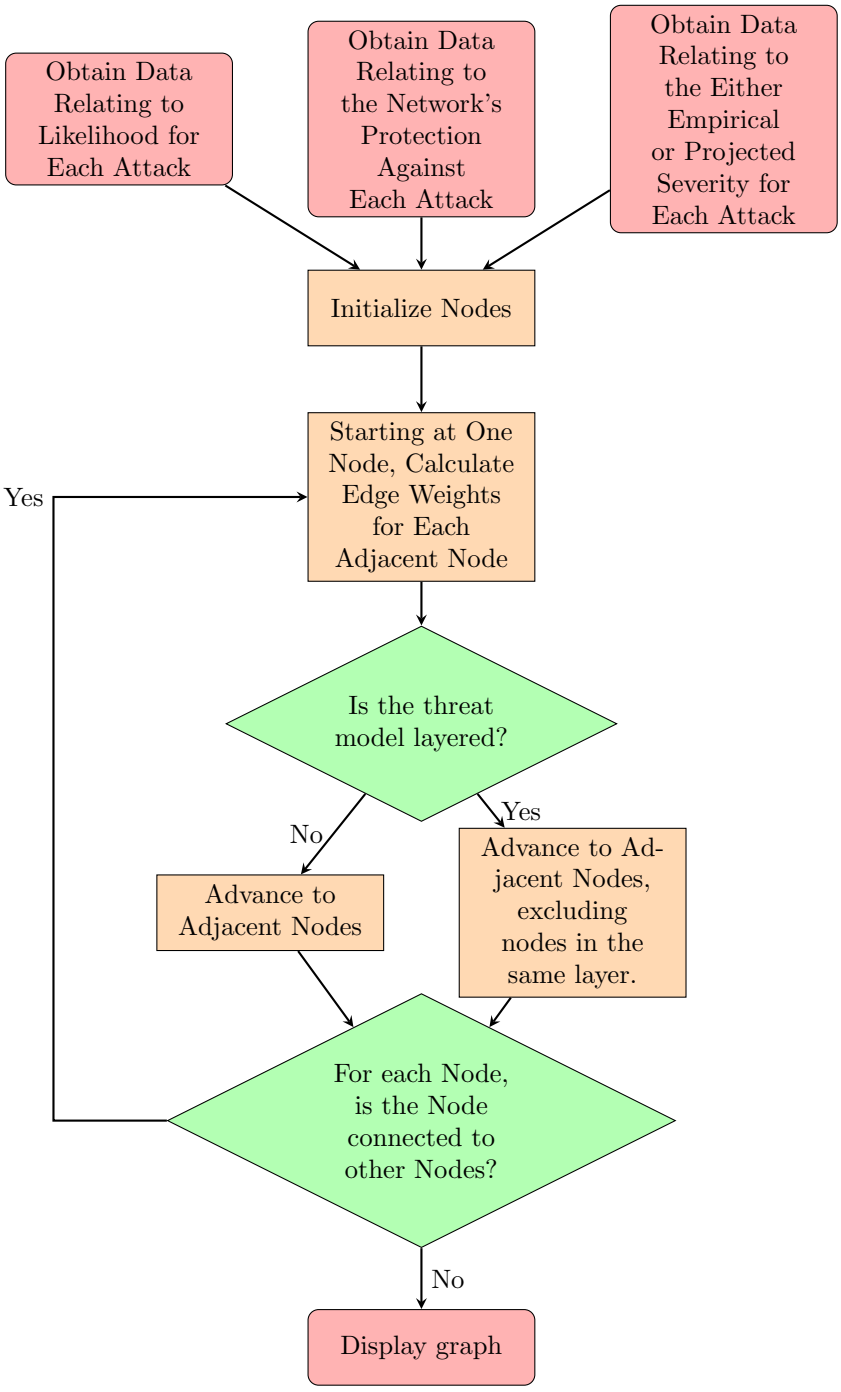
- w is the weight of the edge.
- s is the severity of the attack.
- l is the likelihood of the next attack occurring, given that the first attack occurred.
- p is the protection level of the destination node.
- f is a custom function that calculates the weight based on the given parameters.

2.3 Not Considering Derivatives

This equation can easily be modified to work with single data points (which don't have a derivative), like so:

$$
w_{const} = p_{part} l_{part} s_{part} \tag{3}
$$

3 Process of Creating a Weighted Attack Graph



4 Breakdown of Graph Generation Code

4.1 Overview

The provided code uses Dash and Plotly to generate a 3D scatter plot representing different stages of an attack with connections (edges) between them. The points on the graph represent different attack stages with attributes such as severity, likelihood, and protection.

4.2 Edge Weights

Edge weights are calculated using a custom function `calculate_weight_at_const_rate`. The weight function considers the severity, likelihood, and protection values of connected points. The formula used is:

$$w = 5 \times (1-p) \times (l + (1-l) \times \text{tolerance\_of\_likelihood}) \times (s + (1-s) \times \text{tolerance\_of\_severity})$$

where:

- s is the severity of the destination point.
- l_a and l_b are the likelihoods of the source and destination points.
- p is the protection value of the destination point.
- tolerance_of_severity = 0.5
- tolerance_of_likelihood = 0.5

4.3 Algorithmic Process of Generating the Graph

1. **Initialization:** Import necessary libraries and define the number of data points per x-coordinate and custom labels for the x-axis.
2. **Coordinate Generation:**
   - Reverse the list `num_points_per_x_list`.
   - Generate x-coordinates based on the number of data points per x-coordinate.
   - Generate corresponding labels for each stage.
3. **Data Point Generation:**
   - Generate random y-values (severity) and z-values (likelihood) for each point.
   - Normalize the z-values to ensure they sum up to 1.
   - Generate random sizes for the spheres representing data points.
4. **Connection Generation:** Create a list of connections between points where the x-coordinate of one point is one less than the other.

5. **Scatter Plot Creation:** Create a 3D scatter plot of the data points with hover text displaying detailed information.

6. **Edge Plot Creation:** For each connection, calculate the weight and create a line (edge) with a width proportional to the weight.

7. **Layout Configuration:** Define the layout of the 3D plot, including axis titles and aspect ratio.

8. **Dash Application Setup:** Initialize the Dash application, define the layout, and implement a callback function to handle click events and draw lines between clicked points.

4.4 Features of the Graph

- **3D Scatter Plot:** The graph displays data points in a 3D space with x-axis representing stages, y-axis representing severity, and z-axis representing likelihood.
- **Hover Text:** Each data point has hover text showing its name, stage, severity, likelihood, and protection.
- **Connections (Edges):** The graph shows connections between points with widths based on calculated weights.
- **Interactive Click Events:** Users can click on points to draw connections manually, with edges being dynamically added to the graph.

4.5 Code Listing

```
import dash
from dash import dcc, html
from dash.dependencies import Input, Output
import plotly.graph_objs as go
import numpy as np

# Define the number of data points per x-coordinate
num_points_per_x_list = [3, 5, 4, 6, 7, 3, 3, 4, 5, 4, 2, 2, 1, 2]
# Sample list of data points per x-coordinate

num_points_per_x_list.reverse()
# Define custom labels for x-axis
custom_labels = ['Reconnaissance',
                 'Resource Development',
                 'Inital Access',
                 'Execution',
                 'Persistance',
                 'Privilege Escalation',
```

```
                  'Defense Evaluation',
                  'Credential Access',
                  'Discovery',
                  'Laterial Movement',
                  'Collection',
                  'Command and Control',
                  'Exfiltration',
                  'Impact']

letters = 'abcdefghijklmnopqrstuvwxyz'

names = []

label_for_stages = []

# Generate x-coordinates based on the number of data points per x-coordinate
x_coordinates = []
for i, num_points in enumerate(num_points_per_x_list):
    x_coordinates.extend([len(num_points_per_x_list)-1-i] * num_points_per_x_list[-(i+1)])
    label_for_stages.extend([custom_labels[i]] * num_points_per_x_list[-(i+1)])
    for j in range(num_points_per_x_list[-(i+1)]):
        names.append(custom_labels[i] + f" ({letters[j]})")

print(label_for_stages)
# Generate random spherical data points on the x-coordinates

x = x_coordinates
num_layers = len(num_points_per_x_list)
print(x)
custom_labels.reverse()
# Initialize y-values list
z = []

# Generate random y-values for each layer
for i in range(num_layers):
    # Generate random probabilities for each data point in the layer
    probabilities = np.random.rand(num_points_per_x_list[-(i+1)])
    print(f"Stage {i}")
    print(sum(probabilities))
    # Normalize probabilities to ensure they sum up to 1
    probabilities /= np.sum(probabilities)
    print(probabilities)
    print(sum(probabilities))
    z.extend(probabilities)
print(z)
y = np.random.uniform(0, 1, len(x))
```

```
# Generate random sizes for spheres
sizes = np.random.uniform(0, 100, len(x))

print(x)

connections = []

for i in range(len(x)-1):
    for j in range(1, len(x)):
        if x[i]-1 == x[j]:
            connections.append((i,j))

attack_hover_text = [
    f"Name = {names[i]}\
    <br>Stage = {label_for_stages[i]}\
    <br>Severity = {y[i]}\
    <br>Likelihood = {z[i]}\
    <br>Protection = {1- sizes[i]/100}"
 for i in range(len(x))]

# Define scatter plot for data points
scatter = go.Scatter3d(
    name="Attacks",
    x=x,
    y=y,
    z=z,
    mode='markers',
    marker=dict(size=sizes, color='red'),
    hoverinfo='text',  # Show hover text
    text=attack_hover_text
)


def calculate_weight_at_const_rate(s, l_a, l_b, p):
    #TODO: Weight function doesn't seem to be getting the right values
    tolerance_of_severity = 0.5
    tolerance_of_likelihood = 0.5


    if l_a == 0 or l_b == 0:
        return 0
    l = (l_a * l_b) / l_a

    if s == 0:
        return 0
```

```
    w = (1-p)*(l + (1-l)*tolerance_of_likelihood)*(s + (1-s)*tolerance_of_severity)
    return w*5


# Define scatter plot for connections with different widths
lines = []
for connection in connections:
    i, j = connection
    w = calculate_weight_at_const_rate(y[j], z[i], z[j], 1-(sizes[j]/100))
    hover = f'Weight = {w/5}\
    <br>Change in Severity = {y[j]-y[i]}\
    <br>Change in Likelihood = {z[j]-z[i]}'
    line = go.Scatter3d(
        name=f"{names[i]}-{names[j]}",
        x=[x[i], x[j]],
        y=[y[i], y[j]],
        z=[z[i], z[j]],
        mode='lines',
        line=dict(color='blue', width=w),  # Random width for each line
        hoverinfo='text',  # Disable hover for connections
        hovertext=hover,
        meta=w/5  # Store weight as metadata
    )
    lines.append(line)

# Sort connections based on weight (descending order)
lines.sort(key=lambda line: line.meta, reverse=True)

# Define layout with gridlines and adjusted x-axis range
layout = go.Layout(
    scene=dict(
        xaxis=dict(title='Stages',\
        gridcolor='white',\
        zeroline=False,\
        titlefont=dict(color='white'),\
        tickfont=dict(color='white'),\
        ticktext=custom_labels,\
        tickvals=list(range(len(custom_labels)))),
        yaxis=dict(title='Severity',\
        gridcolor='white',\
        zeroline=False,\
        titlefont=dict(color='white'),\
        tickfont=dict(color='white')),
        zaxis=dict(title='Likelihood',\
        gridcolor='white',\
```

```
        zeroline=False,\
        titlefont=dict(color='white')),
        aspectmode='manual',  # Manual control of aspect ratio
        aspectratio=dict(x=2.5, y=1, z=1)  # Increase the relative length of the x-axis
    ),
    paper_bgcolor='grey'  # Set background color to black
)

# Create plotly figure
fig = go.Figure(data=[scatter, *lines], layout=layout)

fig.update_xaxes(
    gridcolor="black"
)
fig.update_yaxes(
    gridcolor="black"
)

# Initialize global variable to store clicked points
clicked_points = []

# Create Dash app
app = dash.Dash(__name__)

# Define app layout
app.layout = html.Div([
    dcc.Graph(
        id='graph',
        figure=fig,
        style={'width': '100vw', 'height': '100vh'}
        # Set width and height to viewport width and height
    ),
])

# Define callback function for click event
@app.callback(Output('graph', 'figure'), [Input('graph', 'clickData')])
def draw_line(clickData):
    global clicked_points
    if clickData:
        if 'points' in clickData:
            points = clickData['points']
            if len(points) == 1:
                p = points[0]['pointNumber']
                clicked_points.append(p)
                print(clicked_points)
```

```
                if len(clicked_points) == 2:
                    p1, p2 = clicked_points
                    if(p1==p2):
                        clicked_points = []
                        return
                    if(p1>p2):
                        p2, p1 = p1, p2
                    x1, y1, z1 = fig['data'][0]['x'][p1],\
                    fig['data'][0]['y'][p1],\
                    fig['data'][0]['z'][p1]
                    x2, y2, z2 = fig['data'][0]['x'][p2],\
                    fig['data'][0]['y'][p2],\
                    fig['data'][0]['z'][p2]
                    w = calculate_weight_at_const_rate(y2, z1, z2, 1-(sizes[p2]/100))
                    line_trace_name1 = f"{names[p1]}-{names[p2]}"
                    line_trace_name2 = f"{names[p2]}-{names[p1]}"
                    line_index = None
                    for i, trace in enumerate(fig['data']):
                        if trace['name'] == line_trace_name1 or\
                        trace['name'] == line_trace_name2:
                            line_index = i
                    if line_index is None:
                        print("add")
                        fig.add_scatter3d(
                            name=line_trace_name1,
                            x=[x1, x2],
                            y=[y1, y2],
                            z=[z1, z2],
                            mode='lines',
                            line=dict(color='blue', width=w),
                            hoverinfo='text',  # Enable hover for the line
                            hovertext=f'Weight = {w/5}\
                            <br>Change in Severity = {y2-y1}\
                            <br>Change in Likelihood = {z2-z1}',
                            meta=w/5  # Store weight as metadata
                        )
                    clicked_points = []  # Reset clicked points
        return fig


    if __name__ == '__main__':
        app.run_server(debug=True)
```

5 Conclusion

In conclusion, visualizing attack graphs in an interactive 3D environment can provide valuable insights into potential cybersecurity threats and vulnerabilities. Using Plotly and Dash, cybersecurity professionals can create customized visualizations allowing for deeper analysis and exploration of attack graphs. Customizing the weights of the edges between nodes is essential for prioritizing cybersecurity efforts and mitigating potential risks effectively. The custom equation presented in this whitepaper quantitatively measures the risk associated with each attack path, enabling organizations to allocate resources more efficiently and focus on addressing the most critical security vulnerabilities.