CISC 844 Exploring the Use of Behavior Trees for Security Modeling with PyTrees

Nicholas Dionne (24kg1@queensu.ca)

1 INTRODUCTION

The threat to security continues to evolve in both frequency and complexity, cybersecurity professionals find themselves increasingly overwhelmed and so security automation has developed to help monitor systems, detect intrusions, and respond to any incidents. However, these existing approaches such as rule-based systems and machine learning (ML) or artificial intelligence (AI) can lack transparency and flexibility. These limitations in some cases have led to a growing interest in alternative methods like modeling security using decision models like behavior trees, as they are generally more structured and easy to interpret. In this project were exploring behavior trees from the PyTrees Python library as a method for security modeling which will not only show the capabilities of both behavior trees as a potential security solution but also PyTrees as domain specific language (DSL) helping potential domain experts to more easily access and utilizes behavior trees in there domain such as security. To be more specific, the project utilizes PyTrees to model and simulate a Red Team (Attackers) vs. Blue Team (Defenders) scenario with both teams utilizing their separate behavior trees. The Red Team finds, exploits, and exfiltrates data from vulnerable systems; the Blue Team, on the other hand, attempts to detect and mitigate attacks to the systems through monitoring, patching, and effective responses. Through this simulation it will be demonstrated how behavior trees can be utilized in security as an individual measure and potentially to support approaches such as ML and AI in security by offering a realistic training environment.

2 BACKGROUND AND MOTIVATION

2.1 Behavior Trees

Behavior Trees are used for modeling hierarchical decision-based behavior [4]. Behavior trees, I believe, were originally created in the video game industry for managing complex AI decision-making in non-player characters (NPCs) and behavior trees further expanded towards applications in robotics, autonomous systems, and simulation [4]. The bullet points below define and explain the parts of a behavior tree, such as composite nodes, decorator nodes, and leaf nodes, as seen in the PyTrees documentation [4].

- **Composite Nodes:** These manage the control flow among child nodes. Common types include:
  - *Sequence:* Executes child nodes in order until one fails (AND logic).
  - *Selector:* Executes child nodes in order until one succeeds (OR logic).
  - *Parallel:* Executes all child nodes simultaneously (not actually simultaneous, as each child is ticked individually, but it simulates parallelism by returning to the parent that it all happened at once). The result is based on a specified success/failure threshold (succeeds if at least N children succeed). Can be useful for coordinating concurrent behaviors, such as monitoring while responding.
- **Decorator Nodes:** Modify the behavior of a single child, such as inverting success/failure results or repeating execution.
- **Leaf Nodes:** These represent actions ("send alert", "patch system") or conditions ("is compromised?", "is firewall active?").

On top of this, each node returns one of four potential statuses: failure, success, invalid, and running, which are all rather self-explanatory as the words define them for exactly what they are [4]. Anyway, these statuses are what dictate the general control flow of the behavior trees, so a lot is dependent on these status returns.

In this project, behavior trees were not utilized for typical applications such as robotics, which is what PyTrees was also originally created for [4]. Instead, behavior trees in this project are utilized for simulating cybersecurity scenarios as a red (attackers) vs. blue (defenders) game, the idea being that a detailed enough game could potentially inform defenders on potential chains of attacks and or vulnerabilities they had not accounted for. Moreover, such a simulation with realistic human-like attackers and defenders could also be valuable for training effective agents for automated security [3].

2.2 Domain-Specific Languages and Model-Driven Development

Domain-Specific Languages (DSLs) are programming languages that are specifically built for application in a domain [1, 2]. However, these languages are typically different from general languages (Python, Java) in the way that these languages (DSLs) are optimized to be expressive and simple in their domain [1, 2]. This is to allow experts in a domain to model or program solutions without the need for any deep or extensive programming knowledge [1, 2]. Some examples of DSLs include SQL (for relational databases) or PDDL (for planning problems in AI/robotics). When it comes to software engineering DSLs can often be integral to modeling during development as the use of DSLs allows developers to work a higher level of abstraction which reduces the complexity, improves maintainability, and helps to align implementation with domain concepts [1, 2]. PyTrees is a Python library, so while not technically a full DSL in a formal sense, it does behave like a DSL in practice, by providing a domain-focused API that helps users to model behavior trees relatively easily by abstracting away most of the lower-level complexity [1, 2, 4]. PyTrees likes to be modular in the sense that it supports reusing and adding additional components rather well, and on top of this, by abstracting away the lower level complexity, PyTrees focuses on the logic of decision making in behavior trees. PyTrees in this project will serve as the library and or DSL in which I explore modeling with behavior trees and implement them for an understanding of how modeling behavior trees may be utilized for security purposes [4].

2.3 Security Automation Challenges

Systems are receiving an ever-increasing number of threats, many of which require timely or immediate responses to maintain a certain level of security. This is where security automation comes in, typically speaking, these are often rule-based, such that admins create rules with signatures, which are essentially if-then logic that provide a response when they detect specific conditions in say a network or host. These responses can be anything, such as blocking an IP after failed authentication attempts. The thing is, these rule-based approaches are effective but inflexible, as this approach often misses less well-known threats, and it can be hard to keep their signatures maintained and properly extended to cover a multitude of attacks, as they are always evolving. So recently, machine learning (ML) and artificial intelligence (AI) have been favored for cybersecurity tasks as these are adaptable approaches that can be utilized to detect anomalies or even predict intrusions into a system. However, there's a rather well-known issue that whenever these methods are applied to a domain, they suffer from a lack of transparency, which makes it difficult for just about any professional in a domain to determine why certain actions were taken. Moreover, there's another well-known issue: these methods require a large amount of training data that's accurate to the environment they'll be operating in, with examples of attacks sprinkled in, and if this data isn't good enough, it may cause these methods to act unpredictably. This is where behavior trees come in, as behavior trees could be the happy median between rule-based, ML, and AI as behavior trees allow for a structured and more explainable approach on top of the additional benefit of being modular, allowing for the easy addition of new behaviors and parts of a system [4]. In security, it's important to be able to properly trace and justify actions, not only for future patching but in some cases for investigation and or legal reasons. To put it simply, behavior trees are well visualized, transparent, and allow for decent maintainability compared to rule-based or black-box ML/AI approaches. Lastly, behavior trees offer a way to not only potentially act as a responsive security measure but also a method for training agents or for even modeling systems and testing those systems for potential security vulnerabilities.

3 PROJECT DESIGN AND IMPLEMENTATION

3.1 Materials and Software to Be Utilized

- PyTrees: The primary DSL framework using behavior trees.
- Python: The programming language.
- Vscode.
- Linux Environment (Ubuntu).
- Research Papers and PyTrees Documentation.

3.2 Simulation Overview

For the project's exploration of behavior trees for security modeling, I implemented a simplified Red Team (Attackers) vs. Blue Team (Defenders) simulation. The implementation is done utilizing PyTrees, and it simulates a small network with relatively standard components, such as some systems (systems have three states: vulnerable, compromised, patched), a firewall, and an intrusion detection system (IDS). Each of the teams, Red and Blue, operates in turns, first Red, then Blue, with each team operating independently through its own behavior tree, which decides the actions the team will take at every tick or step. Also the simulation maintains or keeps the relevant information from each round (tick) such as the state of each system, firewall status, detected attack logs, and attacks made logs, This takes the place of a blackboard in sharing memory with the behavior trees for querying and decision making as I ran into issues utilizing blackboards for sharing memory among behavior trees which seems to be a common or popular method used among behavior tree implementations [4]. Without implementing a certain amount of randomness of our own initiative the simulation would be rather deterministic, boring, and lacking in realism so to fix this we add a certain amount of randomness to actions and there outcomes such as exploiting a vulnerability or detecting an intrusion (attack) because in reality theirs a certain level of uncertainty in the actions taken by both attackers and defenders. As for the rates at which things can succeed or fail right now its set more in the favor of defenders wining however, the rates I decided on for the simulations are more intuitive then factual in reality if a simulation were to really be of use and realistic it would have to have some research behind it on the odds of attackers and defenders successfully attacking and or defending certain systems as well as an element of learning from each round (tick) involved as humans are not static.

3.3 Red Team Behavior Tree

The Red Team, as the attackers, has the ultimate goal of compromising as many of the accessible systems on the network as possible to exfiltrate those systems data for personal gain, all while avoiding detection from the Blue Team defenders. The behavior tree of the Red Team is structured as follows

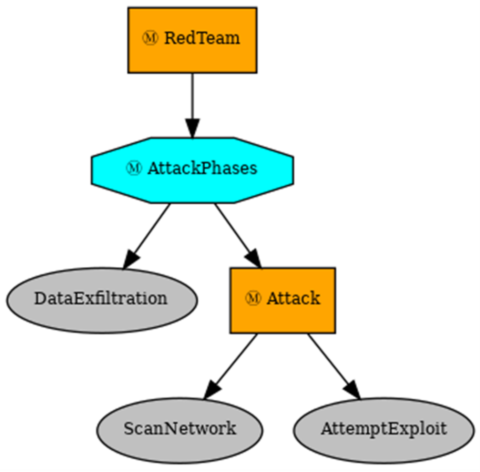


Figure 1: Red Team Behavior Tree

the top level (root) node is a sequence node, which enters directly into the AttackPhases selector node. This node uses OR logic from left to right, ticking the nodes to select the first node with a success return. So if DataExfiltration returns failure after being ticked, the behavior tree continues to the Attack node, which is a sequence node following AND logic, also from left to right, so it will tick ScanNetwork and expect a success return before being allowed to tick AttemptExploit and await its return of success or failure.

- **ScanNetwork:** Tries to identify a vulnerable target within the network. ScanNetwork is always the first step in the sequence and is required for subsequent actions to proceed due to the AND logic utilized by sequences.
- **AttemptExploit:** When a vulnerable system is found by ScanNetwork, the Red Team then attempts to exploit it. Exploits only succeed probabilistically (introducing randomness), to help in simulating real-world challenges like missing the exploit window or encountering a patched system. Successfully exploiting a system on the network will change its status to compromised as well.
- **DataExfiltration:** This is for when access has already been gained through previous use of AttemptExploit. DataExfiltration allows the Red Team to attempt to steal valuable data from a compromised system. This action also comes with the chance of triggering alerts depending on the Blue Team's current monitoring state or capabilities.

3.4 Blue Team Behavior Tree

The Blue Team as the defenders are constantly on watch for potential compromises in the systems present on the network, the Blue Teams goal is to maintain and strengthen the security of the systems. The behavior tree of the Blue Team is structured as follows

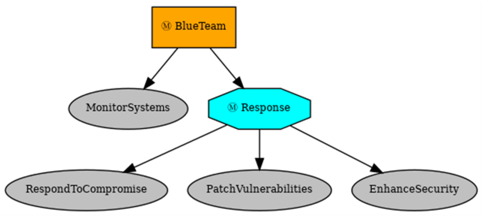


Figure 2: Blue Team Behavior Tree

the top level (root) node is a sequence node so again ticking the nodes from left to right the behavior tree first checks that the Blue Team has the capability to monitor there systems and detect potential attacks through the MonitorSystems node. Once this is confirmed through a success return, the behavior tree moves onto the Response selector node, ticking each behavior node from left to right before selecting the first node to return success as the Blue Team response action for the round (tick).

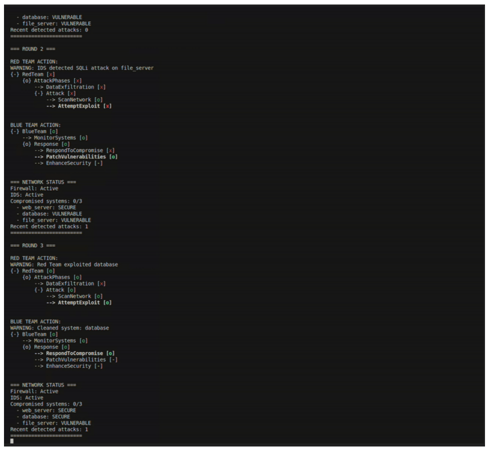
- **MonitorSystems:** Through each round (tick), MonitorSystems checks for signs of compromise or anomalies in the systems on the network. If an intrusion is detected, then the Blue Team can trigger incident response actions which are located further in the behavior tree.
- **PatchVulnerabilities:** It attempts to update and patch systems from their vulnerable state. If the Blue Team succeeds, PatchVulnerabilities will improve resistance to known exploits, making the status of a system patched. But since this requires time and resources, monitoring or other actions might no longer have priority.
- **RespondToCompromise:** If a system is detected to have been compromised by the Red Team, then the Blue Team defenders may utilize RespondToCompromise to simulate making an attempt at containing, reimaging, or quarantining the system in question, changing its status to patched.
- **EnhanceSecurity:** This simulates deploying longer-term improvements such as a firewall, new firewall rules, updated IDS signatures, or system baselining. By utilizing EnhanceSecurity, the Blue Team will activate the firewall or enhanced security method, and this helps protect the systems considerably from Red Team exploit attempts.

3.5 Using PyTrees

PyTrees is the Python library that was used in this project to manage and implement behavior trees, as this library effectively acts as a DSL, successfully simplifying the process of implementing behavior trees [4]. PyTrees provides rather intuitive methods for defining composite nodes (py_trees.composites.Sequence, Selector, Parallel) and for leaf behaviors as well (py_trees.common.Status.Success, Failure, Invalid, Running) also decorators follow a similar intuitive structure (py_trees.decorators.Inverter, Repeat, Retry, etc.) [4]. Custom behaviors can also be built and referenced this way using PyTrees [4]. The behavior trees themselves were implemented as subclasses of the base class (py_trees.behaviour.Behaviour) in PyTrees [4]. In PyTrees and I assume most other behavior tree implementation methods, the behaviors contain the logic that checks conditions and performs actions using information [4]. In the case of this project, that information comes in the form of a shared simulation state between the Red Team and Blue Team behavior trees [4]. There's a key aspect of PyTrees that I've referred to a bit in this paper so far but have neglected to explain and that is a tick or ticking a single tick refers to advancing a tree by evaluating nodes in a tree by their logic and uses the update functions in each of the behaviors to well update the statuses accordingly to Success, Failure, Running, or Invalid [4]. Ticking is a little complicated, as the source documentation refers to both the activation of a single behavior node and its update process as a tick, but also the entire flow through a behavior tree as a single tick or tick slice [4]. So a good way of looking at this is the figures above showing the behavior trees, those figures are showing a single tick slice after the necessary behavior nodes have been ticked through [4]. Anyway, the simulation operates by first ticking the Red team and then the Blue Team, and this is considered one round of play in the simulation or a single full tick; furthermore, the simulation runs for a total of 10 rounds (ticks) and this can be adjusted as needed [4]. Lastly, any operations within a behavior have to consider the fact that the whole system runs on ticks if any one behavior takes to long to return then the entire everything else ends up held up this is why the PyTrees documentation recommends that behavior trees be utilized purely in the decision making process and not one actually acting on decisions made as that is best left to exterior operations [4].

4 RESULTS AND OBSERVATIONS

The simulation implementation was a success overall as it captures dynamic play between the attackers and defenders through behavior trees implemented utilizing PyTrees. Both teams use the corresponding behavior correctly as the game develops during each tick, and the outcomes vary, meaning the simulation is not deterministic and therefore a little more realistic. The variations in play through additionally added randomness can include successful breaches, attack detections, or an effective defender response to stop breaches. It should also be mentioned that the simulation has an initial state set for the environment that does not change between games unless done so manually. The figure below is a look at the first two rounds (ticks) of the simulation played by both teams.



```
  - database: VULNERABLE
  - file_server: VULNERABLE
Recent detected attacks: 0
========================

=== ROUND 2 ===

RED TEAM ACTION:
WARNING: IDS detected SQLi attack on file_server
{-} RedTeam [x]
    {o} AttackPhases [x]
        --> DataExfiltration [x]
        {-} Attack [x]
            --> ScanNetwork [o]
            --> AttemptExploit [x]


BLUE TEAM ACTION:
{-} BlueTeam [o]
    --> MonitorSystems [o]
    {o} Response [o]
        --> RespondToCompromise [x]
        --> PatchVulnerabilities [o]
        --> EnhanceSecurity [-]


=== NETWORK STATUS ===
Firewall: Active
IDS: Active
Compromised systems: 0/3
  - web_server: SECURE
  - database: VULNERABLE
  - file_server: VULNERABLE
Recent detected attacks: 1
========================

=== ROUND 3 ===

RED TEAM ACTION:
WARNING: Red Team exploited database
{-} RedTeam [o]
    {o} AttackPhases [o]
        --> DataExfiltration [x]
        {-} Attack [o]
            --> ScanNetwork [o]
            --> AttemptExploit [o]


BLUE TEAM ACTION:
WARNING: Cleaned system: database
{-} BlueTeam [o]
    --> MonitorSystems [o]
    {o} Response [o]
        --> RespondToCompromise [o]
        --> PatchVulnerabilities [-]
        --> EnhanceSecurity [-]


=== NETWORK STATUS ===
Firewall: Active
IDS: Active
Compromised systems: 0/3
  - web_server: SECURE
  - database: SECURE
  - file_server: VULNERABLE
Recent detected attacks: 1
========================
```

Figure 3: Red vs. Blue Simulation Output Example

The initial state for the simulation above in this case was that all three systems were set to the vulnerable state, the firewall is set to active (true), the IDS is set to inactive (false), and both attack history and attacks detected are initialized as empty. In the figure above, the output shows some custom prints to the console for additional information; however, most of the information output to the console in each round (tick) is in the form of ASCII trees, which are a provided display method in Pytrees (py_trees.display.ascii_tree) [4]. The ASCII trees show how each of the team's behavior trees develop during execution of the simulation through the red "x" indicating a failure returned and a green "o" representing a success returned upon ticking the behavior in the tree [4]. There are other methods of showcasing results and acheiving proper transparency using PyTrees such as the previous figures which were created using dot file outputs from a single tick of the simulation and there is also an extension to PyTrees called py_trees_js this can create a real time html interface with additional information on how the beahvior trees in implementation are changing and behaving [4]. One thing I would like to point out is that although the results are fine, there are some glaring issues, such as how to actually define or get realistic human behavior. I've attempted to replicate this to a limited extent through the addition of simple randomness to certain outcomes or procedures. However, human beings are not that simple, and the rates that I picked for the introduced randomness were picked intuitively, which is likely problematic in a sense.

5 LESSONS LEARNED

There are several valuable lessons in this project if a closer look is taken, both technical and conceptual. I found that building behavior trees in PyTrees was clean, relatively easy, and that the behavior trees could be easily changed with different behaviors swapped in and out. Also, since PyTrees offers Sequence, Selector, and Decorator node logic prebuilt into the library, and these cover most of anything that someone might want to implement with behavior trees, I did not find any need to build custom node types or other approaches for this project. The visual options for PyTrees help not only with transparency and seeing the decision-making process, but it is also quite helpful for debugging. For this, I recommend using the ASCII trees as they're easy to set up and are sufficiently detailed. There are definitely some challenges involved as well, like actually translating real cybersecurity tasks into behavior trees, which requires carefully carrying out proper abstraction, which requires an individual knowledgeable in the domain to differing degrees depending on the level of detail and the size of the system involved. The ticking aspect of PyTrees can be difficult to understand, and the PyTrees documentation is a must-read to really understand the peculiarities involved, although the documentation on ticking is not exactly straightforward to read either [4]. One of the main difficulties with behavior trees and PyTrees for that matter is properly sharing and keeping track of the overall state for this PyTrees introduces the use of blackboards which are rather useful that if you can get them to work as intended which was not able to do in my implementation as i find them rather tricky [4]. If I were to have more time with this project, after the lessons I have learned in implementing it, I would first start by improving my approach and potentially expanding on but I would ultimately like to take another look at blackboards for sharing the state between behavior trees. There's also the application of adaptive behavior or learning, which can be applied to behavior trees, and this could make the simulation more realistic [3]. There's the potential to take the simulation further as well by utilizing a network simulation of some kind or real security event data.

6 DISCUSSION AND RELATED WORK

The following is a brief discussion of some related work on the subject of behavior trees for security modeling, and in this paper *Designing Robust Cyber-Defense Agents with Evolving Behavior Trees*, which I have previously referenced in this report, the authors also utilize PyTrees [3]. Security automation typically falls into two categories either rule-based or machine learning (ML). Rule-based security automation operates through predefined logic, and although these are rather transparent, they are difficult to scale and they when faced with new attacks. ML for security automation, on the other hand, offers the adaptability that rule-based automation lacks, meaning that new attacks are not necessarily the end for ML. However, ML loses the transparency offered by rule-based automation as ML is often a black box, not to mention the difficulties with acquiring the large amount of labeled training data accurate to the system that needs to be protected. Behavior trees are kind of a middle ground offering transparency and some amount of adaptability to situations. The paper mentioned above shows recent research on the subject in this report from 2024, where they apply behavior trees to cybersecurity simulation in a bid to demonstrate the suitability of behavior trees for modeling reactive, and intelligent agents [3]. The work presented in the paper aims to mesh behavior trees and their transparent and explainable nature with the more adaptable approach of ML for what they call evolving behavior trees (EBT) [3]. These EBTs could be used for security automation, and when the researchers tested them for just that, the results were rather impressive [3].

7 CONCLUSION

The objective of this project was to explore the use of behavior trees implemented through the use of the PyTrees library as a method for security modeling. To achieve this, to an extent, a semi-realistic Red Team (Attackers) vs. Blue Team (Defenders) was simulated. The results were that behavior trees implemented through PyTrees are easy to implement as PyTrees effectively removes the lower-level logic and offers a friendly framework for domain experts who are not necessarily programmers to utilize. The implemented behavior trees also showed how complex strategies can be captured and adapted to with a rather clear and explainable display of the decision-making involved in the simulation. Lastly, it can be seen from this report that behavior trees can potentially be an effective middle ground between rule-based security automation and ML security automation, as they provide a more balanced approach that can, in some cases, be utilized to make up for the faults of approaches such as ML. To conclude, it can be said that behavior trees and PyTrees in this case are well-suited, although limited, as an approach to security modeling and as security threats become more dynamic and complex unique tools such as behavior trees may offer a new unseen value when an explainable and yet adaptable approach to security is needed.

REFERENCES

[1] Juergen Dingel. 2024. CISC 844 Materials.

[2] Martin Fowler. 2008. Domain Specific Language. https://martinfowler.com/bliki/DomainSpecificLanguage.html Accessed: 2024-04-10.

[3] Nicholas Potteiger, Ankita Samaddar, Hunter Bergstrom, and Xenofon Koutsoukos. 2024. Designing Robust Cyber-Defense Agents with Evolving Behavior Trees. In *2024 International Conference on Assured Autonomy (ICAA)*. 1–10. https://doi.org/10.1109/ICAA64256.2024.00011

[4] Splintered Reality. 2024. py_trees: A behaviour tree implementation for Python. https://github.com/splintered-reality/py_trees Accessed: 2024-03-09.