CSE 233 Final Project Presentation

By Neel Apte, Gavin Cooke, Jimmy Franknedy, Sammy Tesfai

Reinforcement Learning Algorithm

- Type of machine learning technique that trains a system to make decisions that will gain the most optimal reward possible.
- Use a reward and punishment system while processing data
  - Learn the effect of taking such actions and able to discover paths to achieve the highest reward
- RL state gives information about an environment to the agent
  - Information can be previous rewards, previous locations, and previous actions taken
- RL observation tells the information about an environment to the agent, which helps build state
- RL training is required to learn proper steps needed to acquire the maximum reward

Blue and Red Agent

- Blue Agent - Starts every scenario with tools installed on every host
  - During the first 3 timesteps of every game, monitors network activity and chooses a strategy depending on the perceived red agent.
  - 3 strategies. One for Bline, one for Meander, and one which sleeps if no attack is perceived.
  - env.reset() Blue Agent continues to sleep
- Red Agent
  - An easy strategy for a red agent would be to simply sleep for 3 timesteps before beginning its attack.
  - We implemented a simple agent which sleeps three times and switches to Meander which works well and gets a score of 21.8

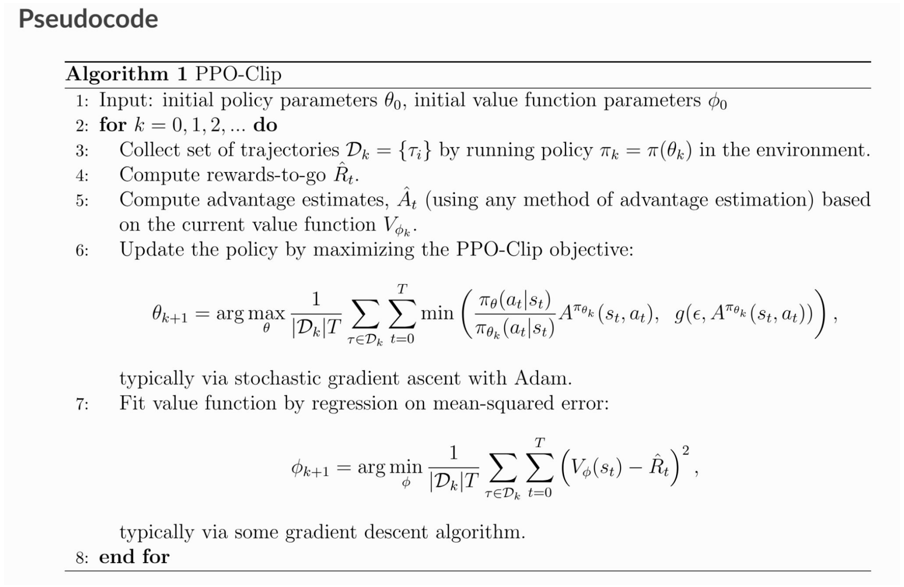
Design Choice Justification

- Original Plan
  - Train the red agent using the Competitive Reinforcement Learning Framework provided by the paper “Competitive Reinforcement Learning for Autonomous Cyber Operations”
- Problem
  - Setup was complicated
  - Training time and sample size was infeasible within given time frame

- Current Plan
  - Train the red agent using PPO and Actor-Critic Network
- Advantages
  - Found an off-the-shelf implementation that already incorporated Cage 2
  - Training time more efficient, requiring less time and smaller batch sizes
  - Simple and straightforward
    - No need for the Competitive RL framework as the goal of this agent is to get as many points as possible, rather than establishing Nash Equilibrium.

General Algorithm Discussion

- Simple PPO Algorithm
    - Followed the PPO Clip Update
- Simple Actor-Critic Network
    - Hidden Layer nodes set to 64/256
- Observation State
    - Received from env.reset() and env.step()
    - Time vector
- Action Space
    - Received from "red_agent.get_action(state)"
        - Full set included 888 possible different actions



Pseudocode

```
Algorithm 1 PPO-Clip
1: Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
2: for $k = 0, 1, 2, ...$ do
3:     Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4:     Compute rewards-to-go $\hat{R}_t$.
5:     Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based
       on the current value function $V_{\phi_k}$.
6:     Update the policy by maximizing the PPO-Clip objective:
           $\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left( \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \;\; g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$
       typically via stochastic gradient ascent with Adam.
7:     Fit value function by regression on mean-squared error:
           $\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left( V_\phi(s_t) - \hat{R}_t \right)^2,$
       typically via some gradient descent algorithm.
8: end for
```

Hyperparameter Tuning - Brief Overview

We cover our hyperparameter tunes in the following 3 sections

Constant Components → **Varied Components** → **Optimal Components**

- **Constant Components**: Training or agent hyperparameters that were **consistent** during agent training
- **Varied Components**: Training or agent hyperparameters that **changed** during agent training
- **Optimal Components**: Training or agent hyperparameters that **yielded the most points**

Constant components - Training

- Learning Rate (0.002)

  A "step size" or "speed limit" for how fast an RL algorithm learns from its experiences.

- Beta ([0.9,0.990])

  A "safety belt" or "brake" for controlling how much the policy (the set of actions the algorithm takes) can change during training

- Gamma (0.99)

  A discount factor that decides how much the agent value points earned in the future compared to points earned right now

- K_Epochs (6)

  How many times to sample and update the actor-critic networks based on the collected data from the environment

- EPS_clip (0.2)

  A boundary or limit on how much the agent allows its current strategy to change in one step

Varied Components - Training

- Mini-batching size

  A value that determines how many steps the agent should take before updating the Actor-Critic network

  Experimented with 20,000 - 150,000 steps

- Training batch size

  A value that determines how many total steps the agent during the course of it's training

  Experimented with 100,000 - 15,000,000 steps

Varied Components - Agent Design

- Observation state

  A bit vector that represents what the agent "sees" at the beginning of each time step. Included a Bit Time Vector with default observ.

  - Bit Time Vector

    A bit vector counting time steps, shifting the '1' bit right each step till game ends.

- Action space

  Experimented with the following action sets

  - Optimal Action Space with/without "Sleep"

    Optimal actions for the red agent to maximize points by tricking the blue agent into sleeping.

  - Safe Action Space

    Actions ensuring CybORG doesn't error in training/testing. Some actions aren't pre-checked before execution.

- Actor-Critic Network

  The neural network determining red agent's action based on its observation.

  Experimented with 64 - 256 nodes per hidden layers.

Optimal Components - Training

Learning Rate	0.002
Beta	[0.9,0.990]
Gamma	0.99
K_Epochs	6
EPS_clip	0.2
Mini-batching size	20 000
Training batch size	1 000 000

Optimal Components - Agent Design

- Observation state

  40 bit observation vector (default from CybORG)

- Action Space

  Complete set of 888 red actions (default from CybORG)

- Actor-Critic Network

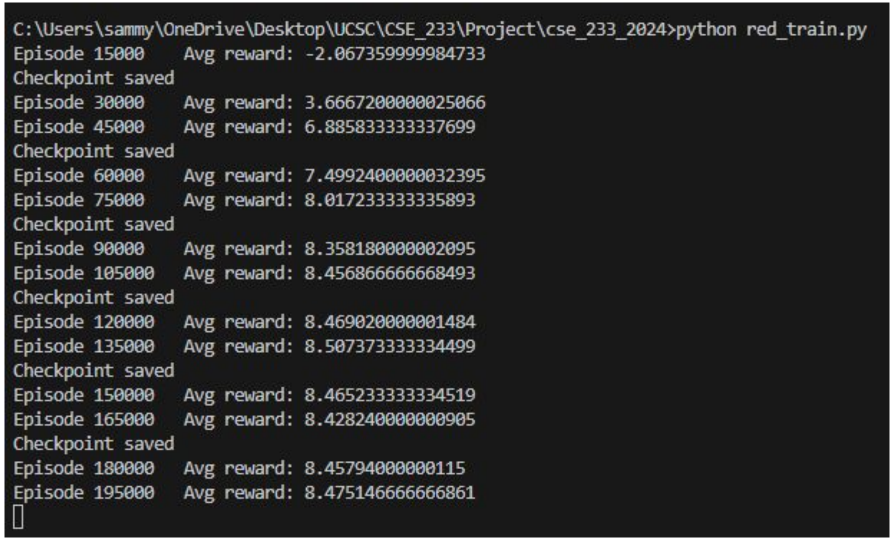
  256 Nodes per Hidden Layer

Analysis of Results & Challenges

- 1st Model reward between 8.2-8.5
  - Uses action space of all possible red actions except for privilege escalate and ssh brute force
  - Actions caused KeyError within privilege escalate module
- 2nd Model reward between 30-32
  - Used full action space with error checking to prevent KeyError mentioned above
  - Noticed CybORG does not penalized for Invalid Actions
- 3rd Model (Final Model) 28-29
  - Modified invalid actions to reward -0.1 instead of a positive score.

Red Agent Training Model 1

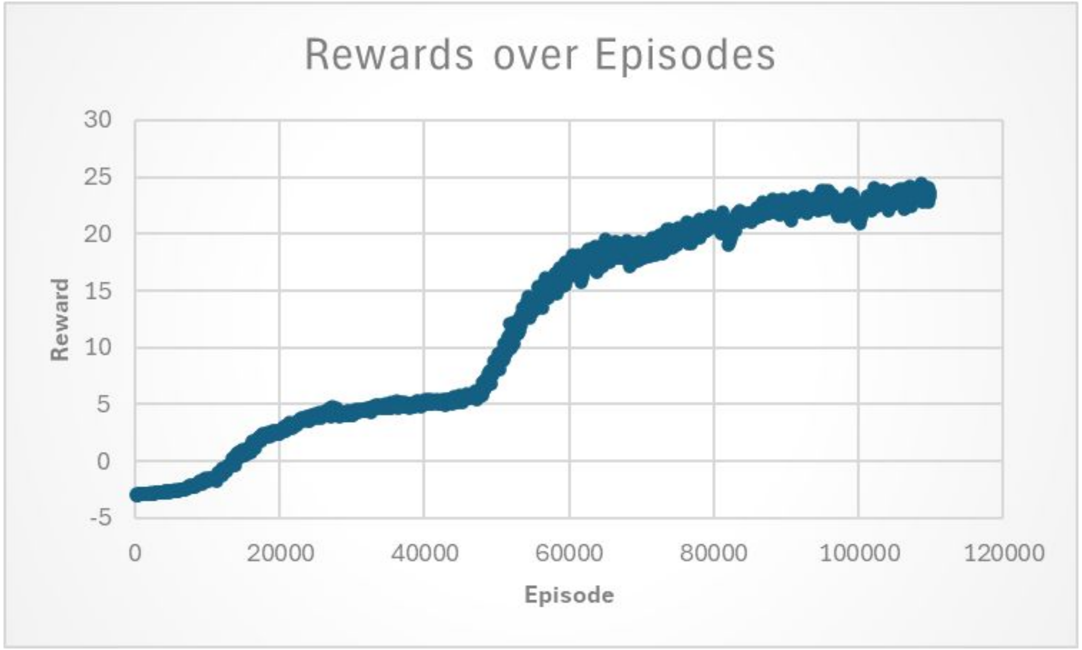
- Produced ~8 reward points for this model.
- Did not record reward over episodes so there is not graph
- Did not retain 2nd model so there is no screenshot or graph to present on this.



```
C:\Users\sammy\OneDrive\Desktop\UCSC\CSE_233\Project\cse_233_2024>python red_train.py
Episode 15000    Avg reward: -2.067359999984733
Checkpoint saved
Episode 30000    Avg reward: 3.6667200000025066
Episode 45000    Avg reward: 6.885833333337699
Checkpoint saved
Episode 60000    Avg reward: 7.4992400000032395
Episode 75000    Avg reward: 8.017233333335893
Checkpoint saved
Episode 90000    Avg reward: 8.358180000002095
Episode 105000   Avg reward: 8.456866666668493
Checkpoint saved
Episode 120000   Avg reward: 8.469020000001484
Episode 135000   Avg reward: 8.507373333334499
Checkpoint saved
Episode 150000   Avg reward: 8.465233333334519
Episode 165000   Avg reward: 8.428240000000905
Checkpoint saved
Episode 180000   Avg reward: 8.45794000000115
Episode 195000   Avg reward: 8.475146666666861
```

Red Agent Training Model 3

- Recorded reward over every 100 episodes
- Reward gain slowly starts to plateau around 25-28
- Resulted in our final reward being ~28.



Red Agent Analysis - Action Sequence

#	Blu Agent	Red Agent	Rew	#	Blu Agent	Red Agent	Rew	#	Blu Agent	Red Agent	Rew
1	Decoy	DiscoverRem	0	11	Sleep	BlueKeep	1.3	21	Sleep	FTPDirTrav	1.4
2	Decoy	InvalidAct	-0.1	12	Sleep	SQLInject	1.4	22	Sleep	FTPDirTrav	1.4
3	Decoy	FTPDirTrav	0.1	13	Sleep	InvalidAct	-0.1	23	Sleep	BlueKeep	1.4
4	Sleep	PrivEsc	0.1	14	Sleep	BlueKeep	1.4	24	Sleep	EternBlue	1.4
5	Sleep	HarakaRCE	0.2	15	Sleep	EternBlue	1.4	25	Sleep	HTTPRFI	1.4
6	Sleep	EternalBlue	0.3	16	Sleep	InvalidAct	-0.1	26	Sleep	BlueKeep	1.4
7	Sleep	DiscoverRem	0.4	17	Sleep	EternBlue	1.4	27	Sleep	HTTPRFI	1.4
8	Sleep	HTTPSRFI	0.3	18	Sleep	BlueKeep	1.4	28	Sleep	BlueKeep	1.4
9	Sleep	BlueKeep	1.3	19	Sleep	BlueKeep	1.4	29	Sleep	BlueKeep	1.4
10	Sleep	BlueKeep	1.3	20	Sleep	EternBlue	1.4	30	Sleep	ExploitRem	1.4

Total Reward: 28.8

Red Agent learns to trick Blue Agent into Sleeping by doing invalid actions

Red Agent learns to privilege escalate and take over a User

Red Agent continues to try the last action that gave it a positive reward

Red Agent Analysis - Behavior Reasoning

Red Agent learns to trick Blue Agent into Sleeping by doing invalid actions

We observe the Red Agent successfully tricked the Blue Agent into Sleeping, evident after turn 3.

↓

Blue Agent appears to mistake the Red Agent's actions for those of a sleeping opponent

↓

We deduce the invalid actions don't advance Red's foothold in the network and thus Blue Agent views this as a Sleeping Red Agent

Red Agent learns to privilege escalate and take over a User

We observe the Red Agent successfully escalating to Root Privileges on a User in the subnet

↓

We infer that the Red Agent has mastered advancing its foothold in the network based on its acquisition of root access on a User host within 10 timesteps.

↓

Otherwise we would see the Red Agent doing random actions

Red Agent continues to try the last action that gave a positive reward

We observe the Red Agent attempting host exploitation, attribute this behavior to the initial positive reward, and reason the following

↓

The Bline agent took 16 steps to reach and impact the Operational Server.

↓

Given the total action space is 888 actions within the first 16 steps of a game, our agent has a $1/888^{16}$ chance of learning this behavior.

↓

Since each game has 30 timesteps, we would require a batch size of 30×888^{16} to capture this behavior at least once.

↓

Our update timestep is only 20,000 and thus we reason this limits the agent exploration and binds to learning only to exploit a host.