CSCI 6550 Final Project Report

Jesse Franks

Background and motivation

Software testing is tricky, and it's hard to cover every combination of inputs a user can give. While creating and testing robust software helps to mitigate risk, you can never achieve 0 risk. Software with security vulnerabilities can lead to a wide range of problems, including loss of personal data, negative PR, and lost revenue.

Objective: Create an agent that can expose (initially simple) SQL injection vulnerabilities in an application

Related concepts

The primary concepts involved with this project are from Chapter 3: Search Algorithms. Specifically, uninformed and informed search algorithms. The uninformed algorithm was implemented as Breadth First Search (BFS), while the informed algorithm was implemented as A*. The A* implementation uses a custom heuristic, which will be discussed later. The project also utilizes the idea of representing search spaces with "states" that hold information about a given step in an algorithm.

Proposed Solution

The proposed solution requires two main components: A victim app that will be vulnerable to SQL injections and an agent that attempts to gain access to the application.

The victim app will be a lightweight Flask application with an SQLite instance to hold an admin user. There needs to be a login functionality that contains an intentional vulnerability by using python f strings to construct SQL queries.

The agent will act as an adversary. It will utilize the login functionality (an endpoint) to explore a state space and try to get past the login page. A custom heuristic will be used to determine the informed algorithm's result. We will reward SQL syntax errors (+10), treat invalid logins as unwanted (+20), and if the login is bypassed, we will add 0, signaling a goal state was found. The states will contain information on username, password, parent state, and the current cost. The initial state will have username = "admin" and password =

“password”. As the algorithm iterates, we will add selected payloads on. These payloads are common paths used to achieve SQL injections. These are:

1. a single quote, “’”, to break string literals
2. comment characters, “ --", to ignore the rest of a query
3. tautologies, “OR 1=1”, to force a true condition.

Implementation Details

Dependencies/Libraries: Python 3.14, Flask, requests, beautifulsoup4, sqlite3, heapq, deque, and time

With Python and Flask, a lightweight app was created that contains vulnerability in the login route, which has a POST endpoint exposed. Another GET endpoint is exposed to create an initial landing page. The victim app uses SQLite to add an initial user, admin, to the database of users. The query used by the login is:

query = f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'"

For the adversary, auditor.py, heapq (a heap/priority queue) is used in the A* algorithm, and deque (queue) is used for BFS. The algorithms use the same state set as discussed previously, and track visited states with a tuple of (username, password). For simplicity, BFS utilizes a “fake” heuristic to maintain consistency between implementations. A script to kick off each algorithm and display the results exists in reporter.py.

Evaluation and Results

Both algorithms were able to successfully detect the SQL injection vulnerability. A* detected the vulnerability in 14.3 seconds after seeing 7 states. BFS detected it in 32.7 seconds after seeing 16 states. Both achieved a path cost of 2, which makes sense given BFS guarantees an optimal path cost. Both algorithms show that they can be successful, and it is a good demonstration of how A* should outperform BFS given a good heuristic.

Analysis

These results were possible due to a couple of configurations. First, the order of mutations is purposeful. Through experimenting and analyzing results, it was clear that username: “admin'—" and password: “password” were the cheapest injections that could be created. Without ordering the mutation in a way that would apply the a quote first, then a comment, then a tautology, the algorithm took many minutes to find a goal state, if it ever did. For the purpose of the project, I reordered the mutations to allow for a solution to be found quicker. The heuristic is kept simple on purpose because a more complicated one also added more time. I had experimented briefly with adding more responses and varying the heuristic, but this quickly seemed overcomplicated and forced. By only representing 3 states, the algorithm converged quicker and was much easier to understand.

At first, I saw these results as somewhat artificial, but I realized that in real life this is exactly what you would want to do. Given the information available, you would want to configure an algorithm to run as efficiently as possible. In fact, that’s the whole point of a heuristic. Without these changes in configuration, the algorithm would have taken much more time to run, and in turn must explore many unnecessary states. What seemed artificial was just using the information I had to my advantage.

Conclusions

This project has good potential to grow. Future improvements include:

- Building out a more complicated query for the algorithm to explore
- Building a more complicated backend that has more possible responses
- Add more algorithms to compare the results with
- Trying to incorporate machine learning to create a better, more general algorithm
- With permission, utilizing the algorithm on real world applications to help companies detect vulnerabilities

From this project though, I got hands-on experience with building a simple app, an adversarial agent, and implementing search algorithms. I had to decide on and build a custom state space and analyze the situation to determine a valid heuristic.