Final Project Report

Gabriel Adkins

[1] University of Memphis, 3720 Alumni Ave, Memphis, TN 38152
gpadkins@memphis.edu

Abstract. This paper details the development of a game I created for the purpose of showcasing network vulnerabilities, exploits, and mitigations. This project aims takes network security concepts and showcase them in an engaging way through a real-world scenario. It not only showcases vulnerabilities and mitigations, but details the attack path for exploiting them, and the result of those exploits.

1 Introduction

This project's goal is to take an interesting and interactive approach to network security concepts. To do this, I reworked the networking for a LAN based multiplayer game I made in a previous project. In this project, I use it to showcase how encryption and message validation are important and can be used to prevent attacks like packet sniffing and message spoofing.

This paper will first detail the different Python libraries used for the project, as well as some additional background on the game's design. The next section will detail the different aspects of the games design, as well as the vulnerabilities, exploits, and mitigations used on the application. Finally, the paper will conclude with an evaluation of the project and a potential future work on the project.

2 Related & Previous Work

2.1 Python Libraries: Tkinter and Threading

For the game portion of this project, I use a couple of important Python libraries—tkinter and threading. Tkinter handles the GUI for playing the game, as well as the core gameplay loop. Tkinter applications utilize its mainloop for things like updating the GUI, listening for events like user interaction, and processing/handling initiated events. In my application, I also use threading to make sure the mainloop isn't held up by certain function calls that maintain their own loops.

2.2 Python Libraries: Socket and Threading

This project also heavily features Python's socket library. It is a library for low level socket programming—perfect for a small project like this. In this project it's used for

making UDP socket connections. I chose UDP over TCP since that's typically what's used in video games. I figured this would also be beneficial for showcasing more vulnerabilities within the networking aspect of the game.

2.3 Python Libraries: Cryptography and Message Signatures

For the secure version of the networking code, a few different libraries are used. First, hashlib is used to create hashes of secrets. Then, Fernet from the cryptography library is used to create a cipher for encryption and decryption. This library also requires the use of base64 to encode keys in a base64 URL safe format. Finally, hmac is used to sign and verify messages.

2.4 Game Inspiration and Basis

This game is inspired/based on the childhood classic—Tic Tac Toe—and the computer classic—Minesweeper. For the readers that don't know, here's a quick explanation of the two games.

Tic-Tac-Toe is a simple turn based paper-and-pencil game that takes place on a three by three grid. Players take turns marking the board with their corresponding symbol (X or O) with the goal to get three of their marks in a row, column, or diagonal. Often times these games end in a tie, so I decided to combine it with another well-known game to make things more interesting.

Minesweeper is a singleplayer computer game that takes place on a grid of cells. Cells are randomly generated to be empty or contain bombs, where the objective is to flag the bombs and clear the empty cells. The loss condition is clearing a cell that contains a bomb. Because the game starts out with no cells cleared, some of the game requires luck, but there's a good bit of skill and logic involved in between the occasionally required guessing. This is because empty, cleared cells display a number showing how many of the surrounding cells contain bombs. These games are typically customizable with a different size grid and number of bombs.

3 Design

3.1 The Gameplay

The multiplayer gets its name, Tic-Tac-Sweep, and gameplay from two classics, Tic-Tac-Toe and Minesweeper. Rather than being turn-based like Tic-Tac-Toe, both players begin interacting with the board in real-time. The trick comes with marking a square on the Tic-Tac-Toe board. To do so, the player is required to clear a minesweeper board that is randomly generated for each attempt. The minesweeper portion of the game has customizable parameters that the host can adjust before opening their game to connections. These variations create an interesting twist on the game, requiring players to balance speed and logic.

It's worth noting that the project includes a single player “practice” version of Minesweeper that tracks your time. This was made for the previous version of the project and used for testing the base Minesweeper game logic before developing the Tic-Tac-Toe implementation. It wasn’t changed much for this project, only refactored slightly, as there was a change to have more separation between the game’s logic and Tkinter’s GUI game loop. Some elements of Tkinter are a part of the game’s logic, due to Minesweeper cells being Tkinter buttons. There is however, potential to refactor this further and keep logic and UI completely separate. For the sake of keeping the project’s goal focused, I cleaned up most of the overlap but left some of it as is.

3.2 The GUI & Game Loop

Tkinter’s mainloop handles the gameplay loop for the whole game and handles methods for starting, searching, and stopping games. It starts out on a simple menu page where the user can select menu buttons for single player, multi-player, instructions on how to play, or quit. The structure of the application follows a page based approach where the different menus are initialized, constructed and displayed by a parent controller class. All of these menus share the same top bar design for going back to the menu page, or in the multiplayer’s multi-page functionality, going back a page. They also contain a quit button for closing the application and a title for the current page.

Singleplayer Gameplay

The SinglePlayer page class handles single player gameplay and related attributes like time. It contains the frames for the Minesweeper gameboard and a side bar. Upon the application’s initialization it creates a side bar, info box, and settings box widgets. The side bar contains an info box for the things like the timer and the amount of bombs left/unflagged. It also contains a settings box for adjusting the game’s grid size and bomb percentage. These are controlled with custom arrow buttons and loop back around at the minimum and maximum values. The side bar also contains start and stop buttons which are connected the start and stop methods within the SinglePlayer class.

The gameplay loop begins with the class’s start method. First it changes the start button to a reset button, then initializes the timer and creates and displays the game. Next, it calls the update info method to continuously update the timer. An important method used in this class is the stop method. It ensures that the game can be reinitialized correctly by clearing and resetting widgets and attributes.

Multiplayer Gameplay

The multiplayer gameplay is held within two page classes—MultiPlayer and ServerListDisplay.

ServerListDisplay is the first page class the user will encounter. This page serves the function of displaying the currently hosted games awaiting a connection and their game settings. It also features a button for refreshing the page, a select menu of the games for choosing one, a join button for joining the game, and a host button for host-

ing a game instead. The join button calls an internal method to display the MultiPlayer page class if the connection is made. The host button on the other hand, simply displays the MuliPlayer page class allowing players to interact with the widgets on that screen for creating/starting a game.

MultiPlayer is the second page class users will encounter. Like the SinglePlayer page, it contains info bars, settings adjustments, start and stop buttons, and frames for containing the games boards. In this case, it has a primary Tic-Tac-Toe game frame that initializes when the game starts, and a Minesweeper frame to the side that initializes when a Tic-Tac-Toe button is pressed. The info box for displaying game information, shows text for the gameplay status when a game isn't happening like "select your settings" and "waiting for player." When a connection is made the title of the top bar changes to show the connection, and the info box changes to display the bomb count for active Minesweeper games.

This class also contains a method for updating the game state. It functions as a middleman between the networking layer and the game logic later. It does this by handling incoming game updates from the opponent. Outgoing updates on the other hand, are sent from inside the game logic layer. The update game method also has a message parameter. If the message isn't an end game message, then it updates the state of the gameboard, but if it's a win or quit message, it will update the game state accordingly, end the connection, and reset the game.

Due to the odd nature of MultiPlayer's pagination, the functions for stopping and resetting the game/page are extra thorough and over cautious. When a game is stopped, all widgets are recursively destroyed and recreated except for the Tic-Tac-Toe frame and its child widgets. This is to ensure a full reinitialization is done, but the game state on a win or loss still shows.

3.3 Matchmaking / Networking

Due to the nature of this project being about demonstrating vulnerabilities, attacks, and mitigations, there's two versions of the matchmaking code for this project. One version contains full mitigations, while the other one is the vulnerable version, and strictly the bare minimum for handling the network connections between games. This section will discuss only the vulnerable one. The secure version will be detailed in the later section dedicated to the mitigations in response to the showcased attacks. The matchmaking is all contained within two classes, Match and MatchSearch. MatchSearch is used by the ServerListDisplay class to search for available Match.

Match

A Match object is first initialized when called in MultiPlayer's host or join methods. It's required to be a passed the MultiPlayer object, so it can be set as an attribute called session. This is so that the MutiPlayer update method can be called as a messages are received from the other player. Match also has optional parameters for game settings that are set on the host's side. This is so that the host's game's settings can be

displayed on the ServerListDisplay page and also sent to players when joining their game.

When a Match is created on the host side, the session will call the start method. This is a method first calls the create host port method, which finds a valid port to host on within the designated range of game ports and creates a socket on that port for communication. It then creates a unique game ID. The game ID was originally used to show the game on the ServerListDisplay page but is later repurposed for creating a shared encryption key. Once a game ID is made and a port is determined, it will start a thread on the listening method. This waits for potential players to connect. If it receives a discover message, meaning the player is searching for a match, it will respond with the port and game settings so that information can be displayed in the ServerListDisplay page for that player. If the message received is a join request it will respond with the game information, save the connection information, and wait for a join notice from the same client. Once the player has connected the start communication method is called to start the game.

When a match is created on the client side, it will call the connect method. This creates a socket for communication and sends the join request, updates the match settings with what it receives from the host, and then sends the join notice. Before calling the start communication method like the host method does, it first calls the MultiPlayer create game method sending the game settings as arguments. This is another reason for having the MultiPlayer class passed as a Match attribute. When the game is hosted, this method was called internally using the selected settings as the arguments.

Once the connection is made the start connection method is called. This updates widgets to show that the connection was made, and the game has started. It also calls the MultiPlayer start game method, to create the game boards, so players can actually play the game. The last and most important thing it does is create a thread for the receive messages method. This method loops while the connection is active and calls the MultiPlayer update game method with the received message as an argument.

Another important method used after the connection is made, is the send messages method. This method is called from the game logic level and is passed a message argument. This message is used on the network layer to send game update messages to the other player.

The last method used in the Match class is the stop method. This method will first send a stop message to the other player, so that their game will also end in case the match ended prematurely. It will then clean up any running threads used to wait for a match or receive messages. Once it's done both of those, it will close the socket.

MatchSearch is created within ServerListDisplay and is used to port sweep the range of game ports and return a list of verified matches.

3.4 The Attack

The attack starts with analyzing how the application communicates messages across the network, so that the attacker can spoof messages, or even the entire connection. There were two main ways I did this in my example.

The first was using strace with an argument to specify network related system calls only. This involves running two applications with strace, starting a game between them, and outputting the system call logs into a file for analysis. First, I filtered and cleaned up the output file, then scanned it for suspicious things that looked like discovery messages and game data messages.

The second method used Wireshark. This method involves running two games like before, but this time with Wireshark running in the background under the option for "adapter for loopback traffic capture." I sorted this traffic by the protocol used, since there was a lot of unrelated TCP traffic and the game uses UDP. I was then able to find the packets being sent and could view the content within them.

Once the analysis for understanding how the game communicates was done, it was time to write a program to spoof messages. The first step was creating a socket to communicate with whichever port in the port range the attacker chooses. Next, the attacker might have to wait for the host to receive a connection so that it will start listening for messages. This can be done by waiting for a client to connect, starting up the application and connecting to that game, or the attacker could send their own discovery and join messages to initialize the connection. Once the host starts its connection and is ready to receive messages, the attacker can send fake game information, and it will affect the game state on the host side.

3.5 Mitigations

This project handles the mitigations of this attack in two primary ways. The first being symmetric encryption. The second is message signatures. With both of these protections the packet sniffing and message spoofing attacks can be prevented. Before implementing message signatures to prevent message spoofing, I needed a way to encrypt them, so encryption is where I started

Encryption implementation is handled through the previously mentioned hashlib and cryptography libraries. The former is used to create a hash of the globally used game key, where the ladder will utilize that hash to make a global game cipher used to communicate discovery information between players. It's also used in the communication of the individual game id when a connection is being made. This is important for how the signing message encryption is handled.

For signing messages, a session and message specific encryption scheme is used. The session specific cipher is created using a secret key derived from the game's id communicated at the point of connection. Messages are encrypted using this new cipher. Signed messages are then created using the previously mentioned hmac library and are derived using the previously encrypted message and secret key.

Messages are sent in a json format containing a message and signature, both of which are encrypted as previously described. A new function is used to verify signatures within the receive message thread. It uses the received message to recreate the signature in the same way it would when sending a message, and it compares this signature with the one received. If a message doesn't pass this verification step it is ignored.

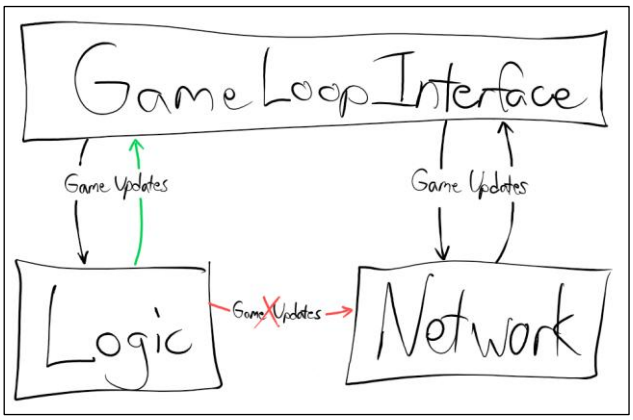
4 Evaluation / Analysis / Results

Overall, this was an amazingly enriching project. Once again, as with every project I do, I still see so much more potential and areas for improvement. I'm almost sad to be leaving the project but look forward to revisiting again with new perspectives! There are quite a few different ways I could've implemented some ideas and features. Like before when I worked on this game, I also see some areas where refactoring is needed.

4.1 Refactoring Application Layers

The main issue with the overall design is the lack of separation between the main layers of the application—application interface / gameplay loop, game logic, and networking. In the game's current iteration, all layers of the application talk to each other in some way, but it makes more sense for the game logic and network layers to be separated as shown in figure 1. The game logic layer directly calls the send message function within the networking layer to send game updates, but this could instead use the main application layer as a middleman between the two like it already does. I feel that this would greatly simplify and clarify how the game works. The layers are separated as three files, so calling object methods to objects created outside of the file can really make things confusing. Understanding the file pertaining to game logic shouldn't be required to understand how the game handles communication.

Fig. 1. Refactor/Segmentation of Application Layers' Communication



4.2 Further Separation of UI and Logic

Minesweeper and Tic-Tac-Toe cells both utilize Tkinter to have their own buttons associated with them. This is because the logic layer handles visual game updates internally, like marking a cell. Ideally, the visual part of these updates should be handled separately for best practice and clarity. It can be very confusing as to how game

updates are handled, especially with the object communication across files. If the cell classes were to continue to function this way, it would also make more sense for them to be buttons themselves, inheriting from that class. This would make button configurations like changing their text and other affects much simpler and straightforward.

5 Future Work & Conclusion

There are a few areas where this project could be expanded. The network security demonstration side could easily showcase more concepts, but other gameplay and networking aspects could be improved greatly as well.

5.1 Gameplay

Gameplay might not seem like an important feature, but for a project's whose goals is to showcase network security in an engaging and interactive way, this is an area that shouldn't be neglected. The gameplay of games like Tic-Tac-Toe and Minesweeper can be fun at first but quickly become boring.

One idea for spicing up gameplay would be to add things like power ups that can dynamically affect the gameplay. These could be things like resetting the opponents board, decreasing the bomb percentage, or even increasing the opponent's bomb percentage.

A simpler gameplay improvement idea would be adding a way to see the state of the opponent's minefield. This could be something simple like how many bombs they have left or actually seeing the current state of their board visually.

The first idea would require a rework on the networking on how game states are communicated and handled. Since there are more different ways that the game state can change, it would require a new way of formatting messages and interpreting them.

5.2 Scaling

A glaring issue in this project is the limitation to the localhost connection. It would be much better if this was easily playable across devices. This not only includes adjusting the way communication is handled, but also making the application easily functional across different operating systems and environments. With these changes, the project could be useful in a learning environment, where students could be given a version of the application and told to pen test for vulnerabilities. Alternatively, they could be given the source code and told to secure it from vulnerabilities.

5.3 Network Security Concepts

While this project already covers some interesting networking topics, there's still a lot of room for exploration. For the attacking side, DDoS and Man-in-the-Middle attacks could be showcased, as well as their mitigations. For the defense side, asymmetric encryption methods could be showcased as well.

This project also introduces an interesting research area to explore networking concepts in video games. Video games tend to use a mix of UDP and TCP with their own unique protocols on how communication is handled between servers and clients. Research could be done into how different styles of games communicate, and the security protections in place.

5.4 Testing Framework

One frustrating thing I found during the development and reworking of the networking was constantly testing if everything was working smoothly. Creating test frameworks for applications, especially security tests, would be something that not only greatly benefits the application and future development, but it could be a great learning opportunity for students. I feel that this is a part of software development and computer security that is talked about but never fully explored when it should be.

5.5 Conclusion

This project only scratches the service of what it could achieve. While I feel that I accomplished a lot, the room for growth is still plentiful, and these possibilities for expanding can cover many different areas of computer science. Potential areas include software development, secure coding and testing, and network security. A game like this presents an interesting way that students can interact with concepts they're learning in class.