Report – Group 1

Penetration Testing – Assignment 1

Graeme Brodie, Dee Dixon, Kyle MacQueen, Calvin Menezes, Andreea Petcu

1 Initial exploration

PwnAdventure3 uses a client-server model as a platform for its multiplayer functionality. Communication between client and server takes place over TCP using a custom protocol. We used Wireshark to capture and examine the traffic during the establishment of a game session in order to reverse engineer the protocol used by the game. This analysis immediately revealed two aspects of the protocol that would be key in our later work:

1. Initial Authentication Connection: The game makes an encrypted connection to port 3333 of the server during authentication.
2. Game World Connection: Upon successful authentication, the game then makes an unencrypted connection to a port in the range 3000-3009, and exchanges game world data.

Wireshark categorised traffic on the first connection as SSL. The presence of a TLS header in the authentication connection to port 3333 confirms that this connection is encrypted. This makes a cursory examination of the contents very difficult, so we ignored this connection for the remainder of our initial exploration.

ASCII-encoded text was immediately visible in packets transmitted in the second connection, suggesting a lack of encryption. We started searching for repeated units or common elements within this data in order to discern individual packet frames. We discovered several 2-byte candidates for packet headers (as on slide 3 of our presentation). We also noted that some packets with the same header candidates were the same length, and that packets with variable lengths often had certain structural features in common.

2 Networking

Based on the initial examination of the game's network protocol, we implemented a proxy to capture and decode packets. The aim of the proxy is to modify and inject packets of our own, as well as to analyse information available to the game client that was not shared with the player.

2.1 Protocol Decoding

In order to access and later manipulate information shared between the client and server, it is necessary to determine the structure of each packet in the protocol used by the game. From our initial examination we already suspected that packets consisted of an initial two-byte header followed by corresponding data, so we took a large array of captures from normal gameplay and identified as many of these different packet types as we could.
Without a frame of reference, decoding individual packets required an amount of guesswork. However, we quickly learned techniques for matching packet types to their functionality:

- Matching changes in the player character's behavior in the game to unknown packet types in the captures.
- Using ASCII text within packets as a clue to their purpose.
- Considering the characters used in headers as word abbreviations when making hypotheses.

For each packet type, we split the raw hex data into fields. Our process is described on slides 6-11 of our presentation. The full results of our protocol decoding are tabulated separately.

2.2 Proxy

We used the Python socket and threading libraries to achieve our initial goal of intercepting the client-server TCP connection and passing packets unparsed and unaltered. Our transparent MITM attack used a pair of threaded, duplex connections – one to the game client, and one to the server. We could observe packets sent in both directions and log them, aiding our continued protocol decoding work.

2.2.1 Switchboard

In order to support the proxying of connections made to different game worlds, we created a threaded `Switchboard` class that listened on all ten ports simultaneously. Switchboard threads could be activated seamlessly to proxy traffic and return to a listening state when the connection was transferred to a new port.

2.2.2 Packetisation

In order to ensure that our implementation was extensible so that we could keep adding new parsable packets, we opted to represent each type of packet as an extension of an abstract `Packet` class. Each class contained methods that allowed it to understand the specific structural information for that packet, as discerned via ongoing protocol decoding.

As discussed, a two byte header identifies packet type. We then instantiated the matching class to both contain and parse the data in that bytestring, extracting the useful information into a set of fields within that class. We found the Python `struct` library perfectly suited to this task, as it allowed us to automatically convert sections of the bytestring into their associated C++ types and store them as variables.

Packets can be of static or variable length, as encoded in a length field within the packet. We used this knowledge to extract multiple packets from the single concatenated bytestring. This avoided any issues wherein packet data could potentially contain pairs of bytes that matched headers but were within unrelated packets.

Our aim was to packetise all packet types in the game protocol, but as the number of discovered packets continued to grow, we found that this aim was infeasible given time constraints. As such, we decided to focus mainly on packets that were related to functionality we wanted to modify. For some of these packets we only parsed certain fields, extracting data we needed but leaving the rest, while other packets we had to leave unparsed, encapsulated in a generic `UnknownPacket` class instance.

Unfortunately, because the length of these unknown packets could not be discerned, all packets that came afterwards in the same bytestring couldn't be decoded. We do not consider this a flaw in our design however, as the eventual aim would be to parse and encapsulate every type of packet.

2.2.3 Packet analysis, modification, and injection

With useful packets in an easily readable and mutable format, we were able to easily examine packet fields of our choice. This allowed us to extract data available to the game client, but not necessarily available to the player of the game.

While we were able to print packet information to the console window, we wanted to streamline the interface so that a player could access useful information from within the game. To that end we developed a "chatback" function that injects a newly constructed chat packet into the server -> client connection, displaying information of our choosing in the chat window.

We then intercepted chat packets containing the text `/whereami`, set a flag within our packet analysis code, and then dropped the chat packet so it did not arrive at the server. When the packet analyser thread next received a position update packet from the client, the flag was checked, the position information extracted, and returned to the player via our chatback function. This covertly revealed previously hidden information to the player

We noted that most world objects are interacted with via the `ee` packet. We found that, using the list of IDs announced by the game server when initiating a connection to a new game world, we could interact with certain actors remotely by injecting these packets. We implemented a command `/poke` that allowed players to speak to NPCs from anywhere in the game and to pick up treasure without walking over it. However, we could not interact with certain game reward objects like eggs in this way and so hypothesised that there was a server check on player position associated with these objects.

To that end, we implemented a server-only teleport. We added a command that modified all position update packets from the client to instead contain a position of the player's choosing. This immediately placed the player character at that position from the server's position, though from the client's perspective the position hadn't changed. This `/servertp` command allowed us to pick up protected objects with `/poke`.

Using a similar technique, we were able to inject a packet that unlocked the pirate treasure chest, yielding treasure instantly even for newly created characters. We were also able to fire our weapons with injected packets and choose which direction the projectile travelled.

Our final major injection ability was the `/radar`. The server constantly updates the client on the position of nearby monsters and players, and that these updates occur even when entities were outside player visual range. We were able to use these in conjunction with the chatback feature to report the number of monsters and players nearby at set intervals.

2.2.4 Limitations

While we can inject any type of packet that we are able to parse, there are some aspects of the game world that are recorded and checked on the server, the client, or both. Injections that contradict these records have no effect. Some limitations can be overcome – for example, remotely collecting eggs via a server-only teleport. Others we could not find ways to circumvent:

- We could not give ourselves “free” items.
- We could not fire weapons we did not own.
- We could not lock or meaningfully modify our mana value or ammunition.
- We could not achieve true teleportation on both server side and client side via injection alone.
- Similarly, levitation could be achieved on the server by locking the player character’s Z position, but the client player position would still be grounded and subject to collision detection.
- We could not inject packets that would be interpreted by the server as having come from a different player.

We had also hoped to add new graphical elements to the user interface but were unable to do so in the time available.

2.2.5 Overriding the ConnectToGameServer function

While we initially modified `/etc/hosts` to force the game client to connect to our proxy, we later opted to develop a library preload. The server provides an address for a game world in the encrypted authentication connection; by overriding the function `ConnectToGameServer,` we force the game to ignore this address and replace it with `127.0.0.1`.

3 Overriding library functions with LD_PRELOAD

Position data on game objects is constantly updated between client and server. We exposed this communication via our proxy, but also wanted to explore client-side positional data. In addition, our packet modification teleportation was limited. This prompted us to develop binary exploitations that allowed true teleportation.

3.1 Getting location of players

To save on game resources, only positional data on the nearby players is given. We retrieve the location of nearby players by overriding library functions using LD_PRELOAD.

In order to find the nearby players connected to the same server, the **World**::**GetPlayersInRadius()** method was used. The function expects an argument of type **Vector3**, which is the current position of the active player, and a second argument of type **float**, which represents the radius we are interested in. Each **IPlayer*** object is cast into an **Actor** object, on which the **GetPosition()** method is called to extract the position. **Vector3** objects contain the x, y, z coordinates, which are written to a CSV file.

Our positional data for the nearby players is updated on a regular basis in order to ensure that it accurately represents the state of the game world. We achieve this by overriding the **World::Tick()** method. Due to the high call rate of **World ::Tick()**, which prevents the complete write of positions to the CSV file, controls have been created such that the location extraction code is executed every 25th call.

3.2 Teleportation and Enhanced Mobility

Our packet injection teleports do not update the client side. Once again, the tick function was overloaded to allow us to continuously retrieve and update the position of the player.

We overrode **Player::Chat()** using **LD_PRELOAD**. This enabled us to expose an interface for our teleportation and mobility functionality to the user, including teleportation to named locations and coordinates and increased movement speed and jump height. We parse the chat function’s **const char*** parameter.

The teleport command **tp** allows location-based teleportation with names associated with coordinates in our code (e.g. *Town*) or vector-based teleportation with x, y, and z coordinates. We used the game’s **Vector3** datatype, with its constituent float compnents, to store coordinates and the **Player::SetPosition(const Vector3 *)** to achieve teleportation. We also call the original **Player::Chat** function at the end of this loop to ensure that normal chat functionality stays implemented.

When we display messages in the chat, we include the **[!CHAT]** flag in the text. The proxy can identify these messages and only display the message to the player, rather than all players in the game. This way other players on the server are unable to see the user's private messages and commands that are being used.

4 Mapping

Several libraries were tried and then removed as ease of use was important and they had too many dependencies that were messy to install. We selected **NumPy** arrays for processing CSV files and **matplotlib** for all graphical components and updating of data. The time and random libraries are not needed to be functional but help with debugging and ensured the code ran smoother with the proxy. Because the usual use of matplotlib is graphing that only needs to be displayed once, the figure must be continuously deleted and then recreated (**fig.canvas.draw()** and **fig.canvas.flush_events()**).

4.1 Local vs Global

Slide 17 shows that the figure consists of two maps, global and local. These both use the same files to gain data and the same color schemes. However, while the global map's coordinate system is static and the player moves within it, both the map and player need to update locally. The elegant solution to achieve this was to constantly change the limits of the map (**set_xlim** and **set_ylim**) based on the player position which was found after several failed attempts at zooming into the global map.

4.2 Map Click Teleportation

Teleporting by clicking on map was possible by combining our previous developments.

Using Matplotlib, map clicks are registered with an event manager using **mpl_connect()** which calls the function **onclick(event)**. The click event is bound to the figure and returns the screen position, whether the click was single or double and the position on the graph axis when clicked. The graph coordinates are used in the following stage after being outputted to a CSV file.

The **World::Tick()** function contains the same teleportation implementation used in **Player::Chat()**. Every 25 iterations, The x and y are then read from the CSV file. The z coordinate is always set to be the maximum height on the map, which was found to be approximately 15500, and the player slowly falls on the ground.

5 Abandoned Functionality

Proxy:

- Proxying the authorisation connection was unnecessary due to encryption.
- Functionality to log specific data to other components via TCP connections
- Augmentation of the game connection initialisation packets with a library override that allowed the proxy to listen on only one port but pass through to a game client specified port on the server - worked, but was prone to race condition bugs.

Teleportation:

- Having continuous teleportation within a small area so the player was hard to hit

Map:

- A zoomed in view of the map picture for local
- 3D mapping (hence the inclusion of z axis data); Severe graphical limitations requiring complex code workarounds were outside of scope with time constraints
- Implementing the map as a graphical overlay within the game – had too many difficulties accessing this data in the time given