Minecruft

Jun 21, 2021

CONTENTS:

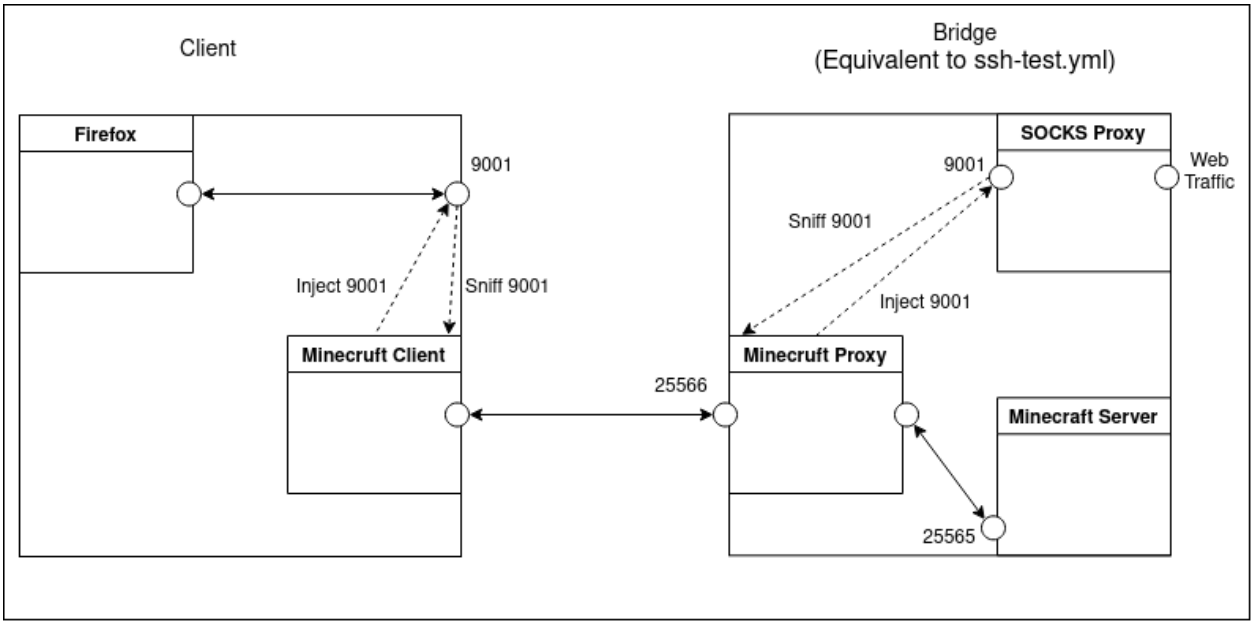
CHAPTER

ONE

MINECRUFT: A MINECRAFT PROTOCOL PROXY

A protocol proxy PT that transforms traffic into Minecraft Traffic. This transport aims to go beyond FTE and actually transform traffic into running Minecraft Sessions with a real Minecraft Server.

1.1 Basic Configuration



1.2 Installation

To install run:

```
git clone git@130.127.248.99:ntusing/minecruft.git
cd minecruft
```

Then run,

```
sudo ./install/setup ./
```

1.3 Before Running the PT

Always run this command after booting. If you do not, the transport will not work properly. This is due to the fact that TCP RST packets much be ignored on your local machine. Otherwise, services like Firefox will see that no service is listening on proxy port 9001 and will terminate the TCP session.

```
sudo utils/iptables_prestart
```

1.4 SOCKS End to End Demo

This demo will set up the environment shown in the basic configuration section.

The .yml files in the docker/ folder contain test configurations for the docker-compose command. For example,

```
cd docker
sudo docker-compose -f socks-test.yml up mserver
```

and in a separate terminal window

```
sudo docker-compose -f socks-test.yml up --build testproxy sshproxy
```

will bring up the minecruft proxy, minecraft server, and a SOCKS proxy in docker containers with a shared network namespace (this is separate from the client's network namespace).

Then using docker-compose in another terminal window, start the Minecruft client using the command.

```
sudo docker-compose -f socks-test.yml up --build testclient
```

However, this docker container shares the host network namespace - meaning that the Minecruft client will be able to sniff packets from Firefox, TOR, iperf, netcat, or any other process running on the host machine. Specifically if you look at the configuration file for the service testclient, you will see

```
testclient:
  build:
    dockerfile: ./docker/client/Dockerfile
    context: ..
  network_mode: host
```

Note that the network_mode is set to "host".

Finally, to demonstrate that the service is running, set the Firefox Proxy settings to listen on 127.0.0.1 on port 9001.

1.5 Stopping the Demo

To cleanly stop the demo run

```
docker-compose -f socks-test.yml down
```

1.6 Running the PT Client without Docker

```
sudo python3 minecruft/Minecruft_Main.py client configs/aes_ecb_key.bin 127.0.0.1 25566␣
↪9001 configs/client_encoder_actions_v12.2.json configs/client_decoder_actions_v12.2.
↪json
```

1.7 Running the PT Proxy without Docker

```
sudo python3 minecruft/Minecruft_Main.py proxy configs/aes_ecb_key.bin 127.0.0.1 25565␣
↪9001 configs/client_decoder_actions_v12.2.json configs/client_encoder_actions_v12.2.
↪json
```

1.8 Running just the Minecraft Server

A minecraft server is needed for the proxy/client to run correctly. To run and build a server listening on port 25565, run

```
cd docker/
sudo docker-compose -f socks-test.yml up mserver
```

However, the instructions in SOCKS End to End Demo handle running the Minecraft server.

1.8.1 Troubleshooting

Running utils/iptables_prestart must be done upon each reboot/iptables reset. This is due to the fact TCP RST packets must be ignored for listening to unused TCP ports.

Note that the docker daemon must be running or docker commands and docker-compose will not function properly. Use

```
systemctl status docker
```

to check if docker is running, and start docker with

```
systemctl start docker
```

CHAPTER

TWO

DEVELOPER NOTES

2.1 Runtime Logic Flow

Much of the Runtime logic flow of the program can be seen in Minecruft_Main.py. The primary flow is as follows:

1. Sniff TCP traffic sent to dest_port
2. Encrypt the queued sniffed traffic
3. Dequeue each encrypted packet, prepend length of packet
4. Encode into Minecraft Action (After bot reaches play state)
5. Decode Minecraft Action on the proxy side
6. Wait until length bytes are decoded and queue as whole TCP packet
7. Decrypt TCP Packet
8. Inject packet onto proxy's localhost

Note that this ordering occurs slightly differently from the proxy's perspective. Since the client transmits first, the proxy would run 5-8 first and then 1-4. Note that each one these functions will all need to occur simultaneously.

2.2 Setup Logic Flow

Before the primary setup flow is started, arguments are parsed (using docopt in Minecruft_Main), encoder/decoder functions are added dyanamically (if defined in json files), and minecraft sessions are started. More detail about the encoder/decoder functions is discussed in the following sections.

2.3 Adding New Encoder/Decoders

There are two ways to add encoder/decoder pairs to minecruft: statically or dyanamically. Note that both ways require defining the encoder function on one endpoint (Client or Proxy)

2.3.1 Quarry's Packet Handlers

Quarry already defines functions for receiving every minecraft packet type, and registers events for them when a user overrides these functions. For example, look at the Minecraft Networking Protocol (for version 1.12.2) at the use entity packet. If you wanted to print "hello world" every time one of these packets was sent you could overwrite the function:

```
packet_upstream_use_entity
```

within the MinecraftProxyEncoder file within the MinecraftProxyBridge class. This would be the resulting code:

```
packet_upstream_use_entity(self, buff):
    self.logger.info("hello world")
```

Whenever a packet_EVENTNAME is registered from the minecraft protocol, Quarry will register that function (if EVENTNAME exists in the protocol and quarry's builtin csv listings) and will trigger a callback to the packet function if a minecraft action packet of EVENTNAME is received. This means event handling can be done transparently, and the developer can simply add desired functionality to each handler. If you want to learn more, look at quarry's protocol and dispatcher classes/functions.

If you also wanted to forward the received packet to the minecraft server, you would also add:

```
packet_upstream_use_entity(self, buff):
    self.logger.info("hello world")
    self.upstream.send_packet("use_entity", buff.read())
```

Look into Quarry's documentation about it's buffer class to learn more (it's based on structs in python). Also look into Minecraft's data types that are used. For packets that end with minecraft NBT metadata, put the byte value 255 to signal the end of the metadata, otherwise the packet will be malformed and the minecraft server will not accept it.

2.3.2 Adding Static/Explict Encoders/Decoders

Quarry's builtin packet handlers are used for defining decoders within Minecruft. There are slight differences between decoders defined in the client (MinecraftClientEncoder) and proxy (MinecraftProxyBridge). A static decoder within the client code should look like this:

```
var1 = buff.unpack(fmt)
var2 = buff.unpack_varint()
...
self.incoming_decode_buffer.append(var1)
self.incoming_decode_buffer.append(var2)
...
update_incoming_buffer(self, self.downstream_factory.receiving_packet_queue, buff)
```

and a static decoder within the proxy will look something like this (note the additional save/restore functions):

```
buff.save()

var1 = buff.unpack(fmt)
var2 = buff.unpack_varint()
...
self.incoming_decode_buffer.append(var1)
self.incoming_decode_buffer.append(var2)
...
buff.restore()
```

(continues on next page)

(continued from previous page)

```
self.downstream.send_packet(packet_name, buff.read())
update_incoming_buffer(self, self.downstream_factory.receiving_packet_queue, buff)
```

The encoders for the client and proxy also look slightly different. However, the entry point for handling encoder functions is the encoder() function in both MinecraftClientEncoder and MinecraftProxyBridge. This function is registered to execute every game tick (1/20th second) in the packet_player_position_and_look (client) and packet_downstream_player_position_and_look (proxy) functions. A client encoder will look something like this:

```
#Add more bytes to the buffer if needed (typically this is wrapped in an if statement)
check_buff(self.forwarding_packet_queue, self.outgoing_encode_buffer)

#This used to be get_byte_from_buff(self.outgoing_encode_buffer, num_bytes)
buff = ByteChopper(self.outgoing_encode_buffer)
consumed_bytes = buff.chop(num_bytes)

#Note that multiple self.packs for different data types will be included in call to send␣
↪packet depending on the packet event name
self.send_packet('packet_event_name', self.pack(fmt, consumed_bytes[:somerange]), more␣
↪packing calls...)
```

and a proxy encoder will look something like this:

```
check_buff(self.forwarding_packet_queue, self.outgoing_encode_buffer)
buff = ByteChopper(self.outgoing_encode_buffer)
consumed_bytes = buff.chop(num_bytes)

#The difference here are the fact that bridges have upstreams (real minecraft server␣
↪facing) and downstreams (client/bot facing)
self.downstream.send_packet('packet_event_name', self.pack(fmt, consumed_
↪bytes[:somerange]), more packing calls...)
```

Note that adding a new encoder/decoder requires looking at the required data types for that packet in the minecraft protocol wiki. Fortunately, by using a json defined version of the protocol (see Python Minecraft Data Repository) encoders and decoders can be defined dynamically. However, defining functions dynamically will only allow packing based off fields and the fields types. A rule description can be added to the dynamic functions to enforce action specific rules, but a module that parses and executes these rules has not yet been implemented. If no specific rule is required, then it is simiplier to define dynamic rules.

2.3.3 Dynamic/Implicit

Instead of defining generic rules for each packet type, the functions in the Dynamic_Method_Loader can load json defined functions into the classes and set up handlers for these functions. See create_decoder_function, create_encoder_function, encode_action, and decode action for more details. Also, review the json files in the configs folder to see how each action is defined by its fields and data types. Note, that new dynamically defined encoder actions will need to be directly called in the encode() function.

2.4 References

Minecraft Networking Protocol (for version 1.12.2)

Minecraft Bot and Proxy Library

Quarry Documentation

Minecraft Data Viewing Tool

Python Minecraft Data Repository

CHAPTER

THREE

MINECRUFT

3.1 DynamicMethodLoader module

Bytearray parsing tools and Encoder/Decoder templates

This file provides functions for dynamically adding encoder and decoder functions to the MinecraftProxyBridge and MinecraftClientEncoder (M) classes. It also includes the primary functions for parsing the incoming and outgoing byte buffers.

class DynamicMethodLoader.**ByteChopper**
Bases: bytearray

Simple class for chopping bytearrays

chop(*end*)
Chops from beginning of bytearray up to num_bytes. Stores the remaining bytes in self.

Returns num_bytes from the beginning of the bytes

Return type bytearray

DynamicMethodLoader.**check_buff**(*fqueue*, *buff*)
Checks when a buffer or queue is emptied. When a buffer is emptied, that means a full TCP packet has been transmitted to the client. When a full TCP packet is transmitted, it signals the Minecruft client with an encode_enemy_look packet.

DynamicMethodLoader.**create_decoder_function**(*cl*, *prefix*, *name*, *fmt*, *mode*)

DynamicMethodLoader.**create_encoder_function**(*cl*, *name*, *fmt*)
Currently for special minecraft types, we will assume no bytes of data can be packed.

DynamicMethodLoader.**decode_action**(*unpacker*, *incoming_decode_buffer*, *fmt*)

DynamicMethodLoader.**encode_action**(*self*, *packer*, *outgoing_encode_buffer*, *fmt*)
Self expects the type to be a Minecraft Encoder (either Client or Proxy).

DynamicMethodLoader.**pack_bool**(*packer*)
A placeholder function so booleans aren't encoded

DynamicMethodLoader.**pack_vint**(*packer*)
A placeholder function so pack

DynamicMethodLoader.**safe_unpack**(*f*, *unpacker*)

DynamicMethodLoader.**unpack_bool**(*unpacker*)

DynamicMethodLoader.**unpack_slot_item**(*unpacker*)

DynamicMethodLoader.**unpack_varint_to_bytes**(*unpacker*)

DynamicMethodLoader.**update_incoming_buffer**(*encoder*, *encoder_queue*, *buff*)

3.2 MinecraftClientEncoder module

Python 3 module with core classes for establishing and maintaing Minecraft Client connections, encoding encrypted byte streams, and factory class as an interface to build different types of Minecraft Class Encoders.

class MinecraftClientEncoder.**MinecraftClientEncoder**(*factory*, *remote_addr*)
Bases: quarry.net.client.SpawningClientProtocol

The MinecraftClient Encoder is a class that represents a Minecraft Client connection and performs the FTE encoding of an encrypted bit stream.

check_entity(*data*)
Checks an entity if its ID is over 20000. Should be depreciated due to how enitity IDs are assigned.

encode()
Encode packet bytes as minecraft movements. Currently just encodes as creative mode inventory actions, but this can be expanded to other movement types. This is where any new encoder functions should be called.

encode_chat_message()

encode_player_position()
Encode and send original player's position from the start of the game.

encode_player_position_and_look()
Not currently used. Bytes can be injected in to the expected fields for the server such as x, y, z, yaw, and pitch.

packet_player_position_and_look(*buff*)
Receives the inital player position from the server and store the player's information for other turns.

update_player_full()
Sends a player's position to the server every 20 ticks (1 second).

class MinecraftClientEncoder.**MinecraftEncoderFactory**(*encoder_actions=None*, *decoder_actions=None*, *profile=None*, *f_queue=None*, *r_queue=None*)
Bases: quarry.net.client.ClientFactory

Factory for building Client Connections. Also serves as the interface for the two packet queues for encoded packets being set between the client and server of the Pluggable Transport.

protocol
alias of *MinecraftClientEncoder.MinecraftClientEncoder*

3.3 MinecraftPlayerState module

class MinecraftPlayerState.**MinecraftPlayer**
Bases: object

Simple class for keeping track of a player's state

property position

update_position(*coords: list*)
Update the player's position coordinates for tracking position updates.

3.4 MinecraftProxyEncoder module

class MinecraftProxyEncoder.**MinecraftProxyBridge**(*downstream_factory*, *downstream*)
Bases: quarry.net.proxy.Bridge

Class that holds references to both Upstream and Downstream. Waits for both client (Upstream) and server (Downstream) connections to be established before forwarding packets through the proxy.

downstream_disconnected(*reason=None*)
Called when the connection to the remote client is closed.

encode()
This is the main encode function that encodes packet bytes as minecraft movements. This is where any new encoder functions should be called.

events_enabled = False

gen_rand(*bound*)
Uses os.urandom to generate random integer.

get_byte_from_buff(*buff*)
Right now returns 256 for no bytes in buff. Later change to None to be more Pythonic.

get_bytes_from_buff(*buff*, *num_bytes*)
Read num_bytes from a buffer. Returns a list of bytes.

packet_downstream_entity_head_look(*buff*)

packet_downstream_entity_look(*buff*)

packet_downstream_player_position_and_look(*buff*)

packet_upstream_creative_inventory_action(*buff*)

packet_upstream_player_position(*buff*)

quiet_mode = False

send_packet(*name*, *buff*)

spawn_mobs(*player_position*)
Spawn mobs in a random position in a sqaure 255x255 blocks centered on the player's starting position. This function should not be used if the Minecraft server is meant to generate mobs.

upstream_factory_class
alias of *MinecraftProxyEncoder.UpstreamEncoderFactory*

class MinecraftProxyEncoder.**MinecraftProxyFactory**(*downstream_encoder_actions=None*, *upstream_decoder_actions=None*, *c_p_queue=None*, *s_p_queue=None*)
Bases: quarry.net.proxy.DownstreamFactory

Factory class for easy instantiation of new Bridge connections between client and server.

bridge_class
alias of *MinecraftProxyEncoder.MinecraftProxyBridge*

client_bound_buff = bytearray(b'')

connectionMade()

forwarding_packet_queue = None

motd = 'Proxy Server'

num_client_encoders = 0

num_waiting_encoders = 0

sync_buff(*oldbuff*)
Each Downstream must send the same data for enemy movement packets from the global buffer in order for the game sessions to be authentic.

class MinecraftProxyEncoder.**UpstreamEncoder**(*factory*, *remote_addr*)
Bases: quarry.net.proxy.Upstream

Inherits from Upstream Class. Represents the Client Connection to the Minecraft Server. Contains a buffer <Proxy_bound_buff> for holding packets destined for the Minecruft Client.

assigned_enemy = 0

assigned_id = 0

class MinecraftProxyEncoder.**UpstreamEncoderFactory**(*profile=None*)
Bases: quarry.net.proxy.UpstreamFactory

Factory class for Upstream encoder for easy instantiation of multiple Upstreams for multiple Minecraft PT clients.

protocol
alias of *MinecraftProxyEncoder.UpstreamEncoder*

3.5 Minecruft_Main module

Usage:

Minecruft_Main.py client <aes_keyfile> <dest_ip> <dest_port> <fwd_ports> <encoder_actions_file> <decoder_actions_file> [options]

Minecruft_Main.py proxy <aes_keyfile> <dest_ip> <dest_port> <fwd_ports> <encoder_actions_file> <decoder_actions_file> [options]

Options: -h –help Show help message –proxy_port <proxy_port> Proxy Listen Address [default: 25566] -t –test Send test message through Minecruft instead of TCP packets

Minecruft_Main.**client_forward_packet**(*encrypt_tcp_q*, *decrypt_tcp_q*, *encoder_actions*, *decoder_actions*, *forward_addr*, *proxy_port*)
Starts twisted's event listener for sending and recieving minecraft packets, and sets up the client bot to connect to the proper host machine using the runargs function. Also passes the encrypt_tcp_q (for forwarding encrypted packets) and the decrypt_tcp_q (for receiving encrypted packets) to the minecraft client encoder object.

Minecruft_Main.**decrypt_enc_data**(*decrypt_tcp_q*, *response_q*, *key*)
This is a function for receiving packets on the host.

Removes packets from decrypt_tcp_q, decrypts them with AES_ECB, and then stores the encrypted packets into the reponse_q.

Minecruft_Main.**decrypt_load**(*cipher_str*, *key*)
Encrypts with AES_ECB using a default password. This is not for security it is for creating an even chance of sending binary ones and zeros.

Minecruft_Main.**encrypt_load**(*message*, *key*)
Encrypts with AES_ECB using a default password. This is not for security it is for creating an even chance of sending binary ones and zeros.

Minecruft_Main.**encrypt_tcp_data**(*incoming_tcp_q*, *encrypt_tcp_q*, *direction*, *key*)
This is a function for sending packets from the host.

Removes packets from incoming_tcp_q, encrypts them with AES_ECB, and then stores the encrypted packets into the encrypt_tcp_q.

Minecruft_Main.**filter_packets**(*incoming_tcp_q*, *duplicate_packets*)
Factory Function for generating filters

Minecruft_Main.**inject_tcp_packets**(*response_q*)
Takes the received unencrypted response_q TCP packets and injects them onto the localhost's machine.

Minecruft_Main.**proxy_forward_packet**(*encrypt_tcp_q*, *decrypt_tcp_q*, *downstream_encoder_actions*, *upstream_decoder_actions*, *minecraft_server_addr*, *minecraft_server_port*, *listen_addr*, *listen_port*)
Starts twisted's event listener for sending and recieving minecraft packets, and sets up the proxy server which first connects to a Minecraft server set in offline mode and then waits for incoming Minecraft Client connections. Also passes the encrypt_tcp_q (for forwarding encrypted packets) and the decrypt_tcp_q (for receiving encrypted packets) to the minecraftProxyEncoder object.

Minecruft_Main.**receive_tcp_data**(*tcp_port*, *direction*, *incoming_tcp_q*)
This is a function for sending packets from the host.

Sniffs all packets of localhost that have ports with the value tcp_port traveling either from src or dst using the direction variable.

direction: src or dst tcp_port: String resembling TCP ports such as 9000, 20, 80 incoming_tcp_q: Queue for saving incoming packets

Minecruft_Main.**runargs**(*encoder_actions*, *decoder_actions*, *args*, *encrypt_tcp_q*, *decrypt_tcp_q*)
Creates a Minecraft Client Bot, and connects to the Minecraft proxy. This assumes that the Minecraft Proxy is already running on another machine.

Minecruft_Main.**sim_tcp_data**(*incoming_tcp_q*, *encrypt_tcp_q*, *direction*, *key*)

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

m

INDEX

N

P

Q

R

S

U