

Mục lục

1	Giới thiệu	2
1.1	Tổng quan dự án	2
1.2	Mục tiêu và nhiệm vụ	2
2	Kiến trúc hệ thống	3
2.1	Kiến trúc tổng thể	3
2.2	Kiến trúc chi tiết	3
2.2.1	Thiết kế rat-server	4
2.2.2	Thiết kế rat-client	6
2.3	Đồng bộ hóa gửi-nhận	7
3	Giao thức truyền tin	8
3.1	Trao đổi shared secret	8
3.2	Định dạng và mã hóa packet	10
4	Sản phẩm	10
4.1	Đóng gói và triển khai	10
4.2	Kết quả thực thi	11

1 Giới thiệu

Báo cáo trình bày chi tiết về thiết kế, phát triển và kiểm thử Remote Access Tool (RAT), một công cụ quản trị từ xa theo mô hình client-server an toàn. Mục tiêu chính của dự án này là cung cấp cho quản trị viên hệ thống một phương tiện hiệu quả và đáng tin cậy để quản lý nhiều máy từ xa trong khi tuân thủ các tiêu chuẩn nghiêm ngặt về an ninh và hiệu suất.

1.1 Tổng quan dự án

Dự án bao gồm việc phát triển hai thành phần chính: ứng dụng server (máy chủ) và ứng dụng client (máy khách), được thiết kế để hoạt động trên các máy khác nhau. Thành phần server đóng vai trò là điểm kiểm soát trung tâm, có khả năng đưa ra lệnh cho nhiều client cùng lúc. Thành phần client, được cài đặt trên các máy từ xa, thực thi các lệnh này và trả kết quả về server. Giao tiếp giữa client và server được bảo mật bằng thuật toán `chacha20-poly1305@openssh.com`. Khi bắt đầu kết nối, client và server trao đổi `shared secret` dựa trên thuật toán `curve25519-sha256`.

1.2 Mục tiêu và nhiệm vụ

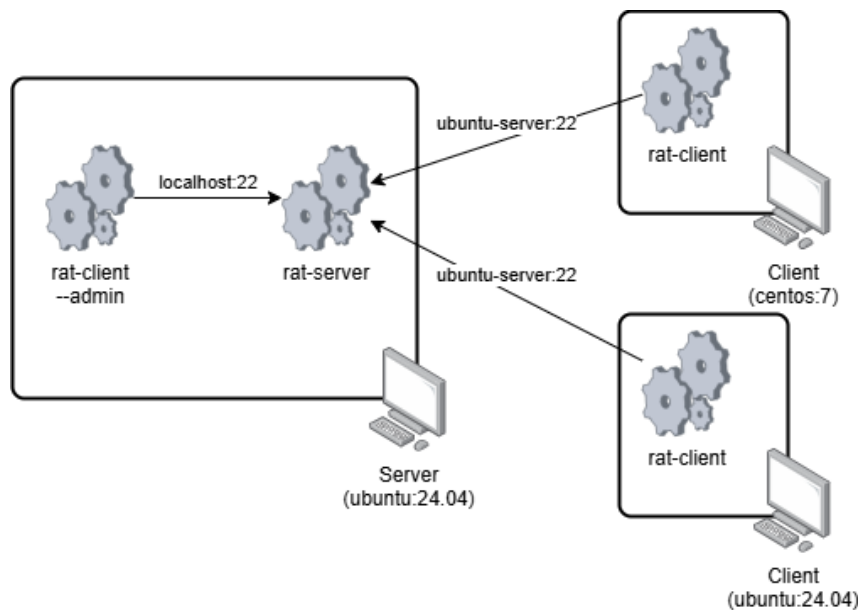
Mục tiêu chính của dự án này là tạo ra một công cụ quản trị từ xa ổn định và an toàn đáp ứng các yêu cầu chức năng và phi chức năng được chỉ định.

Các nhiệm vụ chính bao gồm:

- Phát triển thành phần server có khả năng điều khiển nhiều client.
- Triển khai chức năng phía client, cho phép server:
 - Liệt kê tệp và thư mục.
 - Truy xuất nội dung tệp, bao gồm cả các tệp lớn hơn 1 GB.
 - Liệt kê các tiến trình đang chạy.
 - Kết thúc các tiến trình được chỉ định.
- Đảm bảo giao tiếp an toàn giữa client và server, chủ yếu bằng cách tận dụng giao thức SSH.
- Đóng gói trình cài đặt cho thành phần client để dễ dàng triển khai trên Ubuntu 24.04 và CentOS 7.
- Phát triển ứng dụng theo nguyên tắc thiết kế hướng đối tượng (OOD), với các kiểm thử đơn vị toàn diện.
- Duy trì mức tiêu thụ tài nguyên thấp (CPU <5% trên 1 lõi, Bộ nhớ <100 MB) cho cả thành phần client và server.
- Đảm bảo thành phần server tương thích với Ubuntu 24.04 và thành phần client tương thích với Ubuntu 24.04 và CentOS 7.
- Phát triển hệ thống không có lỗ hổng bảo mật đã biết.

2 Kiến trúc hệ thống

2.1 Kiến trúc tổng thể



Hình 1: Các kết nối tới tiến trình `rat-server`

Hệ thống triển khai trên một máy server sử dụng hệ điều hành Ubuntu 24.04 và một hoặc nhiều máy client sử dụng hệ điều hành Ubuntu 24.04 hoặc CentOS 7. Trên máy server sử dụng file khóa bí mật (ví dụ tại `/host`) chứa khóa bí mật và công khai RSA. Phần khóa công khai được sử dụng để client xác thực kết nối tới server và phần khóa bí mật được sử dụng để server xác thực chế độ admin của client.

Trên máy server có 2 tiến trình được thực thi:

- Tiến trình chạy thành phần `rat-server` chấp nhận các kết nối TCP từ port 22 (port number có thể được điều chỉnh khi bắt đầu chạy).
- Tiến trình chạy thành phần `rat-client` trong chế độ admin và giao tiếp với tiến trình chạy `rat-server` thông qua port 22 ở trên. Sau khi trao đổi khóa thành công, tiến trình này sử dụng chính file khóa bí mật RSA để xác thực quyền admin với `rat-server`.

Khi server muốn thực thi một yêu cầu trên máy client, người dùng nhập lệnh vào standard input của tiến trình `rat-client` (admin mode), tiến trình này gửi yêu cầu đến `rat-server` và chuyển tiếp cho `rat-client` đích trên máy client. Quy trình này tương tự lệnh `"docker exec"` của Docker: `rat-client` admin đóng vai trò terminal trên máy host, `rat-server` đóng vai trò docker daemon và `rat-client` đóng vai trò container.

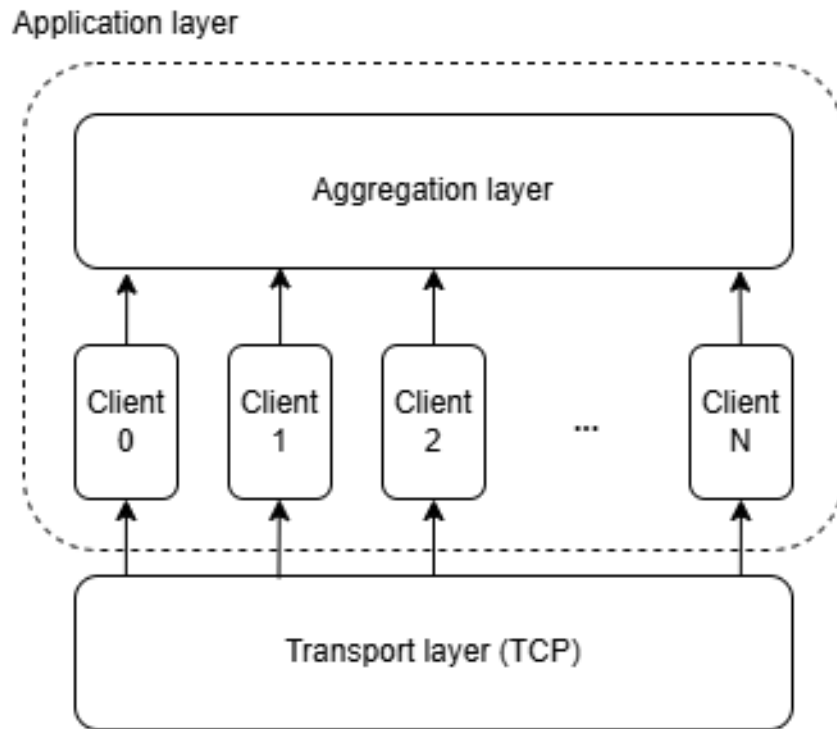
2.2 Kiến trúc chi tiết

Hệ thống được cài đặt bằng Rust [1] - ngôn ngữ cho phép biên dịch thành các file thực thi với hiệu năng cao, an toàn về bộ nhớ mà không cần sử dụng garbage collector.

Repository được host tại <https://github.com/Serious-senpai/remote-access-tool>, bao gồm 1 Rust workspace với 3 project: `rat-client`, `common` và `rat-server`. Trong đó `rat-client` và `rat-server` được dùng để biên dịch ra các file thực thi, `common` chứa các phần mã nguồn dùng chung chia sẻ giữa các file thực thi đó.

Hệ thống sử dụng `crate tokio.rs` [2] để quản lý việc thiết lập và ngắt kết nối TCP cũng như việc giao tiếp giữa các *task* thông qua các *channel*.

2.2.1 Thiết kế rat-server



Hình 2: Kiến trúc rat-server

Thành phần rat-server gồm một *aggregation layer* có nhiệm vụ tổng hợp và xử lý các packet đến từ một hoặc nhiều *client node*. Các layer và node được cài đặt như sau:

```

1  /// Packet aggregation layer
2  #[derive(Debug)]
3  pub struct AggregationLayer<C>
4  where
5      C: Cipher + 'static,
6  {
7      _listener: TcpListener,
8      _host_key: Vec<u8>,
9      _private_key: PrivateKey,
10
11     /// Mapping from client addresses to their appropriate contexts.
12     _clients: RwLock<HashMap<SocketAddr, ClientLayerCtx<C>>>,
13
14     /// Clone this sender to new incoming clients so that they can send packets to this aggregation node.
15     /// We can also subscribe to this sender to receive packets from all clients.
16     _primordial_client_sender: broadcast::Sender<(SocketAddr, Packet<C>)>,
17
18     _exit: Arc<Notify>,
19 }
20
21 impl<C> AggregationLayer<C>
22 where
23     C: Cipher + 'static,
24 {
25     pub fn new(
26         listener: TcpListener,
27         host_key: Vec<u8>,
28         private_key: PrivateKey,
29         capacity: usize,
30     ) -> Self;
31
32     async fn _handle_packet(self: Arc<Self>, origin: SocketAddr, packet: Packet<C>);

```

```

33  async fn _handle_connection<K, H>(self: Arc<Self>, socket: TcpStream, addr: SocketAddr)
34  where
35      K: KexAlgorithm,
36      H: HostKeyAlgorithm;
37
38  async fn _poll_packets(self: Arc<Self>);
39  async fn _poll_new_connections<K, H>(self: Arc<Self>)
40  where
41      K: KexAlgorithm,
42      H: HostKeyAlgorithm;
43
44  pub async fn listen_loop<K, H>(self: Arc<Self>)
45  where
46      K: KexAlgorithm + 'static,
47      H: HostKeyAlgorithm + 'static;
48  }

```

Code 1: Attribute và method signature của *aggregation layer*

Một *aggregation layer* bao gồm một `_listener: TcpListener` lắng nghe các kết nối TCP tới, `_host_key` và `_private_key` là cặp khóa công khai và bí mật RSA (được lưu dưới 2 dạng khác nhau để thuận tiện cho 2 mục đích sử dụng khác nhau). Bên cạnh đó, `_primordial_client_sender` là một multi-producer-multi-consumer channel: nó nhận các cặp (địa chỉ client, packet) được gửi lên từ các *client node* và broadcast các cặp giá trị này đến mọi subscriber. Ngoài ra, `_clients` là ánh xạ từ địa chỉ của mỗi client đến một struct `ClientLayerCtx`:

```

1  #[derive(Debug)]
2  struct ClientLayerCtx<C>
3  where
4      C: Cipher + 'static,
5  {
6      pub is_admin: bool,
7      // pub task: JoinHandle<()>,
8      pub ptr: Arc<ClientLayer<C>>,
9  }

```

Code 2: Cấu trúc `ClientLayerCtx`

Mỗi `ClientLayerCtx` chứa field `is_admin` cho biết client này có được xác thực là admin mode không và `ptr` chỉ đến một struct `ClientLayer` (tương ứng với một *client node* như trong hình 2):

```

1  /// Client communication layer
2  #[derive(Debug)]
3  pub struct ClientLayer<C>
4  where
5      C: Cipher + 'static,
6  {
7      _addr: SocketAddr,
8      _ssh: SSH<C>,
9
10     pub version: String,
11 }
12
13 impl<C> ClientLayer<C>
14 where
15     C: Cipher + 'static,
16 {
17     pub async fn accept_connection<K, H>(
18         socket: TcpStream,
19         addr: SocketAddr,
20         host_key: Vec<u8>,
21         private_key: PrivateKey,
22     ) -> Result<Self, Box<dyn Error + Send + Sync>>
23     where
24         K: KexAlgorithm,
25         H: HostKeyAlgorithm;
26
27     pub async fn send<P>(&self, payload: &P) -> Result<Packet<C>, Box<dyn Error + Send + Sync>>

```

```

28     where
29         P: PayloadFormat;
30
31     pub async fn listen_loop(self: Arc<Self>, sender: broadcast::Sender<(SocketAddr, Packet<C>>>);
32 }

```

Code 3: Attribute và method signature của *client node*

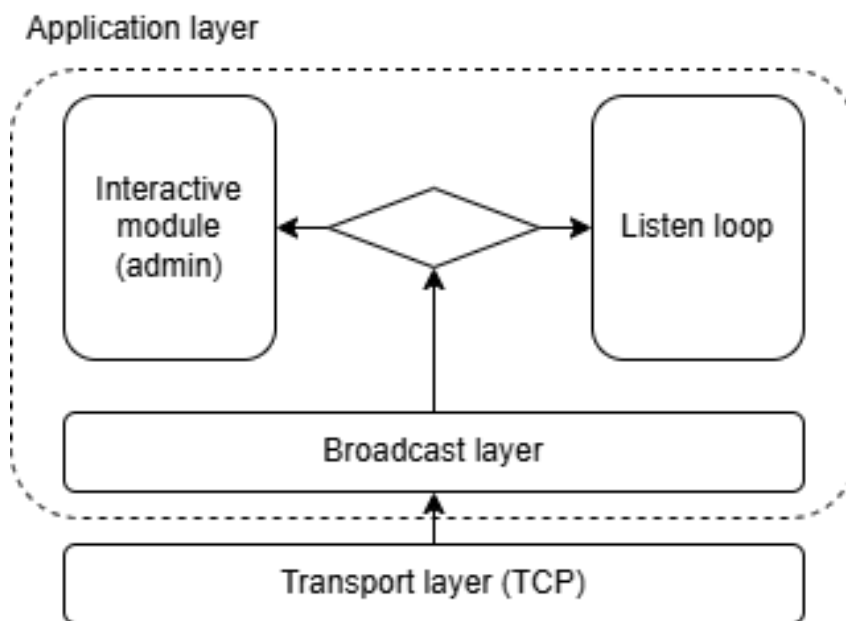
Luồng hoạt động diễn ra như sau: khi một client kết nối tới rat-server, phương thức `AggregationLayer<C>::_handle_connection<K, H>` được thực thi, gọi tới `ClientLayer<C>::accept_connection<K, H>` để tiến hành trao đổi khóa và quay lại `_handle_connection` để cập nhật `AggregationLayer<C>::_clients` sau khi cả 2 bên đã thống nhất các khóa mã hóa.

Mỗi khi client gửi một packet tới, vòng lặp vô hạn trong `ClientLayer<C>::listen_loop` sẽ nhận packet này, gắn thêm địa chỉ của client để tạo thành một cặp (địa chỉ, packet) và gửi lên *aggregation layer* thông qua một multi-producer-single-consumer channel. Tại đây packet được xử lý bởi `AggregationLayer<C>::_handle_packet`.

Các phương thức `AggregationLayer<C>::_poll_packets` và `AggregationLayer<C>::_poll_new_connections` chỉ đơn giản dùng để lặp `_handle_packet` và `_handle_connection` cho đến khi nhận được sự kiện dừng.

Khi nhận được DISCONNECT packet trong `_handle_packet`, rat-server ngay lập tức ngắt kết nối tới client có địa chỉ tương ứng. Tương tự, khi nhận được tín hiệu Ctrl-C, `AggregationLayer<C>::listen_loop` sẽ gửi DISCONNECT packet tới tất cả client trước khi dừng tiến trình.

2.2.2 Thiết kế rat-client



Hình 3: Kiến trúc rat-client

Kiến trúc rat-client gồm *broadcast layer* nhận và chuyển tiếp packet giữa các tầng (trừ khi nhận được DISCONNECT hoặc PING packet từ TCP, khi này broadcast layer sẽ kết thúc tiến trình hoặc phản hồi PONG packet tương ứng).

Nếu chạy trong chế độ admin, ngay sau khi trao đổi khóa thành công, client gửi QUERY packet với `QueryType = Authenticate` cùng với dữ liệu chính là khóa bí mật RSA của rat-server. Khi này server phản hồi RESPONSE packet với `ResponseType ∈ {Success, Error}`. Nếu thành công, client chuyển qua chế độ admin (thực thi *interactive module*). Nếu thất bại, server ngắt kết nối với client ngay lập tức.

Nếu không chạy trong chế độ admin, client thực thi *listen loop* sau khi trao đổi khóa thành công. Ở chế độ này, client nhận các REQUEST packet từ server, thực thi các yêu cầu và trả về một hoặc nhiều RESPONSE packet tương ứng.

2.3 Đồng bộ hóa gửi-nhận

Quan trọng nhất trong hệ thống là struct SSH được cài đặt trong project common. Struct này đảm bảo các quá trình gửi và nhận là atomic, cũng như không có hiện tượng chờ vô hạn.

```

1  /// Thread-safe SSH connection controller.
2  ///
3  /// It is guaranteed that both reading and writing requests will not suffer from starvation.
4  #[derive(Debug)]
5  pub struct SSH<C>
6  where
7      C: Cipher,
8  {
9      _local_addr: SocketAddr,
10     _peer_addr: SocketAddr,
11     _send: Mutex<OwnedWriteHalf>,
12     _send_ctx: Mutex<CipherCtx<C>>,
13     _receive: Mutex<OwnedReadHalf>,
14     _receive_ctx: Mutex<CipherCtx<C>>,
15 }
16
17 impl<C> SSH<C>
18 where
19     C: Cipher,
20 {
21     /// Construct a new SSH connection controller.
22     pub fn new(stream: TcpStream, send_ctx: CipherCtx<C>, receive_ctx: CipherCtx<C>) -> Self;
23     pub fn local_addr(&self) -> SocketAddr;
24     pub fn peer_addr(&self) -> SocketAddr;
25
26     /// Read the version string from peer
27     pub async fn read_version_string(
28         &self,
29         verbose: bool,
30     ) -> Result<String, Box<dyn Error + Send + Sync>>;
31
32     /// Send the version string to peer
33     pub async fn write_version_string(
34         &self,
35         version: &str,
36     ) -> Result<(), Box<dyn Error + Send + Sync>>;
37
38     /// Peek the next byte in the stream without consuming it.
39     ///
40     /// This method is cancel-safe.
41     pub async fn peek(&self) -> io::Result<u8>;
42
43     /// Read a packet from the stream.
44     ///
45     /// WARNING: This is not cancel-safe - canceling the future will leave us in a random position
46     /// in the stream. It is recommended to [peek] first to wait until a packet is available
47     pub async fn read_packet(&self) -> Result<Packet<C>, Box<dyn Error + Send + Sync>>;
48
49     fn _log_send(&self, seq: u32, opcode: Option<u8>);
50
51     pub async fn write_packet(
52         &self,
53         packet: &Packet<C>,
54     ) -> Result<(), Box<dyn Error + Send + Sync>>;
55
56     pub async fn write_payload<P>(
57         &self,
58         payload: &P,
59     ) -> Result<Packet<C>, Box<dyn Error + Send + Sync>>
60     where
61         P: PayloadFormat + Sync;
62
63     pub async fn write_raw_payload(
64         &self,
65         payload: Vec<u8>,

```

```

66 ) -> Result<Packet<C>, Box<dyn Error + Send + Sync>>;
67
68 pub async fn switch_encryption<K, C2, const SERVER: bool>(
69     self,
70     shared_secret: &[u8],
71     exchange_hash: &[u8],
72     session_id: &[u8],
73 ) -> Result<SSH<C2>, Box<dyn Error + Send + Sync>>
74 where
75     K: KexAlgorithm,
76     C2: Cipher;
77 }

```

Code 4: Attribute và method signature của *SSH*

Cụ thể, khi khởi tạo thông qua `SSH<C>::new`, `TcpStream` sẽ được tách thành 2 stream `OwnedWriteHalf` và `OwnedReadHalf`. Mỗi quá trình gửi hoặc nhận cần acquire được 2 khóa: 1 khóa cho stream tương ứng và 1 khóa cho `CipherCtx`. Struct `CipherCtx` có giá trị sequence number thay đổi sau mỗi lần gửi/nhận (giá trị sequence number này được sử dụng để tạo nonce cho `chacha20-poly1305@openssh.com` [4]):

```

1 #[derive(Debug)]
2 pub struct CipherCtx<C>
3 where
4     C: Cipher,
5 {
6     pub seq: u32,
7     pub iv: Vec<u8>,
8     pub enc_key: Vec<u8>,
9     pub int_key: Vec<u8>,
10    _cipher: PhantomData<C>,
11 }

```

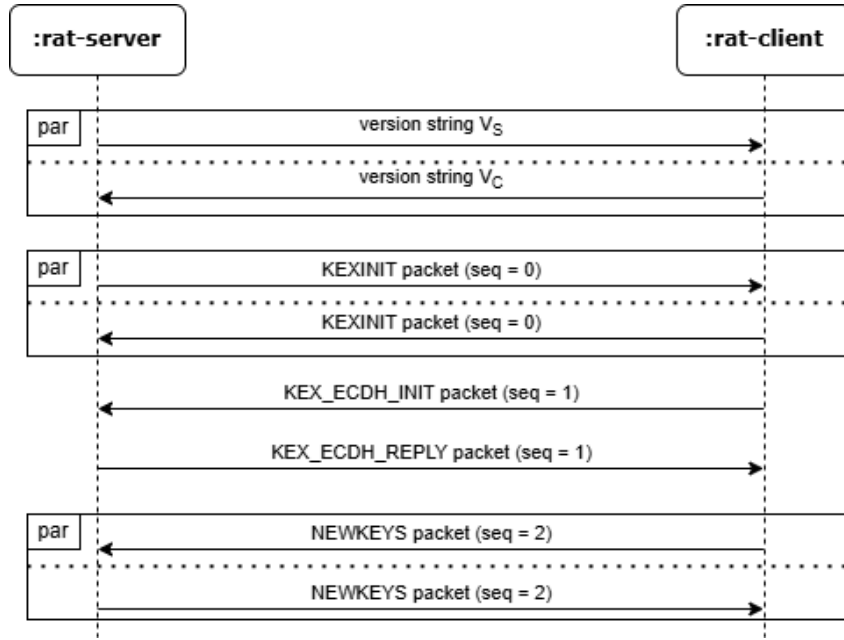
Code 5: Cấu trúc `CipherCtx`

3 Giao thức truyền tin

3.1 Trao đổi shared secret

Thuật toán `chacha20-poly1305@openssh.com` là thuật toán mã hóa đối xứng, do đó trước hết client và server cần thống nhất khóa bí mật dùng chung. Thuật toán trao đổi khóa được sử dụng là `curve25519-sha256` và chữ ký xác thực sử dụng `rsa-sha2-512`. Quy trình trao đổi khóa được tham khảo từ quy trình trao đổi khóa của giao thức SSH trong RFC 4253 [3] (hình 4).

Để đảm bảo thuật toán trao đổi khóa đúng với quy trình được trình bày trong RFC 4253 [3], `rat-client` được kiểm tra bằng cách thực hiện trao đổi khóa với OpenSSH v10. Vì vậy, một số packet trong quy trình trao đổi khóa không thật sự cần thiết giữa 2 thành phần `rat-client` và `rat-server` mà được sử dụng để kiểm thử thuật toán giữa `rat-client` và OpenSSH v10 server.



Hình 4: Quy trình trao đổi khóa

- Sau khi client và server thiết lập thành công kết nối TCP, cả 2 bên gửi version string (mã hóa UTF-8) cùng 2 byte CRLF ("\r\n"). Ở đây cả 2 version string đều là $V_C = V_S = \text{"SSH-2.0-remote-access-tool linux"}$.
- Bắt đầu từ tin nhắn tiếp theo, mỗi bên gửi packet theo định dạng quy ước trong RFC 4253 [3] và tính sequence number cho mỗi bên bắt đầu từ 0 (giao thức TCP đảm bảo các packet đến với bên nhận đúng như thứ tự được gửi đi). Định dạng packet được trình bày trong phần 3.2. Ban đầu các packet chưa được mã hóa.
- Cả 2 bên gửi KEXINIT packet trao đổi về thuật toán 2 bên hỗ trợ.
- Client tạo cặp khóa (k_{uc}, k_{rc}) cho giao thức trao đổi khóa ECDH (Elliptic-curve Diffie–Hellman, ở đây là curve25519-sha256) và gửi KEX_ECDH_INIT packet chứa 32 byte khóa công khai k_{uc} . Server cũng tạo cặp khóa (k_{us}, k_{rs}) và tính được:
 - Shared secret $K = k_{rs} * P(k_{uc})$ (tọa độ x của phép nhân điểm P có tọa độ x = k_{uc} với hệ số k_{rs} trên đường cong Montgomery).
 - Exchange hash $H = \text{SHA256}(V_C || V_S || I_C || I_S || K_S || k_{uc} || k_{us} || K)$. Trong đó $||$ là phép nối string; I_C , I_S lần lượt là payload của 2 packet KEXINIT gửi từ client và server và K_S là phần payload chứa khóa công khai RSA sẽ trình bày ở dưới đây.
 - Chữ ký $S = \text{Sign}(k_{rs}, H)$.
- Sau khi tính được K và H như trên, server gửi KEX_ECDH_REPLY packet có payload bao gồm 3 phần:
 - K_S : chứa string `host_key_algorithm = "ssh-rsa"` và khóa công khai RSA.
 - Q_S : chứa k_{us} .
 - SIG_S : chứa string `signature_algorithm = "rsa-sha2-512"` và chữ ký S .
- Client nhận được KEX_ECDH_REPLY packet, tính được K , H và kiểm tra S . Nếu S không hợp lệ hoặc client thấy khóa công khai RSA không phải khóa của server muốn kết nối thì ngắt kết nối ngay lập tức.

Như vậy, nếu thuật toán trao đổi khóa thành công, cả 2 bên đã thống nhất được 32 byte shared secret K và exchange hash H . Để sử dụng các giá trị này cho thuật toán mã hóa chacha20-poly1305@openssh.com cần tính toán các giá trị sau:

- Khóa mã hóa từ client đến server:

- $k_{payload} = \text{SHA256}(K||H||'C'||H)$ (trong đó 'C' là ký tự ASCII 67).
- k_{length} và phần còn lại của packet (chưa có) = $\text{SHA256}(K||H||k_{payload})$
- Khóa mã hóa từ server đến client:
 - $k_{payload} = \text{SHA256}(K||H||'D'||H)$ (trong đó 'D' là ký tự ASCII 68).
 - $k_{length} = \text{SHA256}(K||H||k_{payload})$

3.2 Định dạng và mã hóa packet

Mỗi packet có định dạng theo quy ước như RFC 4253 [3], gồm các phần:

1. packet_length: uint32 là độ dài packet.
2. padding_length: byte là độ dài padding.
3. payload: byte[packet_length - padding_length - 1] là thông tin có nghĩa chứa trong packet.
4. random_padding: byte[padding_length] là padding để packet có độ dài theo quy ước.
5. mac: byte[16] là mã xác thực thông điệp MAC (mã xác thực của poly1305 dài 16 byte).

Phần payload chứa thông tin có nghĩa của packet tương ứng. Byte đầu tiên trong payload là opcode của packet. Hệ thống RAT sử dụng các opcode sau:

- 1 DISCONNECT
- 2 IGNORE
- 20 KEXINIT
- 21 NEWKEYS
- 30 KEX_ECDH_INIT
- 31 KEX_ECDH_REPLY
- 192 RESPONSE
- 193 REQUEST
- 194 CANCEL
- 195 QUERY
- 196 PING
- 197 PONG

Thuật toán chacha20-poly1305@openssh.com [4] sử dụng chacha20 để mã hóa và poly1305 để xác thực. Khi mã hóa, 4 byte nonce được đặt bằng sequence number của bên gửi, 4 byte packet_length được mã hóa bằng k_{length} (gọi bản mã là C_1) và phần còn lại của packet (chưa có mac) được mã hóa bằng $k_{payload}$ (gọi bản mã là C_2).

Khóa cho poly1305 được tạo ra bằng cách mã hóa chacha20 32 byte 0 với khóa $k_{payload}$, từ đó tính được 16 byte mac xác thực thông điệp ($C_1||C_2$).

4 Sản phẩm

4.1 Đóng gói và triển khai

Hệ thống sau khi cài đặt được triển khai trên Docker. Các container bao gồm: 2 container ubuntu:24.04 và centos:7 có vai trò client, 1 container ubuntu:24.04 có vai trò server. Image build bao gồm 2 stage: stage đầu tiên tạo package cho hệ điều hành tương ứng (.deb hoặc .rpm), stage tiếp theo copy package tạo từ stage trước và install bằng apt hoặc yum.

```

1 # Reference: https://docs.docker.com/reference/dockerfile/
2 FROM rust:1.87 AS builder
3
4 RUN cargo install cargo-deb
5
6 COPY . /app
7 WORKDIR /app
8
```

```

9 RUN cargo deb -p rat-client -o rat-client.deb
10
11 FROM ubuntu:24.04
12
13 COPY --from=builder /app/rat-client.deb /app/rat-client.deb
14 RUN apt-get update && apt-get install -y openssh-client /app/rat-client.deb

```

Code 6: Build image cho RAT client trên ubuntu:24.04

```

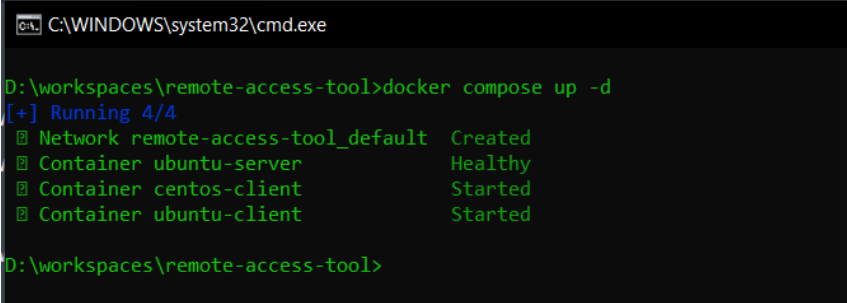
1 # Reference: https://docs.docker.com/reference/dockerfile/
2 FROM centos:7 AS builder
3
4 # https://serverfault.com/a/1161847
5 RUN sed -i s/mirror.centos.org/vault.centos.org/g /etc/yum.repos.d/CentOS-*.repo
6 RUN sed -i s/^.*baseurl=http/baseurl=http/g /etc/yum.repos.d/CentOS-*.repo
7 RUN sed -i s/^mirrorlist=http/#mirrorlist=http/g /etc/yum.repos.d/CentOS-*.repo
8
9 RUN yum update -y && yum install -y curl gcc
10 RUN curl --proto 'https' --tlsv1.2 https://sh.rustup.rs -sSf | sh -s -- --default-toolchain=1.87 -y
11 ENV PATH="/root/.cargo/bin:${PATH}"
12
13 RUN cargo install cargo-generate-rpm
14
15 COPY . /app
16 WORKDIR /app
17
18 RUN cargo build -p rat-client --release
19 RUN cargo generate-rpm --payload-compress none -p client -o rat-client.rpm
20
21 FROM centos:7
22
23 # https://serverfault.com/a/1161847
24 RUN sed -i s/mirror.centos.org/vault.centos.org/g /etc/yum.repos.d/CentOS-*.repo && \
25     sed -i s/^.*baseurl=http/baseurl=http/g /etc/yum.repos.d/CentOS-*.repo && \
26     sed -i s/^mirrorlist=http/#mirrorlist=http/g /etc/yum.repos.d/CentOS-*.repo
27
28 COPY --from=builder /app/rat-client.rpm /app/rat-client.rpm
29 RUN yum install -y /app/rat-client.rpm

```

Code 7: Build image cho RAT client trên centos:7

Các container sau đó được khởi chạy bằng "docker compose up -d". Container vai trò server có file khóa bí mật nằm ở /host.

4.2 Kết quả thực thi



```

C:\WINDOWS\system32\cmd.exe

D:\workspaces\remote-access-tool>docker compose up -d
[+] Running 4/4
  Network remote-access-tool_default Created
  Container ubuntu-server                 Healthy
  Container centos-client                 Started
  Container ubuntu-client                 Started
D:\workspaces\remote-access-tool>

```

Hình 5: Khởi tạo container

```

root@ubuntu-server: /

0:\workspaces\remote-access-tool>docker exec -it ubuntu-server bash
root@ubuntu-server:/# rat-client localhost:22 --admin /host
[*] SSH-2.0-remote-access-tool linux
[*] Host public key is C696E31D141F7D5E53BF1A78070C51501A2A7541ECD0EE76EE0CD0349D8FA7B3
[*] Authentication successful

server>help
Remote Access Tool (RAT) client component

Usage: server> [COMMAND]

Commands:
client      Manage connected clients
cd          Change the working directory on the client side
ls          List information about a directory on the client side
pwd         Print working directory on the client side
rm          Remove a directory or file on the client side
download    Download a file from the client
ps          View running processes on the client side
kill        Kill a process on the client side
target      Set the default target address for commands
exit        Shut down the server
help        Print this message or the help of the given subcommand(s)

server>_

```

Hình 6: Các lệnh điều khiển client

```

root@ubuntu-server: /

server>help client
Manage connected clients

Usage: client [COMMAND]

Commands:
ls          List connected clients
disconnect  Disconnect a client
help        Print this message or the help of the given subcommand(s)

server>client ls

```

Address	Version string	Admin
127.0.0.1:49014	SSH-2.0-remote-access-tool linux	true
172.18.0.4:38262	SSH-2.0-remote-access-tool linux	false
172.18.0.3:41768	SSH-2.0-remote-access-tool linux	false

```

server>

```

Hình 7: Hiển thị các client đang kết nối tới server

```

root@ubuntu-server: /

server>ls -a 172.18.0.4:38262 /app
File name      | File type | Created at      | Modified at      | Size
-----+-----+-----+-----+-----
rat-client.rpm | file      | 2025-06-11 05:33:07 | 2025-06-11 05:33:04 | 5.73 MiB

server>ls -a 172.18.0.3:41768 /app
File name      | File type | Created at      | Modified at      | Size
-----+-----+-----+-----+-----
rat-client.deb | file      | 2025-06-10 14:29:34 | 2025-06-10 14:28:48 | 1.33 MiB
test.txt       | file      | 2025-06-11 05:50:47 | 2025-06-11 05:50:47 | 5.00 GiB

server>

```

Hình 8: Hiển thị nội dung directory /app của client

```

root@ubuntu-server: /

server>target 172.18.0.3:41768
Setting primary target to 172.18.0.3:41768

server:172.18.0.3:41768>download /app/test.txt /test.txt
Downloading: 363.28 MiB/5.00 GiB (693.92 KiB/s)    ^CRequest cancelled

server:172.18.0.3:41768>_

```

Hình 9: Tải file /app/test.txt từ client

```

root@ubuntu-server: /

server:172.18.0.3:41768>ps
Total number of processes: 12

```

PID	Name	CPU	Mem	Command	Run time
11	tokio-runtime-w	0.00%	5.75 MiB	rat-client ubuntu-server:22	37m 27s
7	tokio-runtime-w	0.00%	5.75 MiB	rat-client ubuntu-server:22	37m 27s
38	sleep	0.00%	1.25 MiB	sleep infinity	4s
13	tokio-runtime-w	0.00%	5.75 MiB	rat-client ubuntu-server:22	37m 27s
1	rat-client	0.00%	5.75 MiB	rat-client ubuntu-server:22	37m 27s
10	tokio-runtime-w	0.00%	5.75 MiB	rat-client ubuntu-server:22	37m 27s
8	tokio-runtime-w	0.00%	5.75 MiB	rat-client ubuntu-server:22	37m 27s
14	tokio-runtime-w	0.00%	5.75 MiB	rat-client ubuntu-server:22	37m 27s
9	tokio-runtime-w	0.00%	5.75 MiB	rat-client ubuntu-server:22	37m 27s
39	tokio-runtime-w	0.00%	5.75 MiB	rat-client ubuntu-server:22	0s
16	sh	0.00%	1.62 MiB	/bin/sh	24m 46s
12	tokio-runtime-w	0.00%	5.75 MiB	rat-client ubuntu-server:22	37m 27s

```

server:172.18.0.3:41768>kill 38

server:172.18.0.3:41768>ps
Total number of processes: 11

```

PID	Name	CPU	Mem	Command	Run time
13	tokio-runtime-w	0.00%	5.75 MiB	rat-client ubuntu-server:22	38m 11s
11	tokio-runtime-w	0.00%	5.75 MiB	rat-client ubuntu-server:22	38m 11s
8	tokio-runtime-w	0.00%	5.75 MiB	rat-client ubuntu-server:22	38m 11s
10	tokio-runtime-w	0.00%	5.75 MiB	rat-client ubuntu-server:22	38m 11s
14	tokio-runtime-w	0.00%	5.75 MiB	rat-client ubuntu-server:22	38m 11s
40	tokio-runtime-w	0.00%	5.75 MiB	rat-client ubuntu-server:22	0s
1	rat-client	0.00%	5.75 MiB	rat-client ubuntu-server:22	38m 11s
7	tokio-runtime-w	0.00%	5.75 MiB	rat-client ubuntu-server:22	38m 11s
9	tokio-runtime-w	0.00%	5.75 MiB	rat-client ubuntu-server:22	38m 11s
16	sh	0.00%	1.62 MiB	/bin/sh	25m 30s
12	tokio-runtime-w	0.00%	5.75 MiB	rat-client ubuntu-server:22	38m 11s

```

server:172.18.0.3:41768>

```

Hình 10: Liệt kê và dừng tiến trình

Tài liệu

- [1] <https://github.com/rust-lang/rust> Empowering everyone to build reliable and efficient software.
- [2] <https://github.com/tokio-rs/tokio> A runtime for writing reliable asynchronous applications with Rust. Provides I/O, networking, scheduling, timers, ...
- [3] <https://datatracker.ietf.org/doc/html/rfc4253> The Secure Shell (SSH) is a protocol for secure remote login and other secure network services over an insecure network.
- [4] <https://datatracker.ietf.org/doc/draft-ietf-sshm-chacha20-poly1305/> This document describes the Secure Shell (SSH) chacha20-poly1305 authenticated encryption cipher.