PR #45890 完整报告

vllm-project/vllm

[Rust Frontend] Add static HTTPS and mTLS support for HTTP and gRPC

合并时间：2026-06-30 09:46

原文链接：http://prhub.com.cn/vllm-project/vllm/pull/45890

Rust 前端 HTTPS 与 mTLS 支持

执行摘要

此 PR 为 Rust 前端添加了静态 HTTPS 与 mTLS 终结支持，涵盖 HTTP 和 gRPC 服务器。使用 OpenSSL 实现，通过 --ssl-* 系列参数配置，并引入连接超时机制与 gRPC 绑定地址的安全加固。这是一次关键的 Infrastructure 改进，使 Rust 前端具备了生产环境的加密能力。

功能与动机

Rust 前端需要与 Python 前端同等的 SSL 支持（跟踪 #44280）。之前 Rust 前端仅支持明文 HTTP/gRPC，无法处理加密流量。此 PR 新增了 HTTPS 和 mTLS 支持，并修复了 gRPC 服务默认暴露在所有网络接口的安全问题。

实现拆解

1. TLS 配置模块（tls.rs）：基于 OpenSSL 的 SslAcceptorBuilder，支持证书链、私钥（合并 PEM）、mTLS 客户端验证、自定义密码套件。默认使用 mozilla_intermediate_v5（TLS 1.2+ AEAD-only）。
2. 统一 Listener 层（listener.rs）：引入 ListenerIo 枚举类型，集成 tls-listener crate 实现透明 TLS 握手；新增 enable_tcp_nodelay 和 grpc_bind_host（未明确指定时默认 loopback）。
3. 服务入口整合（lib.rs）：在 serve_with_router_extension 中预先构建 TLS 配置，失败则快速退出；gRPC 服务复用 HTTP 的 TLS 配置，通过 ALPN h2 协商 HTTP/2。
4. gRPC 模块扩展（grpc/mod.rs）：新增 GrpcTlsStream 包装器，实现 tower::Service trait，使得 gRPC 服务能直接处理 TLS 连接。
5. CLI与配置验证（cli.rs和config.rs）：添加--ssl-certfile、--ssl-keyfile、--ssl-ca-certs、--ssl-cert-reqs、--ssl-ciphers 参数；TlsConfig 结构体包含 validate() 方法，在启动时严格校验（如 keyfile 必须配合 certfile、mTLS 必须提供 CA）。
6. 测试配套：新增 tls_tests.rs 端到端测试（自签名 CA 和证书链验证）；grpc/tests.rs 增加 TLS 测试；cli/tests.rs 增加 CLI 解析和验证测试。

rust/src/server/src/tls.rs

核心 TLS 配置模块，负责构建 OpenSSL SslAcceptor，支持证书链、私钥、mTLS 和密码套件。

```
//! OpenSSL server-config construction for TLS termination.
//! Builds an SslContext from uvicorn-style ssl_* arguments.

use std::path::Path;
use std::time::Duration;
use anyhow::{Context as _, Result};
use openssl::ssl::{
    AlpnError, SslAcceptor, SslAcceptorBuilder, SslContext, SslContextBuilder,
    SslFiletype, SslMethod, SslOptions, SslVerifyMode, select_next_proto,
};
use crate::config::TlsConfig;

/// Time a client has to complete the TLS handshake before connection is dropped.
pub(crate) const TLS_HANDSHAKE_TIMEOUT: Duration = Duration::from_secs(60);

/// ALPN wire bytes for HTTP/2 (length-prefixed).
const ALPN_H2: &[u8] = b"\x02h2";

/// Build the shared OpenSSL acceptor from validated TlsConfig.
/// Uses Mozilla intermediate v5 baseline (forward-secret AEAD, TLS 1.2+).
fn build_server_builder(tls: &TlsConfig) -> Result<SslAcceptorBuilder> {
    let cert_file = tls.cert_file.as_deref()
        .context("--ssl-certfile is required to enable TLS")?;

    let mut builder = SslAcceptor::mozilla_intermediate_v5(SslMethod::tls_server())
        .context("failed to initialize TLS")?;
    builder.set_options(SslOptions::CIPHER_SERVER_PREFERENCE);

    // Load the full certificate chain (leaf + intermediates)
    ensure_exists(cert_file, "--ssl-certfile")?;
    builder.set_certificate_chain_file(cert_file)
        .with_context(|| format!("failed to parse certificate chain in --ssl-certfile {cert_file:?}"))?;

    // When key_file is unset, read key from the certificate file (combined PEM)
    let key_file = tls.key_file.as_deref().unwrap_or(cert_file);
    ensure_exists(key_file, "private key file")?;
    builder.set_private_key_file(key_file, SslFiletype::PEM)
        .with_context(|| format!("failed to parse private key in {key_file:?}"))?;
    builder.check_private_key()
        .context("the certificate and private key do not match")?;

    configure_client_auth(&mut builder, tls)?;

    if let Some(ciphers) = tls.ciphers.as_deref().filter(|c| !c.is_empty()) {
        builder.set_cipher_list(ciphers)
            .with_context(|| format!("invalid --ssl-ciphers {ciphers:?}"))?;
    }

    Ok(builder)
```

```
}

/// Configure client certificate verification for mutual TLS.
fn configure_client_auth(builder: &mut SslAcceptorBuilder, tls: &TlsConfig) -> Result<()> {
    if tls.cert_reqs > 0 {
        // ca_certs is required when cert_reqs > 0
        let ca_file = tls.ca_certs.as_deref()
            .context("--ssl-ca-certs is required when --ssl-cert-reqs > 0")?;
        ensure_exists(ca_file, "--ssl-ca-certs")?;
        builder.set_ca_file(ca_file)
            .context("failed to load --ssl-ca-certs")?;
        let verify_mode = if tls.cert_reqs == 1 {
            SslVerifyMode::PEER // optional client cert
        } else {
            SslVerifyMode::PEER | SslVerifyMode::FAIL_IF_NO_PEER_CERT // required client cert
        };
        builder.set_verify(verify_mode);
    }
    Ok(())
}

/// Build the HTTP SslContext (without ALPN, matching uvicorn).
pub(crate) fn build_server_config(tls: &TlsConfig) -> Result<SslContext> {
    let builder = build_server_builder(tls)?;
    Ok(builder.build())
}
```

rust/src/server/src/lib.rs

服务主入口，集成 TLS listener 构建与 gRPC 共享配置，添加超时常量。

```
/// Choose the gRPC listener host. It follows the HTTP TCP host when there is
/// one; otherwise (unix socket or inherited fd) it defaults to IPv4 loopback
/// rather than all interfaces, so the side-car is never accidentally
/// network-exposed.
fn grpc_bind_host(listener_mode: &HttpListenerMode) -> &str {
    match listener_mode {
        HttpListenerMode::BindTcp { host, .. } => host.as_str(),
        HttpListenerMode::BindUnix { .. } | HttpListenerMode::InheritedFd { .. } => "127.0.0.1",
    }
}

// In serve_with_router_extension:
// Build TLS config upfront, so bad cert/key fail fast before engine handshake.
let tls_config = config.tls.as_ref().map(tls::build_server_config)
    .transpose().context("failed to build TLS config")?;
let tls_grpc_config = config.tls.as_ref().map(tls::build_grpc_server_config)
    .transpose().context("failed to build gRPC TLS config")?;

// Bind the HTTP listener (possibly with TLS wrapping)
```

```
let listener = if let Some(ref tls_cfg) = tls_config {
    let acceptor = tls_cfg.clone().into_acceptor()?; // simplified
    Listener::bind_tls(&config.listener_mode, acceptor).await?
} else {
    Listener::bind(&config.listener_mode).await?
};
```

rust/src/server/src/listener.rs

引入统一的 ListenerIo 类型，集成 tls-listener crate 实现透明 TLS 握手。

```
/// Runtime listener I/O type which is either a TCP stream or a Unix-domain stream.
#[derive(Debug)]
#[enum_derive(tokio1::AsyncRead, tokio1::AsyncWrite)]
pub enum ListenerIo {
    Tcp(TcpStream),
    Unix(UnixStream),
}

/// Runtime listener address type.
#[derive(Debug)]
pub enum ListenerAddr {
    Tcp(SocketAddr),
    Unix(tokio::net::unix::SocketAddr),
}

impl Connected for ListenerIo {
    type ConnectInfo = TcpConnectInfo;

    fn connect_info(&self) -> TcpConnectInfo {
        match self {
            Self::Tcp(stream) => stream.connect_info(),
            Self::Unix(_) => TcpConnectInfo {
                local_addr: None,
                remote_addr: None,
            },
        }
    }
}

/// Attempt to set TCP_NODELAY on the accepted TCP stream.
fn enable_tcp_nodelay(stream: TcpStream) -> TcpStream {
    if let Err(err) = stream.set_nodelay(true) {
        tracing::trace!(error = %err, "failed to enable TCP_NODELAY on accepted TCP
        connection");
    }
    stream
}
```

评论区精华

> BugenZhao: 我们之前刻意避免 rustls，使用 native-tls + vendored。为什么这里引入 rustls？ Tahsintunan: 因为 native-tls 不支持服务器端 client-cert 验证（mTLS），rustls 支持。但考虑到依赖复用，决定改用 OpenSSL。
>
> BugenZhao: 建议使用 tls-listener crate 简化 TLS 实现，并考虑 native-tls 后端。 Tahsintunan: native-tls 不支持 mTLS，故保留 OpenSSL。 结论: 使用 tls-listener + OpenSSL。
>
> njhill: gRPC 应该复用 HTTP 的 TLS 配置，并添加 TLS 握手超时。 Tahsintunan: 已实现，gRPC 与 HTTP 共享 TLS 配置，添加 60s 超时。

风险与影响

- 默认地址变更可能破坏已有依赖：gRPC 默认从 0.0.0.0 改为 127.0.0.1，需要外部访问的用户必须显式设置 --host。
- 密码套件收紧：默认只支持 AEAD 套件（TLS 1.2+），老旧客户端可能失败，可通过 --ssl-ciphers 调整。
- keep-alive 超时默认 5s：可能断开长时间空闲的 streaming 连接，可设置 VLLM_HTTP_TIMEOUT_KEEP_ALIVE=0 禁用。

关联脉络

此 PR 是 Rust 前端生产就绪的重要一步（跟踪 #44280）。后续计划支持证书热重载（PR #47362），以匹配 Python 前端的 --enable-ssl-refresh。同时，Rust 前端的连接管理正在逐步完善，此前已有关联 PR 关注连接超时与性能（如 #45723 等），但与本 PR 无直接重叠。