```
#!/bin/bash
# YuKKi OS 3.2
# new prompt config!
set -euo pipefail
R=$(pwd)/yukki_os_final;rm -rf "$R";mkdir -p
"$R"/{common,vendor,bin,logs,pki,docs,yukki_c2_suite/{bootstrap_server
,peer_client/{received_files,client_data}}}
MV="7.10";MS="4f85a4b76e159235e839e1553c4e0c33a8258384d510b65103c1d683
d7bd495a";CV="1.7.17";CS="7508e75e9e03c2f16a04870c3ce380e2272e1e07b89a
de1e909673b0f55c3c08"
LV="master";LHS="dc0c466e03c4f035f8c3c706e4695a70513e8d1b11794d2105c3c
0f6ab483a9a";LCS="52b96377e3c3c1b05a76c1256b82541315e06f236e788c6f5053
1853f2c5d64a"
wget -O "$R/vendor/mongoose.h"
"https://raw.githubusercontent.com/cesanta/mongoose/$MV/mongoose.h";wg
et -O "$R/vendor/mongoose.c"
"https://raw.githubusercontent.com/cesanta/mongoose/$MV/mongoose.c"
wget -O "$R/vendor/cJSON.h"
"https://raw.githubusercontent.com/DaveGamble/cJSON/v$CV/cJSON.h";wget
-O "$R/vendor/cJSON.c"
"https://raw.githubusercontent.com/DaveGamble/cJSON/v$CV/cJSON.c"
wget -O "$R/vendor/linenoise.h"
"https://raw.githubusercontent.com/antirez/linenoise/$LV/linenoise.h";
wget -O "$R/vendor/linenoise.c"
"https://raw.githubusercontent.com/antirez/linenoise/$LV/linenoise.c"
echo "$MS $R/vendor/mongoose.h
$MS $R/vendor/mongoose.c
$CS $R/vendor/cJSON.h
$CS $R/vendor/cJSON.c
$LHS $R/vendor/linenoise.h
$LCS $R/vendor/linenoise.c"|sha256sum -c -
a(){ cat > "$R/docs/CRTC_Compliance_Framework.md" <<'EOF'
# Yukki OS 3.2 - CRTC Compliance Framework (U-C-2019-288)
This document outlines the compliance mechanisms within Yukki OS
intended to align with the principles of CRTC's Unsolicited
Telecommunications Rules (UTR), specifically regarding P2P network
interactions.
## 1. User Consent (PIPEDA Principle)
- **Mechanism**: Explicit "Opt-In" Consent.
- **Implementation**: On first launch, the `peer_client` binary will
not connect to any network (bootstrap or P2P) until the user
explicitly provides consent.
- **Audit**: A consent log is written to `$ROOT/logs/consent_log.md`
containing the user's UUID and a timestamp. This log is for local
audit only and is not transmitted.
## 2. Internal Block List (Voter Contact Registry Analogue)
- **Mechanism**: Per-user internal block list.
- **Purpose**: Provides users with a direct mechanism to block all
```

incoming P2P communications from a specific peer UUID, analogous to an internal "Do Not Call" list.
- **Implementation**:
  - `block <uuid>`: Adds a peer's UUID to the local `client_data/blocklist.conf`.
  - `unblock <uuid>`: Removes a peer from the list.
  - `profile`: Displays the current user's block list.
- **Enforcement**:
  1. **Incoming P2P Connections**: The P2P listener thread will perform an mTLS handshake, extract the peer's UUID from their certificate's Common Name (CN), and compare it against the block list. If a match is found, the connection is immediately terminated.
  2. **Outgoing Commands**: Commands like `msg`, `get`, `send`, `rjob` will fail if the target UUID is on the block list.
3. Identification (mTLS Certificate)
- **Mechanism**: Cryptographic Identity.
- **Implementation**: All P2P communication is secured via mTLS. Each peer must present a valid client certificate signed by the network's Certificate Authority (CA). The peer's unique UUID is embedded in the certificate's Common Name (CN).
- **Enforcement**: Any peer failing to provide a valid, signed certificate is rejected at the transport layer.
4. Record Keeping
- **Mechanism**: Local-only logging.
- **Implementation**: The `common/logger` module provides functions for logging critical events (e.g., connections, errors, file transfers, job submissions).
- **Compliance**: All logs are stored *locally* on the client machine in `$ROOT/logs/`. No logs are transmitted to the C2 server or other peers, respecting data minimization principles.
EOF
cat > "$R/docs/PIPEDA_Data_Handling.md" <<'EOF'
Yukki OS 3.2 - PIPEDA Data Handling Policy
This document details how Yukki OS handles "Personal Information" (PI) in the context of the Personal Information Protection and Electronic Documents Act (PIPEDA).
Definition of Personal Information (PI)
In the context of Yukki OS, PI is limited to:
1. **User UUID**: A unique identifier for a network participant.
2. **User IP Address & Port**: Network location information.
3. **Data in Transit**: Any files, messages, or job commands a user chooses to send.
PIPEDA Principles & Yukki Implementation
1. **Accountability**: The user is accountable for the data they transmit. The `yukki_configurator.sh` script establishes a "profile" that is locally managed.
2. **Identifying Purposes**: The purpose of data collection (UUID, IP) is solely for establishing a P2P network. This is stated to the

```
user.
3.  **Consent**:
  - **Explicit Consent**: The client requires explicit user consent
before *any* network activity (see `CRTC_Compliance_Framework.md`).
  - **Withdrawal of Consent**: A user can withdraw consent by deleting
their profile and logs.
4.  **Limiting Collection**:
  - **Bootstrap Server**: Collects *only* the UUID and IP/Port of
connected peers. It does not see, log, or intermediate file transfers,
messages, or remote jobs.
  - **Peer Client**: Collects only the peer list from the bootstrap
server.
5.  **Limiting Use, Disclosure, and Retention**:
  - **Use**: UUID/IP is used *only* to facilitate mTLS P2P
connections.
  - **Disclosure**: The *only* disclosure is the C2 server providing
the peer list (UUID/IP) to other authenticated peers.
  - **Retention**:
      - **Bootstrap Server**: Peer information is volatile and kept
in-memory only. When a peer disconnects, their information is removed.
      - **Peer Client**: The peer list is refreshed periodically. Data
in `received_files/` and `logs/` is retained until the user manually
deletes it.
6.  **Accuracy**: The C2 server relies on peers to provide accurate
IP/Port information.
7.  **Safeguards**:
  - **Encryption in Transit**: All C2-client and peer-to-peer
communication is encrypted using mTLS (OpenSSL).
  - **Authentication**: All parties (C2, peers) are authenticated
using x509 certificates.
8.  **Openness**: This document and the `CRTC_Compliance_Framework.md`
serve as open documentation of the system's data handling policies.
9.  **Individual Access**: A user can see all PI held about them by
using the `profile` command, which displays their UUID, IP (as known
to C2), and block list.
10. **Challenging Compliance**: As a decentralized tool, compliance
challenges are managed by the user's ability to `block` peers or
disconnect from the network.
EOF
};b(){ cat > "$R/common/logger.h" <<'EOF'
#pragma once
#include <stdio.h>
#include <time.h>
#include <string.h>
#define LOG_FILE "logs/yukki_client.log"
#define C2_LOG_FILE "logs/yukki_c2.log"
#define LOG_LEVEL_DEBUG 0
#define LOG_LEVEL_INFO  1
```

```
#define LOG_LEVEL_WARN  2
#define LOG_LEVEL_ERROR 3
#define CURRENT_LOG_LEVEL LOG_LEVEL_INFO
#define __FILENAME__ (strrchr(__FILE__, '/') ? strrchr(__FILE__, '/')
+ 1 : __FILE__)
#define LOG_GENERIC(file, level_str, level, fmt, ...) \
  do { \
      if (level >= CURRENT_LOG_LEVEL) { \
          time_t now = time(0); \
          char buf[32]; \
          strftime(buf, sizeof(buf), "%Y-%m-%d %H:%M:%S",
localtime(&now)); \
          FILE* f = fopen(file, "a"); \
          if (f) { \
              fprintf(f, "[%s] [%s] [%s:%d] " fmt "\n", buf,
level_str, __FILENAME__, __LINE__, ##__VA_ARGS__); \
              fclose(f); \
          } \
          fprintf(stdout, "[%s] [%s] " fmt "\n", buf, level_str,
##__VA_ARGS__); \
      } \
  } while (0)
#define log_debug(fmt, ...) LOG_GENERIC(LOG_FILE, "DEBUG",
LOG_LEVEL_DEBUG, fmt, ##__VA_ARGS__)
#define log_info(fmt, ...)  LOG_GENERIC(LOG_FILE, "INFO",
LOG_LEVEL_INFO,  fmt, ##__VA_ARGS__)
#define log_warn(fmt, ...)  LOG_GENERIC(LOG_FILE, "WARN",
LOG_LEVEL_WARN,  fmt, ##__VA_ARGS__)
#define log_error(fmt, ...) LOG_GENERIC(LOG_FILE, "ERROR",
LOG_LEVEL_ERROR, fmt, ##__VA_ARGS__)
#define c2_log_debug(fmt, ...) LOG_GENERIC(C2_LOG_FILE, "DEBUG",
LOG_LEVEL_DEBUG, fmt, ##__VA_ARGS__)
#define c2_log_info(fmt, ...)  LOG_GENERIC(C2_LOG_FILE, "INFO",
LOG_LEVEL_INFO,  fmt, ##__VA_ARGS__)
#define c2_log_warn(fmt, ...)  LOG_GENERIC(C2_LOG_FILE, "WARN",
LOG_LEVEL_WARN,  fmt, ##__VA_ARGS__)
#define c2_log_error(fmt, ...) LOG_GENERIC(C2_LOG_FILE, "ERROR",
LOG_LEVEL_ERROR, fmt, ##__VA_ARGS__)
EOF
cat > "$R/common/file_utils.h" <<'EOF'
#pragma once
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/stat.h>
static inline int read_config_line(FILE* f, char* buffer, size_t
max_len) {
  if (fgets(buffer, max_len, f) == NULL) {
```

```
      return 0;
  }
  buffer[strcspn(buffer, "\r\n")] = 0;
  return 1;
}
static inline int get_config_value(const char* file_path, const char*
key, char* value, size_t max_len) {
  FILE* f = fopen(file_path, "r");
  if (f == NULL) {
      return 0;
  }
  char line[512];
  char key_buffer[256];
  char value_buffer[256];
  int found = 0;
  while (read_config_line(f, line, sizeof(line))) {
      if (sscanf(line, "%255[^=]=%255[^\n]", key_buffer, value_buffer)
== 2) {
          if (strcmp(key_buffer, key) == 0) {
              strncpy(value, value_buffer, max_len - 1);
              value[max_len - 1] = '\0';
              found = 1;
              break;
          }
      }
  }
  fclose(f);
  return found;
}
static inline void ensure_directory_exists(const char* path) {
  struct stat st = {0};
  if (stat(path, &st) == -1) {
      mkdir(path, 0700);
  }
}
EOF
cat > "$R/common/pki_utils.h" <<'EOF'
#pragma once
#include <openssl/ssl.h>
#include <openssl/err.h>
#include <string.h>
static inline SSL_CTX* create_ssl_context(const char* ca_cert, const
char* cert_file, const char* key_file, int is_server) {
  SSL_CTX* ctx;
  SSL_load_error_strings();
  OpenSSL_add_ssl_algorithms();
  const SSL_METHOD* method = is_server ?
  TLS_server_method() : TLS_client_method();
```

```
  ctx = SSL_CTX_new(method);
  if (!ctx) {
      perror("Unable to create SSL context");
      ERR_print_errors_fp(stderr);
      exit(EXIT_FAILURE);
  }
  if (SSL_CTX_load_verify_locations(ctx, ca_cert, NULL) <= 0) {
      ERR_print_errors_fp(stderr);
      exit(EXIT_FAILURE);
  }
  if (SSL_CTX_use_certificate_file(ctx, cert_file, SSL_FILETYPE_PEM)
<= 0) {
      ERR_print_errors_fp(stderr);
      exit(EXIT_FAILURE);
  }
  if (SSL_CTX_use_PrivateKey_file(ctx, key_file, SSL_FILETYPE_PEM) <=
0) {
      ERR_print_errors_fp(stderr);
      exit(EXIT_FAILURE);
  }
  if (!SSL_CTX_check_private_key(ctx)) {
      fprintf(stderr, "Private key does not match the public
certificate\n");
      exit(EXIT_FAILURE);
  }
  if(is_server) {
      SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER |
SSL_VERIFY_FAIL_IF_NO_PEER_CERT, NULL);
  } else {
      SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL);
  }
  return ctx;
}
static inline int get_peer_uuid_from_ssl(SSL* ssl, char* uuid_buffer,
size_t max_len) {
  X509* peer_cert = SSL_get_peer_certificate(ssl);
  if (!peer_cert) {
      return 0;
  }
  X509_NAME* subject_name = X509_get_subject_name(peer_cert);
  if (!subject_name) {
      X509_free(peer_cert);
      return 0;
  }
  X509_NAME_get_text_by_NID(subject_name, NID_commonName, uuid_buffer,
max_len);
  X509_free(peer_cert);
  if (strlen(uuid_buffer) == 0) {
```

```
      return 0;
  }
  return 1;
}
EOF
cat > "$R/common/compliance.h" <<'EOF'
#pragma once
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "logger.h"
#include "file_utils.h"
#define CONSENT_LOG_FILE "logs/consent_log.md"
#define BLOCKLIST_FILE "client_data/blocklist.conf"
static inline int check_consent(const char* profile_uuid) {
  FILE* f = fopen(CONSENT_LOG_FILE, "r");
  if (!f) return 0;
  char line[256];
  int found = 0;
  while (read_config_line(f, line, sizeof(line))) {
      if (strstr(line, profile_uuid) != NULL) {
          found = 1;
          break;
      }
  }
  fclose(f);
  return found;
}
static inline void record_consent(const char* profile_uuid) {
  FILE* f = fopen(CONSENT_LOG_FILE, "a");
  if (f) {
      time_t now = time(0);
      char buf[64];
      strftime(buf, sizeof(buf), "%Y-%m-%d %H:%M:%S",
localtime(&now));
      fprintf(f, "- Consent given by user %s at %s\n", profile_uuid,
buf);
      fclose(f);
      log_info("User consent has been recorded.");
  } else {
      log_error("Failed to open consent log file for writing.");
  }
}
static inline int get_user_consent(const char* profile_uuid) {
  if (check_consent(profile_uuid)) {
      log_info("User consent previously given.");
      return 1;
  }
```

```
  printf("\n--- [ CRTC & PIPEDA COMPLIANCE ] ---\n\n");
  printf("You are about to connect to the Yukki P2P network.\n");
  printf("By proceeding, you explicitly consent to the
following:\n\n");
  printf("1. Your client profile UUID (%s) and IP/Port will be sent to
the\n", profile_uuid);
  printf("   central bootstrap server.\n");
  printf("2. The bootstrap server will share your UUID and IP/Port
with other\n");
  printf("   authenticated peers on the network to facilitate P2P
connections.\n");
  printf("3. Other peers may connect directly to your client to
exchange\n");
  printf("   messages, files, or submit remote build jobs.\n\n");
  printf("All communication is secured with mTLS. Your consent is
logged locally\n");
  printf("in '%s' for audit purposes. This log is not
transmitted.\n\n", CONSENT_LOG_FILE);
  printf("Do you consent to these terms? (yes/no): ");
  char response[10];
  if (fgets(response, sizeof(response), stdin) != NULL) {
      response[strcspn(response, "\r\n")] = 0;
      if (strcmp(response, "yes") == 0) {
          record_consent(profile_uuid);
          return 1;
      }
  }
  log_error("User did not provide consent. Exiting.");
  return 0;
}
static inline int is_peer_blocked(const char* peer_uuid) {
  FILE* f = fopen(BLOCKLIST_FILE, "r");
  if (!f) return 0;
  char line[256];
  int found = 0;
  while (read_config_line(f, line, sizeof(line))) {
      if (strcmp(line, peer_uuid) == 0) {
          found = 1;
          break;
      }
  }
  fclose(f);
  return found;
}
static inline void block_peer(const char* peer_uuid) {
  if (is_peer_blocked(peer_uuid)) {
      printf("Peer %s is already blocked.\n", peer_uuid);
      return;
```

```
  }
  FILE* f = fopen(BLOCKLIST_FILE, "a");
  if (f) {
      fprintf(f, "%s\n", peer_uuid);
      fclose(f);
      printf("Blocked peer %s.\n", peer_uuid);
      log_info("Blocked peer: %s", peer_uuid);
  } else {
      log_error("Failed to open blocklist for writing.");
  }
}
static inline void unblock_peer(const char* peer_uuid) {
  if (!is_peer_blocked(peer_uuid)) {
      printf("Peer %s is not on the blocklist.\n", peer_uuid);
      return;
  }
  FILE* f_in = fopen(BLOCKLIST_FILE, "r");
  FILE* f_out = fopen(BLOCKLIST_FILE ".tmp", "w");
  if (!f_in || !f_out) {
      log_error("Failed to open blocklist files for update.");
      if (f_in) fclose(f_in);
      if (f_out) fclose(f_out);
      return;
  }
  char line[256];
  while (read_config_line(f_in, line, sizeof(line))) {
      if (strcmp(line, peer_uuid) != 0) {
          fprintf(f_out, "%s\n", line);
      }
  }
  fclose(f_in);
  fclose(f_out);
  remove(BLOCKLIST_FILE);
  rename(BLOCKLIST_FILE ".tmp", BLOCKLIST_FILE);
  printf("Unblocked peer %s.\n", peer_uuid);
  log_info("Unblocked peer: %s", peer_uuid);
}
static inline void show_profile(const char* profile_uuid) {
  printf("\n--- [ User Profile ] ---\n");
  printf("UUID: %s\n", profile_uuid);
  printf("\n--- [ Blocked Peers ] ---\n");
  FILE* f = fopen(BLOCKLIST_FILE, "r");
  if (f) {
      char line[256];
      int count = 0;
      while (read_config_line(f, line, sizeof(line))) {
          printf("- %s\n", line);
          count++;
```

```
      }
      if (count == 0) {
          printf("(No peers are blocked)\n");
      }
      fclose(f);
  } else {
      printf("(Blocklist file not found)\n");
  }
  printf("\n");
}
EOF
};c(){ cat > "$R/common/adi_protocol.h" <<'EOF'
#pragma once
#include <stdint.h>
#include <string.h>
#include <openssl/ssl.h>
#include <pthread.h>
#include "logger.h"
#if defined(__x86_64__) || defined(_M_X64)
  #if defined(_MSC_VER)
      #include <intrin.h>
  #else
      #include <x86intrin.h>
  #endif
#endif
static inline void adi_write_u32_be(uint32_t value, uint8_t* dst) {
#if defined(_MSC_VER) && (defined(_M_X64) || defined(_M_ARM64))
  *(uint32_t*)dst = _byteswap_ulong(value);
#elif defined(__GNUC__) || defined(__clang__)
  *(uint32_t*)dst = __builtin_bswap32(value);
#else
  dst[0] = (value >> 24) & 0xFF;
  dst[1] = (value >> 16) & 0xFF;
  dst[2] = (value >> 8) & 0xFF;
  dst[3] = value & 0xFF;
#endif
}
static inline uint32_t adi_read_u32_be(const uint8_t* src) {
#if defined(_MSC_VER) && (defined(_M_X64) || defined(_M_ARM64))
  return _byteswap_ulong(*(uint32_t*)src);
#elif defined(__GNUC__) || defined(__clang__)
  return __builtin_bswap32(*(uint32_t*)src);
#else
  return ((uint32_t)src[0] << 24) |
         ((uint32_t)src[1] << 16) |
         ((uint32_t)src[2] << 8)  |
         ((uint32_t)src[3]);
#endif
```

```
}
static inline int adi_send_all(SSL* ssl, const void* buffer, size_t
len) {
  const char* p = (const char*)buffer;
  while (len > 0) {
      int bytes_sent = SSL_write(ssl, p, (int)len);
      if (bytes_sent <= 0) {
          int ssl_err = SSL_get_error(ssl, bytes_sent);
          log_warn("SSL_write error: %d (code %d)", bytes_sent,
ssl_err);
          ERR_print_errors_fp(stderr);
          return -1;
      }
      p += bytes_sent;
      len -= bytes_sent;
  }
  return 0;
}
static inline int adi_read_exact_n(SSL* ssl, void* buffer, size_t len)
{
  char* p = (char*)buffer;
  while (len > 0) {
      int bytes_read = SSL_read(ssl, p, (int)len);
      if (bytes_read <= 0) {
          int ssl_err = SSL_get_error(ssl, bytes_read);
          if (ssl_err == SSL_ERROR_ZERO_RETURN || ssl_err ==
SSL_ERROR_SYSCALL) {
              log_info("Peer disconnected.");
          } else {
              log_warn("SSL_read error: %d (code %d)", bytes_read,
ssl_err);
              ERR_print_errors_fp(stderr);
          }
          return -1;
      }
      p += bytes_read;
      len -= bytes_read;
  }
  return 0;
}
typedef enum {
  P2P_MSG_CMD = 1,
  P2P_GET_REQ = 2,
  P2P_SEND_REQ = 3,
  P2P_LS_REQ = 4,
  P2P_FILE_CHUNK = 5,
  P2P_FILE_END = 6,
  P2P_ERROR = 7,
```

```
  P2P_LS_RESP = 8,
  P2P_JOB_SUBMIT_REQ = 9,
  P2P_JOB_SUBMIT_RESP = 10,
  P2P_JOB_STATUS_REQ = 11,
  P2P_JOB_STATUS_RESP = 12
} YukkiPacketType;
#define YUKKI_HEADER_LEN 5
#define MAX_PAYLOAD_SIZE (1024 * 64)
#define FILE_CHUNK_SIZE (1024 * 32)
static inline int adi_send_packet(SSL* ssl, YukkiPacketType type,
const void* payload, uint32_t payload_len) {
  uint32_t total_payload_len = payload_len + 1;
  if (total_payload_len > MAX_PAYLOAD_SIZE) {
      log_error("Attempted to send packet larger than
MAX_PAYLOAD_SIZE");
      return -1;
  }
  static uint8_t packet_buf[MAX_PAYLOAD_SIZE + YUKKI_HEADER_LEN];
  static pthread_mutex_t packet_buf_mutex = PTHREAD_MUTEX_INITIALIZER;
  pthread_mutex_lock(&packet_buf_mutex);
  adi_write_u32_be(total_payload_len, packet_buf);
  packet_buf[4] = (uint8_t)type;
  if (payload_len > 0) {
      memcpy(packet_buf + YUKKI_HEADER_LEN, payload, payload_len);
  }
  int result = adi_send_all(ssl, packet_buf, YUKKI_HEADER_LEN +
payload_len);
  pthread_mutex_unlock(&packet_buf_mutex);
  return result;
}
EOF
};d(){ cat > "$R/yukki_c2_suite/bootstrap_server/server.c" <<'EOF'
#include "../../vendor/mongoose.h"
#include "../../common/logger.h"
#include "../../common/pki_utils.h"
#include <openssl/ssl.h>
#include <pthread.h>
#include <stdlib.h>
#define MAX_PEERS 1024
#define BROADCAST_INTERVAL_MS 5000
#define CA_CERT "pki/ca_cert.pem"
#define SERVER_CERT "pki/server_cert.pem"
#define SERVER_KEY "pki/server_key.pem"
typedef struct {
  char uuid[64];
  char address[128];
  struct mg_connection* conn;
} Peer;
```

```
static Peer peer_db[MAX_PEERS];
static int peer_count = 0;
static pthread_mutex_t peer_db_mutex = PTHREAD_MUTEX_INITIALIZER;
static int find_peer_by_conn(struct mg_connection* conn) {
  for (int i = 0; i < peer_count; i++) {
      if (peer_db[i].conn == conn) return i;
  }
  return -1;
}
static void remove_peer(struct mg_connection* conn) {
  pthread_mutex_lock(&peer_db_mutex);
  int index = find_peer_by_conn(conn);
  if (index != -1) {
      c2_log_info("Peer %s disconnected.", peer_db[index].uuid);
      for (int j = index; j < peer_count - 1; j++) {
          peer_db[j] = peer_db[j + 1];
      }
      peer_count--;
  }
  pthread_mutex_unlock(&peer_db_mutex);
}
static void add_peer(const char* uuid, const char* address, struct
mg_connection* conn) {
  pthread_mutex_lock(&peer_db_mutex);
  if (peer_count < MAX_PEERS) {
      Peer* p = &peer_db[peer_count];
      strncpy(p->uuid, uuid, sizeof(p->uuid) - 1);
      strncpy(p->address, address, sizeof(p->address) - 1);
      p->conn = conn;
      peer_count++;
      c2_log_info("Peer %s connected from %s. Total peers: %d", uuid,
address, peer_count);
  } else {
      c2_log_warn("Max peers reached. %s from %s rejected.", uuid,
address);
  }
  pthread_mutex_unlock(&peer_db_mutex);
}
static void build_peer_list_json(char* buffer, size_t max_len) {
  pthread_mutex_lock(&peer_db_mutex);
  snprintf(buffer, max_len, "{\"type\": \"peer_list\", \"peers\": [");
  for (int i = 0; i < peer_count; i++) {
      char peer_entry[256];
      snprintf(peer_entry, sizeof(peer_entry), "{\"uuid\": \"%s\",
\"addr\": \"%s\"}%s",
               peer_db[i].uuid, peer_db[i].address, (i == peer_count -
1) ? "" : ",");
      strncat(buffer, peer_entry, max_len - strlen(buffer) - 1);
```

```
  }
  strncat(buffer, "]}", max_len - strlen(buffer) - 1);
  pthread_mutex_unlock(&peer_db_mutex);
}
static void broadcast_peer_list(struct mg_mgr* mgr) {
  char json_buffer[4096 * 4];
  build_peer_list_json(json_buffer, sizeof(json_buffer));
  pthread_mutex_lock(&peer_db_mutex);
  for (int i = 0; i < peer_count; i++) {
      mg_ws_send(peer_db[i].conn, json_buffer, strlen(json_buffer),
WEBSOCKET_OP_TEXT);
  }
  pthread_mutex_unlock(&peer_db_mutex);
}
static void fn(struct mg_connection* c, int ev, void* ev_data, void*
fn_data) {
  if (ev == MG_EV_ERROR) {
      c2_log_error("%p %s", c->fd, (char*)ev_data);
  } else if (ev == MG_EV_ACCEPT) {
      char uuid_buf[64];
      if (get_peer_uuid_from_ssl(c->tls, uuid_buf, sizeof(uuid_buf)))
{
          c2_log_info("mTLS handshake successful for peer: %s",
uuid_buf);
      } else {
          c2_log_error("mTLS handshake failed: Could not extract UUID
from client cert.");
          mg_close_conn(c);
      }
  } else if (ev == MG_EV_WS_OPEN) {
      char uuid_buf[64];
      char peer_addr[128];
      struct mg_http_message* hm = (struct mg_http_message*)ev_data;
      if (!get_peer_uuid_from_ssl(c->tls, uuid_buf, sizeof(uuid_buf)))
{
          c2_log_error("Rejecting connection: Failed to get UUID
post-handshake.");
          mg_close_conn(c);
          return;
      }
      struct mg_str* port_hdr = mg_http_get_header(hm, "X-Peer-Port");
      if (!port_hdr) {
          c2_log_error("Rejecting peer %s: Missing 'X-Peer-Port'
header.", uuid_buf);
          mg_close_conn(c);
          return;
      }
      char ip_str[64];
```

```
      mg_ntoa(&c->rem, ip_str, sizeof(ip_str));
      snprintf(peer_addr, sizeof(peer_addr), "%s:%.*s", ip_str,
(int)port_hdr->len, port_hdr->ptr);
      add_peer(uuid_buf, peer_addr, c);
  } else if (ev == MG_EV_CLOSE) {
      remove_peer(c);
  } else if (ev == MG_EV_TIMER) {
      broadcast_peer_list(c->mgr);
      mg_timer_add(c->mgr, BROADCAST_INTERVAL_MS, MG_TIMER_ONESHOT,
c->fn, NULL);
  }
  (void)fn_data;
}
int main(void) {
  struct mg_mgr mgr;
  SSL_CTX* ctx;
  mkdir("logs", 0755);
  c2_log_info("Starting Yukki OS 3.2 Bootstrap Server...");
  c2_log_info("Initializing mTLS context...");
  ctx = create_ssl_context(CA_CERT, SERVER_CERT, SERVER_KEY, 1);
  if (!ctx) {
      c2_log_error("Failed to create SSL context. Check PKI files.");
      return 1;
  }
  c2_log_info("mTLS context created successfully.");
  mg_mgr_init(&mgr);
  struct mg_listen_opts opts = {
      .ssl_ca = CA_CERT,
      .ssl_cert = SERVER_CERT,
      .ssl_key = SERVER_KEY,
      .user_data = ctx
  };
  c2_log_info("Attempting to listen on wss://0.0.0.0:8443");
  if (mg_listen(&mgr, "wss://0.0.0.0:8443", fn, &opts) == NULL) {
      c2_log_error("Failed to start listener. Is port 8443 in use?");
      SSL_CTX_free(ctx);
      mg_mgr_free(&mgr);
      return 1;
  }
  c2_log_info("Server listening on wss://0.0.0.0:8443");
  mg_timer_add(&mgr, BROADCAST_INTERVAL_MS, MG_TIMER_ONESHOT, fn,
NULL);
  while (1) {
      mg_mgr_poll(&mgr, 1000);
  }
  mg_mgr_free(&mgr);
  SSL_CTX_free(ctx);
  EVP_cleanup();
```

```
  return 0;
}
EOF
cat > "$R/yukki_c2_suite/bootstrap_server/Makefile" <<'EOF'
CC = gcc
CFLAGS = -I"$(CURDIR)/../../vendor" -Wall -Wextra -g -pthread
LDFLAGS = -lssl -lcrypto -pthread
VENDOR_SRC = $(wildcard $(CURDIR)/../../vendor/*.c)
VENDOR_OBJ = $(VENDOR_SRC:.c=.o)
TARGET = bootstrap_server
all: $(TARGET)
$(TARGET): server.o $(VENDOR_OBJ)
	$(CC) $(CFLAGS) -o $(TARGET) server.o $(VENDOR_OBJ) $(LDFLAGS)
server.o: server.c
	$(CC) $(CFLAGS) -c server.c -o server.o
$(CURDIR)/../../vendor/%.o: $(CURDIR)/../../vendor/%.c
	$(CC) $(CFLAGS) -c $< -o $@
clean:
	rm -f $(TARGET) *.o $(VENDOR_OBJ)
EOF
};e(){ cat > "$R/yukki_c2_suite/peer_client/build_worker.h" <<'EOF'
#pragma once
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <unistd.h>
#include "../../common/logger.h"
#define MAX_JOBS 100
#define MAX_CMD_LENGTH 1024
#define MAX_NAME_LENGTH 128
#define MAX_JOB_DEPS 10
typedef enum {
  JOB_STATUS_PENDING,
  JOB_STATUS_RUNNING,
  JOB_STATUS_COMPLETED,
  JOB_STATUS_FAILED
} JobStatus;
static const char* get_job_status_string(JobStatus status) {
  switch (status) {
      case JOB_STATUS_PENDING:   return "PENDING";
      case JOB_STATUS_RUNNING:   return "RUNNING";
      case JOB_STATUS_COMPLETED: return "COMPLETED";
      case JOB_STATUS_FAILED:    return "FAILED";
      default:                   return "UNKNOWN";
  }
}
typedef struct {
```

```
  int id;
  char name[MAX_NAME_LENGTH];
  char command[MAX_CMD_LENGTH];
  JobStatus status;
  int dependencies[MAX_JOB_DEPS];
  int dependency_count;
} BuildJob;
static BuildJob job_queue[MAX_JOBS];
static int job_count = 0;
static int next_job_id = 1;
static pthread_mutex_t job_queue_mutex = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t job_queue_cond = PTHREAD_COND_INITIALIZER;
static int stop_worker = 0;
static int submit_job(const char* name, const char* command, const
int* deps, int dep_count) {
  pthread_mutex_lock(&job_queue_mutex);
  if (job_count >= MAX_JOBS) {
      printf("Error: Job queue is full.\n");
      pthread_mutex_unlock(&job_queue_mutex);
      return -1;
  }
  BuildJob* job = &job_queue[job_count];
  job->id = next_job_id++;
  strncpy(job->name, name, MAX_NAME_LENGTH - 1);
  job->name[MAX_NAME_LENGTH - 1] = '\0';
  strncpy(job->command, command, MAX_CMD_LENGTH - 1);
  job->command[MAX_CMD_LENGTH - 1] = '\0';
  job->status = JOB_STATUS_PENDING;
  job->dependency_count = 0;
  if (deps && dep_count > 0) {
      job->dependency_count = (dep_count > MAX_JOB_DEPS) ?
MAX_JOB_DEPS : dep_count;
      memcpy(job->dependencies, deps, job->dependency_count *
sizeof(int));
  }
  job_count++;
  int new_job_id = job->id;
  printf("Job %d ('%s') submitted to local queue.\n", new_job_id,
name);
  log_info("Job %d submitted: %s", new_job_id, command);
  pthread_cond_signal(&job_queue_cond);
  pthread_mutex_unlock(&job_queue_mutex);
  return new_job_id;
}
static void format_job_status_string(BuildJob* job, char* buf, size_t
max_len) {
  char deps_buf[256] = {0};
  if (job->dependency_count > 0) {
```

```
      char temp[32];
      snprintf(deps_buf, sizeof(deps_buf), " (Deps: ");
      for (int i = 0; i < job->dependency_count; i++) {
          snprintf(temp, sizeof(temp), "%d%s", job->dependencies[i],
(i == job->dependency_count - 1) ? ")" : ", ");
          strncat(deps_buf, temp, sizeof(deps_buf) - strlen(deps_buf)
- 1);
      }
  }
  snprintf(buf, max_len, "ID %d: %s (%s)%s\n  Cmd: %s",
           job->id, job->name, get_job_status_string(job->status),
           deps_buf, job->command);
}
static int find_job_status(int job_id, char* buf, size_t max_len) {
  pthread_mutex_lock(&job_queue_mutex);
  int found = 0;
  for (int i = 0; i < job_count; i++) {
      if (job_queue[i].id == job_id) {
          format_job_status_string(&job_queue[i], buf, max_len);
          found = 1;
          break;
      }
  }
  pthread_mutex_unlock(&job_queue_mutex);
  if (!found) {
      snprintf(buf, max_len, "Job ID %d not found.", job_id);
  }
  return found;
}
static void show_job_queue() {
  pthread_mutex_lock(&job_queue_mutex);
  printf("\n--- [ Local Build Job Queue ] ---\n");
  if (job_count == 0) {
      printf("(No jobs in queue)\n");
  } else {
      char status_buf[2048];
      for (int i = 0; i < job_count; i++) {
          format_job_status_string(&job_queue[i], status_buf,
sizeof(status_buf));
          printf("- %s\n", status_buf);
      }
  }
  printf("\n");
  pthread_mutex_unlock(&job_queue_mutex);
}
static int check_dependencies(BuildJob* job) {
  for (int d = 0; d < job->dependency_count; d++) {
      int dep_id = job->dependencies[d];
```

```
      int dep_met = 0;
      for (int k = 0; k < job_count; k++) {
          if (job_queue[k].id == dep_id) {
              if (job_queue[k].status == JOB_STATUS_COMPLETED) {
                  dep_met = 1;
              }
              break;
          }
      }
      if (!dep_met) {
          return 0;
      }
  }
  return 1;
}
static void* build_worker_thread(void* arg) {
  (void)arg;
  log_info("Build worker thread (JobbySlotty, Dependency-Aware)
started.");
  while (!stop_worker) {
      int job_executed_this_cycle = 0;
      pthread_mutex_lock(&job_queue_mutex);
      BuildJob* job = NULL;
      for (int i = 0; i < job_count; i++) {
          if (job_queue[i].status == JOB_STATUS_PENDING) {
              if (check_dependencies(&job_queue[i])) {
                  job = &job_queue[i];
                  job->status = JOB_STATUS_RUNNING;
                  job_executed_this_cycle = 1;
                  break;
              }
          }
      }
      if (!job && !stop_worker) {
          pthread_cond_wait(&job_queue_cond, &job_queue_mutex);
      }
      pthread_mutex_unlock(&job_queue_mutex);
      if (stop_worker) break;
      if (job) {
          log_info("Worker picking up job %d: %s", job->id,
job->command);
          printf("--- [Build Worker] Starting job %d: %s ---\n",
job->id, job->name);
          char log_filename[256];
          snprintf(log_filename, sizeof(log_filename),
"logs/job_%d.log", job->id);
          char full_command[MAX_CMD_LENGTH + 128];
          snprintf(full_command, sizeof(full_command), "%s > %s 2>&1",
```

```
job->command, log_filename);
          int result = system(full_command);
          pthread_mutex_lock(&job_queue_mutex);
          job->status = (result == 0) ? JOB_STATUS_COMPLETED :
JOB_STATUS_FAILED;
          pthread_cond_broadcast(&job_queue_cond);
          pthread_mutex_unlock(&job_queue_mutex);
          printf("--- [Build Worker] Job %d ('%s') finished with
status: %s ---\n",
                 job->id, job->name, (result == 0) ? "COMPLETED" :
"FAILED");
          printf("--- [Build Worker] Log available at: %s ---\n",
log_filename);
          log_info("Job %d finished with status %d", job->id, result);
      }
  }
  log_info("Build worker thread shutting down.");
  return NULL;
}
EOF
cat > "$R/yukki_c2_suite/peer_client/client.c" <<'EOF'
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include "../../vendor/mongoose.h"
#include "../../vendor/cJSON.h"
#include "../../common/logger.h"
#include "../../common/pki_utils.h"
#include "../../common/file_utils.h"
#include "../../common/compliance.h"
#include "../../common/adi_protocol.h"
#include "build_worker.h"
#include "../../vendor/linenoise.h"
#define MAX_PEERS 1024
#define CONFIG_FILE_PREFIX "client_data/"
#define RECEIVED_FILES_DIR "received_files"
static char g_profile_name[64];
static char g_profile_uuid[64];
static char g_config_file[128];
static char g_p2p_port[10];
static char g_ca_cert[128];
static char g_client_cert[128];
static char g_client_key[128];
```

```
static int g_stop_client = 0;
// --- [3.2 MOD] Add global to hold prompt setting ---
static int g_fancy_prompt = 0; // Default to 'off'
// --- [3.2 END] ---
// NEW: Tab completion list
static const char* commands[] = {
   "profile", "block", "unblock", "job", "rjob", "peers", "msg",
"say",
   "join", "send", "get", "ls", "help", "quit", "exit", NULL
};
// NEW: Tab completion callback
void completion(const char *buf, linenoiseCompletions *lc) {
   char cmd_buf[128];
   strncpy(cmd_buf, buf, sizeof(cmd_buf) - 1);
   cmd_buf[sizeof(cmd_buf) - 1] = '\0';
   char *first_space = strstr(cmd_buf, " ");
   if (!first_space) {
       // Completing top-level command
       for (int i = 0; commands[i] != NULL; i++) {
           if (strncmp(buf, commands[i], strlen(buf)) == 0) {
               linenoiseAddCompletion(lc, commands[i]);
           }
       }
   } else {
       // Completing an argument (likely a UUID)
       *first_space = '\0'; // Isolate command
       char* prefix = (char*)buf + (first_space - cmd_buf) + 1; //
Start of arg
       // List of commands that take a UUID as their *first* argument
       if (strcmp(cmd_buf, "block") == 0 || strcmp(cmd_buf, "unblock")
== 0 ||
           strcmp(cmd_buf, "msg") == 0 || strcmp(cmd_buf, "send") == 0
||
           strcmp(cmd_buf, "get") == 0 || strcmp(cmd_buf, "ls") == 0)
       {
           // Check if we are still completing the *first* arg (no
more spaces)
           if (strstr(prefix, " ") == NULL) {
               pthread_mutex_lock(&g_peer_list_mutex);
               for (int i = 0; i < g_peer_count; i++) {
                   if (strncmp(prefix, g_peer_list[i].uuid,
strlen(prefix)) == 0) {
                       char full_line[256];
                       snprintf(full_line, sizeof(full_line), "%s %s",
cmd_buf, g_peer_list[i].uuid);
                       linenoiseAddCompletion(lc, full_line);
                   }
               }
```

```
                pthread_mutex_unlock(&g_peer_list_mutex);
            }
        }
    }
}
typedef struct {
  char uuid[64];
  char addr[128];
} PeerInfo;
static PeerInfo g_peer_list[MAX_PEERS];
static int g_peer_count = 0;
static pthread_mutex_t g_peer_list_mutex = PTHREAD_MUTEX_INITIALIZER;
typedef struct {
  SSL* ssl;
  char uuid[64];
  int in_use;
  FILE* incoming_file_handle;
} PeerConnection;
static PeerConnection g_peer_connections[MAX_PEERS];
static pthread_mutex_t g_connections_mutex =
PTHREAD_MUTEX_INITIALIZER;
static int find_free_connection_slot() {
  for (int i = 0; i < MAX_PEERS; i++) {
      if (!g_peer_connections[i].in_use) return i;
  }
  return -1;
}
static int add_peer_connection(SSL* ssl, const char* uuid) {
  pthread_mutex_lock(&g_connections_mutex);
  int slot = find_free_connection_slot();
  if (slot != -1) {
      g_peer_connections[slot].in_use = 1;
      g_peer_connections[slot].ssl = ssl;
      strncpy(g_peer_connections[slot].uuid, uuid,
sizeof(g_peer_connections[slot].uuid) - 1);
      g_peer_connections[slot].incoming_file_handle = NULL;
  }
  pthread_mutex_unlock(&g_connections_mutex);
  if (slot == -1) log_warn("Max peer connections reached.");
  return slot;
}
static void remove_peer_connection(int slot) {
  if (slot < 0 || slot >= MAX_PEERS ||
!g_peer_connections[slot].in_use) return;
  pthread_mutex_lock(&g_connections_mutex);
  if (!g_peer_connections[slot].in_use) {
      pthread_mutex_unlock(&g_connections_mutex);
      return;
```

```
  }
  log_info("Closing P2P connection with %s",
g_peer_connections[slot].uuid);
  if (g_peer_connections[slot].ssl) {
      SSL_shutdown(g_peer_connections[slot].ssl);
      SSL_free(g_peer_connections[slot].ssl);
  }
  if (g_peer_connections[slot].incoming_file_handle) {
      fclose(g_peer_connections[slot].incoming_file_handle);
  }
  memset(&g_peer_connections[slot], 0, sizeof(PeerConnection));
  pthread_mutex_unlock(&g_connections_mutex);
}
static SSL* get_ssl_for_peer_uuid(const char* uuid) {
  pthread_mutex_lock(&g_connections_mutex);
  SSL* ssl = NULL;
  for (int i = 0; i < MAX_PEERS; i++) {
      if (g_peer_connections[i].in_use &&
strcmp(g_peer_connections[i].uuid, uuid) == 0) {
          ssl = g_peer_connections[i].ssl;
          break;
      }
  }
  pthread_mutex_unlock(&g_connections_mutex);
  return ssl;
}
static int get_slot_for_ssl(SSL* ssl) {
  pthread_mutex_lock(&g_connections_mutex);
  int slot = -1;
  for (int i = 0; i < MAX_PEERS; i++) {
      if (g_peer_connections[i].in_use && g_peer_connections[i].ssl ==
ssl) {
          slot = i;
          break;
      }
  }
  pthread_mutex_unlock(&g_connections_mutex);
  return slot;
}
static int get_peer_addr_by_uuid(const char* uuid, char* addr_buf,
size_t max_len) {
  pthread_mutex_lock(&g_peer_list_mutex);
  int found = 0;
  for (int i = 0; i < g_peer_count; i++) {
      if (strcmp(g_peer_list[i].uuid, uuid) == 0) {
          strncpy(addr_buf, g_peer_list[i].addr, max_len - 1);
          addr_buf[max_len - 1] = '\0';
          found = 1;
```

```
            break;
        }
    }
    pthread_mutex_unlock(&g_peer_list_mutex);
    return found;
}
void* handle_peer_client_thread(void* arg);
static SSL* connect_to_peer(const char* uuid) {
    SSL* ssl = get_ssl_for_peer_uuid(uuid);
    if (ssl) {
        log_debug("Reusing existing connection for peer %s", uuid);
        return ssl;
    }
    if (is_peer_blocked(uuid)) {
        printf("Error: Peer %s is on your blocklist.\n", uuid);
        return NULL;
    }
    char addr_str[128];
    if (!get_peer_addr_by_uuid(uuid, addr_str, sizeof(addr_str))) {
        printf("Error: Peer %s not found in network list.\n", uuid);
        return NULL;
    }
    char peer_ip[64];
    int peer_port;
    if (sscanf(addr_str, "%[^:]:%d", peer_ip, &peer_port) != 2) {
        printf("Error: Invalid address format for peer %s: %s\n", uuid,
addr_str);
        return NULL;
    }
    log_info("Attempting new P2P connection to %s at %s:%d", uuid,
peer_ip, peer_port);
    SSL_CTX* ctx = create_ssl_context(g_ca_cert, g_client_cert,
g_client_key, 0);
    if (!ctx) {
        log_error("Failed to create SSL context for outgoing P2P.");
        return NULL;
    }
    int sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("socket");
        SSL_CTX_free(ctx);
        return NULL;
    }
    struct sockaddr_in sa = {0};
    sa.sin_family = AF_INET;
    sa.sin_port = htons(peer_port);
    sa.sin_addr.s_addr = inet_addr(peer_ip);
    if (connect(sock, (struct sockaddr*)&sa, sizeof(sa)) != 0) {
```

```
        perror("connect");
        close(sock);
        SSL_CTX_free(ctx);
        return NULL;
    }
    ssl = SSL_new(ctx);
    SSL_set_fd(ssl, sock);
    if (SSL_connect(ssl) <= 0) {
        log_error("SSL_connect failed for peer %s", uuid);
        ERR_print_errors_fp(stderr);
        SSL_free(ssl);
        close(sock);
        SSL_CTX_free(ctx);
        return NULL;
    }
    char cert_uuid[64];
    if (!get_peer_uuid_from_ssl(ssl, cert_uuid, sizeof(cert_uuid))) {
        log_error("Peer %s provided invalid/missing certificate.",
uuid);
        SSL_free(ssl);
        close(sock);
        SSL_CTX_free(ctx);
        return NULL;
    }
    if (strcmp(uuid, cert_uuid) != 0) {
        log_error("Peer UUID mismatch! Expected %s, got %s.", uuid,
cert_uuid);
        SSL_free(ssl);
        close(sock);
        SSL_CTX_free(ctx);
        return NULL;
    }
    log_info("Successfully connected to peer %s", uuid);
    int slot = add_peer_connection(ssl, uuid);
    if (slot == -1) {
        log_error("Failed to add peer connection, pool is full.");
        SSL_free(ssl);
        close(sock);
        SSL_CTX_free(ctx);
        return NULL;
    }
    SSL_CTX_free(ctx);
    int* thread_arg = malloc(sizeof(int));
    *thread_arg = slot;
    pthread_t handler_thread;
    if (pthread_create(&handler_thread, NULL, handle_peer_client_thread,
thread_arg) != 0) {
        log_error("Failed to create handler thread for outgoing peer
```

```
%s", uuid);
      remove_peer_connection(slot);
      return NULL;
  }
  pthread_detach(handler_thread);
  return ssl;
}
void* handle_peer_client_thread(void* arg) {
  int slot = *(int*)arg;
  free(arg);
  pthread_mutex_lock(&g_connections_mutex);
  if (slot < 0 || slot >= MAX_PEERS ||
!g_peer_connections[slot].in_use) {
      pthread_mutex_unlock(&g_connections_mutex);
      return NULL;
  }
  SSL* ssl = g_peer_connections[slot].ssl;
  char peer_uuid[64];
  strncpy(peer_uuid, g_peer_connections[slot].uuid, sizeof(peer_uuid)
- 1);
  pthread_mutex_unlock(&g_connections_mutex);
  log_info("Handler thread started for peer %s", peer_uuid);
  uint8_t header[YUKKI_HEADER_LEN];
  uint8_t* payload_buf = malloc(MAX_PAYLOAD_SIZE + 1);
  if (!payload_buf) {
      log_error("Failed to allocate payload buffer for thread %s",
peer_uuid);
      remove_peer_connection(slot);
      return NULL;
  }
  while (!g_stop_client) {
      if (adi_read_exact_n(ssl, header, YUKKI_HEADER_LEN) != 0) {
          log_info("Peer %s disconnected (header read).", peer_uuid);
          break;
      }
      uint32_t payload_len = adi_read_u32_be(header) - 1;
      YukkiPacketType type = (YukkiPacketType)header[4];
      if (payload_len > MAX_PAYLOAD_SIZE) {
          log_error("Peer %s sent oversized payload (%u bytes).
Disconnecting.", peer_uuid, payload_len);
          break;
      }
      if (payload_len > 0) {
          if (adi_read_exact_n(ssl, payload_buf, payload_len) != 0) {
              log_info("Peer %s disconnected (payload read).",
peer_uuid);
              break;
          }
```

```
        }
        payload_buf[payload_len] = '\0';
        switch (type) {
            case P2P_MSG_CMD:
                printf("\n[Message from %s]: %s\n> ", peer_uuid,
(char*)payload_buf);
                fflush(stdout);
                break;
            case P2P_SEND_REQ: {
                char* filename = (char*)payload_buf;
                char safe_path[512];
                char* safe_filename = strrchr(filename, '/');
                safe_filename = (safe_filename) ? safe_filename + 1 :
filename;
                snprintf(safe_path, sizeof(safe_path), "%s/%s",
RECEIVED_FILES_DIR, safe_filename);
                pthread_mutex_lock(&g_connections_mutex);
                if (g_peer_connections[slot].incoming_file_handle) {

fclose(g_peer_connections[slot].incoming_file_handle);
                }
                g_peer_connections[slot].incoming_file_handle =
fopen(safe_path, "wb");
                pthread_mutex_unlock(&g_connections_mutex);
                if (g_peer_connections[slot].incoming_file_handle) {
                    printf("\n[File Transfer] Receiving '%s' from
%s...\n> ", safe_filename, peer_uuid);
                    log_info("Receiving file %s from %s", safe_path,
peer_uuid);
                    fflush(stdout);
                } else {
                    log_error("Failed to open file %s for writing.",
safe_path);
                    adi_send_packet(ssl, P2P_ERROR, "Failed to open file
for writing", 30);
                }
                break;
            }
            case P2P_FILE_CHUNK: {
                pthread_mutex_lock(&g_connections_mutex);
                FILE* f = g_peer_connections[slot].incoming_file_handle;
                if (f) {
                    fwrite(payload_buf, 1, payload_len, f);
                }
                pthread_mutex_unlock(&g_connections_mutex);
                break;
            }
            case P2P_FILE_END: {
```

```
            pthread_mutex_lock(&g_connections_mutex);
            if (g_peer_connections[slot].incoming_file_handle) {

fclose(g_peer_connections[slot].incoming_file_handle);
                g_peer_connections[slot].incoming_file_handle =
NULL;
                printf("\n[File Transfer] File from %s complete.\n>
", peer_uuid);
                log_info("File transfer from %s complete.",
peer_uuid);
                fflush(stdout);
            }
            pthread_mutex_unlock(&g_connections_mutex);
            break;
        }
        case P2P_GET_REQ: {
            char* filename = (char*)payload_buf;
            char safe_path[512];
            char* safe_filename = strrchr(filename, '/');
            safe_filename = (safe_filename) ? safe_filename + 1 :
filename;
            snprintf(safe_path, sizeof(safe_path), "%s/%s",
RECEIVED_FILES_DIR, safe_filename);
            printf("\n[File Transfer] Peer %s requests '%s'...\n> ",
peer_uuid, safe_filename);
            fflush(stdout);
            FILE* f = fopen(safe_path, "rb");
            if (!f) {
                log_warn("Peer %s requested non-existent file: %s",
peer_uuid, safe_path);
                adi_send_packet(ssl, P2P_ERROR, "File not found or
access denied.", 30);
                break;
            }
            log_info("Sending file %s to %s", safe_path, peer_uuid);
            uint8_t* file_chunk_buf = malloc(FILE_CHUNK_SIZE);
            if (!file_chunk_buf) {
                log_error("Failed to allocate file chunk buffer for
%s", peer_uuid);
                fclose(f);
                break;
            }
            size_t bytes_read;
            while ((bytes_read = fread(file_chunk_buf, 1,
FILE_CHUNK_SIZE, f)) > 0) {
                if (adi_send_packet(ssl, P2P_FILE_CHUNK,
file_chunk_buf, bytes_read) != 0) {
                    log_warn("Connection lost sending file chunk to
```

```
%s", peer_uuid);
                    break;
                }
            }
            free(file_chunk_buf);
            fclose(f);
            adi_send_packet(ssl, P2P_FILE_END, NULL, 0);
            log_info("File send %s to %s complete.", safe_path,
peer_uuid);
            break;
        }
        case P2P_LS_REQ: {
            printf("\n[File Transfer] Peer %s requests 'ls'...\n> ",
peer_uuid);
            fflush(stdout);
            char ls_cmd[512];
            snprintf(ls_cmd, sizeof(ls_cmd), "ls -l %s",
RECEIVED_FILES_DIR);
            FILE* pipe = popen(ls_cmd, "r");
            if (!pipe) {
                adi_send_packet(ssl, P2P_ERROR, "Failed to execute
'ls' on server.", 33);
                break;
            }
            char* ls_buffer = malloc(4096);
            if (!ls_buffer) {
                pclose(pipe);
                adi_send_packet(ssl, P2P_ERROR, "Server internal
memory error.", 28);
                break;
            }
            size_t bytes_read = fread(ls_buffer, 1, 4096 - 1, pipe);
            ls_buffer[bytes_read] = '\0';
            pclose(pipe);
            adi_send_packet(ssl, P2P_LS_RESP, ls_buffer,
bytes_read);
            free(ls_buffer);
            break;
        }
        case P2P_LS_RESP:
            printf("\n[List from %s]:\n%s\n> ", peer_uuid,
(char*)payload_buf);
            fflush(stdout);
            break;
        case P2P_JOB_SUBMIT_REQ: {
            printf("\n[Remote Job] Received job request from %s\n>
", peer_uuid);
            fflush(stdout);
```

```
            char* payload_copy = strdup((char*)payload_buf);
            char* job_name = strtok(payload_copy, "\n");
            char* job_cmd = strtok(NULL, "\n");
            char* deps_str = strtok(NULL, "\n");
            int deps[MAX_JOB_DEPS];
            int dep_count = 0;
            if (deps_str && strlen(deps_str) > 0) {
                char* dep_id_str = strtok(deps_str, ",");
                while (dep_id_str != NULL && dep_count <
MAX_JOB_DEPS) {
                    deps[dep_count++] = atoi(dep_id_str);
                    dep_id_str = strtok(NULL, ",");
                }
            }
            if (job_name && job_cmd) {
                int new_job_id = submit_job(job_name, job_cmd, deps,
dep_count);
                char resp_payload[256];
                if (new_job_id != -1) {
                    snprintf(resp_payload, sizeof(resp_payload),
"Job %d ('%s') submitted locally.", new_job_id, job_name);
                } else {
                    snprintf(resp_payload, sizeof(resp_payload),
"Error: Failed to submit job ('%s'). Queue full.", job_name);
                }
                adi_send_packet(ssl, P2P_JOB_SUBMIT_RESP,
resp_payload, strlen(resp_payload));
            } else {
                adi_send_packet(ssl, P2P_ERROR, "Invalid job submit
payload.", 27);
            }
            free(payload_copy);
            break;
        }
        case P2P_JOB_SUBMIT_RESP:
            printf("\n[Remote Job Resp] %s: %s\n> ", peer_uuid,
(char*)payload_buf);
            fflush(stdout);
            break;
        case P2P_JOB_STATUS_REQ: {
            int job_id = atoi((char*)payload_buf);
            char* status_buf = malloc(2048);
            if(status_buf) {
                find_job_status(job_id, status_buf, 2048);
                adi_send_packet(ssl, P2P_JOB_STATUS_RESP,
status_buf, strlen(status_buf));
                free(status_buf);
            }
```

```
            break;
        }
        case P2P_JOB_STATUS_RESP:
            printf("\n[Remote Job Status] %s:\n%s\n> ", peer_uuid,
(char*)payload_buf);
            fflush(stdout);
            break;
        case P2P_ERROR:
            printf("\n[Error from %s]: %s\n> ", peer_uuid,
(char*)payload_buf);
            fflush(stdout);
            break;
        default:
            log_warn("Received unknown packet type %d from %s",
type, peer_uuid);
      }
  }
  free(payload_buf);
  remove_peer_connection(slot);
  return NULL;
}
static void* p2p_listener_thread(void* arg) {
  (void)arg;
  log_info("P2P Listener thread started on port %s", g_p2p_port);
  SSL_CTX* ctx = create_ssl_context(g_ca_cert, g_client_cert,
g_client_key, 1);
  if (!ctx) {
      log_error("P2P Listener: Failed to create SSL context. Exiting
thread.");
      return NULL;
  }
  int server_sock = socket(AF_INET, SOCK_STREAM, 0);
  if (server_sock < 0) {
      log_error("P2P Listener: socket() failed.");
      SSL_CTX_free(ctx);
      return NULL;
  }
  int opt = 1;
  setsockopt(server_sock, SOL_SOCKET, SO_REUSEADDR, &opt,
sizeof(opt));
  struct sockaddr_in sa = {0};
  sa.sin_family = AF_INET;
  sa.sin_port = htons(atoi(g_p2p_port));
  sa.sin_addr.s_addr = INADDR_ANY;
  if (bind(server_sock, (struct sockaddr*)&sa, sizeof(sa)) < 0) {
      log_error("P2P Listener: bind() failed on port %s.",
g_p2p_port);
      close(server_sock);
```

```
        SSL_CTX_free(ctx);
        return NULL;
    }
    if (listen(server_sock, 10) < 0) {
        log_error("P2P Listener: listen() failed.");
        close(server_sock);
        SSL_CTX_free(ctx);
        return NULL;
    }
    while (!g_stop_client) {
        struct sockaddr_in client_sa;
        socklen_t client_len = sizeof(client_sa);
        int client_sock = accept(server_sock, (struct
sockaddr*)&client_sa, &client_len);
        if (client_sock < 0) {
            if (g_stop_client) break;
            log_warn("P2P Listener: accept() failed.");
            continue;
        }
        SSL* ssl = SSL_new(ctx);
        SSL_set_fd(ssl, client_sock);
        if (SSL_accept(ssl) <= 0) {
            log_warn("P2P Listener: SSL_accept() failed.");
            ERR_print_errors_fp(stderr);
            SSL_free(ssl);
            close(client_sock);
            continue;
        }
        char peer_uuid[64];
        if (!get_peer_uuid_from_ssl(ssl, peer_uuid, sizeof(peer_uuid)))
{
            log_warn("P2P Listener: Peer provided invalid/missing cert.
Rejecting.");
            SSL_free(ssl);
            close(client_sock);
            continue;
        }
        if (is_peer_blocked(peer_uuid)) {
            log_warn("P2P Listener: Rejected blocked peer %s",
peer_uuid);
            SSL_free(ssl);
            close(client_sock);
            continue;
        }
        log_info("P2P Listener: Accepted connection from %s",
peer_uuid);
        int slot = add_peer_connection(ssl, peer_uuid);
        if (slot == -1) {
```

```
            log_warn("P2P Listener: Connection pool full. Rejecting %s",
peer_uuid);
            SSL_free(ssl);
            close(client_sock);
            continue;
        }
        int* thread_arg = malloc(sizeof(int));
        *thread_arg = slot;
        pthread_t handler_thread;
        if (pthread_create(&handler_thread, NULL,
handle_peer_client_thread, thread_arg) != 0) {
            log_error("Failed to create handler thread for peer %s",
peer_uuid);
            remove_peer_connection(slot);
        } else {
            pthread_detach(handler_thread);
        }
    }
    log_info("P2P Listener thread shutting down.");
    close(server_sock);
    SSL_CTX_free(ctx);
    return NULL;
}
static void c2_event_handler(struct mg_connection* c, int ev, void*
ev_data, void* fn_data) {
    if (ev == MG_EV_ERROR) {
        log_error("C2 Connection Error: %s", (char*)ev_data);
        g_stop_client = 1;
    } else if (ev == MG_EV_WS_OPEN) {
        log_info("Connected to C2 Bootstrap Server.");
    } else if (ev == MG_EV_WS_MSG) {
        struct mg_ws_message* wm = (struct mg_ws_message*)ev_data;
        cJSON* json = cJSON_ParseWithLength(wm->data.ptr, wm->data.len);
        if (!json) {
            log_warn("Received invalid JSON from C2: %.*s",
(int)wm->data.len, wm->data.ptr);
            return;
        }
        cJSON* peers_array = cJSON_GetObjectItem(json, "peers");
        if (cJSON_IsArray(peers_array)) {
            pthread_mutex_lock(&g_peer_list_mutex);
            g_peer_count = 0;
            cJSON* peer_obj;
            cJSON_ArrayForEach(peer_obj, peers_array) {
                if (g_peer_count >= MAX_PEERS) break;
                cJSON* uuid = cJSON_GetObjectItem(peer_obj, "uuid");
                cJSON* addr = cJSON_GetObjectItem(peer_obj, "addr");
                if (cJSON_IsString(uuid) && cJSON_IsString(addr)) {
```

```
                    if (strcmp(uuid->valuestring, g_profile_uuid) != 0)
{
                        PeerInfo* p = &g_peer_list[g_peer_count];
                        strncpy(p->uuid, uuid->valuestring,
sizeof(p->uuid) - 1);
                        strncpy(p->addr, addr->valuestring,
sizeof(p->addr) - 1);
                        g_peer_count++;
                    }
                }
            }
            pthread_mutex_unlock(&g_peer_list_mutex);
        }
        cJSON_Delete(json);
    } else if (ev == MG_EV_CLOSE) {
        log_warn("Disconnected from C2 Bootstrap Server.
Reconnecting...");
    }
    (void)fn_data;
}
static void* c2_connect_thread(void* arg) {
    (void)arg;
    struct mg_mgr mgr;
    mg_mgr_init(&mgr);
    struct mg_tls_opts tls_opts = {
        .ca = g_ca_cert,
        .cert = g_client_cert,
        .key = g_client_key
    };
    char header_buf[64];
    snprintf(header_buf, sizeof(header_buf), "X-Peer-Port: %s\r\n",
g_p2p_port);
    struct mg_connect_opts opts = {
        .tls_opts = tls_opts,
        .extra_headers = header_buf
    };
    char c2_url[128];
    get_config_value(g_config_file, "c2_server", c2_url,
sizeof(c2_url));
    log_info("Connecting to C2 server at %s", c2_url);
    mg_ws_connect(&mgr, c2_url, c2_event_handler, NULL, &opts);
    while (!g_stop_client) {
        mg_mgr_poll(&mgr, 1000);
    }
    mg_mgr_free(&mgr);
    log_info("C2 connection thread stopped.");
    return NULL;
}
```

```
void print_help() {
  printf("\n--- [ Yukki OS 3.2 Help ] ---\n");
  printf("Compliance Commands:\n");
  printf("  profile               - View your UUID and blocklist.\n");
  printf("  block <uuid>          - Block all P2P from a peer.\n");
  printf("  unblock <uuid>        - Unblock a peer.\n");
  printf("\nLocal Build Commands (JobbySlotty):\n");
  printf("  job submit \"<name>\" \"<cmd>\" [deps:id1,id2] - Submit
local job with optional deps.\n");
  printf("  job queue             - View the local build job
queue.\n");
  printf("  job status <id>       - Check status of a local job.\n");
  printf("\nRemote Build Commands (JobbySlotty):\n");
  printf("  rjob submit <uuid> \"<name>\" \"<cmd>\" [deps:id1,id2] -
Submit job to a remote peer.\n");
  printf("  rjob status <uuid> <id> - Check status of a job on a
remote peer.\n");
  printf("\nP2P Commands (ADI Protocol):\n");
  printf("  peers                 - List peers visible on the
network.\n");
  printf("  msg <uuid> <message>  - Send a secure message to a
peer.\n");
  printf("  say <message>         - [Chat] Broadcast message to all
known peers.\n");
  printf("  join <room>           - [Chat] (Alias for 'peers', chat is
global).\n");
  printf("  send <uuid> <path>    - Securely send a local file to a
peer.\n");
  printf("  get <uuid> <filename> - Request a file from a peer (must
be in their share dir).\n");
  printf("  ls <uuid>             - List files in a peer's share
directory.\n");
  printf("\nOther Commands:\n");
  printf("  help                  - Show this message.\n");
  printf("  quit, exit            - Shut down the client.\n\n");
}
int main(int argc, char* argv[]) {
  if (argc != 2) {
      fprintf(stderr, "Usage: %s <profile_name>\n", argv[0]);
      return 1;
  }
  strncpy(g_profile_name, argv[1], sizeof(g_profile_name) - 1);
  snprintf(g_config_file, sizeof(g_config_file), "%s%s.conf",
CONFIG_FILE_PREFIX, g_profile_name);
  mkdir("logs", 0755);
  mkdir("client_data", 0755);
  mkdir(RECEIVED_FILES_DIR, 0755);
  if (!get_config_value(g_config_file, "uuid", g_profile_uuid,
```

```
sizeof(g_profile_uuid)) ||
      !get_config_value(g_config_file, "p2p_port", g_p2p_port,
sizeof(g_p2p_port)) ||
      !get_config_value(g_config_file, "ca_cert", g_ca_cert,
sizeof(g_ca_cert)) ||
      !get_config_value(g_config_file, "client_cert", g_client_cert,
sizeof(g_client_cert)) ||
      !get_config_value(g_config_file, "client_key", g_client_key)) {
      fprintf(stderr, "Error: Failed to load configuration from
'%s'.\n", g_config_file);
      fprintf(stderr, "Please run './bin/yukki_configurator.sh' to
create the profile.\n");
      return 1;
  }
  // --- [3.2 MOD] Load the fancy_prompt setting ---
  char fancy_prompt_str[10];
  if (get_config_value(g_config_file, "fancy_prompt",
fancy_prompt_str, sizeof(fancy_prompt_str))) {
      if (strcmp(fancy_prompt_str, "yes") == 0) {
          g_fancy_prompt = 1;
      }
  }
  // --- [3.2 END] ---
  log_info("Client starting with profile: %s (UUID: %s)",
g_profile_name, g_profile_uuid);
  if (!get_user_consent(g_profile_uuid)) {
      printf("Consent not given. Exiting.\n");
      return 1;
  }
  // NEW: Init linenoise
  linenoiseSetCompletionCallback(completion);
  linenoiseSetMultiLine(0);
  linenoiseHistorySetMaxLen(100);
  pthread_t build_thread;
  if (pthread_create(&build_thread, NULL, build_worker_thread, NULL)
!= 0) {
      log_error("Failed to start build worker thread. Exiting.");
      return 1;
  }
  pthread_t c2_thread;
  if (pthread_create(&c2_thread, NULL, c2_connect_thread, NULL) != 0)
{
      log_error("Failed to start C2 connection thread. Exiting.");
      g_stop_client = 1;
      pthread_join(build_thread, NULL);
      return 1;
  }
  pthread_t p2p_thread;
```

```
  if (pthread_create(&p2p_thread, NULL, p2p_listener_thread, NULL) !=
0) {
      log_error("Failed to start P2P listener thread. Exiting.");
      g_stop_client = 1;
      pthread_join(c2_thread, NULL);
      pthread_join(build_thread, NULL);
      return 1;
  }
  printf("Yukki OS Client 3.2 (Final Integrated Edition) started. Type
'help' for commands.\n");
  char* line_buf;
  // --- [3.2 MOD] Add buffers for fancy prompt ---
  char prompt_buf[256];
  char time_buf[16];
  // --- [3.2 END] ---
  while (!g_stop_client) {
      // --- [3.2 MOD] Build prompt dynamically ---
      char* current_prompt = "> "; // Default simple prompt
      if (g_fancy_prompt) {
          time_t now = time(NULL);
          struct tm* t = localtime(&now);
          strftime(time_buf, sizeof(time_buf), "%H:%M:%S", t);
          snprintf(prompt_buf, sizeof(prompt_buf),
                   "\x1B[36m[%s]\x1B[0m \x1B[33m%s\x1B[0m
\x1B[32m✔\x1B[0m > ",
                   time_buf, g_profile_name);
          current_prompt = prompt_buf;
      }
      line_buf = linenoise(current_prompt);
      // --- [3.2 END] ---
      if (line_buf == NULL) break; // Ctrl-D or error
      if (strlen(line_buf) == 0) {
          linenoiseFree(line_buf);
          continue;
      }
      linenoiseHistoryAdd(line_buf);
      if (strlen(line_buf) == 0) continue;
      char* cmd_line_copy = strdup(line_buf);
      char* cmd = strtok(cmd_line_copy, " ");
      char* args = strtok(NULL, "");
      if (!cmd) {
          free(cmd_line_copy);
          continue;
      }
      log_debug("User command: %s %s", cmd, args ? args : "");
      if (strcmp(cmd, "quit") == 0 || strcmp(cmd, "exit") == 0) {
          free(cmd_line_copy);
          break;
```

```
      } else if (strcmp(cmd, "help") == 0) {
          print_help();
      } else if (strcmp(cmd, "profile") == 0) {
          show_profile(g_profile_uuid);
      } else if (strcmp(cmd, "block") == 0) {
          if(args) block_peer(args);
          else printf("Usage: block <uuid>\n");
      } else if (strcmp(cmd, "unblock") == 0) {
          if(args) unblock_peer(args);
          else printf("Usage: unblock <uuid>\n");
      } else if (strcmp(cmd, "job") == 0) {
          if (!args) { printf("Usage: job <submit|queue|status>\n");
free(cmd_line_copy); continue; }
          char* sub_cmd = strtok(args, " ");
          char* sub_args = strtok(NULL, "");
          if (strcmp(sub_cmd, "submit") == 0) {
              if (!sub_args) { printf("Usage: job submit \"<name>\"
\"<command>\" [deps:id1,id2]\n"); free(cmd_line_copy); continue; }
              char* name = strtok(sub_args, "\"");
              if (!name) { printf("Usage: job submit \"<name>\"
\"<command>\" [deps:id1,id2]\n"); free(cmd_line_copy); continue; }
              char* cmd_part = strtok(NULL, "\"");
              if (cmd_part) {
                  cmd_part = strtok(NULL, "\"");
                  if (cmd_part) {
                      int deps[MAX_JOB_DEPS];
                      int dep_count = 0;
                      char* deps_str = strstr(cmd_part, "deps:");
                      if (deps_str) {
                          *(deps_str - 1) = '\0';
                          deps_str += 5;
                          char* dep_id_str = strtok(deps_str, ",");
                          while (dep_id_str && dep_count <
MAX_JOB_DEPS) {
                              deps[dep_count++] = atoi(dep_id_str);
                              dep_id_str = strtok(NULL, ",");
                          }
                      }
                      submit_job(name, cmd_part, deps, dep_count);
                  } else printf("Usage: job submit \"<name>\"
\"<command>\" [deps:id1,id2]\n");
              } else printf("Usage: job submit \"<name>\"
\"<command>\" [deps:id1,id2]\n");
          } else if (strcmp(sub_cmd, "queue") == 0) {
              show_job_queue();
          } else if (strcmp(sub_cmd, "status") == 0) {
              if (sub_args) {
                  char status_buf[2048];
```

```
                    find_job_status(atoi(sub_args), status_buf,
sizeof(status_buf));
                    printf("- %s\n", status_buf);
                } else printf("Usage: job status <id>\n");
            } else {
                printf("Usage: job <submit|queue|status>\n");
            }
        } else if (strcmp(cmd, "rjob") == 0) {
            if (!args) { printf("Usage: rjob <submit|status> ...\n");
free(cmd_line_copy); continue; }
            char* sub_cmd = strtok(args, " ");
            char* sub_args = strtok(NULL, "");
            if (!sub_args) { printf("Usage: rjob %s ...\n", sub_cmd);
free(cmd_line_copy); continue; }
            char* peer_uuid = strtok(sub_args, " ");
            char* rest_args = strtok(NULL, "");
            if (!peer_uuid) { printf("Usage: rjob %s <uuid> ...\n",
sub_cmd); free(cmd_line_copy); continue; }
            SSL* ssl = connect_to_peer(peer_uuid);
            if (!ssl) {
                printf("Error: Could not connect to peer %s.\n",
peer_uuid);
                free(cmd_line_copy);
                continue;
            }
            if (strcmp(sub_cmd, "submit") == 0) {
                if (!rest_args) { printf("Usage: rjob submit <uuid>
\"<name>\" \"<command>\"\n"); free(cmd_line_copy); continue; }
                char* name = strtok(rest_args, "\"");
                if (!name) { printf("Usage: rjob submit <uuid>
\"<name>\" \"<command>\"\n"); free(cmd_line_copy); continue; }
                char* cmd_part = strtok(NULL, "\"");
                if (!cmd_part) { printf("Usage: rjob submit <uuid>
\"<name>\" \"<command>\"\n"); free(cmd_line_copy); continue; }
                cmd_part = strtok(NULL, "\"");
                if (!cmd_part) { printf("Usage: rjob submit <uuid>
\"<name>\" \"<command>\"\n"); free(cmd_line_copy); continue; }
                char payload[2048];
                char* deps_str = strstr(cmd_part, "deps:");
                char deps_only_str[256] = {0};
                if (deps_str) {
                    *(deps_str - 1) = '\0';
                    deps_str += 5;
                    strncpy(deps_only_str, deps_str,
sizeof(deps_only_str) - 1);
                }
                snprintf(payload, sizeof(payload), "%s\n%s\n%s", name,
cmd_part, deps_only_str);
```

```
            if (adi_send_packet(ssl, P2P_JOB_SUBMIT_REQ, payload,
strlen(payload)) == 0) {
                printf("Remote job submitted to %s.\n", peer_uuid);
            } else {
                printf("Failed to submit remote job to %s.\n",
peer_uuid);
            }
        } else if (strcmp(sub_cmd, "status") == 0) {
            char* job_id_str = rest_args;
            if (job_id_str) {
                adi_send_packet(ssl, P2P_JOB_STATUS_REQ, job_id_str,
strlen(job_id_str));
            } else {
                printf("Usage: rjob status <uuid> <job_id>\n");
            }
        } else {
            printf("Unknown rjob command: %s. Use 'submit' or
'status'.\n", sub_cmd);
        }
    } else if (strcmp(cmd, "peers") == 0) {
        printf("\n--- [ Visible Peers ] ---\n");
        pthread_mutex_lock(&g_peer_list_mutex);
        if (g_peer_count == 0) {
            printf("(No other peers visible on network)\n");
        } else {
            for (int i = 0; i < g_peer_count; i++) {
                printf("- %s (%s)\n", g_peer_list[i].uuid,
g_peer_list[i].addr);
            }
        }
        pthread_mutex_unlock(&g_peer_list_mutex);
        printf("\n");
    } else if (strcmp(cmd, "join") == 0) {
        printf("[Chat] Joined global broadcast. Use 'say <msg>' to
talk.\n");
    } else if (strcmp(cmd, "say") == 0) {
        if (!args) { printf("Usage: say <message>\n");
free(cmd_line_copy); continue; }
        printf("[Broadcasting to all peers]...\n");
        pthread_mutex_lock(&g_peer_list_mutex);
        for (int i = 0; i < g_peer_count; i++) {
            if(is_peer_blocked(g_peer_list[i].uuid)) continue;
            SSL* ssl = connect_to_peer(g_peer_list[i].uuid);
            if(ssl) {
                adi_send_packet(ssl, P2P_MSG_CMD, args,
strlen(args));
            }
        }
```

```
            pthread_mutex_unlock(&g_peer_list_mutex);
        } else if (strcmp(cmd, "msg") == 0) {
            char* peer_uuid = strtok(args, " ");
            char* msg = strtok(NULL, "");
            if(peer_uuid && msg) {
                SSL* ssl = connect_to_peer(peer_uuid);
                if (ssl) {
                    if (adi_send_packet(ssl, P2P_MSG_CMD, msg,
strlen(msg)) == 0) {
                        printf("Message sent to %s.\n", peer_uuid);
                    } else {
                        printf("Failed to send message to %s.\n",
peer_uuid);
                    }
                }
            } else { printf("Usage: msg <uuid> <message>\n"); }
        } else if (strcmp(cmd, "send") == 0) {
            char* peer_uuid = strtok(args, " ");
            char* local_path = strtok(NULL, "");
            if (!peer_uuid || !local_path) {
                printf("Usage: send <uuid> <localpath>\n");
                free(cmd_line_copy);
                continue;
            }
            SSL* ssl = connect_to_peer(peer_uuid);
            if (!ssl) { free(cmd_line_copy); continue; }
            FILE* f = fopen(local_path, "rb");
            if (!f) {
                perror("Error: Failed to open local file");
                free(cmd_line_copy);
                continue;
            }
            const char* filename = strrchr(local_path, '/');
            filename = (filename) ? filename + 1 : local_path;
            printf("Sending file %s to %s as '%s'...\n", local_path,
peer_uuid, filename);
            if (adi_send_packet(ssl, P2P_SEND_REQ, filename,
strlen(filename)) != 0) {
                printf("Error: Failed to send file request.\n");
                fclose(f);
                free(cmd_line_copy);
                continue;
            }
            uint8_t* file_chunk_buf = malloc(FILE_CHUNK_SIZE);
            if (!file_chunk_buf) {
                log_error("Failed to allocate file chunk buffer for
send");
                fclose(f);
```

```
            free(cmd_line_copy);
            continue;
        }
        size_t bytes_read;
        while ((bytes_read = fread(file_chunk_buf, 1,
FILE_CHUNK_SIZE, f)) > 0) {
            if (adi_send_packet(ssl, P2P_FILE_CHUNK, file_chunk_buf,
bytes_read) != 0) {
                printf("Error: Connection lost during file
transfer.\n");
                break;
            }
        }
        free(file_chunk_buf);
        adi_send_packet(ssl, P2P_FILE_END, NULL, 0);
        fclose(f);
        printf("File transfer complete.\n");
    } else if (strcmp(cmd, "get") == 0) {
        char* peer_uuid = strtok(args, " ");
        char* filename = strtok(NULL, "");
        if(peer_uuid && filename) {
            SSL* ssl = connect_to_peer(peer_uuid);
            if (ssl) {
                printf("Requesting file '%s' from %s...\n",
filename, peer_uuid);
                adi_send_packet(ssl, P2P_GET_REQ, filename,
strlen(filename));
            }
        } else { printf("Usage: get <uuid> <filename>\n"); }
    } else if (strcmp(cmd, "ls") == 0) {
        if (args) {
            SSL* ssl = connect_to_peer(args);
            if (ssl) {
                printf("Requesting file list from %s...\n", args);
                adi_send_packet(ssl, P2P_LS_REQ, NULL, 0);
            }
        } else { printf("Usage: ls <uuid>\n"); }
    } else {
        printf("Unknown command: '%s'. Type 'help' for a list.\n",
cmd);
    }
    free(cmd_line_copy);
    linenoiseFree(line_buf); // MODIFIED: for linenoise
  }
  // if (line_buf) free(line_buf); // MODIFIED: Handled by
linenoiseFree in loop
  log_info("Shutting down client...");
  g_stop_client = 1;
```

```
  pthread_cond_broadcast(&job_queue_cond);
  pthread_join(build_thread, NULL);
  pthread_join(c2_thread, NULL);
  int sock = socket(AF_INET, SOCK_STREAM, 0);
  struct sockaddr_in sa = {0};
  sa.sin_family = AF_INET;
  sa.sin_port = htons(atoi(g_p2p_port));
  sa.sin_addr.s_addr = inet_addr("127.0.0.1");
  if(connect(sock, (struct sockaddr*)&sa, sizeof(sa)) == 0) {
      close(sock);
  }
  pthread_join(p2p_thread, NULL);
  for (int i = 0; i < MAX_PEERS; i++) {
      remove_peer_connection(i);
  }
  linenoiseHistoryFree(); // NEW: Clean up history
  log_info("Client shut down complete.");
  printf("Goodbye.\n");
  return 0;
}
EOF
cat > "$R/yukki_c2_suite/peer_client/Makefile" <<'EOF'
CC = gcc
CFLAGS = -I"$(CURDIR)/../../vendor" -Wall -Wextra -g -pthread -std=c11
LDFLAGS = -lssl -lcrypto -pthread -lm
VENDOR_SRC = $(wildcard $(CURDIR)/../../vendor/*.c)
VENDOR_OBJ = $(VENDOR_SRC:.c=.o)
TARGET = peer_client
all: $(TARGET)
$(TARGET): client.o $(VENDOR_OBJ)
       $(CC) $(CFLAGS) -o $(TARGET) client.o $(VENDOR_OBJ) $(LDFLAGS)
client.o: client.c
       $(CC) $(CFLAGS) -c client.c -o client.o
$(CURDIR)/../../vendor/%.o: $(CURDIR)/../../vendor/%.c
       $(CC) $(CFLAGS) -c $< -o $@
clean:
       rm -f $(TARGET) *.o $(VENDOR_OBJ)
EOF
};f(){ cat > "$R/bin/yukki_configurator.sh" <<'EOF'
#!/bin/bash
set -eu
ROOT_DIR=$(cd -- "$(dirname -- "${BASH_SOURCE[0]}")" &> /dev/null &&
cd .. && pwd)
PKI_DIR="$ROOT_DIR/pki"
DATA_DIR="$ROOT_DIR/yukki_c2_suite/peer_client/client_data"
echo "--- Yukki OS 3.2 PKI & Profile Configurator ---"
echo "This will create a new CA, Server Cert, and Client Profile."
echo "PKI Root: $PKI_DIR"
```

```
echo "Client Data: $DATA_DIR"
echo ""
mkdir -p "$PKI_DIR"
mkdir -p "$DATA_DIR"
if [ -f "$PKI_DIR/ca_key.pem" ];
then
  echo "[SKIP] CA key '$PKI_DIR/ca_key.pem' already exists."
else
  echo "[INFO] Generating new Certificate Authority (CA)..."
  openssl genrsa -out "$PKI_DIR/ca_key.pem" 4096
  openssl req -new -x509 -key "$PKI_DIR/ca_key.pem" -sha256 -days 3650
\
            -out "$PKI_DIR/ca_cert.pem" \
            -subj
"/C=CA/ST=Ontario/L=Toronto/O=YukkiNetwork/CN=YukkiRootCA"
  echo "[INFO] New CA Certificate created: $PKI_DIR/ca_cert.pem"
fi
if [ -f "$PKI_DIR/server_key.pem" ];
then
  echo "[SKIP] Server key '$PKI_DIR/server_key.pem' already exists."
else
  echo "[INFO] Generating new Bootstrap Server Certificate..."
  openssl genrsa -out "$PKI_DIR/server_key.pem" 2048
  openssl req -new -key "$PKI_DIR/server_key.pem" \
            -out "$PKI_DIR/server_csr.pem" \
            -subj
"/C=CA/ST=Ontario/L=Toronto/O=YukkiNetwork/CN=yukki-c2-server.local"
  openssl x509 -req -in "$PKI_DIR/server_csr.pem" -CA
"$PKI_DIR/ca_cert.pem" \
            -CAkey "$PKI_DIR/ca_key.pem" -CAcreateserial \
            -out "$PKI_DIR/server_cert.pem" -days 365 -sha256
  rm "$PKI_DIR/server_csr.pem"
  echo "[INFO] New Server Certificate created:
$PKI_DIR/server_cert.pem"
fi
echo ""
read -p "Enter a name for the new client profile (e.g., 'user_alice'):
" PROFILE_NAME
if [ -z "$PROFILE_NAME" ];
then
  echo "Error: Profile name cannot be empty."
  exit 1
fi
CLIENT_CONF="$DATA_DIR/$PROFILE_NAME.conf"
CLIENT_KEY="$PKI_DIR/${PROFILE_NAME}_key.pem"
CLIENT_CERT="$PKI_DIR/${PROFILE_NAME}_cert.pem"
if [ -f "$CLIENT_CONF" ];
then
```

```
  echo "Error: Profile '$PROFILE_NAME' already exists ($CLIENT_CONF)."
  exit 1
fi
echo "[INFO] Generating new Client Certificate for '$PROFILE_NAME'..."
CLIENT_UUID=$(uuidgen)
openssl genrsa -out "$CLIENT_KEY" 2048
openssl req -new -key "$CLIENT_KEY" \
        -out "$PKI_DIR/client_csr.pem" \
        -subj
"/C=CA/ST=Ontario/L=Toronto/O=YukkiNetwork/CN=$CLIENT_UUID"
openssl x509 -req -in "$PKI_DIR/client_csr.pem" -CA
"$PKI_DIR/ca_cert.pem" \
        -CAkey "$PKI_DIR/ca_key.pem" -CAcreateserial \
        -out "$CLIENT_CERT" -days 365 -sha256
rm "$PKI_DIR/client_csr.pem"
echo "[INFO] New Client Certificate created: $CLIENT_CERT"
echo "[INFO] Client UUID (embedded in cert): $CLIENT_UUID"
echo ""
read -p "Enter C2 Server URL (e.g., wss://127.0.0.1:8443): "
C2_SERVER_URL
read -p "Enter this client's P2P Listen Port (e.g., 9001): " P2P_PORT
# --- [3.2 MOD] Add prompt for fancy UI ---
read -p "Enable enhanced visual prompt? (yes/no) [default: no]: "
FANCY_PROMPT
# --- [3.2 END] ---
if [ -z "$C2_SERVER_URL" ] || [ -z "$P2P_PORT" ]; then
  echo "Error: C2 Server URL and P2P Port are required."
  exit 1
fi
echo "[INFO] Writing client config file: $CLIENT_CONF"
cat > "$CLIENT_CONF" << EOL
# Yukki Client Profile: $PROFILE_NAME
uuid=$CLIENT_UUID
c2_server=$C2_SERVER_URL
p2p_port=$P2P_PORT
# PKI Paths (relative to client binary)
ca_cert=pki/ca_cert.pem
client_cert=pki/${PROFILE_NAME}_cert.pem
client_key=pki/${PROFILE_NAME}_key.pem
# --- [3.2 MOD] Add prompt setting to config ---
# UI Settings
fancy_prompt=${FANCY_PROMPT:-no}
# --- [3.2 END] ---
EOL
cd "$R/yukki_c2_suite/peer_client"
mkdir -p pki
cd pki
ln -sf "../../$PKI_DIR/ca_cert.pem" ca_cert.pem
```

```
ln -sf "../../$CLIENT_CERT" "${PROFILE_NAME}_cert.pem"
ln -sf "../../$CLIENT_KEY" "${PROFILE_NAME}_key.pem"
cd "$R"
echo ""
echo "--- Profile '$PROFILE_NAME' Created Successfully ---"
echo "Config: $CLIENT_CONF"
echo "Run client from $R with:"
echo " > ./yukki_c2_suite/peer_client/peer_client $PROFILE_NAME"
echo ""
chmod 600 "$PKI_DIR"/*_key.pem
chmod 644 "$PKI_DIR"/*_cert.pem
chmod 600 "$CLIENT_CONF"
EOF
chmod +x "$R/bin/yukki_configurator.sh"
};a;b;c;d;e;f
echo "============================================================"
echo " Yukki OS 3.2 (Final Integrated) Generation Complete"
echo "============================================================"
echo ""
echo "Project root: $R"
echo ""
echo "CRTC and PIPEDA compliance documentation has been added to:
$R/docs/"
echo "Please review these documents carefully."
echo ""
echo "--- NEXT STEPS ---"
echo "1. Install Dependencies:"
echo " > sudo apt-get update && sudo apt-get install build-essential
libssl-dev uuid-dev"
echo ""
echo "2. Configure Server & Create Client Profiles:"
echo " > cd '$R'"
echo " > ./bin/yukki_configurator.sh"
echo "   (Run this multiple times to create different client profiles,
e.g., 'alice' and 'bob')"
echo "   (You will be prompted to enable the enhanced visual prompt)"
echo ""
echo "3. Compile All Components:"
echo " > make -C '$R/yukki_c2_suite/bootstrap_server'"
echo " > make -C '$R/yukki_c2_suite/peer_client'"
echo ""
echo "4. Run the Network (from $R):"
echo " - Terminal 1 (Bootstrap Server):"
echo " > ./yukki_c2_suite/bootstrap_server/bootstrap_server"
echo " - Terminal 2 (Peer 'alice'):"
echo " > ./yukki_c2_suite/peer_client/peer_client alice"
echo " - Terminal 3 (Peer 'bob'):"
echo " > ./yukki_c2_suite/peer_client/peer_client bob"
```

```
echo " (You will be prompted to provide consent for each client)"
echo ""
echo "5. Key Commands in Client (type 'help' for all):"
echo " > peers                 # See who is online"
echo " > msg <uuid> <message>  # Send a private message"
echo " > send <uuid> <path>    # Send a file to a peer"
echo " > job submit \"MyBuild\" \"make\" [deps:1,2] # Run a local
build"
echo " > job status <id>       # Check a local build"
echo " > rjob submit <uuid> \"RemoteBuild\" \"./run.sh\" # Run a
remote build"
echo " > rjob status <uuid> <job_id> # Check a remote build"
echo ""
}
```