```
//
// Copyright Aliaksei Levin (levlam@telegram.org), Arseny Smirnov (arseny30@gmail.com) 2014-2020
//
// Distributed under the Boost Software License, Version 1.0. (See accompanying
// file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)
//
#pragma once

#include "td/telegram/logevent/LogEvent.h"

#include "td/actor/PromiseFuture.h"

#include "td/utils/buffer.h"
#include "td/utils/common.h"
#include "td/utils/format.h"
#include "td/utils/StringBuilder.h"
#include "td/utils/tl_helpers.h"

#include "td/telegram/secret_api.h"
#include "td/telegram/telegram_api.h"

namespace td {
namespace logevent {

class SecretChatEvent : public LogEventBase<SecretChatEvent> {
 public:
  // append only enum
  enum class Type : int32 {
    InboundSecretMessage = 1,
    OutboundSecretMessage = 2,
    CloseSecretChat = 3,
    CreateSecretChat = 4
  };

  virtual Type get_type() const = 0;

  static constexpr LogEvent::HandlerType get_handler_type() {
    return LogEvent::HandlerType::SecretChats;
  }

  static constexpr int32 version() {
    return 2;
  }

  template <class F>
  static void downcast_call(Type type, F &&f);
};

template <class ChildT>
class SecretChatLogEventBase : public SecretChatEvent {
 public:
  typename SecretChatEvent::Type get_type() const override {
    return ChildT::type;
  }

  constexpr int32 magic() const {
    return static_cast<int32>(get_type());
  }
};

// Internal structure

// inputEncryptedFileEmpty#1837c364 = InputEncryptedFile;
// inputEncryptedFileUploaded#64bd0306 id:long parts:int md5_checksum:string key_fingerprint:int = InputEncryptedFile;
// inputEncryptedFile#5a17b5e5 id:long access_hash:long = InputEncryptedFile;
// inputEncryptedFileBigUploaded#2dc173c8 id:long parts:int key_fingerprint:int = InputEncryptedFile;
struct EncryptedInputFile {
  static constexpr int32 MAGIC = 0x4328d38a;
  enum Type : int32 { Empty = 0, Uploaded = 1, BigUploaded = 2, Location = 3 } type = Type::Empty;
  int64 id = 0;
  int64 access_hash = 0;
  int32 parts = 0;
  int32 key_fingerprint = 0;

  template <class StorerT>
  void store(StorerT &storer) const {
    using td::store;
    store(MAGIC, storer);
    store(type, storer);
    store(id, storer);
    store(access_hash, storer);
    store(parts, storer);
    store(key_fingerprint, storer);
  }

  EncryptedInputFile() = default;
  EncryptedInputFile(Type type, int64 id, int64 access_hash, int32 parts, int32 key_fingerprint)
      : type(type), id(id), access_hash(access_hash), parts(parts), key_fingerprint(key_fingerprint) {
  }

  bool empty() const {
    return type == Empty;
  }

  template <class ParserT>
  void parse(ParserT &parser) {
    using td::parse;
    int32 got_magic;

    parse(got_magic, parser);
    parse(type, parser);
    parse(id, parser);
    parse(access_hash, parser);
    parse(parts, parser);
    parse(key_fingerprint, parser);

    if (got_magic != MAGIC) {
      parser.set_error("EncryptedInputFile magic mismatch");
      return;
    }
  }
  static EncryptedInputFile from_input_encrypted_file(const tl_object_ptr<telegram_api::InputEncryptedFile> &from) {
    if (!from) {
      return EncryptedInputFile{Empty, 0, 0, 0, 0};
    }
    return from_input_encrypted_file(*from);
  }
  static EncryptedInputFile from_input_encrypted_file(const telegram_api::InputEncryptedFile &from) {
    switch (from.get_id()) {
      case telegram_api::inputEncryptedFileEmpty::ID:
        return EncryptedInputFile{Empty, 0, 0, 0, 0};
      case telegram_api::inputEncryptedFileUploaded::ID: {
        auto &uploaded = static_cast<const telegram_api::inputEncryptedFileUploaded &>(from);
        return EncryptedInputFile{Uploaded, uploaded.id_, 0, uploaded.parts_, uploaded.key_fingerprint_};
      }
```

```
      case telegram_api::inputEncryptedFileBigUploaded::ID: {
        auto &uploaded = static_cast<const telegram_api::inputEncryptedFileBigUploaded &>(from);
        return EncryptedInputFile{BigUploaded, uploaded.id_, 0, uploaded.parts_, uploaded.key_fingerprint_};
      }
      case telegram_api::inputEncryptedFile::ID: {
        auto &uploaded = static_cast<const telegram_api::inputEncryptedFile &>(from);
        return EncryptedInputFile{Location, uploaded.id_, uploaded.access_hash_, 0, 0};
      }
      default:
        UNREACHABLE();
    }
  }
  tl_object_ptr<telegram_api::InputEncryptedFile> as_input_encrypted_file() const {
    switch (type) {
      case Empty:
        return make_tl_object<telegram_api::inputEncryptedFileEmpty>();
      case Uploaded:
        return make_tl_object<telegram_api::inputEncryptedFileUploaded>(id, parts, "", key_fingerprint);
      case BigUploaded:
        return make_tl_object<telegram_api::inputEncryptedFileBigUploaded>(id, parts, key_fingerprint);
      case Location:
        return make_tl_object<telegram_api::inputEncryptedFile>(id, access_hash);
    }
    UNREACHABLE();
  }
};

inline StringBuilder &operator<<(StringBuilder &sb, const EncryptedInputFile &file) {
  return sb << to_string(file.as_input_encrypted_file());
}

// encryptedFile#4a70994c id:long access_hash:long size:int dc_id:int key_fingerprint:int = EncryptedFile;
struct EncryptedFileLocation {
  static constexpr int32 MAGIC = 0x473d738a;
  int64 id = 0;
  int64 access_hash = 0;
  int32 size = 0;
  int32 dc_id = 0;
  int32 key_fingerprint = 0;

  tl_object_ptr<telegram_api::encryptedFile> as_encrypted_file() {
    return make_tl_object<telegram_api::encryptedFile>(id, access_hash, size, dc_id, key_fingerprint);
  }
  template <class StorerT>
  void store(StorerT &storer) const {
    using td::store;
    store(MAGIC, storer);
    store(id, storer);
    store(access_hash, storer);
    store(size, storer);
    store(dc_id, storer);
    store(key_fingerprint, storer);
  }

  template <class ParserT>
  void parse(ParserT &parser) {
    using td::parse;
    int32 got_magic;

    parse(got_magic, parser);
    parse(id, parser);
    parse(access_hash, parser);
    parse(size, parser);
    parse(dc_id, parser);
    parse(key_fingerprint, parser);

    if (got_magic != MAGIC) {
      parser.set_error("EncryptedFileLocation magic mismatch");
      return;
    }
  }
};

inline StringBuilder &operator<<(StringBuilder &sb, const EncryptedFileLocation &file) {
  return sb << "[" << tag("id", file.id) << tag("access_hash", file.access_hash) << tag("size", file.size)
            << tag("dc_id", file.dc_id) << tag("key_fingerprint", file.key_fingerprint) << "]";
}

// LogEvents
// TODO: Qts and SeqNoState could be just Logevents that are updated during regenerate
class InboundSecretMessage : public SecretChatLogEventBase<InboundSecretMessage> {
 public:
  static constexpr Type type = SecretChatEvent::Type::InboundSecretMessage;
  int32 qts = 0;

  int32 chat_id = 0;
  int32 date = 0;

  BufferSlice encrypted_message;  // empty when we store event to binlog
  Promise<Unit> qts_ack;

  bool is_checked = false;
  // after decrypted and checked
  tl_object_ptr<secret_api::decryptedMessageLayer> decrypted_message_layer;

  uint64 auth_key_id = 0;
  int32 message_id = 0;
  int32 my_in_seq_no = -1;
  int32 my_out_seq_no = -1;
  int32 his_in_seq_no = -1;

  int32 his_layer() const {
    return decrypted_message_layer->layer_;
  }

  EncryptedFileLocation file;

  bool has_encrypted_file = false;
  bool is_pending = false;

  template <class StorerT>
  void store(StorerT &storer) const {
    using td::store;

    BEGIN_STORE_FLAGS();
    STORE_FLAG(has_encrypted_file);
    STORE_FLAG(is_pending);
    END_STORE_FLAGS();

    store(qts, storer);
    store(chat_id, storer);
    store(date, storer);
    // skip encrypted_message
    // skip qts_ack
```

```
    // TODO
    decrypted_message_layer->store(storer);
    storer.store_long(static_cast<int64>(auth_key_id));

    store(message_id, storer);
    store(my_in_seq_no, storer);
    store(my_out_seq_no, storer);
    store(his_in_seq_no, storer);
    if (has_encrypted_file) {
      store(file, storer);
    }
  }

  template <class ParserT>
  void parse(ParserT &parser) {
    using td::parse;

    BEGIN_PARSE_FLAGS();
    PARSE_FLAG(has_encrypted_file);
    PARSE_FLAG(is_pending);
    END_PARSE_FLAGS();

    parse(qts, parser);
    parse(chat_id, parser);
    parse(date, parser);
    // skip encrypted_message
    // skip qts_ack

    // TODO
    decrypted_message_layer = secret_api::decryptedMessageLayer::fetch(parser);
    auth_key_id = static_cast<uint64>(parser.fetch_long());

    parse(message_id, parser);
    parse(my_in_seq_no, parser);
    parse(my_out_seq_no, parser);
    parse(his_in_seq_no, parser);
    if (has_encrypted_file) {
      parse(file, parser);
    }

    is_checked = true;
  }

  StringBuilder &print(StringBuilder &sb) const override {
    return sb << "[Logevent InboundSecretMessage " << tag("id", logevent_id()) << tag("qts", qts)
              << tag("chat_id", chat_id) << tag("date", date) << tag("auth_key_id", format::as_hex(auth_key_id))
              << tag("message_id", message_id) << tag("my_in_seq_no", my_in_seq_no)
              << tag("my_out_seq_no", my_out_seq_no) << tag("his_in_seq_no", his_in_seq_no)
              << tag("message", to_string(decrypted_message_layer)) << tag("is_pending", is_pending)
              << format::cond(has_encrypted_file, tag("file", file)) << "]";
  }
};

class OutboundSecretMessage : public SecretChatLogEventBase<OutboundSecretMessage> {
 public:
  static constexpr Type type = SecretChatEvent::Type::OutboundSecretMessage;

  int32 chat_id = 0;
  int64 random_id = 0;

  BufferSlice encrypted_message;
  EncryptedInputFile file;

  int32 message_id = 0;
  int32 my_in_seq_no = -1;
  int32 my_out_seq_no = -1;
  int32 his_in_seq_no = -1;

  int32 his_layer() const {
    return -1;
  }

  bool is_sent = false;
  // need send push notification to the receiver
  // should send such messages with messages_sendEncryptedService
  bool need_notify_user = false;
  bool is_rewritable = false;
  // should notify our parent about state of this message (using context and random_id)
  bool is_external = false;

  tl_object_ptr<secret_api::DecryptedMessageAction> action;
  uint64 crc = 0;  // DEBUG;

  // Flags:
  // 2. can_fail = !file.empty() // send of other messages can't fail if chat is ok. It is usless to rewrite them with
  // empty
  // 3. can_rewrite_with_empty // false for almost all service messages

  // TODO: combine these two functions into one macros hell. Or a lambda hell.
  template <class StorerT>
  void store(StorerT &storer) const {
    using td::store;

    store(chat_id, storer);
    store(random_id, storer);
    store(encrypted_message, storer);
    store(file, storer);
    store(message_id, storer);
    store(my_in_seq_no, storer);
    store(my_out_seq_no, storer);
    store(his_in_seq_no, storer);

    bool has_action = static_cast<bool>(action);
    BEGIN_STORE_FLAGS();
    STORE_FLAG(is_sent);
    STORE_FLAG(need_notify_user);
    STORE_FLAG(has_action);
    STORE_FLAG(is_rewritable);
    STORE_FLAG(is_external);
    END_STORE_FLAGS();

    if (has_action) {
      CHECK(action);
      // TODO
      storer.store_int(action->get_id());
      action->store(storer);
    }
  }

  template <class ParserT>
  void parse(ParserT &parser) {
    using td::parse;

    parse(chat_id, parser);
    parse(random_id, parser);
```

```
    parse(encrypted_message, parser);
    parse(file, parser);
    parse(message_id, parser);
    parse(my_in_seq_no, parser);
    parse(my_out_seq_no, parser);
    parse(his_in_seq_no, parser);

    bool has_action;
    BEGIN_PARSE_FLAGS();
    PARSE_FLAG(is_sent);
    PARSE_FLAG(need_notify_user);
    PARSE_FLAG(has_action);
    PARSE_FLAG(is_rewritable);
    PARSE_FLAG(is_external);
    END_PARSE_FLAGS();

    if (has_action) {
      // TODO:
      action = secret_api::DecryptedMessageAction::fetch(parser);
    }
  }

  StringBuilder &print(StringBuilder &sb) const override {
    return sb << "[Logevent OutboundSecretMessage " << tag("id", logevent_id()) << tag("chat_id", chat_id)
              << tag("is_sent", is_sent) << tag("need_notify_user", need_notify_user)
              << tag("is_rewritable", is_rewritable) << tag("is_external", is_external) << tag("message_id", message_id)
              << tag("random_id", random_id) << tag("my_in_seq_no", my_in_seq_no) << tag("my_out_seq_no", my_out_seq_no)
              << tag("his_in_seq_no", his_in_seq_no) << tag("file", file) << tag("action", to_string(action)) << "]";
  }
};

class CloseSecretChat : public SecretChatLogEventBase<CloseSecretChat> {
 public:
  static constexpr Type type = SecretChatEvent::Type::CloseSecretChat;
  int32 chat_id = 0;

  template <class StorerT>
  void store(StorerT &storer) const {
    using td::store;
    store(chat_id, storer);
  }

  template <class ParserT>
  void parse(ParserT &parser) {
    using td::parse;
    parse(chat_id, parser);
  }

  StringBuilder &print(StringBuilder &sb) const override {
    return sb << "[Logevent CloseSecretChat " << tag("id", logevent_id()) << tag("chat_id", chat_id) << "]";
  }
};

class CreateSecretChat : public SecretChatLogEventBase<CreateSecretChat> {
 public:
  static constexpr Type type = SecretChatEvent::Type::CreateSecretChat;
  int32 random_id = 0;
  int32 user_id = 0;
  int64 user_access_hash = 0;

  template <class StorerT>
  void store(StorerT &storer) const {
    using td::store;
    store(random_id, storer);
    store(user_id, storer);
    store(user_access_hash, storer);
  }

  template <class ParserT>
  void parse(ParserT &parser) {
    using td::parse;
    parse(random_id, parser);
    parse(user_id, parser);
    parse(user_access_hash, parser);
  }

  StringBuilder &print(StringBuilder &sb) const override {
    return sb << "[Logevent CreateSecretChat " << tag("id", logevent_id()) << tag("chat_id", random_id)
              << tag("user_id", user_id) << "]";
  }
};

template <class F>
void SecretChatEvent::downcast_call(Type type, F &&f) {
  switch (type) {
    case Type::InboundSecretMessage:
      f(static_cast<InboundSecretMessage *>(nullptr));
      break;
    case Type::OutboundSecretMessage:
      f(static_cast<OutboundSecretMessage *>(nullptr));
      break;
    case Type::CloseSecretChat:
      f(static_cast<CloseSecretChat *>(nullptr));
      break;
    case Type::CreateSecretChat:
      f(static_cast<CreateSecretChat *>(nullptr));
      break;
    default:
      break;
  }
}
}  // namespace logevent

inline auto create_storer(logevent::SecretChatEvent &event) {
  return logevent::detail::StorerImpl<logevent::SecretChatEvent>(event);
}

}  // namespace td
//
// Copyright Aliaksei Levin (levlam@telegram.org), Arseny Smirnov (arseny30@gmail.com) 2014-2020
//
// Distributed under the Boost Software License, Version 1.0. (See accompanying
// file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)
//
#pragma once

#include "td/telegram/logevent/LogEvent.h"

#include "td/actor/PromiseFuture.h"

#include "td/utils/buffer.h"
#include "td/utils/common.h"
#include "td/utils/format.h"
#include "td/utils/StringBuilder.h"
#include "td/utils/tl_helpers.h"
```

```
#include "td/telegram/secret_api.h"
#include "td/telegram/telegram_api.h"

namespace td {
namespace logevent {

class SecretChatEvent : public LogEventBase<SecretChatEvent> {
 public:
  // append only enum
  enum class Type : int32 {
    InboundSecretMessage = 1,
    OutboundSecretMessage = 2,
    CloseSecretChat = 3,
    CreateSecretChat = 4
  };

  virtual Type get_type() const = 0;

  static constexpr LogEvent::HandlerType get_handler_type() {
    return LogEvent::HandlerType::SecretChats;
  }

  static constexpr int32 version() {
    return 2;
  }

  template <class F>
  static void downcast_call(Type type, F &&f);
};

template <class ChildT>
class SecretChatLogEventBase : public SecretChatEvent {
 public:
  typename SecretChatEvent::Type get_type() const override {
    return ChildT::type;
  }

  constexpr int32 magic() const {
    return static_cast<int32>(get_type());
  }
};

// Internal structure

// inputEncryptedFileEmpty#1837c364 = InputEncryptedFile;
// inputEncryptedFileUploaded#64bd0306 id:long parts:int md5_checksum:string key_fingerprint:int = InputEncryptedFile;
// inputEncryptedFile#5a17b5e5 id:long access_hash:long = InputEncryptedFile;
// inputEncryptedFileBigUploaded#2dc173c8 id:long parts:int key_fingerprint:int = InputEncryptedFile;
struct EncryptedInputFile {
  static constexpr int32 MAGIC = 0x4328d38a;
  enum Type : int32 { Empty = 0, Uploaded = 1, BigUploaded = 2, Location = 3 } type = Type::Empty;
  int64 id = 0;
  int64 access_hash = 0;
  int32 parts = 0;
  int32 key_fingerprint = 0;

  template <class StorerT>
  void store(StorerT &storer) const {
    using td::store;
    store(MAGIC, storer);
    store(type, storer);
    store(id, storer);
    store(access_hash, storer);
    store(parts, storer);
    store(key_fingerprint, storer);
  }

  EncryptedInputFile() = default;
  EncryptedInputFile(Type type, int64 id, int64 access_hash, int32 parts, int32 key_fingerprint)
      : type(type), id(id), access_hash(access_hash), parts(parts), key_fingerprint(key_fingerprint) {
  }

  bool empty() const {
    return type == Empty;
  }

  template <class ParserT>
  void parse(ParserT &parser) {
    using td::parse;
    int32 got_magic;

    parse(got_magic, parser);
    parse(type, parser);
    parse(id, parser);
    parse(access_hash, parser);
    parse(parts, parser);
    parse(key_fingerprint, parser);

    if (got_magic != MAGIC) {
      parser.set_error("EncryptedInputFile magic mismatch");
      return;
    }
  }
  static EncryptedInputFile from_input_encrypted_file(const tl_object_ptr<telegram_api::InputEncryptedFile> &from) {
    if (!from) {
      return EncryptedInputFile{Empty, 0, 0, 0, 0};
    }
    return from_input_encrypted_file(*from);
  }
  static EncryptedInputFile from_input_encrypted_file(const telegram_api::InputEncryptedFile &from) {
    switch (from.get_id()) {
      case telegram_api::inputEncryptedFileEmpty::ID:
        return EncryptedInputFile{Empty, 0, 0, 0, 0};
      case telegram_api::inputEncryptedFileUploaded::ID: {
        auto &uploaded = static_cast<const telegram_api::inputEncryptedFileUploaded &>(from);
        return EncryptedInputFile{Uploaded, uploaded.id_, 0, uploaded.parts_, uploaded.key_fingerprint_};
      }
      case telegram_api::inputEncryptedFileBigUploaded::ID: {
        auto &uploaded = static_cast<const telegram_api::inputEncryptedFileBigUploaded &>(from);
        return EncryptedInputFile{BigUploaded, uploaded.id_, 0, uploaded.parts_, uploaded.key_fingerprint_};
      }
      case telegram_api::inputEncryptedFile::ID: {
        auto &uploaded = static_cast<const telegram_api::inputEncryptedFile &>(from);
        return EncryptedInputFile{Location, uploaded.id_, uploaded.access_hash_, 0, 0};
      }
      default:
        UNREACHABLE();
    }
  }
  tl_object_ptr<telegram_api::InputEncryptedFile> as_input_encrypted_file() const {
    switch (type) {
      case Empty:
        return make_tl_object<telegram_api::inputEncryptedFileEmpty>();
      case Uploaded:
```

```
        return make_tl_object<telegram_api::inputEncryptedFileUploaded>(id, parts, "", key_fingerprint);
      case BigUploaded:
        return make_tl_object<telegram_api::inputEncryptedFileBigUploaded>(id, parts, key_fingerprint);
      case Location:
        return make_tl_object<telegram_api::inputEncryptedFile>(id, access_hash);
    }
    UNREACHABLE();
  }
};

inline StringBuilder &operator<<(StringBuilder &sb, const EncryptedInputFile &file) {
  return sb << to_string(file.as_input_encrypted_file());
}

// encryptedFile#4a70994c id:long access_hash:long size:int dc_id:int key_fingerprint:int = EncryptedFile;
struct EncryptedFileLocation {
  static constexpr int32 MAGIC = 0x473d738a;
  int64 id = 0;
  int64 access_hash = 0;
  int32 size = 0;
  int32 dc_id = 0;
  int32 key_fingerprint = 0;

  tl_object_ptr<telegram_api::encryptedFile> as_encrypted_file() {
    return make_tl_object<telegram_api::encryptedFile>(id, access_hash, size, dc_id, key_fingerprint);
  }
  template <class StorerT>
  void store(StorerT &storer) const {
    using td::store;
    store(MAGIC, storer);
    store(id, storer);
    store(access_hash, storer);
    store(size, storer);
    store(dc_id, storer);
    store(key_fingerprint, storer);
  }

  template <class ParserT>
  void parse(ParserT &parser) {
    using td::parse;
    int32 got_magic;

    parse(got_magic, parser);
    parse(id, parser);
    parse(access_hash, parser);
    parse(size, parser);
    parse(dc_id, parser);
    parse(key_fingerprint, parser);

    if (got_magic != MAGIC) {
      parser.set_error("EncryptedFileLocation magic mismatch");
      return;
    }
  }
};

inline StringBuilder &operator<<(StringBuilder &sb, const EncryptedFileLocation &file) {
  return sb << "[" << tag("id", file.id) << tag("access_hash", file.access_hash) << tag("size", file.size)
            << tag("dc_id", file.dc_id) << tag("key_fingerprint", file.key_fingerprint) << "]";
}

// LogEvents
// TODO: Qts and SeqNoState could be just Logevents that are updated during regenerate
class InboundSecretMessage : public SecretChatLogEventBase<InboundSecretMessage> {
 public:
  static constexpr Type type = SecretChatEvent::Type::InboundSecretMessage;
  int32 qts = 0;

  int32 chat_id = 0;
  int32 date = 0;

  BufferSlice encrypted_message;  // empty when we store event to binlog
  Promise<Unit> qts_ack;

  bool is_checked = false;
  // after decrypted and checked
  tl_object_ptr<secret_api::decryptedMessageLayer> decrypted_message_layer;

  uint64 auth_key_id = 0;
  int32 message_id = 0;
  int32 my_in_seq_no = -1;
  int32 my_out_seq_no = -1;
  int32 his_in_seq_no = -1;

  int32 his_layer() const {
    return decrypted_message_layer->layer_;
  }

  EncryptedFileLocation file;

  bool has_encrypted_file = false;
  bool is_pending = false;

  template <class StorerT>
  void store(StorerT &storer) const {
    using td::store;

    BEGIN_STORE_FLAGS();
    STORE_FLAG(has_encrypted_file);
    STORE_FLAG(is_pending);
    END_STORE_FLAGS();

    store(qts, storer);
    store(chat_id, storer);
    store(date, storer);
    // skip encrypted_message
    // skip qts_ack

    // TODO
    decrypted_message_layer->store(storer);
    storer.store_long(static_cast<int64>(auth_key_id));

    store(message_id, storer);
    store(my_in_seq_no, storer);
    store(my_out_seq_no, storer);
    store(his_in_seq_no, storer);
    if (has_encrypted_file) {
      store(file, storer);
    }
  }

  template <class ParserT>
  void parse(ParserT &parser) {
    using td::parse;
```

```
    BEGIN_PARSE_FLAGS();
    PARSE_FLAG(has_encrypted_file);
    PARSE_FLAG(is_pending);
    END_PARSE_FLAGS();

    parse(qts, parser);
    parse(chat_id, parser);
    parse(date, parser);
    // skip encrypted_message
    // skip qts_ack

    // TODO
    decrypted_message_layer = secret_api::decryptedMessageLayer::fetch(parser);
    auth_key_id = static_cast<uint64>(parser.fetch_long());

    parse(message_id, parser);
    parse(my_in_seq_no, parser);
    parse(my_out_seq_no, parser);
    parse(his_in_seq_no, parser);
    if (has_encrypted_file) {
      parse(file, parser);
    }

    is_checked = true;
  }

  StringBuilder &print(StringBuilder &sb) const override {
    return sb << "[Logevent InboundSecretMessage " << tag("id", logevent_id()) << tag("qts", qts)
              << tag("chat_id", chat_id) << tag("date", date) << tag("auth_key_id", format::as_hex(auth_key_id))
              << tag("message_id", message_id) << tag("my_in_seq_no", my_in_seq_no)
              << tag("my_out_seq_no", my_out_seq_no) << tag("his_in_seq_no", his_in_seq_no)
              << tag("message", to_string(decrypted_message_layer)) << tag("is_pending", is_pending)
              << format::cond(has_encrypted_file, tag("file", file)) << "]";
  }
};

class OutboundSecretMessage : public SecretChatLogEventBase<OutboundSecretMessage> {
 public:
  static constexpr Type type = SecretChatEvent::Type::OutboundSecretMessage;

  int32 chat_id = 0;
  int64 random_id = 0;

  BufferSlice encrypted_message;
  EncryptedInputFile file;

  int32 message_id = 0;
  int32 my_in_seq_no = -1;
  int32 my_out_seq_no = -1;
  int32 his_in_seq_no = -1;

  int32 his_layer() const {
    return -1;
  }

  bool is_sent = false;
  // need send push notification to the receiver
  // should send such messages with messages_sendEncryptedService
  bool need_notify_user = false;
  bool is_rewritable = false;
  // should notify our parent about state of this message (using context and random_id)
  bool is_external = false;

  tl_object_ptr<secret_api::DecryptedMessageAction> action;
  uint64 crc = 0;  // DEBUG;

  // Flags:
  // 2. can_fail = !file.empty() // send of other messages can't fail if chat is ok. It is usless to rewrite them with
  // empty
  // 3. can_rewrite_with_empty // false for almost all service messages

  // TODO: combine these two functions into one macros hell. Or a lambda hell.
  template <class StorerT>
  void store(StorerT &storer) const {
    using td::store;

    store(chat_id, storer);
    store(random_id, storer);
    store(encrypted_message, storer);
    store(file, storer);
    store(message_id, storer);
    store(my_in_seq_no, storer);
    store(my_out_seq_no, storer);
    store(his_in_seq_no, storer);

    bool has_action = static_cast<bool>(action);
    BEGIN_STORE_FLAGS();
    STORE_FLAG(is_sent);
    STORE_FLAG(need_notify_user);
    STORE_FLAG(has_action);
    STORE_FLAG(is_rewritable);
    STORE_FLAG(is_external);
    END_STORE_FLAGS();

    if (has_action) {
      CHECK(action);
      // TODO
      storer.store_int(action->get_id());
      action->store(storer);
    }
  }

  template <class ParserT>
  void parse(ParserT &parser) {
    using td::parse;

    parse(chat_id, parser);
    parse(random_id, parser);
    parse(encrypted_message, parser);
    parse(file, parser);
    parse(message_id, parser);
    parse(my_in_seq_no, parser);
    parse(my_out_seq_no, parser);
    parse(his_in_seq_no, parser);

    bool has_action;
    BEGIN_PARSE_FLAGS();
    PARSE_FLAG(is_sent);
    PARSE_FLAG(need_notify_user);
    PARSE_FLAG(has_action);
    PARSE_FLAG(is_rewritable);
    PARSE_FLAG(is_external);
    END_PARSE_FLAGS();

    if (has_action) {
```