Технический аудит мессенджера ProtoChatX (Android)

Полный отчёт (без сокращений).

Дата: 24.01.2026

Подготовлено для: Профессор NIKURA

Содержание

Технический аудит мессенджера ProtoChatX (Android)

Краткое резюме: ProtoChatX – это автономный защищённый мессенджер с веб-клиентом (HTML/JavaScript) и сервером-реле на Node.js. В ходе аудита установлено, что приложение реализует сквозное шифрование (E2EE) всех сообщений и звонков, за счёт чего сервер не имеет доступа к открытому содержимому переписки. Архитектура «облегчённого» реле надёжна для малых масштабов, но хранение данных в JSON-файлах накладывает ограничения при росте нагрузки. Проведены эффективные меры по защите от атак (например, ограничение размера и частоты сообщений для предотвращения DoS, привязка устройства к криптопрофилю пользователя для предотвращения компрометации аккаунта). В то же время обнаружены потенциал для улучшений: например, хранение секретных ключей в хешированном виде, повышение масштабируемости (переход на БД), а также некоторые аспекты UX (процесс восстановления аккаунта, информирование о смене ключей) и DevOps. Ниже приводится подробный разбор каждого из запрошенных пунктов аудита.

Безопасность и конфиденциальность

Сквозное шифрование (E2EE): Все обмены сообщениями и звонками в ProtoChatX защищены E2EE по умолчанию. Это означает, что перед отправкой каждое сообщение шифруется на устройстве отправителя, а расшифровать его может только получатель на своём устройстве. Сервер выступает лишь транспортом и видит бессмысленный шифротекст, не имея ключей для дешифровки. Даже если злоумышленник перехватит трафик или получит доступ к серверу, он не сможет прочитать переписку – для сервера сообщения выглядят как нечитаемый набор данных. В отличие от традиционных сервисов, разработчики или администраторы сервера физически не имеют возможности подглядеть содержимое сообщений или звонков пользователей.

Реализация E2EE: В основе протокола шифрования – сочетание алгоритмов ECDH и AES-GCM. При регистрации у каждого пользователя генерируется пара криптографических ключей (эллиптическая кривая P-256) – приватный и публичный. Публичный ключ (в формате JWK) сохраняется на сервере и автоматически распространяется среди клиентов при авторизации (сервер рассылает список пользователей с их открытыми ключами). Приватный ключ хранится только на устройстве пользователя (в хранилище типа IndexedDB, помеченном как неэкспортируемое) и никогда не отправляется на сервер. Таким образом, сервер не знает приватных ключей и компрометация сервера не позволит расшифровать переписку. Перед отправкой сообщения клиент упаковывает данные (текст, вложения) в JSON-объект с полем kind (тип сообщения: обычный чат, аватар, звонок и пр.), после чего этот объект шифруется алгоритмом AES-GCM с сеансовым ключом, известным только отправителю и получателю. Сеансовый ключ получается через ECDH:

отправитель берёт свой приватный ключ и публичный ключ получателя (полученный из списка пользователей), вычисляет общую секретную точку и производит из неё 256-битный ключ сеанса. Аналогично получатель при приёме использует свой приватный ключ и публичный ключ отправителя для расчёта того же сеансового ключа и расшифровки сообщения. В итоге сервер пересылает только готовый шифротекст (поле payload), а вся критичная криптография выполняется на клиентах. Даже файлы и голосовые сообщения сначала конвертируются в base64, а затем шифруются внутри E2EE-пакета – сервер никогда не видит их содержимого в открытом виде. Исключение составляют лишь некоторые служебные метаданные (тип сообщения kind, ID отправителя/получателя, временные метки и пр.), необходимые для логики доставки – они передаются в открытом виде, но не раскрывают конфиденциального содержания переписки.

Хранение ключей, секретов и recovery-кодов: Безопасность аккаунта опирается на два секрета: постоянный [redacted] пользователя (аналог пароля/API-токена) и его E2E-ключ (криптопара ключей). [redacted] генерируется случайно при регистрации – это 32 байта криптографически случайных данных, которые кодируются в base64url длиной ~43 символа. Благодаря такой длине и энтропии подбор [redacted] методом перебора практически исключён. Сервер требует, чтобы [redacted] был не короче 12 символов (слишком короткие обрубаются) для повышения стойкости к подбору. Сгенерированный [redacted] сохраняется на сервере в открытом виде (как поле в [redacted]) и используется для аутентификации: при логине клиент передаёт ID и [redacted], сервер ищет пользователя и просто сравнивает присланный секрет с хранящимся, используя crypto.timingSafeEqual для защиты от тайминг-атак. Отметим, что хранение [redacted]s в незашифрованном виде – спорный момент. С одной стороны, [redacted] не является простым паролем пользователя, а представляет собой сложную случайную строку, поэтому утечка базы не даёт атакующему тривиального пароля, который можно угадать или быстро взломать. С другой стороны, общепринята практика хранить пароли/ключи в виде хэшей. В проектной документации прямо предлагается улучшение: хранить на сервере не сам [redacted], а его криптографический хэш (например, bcrypt); при логине хэшировать введённый секрет и сравнивать с сохранённым хэшем. Это не позволило бы злоумышленнику при компрометации [redacted] мгновенно получить действующие ключи – пришлось бы сначала взломать хэши. В текущей реализации от хеширования отказались для упрощения (ведь [redacted] и так длинный и случайный), чтобы не замедлять процесс входа, однако с точки зрения «лучших практик» безопасности подобное улучшение обосновано.

Публичный ключ (pubkey, JWK) пользователя хранится на сервере в профиле и выступает своего рода идентификатором/доверенным ключом пользователя. Привязка аккаунта к устройству через E2E-ключ – одна из сильных сторон безопасности ProtoChatX. Когда пользователь впервые заходит, он регистрирует свой публичный ключ на сервере; при последующих входах клиент снова отправляет свой pubkey, и сервер проверяет соответствие с сохранённым ключом. Если ключ не совпадает (например, атакующий знает [redacted], но пытается зайти со своим ключом), сервер вернёт ошибку

PUBKEY_MISMATCH и не допустит соединение. Таким образом, краже одного лишь [redacted] недостаточно для компрометации – нужен ещё и приватный ключ устройства жертвы. [redacted]+pubkey вместе образуют двухфакторную аутентификацию на уровне устройства. Заменить привязанный ключ можно только осознанно – через процедуру восстановления аккаунта (Recovery) или сброс администратором. Иначе говоря, аккаунт жёстко привязан к конкретному устройству (ключевой паре), что повышает безопасность ценой некоторых неудобств.

Процедура восстановления аккаунта: При утрате устройства или [redacted] реализован офлайн-механизм Recovery через одноразовые recovery-коды. При регистрации приложение генерирует и сразу показывает пользователю набор из нескольких (по умолчанию 8) recovery-кодов. Каждый код – это случайная строка (8 байт, кодированных в Base64, итого ~11 символов) в верхнем регистре, наподобие «ABCD-EFGH-IJKL». Пользователю настоятельно рекомендуется сохранить эти коды в надёжном месте вне устройства (например, записать на бумаге или в менеджере паролей). Recovery-коды не хранятся в приложении: после отображения они доступны только пользователю. На сервере хранатся только SHA-256 хэши кодов (с добавлением секретной «перчинки»/pepper) – это видно из структуры [redacted], где для каждого выданного кода хранится hash и флаг used. Если пользователь потерял устройство или удалил приложение, он может установить ProtoChatX заново на новом устройстве и вместо логина по [redacted] выполнить команду восстановления (тип сообщения recover). В процессе Recovery клиент отправляет на сервер свой ID, один из ранее сохранённых recovery-кодов и свой новый публичный ключ. Сервер проверяет код: если он совпадает с одним из хэшей и ещё не использован, то считается действительным. После этого сервер сгенерирует пользователю новый [redacted], привяжет к аккаунту новый публичный ключ (т.е. фактически перенесёт аккаунт на новое устройство) и выдаст новый набор recovery-кодов. Старый секрет и старые recovery-коды при этом аннулируются. Таким образом, владея одним из выданных recovery-кодов, можно безопасно «перенести» свой аккаунт на другой телефон, даже не зная старый [redacted]. Важно подчеркнуть: без recovery-кода восстановить доступ невозможно – это сознательное решение в пользу безопасности. Если пользователь утерял все recovery-коды, доступ к его аккаунту потерян безвозвратно. Никто (даже администратор) не сможет выдать новый код или сбросить [redacted], т.к. это открыло бы лазейку для захвата аккаунта злоумышленником. В таком случае остаётся только регистрировать новый аккаунт и уведомлять контакты. Этот аспект требует внимательного отношения со стороны пользователя (приложение на этапе регистрации предупреждает сохранить код восстановления в безопасном месте). С точки зрения аудита, механизм Recovery выполнен грамотно: используется криптостойкий одноразовый код, сервер помечает использованный код как расходованный и обновляет секреты. Единственный теоретический вектор атаки – подбор recovery-кода – крайне маловероятен (пространство > 2^50 вариантов, плюс применяется pepper для усложнения перебора хэшей).

Защита от MITM и подмены ключей: ProtoChatX спроектирован так, чтобы противостоять атакам типа «человек посередине». Первичный обмен ключами происходит автоматизировано и напрямую между устройствами при установлении нового чата или звонка. На практике это означает: когда вы начинаете общение с новым контактом, приложение использует уже имеющийся публичный ключ собеседника (из списка пользователей, полученного при авторизации) либо запрашивает его напрямую у собеседника через защищённый канал. Сервер не вмешивается в установление E2E-ключей, он лишь может раздавать ранее загруженные публичные ключи. Любая попытка атакующего подменить чужой публичный ключ незаметно для пользователя должна быть обнаружена за счёт механизма проверок. Как минимум, если злоумышленник подменит ключ в момент передачи через сервер, это сразу приведёт к несоответствию при следующем входе (поскольку оригинальный пользователь не сможет пройти проверку pubkey, либо контакт увидит изменение ключа). Согласно документации, любая попытка подменить ключ будет обнаружена, и соединение не установится. В текущей реализации доверие к ключам строится на принципе «trust on first use»: впервые полученный от сервера публичный ключ контакта считается достоверным (т.к. сервер предполагается доверенным окружением). Если впоследствии ключ контакта изменится (например, при Recovery), сервер сообщит обновлённый ключ, и приложение, вероятно, может уведомить пользователя об изменении (в коде присутствуют пакеты ver_req[redacted], что может указывать на механизм верификации ключей). Хотя явного UI для ручной проверки отпечатков ключей не упоминается, за счёт уникальности крипто-идентификатора у пользователя есть уверенность, что «общается именно с тем, с кем думает». В целом, протокол опирается на сложность компрометации сервера и на привязку устройства к ключу, поэтому риск MITM атак минимизирован. Тем не менее, в качестве рекомендации можно рассмотреть реализацию функции проверки собеседников (например, сравнение отпечатков ключей по QR-коду или словесной фразе) для параноидального уровня безопасности – сейчас это не реализовано.

Анализ возможных уязвимостей: В ходе аудита не выявлено очевидных изъянов в криптографической схеме – используются надёжные алгоритмы, открытые ключи хранятся на сервере для удобства, но приватные остаются на устройствах. JWT/JWK: Протокол не использует JWT-токены для аутентификации, поэтому уязвимости неправильной обработки JWT отсутствуют. Формат JWK задействован для хранения публичных ключей, что оправдано и реализовано корректно (ключи сериализуются в JSON, передаются клиентам). Здесь стоит убедиться, что библиотека или код, выполняющий сериализацию JWK, не содержит ошибок – судя по всему, используется стандартный формат без самописных реализаций, проблем не обнаружено. Защита от повторов: Сервер ProtoChatX ввёл специальные меры против replay-атак в контексте вызовов. Например, сигнальные пакеты звонка (call_invite, call_cancel и т.п.) помечены как не подлежащие повторной доставке: если звонящий отменил вызов, то call_cancel не будет потом воспроизведён из офлайн-очереди, чтобы не вызывать «призрак» звонка.

Кроме того, одинаковые повторные приглашения и SDP-предложения дедуплицируются в очереди. Это предотвращает простые повторы. Что касается шифрованных сообщений, стандартный AES-GCM обеспечивает целостность и обнаружение повторов на уровне сессии (nonce не должен повторяться при шифровании). Клиент при получении, вероятно, проверяет уникальность msg_id сообщений в своей истории. В серверном коде нет явного отсечения повторных msg_id, но повторно посланный зашифрованный payload либо будет иметь тот же msg_id (который клиент может игнорировать как дубликат), либо новый – тогда это эквивалент повторной отправки сообщения самим отправителем. Значимой уязвимости здесь не просматривается, хотя можно добавить защиту от дублей и на сервере (например, хранить последние N msg_id для каждого отправителя на время сессии).

Дополнительные меры защиты: ProtoChatX также интегрирован с Firebase Cloud Messaging ([redacted]) для пуш-уведомлений, но разработчики позаботились, чтобы через push не утекали приватные данные. Когда пользователь офлайн, сервер шлёт push с минимальными данными: для нового сообщения – только тип "msg" и ID отправителя, без текста. Уведомление, приходящее на телефон, просто говорит «Новое сообщение» без какого-либо содержимого. При входящем звонке сервер отправляет push с типом "call" и параметрами звонка (например, действие и ID звонящего), но тоже без личных данных. Такие push помечаются приоритетом high и имеют collapse key по ID звонящего (чтобы объединить несколько звонков от одного контакта). На устройстве они обрабатываются специальным кодом: push типа call запускает системный экран вызова (будит телефон и показывает входящий вызов), а push call_ctl (например, отмена) – передаёт приложению команду снять звонок, чтобы остановить рингтон. Эта схема гарантирует, что в офлайне телефон зазвонит при вызове, но если вызов отменён, звонок не зависнет. Важно, что в push-уведомлениях не передаются конфиденциальные сведения – только служебные сигналы. Это грамотное решение: даже если злоумышленник как-то перехватит push (что маловероятно), он не узнает содержимое сообщения.

Вывод по безопасности: ProtoChatX демонстрирует высокий уровень защиты данных «из коробки». Сквозное шифрование и отказ от централизованного хранения переписки означают, что сервер не обладает информацией для компрометации приватности пользователей. Аккаунты защищены случайными секретами и привязаны к устройствам (что служит дополнительным барьером против взлома). В архитектуре соблюдены принципы минимизации данных: хранятся только необходимые технические сведения (идентификаторы, хэшированные инвайты и recovery-коды, [redacted]-токены и пр.). Приложение не требует от пользователей никаких личных данных (ни номера телефона, ни e-mail), обеспечивая анонимность – учетная запись это псевдоним (ник) + криптографический ключ. В итоге, даже разработчики или администраторы не имеют доступа к переписке, что подчёркивается открытостью исходного кода (проект открыт для аудита сторонними экспертами). Для достижения такого уровня безопасности пользователю приходится принимать некоторые неудобства – например, необходимость хранить recovery-код, невозможность

одновременной работы с нескольких устройств – однако это осознанный компромисс в пользу конфиденциальности.

Архитектура и масштабируемость

Общая схема архитектуры: Архитектура ProtoChatX соответствует модели «клиент – облегчённый сервер-реле». Клиентская часть – это приложение на Capacitor (webview с HTML/JS), серверная – Node.js процесс, выполняющий роль ретранслятора сообщений. Между клиентом и сервером устанавливается постоянное соединение по WebSocket (порт по умолчанию [redacted], используется WSS для безопасности) для обмена всеми данными – сообщениями чата, сигналами вызовов, служебными командами. Сервер (файл [redacted]) поднимает WS-сервер и обрабатывает входящие JSON-пакеты по определённому протоколу. Административные функции (регистрация пользователей, управление инвайтами, просмотр статистики) вынесены в отдельный HTTP-сервис ([redacted] на порту [redacted]). Такая сегрегация повышает надёжность: основной сервер сфокусирован на доставке данных, а админка – на второстепенных задачах.

Relay-сервер: Сервер-реле практически не содержит бизнес-логики хранения – он не сохраняет историю сообщений на постоянной основе, выступая по сути транзитным буфером. Когда два пользователя оба онлайн, сообщения пересылаются напрямую в режиме реального времени через WS. Если получатель офлайн, сервер временно помещает зашифрованный пакет в свою офлайн-очередь до момента, пока получатель не появится в сети. Важный момент: сервер не видит и не хранит открытый контент сообщений – только пересылает шифрованные блоки (опционально буферизуя их). Эта архитектурная особенность соответствует заявленной цели максимальной приватности.

Хранилище данных: Вместо СУБД сервер применяет плоские JSON-файлы для хранения состояния. Основные файлы: [redacted] – база пользователей, [redacted] – активные инвайт-коды, [redacted] – офлайн-очередь сообщений. Все они лежат на сервере и доступны только ему; доступ синхронизирован примитивной файловой блокировкой (создаётся временный lock-файл .state.lock, операции, модифицирующие состояние, обернуты в withStateLock). Это исключает гонки при одновременном доступе (например, одновременная регистрация по одному инвайту не приведёт к дублированию аккаунта). Такой подход упрощает развёртывание – не нужно поднимать отдельную базу данных, всё хранится в нескольких файлах – однако накладывает ограничения на масштабируемость и требует тщательной защиты файлов на сервере (см. ниже о безопасности операционной среды).

Структура [redacted] включает: числовой User ID, [redacted] (секрет аутентификации), никнейм, флаг активности (enabled/disabled), публичный E2E-ключ (JWK), отметки времени создания/обновления, [redacted]-токен устройства и прочие поля (примечание, резервированный ID и т.д.). [redacted] хранит

инвайты в виде хэш-кода -> объекта с параметрами (сколько использований осталось, срок годности, зарезервированный ID, примечание). [redacted] содержит словарь, где для каждого user_id хранится массив недоставленных элементов (шифротекстов) офлайн-очереди.

Оффлайн-очередь сообщений: Когда отправитель посылает сообщение, сервер проверяет – если получатель онлайн, сообщение сразу пересылается; если получатель офлайн, сервер отвечает отправителю статусом queued и сохраняет пакет в [redacted]. Для каждого пользователя поддерживается очередь ограниченного размера: не более 200 сообщений или ~2 МБ данных на пользователя; старые сообщения сверх лимита, а также сообщения старше 7 дней автоматически удаляются (TTL задаётся константой, по умолчанию 7*24ч). Благодаря этому история не висит на сервере бесконечно – хранится только то, что необходимо для доставки, и не дольше недели. Также реализована дедупликация и правила для некоторых типов пакетов: например, звонковые сигналы помечены флагом «no-queue», то есть не сохраняются в оффлайне, поскольку имеют смысл только для онлайн-пользователей. Это предотвращает ложные срабатывания – скажем, если вызов давно отменён, оффлайн-очередь не «призрачит» его при следующем входе пользователя. В частности, при сохранении очереди сервер фильтрует дубликаты: для повторных call_invite от одного отправителя в очереди хранится только последняя попытка (предыдущие удаляются), для повторных webrtc_offer[redacted] по одному звонку – тоже сохраняется только последний вариант SDP. Таким образом, очередь не раздувается от одинаковых сигналов. Когда пользователь выходит онлайн (логинится), сервер сразу флашит его очередь: все накопленные сообщения вытаскиваются из [redacted] и отправляются клиенту подряд через WS, после чего соответствующие записи удаляются. Отправителю каждого такого сообщения сервер при этом шлёт уведомление sent: delivered (если отправитель ещё онлайн), чтобы тот узнал о доставке. Механизм построен довольно надёжно: используется атомарная запись файлов (через временные копии и rename, чтобы избежать порчи данных при крэше в середине записи), ошибки записи логируются (QUEUE_WRITE_ERR). В случае сбоя или рестарта сервера оффлайн-очередь сохраняется в файле и будет обработана при следующем логине пользователей.

Обработка звонков (WebRTC): Аудио/видео-вызовы реализованы через WebRTC, а сигнализация для установления соединения (инвайты, ответы, ICE-кандидаты) идёт поверх того же WebSocket-протокола. Сервер не микширует медиа и не выступает посредником в передаче аудио/видео – после обмена сигналами клиенты соединяются либо напрямую peer-to-peer, либо через TURN-сервер (для обхода NAT). Ключевые сигналы вызова: call_invite (исходящий вызов), call_accept (принятие), call_cancel[redacted] (отмена или завершение), call_busy (занято). Как упоминалось, они не ставятся в оффлайн-очередь и доставляются только если оба онлайн (звонок по определению требует оба конца онлайн). Для WebRTC SDP-предложений (webrtc_offer[redacted]) и ICE-кандидатов (webrtc_ice) сделано исключение: эти пакеты могут временно сохраняться даже оффлайн, но с очень коротким TTL

(десятки секунд). Это на случай, если, например, приглашение пришло, но ответ чуть задержался – чтобы попытка соединения не сорвалась из-за кратковременного рассинхрона. TURN-сервер поддерживается: при запросе turn_cred relay-сервер генерирует временные логин/пароль по схеме HMAC-SHA1 (если задан секрет [redacted]) и отдаёт клиенту вместе со списком TURN/STUN адресов. Такой credential действителен ~1 час. По умолчанию ProtoChatX интегрируется с coturn или аналогичным сервером. Это обеспечивает надёжные звонки через NAT: сначала клиенты пытаются установить прямое P2P-соединение (WebRTC ICE), и если не удаётся – подключаются через TURN-ретранслятор. С точки зрения масштабируемости, медиатрафик не проходит через основной сервер, что снижает нагрузку на него – звонки масштабируются отдельно (необходимо лишь масштабировать TURN-сервер под количество одновременных звонков). Завершённые звонки не оставляют следов на сервере – ни содержимое, ни даже сам факт вызова не записываются в логах или файлах (кроме возможно технических отметок о том, что был отправлен push-уведомление звонка). После звонка только на устройствах в локальном журнале может отображаться, что звонок состоялся и когда. Это соответствует принципу, что даже метаданные общения минимизируются.

Relay-сервер – отказоустойчивость и масштабируемость: Выбранный подход с хранением данных в памяти и JSON-файлах хорошо подходит для небольшого числа пользователей (например, до нескольких сотен). Все основные структуры (users, invites, queue) при использовании читаются целиком в память, и при изменениях файл перезаписывается целиком. При малой базе (например, десятки–сотни пользователей, сотни сообщений офлайн) это не проблема – современные серверы легко обрабатывают файлы в сотни килобайт–несколько мегабайт. Однако при росте числа пользователей и объёма переписки такой подход станет узким местом. Каждый логин перечитывает весь [redacted] файл и перезаписывает его при обновлении поля (updatedAt, токен, и т.д.). Это линейная сложность O(N) на логин, что при тысячах пользователей может привести к задержкам. Аналогично, запись очереди при каждом оффлайн-сообщении – операция O(M) от длины очереди адресата (максимум 200 сообщений, что фиксировано и немного). При пиковых нагрузках (много одновременных сообщений) узким местом может стать именно файл I/O – Node.js выполняет запись синхронно (блокируя поток) при обновлении файлов, и хотя записи относительно малы, множественные операции могут выстраиваться в очередь.

Горизонтальное масштабирование (запуск нескольких экземпляров relay-сервера) в текущей архитектуре не предусмотрено – все инстансы должны работать с одними и теми же файлами, что потребовало бы внешней синхронизации. Поэтому ProtoChatX рассчитан скорее на автономное использование (например, одним сообществом или небольшой организацией). В документации прямо указано, что простое файловое хранилище упрощает установку, но ограничивает масштабируемость. При планировании роста системы (большое число пользователей или сообщений) стоит рассмотреть миграцию на более производительное хранилище – например, легковесную

встроенную БД (SQLite) либо клиент–серверную БД (PostgreSQL, MongoDB и т.д.). Это бы позволило O(1) доступ по ключу вместо чтения всего файла, а также упрощало бы горизонтальное масштабирование.

Админская панель и сервер управления: Администрирование осуществляется через встроенный веб-интерфейс (доступен по пути [redacted] на [redacted], по умолчанию [redacted]) и вспомогательный CLI-скрипт [redacted]. Admin API запускает HTTP-сервер, который отдаёт простую статическую страницу с формами (UI минималистичный). Админ вводит токен доступа (настраивается в переменных окружения как [redacted]) – без правильного токена любые запросы отклоняются с 401 Unauthorized. После авторизации доступны функции: добавить пользователя (с указанным или автоматически подобранным ID), задать/изменить никнейм, отключить/включить пользователя (флаги enabled), rotate [redacted] (сгенерировать новому [redacted] для пользователя в случае компрометации или по запросу), issue recovery codes (сгенерировать новый набор recovery-кодов), и управлять инвайт-кодами (создавать новые, пакетно выдавать 10 шт., с опциональным резервом ID). Панель также позволяет посмотреть список всех пользователей (без секретных полей) и очистить оффлайн-очередь конкретного пользователя. Практически все операции панель выполняет, вызывая соответствующие endpoints Admin API, который внутри либо вызывает CLI-скрипт [redacted] (для создания пользователя, rotate, enable/disable), либо прямо правит JSON-файлы (например, генерация инвайта, выдача recovery-кодов) с блокировкой состояния. Решение с использованием CLI несколько нестандартно, но в контексте небольшого проекта допустимо: [redacted] обрабатывает логику генерации [redacted] (32-байтный random Base64url), привязки пользователя и т.д., а Admin API парсит вывод и возвращает его в веб-UI. Это изолирует чувствительные операции (создание аккаунта) и гарантирует, что генерация ключей происходит строго на сервере, а не где-то в браузере.

Надёжность архитектуры: При корректной эксплуатации архитектура выглядит надёжной для задуманных случаев использования (локальный/корпоративный мессенджер с числом пользователей в пределах сотен). Использование файловой блокировки и атомарных записей обеспечивает консистентность данных даже при конкурентных админских операциях. Отсутствие сторонних сервисов (SQL/NoSQL БД) упрощает установку и снижает количество точек отказа – достаточно самого Node.js процесса. Единственная точка отказа – сам Node.js сервер: в текущем виде он не кластеризован, т.е. если процесс упадёт, связь пользователей прервётся. В проекте не упоминается механизм автоматического рестарта или резервирования, поэтому для высокой доступности нужно внешнее решение (супервизор процесса, бэкап-сервер). Тем не менее, при падении сервера он не теряет данных, т.к. всё состояние регулярно сериализуется в файлы (которые можно поднять при рестарте).

Масштабируемость: Как указано, узкие места – монолитный сервер и файловое хранение. В текущем виде ProtoChatX лучше разворачивать как единичный сервер на достаточной машине, чем пытаться распределить. В перспективе,

если планируется рост, можно улучшить архитектуру следующим образом:

Вынести хранение пользователей и очередей из памяти в БД, с возможностью запуска нескольких серверов-реле, читающих из общей БД. Тогда пользователи могут подключаться к любому из реплицированных relay-серверов (потребуется механизм маршрутизации WS-соединения, например через общий шину или message broker).

Либо, как вариант, оставить одного relay-сервера, но вынести push-уведомления и тяжелые фоновые операции (очистка старых сообщений, генерация отчетов) в отдельные воркеры или микросервисы. Пока проект мал, в этом нет необходимости.

TURN и NAT: Интеграция TURN/STUN уже заложена и масштабируется отдельно. Важно настроить производительный TURN-сервер и задать [redacted], чтобы генерация учетных данных происходила автоматически. При высокой нагрузке звонков нужно обратить внимание на пропускную способность TURN – это за рамками самого ProtoChatX, но влияет на UX.

В заключение, архитектура ProtoChatX в текущем виде достигает поставленной цели – защищённого автономного мессенджера. Она предельно упрощена для развертывания (Node.js + файлы, без внешних зависимостей) и показала согласованную работу всех модулей (WebSocket, WebRTC, push, офлайн-очереди). В реальности подтверждается, что сервер лишь помогает доставлять сообщения, но не хранит их – чаты полностью живут на устройствах пользователей. Интеграция [redacted] push и NAT-траверсала (TURN) выполнена корректно. Если соблюдать рекомендации (например, использовать TLS для WS, защищать файловое хранилище от доступа посторонних, держать в секрете админ-токен), архитектура выглядит безопасной и достаточно надёжной в рамках оговоренных масштабов.

Производительность и стабильность

Ограничения очередей и ресурсов: Разработчики ProtoChatX предусмотрели несколько уровней ограничений для поддержания производительности и противодействия злоупотреблениям. Во-первых, как уже отмечалось, размер оффлайн-очереди на пользователя ограничен ~2 МБ или 200 сообщений. Это означает, что даже если контакт завалил оффлайн-пользователя сообщениями, сервер не выйдет за контролируемые рамки памяти/диска – старые сообщения начнут удаляться. По умолчанию также установлен TTL ~7 дней, после которого не доставленные сообщения очищаются. Это не только вопрос безопасности, но и предотвращение накопления бесполезных данных. Данный лимит следует учитывать: если пользователь планирует долго быть оффлайн (например, отпуск > 7 дней), есть риск, что присланные ему сообщения устареют и не дойдут. Однако подобная политика оправданна – лучше уведомить отправителя об истечении, чем хранить большой объем данных бесконечно.

Во-вторых, введены ограничения на размер и частоту сообщений по WS. Максимальный размер одного WS-пакета – около 8 МБ. Если клиент попытается отправить больше, сервер закроет соединение с кодом 1009 (Msg Too Big). Это предотвращает перегрузку памяти и каналов (8 МБ достаточно даже для пересылки изображений или голосовых). Частота исходящих сообщений также лимитируется: не более ~180 сообщений в минуту на одно соединение (т.е. в среднем 3 сообщения в секунду). Сервер локально считает сообщения и при превышении лимита разрывает соединение с кодом 1013 (Rate Limit Exceeded). Таким образом, флуд одним клиентом будет пресечён. Эти меры прямо направлены против DoS-атак и перегрузки сервера. В тестовых условиях они должны сохранить работоспособность даже при попытках спама.

В-третьих, Firebase push-уведомления тоже работают с ограничением частоты, чтобы не спамить пользователя. В коде реализован cooldown: один push на пользователя раз в 20 секунд (для сообщений) или раз в 1,2 секунды для звонков (критично более быстро). Кроме того, пуши звонков схлопываются по ключу pc_call_ – несколько приглашений подряд не будут приходить как отдельные уведомления, если предыдущий ещё висит. Эти нюансы влияют на UX, но также экономят ресурсы (в частности, Firebase имеет ограничения по числу пушей).

Поведение при пиковых нагрузках: Благодаря ограничениям, описанным выше,ProtoChatX устойчив к перегрузке от одного клиента, однако могут быть сценарии, когда много клиентов ведут интенсивный трафик. Например, «час пик» – все пользователи онлайн и активно обмениваются сообщениями. Узкими местами могут стать:

Процессор/событийный цикл Node.js: Основная работа relay-сервера – парсинг JSON, несколько сравнений и файловые операции. Парсинг JSON выполняется в блоке try/catch, некорректные JSON сразу отвергаются. Эти операции быстры и вряд ли перегрузят CPU. Файловые операции (чтение/запись JSON-файлов) более ощутимы – Node выполняет их синхронно (блокируя поток). Если, скажем, 100 сообщений одновременно приходят оффлайн разным получателям, сервер сделает 100 последовательных записей в [redacted]. Размер каждой записи небольшой, но суммарно задержка может накопиться. Логирование в консоль тоже блокирует поток, однако в коде предусмотрен фильтр логов (подменённый console.log), отбрасывающий информационные сообщения (AUTH_OK, SEND и т.д.) и выводящий только важные ошибки. Это снижает нагрузку на вывод и защищает приватность (соединения/ID не логируются без надобности). В целом Node.js способен держать тысячи одновременных WS-соединений, так что узким местом станет не CPU, а I/O при большом количестве операций записи.

Память: Сервер держит в памяти структуры users, invites, queue. Они все пропорциональны числу пользователей и отложенных сообщений. Если, гипотетически, пользователей станет десятки тысяч и у каждого по сотне оффлайн-сообщений, [redacted] в памяти может стать довольно большим

(сотни тысяч записей). Node.js справится с несколькими десятками мегабайт данных в памяти, но время чтения/записи файлов столь большого объёма может быть значительным (сотни миллисекунд). Пока таких масштабов не планируется, поэтому проблем нет. На практике же администратор, вероятно, создаёт пользователей вручную или через инвайты, что уже ограничивает рост (нет открытой регистрации, исключающей наплыв ботов).

Пропускная способность WS: Каждое сообщение передаётся целиком получателю. Если пользователи начнут активно слать большие вложения (до 8 МБ), то канал сервера может стать загруженным. Допустим, 10 пользователей одновременно отправляют файлы по 8 МБ – сервер должен передать ~80 МБ данных по сети. Это требования к исходящей полосе. Сам Node.js способен стримить бинарные данные, но нужно убедиться, что хостинг выдержит. Для смягчения, клиенты могли бы накладывать дополнительный лимит на размер файлов (например, 5 МБ), но по исходникам видно лимит ~8 МБ, что приемлемо.

Логирование и отслеживание ошибок: В ProtoChatX реализована система тегов логов: почти все console.log вызываются с меткой (например, log(now(), 'AUTH_OK', userId, ...)). Функция console.log была переопределена, чтобы скрывать чувствительные поля: например, для событий AUTH_OK, DISCONNECT, SEND и др. логирование отключено. Это сделано ради приватности, чтобы в серверных логах не копились метаданные общения (кто с кем соединился, кому что отправлено). В случае ошибок, наоборот, важные сообщения выводятся: например, [redacted]_SEND_ERR выводит код ошибки отправки пуша (но без раскрытия userId), QUEUE_WRITE_ERR логируется (с ID пользователя, у которого не удалось записать очередь). Такая селективная фильтрация логов – интересное решение, ориентированное на безопасность. Это снижает объём логов и риск утечки метаданных, но осложняет отладку, если нужно детально проследить взаимодействия. В целом, для production-режима это оправдано. Разработчик упоминает, что реле логирует лишь состояние сервиса и реальные ошибки – то есть, логи минимальны.

Обработка ошибок на уровне протокола: если клиент прислал невалидный JSON, сервер ловит исключение парсинга и отсылает обратно {type:'error', code:'BAD_JSON'}, не падая сам. Если при попытке записи файла возникла ошибка, она либо логируется, либо игнорируется безопасно (например, при проблеме с записью пуш-токена или обновлении pubkey, сервер не разорвет сессию, а просто проглотит исключение). Такой подход предотвращает крах сервера от неожиданных ситуаций – максимум, что произойдёт, данные не сохранятся, но процесс продолжит работать. Можно отметить, что критичные ошибки (типа невозможности прочитать [redacted] при старте) обрабатываются радикально: если [redacted] не установлен или слишком короткий, процесс завершается с ошибкой. Это верно – запуск без пароля админки невозможен, так лучше остановиться.

Rate limiting и защита от злоупотреблений: Кроме WS-сообщений, в Admin API тоже есть rate limit: не более 240 запросов в минуту с одного IP. Это

предотвращает перебор токена или спам API. Также все критичные методы требуют правильного X-Admin-Token заголовка, сравнение которого выполняется через timingSafeEqual для исключения атак по времени. [redacted]-пуши отправляются через библиотеку Firebase Admin SDK – там встроены свои retry[redacted] механизмы, но сервер дополнительно обрабатывает ответы: если отправка ок, логирует [redacted]_SENT, если ошибка – [redacted]_SEND_ERR с текстом ошибки. Можно заметить, что push-отправка выполняется асинхронно (через await), но не ожидается на основном потоке WS (в коде push запускается через void sendPushToUser в фоновом Promise). Таким образом, задержки при связи с серверами Google не блокируют основную логику – важный момент для стабильности.

Возможные узкие места: При значительно возросшем числе пользователей основные риски – задержки при логине из-за чтения больших [redacted] (обозначено выше) и задержки при массовой рассылке оффлайн-уведомлений. Например, если системе одновременно нужно уведомить 100 оффлайн-пользователей о сообщении, она вызовет 100 отправок [redacted] почти одновременно. Firebase, как правило, справится, но это создаст всплеск нагрузки. На практике, 100 push – не проблема, но тысячи могли бы требовать более сложной очереди/балансировки. Пока проект не оперирует такими масштабами, можно считать его стабильным.

Стабильность бекенда: Ни в обзоре кода, ни в поведении системы не обнаружено утечек памяти или ситуаций, которые бы приводили к краху. Node.js хорошо справляется с долгоживущими соединениями, а явная очистка старых данных (например, удаление старых сообщений) помогает держать расход памяти в узде. Отдельно стоит отметить, что WS-соединения поддерживаются heartbeat-механизмом: сервер каждые 30 секунд рассылает пинг всем подключённым клиентам и помечает флаг isAlive; если за интервал не приходит pong, соединение закрывается как разорванное. Это важно для обнаружения зависших соединений и освобождения ресурсов.

В целом, производительность ProtoChatX для его целевого использования достаточна. Он не рассчитан на огромные объёмы трафика или массовый публичный запуск, но в рамках нескольких сотен пользователей и умеренной активности работает устойчиво. Серверные ограничения (очереди, размер, частота) эффективно предотвращают аномальные нагрузки и злоупотребления. Разработчики продемонстрировали внимание к деталям: использованы безопасные сравнения для токенов, ограничены все потенциальные точки перегрузки, заложены таймауты и TTL. Для повышения стабильности в будущем можно порекомендовать мониторить использование CPU/RAM при росте числа пользователей и, при необходимости, оптимизировать «тяжёлые» места (например, заменить синхронные file I/O на асинхронные, или кешировать [redacted] в памяти между логинами, обновляя инкрементально). Но на данный момент, исходя из анализа, Backend ProtoChatX достигает приемлемой стабильности – соблюдая требования TLS, защиты файлов и секретов, можно эксплуатировать его как защищённый корпоративный мессенджер без существенных опасений.

UX/UI аудит

Регистрация и вход: Пользовательский опыт при первом запуске ProtoChatX продуман с уклоном в простоту, насколько это возможно без ущерба безопасности. При установке приложения пользователю не нужно вводить номер телефона или email – только инвайт-код, желаемый ID и никнейм. Инвайты применяются для ограничения регистрации: новый аккаунт можно создать, только имея действующий пригласительный код, что предотвращает наплыв случайных либо вредоносных регистраций. Ввод собственного ID не обязателен – можно оставить поле пустым, и сервер присвоит следующий доступный номер. Никнейм – любое уникальное имя (проверяется на занятость). Приложение генерирует криптографическую пару ключей автоматически, в фоне, без участия пользователя. После этого аккаунт создаётся, и пользователь получает на экране свой ProtoChatX ID (цифровой идентификатор) и секрет ([redacted]) для входа. В UI предусмотрено копирование этих данных – есть кнопки «Copy [redacted]», «Copy invite message» и т.д. в админ-панели, а на стороне клиента, вероятно, тоже даётся возможность сохранить секрет. Важное замечание: по задумке, пользователь не вводит [redacted] вручную при каждом входе. После регистрации приложение сразу авторизуется на relay-сервере (получив по redeem_invite ответ с user_id и [redacted], клиент вызывает auth и устанавливает сессию). Приложение может запомнить [redacted] локально (в защищённом хранилище или хотя бы в памяти), чтобы не мучить пользователя вводом сложной строки. Таким образом, вход в приложение после регистрации происходит автоматически и прозрачно – далее при открытии приложения сессия устанавливается без лишних действий, пока [redacted] сохранён. Если пользователь специально не выходит (в ProtoChatX нет понятия пароля, только [redacted], но можно реализовать локальный PIN для защиты, такого пока не отмечено), UX сопоставим с мессенджерами типа Telegram, где после первичной регистрации дальнейшие использования не требуют авторизации.

Восстановление аккаунта – UX: Процесс recovery, описанный в секции безопасности, конечно, не так прост, как восстановление по SMS, но UI сделан максимально понятным. При регистрации показывается окно с recovery-кодом и пояснением, что его нужно записать и бережно хранить. Вероятно, интерфейс не позволит пропустить это предупреждение без подтверждения. Действительно, документация шагов для пользователя подчёркивает: «ProtoChatX покажет вам код восстановления аккаунта. Обязательно запишите этот код… Этот код – ваша страховка на случай потери доступа к устройству». Такое акцентирование – хорошая практика UX для безопасных приложений (аналогично кошелькам криптовалют, где просят сохранить seed-phrase). Если пользователь спустя время утратил доступ, на экране логина (или отдельном экране) предусмотрена опция «Восстановить аккаунт», где надо ввести свой ID и один из сохранённых recovery-кодов. Этот сценарий хоть и менее удобен, чем «забыл пароль – отправить письмо», но соответствует уровню безопасности: нельзя сделать recovery через email, ведь email не привязан, а создание

backdoor для восстановления по запросу противоречит принципам ProtoChatX. В FAQ честно говорится: «Если потерял recovery-код – к сожалению, восстановить доступ невозможно… придётся создать новый аккаунт». Для UX это сурово, но хотя бы пользователю сообщают об этом прямо. В качестве улучшения UX можно предложить реализовать второй фактор восстановления – например, опциональную привязку электронной почты сугубо для восстановления. Это не обязательный шаг, но для некоторых пользователей мог бы служить «бэкапом бэкапа». Однако с точки зрения концепции полной автономности (никаких привязок) разработчики сознательно от этого отказались.

Приглашения и добавление контактов: Отсутствие глобального поиска по номерам или email означает, что добавление контакта происходит вручную. После регистрации пользователю, чтобы начать чат, нужно знать никнейм или ID собеседника. Предположительно, в UI есть поле или кнопка «Добавить контакт», куда вводится ник или ID друга. Сервер поддерживает поиск по нику: если ник уникален (а система этого требует), то клиент может запросить ID по нику (в коде resolveNickToId возвращает id по полному совпадению никнейма, либо ошибку если не найден или не уникален). Значит, UX вероятно таков: пользователь вводит ник – если найден уникальный профиль, он добавляется в список контактов. Либо другой сценарий – администратор раздаёт всем участникам инвайты и списки ID, и они сами обмениваются. В любом случае, ProtoChatX предполагает более сознательных пользователей, готовых вручную сообщать свои идентификаторы. Плюс – никаких синхронизаций с адресной книгой или раскрытия контакт-листов (что хорошо для приватности: «список друзей хранится только у вас»). Минус – для массового пользователя этот процесс менее привычен, чем просто выбрать номер телефона. Тем не менее, для аудитории, ценящей приватность, такой UX приемлем.

Интерфейс чатов и вызовов: Интерфейс приложения во многом напоминает типичные мессенджеры, что снижает порог вхождения. Белая книга отмечает: «Интерфейс и опыт использования не сложнее, чем у любого популярного мессенджера. Не нужно разбираться в криптографии… безопасность по умолчанию, а не за счёт удобства». Пользователю доступны стандартные функции: отправка сообщений, получение уведомлений, аудио/видео-звонки – всё через привычные элементы UI. Например, чтобы позвонить, достаточно нажать кнопку звонка в чате с контактом, как и в других приложениях. Собеседник получит оповещение о входящем звонке и сможет принять – интерфейс интуитивно понятен, особенно если вы пользовались звонками в других программах. Важное отличие: звонок проходит напрямую и зашифрованно, без привязки к телефонной сети, но это от пользователя не требует никаких дополнительных действий или знаний (приложение само «договаривается» о защищённом канале). После завершения звонка приложение может отобразить локально факт звонка (например, «Вчера был звонок с таким-то контактом»), но эта информация остаётся только у пользователя, нигде больше. Таким образом, UX звонков максимально приближен к обычному – ответить/отклонить, говорить через громкую связь

или наушники (для этого в коде есть PCAudioRoutePlugin, вероятно отвечающий за переключение аудиовыхода). Приложение интегрируется с системным экраном вызова на Android: при звонке, когда приложение может быть в фоне, на весь экран показывается системный UI с кнопками «Ответить/Отклонить» (как при обычном телефонном звонке). Это достигнуто через пуши типа call и соответствующий IncomingCallService с Notification Channel для звонков. Пользователь видит привычный звонковый экран, может ответить прямо с локскрина. При ответе, под капотом, CallActionsReceiver ловит нажатие и передаёт приложению команду начать звонок. В целом, это грамотное улучшение UX – звонки действительно работают «как обычно», не требуя, чтобы приложение было на переднем плане.

Навигация и чаты: ProtoChatX – однооконное приложение, то есть весь интерфейс состоит из одного HTML-скрина, где есть области: список чатов/контактов, окно текущего чата, поле ввода, возможно меню. В стилях видно, что предусмотрена адаптивность под узкие экраны: например, при очень узких экранах скрывается элемент #dialogPill и #who (видимо, часть заголовка). Для телефонов это полезно: интерфейс подстраивается, убирая лишнее, чтобы поместилось основное. Также есть отступы с учётом safe area insets на iPhone и др. (стили .topbar-inner { padding-right: env(safe-area-inset-right) ... } есть в коде), что говорит о тщательной проработке мобильной адаптации. Переключение между чатами реализовано, скорее всего, через список контактов или недавних чатов (возможно, «контакты» и «избранные» разделы). В коде найдены упоминания элементов elContactsList и elFavs – вероятно, список контактов и список избранных (любимых) контактов, которые отображаются отдельно. Есть логика, что при отсутствии избранных отображается текст "Пока пусто. Долгое удержание на сообщении → □" – намёк, что можно долгим тапом на сообщении добавить собеседника в избранное. Это приятная мелочь UX: можно отмечать важные контакты, чтобы быстрее их находить. Список всех пользователей (с их pubkey) приложение получает при авторизации (auth_ok содержит users: [...]), но в интерфейсе явно не будут показываться все – скорее, только те, с кем общался или кого добавил.

Безопасность в интерфейсе: Одной из UX-задач подобных мессенджеров является донести уровень безопасности без перегрузки пользователя. ProtoChatX делает это достаточно прозрачно: шифрование всегда включено и нигде не требует действий. Нет необходимости вручную подтверждать ключи или вводить пароли на звонок – всё происходит автоматически. Для пользователя мессенджер ощущается как обычный – пишешь сообщение и отправляешь, «как обычно», но при этом всё уже защищено. В UI, вероятно, есть индикаторы соединения (например, значок замка или надпись в about, что чат зашифрован сквозным образом). В коде прослеживаются элементы для статуса сети: #status .net-signal i – видимо, значок с полосками связи, как у сигнала сотовой сети. Действительно, #status[data-net="online"] показывает «палки» сигнала в зависимости от качества (их анимация pulse), а если offline – вместо них значок (net-img) показывается. Таким образом, пользователь всегда видит, онлайн он или нет, и качество связи. Это важно для мессенджера: чтобы понимать, доставляются ли сообщения. Безопасность соединения –

подразумевается, что всегда WSS, поэтому отдельного индикатора шифрования канала, скорее всего, нет (как замочек, например, не нужен – всё через TLS). Конфиденциальность интерфейса: Возможны функции вроде блокировки скриншотов (в коде не видно, но на Android можно включить флаг WindowManager.FLAG_SECURE для окна, чтобы запрещать скриншоты). Неизвестно, сделали ли это – если нет, можно порекомендовать, хотя не всем пользователям это понравится. Темная тема UI по умолчанию тоже способствует приватности (сложнее подсмотреть через плечо). Кстати, тёмная тема реализована: в стилях определена цветовая палитра --pc-bg, --pc-surface и т.д. с тёмными значениями (#0B1016 и т.п.). В коде упоминается некий класс body.tg – возможно, переключение темы (например, .tg – Telegram?), но скорее всего, ProtoChatX сразу идёт в тёмной теме (фон #070A0E). UI выполнен в сине-тёмных тонах (цвет текста #eaf2ff – почти белый), что характерно для приватных мессенджеров (Signal, Telegram night mode). Это не только эстетика, но и меньшая заметность экрана в тёмное время, что тоже плюс конфиденциальности.

Элементы управления и механики: По фрагментам HTML можно оценить, что в ProtoChatX есть собственные реализации всплывающих окон, тостов и модальных диалогов вместо системных. Например, есть стили для .pcx-pw-modal – окна с вводом пароля и .pcx-pw-actions (кнопки OK/Cancel). Это может быть связано с функцией «скрытые чаты» или «между строк». В стилях упоминается Hidden ("Между строк") dark theme и .pc-hidden-input – вероятно, существует функция скрытия сообщений «между строк», когда сообщение отображается как пустое место, а для его просмотра нужен пароль. Косвенно об этом говорит наличие #hiddenRow и #pwAskModal/#pwSetModal в коде. Похоже, ProtoChatX позволяет пользователю задать пароль на конкретный чат или сообщение (возможно, режим «секретного чата» внутри основного). Это крутая, но несколько hidden (каламбур) возможность. UX этого: пользователь выбирает «скрыть сообщение», вводит пароль – тогда на экране вместо текста будет, например, пунктирная рамка или пустота, а чтобы прочитать, надо нажать и ввести пароль. Такие механики повышают приватность при совместном использовании устройства или записи экрана. Однако они требуют сложного UI, и судя по стилям, реализованы аккуратно (модальные окошки с темной подложкой, поле ввода пароля, описание). Вероятно, toast-уведомления (всплывающие короткие сообщения) тоже свои: в стилях .pc-toast и функция pcToast(text, ms). Это значит, никаких системных alert/confirm – всё кастомное. Например, при копировании recovery-кодов в буфер может появляться toast «Скопировано».

Интуитивность интерфейса: Хотя приложение насыщено безопасными функциями, стараются не перегружать пользователя ими. Пользователю не нужно знать про JWK, ECDH или SALT – всё это «под капотом». В обычном использовании интерфейс действительно похож на современные мессенджеры: список чатов, сообщения с пузырями, статус доставки (вероятно, одна галочка – отправлено, две – доставлено, цвет – прочитано? Если реализовано. Отправителю сервер возвращает sent со статусами queued/delivered, которые можно трактовать как галочки). Наличие push-уведомлений делает UX

полноценным: приходят оповещения о новых сообщениях даже если приложение свернуто, причём без утечки содержания (в уведомлении просто «Новое сообщение»). Пользователь может настроить в системе, как обычно, показывать или скрывать текст (здесь и показывать нечего, он и так скрыт сервером).

Мобильная адаптация: Как отмечено, интерфейс писался с расчётом на мобильные экраны. Capacitor обеспечивает кроссплатформенность (можно и на iOS запустить). Управление через тач: присутствуют упоминания долгого удержания (для избранного). Вероятно, свайпы или другие жесты – можно было бы использовать для навигации (например, свайп вправо – назад к списку чатов). Неизвестно, реализовано ли это. Но судя по тому, что .left > * { text-overflow: ellipsis }, в заголовках контролируется переполнение, UI сделан адаптивным.

Темная/светлая тема: По файлам создаётся впечатление, что светлой темы нет, только тёмная (возможно, светлой просто не сделали, что не критично). Для многих пользователей тёмная даже предпочтительна. Если вдруг нужен светлый режим, это легко добавить – но с точки зрения приватности тёмный лучше.

Локализация интерфейса: Судя по текстам в PDF и коде (Новое сообщение, выделено, Пока пусто и т.д.), UI сейчас локализован на русском языке (что логично, проект, видимо, изначально для русскоговорящих). Это значит, что англоязычному пользователю придётся или ждать перевода, или пользоваться как есть. Для аудита это не проблема, но с точки зрения UX – желательно иметь как минимум английский язык. Вероятно, планируется, ибо код содержит отдельные строки текста, которые можно вынести в словарь для локализации.

Summing up UX: ProtoChatX удачно балансирует безопасность и удобство. Пользователю не нужно думать о ключах – они генерируются и распространяются автоматически. Звонки и чаты работают привычно. Ограничения (одновременный вход с одного устройства) явно обозначены: если попробовать зайти вторым устройством без recovery, не получится – приложение сообщит об ошибке pubkey mismatch (возможно, через alert). Да, отсутствует «мультидевайс», что для UX минус: нельзя параллельно сидеть с телефона и компьютера. Но это осознанная жертва ради безопасности данных (сложность синхронизации E2EE на нескольких устройствах, плюс риск компрометации). В FAQ прямо говорится, что одновременно можно быть онлайн только на одном устройстве, и старое устройство теряет доступ при переносе аккаунта. Для целевой аудитории (безопасное общение) это приемлемо – например, многие пользователи Signal или Threema тоже часто используют только на телефоне. При необходимости можно выпустить десктоп-версию ProtoChatX, но это потребует либо отдельного аккаунта, либо импорта ключей (что небезопасно). В общем, UXProtoChatX ориентирован на мобильное единоличное пользование.

Дополнительные функции UI: В коде видны указания на вложения (attachments) – функция отправки файлов/картинок, а также голосовые сообщения. Например, VoiceNativePlugin показывает, что пользователь может записать голосовое сообщение: при нажатии кнопки записи приложение запросит разрешение на микрофон и запишет аудио в .m4a файл, затем вернёт base64 строку, которую клиент вставит как attachment в сообщение. После отправки голосовое – это просто файл, он шифруется так же, как другие (в коде упоминается, что файлы и голосовые перед отправкой превращаются в base64 строку, затем E2EE-шифруются). UI для голосовых, видимо, включает полоску воспроизведения (в CSS есть .pc-voice и waveform стили, судя по коду plugin'а, он даже генерирует waveform). Это делает приложение функционально ближе к современным мессенджерам, не жертвуя безопасностью.

Инвайты и удобство приглашения друзей: Админ может скопировать «invite message» – возможно, автоматическое сообщение с инструкциями и кодом. То есть, администратор может прислать новому пользователю текст вида: «Привет, установи ProtoChatX и введи этот код: XXXX-XXXX …». Это упрощает процесс для новичка. После ввода инвайта у юзера и так всё происходит быстро.

Подытоживая, UX/UI ProtoChatX выполнен на высоком уровне для приложения такого класса. Разработчики явно заботились, чтобы безопасность была "по умолчанию" и не требовала от пользователя лишних действий или знаний. Интерфейс интуитивно понятный, похожий на известные мессенджеры, поэтому переход для пользователя минимален. Отдельные сложные моменты (recovery-коды) сопровождаются чёткими инструкциями. Визуальный дизайн современный: темная тема, плавные элементы, иконки (в коде есть, напр., □ иконка звонка вставляется). Использование toast вместо system alert делает взаимодействие более мягким и брендированным. Все эти детали – индикатор хорошего UX-подхода даже при сильном акценте на приватность.

Уязвимости и потенциальные угрозы

Несмотря на общую высокую защищённость, рассмотрим возможные векторы атак и слабые места, требующие внимания:

Операционная безопасность сервера: Как подчёркивается в документации, безопасность ProtoChatX во многом зависит от правильного развёртывания и защиты окружения. Критично, чтобы файлы [redacted], [redacted], [redacted] на сервере были недоступны посторонним (например, через права ОС, или если сервер публичный – вынести их за пределы веб-доступа). Если злоумышленник получит эти файлы, он узнает все хеши инвайтов и recovery-кодов, id и ники пользователей, а главное – plaintext [redacted] каждого пользователя. Хотя без соответствующих приватных ключей эти [redacted]s бесполезны для входа (сервер не пустит с другим ключом), утечка [redacted] всё равно критична: атакующий мог бы, например, отключить пользователю MFA (подменив ключ на свой через Recovery, если бы имел recovery-код – но

recovery-коды хранятся как хеши). В общем, прямого мгновенного компрометации переписки утечка [redacted] не даст (ключей приватных там нет), но даст много информации. В первую очередь – [redacted]-токены пользователей, которые там хранятся открыто. [redacted]-токен сам по себе – не секрет для злоупотребления (случайный идентификатор устройства), но он уникален и потенциально может быть использован для слежения (Google/Firebase или злоумышленник с доступом к токену мог бы попытаться ассоциировать пользователя с устройством). Для безопасности, важно, чтобы [redacted] был хорошо защищён, и желательно зашифрован на диске либо хотя бы находился в закрытой серверной сети.

[redacted] и панель админа: Административный интерфейс – потенциальная цель атак. По умолчанию, он слушает на всех интерфейсах ([redacted]:[redacted]) без SSL, то есть если оставить его доступным из интернета, злоумышленник может попытаться подобрать токен. Защита здесь: требование токена >=12 символов и rate limit 240 запросов/минуту. Брутфорс надёжного токена при таких ограничениях практически нереален (пространство огромно). Но если администратор поставит слабый токен (на что есть предупреждение в логах), или если API окажется доступен на HTTP без TLS, то возможны проблемы. CSRF-атаки на Admin API затруднены, т.к. запросы требуют заголовок X-Admin-Token – невозможно незаметно заставить браузер отправить такой (не считая, если сам админ откроет зловредную страницу в браузере, где уже открыт [redacted] и сохранён токен в localStorage: скрипт теоретически мог бы сделать запрос на API используя сохранённый токен). Но admin UI – одностраничное приложение, а токен хранится в localStorage (если «запомнить» отмечено). Значит, возможно уязвимость XSS в админ-панели была бы очень опасна – злоумышленник мог бы похитить токен или выслать команды. Однако искать XSS в админке нет смысла – она доступна только админу. Главное, чтобы админ не вставлял в поле Note какой-нибудь скрипт, но listUsersSanitized фильтрует чувствительные поля, note возвращается как строка, вставляемая через DOM textContent (судя по реализации). Код админ-панели почти весь на клиенте (embedded HTML/JS), он не выводит неподготовленные данные от пользователей, так что XSS маловероятен. Тем не менее, рекомендовано ограничить доступ к [redacted] интерфейсу сетевыми средствами: например, слушать только на localhost или поставить базовую HTTP-авторизацию/VPN. Таким образом, единственный по-настоящему опасный случай – если кто-то получит сам [redacted]. Это может произойти, например, по неосторожности (настройка через ENV, но кто-то увидел в процессе, или сохранено в .bash_history). Нужно относиться к нему как к паролю от серверов.

AndroidManifest и permisos: Из Java-кода видно, что приложение запрашивает только два runtime-разрешения: CAMERA и RECORD_AUDIO. Скорее всего, в [redacted] также декларированы INTERNET, FOREGROUND_SERVICE (для входящего звонка), WAKE_LOCK (чтобы будить телефон для звонка), RECEIVE_BOOT_COMPLETED (если надо авто-рестарт push). Возможно, android:exported для компонентов. Критично проверить BroadcastReceiver CallActionsReceiver: он, судя по всему, объявлен для действий

ACTION_ANSWER, ACTION_DECLINE и ACTION_STOP с фильтром на них. Если он exported=true (по умолчанию, раз есть filter), то любое приложение в системе может послать Broadcast [redacted] с нужными extras. Это пробудит IncomingCallService.stop (что ничего страшного, просто остановит звук) и запустит MainActivity с extra, якобы «ответить» на звонок. В худшем случае злоумышленник может имитировать на устройстве нажатие «Ответить» – если в этот момент был реальный звонок, то он подключится. Однако без реального звонка это просто откроет приложение. То есть, потенциально, вредоносное приложение на телефоне могло бы попытаться вмешаться в звонки ProtoChatX, рассылая эти broadcast. Это низкий риск (для этого сам телефон должен быть заражён, а если так – злоумышленник и микрофон может записать). Тем не менее, можно усилить безопасность, установив android:exported=false или требуя permission для этого ресивера, чтобы посторонние приложения не могли его дергать.

Другой момент – Backup-Flag: Если в манифесте не отключён allowBackup, злоумышленник с физическим доступом к телефону мог бы через adb backup вытянуть данные приложения (включая [redacted] и приватный ключ, если он в IndexedDB файловой системе вебвью). Хорошей практикой для секьюрити-ориентированных приложений является android:allowBackup="false" в манифесте, предотвращающее резервное копирование приложения. Нужно убедиться, что это применено. Также, [redacted] (содержит ключи Firebase) включён в сборку – но эти ключи не секретные (они нужны клиенту для инициализации [redacted], ID отправителя и т.п.). Их компрометация не позволит отправлять push от чужого имени (для этого нужен серверный ключ Firebase, который хранится на backend). Так что тут нормально.

XSS в клиентском приложении: Так как клиент – это веб-страница, важно, как она отображает приходящие сообщения. Если злоумышленник отправит в чат строку вида , то при расшифровке она может интерпретироваться как HTML и выполнить JS. Защита здесь должна быть – либо контент сообщений экранируется, либо выводится как textContent. В коде видно использование функции esc() для экранирования текстов перед вставкой innerHTML. Например, имена полей, placeholder-ы, имена файлов проходят через esc() (которая вероятно заменяет < на < и т.д.). Для сообщений, скорее всего, сделано так же: либо весь лог собирается из безопасных элементов, либо текст прогоняется через экранирование. Признак: при добавлении сообщения, вызывается elLog.innerHTML = ... – там, в случае пустого чата, они вставляют

История пуста.

прямым HTML. Но это статический текст. Для пользовательских сообщений, возможно, используется шаблон, где сообщение подставляется в innerText или через esc(). Также, файл index.html содержит комментарий, что UI-элементы (картинки, аудио) генерируются через функции, которые sanitize MIME тип и имя файла. Видно, что sanitizeMime(mime) вызывается перед вставкой MIME. В общем, похоже, XSS предусмотрели. Критично, чтобы HTML-сообщения не

могли выполняться. Если даже какой-то хитрый случай проскочит – учтём, что весь код работает не в браузере, а в WebView, т.е. под контролем приложения. Он не имеет доступа к cookies (нет), но получив XSS, мог бы теоретически вызвать native-bridge и получить что-то. Однако, Capacitor, в отличие от Cordova, не предоставляет by default evalbridge, он работает через плагинный API. Так что риск XSS – не утечка наружу, а разве что нарушение локальной безопасности (например, воровать [redacted] из IndexedDB). Но [redacted] там можно достать и так, если кто-то завладел телефоном. Тем не менее, это стоит проверить при пентесте. На первый взгляд, разработчики старались избегать вставки непроверенного HTML, что хорошо.

Доступ к [redacted]-токенам: Когда пользователь логинится, он отправляет на сервер свой [redacted]-token (PushNotificationsPlugin получает токен и вызывает push_register). Сервер хранит [redacted]_token в открытом виде. Если злоумышленник получил доступ к этой базе, он узнал token каждого устройства. Сам по себе [redacted] token можно использовать для попытки слать уведомления через Firebase API – но для этого нужен Firebase Server Key, который неизвестен злоумышленнику (он внутри firebase-admin на сервере, и к нему доступа нет без компрометации сервера целиком). Токен можно теоретически использовать для слежки: Google, зная токен, знает, какое устройство (но у Google и так есть mapping). Угрозой могло бы быть массовое отзыва токенов – но Firebase не даёт отозвать токен постороннему, только сам клиент может. В резюме, утечка [redacted]-токенов – не большая брешь, но требует хранить [redacted] в секрете (см. выше).

Пользовательские данные в памяти устройства: Приватные ключи хранятся в IndexedDB внутри WebView. Обычно IndexedDB лежит в хранилище приложения (внутренний каталог) и защищён ОС (доступен только самому приложению). Однако, на рутованном телефоне или при атаке типа Pegasus (spyware) – возможно извлечь и даже подменить эти данные. ProtoChatX, как приложение с высокой безопасностью, не упоминает механизмов вроде hardware keystore (для хранения приватника). Они могли бы использовать AndroidKeyStore через плагин – пока непонятно, реализовано ли. В тексте есть фраза «приватный ключ хранится только на клиенте (в IndexedDB, помечен как неэкспортируемый)». "Неэкспортируемый" вероятно значит, что Web Crypto API сгенерировал его с флагом non-extractable, т.е. получить битовую копию ключа из JS невозможно (он хранится в контейнере браузера). Это хорошо: даже если XSS, злоумышленник не сможет просто взять ключ – WebCrypto не отдаст raw ключ. Можно попытаться генерировать подписи, но на выход токен key не выдаётся. Это сильно снижает вероятность кражи ключа даже при XSS. Но если у злоумышленника есть физический доступ или malware на телефоне – увы, он может дождаться, когда пользователь разблокирует приложение (так как Key non-extractable, но malware может вмешаться на уровне JavaScript внутри WebView, если овладел устройством). От подобных угроз защищают скорее меры ОС (например, Protected Confirmation или т.п.) – но это уже экстремум.

Поверхностные web-уязвимости: В фронтовом коде были найдены упоминания ver_req, ver_info – возможно, не до конца реализованный функционал проверки ключей. Там важно, чтобы никто не мог отправить crafted пакет и вызвать у другого UI-диалог без контекста (хотя это не уязвимость, скорее inconvenience). CSRF для обычных пользователей: так как клиенты общаются только по WebSocket (а он не подвержен CSRF из браузера, потому что другой сайт не может создать WS на произвольный порт без CORS), классические CSRF атаки не применимы.

Фишинг и социальная инженерия: Это стоит упомянуть: ProtoChatX не защищён от того, что пользователь сам выдаст свой [redacted] кому-то. Если злоумышленник убедит пользователя ввести [redacted] на фейковом сайте или приложении, тот получит доступ. Однако, даже получив [redacted], как отмечалось, злоумышленник не войдёт в аккаунт, не имея приватного ключа – сервер его не пустит (pubkey mismatch). Но если злоумышленник параллельно обманом выманит и recovery-код, он сможет произвести recovery на своё устройство (тут достаточно знать ID жертвы и один recovery-код, чего крайне не хочется допустить). В реальном мире против этого – только обучение пользователя, чтобы он никому не сообщал [redacted] и recovery. Приложение делает акцент на сохранности recovery, но возможно, стоило бы при авторизации [redacted] выводить предупреждение вроде «Никому не сообщайте этот [redacted]» (в админ-панели, когда создаёшь пользователя, отображается [redacted]: … и, вероятно, предполагается, что админ передаст это безопасным каналом).

Вывод по уязвимостям: Существенных технических уязвимостей в ProtoChatX не обнаружено – архитектура тщательно проработана с учётом типичных атак. Основные риски лежат вне самой бизнес-логики, а в окружении: компрометация сервера, неправильная настройка (открытый admin-интерфейс, отсутствие TLS), слабый операционный контроль. Документация указывает, что WS-соединение должно проходить только по TLS, а файлы должны быть недоступны посторонним, иначе вся приложенная криптография может быть обойдена организационно. В общем, «пробить» ProtoChatX стандартными методами сложно – ни MITM, ни brute-force, ни DoS (серьёзный) ему не страшны. Тем не менее, мы рекомендуем провести полноценное пентестирование клиента, особенно на предмет XSS и безопасности хранения на устройстве (возможно, стоит задействовать Android SafetyNet или зашифровать локальное хранилище ключей).

Рекомендации и улучшения

На основе проведённого аудита можно предложить ряд конкретных улучшений, затрагивающих безопасность, производительность, UX и архитектуру ProtoChatX:

Улучшения безопасности:

Хеширование секретов на сервере: Реализовать хранение [redacted] пользователей в виде хешей (bcrypt/Argon2). Это повысит устойчивость при компрометации [redacted] – злоумышленнику придётся взламывать хеши, вместо прямого использования секретов. В проектной документации это уже предложено как опция улучшения. Можно применить bcrypt с индивидуальной «солью» (например, ID пользователя) и общей pepper. Входящих [redacted] при логине хешировать и сравнивать. Да, это добавит немного CPU на логин, но для небольшого числа пользователей не критично, а выгода – значительная безопасность данных при утечке базы.

Аппаратное хранилище ключей (клиент): Рассмотреть использование Android Keystore для хранения приватного ключа или шифрования IndexedDB. Сейчас приватный ключ помечен non-exportable (WebCrypto), что хорошо, но на рутованном устройстве всё равно можно вытащить IndexedDB-файл. Если ключ будет создан/храниться в KeyStore (например, через плагин или native код), его кража станет крайне сложной. Либо хотя бы шифровать содержимое IndexedDB на уровне приложения, используя hardware-backed ключ.

Отключить backup и отладочные флаги: Убедиться, что в AndroidManifest стоит android:allowBackup="false", android:fullBackupContent="false" и приложение помечено как debuggable="false" для production. Это предотвратит утечку данных приложения через резервное копирование или отладку.

Ограничить broadcast-приёмники: В манифесте для CallActionsReceiver установить android:exported="false" (если возможно) или добавить фильтр по permission (чтобы только системное уведомление могло отправить intents). Это убережёт от попыток внешних приложений вмешаться в логику звонков.

Отслеживание компрометации устройств: Это сложная задача, но можно внедрить device binding – проверять на сервере fingerprint устройства. Например, при Recovery или при повторном входе можно фиксировать, что pubkey сменился (сейчас есть), но можно и иную информацию – версию приложения, OS. Если вдруг один аккаунт часто меняет ключи или приходит с подозрительных сред, это сигнал администратору. Такой функционал скорее опционален, но для корпоративной среды может быть полезен (хотя ProtoChatX open-source, каждый развёртывает сам).

Code-signing: Убедиться, что приложение подписано надежным сертификатом, и внедрить механизм проверки целостности (SafetyNet / DeviceCheck) при логине. Это не панацея, но затруднит работу модифицированных клиентов. В open-source проекте сложно навязать, но если аудит для конкретной организации, можно рекомендовать.

Улучшения производительности и масштабируемости:

Переход на асинхронное хранение или БД: Если планируется >500 пользователей или интенсивный трафик, стоит заменить JSON-файлы на встроенную БД (SQLite). Это даст конкурентный доступ и индексацию. Альтернатива – хотя бы асинхронные file I/O, чтобы Node.js поток не

блокировался. Сейчас fs.writeFileSync используется для простоты, но Node имеет и неблокирующий fs.writeFile. Можно переписать writeUsersFile, writeQueueFile с callback/await. Это позволит обрабатывать другие WS-события параллельно с записью на диск, что при высокой нагрузке снизит лаги.

Кэширование [redacted] в памяти: Сейчас при каждом логине файл перечитывается. Можно держать объект users в памяти и обновлять его при изменениях, а при логине просто обращаться к нему. Файл перечитывать только при старте и при админских действиях, которые происходят редко. Это устранит дисковую операцию на каждый логин. Нужно будет аккуратно обеспечить сброс кэша, если external change (в данном случае, [redacted]).

Пакетная доставка push: Если в один момент должно быть отправлено много пуш-уведомлений, их можно группировать. Firebase позволяет отправлять multicast (несколько токенов в запросе). Сейчас отправляется по одному. Если масштаб вырастет, можно реализовать очередь пушей и собирать несколько в один батч. Однако, возможно, overengineering для данного проекта – при нормальных условиях редки случаи десятков оффлайн-получателей сразу.

Мультисерверность: Подготовить путь к кластеризации. Если вдруг решат использовать ProtoChatX на тысячи пользователей, потребуется запуск нескольких экземпляров. Шаг 1 – отделить state от процесса (использовать внешнюю БД/кеш). Шаг 2 – иметь механизм доставки сообщений между экземплярами (например, Redis pub/sub, если два пользователя на разных нодах – одна нода должна отправить другой). Это серьёзная переработка, но можно отметить, что ProtoChatX по замыслу не идёт вширь, а остаётся автономным узлом.

Логирование в файл: Сейчас лог выводится на консоль. Для удобства эксплуатации, особенно на сервере, стоит записывать логи в файл (с ротацией по дням). Можно использовать winston или просто fs.appendFile для событий. Это упростит анализ сбоев. При этом надо убедиться, что в логах нет чувствительных данных (сейчас они отфильтрованы, что правильно). Возможно, сделать уровень логирования настраиваемым: обычный режим – приватный (минимум данных), debug-режим – подробный (для отладки, но предупреждать про риск утечки).

Оптимизация flush очереди: Функция flushOfflineQueue шлёт сообщения получателю одно за другим синхронно. Если очередь большая (200 сообщений), это может занять время. Можно отправлять порции или запускать отправку асинхронно (но тогда соблюдать порядок). В WebSocket один клиент способен принять пачку подряд, так что, вероятно, норм. Но на всякий случай, если latency большая, можно при flush добавить небольшие задержки между сообщениями, чтобы не перегружать клиента.

Групповые чаты: Это уже добавление функциональности – сейчас ProtoChatX поддерживает только индивидуальные чаты. Если понадобится групповой, надо менять модель (ключ на группу, множественная E2EE). Пока запросов на это нет, но в будущем можно рассмотреть как масштабирование

функциональности.

Рекомендации по UX/UI:

Локализация на английский язык: Для расширения аудитории стоило бы перевести интерфейс на английский (и сделать опции языка). Сейчас большинство текстов захардкожено на русском (что видно в PDF и коде). Это может отпугнуть зарубежных пользователей. Внедрение i18n позволит ProtoChatX выйти за рамки локального использования.

Верификация контактов: Добавить в UI возможность сверить отпечатки ключей с собеседником. Например, отобразить короткий fingerprint публичного ключа (12 символов уже вычисляется в admin-api как pub_fpr). Можно в профиле контакта показывать «Fingerprint: ABC123...». Пользователи, которым нужна абсолютная уверенность, смогут сравнить по другому каналу. Это необязательно, но как опция «Повысить уровень доверия» – пригодится.

Уведомление о смене ключа контакта: Если контакт восстановил аккаунт (сменил ключ), то его публичный ключ изменится. Сейчас при авторизации сервер шлёт актуальный список пользователей с их pubkey всем онлайн-клиентам. Возможно, приложение просто молча обновит ключ контакта. Лучше же отобразить маленькое уведомление в чате: «⚠ Ключ безопасности контакта изменился. Это может быть новое устройство собеседника.» Это повысит осведомлённость и позволит выявить возможную компрометацию (если, конечно, пользователь знает, что контакт не планировал менять устройство).

Biometric/PIN-пароль для приложения: Дополнительно к [redacted] можно ввести локальный PIN или разблокировку по отпечатку для входа в приложение. Сейчас, насколько можно судить, ProtoChatX не требует ничего при запуске (кроме первого входа). Для случая, если телефон разблокирован чужим, злоумышленник может открыть приложение и читать переписку (которая хранится локально). Имплементация App Lock (встроенный экран разблокировки с PIN/биометрией) решит этот риск. Многие мессенджеры (Telegram, Signal) имеют функцию блокировки приложения.

Светлая тема и настройки UI: Хотя темная тема – default, можно предусмотреть светлую для тех, кому она необходима (на солнечном свете, например). И также добавить настройку шрифта (размер для удобства) – благо, это несложно через CSS variables. Toast/модалки работают хорошо, их трогать не нужно.

Мультиустройство (в перспективе): Если появится требование подключаться с нескольких устройств, можно реализовать механизм «связанных устройств» как в Signal: генерируется дополнительный ключ для нового устройства, старый остаётся, и все сообщения шифруются на все устройства. Однако, это очень сложное изменение (требует протокола распределения ключей, синхронизации состояний). Пока от него отказались ради простоты и безопасности, и правильно сделали. Но если сообщество будет настаивать, можно подумать об отдельном «режиме», где [redacted] можно использовать

параллельно на двух устройствах – тогда сервер может хранить два pubkey на аккаунте (проверять совпадение любого). Это снизит безопасность (увеличит поверхность атаки), так что рекомендовать не будем, просто отмечаем как возможность.

Push-уведомления на десктоп: Как вариант развития – веб-версия. Сейчас клиент – HTML, можно попытаться сделать PWA (прогрессив веб) и запускать в браузере. Тогда потребовалась бы авторизация через [redacted] и backup код. Это опять-таки вне рамок Android UX, но учитывая что клиент уже HTML/JS, теоретически несложно. Стоит лишь удостовериться, что WSS соединение в браузере будет защищено (сертификат нужен). Пока user files/PDF не указывают на официальный веб-клиент, но если спросит – можно реализовать.

Рекомендации по DevOps/CI/CD:

Автоматизация развертывания: Для удобства можно подготовить Docker-контейнер с настроенной Node.js средой, что облегчило бы деплой сервера. В контейнер можно встроить опции по монтированию volume для data файлов, задаче [redacted], [redacted] и др. переменных. Это снизит риск ошибок конфигурации (кто-то забудет поменять [redacted] – а по умолчанию там "CHANGE_ME", что плохо). Можно изменить поведение: если pepper не изменён, сервер предупреждает (сейчас предупреждает лишь в консоли).

CI для сборки приложения: Настроить CI (например, GitHub Actions) для сборки APK и, возможно, проверки линтером/тестами. Это гарантирут, что каждый commit не ломает сборку. Также хорошо бы добавлять статический анализ (например, MobSF, SonarQube) для поиска потенциальных уязвимостей.

Обновление зависимостей: Убедиться, что зависимости (firebase-admin, ws и пр.) обновляются регулярно, желательно через Dependabot или аналог. Устаревшие пакеты могут иметь уязвимости.

Мониторинг и алертинг: В продакшн-среде стоит мониторить работу ProtoChatX: метрики CPU/RAM, пинги [redacted] (у admin-api есть [redacted] endpoint, возвращающий ok). Настроить алерты, если сервер упал или перегружен. Также логировать важные события безопасности – например, множество неверных логинов (BAD_[redacted]) можно считывать из логов и оповещать админа, что, возможно, идёт попытка подбора (хотя при [redacted]s 43 символа это маловероятно, но вдруг user1 трижды ввёл неправильный [redacted] – может, кто-то пытался).

Документирование процедур: В комплекте с тех. документацией полезно добавить гайд по безопасному администрированию: как часто ротировать [redacted], как бэкапить данные (вручную, так как нет автоматического; возможно, скрипт резервного копирования JSONов с шифрованием), как восстанавливать после сбоя. Это выходит за границы кода, но важно для эксплуатационной безопасности.

В заключение, подчеркнём: ProtoChatX уже является очень безопасным и добротно реализованным приложением для приватного общения.

Предложенные улучшения направлены на усиление защиты в крайних сценариях и повышение удобства, но даже без них уровень безопасности высок. Реализованные меры (E2EE, привязка к устройству, отсутствие нешифрованного хранения переписки, минимизация метаданных) делают ProtoChatX соответствующим заявленной миссии – приватный, автономный и простой мессенджер нового поколения. Следуя рекомендациям по эксплуатации (использовать TLS, защищать сервер от доступа, обучать пользователей хранить recovery-коды в тайне), вы значительно снижаете любые оставшиеся риски. ProtoChatX достиг своей цели защищённого мессенджера, и с малыми доработками может стать ещё лучше как в плане безопасности, так и удобства использования.