12. Реализация защиты от Replay-Attack и перехвата пароля в незащищенном соединении

В данной части я представлю своё видение на реализацию указанного функционала, оно не претендует на абсолютную правильность. Любой актуальный протокол имеет встроенный функционал, эквивалентный разрабатываемому.

Следует внести ясность о том, в чем состоит отличие шифрования от хеширования: шифрование подразумевает возможность обратной операции — расшифровки, хеширование — нет. В данной статье я применяю не совсем корректное понятие соли хеширования — «ключ» хеширования — естественно, добавочные байты не является «ключом» в привычном смысле, ведь ими нельзя ничего «открыть».

Replay-Attack является разновидностью атаки Man-In-The-Middle. Суть атаки, как можно догадаться из названия, состоит в простом перехвате и повторной отправке на сервер запроса настоящего клиента.



Например, пользователь пытается авторизоваться на сайте. Он отправляет на сервер POST-запрос по защищенному HTTPs соединению, в теле запроса передавая свой логин и пароль в виде простого текста. Злоумышленник перехватывает соединение, после чего дублирует вышеуказанный запрос. В данном случае не требуется расшифровка данных, благодаря чему атака может быть произведена даже на защищенное соединение. Сервер, получая оба запроса, отправляет на них два ответа с разными токенами сессии, посему авторизация произойдет как на компьютере пользователя, так и на клиенте злоумышленника.

Защита от данной атаки состоит в создании уникальных токенов авторизации. Один из конкретных вариантов — передача хеша пароля, вычисленного по двум открытым ключам, один из которых — уникальный. Первым открытым ключом является т.н. соль, которая хранится в базе данных сервера и является постоянной до смены пароля ([моя статья по теме](https://vk.com/@65zio-ya-vykachal-vsu-bazu-dannyh-no-do-sih-por-ne-znau-paroli-pol" \o "https://vk.com/@65zio-ya-vykachal-vsu-bazu-dannyh-no-do-sih-por-ne-znau-paroli-pol)). При GET-запросе страницы авторизации сервер должен сгенерировать уникальный токен авторизации с определенным временем жизни — например, 5 минут — для него формируется второй открытый ключ. Оба ключа передаются клиенту. Перед отправкой на сервер клиент сначала вычисляет хэш введенного пользователем пароля по первому ключу, потом по второму, после чего отправляет на сервер вычисленный хэш и токен авторизации. Сервер уже имеет вычисленный по первому ключу хэш — это есть хэш пароля, хранимый в базе данных, ему остается вычислить его по второму ключу, который он сохранил привязанным к токену авторизации. После первого же вычисления токен авторизации выбрасывается. Теперь, если злоумышленник попытается повторить запрос — сервер не сможет его обработать — токен и ключ уже были уничтожены, для сервера они невалидны.

Таким образом, сервер всё еще не хранит пароль пользователя, в то же время клиент не отправляет на сервер никаких паролей. Последнее архитектурно исключает возможность перехвата пароля по незащищенному соединению — что актуально, ведь многие люди используют один пароль для множества сервисов. Однако стоит понимать, что в случае использования незащищенного соединения возможность перехвата токена сессии никак не исключена.

Отмечу, что в отличии от аутентификации, при регистрации пользователя, пароль всё-таки должен будет быть передан на сервер, иначе не представляется возможным проверить его валидность. В таком случае не следует производить автоматическую аутентификацию пользователя после регистрации — при использовании защищенного соединения повторная отправка запроса ничего не даст злоумышленнику. При использовании же незащищенного соединения передаваемый пароль может быть легко извлечен со всеми последствиями.

Теоретическая часть окончена, перейдем к конкретной реализации.

Итак, на горизонте две основные задачи — реализация сервиса, заведующего токенами аутентификации и реализация кода, который будет хешировать пароль на клиенте. Визуальную же часть я позволю себе «накидать на коленке», без излишеств.

Начнем с реализации серверного сервиса. Создаем класс AuthTockenService, наследуемый от [BackgroundService](https://docs.microsoft.com/ru-ru/dotnet/api/microsoft.extensions.hosting.backgroundservice" \o "https://docs.microsoft.com/ru-ru/dotnet/api/microsoft.extensions.hosting.backgroundservice" \t "_blank). Заранее создаем поле для провайдера параметров readonly AppSettingsProvider SettingsProvider, а также поля для собственнно параметров сервиса: int TockenLifeTimeSecs, int TockenIdStringLength, int TokenKeyLengthBytes, int TockensCleanUpIntervalSecs. Помещаем значения указанных полей в файл appsettings.json и создаем сервис, который будет считывать данные параметры из файла (показывать его я не буду). Также создаем словарь, который собственно будет хранить токены авторизации.

AuthTockenService до реализации основного функционала

AuthTockenService до реализации основного функционала

Переходим к реализации основного функционала.

Функция создания токена аутентификации

Функция создания токена аутентификации

Функция подтверждения переданного пароля для переданного токена

Функция подтверждения переданного пароля для переданного токена

С сервисом закончим на том, что добавим в него ExecuteAsync (чего требует наследование от BackgroundService) и запишем в него простое асинхронное удаление устаревших токенов в соответствии с параметрами сервиса, после чего зарегистрируем в MVC строчкой services.AddSingleton<AuthTockenService>() в классе Startup.

Теперь нужно реализовать две страницы для входа на сайт. Первая — с вводом логина. На время разработки назову её NewAuthLogin . Она не будет делать ничего, кроме передачи введенного пользователей логина далее для ввода пароля. По сути, это исключительно фронт-страница. Она нужна для удобства пользователя. При желании можно вообще избавится от какой-либо серверной логики.

Далее будет происходить перенаправление на страницу ввода пароля с простой передачей логина в виде куки-параметра.

Фронт страницы NewAuthLogin ввода логина (почты — в данном случае)

Фронт страницы NewAuthLogin ввода логина (почты — в данном случае)

Бек страницы NewAuthLogin ввода логина (почты — в данном случае)

Бек страницы NewAuthLogin ввода логина (почты — в данном случае)

Ввод логина приводит к GET-запросу страницы, названной на время разработки NewAuthPassword. На сервере следует произвести валидацию введенного логина, проверить его наличие в БД, запросить оттуда первый открытый ключ для пароля (т.н. соль), сформировать токен авторизации и создать второй ключ; добавить всё в куки. Код представлен ниже.

Бек формирования ответа на GET-запрос NewAuthPassword

Бек формирования ответа на GET-запрос NewAuthPassword

Теперь необходимо написать фронт NewAuthPassword. Не буду показывать визуал страницы, он почти идентичен NewLoginPassword. Отличие состоит лишь в том, что input'у для пароля задан id='passwordInputId'. Скриптовая часть должна хешировать пароль и отправлять POST. Как можно заметить, я использую SHA512 — [не самый однозначный выбор](https://crypto.stackexchange.com/questions/35275/whats-the-difference-between-pbkdf-and-sha-and-why-use-them-together" \o "https://crypto.stackexchange.com/questions/35275/whats-the-difference-between-pbkdf-and-sha-and-why-use-them-together" \t "_blank), но я не буду останавливаться на этом в рамках статьи. Функция хеширования взята из самого популярного соответствующего [github-репозитория](https://github.com/emn178/js-sha512" \t "_blank), ибо запросить её через WebAPI можно только при наличии SSL-сертификата у сайта. Далее, после хеширования, на сервер будет отправляться POST-запрос.

Скриптовая часть страницы NewAuthPassword

Скриптовая часть страницы NewAuthPassword

Если авторизация прошла успешно, куки авторизации удаляются, добавляется токен сессии, созданный менеджером сессий (не рассматривается в рамках статьи, однако имеет простейшую реализацию ([его код на github](https://github.com/LuminoDiode/LuminoDiodeWebsite/tree/237c5230f58a1884c5adead7a2709ccbb9ac478a" \o "https://github.com/LuminoDiode/LuminoDiodeWebsite/tree/237c5230f58a1884c5adead7a2709ccbb9ac478a" \t "_blank))).



Бек формирования ответа на POST-запрос NewAuthPassword

Конец.

Post scriptum: За рамками текста я несколько раз переходил между концепциями использования непосредственно бинарных данных и использования string64 для хранения массивов байтов. В ходе этого я не только убедился в достаточной модульности моего кода — переход между этими концепциями внутри сервиса паролей и сервиса аутентификации не влиял на работу остального кода, но и пришел к концепции того, что «внутренние сервисы бекенда не должны ничего знать о том, куда поступают данные из них» — преобразованием должен заниматься только контроллер непосредственно перед передачей на клиент. Всё это было попыткой решить проблему с кодировкой — в итоге я понял, что мне просто следовало использовать base64url взамен обычного (но сейчас я не буду углубляться в эту тему).