**Вступление**

Эта тема посвящена безопасности. О ней нужно позаботиться, если не хотите, чтобы в один день вся ваша работа пошла крахом из-за злоумышленников.

Программист Виталий решил помочь своим соседям. Он разработал веб-сайт, на котором можно отправить показания счётчиков воды и электричества управляющей компании. Также он добавил функциональность получения и оплаты счетов за услуги ЖКХ.

Веб-приложение оказалось популярно, и жильцы соседних домов попросили дать им возможность тоже пользоваться сервисом. Виталий уволился с основной работы и стал работать только над собственным сервисом.

Но веб-приложением Виталия заинтересовались не только жильцы соседних домов — также на него обратили внимание злоумышленники. Пользователи сервиса стали жаловаться на списания денег со своих карточек за операции, которых они не производили. Бухгалтеры управляющих компаний забили тревогу из-за расхождений в балансе.

В результате пострадали все: бизнес Виталия закрылся, ему пришлось искать новую работу. Жильцы снова ходят пешком до отделения банка, чтобы оплатить счёт, а управляющие компании вернулись к бумажному документообороту.

Чтобы такого не произошло, нужно позаботиться о безопасности своего веб-сервиса. Хакеры очень изобретательны: атаки разнообразны, так что и подходы по борьбе с ними — тоже. В этой теме мы рассмотрим основные виды кибератак и научим защищать от них свой сайт.

# Межсайтовый скриптинг (XSS)

Первый тип атак, с которым мы разберёмся — межсайтовый скриптинг (cross-site scripting). Вы уже встречались с этими атаками, когда знакомились со свойством innerHTML. Тогда мы рассказывали историю о Сэмми, который воспользовался недосмотром разработчиков MySpace, собрал за несколько часов миллион друзей и «положил» тем самым сервер соцсети.

И хотя истории Сэмми уже больше 14 лет, XSS по-прежнему входит в десятку самых опасных уязвимостей веб-приложений.

Суть XSS-атак в том, чтобы внедрить вредоносный код в страницу, которую сервер отдаёт пользователю. Сэмми отредактировал свою страницу на MySpace, внедрив в неё вредоносный код. MySpace сохранил эту страницу у себя на сервере и выдавал другим пользователям. Вредоносный код запускался у них на компьютерах и творил бесчинства.

## Если отказаться от innerHTML, всё будет хорошо?

К сожалению, нет. Есть множество способов осуществить межсайтовый скриптинг.

Виктор оставляет на сайте отзыв и передаёт внутри запроса имя:

Скопировать кодJAVASCRIPT

http:*//example.com/?name=Виктор*

Сервер обрабатывает запрос, выкладывает отзыв и подписывает его переданным именем:

Скопировать кодJAVASCRIPT

const express = require('express');

const app = express();

*// код для передачи отзыва*

app.get('/', (req, res) => {

res.send(`Автор отзыва: ${req.query.name}`);

});

app.listen(3000);

Эту логику могут использовать хакеры. В ссылку можно вставить вредоносный код, а потом как-то вынудить другого пользователя эту ссылку открыть: сообщением о выигрыше в лотерею, скидкой 99% на гоночный автомобиль или предложением изменить размеры какого-нибудь органа:

Скопировать кодJSX

// Такая ссылка может прийти в письме «Перейдите по ссылке, чтобы получить свой выигрыш!»

http://example.com/?name=Виктор+<script>alert('Hacked!');</script>

Мы в примере просто показываем окно с сообщением "Hacked", но хакеры предпочитают код повреднее.

## Кто поможет, кто защитит?

В борьбе с киберпреступностью вы не одиноки. Наш пример атаки не сработает в браузерах на движке chromium: Яндекс-браузере, Гугл-хроме. В самом движке предусмотрена защита, так что попытка открыть такую ссылку приведёт к ошибке: "The XSS Auditor refused to execute a script in '[*http://example.com/?name=Viktor+*](http://example.com/?name=Viktor+)<script>alert('Hacked!');</script>' because its source code was found within the request. The auditor was enabled as the server did not send an 'X-XSS-Protection' header." Что означает «XSS-аудитор отклонил выполнение скрипта на странице '[*http://example.com/?name=Viktor+*](http://example.com/?name=Viktor+)<script>alert('Hacked!');</script>', поскольку в запросе был найден исполняемый код. Аудитор включён, поскольку сервер не отправил заголовок „X-XSS-Protection“».

Но в Internet Explorer никакой ошибки не выдаст, а послушно выполнит вредоносный код. Да и на хромиумы полагаться не стоит, от такого кода они не защитят:

Скопировать кодJAVASCRIPT

http:*//example.com/?name=Виктор+<b><a href="http://сайтхакера">YOU WON $100000</a>*

Хакер может пойти ещё дальше: сократить ссылку специальным сервисом, исполнить скрипт на компьютере пользователя и перенаправить его на исходный сайт. Так пользователь и не заметит, что произошло что-то неладное.

## Что же делать?

### Ограничивать пользовательский ввод

Для этого применяют принцип «Что не разрешено — запрещено». Он заключается в том, чтобы заранее установить разрешённые для ввода символы в каждое поле: поле «автор» может содержать только кириллические буквы и пробел, номер телефона — цифры, знак «плюс» и скобки, email — латинские буквы, цифры, нижние подчёркивания, дефисы, точки и символ «собака». Если пользователь вводит неразрешённые символы, сайт либо их отбрасывает, либо сообщает об ошибке.

### Экранировать символы

Иногда нам нужно отправить символы, которые используются злоумышленниками. Например, < и > могут быть знаками «меньше» и «больше», а не границами тега. В этом случае мы не можем запретить их для ввода.

В этом случае символы следует экранировать, тогда браузер не станет воспринимать их как код. Поэтому экранирование защищает от внедрения скриптов.

**Экранирование символов в HTML**

Допустим, мы сделали сервис для решения математических неравенств. Пользователь отправляет пример:

Скопировать кодJAVASCRIPT

"x \* 2 < 4";

Сервер решает неравенство и возвращает ответ:

Скопировать кодJAVASCRIPT

"Неравенство x \* 2 < 4 верно при x < 2";

Мы не можем запретить символ <, но можем его экранировать. Для этого его следует заменить на символ с кодовым обозначением — мнемоникой. Для знака «меньше» это &lt;. Получится такая строка:

Скопировать кодJAVASCRIPT

"Неравенство x \* 2 &lt; 4 верно при x &lt; 2";

Если злоумышленник отправит на сервер скрипт, мы не станем беспокоиться, что тот будет исполнен на компьютере пользователя. Ведь все символы экранированы и браузер этот скрипт не запустит:

Скопировать кодJAVASCRIPT

*// Злоумышленник отправил:*

Виктор+<script>alert('Hacked!');</script>

*// Пользователь получил:*

Виктор+&lt;script&gt;alert('Hacked!');&lt;/script&gt;

### Экранирование на практике

В node.js уже есть модуль, подставляющий мнемоники: escape-html. Его можно импортировать в проект и использовать в коде:

Скопировать кодJAVASCRIPT

const escape = require('escape-html');

Переменная escape содержит в себе функцию, которая принимает на вход строку со спецсимволами и заменяет их на мнемоники:

Скопировать кодJAVASCRIPT

escape('<script>alert("hacked")</script>');

*// '&lt;script&gt;alert(&quot;hacked&quot;)&lt;/script&gt;*

## Content Security Policy

Основная защита от XSS — правильно написанный код веб-приложения: с ограниченным вводом и экранированием символов. Но бывает всякое, и злоумышленник всё равно может найти способ внедрить скрипт в браузер пользователя. На этот случай есть вторая линия защиты — инструкция Content Security Policy.

Эта инструкция позволяет ограничить источники, откуда сайт может загружать скрипты, изображения, файлы стилей и медиафайлы.

Есть два варианта установить инструкцию:

1. В метатеге. Можно добавить в раздел head тег meta и описать ограничения там:

Скопировать кодJAVASCRIPT

<meta http-equiv="Content-Security-Policy" content="ИНСТРУКЦИИ">

1. В HTTP-заголовке:

Скопировать кодJAVASCRIPT

Content-Security-Policy: */\* ИНСТРУКЦИИ \*/*

### Как оформить инструкции

Есть 5 видов инструкций:

* default-src — обязательная инструкция, задаёт источник всех видов ресурсов по умолчанию;
* script-src — источник скриптов;
* img-src — изображений;
* media-src — аудио- и видео-файлов;
* style-src — файлов стилей.

После имени инструкции указывают сами источники через пробел. Чтобы указать домен сайта как источник, устанавливают ключевое слово 'self'.

Скопировать кодJAVASCRIPT

"script-src 'self' \*.site.com"; *// скрипты можно загружать с самого сайта, либо с поддоменов site.com, например, с https://example.site.com*

В конце каждой инструкции должна стоять точка с запятой:

Скопировать кодJAVASCRIPT

Content-Security-Policy: default-src 'self'; img-src \*; media-src media1.com media2.com; script-src userscripts.example.com

## Контроль версии браузера

Более новые браузеры обычно лучше защищены. Поэтому во имя безопасности можно запретить вход на сайт пользователям со старых браузеров.

Браузер отправляет свою версию в любом запросе внутри заголовка User-Agent. Этот заголовок имеет формат:

Скопировать кодJAVASCRIPT

User-Agent: <product> / <product-version> <comment>

Например:

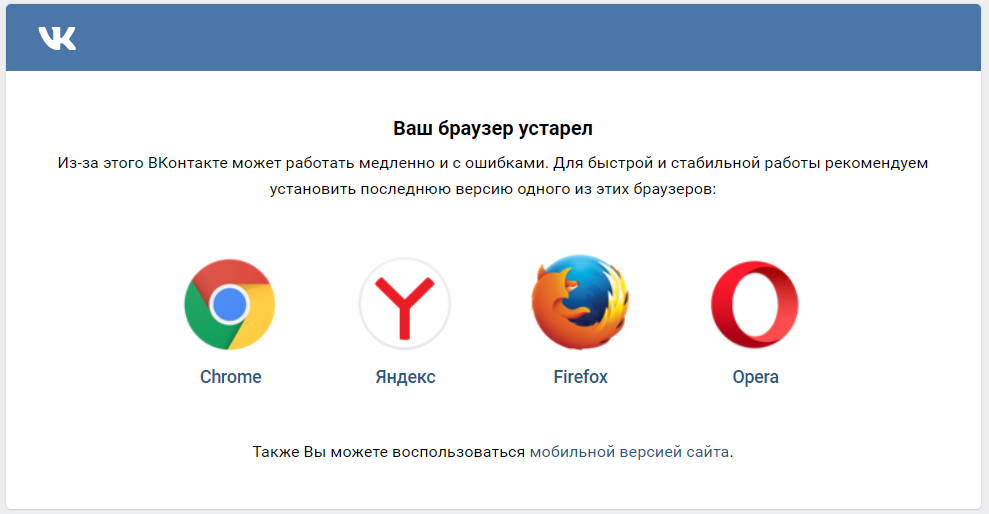
Скопировать кодJAVASCRIPT

user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/77.0.3865.90 Safari/537.36

Не пускать пользователя со старого браузера хорошо для безопасности. Но для пользователя это неудобно. Это противоречие обуславливает два подхода:

* поддерживать максимально возможное число браузеров, вплоть до старых версий Internet Explorer;
* не пускать пользователей старых браузеров.

Второй подход используют Вконтакте:



Какой подход выбрать — решать команде сервиса. Всегда взвешивайте, что важнее: безопасность или дружелюбие к пользователю.

## Итого

При любой XSS-атаке хакеры пользуются тем, что данные от пользователей попадают на сервер, а затем — обратно в браузер без каких-либо проверок. Поэтому чтобы защититься от XSS-атак:

* По возможности не передавайте параметры в URL. Используйте POST для отправки данных форм.
* Экранируйте спецсимволы при получении и отображении данных.
* Устанавливайте Content Security Policy HTTP-заголовки или Meta-тэги.
* Контролируйте пользовательский ввод согласно принципу «что не разрешено — запрещено».
* Напоминайте посетителям обновить браузер.

# Способы хранения JWT в браузере

В предыдущей теме мы разбирали хранение JWT в localStorage. Это хороший способ, потому что с локальным хранилищем удобно работать.

Однако, у него есть большой недостаток — низкий уровень безопасности. Локальное хранилище доступно из JavaScript, поэтому при успешной XSS-атаке злоумышленник может украсть токен и совершать действия от лица пользователя.

Как бы хорошо ни был защищён сервер, он всё равно может быть подвержен XSS-атаке. Если на сайте много зависимостей, в одной из них может оказаться вредоносный код. Поэтому дополнительная защита токена не помешает.

## Куки

Данные в браузере можно сохранять в так называемые куки. Это фрагменты данных, относящихся к определённому домену.

Чтобы записать данные в куки браузера, сервер должен отправить заголовок Set-Cookie:

Скопировать кодJAVASCRIPT

res.set('Set-Cookie', 'name=Стас');

Получив такой заголовок, браузер сохранит name=value в куках.

Вручную устанавливать куки неудобно. Зато это удобно делать в express — тут для этого есть метод res.cookie:

Скопировать кодJAVASCRIPT

res.cookie('name', 'Стас');

Первый аргумент — это ключ, второй — значение.

Метод res.cookie особенно удобен для отправки опций, контролирующих поведение кук. К примеру, куки являются сессионными по умолчанию. То есть, когда пользователь закрывает страницу — они удаляются. Это поведение можно изменить, отправив опцию maxAge. Она хранит срок жизни куки:

Скопировать кодJAVASCRIPT

*// такая кука будет храниться час*

res.cookie('name', 'Стас', { maxAge: 3600000 });

*// а такая 7 дней*

res.cookie('name', 'Стас', { maxAge: 3600000 \* 24 \* 7 });

Также по умолчанию к кукам есть доступ из JavaScript. Но в отличие от localStorage, это можно изменить опцией httpOnly:

Скопировать кодJAVASCRIPT

res.cookie('name', 'Стас', {

maxAge: 3600000,

httpOnly: true

});

Такую куку нельзя прочесть из JavaScript. Защитим токен:

Скопировать кодJAVASCRIPT

*// отправим токен, браузер сохранит его в куках*

res

.cookie('jwt', token, {

*// token - наш JWT токен, который мы отправляем*

maxAge: 3600000,

httpOnly: true

})

.end(); *// если у ответа нет тела, можно использовать метод end*

JWT сохранится в куках, к которым у JavaScript кода не будет доступа. А значит, токен не украсть при XSS-атаке.

А о том, как создавать токен на сервере мы расскажем в одной из следующих тем.

## Говорим fetch отправлять куки

Мы сохранили данные в куках, у которых нет доступа из JS кода, но запросы мы посылаем из JavaScript, используя fetch. Как быть? Браузер автоматически пошлёт куки, если фронтенд и бэкенд работают на одном домене. То есть если express сервер отвечает и за API и за раздачу фронтенда, то куки будут отправлены.

Если фронтенд и бэкенд работают на разных доменах, например, фронтенд мы разместили на Github Pages, а бэкенд ещё где-то, то в fetch нужно включить опцию credentials:

Скопировать кодJAVASCRIPT

fetch('/posts', {

method: 'GET',

credentials: 'include', *// теперь куки посылаются вместе с запросом*

});

И браузер пошлёт куки в специальном заголовке Cookie.

## Разбираем сookie на сервере

С отправкой кук разобрались, переходим к их чтению на сервере. Для этого нужно извлечь данные из заголовка Cookie и затем разобрать полученную строку в объект. Это удобно сделать модулем cookie-parser. Установите его перед использованием и подключите как мидлвэр:

Скопировать кодJAVASCRIPT

const express = require('express');

const cookieParser = require('cookie-parser');

const app = express();

app.use(cookieParser()); *// подключаем парсер кук как мидлвэр*

Теперь куки доступны в объекте req.cookies:

Скопировать кодJAVASCRIPT

const express = require('express');

const cookieParser = require('cookie-parser');

const app = express();

app.use(cookieParser());

app.get('/posts', (req, res) => {

console.log(req.cookies.jwt); *// достаём токен*

});

## Проблемы такого подхода

Подход httpOnly не защищает токен от XSS полностью. Если злоумышленнику удастся внедрить JS-код на сайт, он сможет отправить авторизованный запрос к серверу — так же как его отправляем мы. Но важно, что злоумышленник не сможет прочитать информацию, которая хранится в httpOnly-куке: он не сможет загрузить её к себе на сервер и проанализировать.

Коротко о проблеме, которая появилась, как только мы начали использовать куки.

Теперь наше приложение подвержено CSRF-атаке. В следующем уроке будем защищаться от неё.

## Ссылки

Документация метода res.cookie:

<https://expressjs.com/ru/api.html#res.cookie>

Документация модуля cookie-parser

<https://github.com/expressjs/cookie-parser>

# Межсайтовая подделка запроса (CSRF)

В прошлом уроке мы изменили место хранения JWT: вместо localStorage стали использовать куки. Хотя мы и решили проблему доступности токена из JS-кода, появилась другая проблема — уязвимость к CSRF-атаке. CSRF-атака (Cross Site Request Forgery, англ. «межсайтовая подделка запроса») построена на том, что вместе с http-запросом браузер посылает куки соответствующего домена.

Если обратиться к сайту с другого домена, браузер всё равно отправит куки. Так злоумышленник может разместить на своём сайте форму, которая отправляет запрос на наш сервер. Пользователь зайдёт на этот вредоносный сайт, и от его имени будет отправлен авторизованный запрос на наш домен.

## Это ужасно. Как защититься? опция sameSite

Самый простой способ защиты от автоматической отправки кук — указать браузеру, чтобы тот посылал куки, только если запрос сделан с того же домена. Для этого есть опция sameSite:

Скопировать кодJAVASCRIPT

res

.cookie('jwt', token, {

maxAge: 3600000,

httpOnly: true,

sameSite: true *// добавили опцию*

})

.end(); *// если у ответа нет тела, можно использовать метод end*

Так большинство пользователей оказываются защищены. Мы говорим «большинство», потому что same site — относительно новая опция, и в старых браузерах она не работает. Поэтому пользователей, у которых браузер не поддерживает sameSite, лучше попросить обновиться.

# Создание и хранение секретных ключей

В прошлой теме мы генерировали JWT так:

Скопировать кодJAVASCRIPT

const token = jwt.sign({ \_id: user.\_id }, 'super-strong-secret');

Как вы думаете, 'super-strong-secret' — это хороший секретный ключ? На самом деле, не очень. Он состоит из осмысленных английских слов, такой секретный ключ подобрать проще, чем длинную комбинацию случайных символов. И тут возникает вопрос: какой длины ключа достаточно, чтобы его нельзя было подобрать?

## Криптостойкие псевдослучайные данные

Что это значит:

* **криптостойкость** означает, что данные генерируются алгоритмом, не поддающимся анализу. Проще говоря, на подбор ключа уйдут сотни лет;
* **псевдослучайность** означает, что символы очень похожи на случайные, но всё же сформированы математическим алгоритмом. Псевдослучайные данные гораздо проще генерировать, но почти также сложно подобрать. Поэтому они удовлетворяют нашей задаче.

Секретный ключ должен быть криптостойким и псевдослучайным.

Также нужно определиться с длиной пароля. Обычно сходятся на том, что оптимальный ключ должен быть последовательностью символов, которые в двоичном виде представляют собой цепочку нулей и единиц длиной 128 или 256 символов.

В Node.js есть встроенный модуль crypto, который генерирует такой ключ. На вход он принимает количество байт ключа, а не бит. Поскольку в одном байте 8 бит, для получения 128-битного ключа нужно передать методу crypto на вход 16:

Скопировать кодJAVASCRIPT

const crypto = require('crypto'); *// экспортируем crypto*

const randomString = crypto

.randomBytes(16) *// сгенерируем случайную последовательность 16 байт (128 бит)*

.toString('hex'); *// приведём её к строке*

console.log(randomString); *// 5cdd183194489560b0e6bfaf8a81541e*

Такой код писать не нужно, ведь ключ необходимо сгенерировать единожды. Можно просто сделать это в консоли.

Такое выражение сгенерирует 256-битный (32-байтный) ключ и выведет его в консоль:

Скопировать кодJAVASCRIPT

node -e "console.log(require('crypto').randomBytes(32).toString('hex'));"

Этот ключ нельзя подобрать методом перебора и за сотню лет.

Конечно, есть вероятность, что завтра человечество изобретёт невероятно мощный компьютер, подбирающий такие ключи за секунды. Если это произойдёт, мы вас предупредим. А пока 256-битного ключа достаточно.

## Где хранить ключи

Когда ключ сгенерирован, его нужно сохранить. В прошлой теме, для примера мы держали его прямо в коде:

Скопировать кодJAVASCRIPT

*// super-strong-secret хранится прямо в коде*

const token = jwt.sign({ \_id: user.\_id }, 'super-strong-secret');

**Никогда так не делайте**, иначе вся криптостойкость и псевдослучайность окажется бесполезной: ключ просто украдут. Обычно код хранят в удалённом репозитории, к которому есть доступ у множества людей: фронтенд- и бэкенд-разработчиков, продакт- и проджект-менеджеров, клиентов. Компьютер любого из них может быть взломан, а пострадают от этого пользователи.

Чем меньше людей имеют доступ к ключу, тем безопаснее. Поэтому секретный ключ хранят только на сервере, где запущено приложение.

Ключ заносят в переменные окружения. Для этого создают файл с расширением .env в корне проекта, и в нём объявляют env-переменные:

Скопировать код

NODE\_ENV=production

JWT\_SECRET=eb28135ebcfc17578f96d4d65b6c7871f2c803be4180c165061d5c2db621c51b

Чтобы загрузить этот файл в Node.js, нужно установить в проект модуль dotenv и как можно раньше импортировать его в app.js таким образом:

Скопировать кодJAVASCRIPT

*// app.js*

require('dotenv').config();

После этого env-переменные из файла добавятся в process.env:

Скопировать кодJAVASCRIPT

*// app.js*

require('dotenv').config();

console.log(process.env.NODE\_ENV); *// production*

К ним можно обращаться при генерации токена:

Скопировать кодJAVASCRIPT

const { NODE\_ENV, JWT\_SECRET } = process.env;

const token = jwt.sign(

{ \_id: user.\_id },

NODE\_ENV === 'production' ? JWT\_SECRET : 'dev-secret'

);

.env файл не нужно коммитить, поэтому его следует добавить .gitignore:

Скопировать кодJAVASCRIPT

.env

Документация модуля dotenv: <https://www.npmjs.com/package/dotenv>.

# Использование внешних компонентов

При разработке мы часто пользуемся внешними инструментами: загружаем библиотеки и файлы стилей, подключаем зависимости. Такие инструменты удобны и упрощают жизнь разработчику. Но вместе с удобством приходят и потенциальные угрозы безопасности. В этом уроке разберём, какие и как с этим быть.

## Подключение внешних компонентов тегом <script>

Многие библиотеки подключаются тегом <script> со ссылкой на сайт производителя библиотеки:

Скопировать кодHTML

<script src="https://code.jquery.com/jquery.slim.min.js"></script>

Такой подход опасен, поскольку мы не контролируем чужой сервер: файл может быть изменён, и в нём могут появиться уязвимости.

Первый шаг в защите от изменений — явно указывать версии подключаемых компонентов. Ссылка с указанием версии более надёжна, чем без указания:

| **МЕНЕЕ БЕЗОПАСНО** | **БОЛЕЕ БЕЗОПАСНО** |
| --- | --- |
| https://code.jquery.com/jquery.slim.min.js | https://code.jquery.com/jquery-3.4.1.slim.min.js |

Ещё надежнее загружать код модуля в директорию с проектом и указывать локальный путь: /vendor/jquery-3.4.1.slim.min.js. Но если модуль большой, такой подход чреват не только дублированием кода, но и увеличением кодовой базы и, как следствие, ростом времени сборки проекта.

### Хеши против подмены файла

Файлы можно защитить ключом: так при загрузке браузер проверит, скачивает ли он тот самый файл, который мы указывали в процессе разработки. Для этого у тегов <script> и <link> есть атрибут integrity. В него записывают хеш-сумму — ключ, который рассчитывается, исходя из содержания файла. При скачивании файла с внешнего ресурса браузер рассчитает хеш-сумму файла и загрузит его, только если полученная сумма совпадает с той, что указана в атрибуте integrity. Это гарантирует, что файл не изменился.

Скопировать кодHTML

<script

src="https://code.jquery.com/jquery-3.4.1.slim.min.js"

integrity="sha384-J6qa4849blE2+poT4WnyKhv5vZF5SrPo0iEjwBvKU7imGFAV0wwj1yYfoRSJoZ+n"

crossorigin="anonymous"

>

</script>

Перед самой хеш-суммой через тире указывают алгоритм, по которому её следует рассчитывать, в нашем случае — sha384.

Хеш-сумму файла нужно заранее посчитать. Это можно сделать в специальном сервисе: <https://www.srihash.org/>.

Атрибут crossorigin="anonymous" означает, что при запросе не будут передаваться никакие данные пользователя, даже если он авторизован на том сервере, с которого запрашивается файл.

## Блокировка npm-зависимостей

Уязвимости могут появиться и в новых версиях npm-пакетов, подключённых к проекту. Поэтому зависимости также следует контролировать.

В материале о вебпаке мы описывали зависимости в файле package.json. Для этого в терминале мы вызывали команду npm install, и в package.json появлялась новая зависимость:

Скопировать кодJAVASCRIPT

"bcryptjs": "^2.4.3"

Перед номером версии стоит шляпка ^. Это значит, что если есть более новая версия с такими же первыми двумя цифрами и большей третьей, следует взять самую свежую версию. То есть установить не 2.4.3, а 2.4.4, если она доступна.

Что плохого в таком подходе:

* у разных разработчиков могут быть разные версии зависимостей;
* мы можем установить повышенную версию зависимости без своей на то воли.

Чтобы избежать подобного, в пятой версии npm появилась возможность зафиксировать зависимости. Для этого в специальном файле package-lock.json перечисляют те пакеты, которые нужно установить строго в указанных версиях. Это исключает апдейт версии без ведома разработчика.

Если у вас в проекте нет package-lock.json, проверьте версию npm командой npm -v. Если версия ниже пятой — обновите npm командой npm install npm@latest -g.

Если установлена пятая или более новая версия npm , но package-lock.json всё равно нет, вызовите в терминале команду:

Скопировать кодBASH

npm config set package-lock true

или установите зависимости с ключом --package-lock:

Скопировать кодBASH

npm install --package-lock

## Проверка безопасности зависимостей

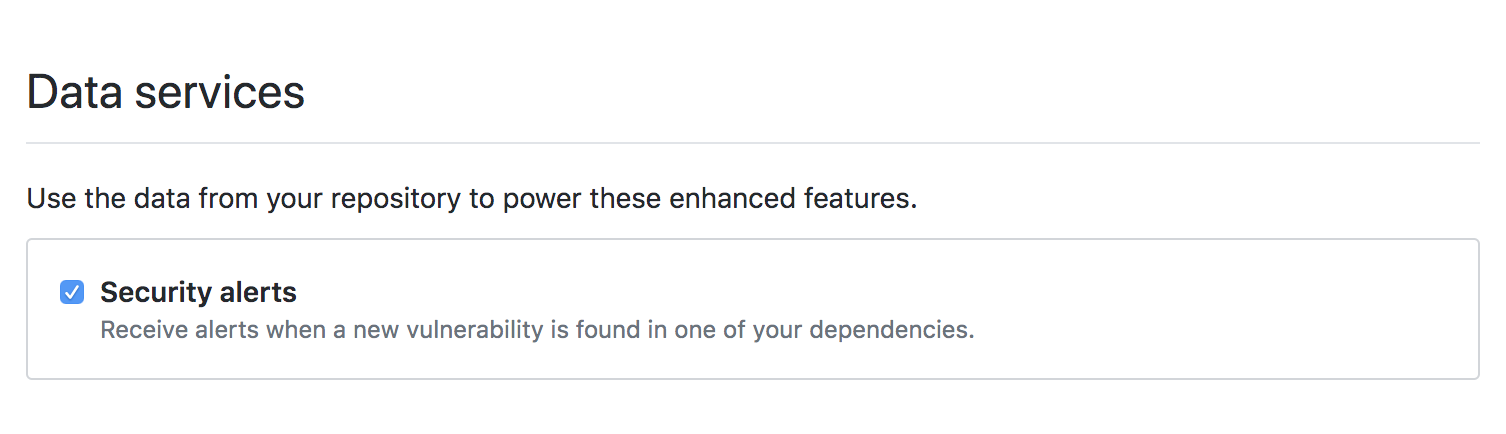
В шестой версии npm появился инструмент для анализа безопасности пакетов. Чтобы воспользоваться им, запустите команду npm audit, находясь в папке проекта. Так npm сообщит о зависимостях, в которых есть уязвимости.

## Github Security Alert

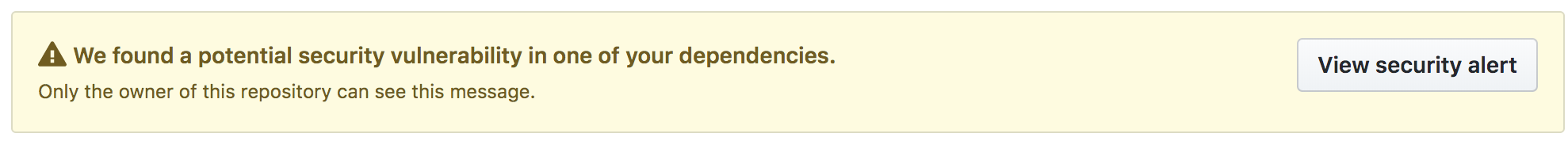
Чтобы постоянно отслеживать уязвимости в пакетах, есть специальные инструменты. Они позволяют не вводить команду npm audit, поскольку берут мониторинг уязвимостей на себя.

Каждый сервис контроля версий разрабатывает собственный инструмент для контроля безопасности пакетов. Например, в Gitlab есть функция Dependency Scanning, а в Github — Security Alert.

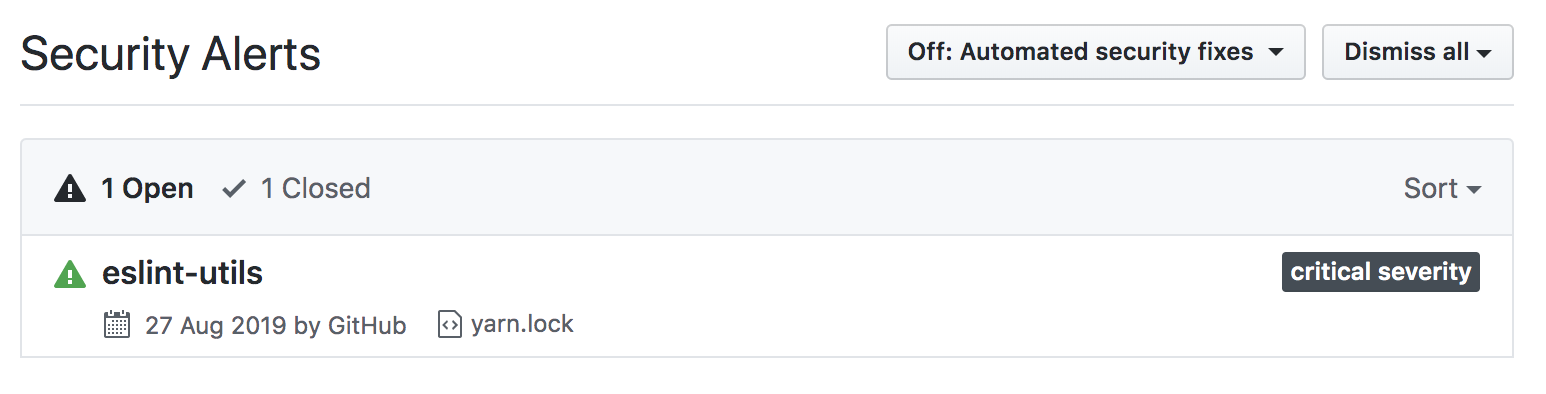
Чтобы подключить Github Security Alert, зайдите в репозиторий на Github, откройте настройки (Settings) и поставьте соответствующую галочку:



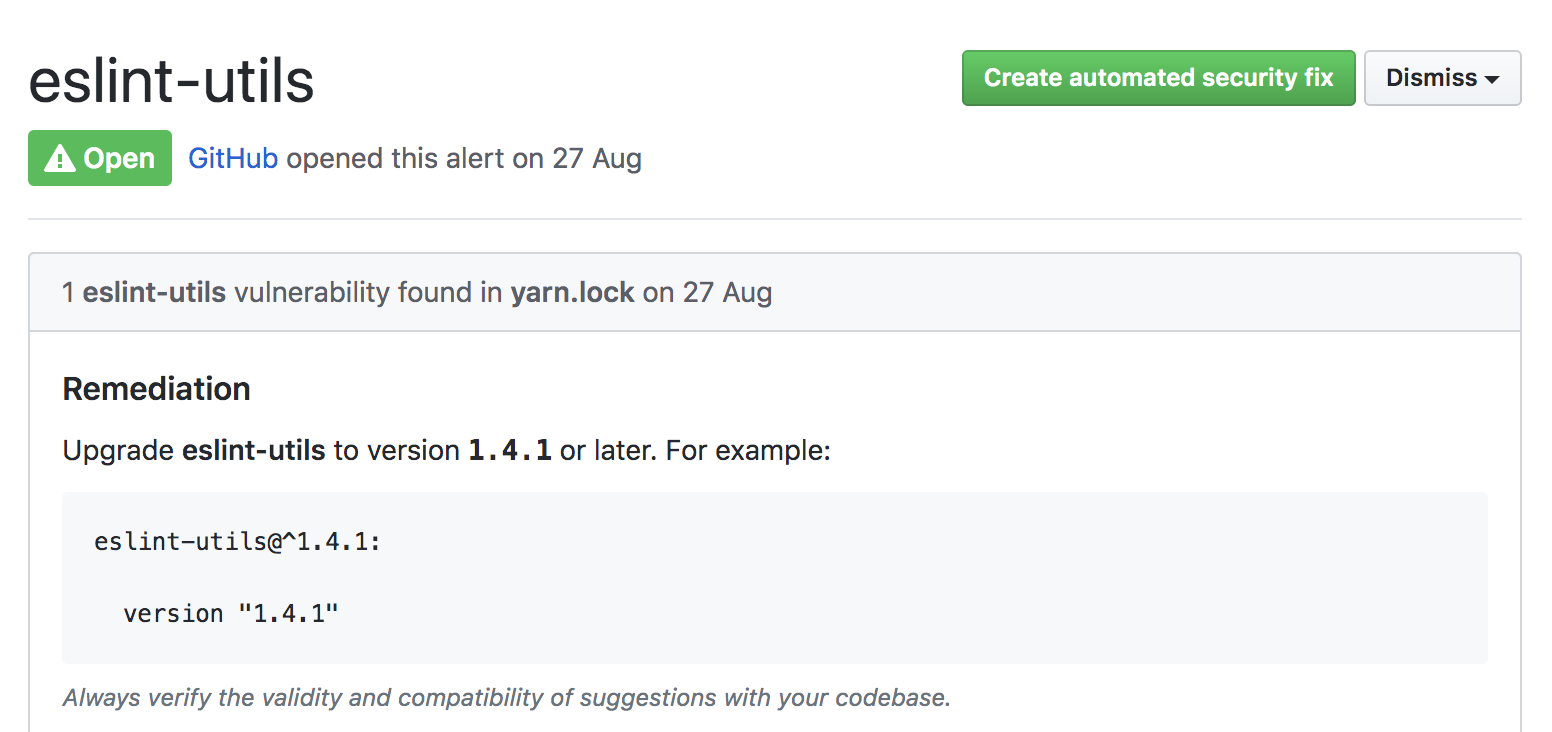
Если одна из зависимостей имеет известную уязвимость, на главной странице репозитория появится уведомление:



Чтобы узнать, какая именно зависимость небезопасна, кликните на View security alert:



Также тут можно открыть подробности: в чём суть уязвимости и до какой версии следует обновиться.



## Базы данных уязвимостей

Угрозы безопасности также собирают в открытых базах данных. Любой компонент следует проверять на уязвимости по таким базам, чтобы оградить от злоумышленников свой проект.

[CVE Details](https://www.cvedetails.com/) — одна из крупнейших открытых баз данных общеизвестных уязвимостей. Все компоненты следует проверять по этой базе, прежде чем использовать их в проекте. Можно искать по имени компании, продукта или типам уязвимостей.

Разумеется, если компонента нет в базе, это не означает, что тот безопасен. Ведь в базу уязвимость попадает уже после того, как она была найдена. То есть тогда, когда ей кто-то воспользовался, и соответственно — пострадал.

# Заключение. Чеклист безопасности

Вот чеклист для проверки своего проекта. Проходитесь по нему перед тем, как вашим кодом начнут пользоваться реальные люди.

### ✅ Настройте заголовки ответа

Есть ряд заголовков, отвечающих за разные аспекты безопасности: вы уже знакомы с Content-Security-Policy, которые позволяют ограничить источники скриптов и других ресурсов.

Заголовки безопасности можно проставлять автоматически — для этого есть модуль Helmet. Нужно установить его, импортировать и подключить как мидлвэр:

Скопировать кодJAVASCRIPT

const helmet = require('helmet');

app.use(helmet());

Документация Helmet: <https://www.npmjs.com/package/helmet>

### ✅ Проверяйте, какие данные присылает пользователь

Валидируйте данные на входе и заменяйте управляющие символы мнемониками. Если фронтенд вставляет их в сайт небезопасно, пользуйтесь escape-html.

### ✅ Задумывайтесь о том, где хранятся данные на клиенте

LocalStorage удобен и прост: он отлично подойдёт для хранения данных, которые нет нужды скрывать. Но к localStorage есть доступ из JavaScript-кода, и это поведение нельзя запретить. Поэтому для хранения секретных данных лучше подходят httpOnly куки.

### ✅ Помните о CSRF

Если ваш сервис при авторизации запросов опирается на автоматическую отправку кук браузером, реализуйте хотя бы простейшую защиту от CSRF. В этом поможет опция same site.

### ✅ Будьте готовы к брутфорсу и DDоSу

Чтобы защититься от множества автоматических запросов, воспользуйтесь специальным мидлвэром — express-rate-limit. Он ограничивает количество запросов с одного IP-адреса в единицу времени. После установки его можно импортировать, сконфигурировать и подключить:

Скопировать кодJAVASCRIPT

const express = require('express');

const rateLimit = require('express-rate-limit');

const app = express();

const limiter = rateLimit({

windowMs: 15 \* 60 \* 1000, *// за 15 минут*

max: 100 *// можно совершить максимум 100 запросов с одного IP*

});

*// подключаем rate-limiter*

app.use(limiter);

Документация модуля express-rate-limit: <https://www.npmjs.com/package/express-rate-limit>.

### ✅ Проверьте зависимости

Запускайте команду npm audit, прежде чем выкатывать проект в продакшн. Также настройте Security Alert на платформе, где храните код.

### ✅ Храните секреты

Храните ключи в переменных окружения. Доступ к файлу .env должен быть у минимально возможного количества людей, поэтому не храните его в репозитории проекта.

### ✅ Не останавливайтесь

Безопасность — неисчерпаемая тема. Крупные компании отдельно нанимают специалистов по информационной безопасности, чтобы защитить свои сервисы от атак. Это логично, ведь у таких компаний очень высокие ставки.

В следующем спринте мы ещё раз коснёмся безопасности. Но также мы рекомендуем не останавливаться на программе нашего курса и изучать веб-безопасность вместе с другими сферами веб-разработки.

Привет! Главное, что необходимо понимать - это для чего используют экранирование или sanitaze по английски.Бывают пользователи - редиски, которые хотят получить из нашей базы, то что они не должны получать или и того хуже удалить что-нибудь, что они не могут удалить.Давай рассмотрим на каком-нибудь примере из нашей проектной работы. Например, updateUserInfo.  
Мы используем метод findByIdAndUpdate и надеемся, что пользователь в качестве id передал нормальный идентификатор, но в mongoDB есть так называемые [логические операторы](https://docs.mongodb.com/manual/reference/operator/query/), которые тоже можно передавать в методы find.Например, пользователь может передать в поле \_id { $ne: '' } и обновить всех пользователей идентификатор которых не равен пустой строке (т.е. считай абсолютно всех, если конечно дополнительных проверок никаких нет).Поэтому вывод: мы не должны доверять данным от пользователя, которые мы используем условно в методах find, т.е. если если мы просто вставляем пользователя в БД, то нет необходимости экранировать его id и другие данные, но если мы осуществляем поиск по БД, то лучше экранировать параметр, который мы используем для поиска (тот же id к примеру)Вывод номер два: не важно как это делать, можно свою функцию написать, можно какой-нибудь сторонний пакет использовать, опять же валидация схемы и запросов помогает.

**docs.mongodb.com**

[**Query and Projection Operators**](https://docs.mongodb.com/manual/reference/operator/query/) (4 kB)

<https://docs.mongodb.com/manual/reference/operator/query/>

https://ca.slack-edge.com/TPV9DP0N4-U01F0MCBYVB-67b9c6736f71-48

[**Александр Катрюхин**](https://app.slack.com/team/U01F0MCBYVB)  [1 hour ago](https://yandex-students.slack.com/archives/G01ENBXA9BR/p1626434119387600?thread_ts=1626358994.330100&cid=G01ENBXA9BR)

Т.е. если я, например в роуте get('/:userId'  вставляю проверку celebrate с параметром hex(), то этот трюк злоумышленника с id уже не пройдет, и дополнительно можно ничего не настраивать? Такой принцип?

image.png

https://ca.slack-edge.com/TPV9DP0N4-U01EVLJ5XJM-7d9fb5e82d6d-48

[**Михаил Зятьков**](https://app.slack.com/team/U01EVLJ5XJM)  [2 minutes ago](https://yandex-students.slack.com/archives/G01ENBXA9BR/p1626436968388500?thread_ts=1626358994.330100&cid=G01ENBXA9BR)

Да, принцип верный