# Атака SQL-Injection (Внедрение SQL кода)

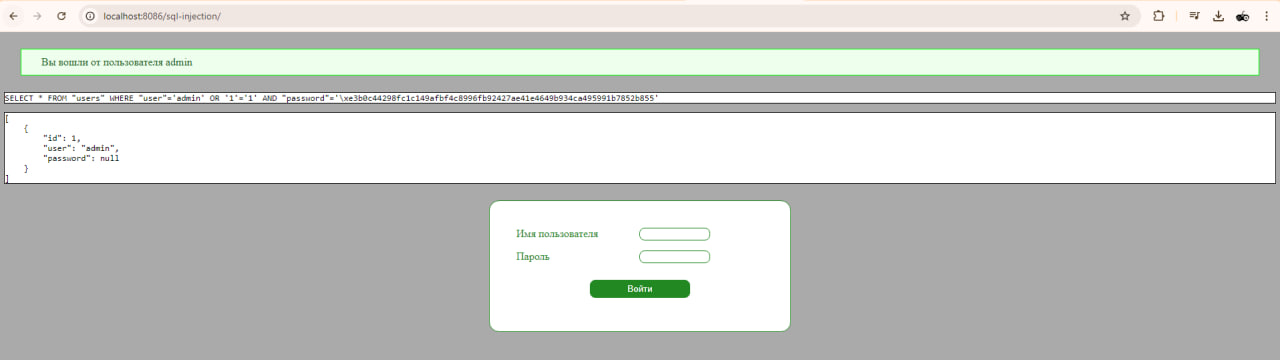
1) В поле логина указываем: admin' OR '1'='1

В поле пароля ничего не пишем.

Получаем запрос:

SELECT \* FROM "users" WHERE "user"='admin' OR '1'='1' AND "password"='\xe3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855'

И мы смогли авторизоваться под пользователем admin:



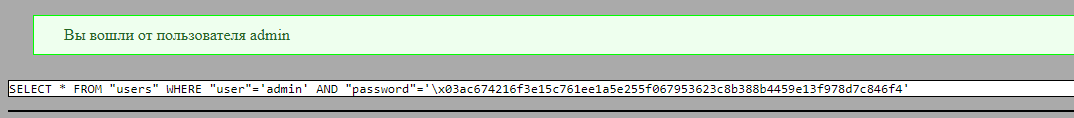
2) Так как мы разработчики, то можем подключиться к бд, развернутой в контейнере и просто установить пароль пользователю admin:

UPDATE public.users

SET password = digest('1234', 'sha256')

WHERE id = 1;

Логинимся без проблем с паролем 1234:



SQL-инъекции являются одной из самых распространенных и опасных уязвимостей в веб-приложениях, позволяющих злоумышленникам выполнять произвольные SQL-запросы к базе данных через вводимые пользователем данные.

**1) Поможет ли экранирование пришедших извне параметров, и что это вообще такое экранирование?**

**Экранирование** — это процесс обработки входных данных для предотвращения интерпретации специальных символов как части SQL-запроса. Основная цель экранирования — обезопасить приложение от SQL-инъекций путем правильной обработки вводимых пользователем данных.

Как работает экранирование?

Когда пользователь вводит данные, которые затем включаются в SQL-запрос, специальные символы (например, одинарные кавычки ', двойные кавычки ", точка с запятой ; и т.д.) могут изменить структуру запроса. Экранирование заменяет или удаляет эти символы, чтобы они не влияли на синтаксис SQL-запроса.

Пример:

До экранирования: SELECT \* FROM users WHERE username = 'O'Brien';

После экранирования: SELECT \* FROM users WHERE username = 'O''Brien';

Экранирование помогает защититься от синтаксических ошибок, но **не помогает** от SQL инъекций.

**2) Помогут ли "вставки" параметров? То есть запросы вида: SELECT \* FROM table WHERE column1=? AND column2=? с последующей подстановкой значений в заранее заданные места с помощью специальной функции?**

**Параметризованные запросы** (также известные как подготовленные выражения или prepared statements) — это способ выполнения SQL-запросов, при котором структура запроса отделена от данных. Вместо прямой вставки данных в SQL-запрос используются параметры-заполнители (? или именованные параметры), а значения передаются отдельно.

**Подготовка запроса**: Вы определяете SQL-запрос с параметрами-заполнителями:

SELECT \* FROM users WHERE username = ? AND password = ?;

**Передача параметров**: Значения для параметров передаются отдельно через API базы данных:

cursor.execute("SELECT \* FROM users WHERE username = ? AND password = ?", (username, password))

Параметризированные запросы помогают защититься от SQL инъекций, так как данные передаются как параметры запроса, а не конкатенируются в строку запроса, вводимые пользователем данные не могут изменить структуру запроса.

**3) Помогут ли для этого специальные библиотеки ORM (Object-Relational Mapping) и прочие, которые сами составляют запросы для получения и сохранения объектов из/в базу данных? И что такое вообще ORM?**

**ORM (Object-Relational Mapping)** — это технология, позволяющая разработчикам работать с базами данных с использованием объектно-ориентированных языков программирования. ORM автоматически преобразует объекты в записи базы данных и обратно, абстрагируя детали SQL-запросов.

**Что такое ORM?**

* Объектно-ориентированный подход: Вместо написания SQL-запросов вы работаете с объектами и их свойствами.
* Автоматизация: ORM-библиотеки автоматически генерируют SQL-запросы на основе операций с объектами.
* Удобство: Снижается объем кода, упрощается поддержка и развитие приложения.

**Как ORM помогает предотвратить SQL-инъекции?**

* Параметризованные запросы под капотом: ORM-библиотеки обычно используют параметризованные запросы при взаимодействии с базой данных, что обеспечивает защиту от SQL-инъекций.
* Валидация данных: ORM может включать механизмы валидации и очистки данных перед сохранением в базу данных.
* Абстракция SQL: Поскольку вы не пишете сырой SQL-код, риск допустить ошибку, ведущую к SQL-инъекции, снижается.

**Ограничения и предостережения:**

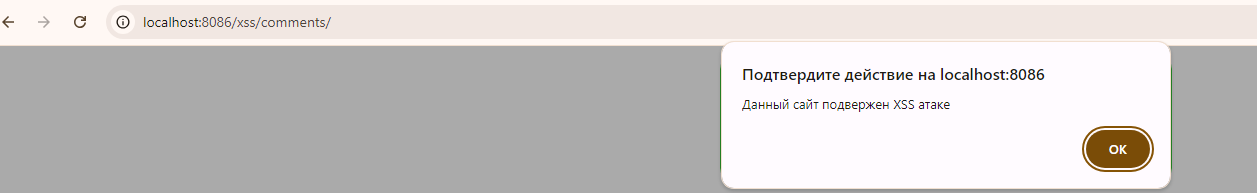
* Ручные SQL-запросы: Если вы используете ORM, но при этом пишете сырой SQL и конкатенируете строки, вы все равно уязвимы для SQL-инъекций.
* Неправильное использование: ORM не гарантирует безопасность, если разработчик обходит встроенные механизмы защиты.
* Производительность: В некоторых случаях ORM может быть менее производительным по сравнению с оптимизированными ручными SQL-запросами.

**Рекомендуемые практики при использовании ORM:**

* Используйте встроенные методы: Всегда предпочитайте встроенные методы ORM для выполнения операций с базой данных.
* Избегайте сырых SQL-запросов: Если необходимо выполнить сырой SQL-запрос, используйте методы ORM для безопасного выполнения параметризованных запросов.
* Валидация и проверка: Дополнительно валидируйте и проверяйте данные на уровне приложения.

# Атака XSS (Межсайтовый скриптинг)

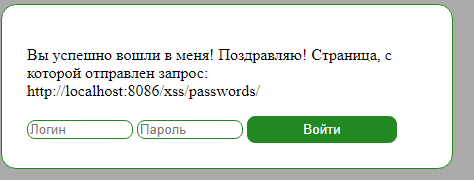
Напишите комментарий такого содержания: <script> alert('Данный сайт подвержен XSS атаке'); </script>.

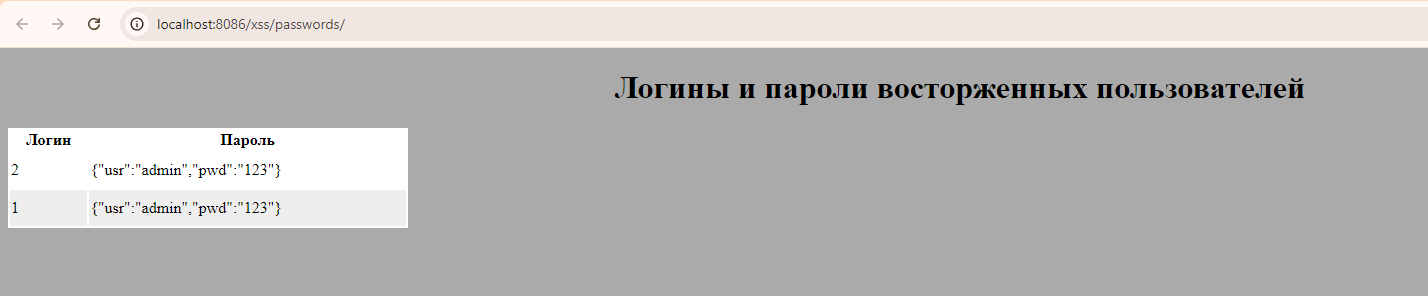
Теперь выскакивает алерт об XSS атаке:  


Теперь напишем комментарий:

<script> document.forms[0].action = '/xss/passwords/'; </script>

После чего любой пользовать, кто попробует войти, отправит злоумышленникам свой пароль:





**Как можно убрать весь JS код из строки?**

В .Net для этого есть встроенные библиотеки:

using Microsoft.Security.Application;

public string SanitizeInput(string input)

{

return Sanitizer.GetSafeHtmlFragment(input);

}

Удаляет или заменяет потенциально опасные части данных, особенно если они не нужны.

**Можно ли это сделать экранированием?**

Экранирование заменяет специальные символы (например, <, >, ", ') на их HTML-сущности, чтобы браузер не воспринимал их как код. Например:

< заменяется на &lt;

> заменяется на &gt;

" заменяется на &quot;

' заменяется на &#39;

& заменяется на &amp;

Например:

Входная строка: <script>alert('XSS');</script>

Станет: &lt;script&gt;alert(&#39;XSS&#39;);&lt;/script&gt;

В итоге браузер отобразит строку как текст, и она **не будет выполнена как JavaScript-код.**

Но этого может быть недостаточно:

**Внедрение в JavaScript-контекст**

Если данные выводятся не в HTML, а в JavaScript-контексте, например:

var userInput = "<%= userInput %>";

И если userInput содержит:

"; alert('XSS'); //

То после экранирования:

var userInput = "&quot;; alert(&#39;XSS&#39;); //";

Все равно выполнится JavaScript-код.

**Атрибуты HTML**

Если данные вставляются в атрибуты:

<a href="<%= userInput %>">Link</a>

И userInput содержит:

" onclick="alert('XSS')

То даже с экранированием браузер может интерпретировать это как часть атрибута href и вызвать событие onclick.

**Различные кодировки**

Злоумышленники могут использовать различные кодировки (например, Unicode) для обхода экранирования:

* Использование символов, похожих на экранированные, но которые браузер интерпретирует по-другому.
* Вставка данных в формате, который игнорирует стандартное экранирование.

**Экранирование не защищает от всех типов XSS**

XSS бывает нескольких видов:

* **Reflected XSS** (Отраженная): когда данные сразу же отображаются на странице без фильтрации.
* **Stored XSS** (Хранимая): когда данные сохраняются в базе данных и затем выводятся на странице.
* **DOM-based XSS**: когда злоумышленник использует изменения в DOM без обращения к серверу.

Экранирование защищает только от первых двух типов, если используется правильно. DOM-based XSS может обойти экранирование, потому что проблема там — в некорректной манипуляции с DOM-деревом.

**Как бороться с XSS?**

**1) Экранирование**

@userInput // Razor автоматически экранирует вывод

**2) Правильная вставка данных**

Используйте textContent или innerText в JavaScript, а не innerHTML:

element.textContent = userInput;

**3) CSP (Content Security Policy)**

Настройка Content Security Policy позволяет контролировать, какие скрипты могут выполняться:

Content-Security-Policy: default-src 'self'; script-src 'self';

Это предотвратит выполнение внедренного скрипта.

**4) Фильтрация и валидация**

Удаляйте или заменяйте потенциально опасные части данных, особенно если они не нужны:

using Microsoft.Security.Application;

public string SanitizeInput(string input)

{

return Sanitizer.GetSafeHtmlFragment(input);

}

**5) Проверка типов и длины**

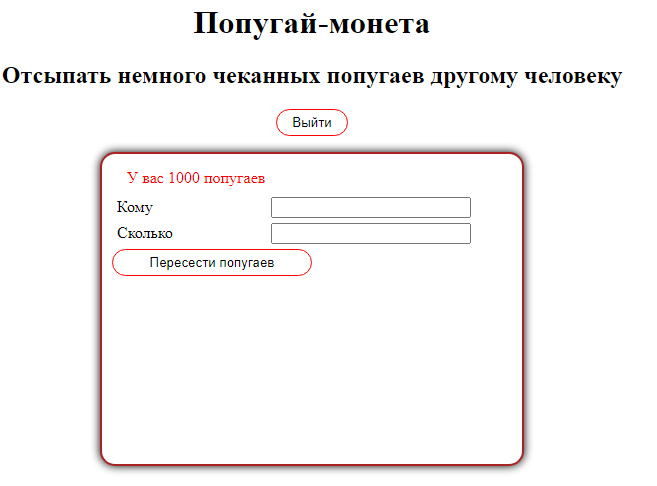
Ограничьте типы и длину данных, которые пользователь может ввести. Например, если поле должно содержать только буквы, запретите все остальное.

**6) Разделение данных и логики**

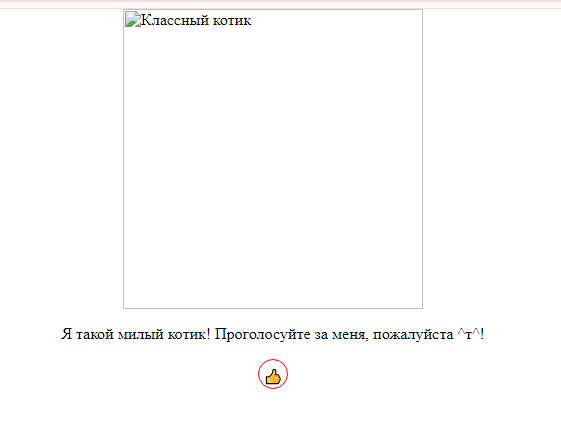
Никогда не вставляйте пользовательские данные напрямую в JavaScript-код или атрибуты без проверки и экранирования.

# Атака CSRF

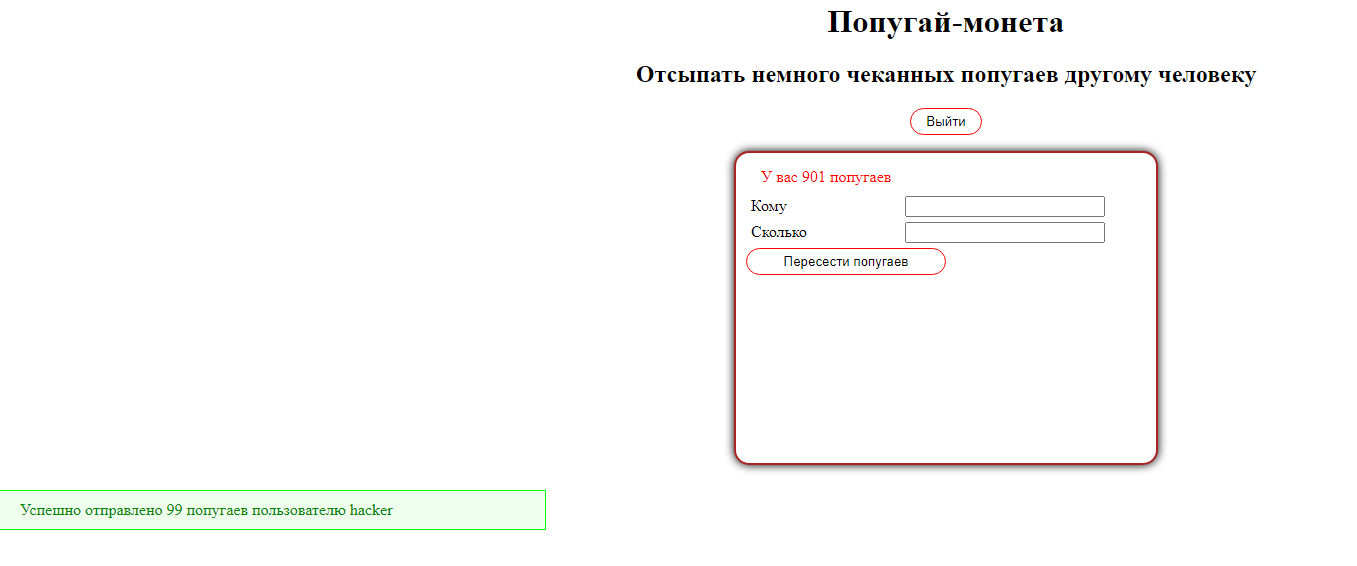
Мы прошли аутентификацию:



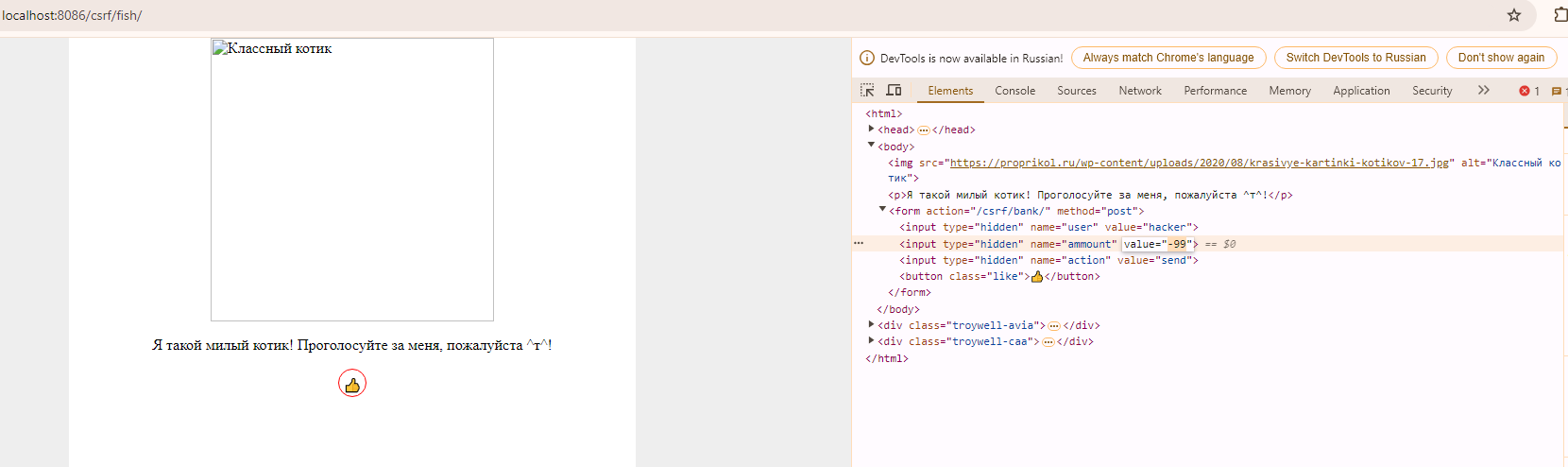
Дальше нажали на лайк:



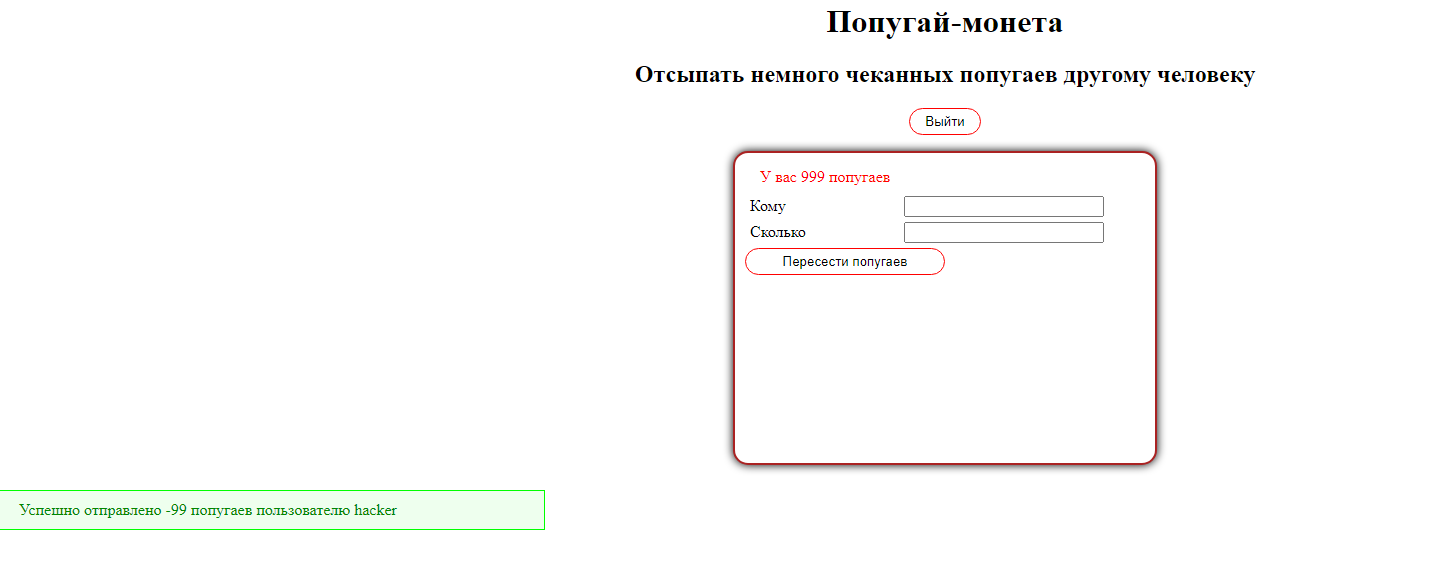
И переслали попугаев:



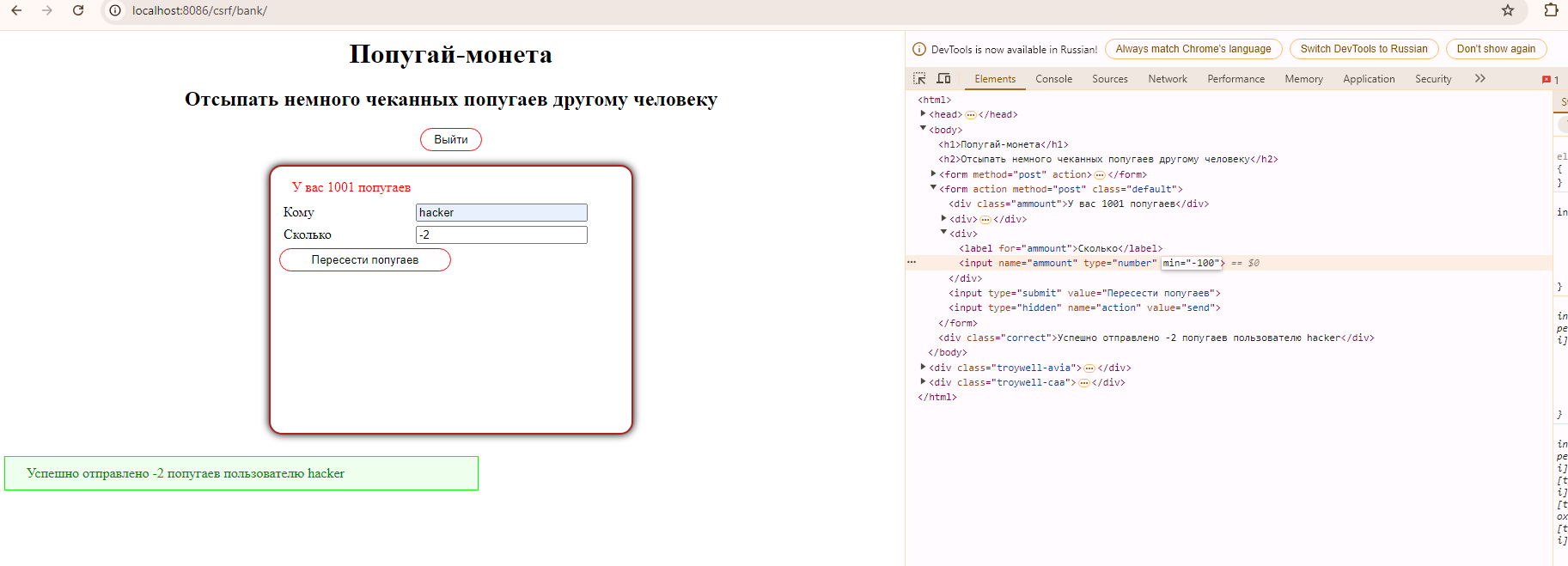
Изменили value при отправке на отрицательное число:



И получили попугаев обратно:



Так же мы можем прямо через страницу банка забирать попугаев у других пользователей:

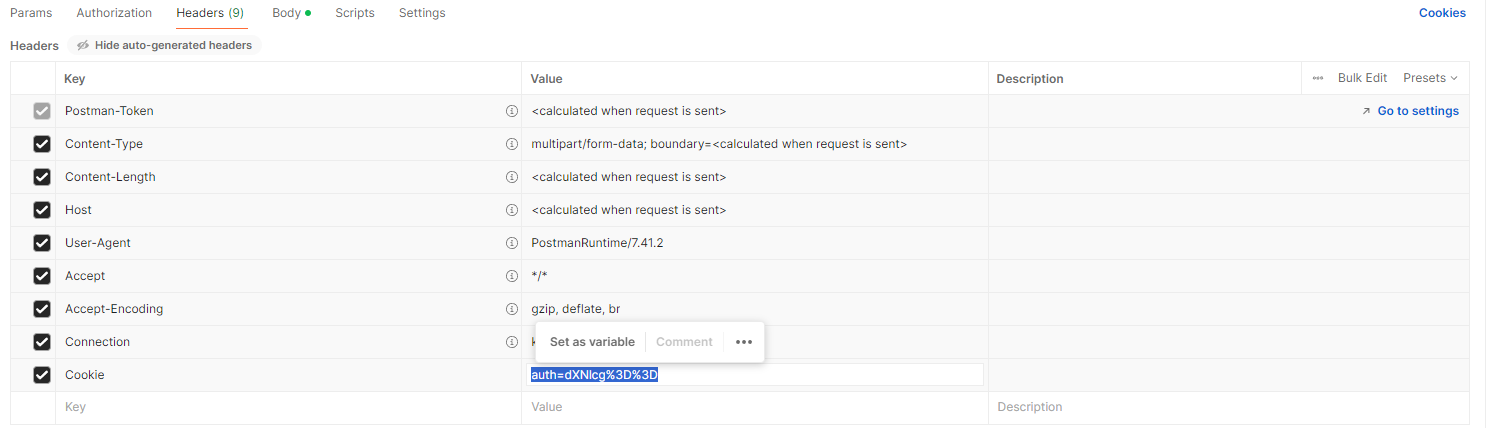


Если изменим минимальное количество отправляемых попугаев на отрицательное число.

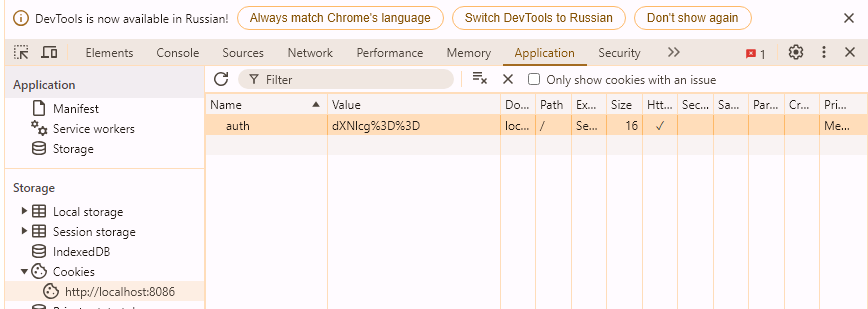
Ну и можно например использовать postman для отправки таких запросов:



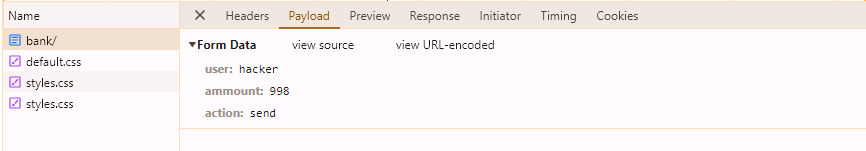
Указываем, что это post запрос и адрес, куда он должен отправляться.



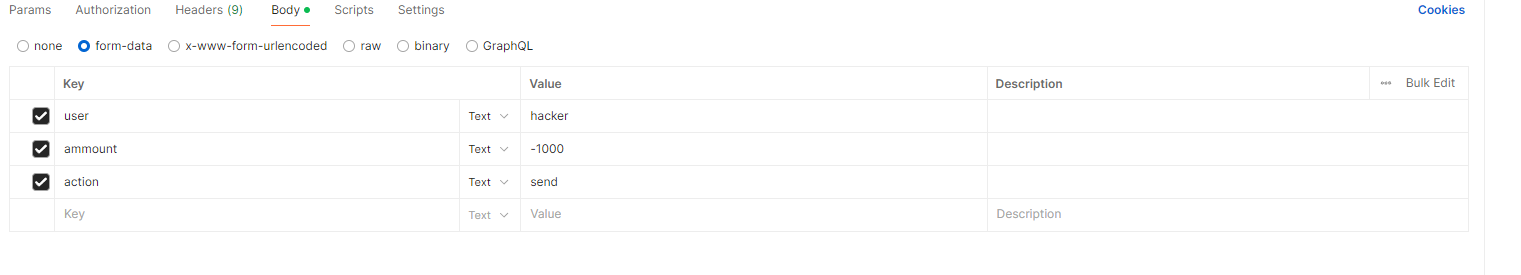
Устанавливаем Cookie в заголовках.

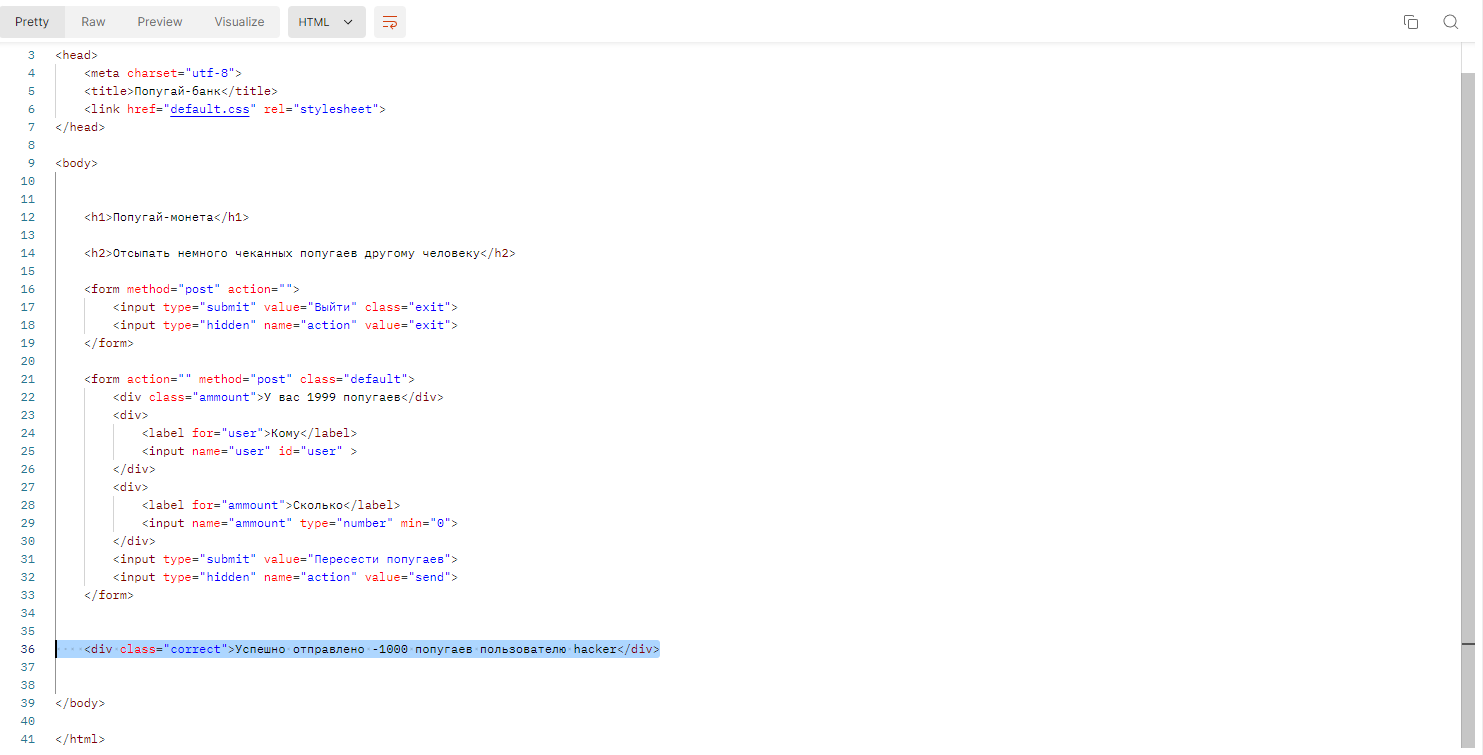


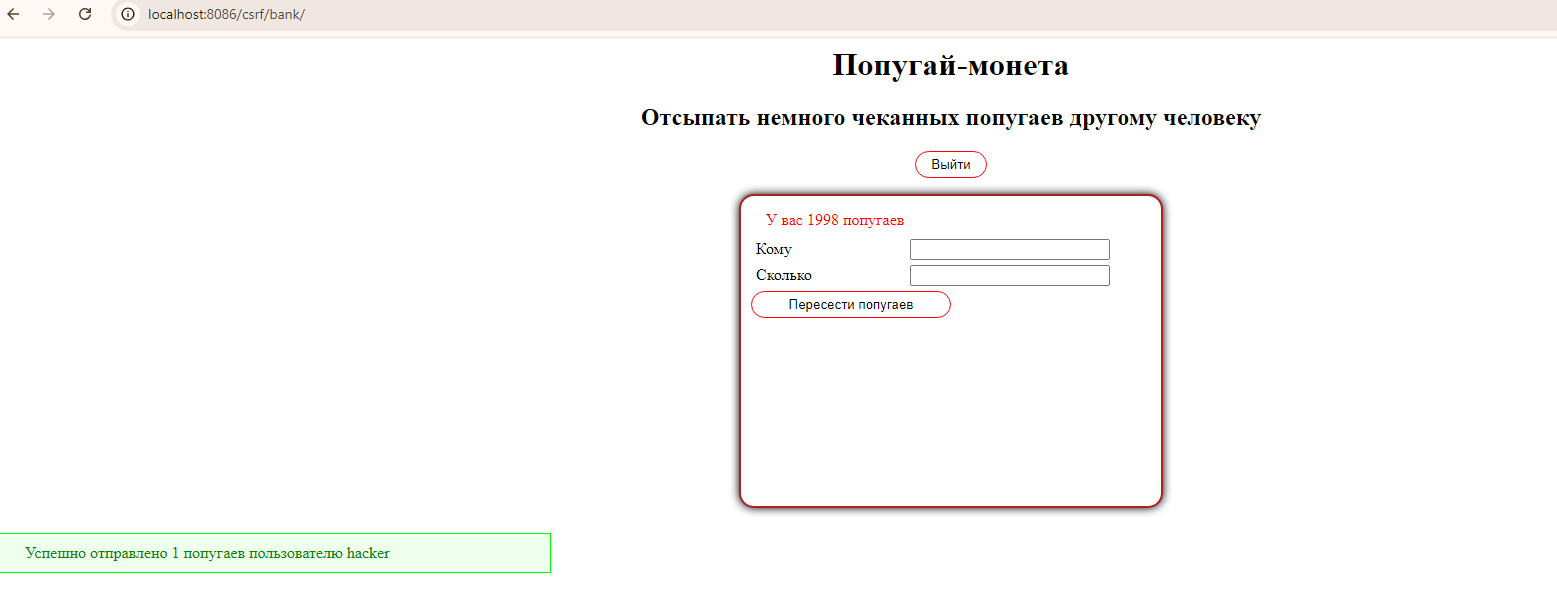
Сам Cookie можно взять нажав F12 в браузере и перейдя во вкладку Application.



Так как у нас отправляется Form Data, то в body выбираем тип Form Data и забираем наших попугаев обратно:







**1. Что происходит?**

В данном задании описывается атака типа CSRF (Cross-Site Request Forgery), или «межсайтовая подделка запроса». Основная идея этой атаки состоит в том, чтобы с одного сайта отправить запрос на другой сайт от имени авторизованного пользователя, не требуя явного согласия пользователя на выполнение этих действий.

Описание процесса атаки:

* Пользователь заходит на сайт банка и авторизуется с использованием логина и пароля. В результате успешной авторизации сервер выдаёт пользователю Cookie сессии, которая хранится в браузере.
* Когда пользователь заходит на другой сайт (в примере — сайт, где предлагается поставить лайк котику или поделиться попугаями), этот сайт может содержать скрытую форму, которая отправляет запрос на сайт банка.
* Если пользователь авторизован в банке, то при отправке запроса из формы браузер автоматически добавит Cookie сессии в этот запрос. Сервер банка примет запрос и выполнит указанные в нем действия, поскольку запрос приходит с авторизованной сессией пользователя.
* Таким образом, злоумышленник с помощью поддельного сайта может выполнить действия от имени пользователя, например, перевести деньги или поделиться попугаями, как указано в примере.

**2. Как защититься от такой атаки? Что такое CSRF-токен?**

Чтобы защититься от атаки типа CSRF, используется механизм CSRF-токенов. Это специальные случайные строки, которые генерируются сервером и вставляются в каждую форму или запрос, требующий защиты.

**Как работает защита с помощью CSRF-токенов:**

* Когда пользователь запрашивает страницу с формой (например, перевод средств), сервер генерирует уникальный CSRF-токен и добавляет его в форму.
* При отправке формы на сервер вместе с остальными данными передаётся и CSRF-токен.
* Сервер проверяет, совпадает ли CSRF-токен, полученный с запросом, с токеном, сохранённым на сервере для этой сессии.
* Если токен совпадает, запрос считается легитимным и обрабатывается.
* Если токен отсутствует или неправильный, запрос отклоняется, так как предполагается, что он был отправлен не с настоящей страницы сайта.
* Таким образом, злоумышленник не может заранее знать значение CSRF-токена и вставить его в свою поддельную форму, даже если у него есть доступ к cookie пользователя.

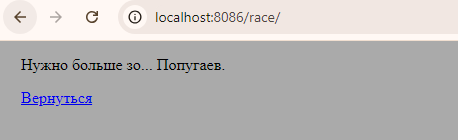
**Дополнительные меры защиты:**

* Использовать заголовок SameSite для Cookie, который запрещает их использование на сторонних сайтах.
* Проводить дополнительную валидацию на сервере, например, проверку источника запроса.
* Ограничивать допустимые значения параметров, например, не допускать отрицательных значений для перевода средств.

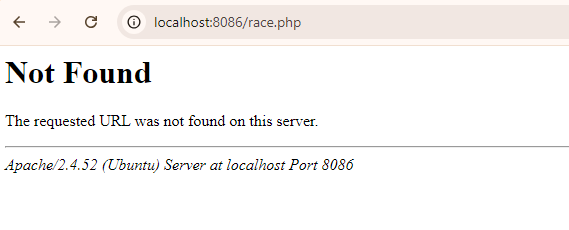
# Соревнование процессов и асинхронность



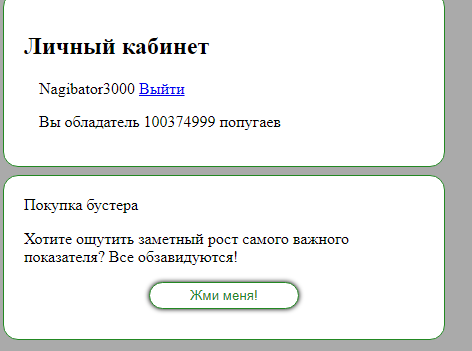
Купили буст и потратили все деньги. Новый буст мы не можем купить, так как закончились деньги, а он нам не понравился.



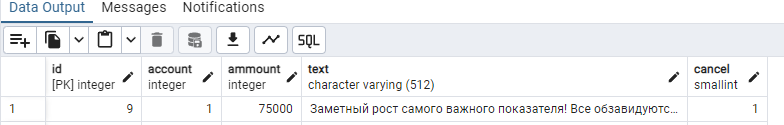
**P.S.** Кнопка вернуться не работает:



Теперь хотим попробовать его вернуть и имеем возможность нажимать много раз на кнопку «Вернуть».



В итоге мы вернули буст и получили даже лишние деньги за несколько возвращений.



Так как между процессом, который устанавливал cancel из 0 в 1 и процессом на проверку состояния буста происходило время. В итоге до того, как cancel стал 1, несколько запросов уже получило ответ, что cancel 1 и возврат возможен.

**Асинхронность и соревнование процессов**

Асинхронность и многопоточность усложняют ситуацию, так как несколько запросов могут выполняться одновременно (в разных потоках или асинхронных задачах), и сервер может не успевать синхронизировать состояние данных между этими запросами. В результате возникают ситуации, когда несколько запросов одновременно проходят проверку и выполняют изменение данных.

**Методы решения проблемы**

**1) Использование транзакций:**

* Транзакции обеспечивают атомарность операций, то есть все операции в транзакции выполняются как одно целое. Это значит, что если транзакция началась, то в рамках её выполнения ни одно из других действий не сможет изменить данные.
* В .NET можно использовать TransactionScope или методы управления транзакциями в базе данных (например, в Entity Framework есть DbContext.Database.BeginTransaction()).

**2) Блокировка данных:**

* Можно использовать блокировки на уровне базы данных (например, SELECT ... FOR UPDATE), чтобы другие процессы не могли изменять данные, пока выполняется текущая операция.
* В .NET можно реализовать блокировки в памяти с помощью SemaphoreSlim или других синхронизаторов.

**3) Контроль целостности данных:**

* Использовать контрольные признаки (флаги или временные метки) для проверки состояния данных.
* Например, добавлять временной штамп последнего изменения и проверять его перед каждым изменением.

**4) Асинхронные задачи (Task):**

* Task позволяют запускать операции асинхронно и эффективно управлять их выполнением.
* Для предотвращения гонки данных можно применять lock или другие синхронизаторы при обращении к общим ресурсам.

**5) Валидация на фронте**

* Использование атрибута disabled для кнопок и других элементов управления является хорошей практикой.
* Прелоадер (анимация загрузки) может сигнализировать пользователю о том, что запрос выполняется, и блокировать дальнейшие действия до получения ответа от сервера. Это уменьшает вероятность того, что пользователь будет пытаться выполнить одно и то же действие несколько раз.