**Отчет по найденным уязвимостям**

**Задание 1.1**

**Описание функционала выполнения кода:** веб-приложение на Flask. При GET запросе по корневому пути в браузере отображается форма загрузки файла и кнопка отправки его на сервер. POST запрос по корневому пути служит для обработки отправленного файла и сохранение его в папку upload. GET запрос /share?filename={file} откроет загруженный файл и отобразит на странице его содержимое.

**Описание найденных уязвимостей:** *Command Injection и XSS*.1)из-за обычной конкатенации строк при разрешении пути сохранения файла получается, что, задав имя файла ../customlog.py мы можем изменить поведение сервера при GET или POST по корневому пути. Внутри обработчика пути / вызывается customlog.log(). Тем самым, можно заставить сервер выполнить любой python код, реализовав в загружаемом файле функцию log(). Это может быть чтение конфига или переменных окружения, копирование в папку upload важных файлов, содержимое которых можно будет просмотреть через /share?filename=passwords.txt например. Отправить данные на удаленный сервер и т.д.

2) Также можно загрузить опасный для пользователей файл. Например html с кодом на javascript, который компрометирует cookie пользователя на веб-сервере и другие подобные файлы.

Оба примера эксплуатации представлены ниже.

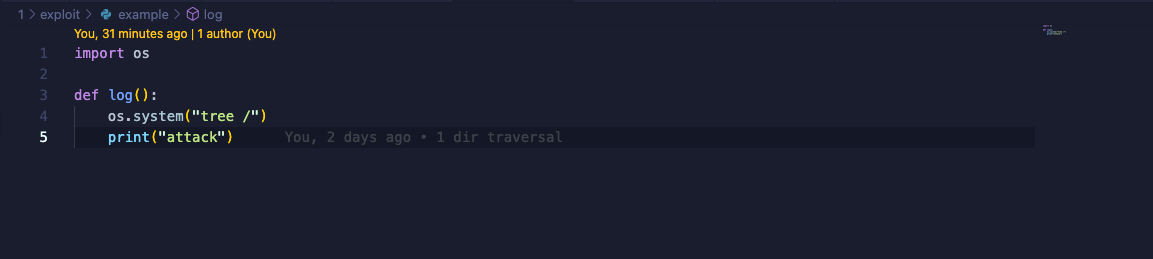
**Пример эксплойта:**

Рисунок 1 – код эксплойта

****

Рисунок 2 – отправка кода эксплойта на сервер

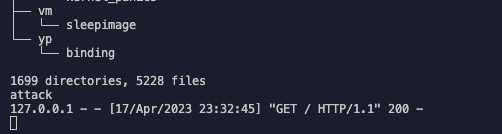


Рисунок 3 – пример эксплуатации уязвимости

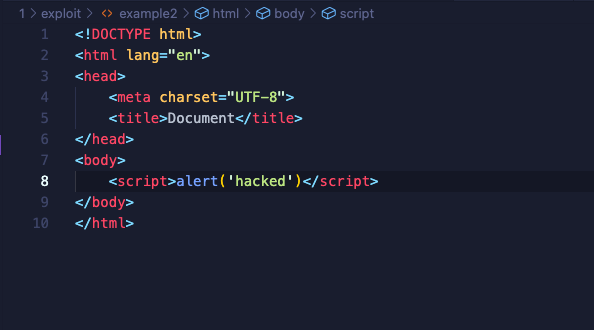


Рисунок 4 – файл с вредоносным js



Рисунок 5 – отправка вредоносного файла на сервер.

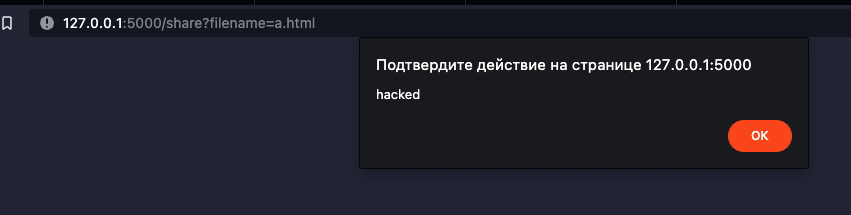


Рисунок 6 – пример эксплуатации уязвимости

**Рекомендация по устранению уязвимости:** использовать функции, 'очищающие' название загружаемого файла (например werkzeug.utils.secure\_filename(filename) для нашего примера). Также стоит проверять, какой контент содержит в себе загружаемый файл.

**Задание 1.2**

**Описание функционала выполнения кода:** загружается форма с полем для ввода электронной почты и кнопкой ‘отправить’. Значение этого поля заносится в параметр url. После нажатия кнопки, берется значение электронной почты, проверяется, что символ ‘@’ делит эту строку ровно на 2 части. Если истина – дальнейшая валидация почты и вывод ее на страницу.

**Описание найденных уязвимостей:** *XSS.*Уязвимость имеют веб-серверы, имеющие php версии ниже 8.1. ‘echo htmlspecialchars’ с url параметром адреса электронной почты заключен в кода javascript в одинарные кавычки. До версии 8.1 функция htmlspecialchars не обрабатывала их как специальный знак, а возвращала в изначальном виде. Тем самым, введя в поле email значение, начинающееся на ‘ закрывается строка в коде javascript и дальше можно внедрять XSS.

**Пример эксплойта:**

****

Рисунок 7 – код эксплойта

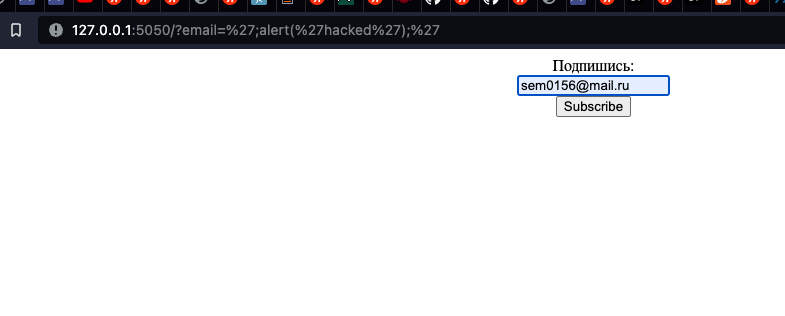


Рисунок 8 – переход по ссылке из зараженной страницы и ввод email

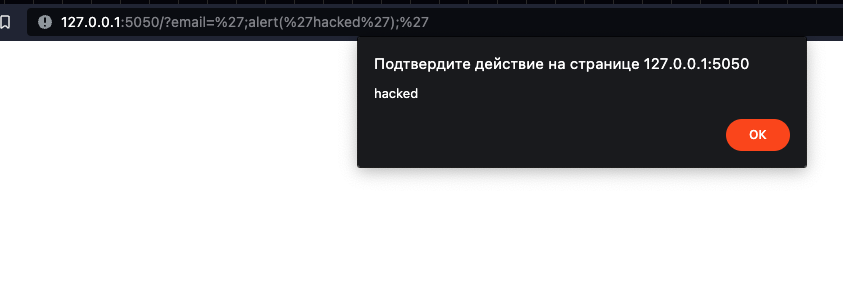


Рисунок 9 – после нажатия кнопки

**Рекомендация по устранению уязвимости:** Обрабатывать одинарную кавычку, как специальный символ и экранировать его. В нашем случае можно использовать htmlentities() или htmlspecialchars() с флагом ENT\_QUOTES, позволяющем экранировать и одинарную кавычку.

**Задание 1.3**

**Описание функционала выполнения кода:** загружается форма с полем для ввода имени пользователя и кнопка ‘START’. Страница ждет входящего события (ожидает подключение к серверу). После получения события, выводит, что подключилось и отображает на странице адрес сервера.

**Описание найденных уязвимостей:** *XSS.*Так как событие не фильтрует, откуда именно пришло событие и можно ли доверять его источнику, можно произвести XSS атаку, вставив на зараженную страницу iframe и отправить в этот iframe сообщение со скриптом (например украсть cookie).

**Пример эксплойта:**

****

Рисунок 10 – код уязвимости

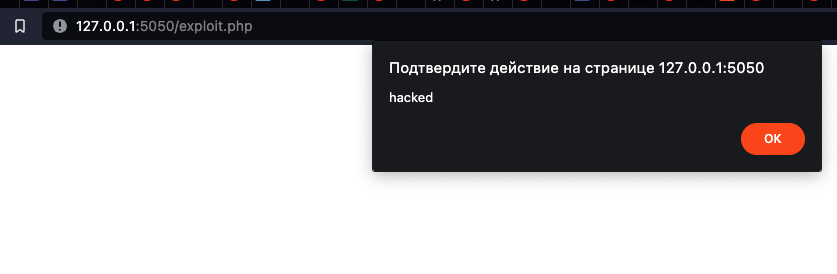


Рисунок 11 – переход на зараженную страницу

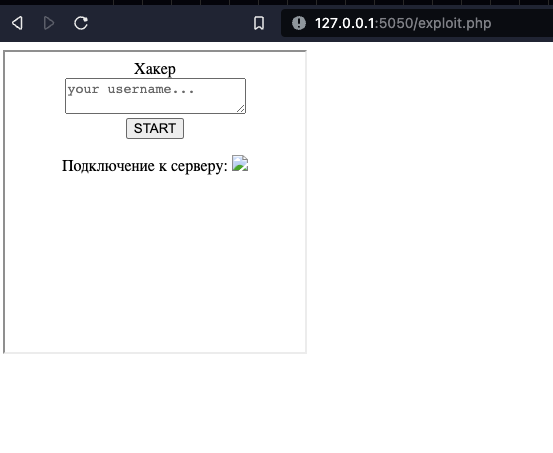
****

Рисунок 12 – отображаемый iframe

**Рекомендация по устранению уязвимости:** Сделать валидацию origin в обработчике addEventListener.

**Задание 1.4**

**Описание функционала выполнения кода:** веб-приложение на golang. Путь /admin проверяет роль пользователя и ip, с которого он отправил запрос. При успешной проверке на права администратора в браузере отображается “Logging in”.

**Описание найденных уязвимостей:** *BrokenAuth*. легко подделать параметры запроса, чтобы сервер определил пользователя, как админа. Достаточно подставить куки 'role' со значением 'admin' и заголовок 'X-Forwarded-For' со значением 'localhost' или '127.0.0.1'. Пример эксплойта на python представлен ниже.

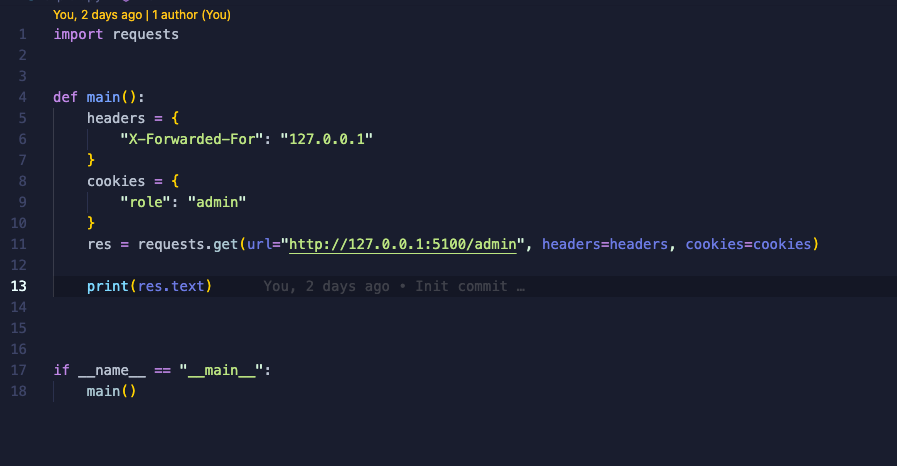
**Пример эксплойта:**

Рисунок 13 – код эксплойта

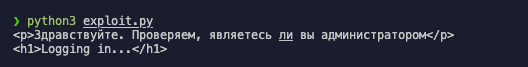
****

Рисунок 14 – эксплуатация уязвимости

**Рекомендация по устранению уязвимости:** не использовать значение куки role с простым значением, использовать сгенерированные на сервере JWT (подписанные сервером, stateless) или стандартные сессии (stateful) без возможности перебрать их значения за вменяемое время. Тогда неприятности могут возникнуть только в том случае, если какой-либо админ нечаянно "сольет" свой токен.

**Задание 1.5**

**Описание функционала выполнения кода:** приложение дает 3 попытки ввода кода доступа. Если был введен правильный код или значение переменной root отлично от нуля, загружается меню. Иначе приложение закрывается.

**Описание найденных уязвимостей:** *Buffer Overflow*. приложение использует функцию gets для считывание кода доступа. Эта функция небезопасна, так как не ограничивает количество считываемых символов и позволяет переполнить буффер. Однако, в зависимости от компилятора, эксплуатация будет разная. В одном случае, следующей в памяти за буффером может располагаться переменная root. Тогда достаточно ввести больше символов, чем вмещает буффер, чтобы поменять значение переменной root и запустилось меню. В других компиляторах, переменная root может расположиться в памяти раньше, чем буффер. Тогда перезаписать ее обычным переполнением буффера не получится. В таком случае, можно внедрить шелл-код (допустим в переменную окружения) и перезаписать адрес возврата, который находится в памяти дальше, чем буффер (они все в стеке вызовов) адресом начала шелл-кода. Когда приложение начнет раскручивать стек и дойдет до адреса возврата, оно начнет исполнять первую инструкцию шелл-кода.

**Пример эксплойта:**

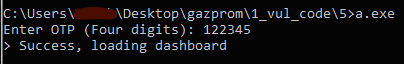
****

Рисунок 15 – эксплуатация уязвимости (при неправильно коде доступа запустилось меню)

**Рекомендация по устранению уязвимости:** использовать функции, ограничивающие количество считываемых символов, если язык программирования не отслеживает выход за пределы массива.

**Задание 1.6**

**Описание функционала выполнения кода:** веб-приложение на NodeJS. При обработке запроса происходит установка в ответ заголовков Access-Control-Allow-Origin, Access-Control-Allow-Credentials. Они должны не позволить другим серверам запросить учетные данные. Затем происходит вставка в ответ учетных данных в формате JSON и отправка ответа клиенту.

**Описание найденных уязвимостей:** *CORS Misconfig.* Origin берется из заголовка origin, который можно подменить. Возникает проблема, что данные можно получить в обход ограничений.

**Пример эксплойта:**

****

Рисунок 16 – код эксплойта

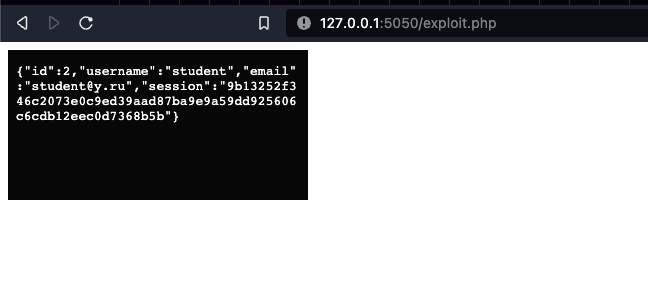


Рисунок 17 – эксплуатация уязвимости.

**Рекомендация по устранению уязвимости:** включить в исходный код, какие ресурсы считать безопасными (белый список). И позволять делать запросы только им.

**Задание 1.7**

**Описание функционала выполнения кода:** если в качестве query параметра передан file={filename} выводит содержимое этого файла, иначе выводится страница index.html.

**Описание найденных уязвимостей:** *LFI*.В query параметр file передать значение абсолютного пути (например /etc/passwd или /root/.ssh/id\_rsa, если выполняется от имени root). В /etc/passwd можно перебрать пользователей и их директории и просмотреть их файлы (приватные ключи, конфиги).

**Пример эксплойта:**

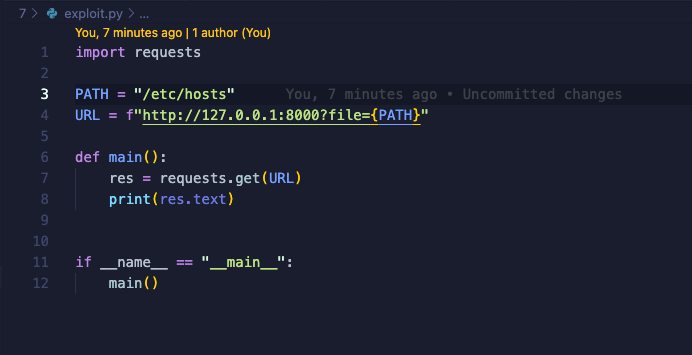
****

Рисунок 18 – пример кода эксплойта

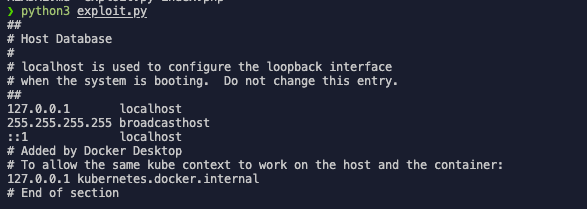


Рисунок 19 – эксплуатацию уязвимости

**Рекомендация по устранению уязвимости:** проводитьпроверку, что файл находится в директории проекта, например в php при помощи str\_starts\_with(filePath, projectPath)

**Задание 1.8**

**Описание функционала выполнения кода:** при открытии страницы выполняется перенаправление по переданному в query параметре r пути. Происходит базовая фильтрация пути, пытаясь избежать точек и слешей.

**Описание найденных уязвимостей:** *OpenRedirect и XSS*. Во-первых можно поэксплуатировать XSS, например так: http://127.0.0.1:8000/?r=javascript:alert(%27attacked%27), есть ограничение - точки внутри скрипта заменяются на \_, выражение должно быть без точек. Во-вторых можно использовать OpenRedirect, заменив стандартную точку китайской, например: http://127.0.0.1:8000/?r=https:xakep%E3%80%82ru

**Пример эксплойта:**

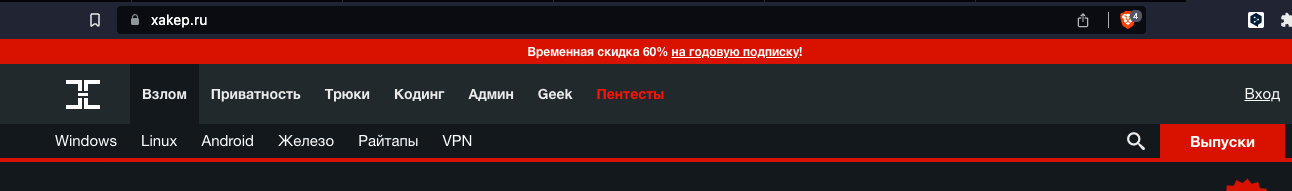
****

Рисунок 20 – эксплуатация уязвимости (перенаправление на сторонний ресурс)

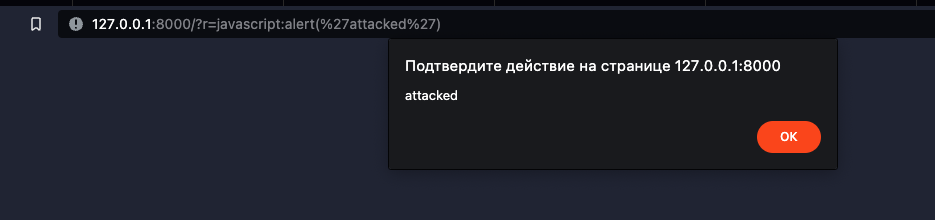


Рисунок 21 – эксплуатация уязвимости (выполнение javascript)

**Рекомендация по устранению уязвимости:** Проверять введенноеы в параметр r значение через белый список (константные строки или регулярное выражение), чтобы избежать XSS и OpenRedirect на нежелательные ресурсы.

**Задание 1.9**

**Описание функционала выполнения кода:** веб-приложение на Flask с использованием шаблонов. Обработчик пути /home.htmlберет query параметр search из запроса и ищет в базе данных продукты c этим параметром. Если продукты не найдены, выводит шаблон ошибки 404, в который подставляется значение из параметра search.

**Описание найденных уязвимостей:** *SSTI*. Подставленный в шаблон параметр можно представить в качестве исполняемого кода. Достаточно в качестве query параметра search например передать {{config.items()}} исходя из вывода можно узнать конфиг веб приложения, а там может содержаться информация о базе данных (хост, логин, пароль), тем самым можно выкрасть важные данные.

**Пример эксплойта:**

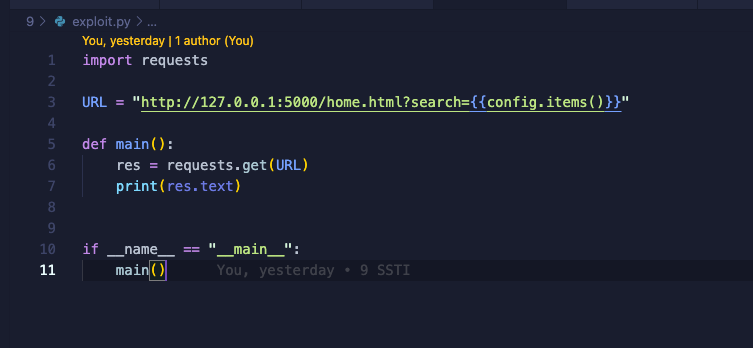
****

Рисунок 22 – пример кода эксплойта



Рисунок 23 – эксплуатация уязвимости

**Рекомендация по устранению уязвимости:** очищать query параметр search от {}, а лучше сделать возможными только [a-z, A-Z, 0-9].

**Задание 2**

**Описание функционала выполнения кода:** веб-приложениена Django с использованием Django ORM и базы данных PostgreSQL. С помощью Django orm описана модель WebLog (method, url, user\_agent, created\_time), сохраняющая в базу данных информацию о http запросах. При http запросе выводится JSON с информацией о количествах запросов в определенный период времени. Этот период времени задается url параметром date (по умолчанию показывает данные по минутам, но можно сделать по часам, дням и т.д.). В обработчике запроса вызывается функция сохранения информации об этом запросе (модель WebLog).

**Описание найденных уязвимостей:** *SQL Injection.*Url параметр date задается пользователем и используется в запросе к БД с помощью Django ORM. С определенным значением date можно внедрить sql код и достать важную информацию из базы данных. Примеры приведены ниже. Также возможно, подменив не только url параметр date, но и user-agent и url, так как они используются при генерации запроса на сохранение информации о запросе, можно тоже сделать sql инъекцию, но не удалось это проверить.

**Пример эксплойта:**

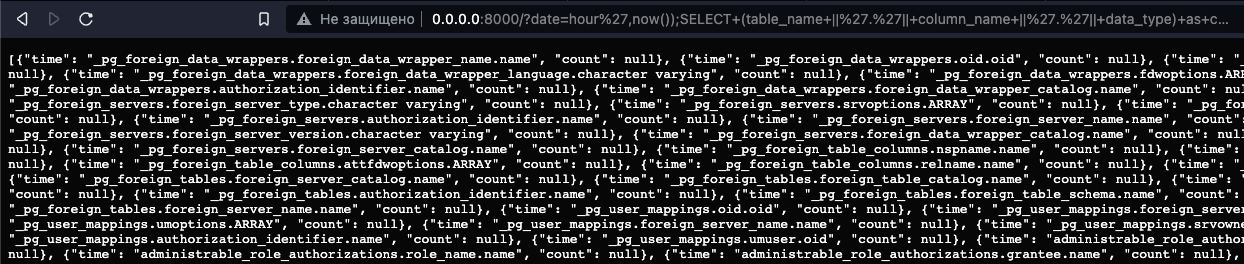
****

Рисунок 24 – пример эксплойта (запрос информации о таблицах, полях и типов полей).

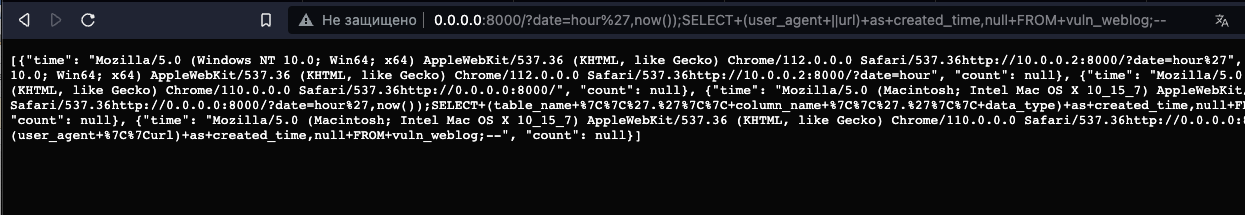


Рисунок 25 – пример эксплойта (вывод user-agent, содержащихся в БД)

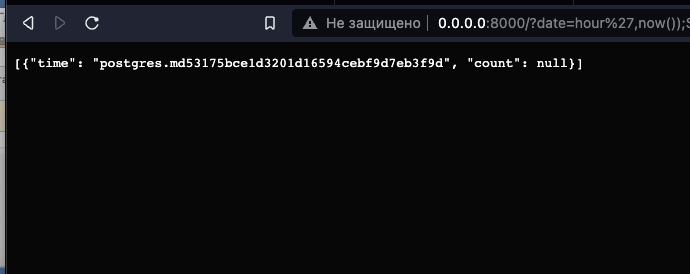


Рисунок 26 – пример эксплойта (вывод имени пользователя и хэша его пароля)



Рисунок 27 – удаление таблицы vuln\_weblog

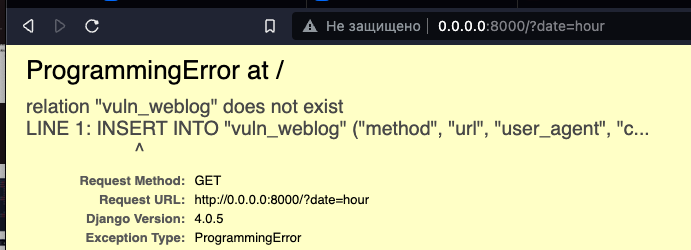


Рисунок 28 – таблица удалена

**Рекомендация по устранению уязвимости:** проверять и очищать введенные пользователями данные (с помощью готовых функций или реализовать самостоятельно).