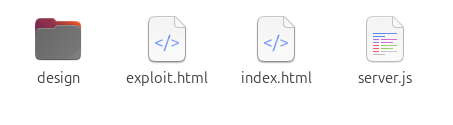
1. Собрали html - страницу, подняли локально сервер на :3000

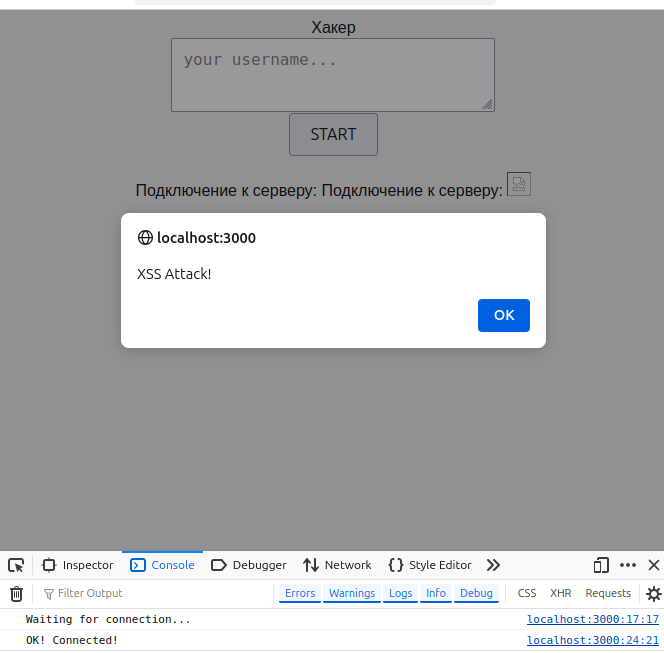


//для визуализации нашёл типовой сервер и немного подредактировал код для эксплуатации

1. Входные данные вставляются в DOM через innerHTML. Отсюда – недостаточная обработка входных данных позволяет произвести XSS – атаку.
2. Код не проверяет источник сообщений (event.origin). Отсюда любой сайт может отправлять сообщения
3. Внешний файлик design.css. При наличии доступа у злоумышленника – может повлиять на безопасность

**КАК ВОСПРОИЗВЕСТИ:**

1. Стандартная атака была выполнена в «Пример2отчёт». Упомянутое изменение кода позволило визуализировать другой эксплоит, но также касающийся XSS.



1. Написанный expoit.html позволяет максимально визуализировать уязвимость XSS в index.html. По сути, он отправляет сообщение через postMassage, наше приложение получает данные (вредонос) и вставляет в <p id="out"> через innerHTML. Отсюда он запускает JS, пытается загрузить несуществующее изображение и при ошибки – всплывает окно (рисунок выше).

**Как защитить:**

1. Изменим метод вставки данных с innerHTML на textContent:

*document.getElementById('out').textContent += ` ${address}`;*

1. Добавим экранирование. Если мы всё равно хотим использовать **innerHTML** (например, для форматирования текста), экранируем данные перед вставкой:

function escapeHtml(unsafe) {

return unsafe.replace(/[&<>"']/g, function (c) {

return {

'&': '&amp;',

'<': '<',

'>': '>',

'"': '&quot;',

"'": '&#039;'

}[c];

});

}

// Получение адреса сервера:

console.log("Waiting for connection...");

window.addEventListener("message", (event) => {

console.log("[OK] Connected!");

const address = event.data;

// Экранируем данные перед вставкой

const safeAddress = escapeHtml(address);

document.getElementById('out').innerHTML += `<u>${safeAddress}</u>`;

});

1. Можно добавить пулл разрешённых источников и подтянуть валидацию для них, если имеется такая возможность