

Курс «Веб-технологии: уязвимости и безопасность»

HTML и CSS

Основы разметки HTML, стили CSS, модель документа DOM и парсеры, HTML-эскейпинг, гиперссылки, HTML-формы и iFrame

Оглавление

[Введение](#)

[Видеоурок 1](#)

[Базовые концепции HTML](#)

[Теги в HTML](#)

[Структура HTML-документа](#)

[Практика](#)

[Итоги](#)

[Видеоурок 2](#)

[Базовые концепции CSS](#)

[Стили через атрибут тега](#)

[Стили через тег](#)

[Стили через class](#)

[Стили из CSS-файла](#)

[Итоги](#)

[Видеоурок 3](#)

[Парсеры HTML и DOM](#)

[Парсинг и проблемы безопасности](#)

[Инъекции в HTML](#)

[XSS-инъекции](#)

[Эскейпинг HTML](#)

[HTML-сущности](#)

[Итоги](#)

[Видеоурок 4](#)

[Гиперссылки и загрузка контента](#)

[Атрибут target](#)

[HTML-формы](#)

[Тег iframe и атака Click-Jacking](#)

[Атрибут src](#)

[Итоги](#)

[Итоги урока](#)

[Ссылки к уроку](#)

Введение

На четвертом уроке мы поговорим про HTML — язык разметки веб-документов — и стилевые возможности их оформления — CSS.

Мы успешно завершили разбор URL и HTTP и теперь приступим к изучению HTML и CSS — как они устроены, для чего нужны — и узнаем, какие проблемы безопасности связаны с ними.

План урока:

- 1) Что такое HTML, базовые действия, понимание его работы.
- 2) Что такое CSS и для чего он нужен.
- 3) Как происходит парсинг HTML, какие у него особенности и как это соотносится с безопасностью. Узнаем, что такое HTML Escape-пинг, для чего он нужен и как применяется.
- 4) Гиперссылки и включение другого контента на страницу.

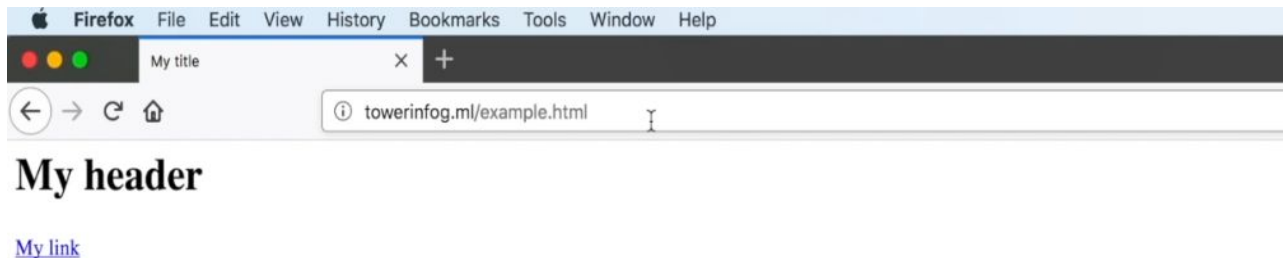
В конце урока вы научитесь писать веб-странички на HTML, украшать их CSS-стилями и понимать, как это все парсится, как это внутри работает и какие последствия безопасности это за собой влечет. Также разберетесь, что такое HTML Escape-пинг будете знать, когда его нужно применять и уметь это делать.

Видеоурок 1

Мы можем читать HTTP-заголовки, знаем методы HTTP, заголовки запроса и ответа. Но ниже заголовков мы еще пока не опускались. Мы знаем, что там находится тело ответа или тело запроса. Когда мы запрашиваем страницу, чаще всего там находится HTML. Что же такое HTML, из чего он состоит и как устроен?

Базовые концепции HTML

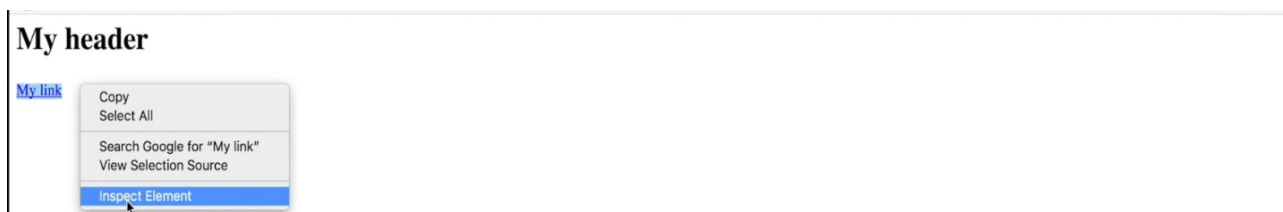
Начнём с примера. Откроем такую тестовую страничку:



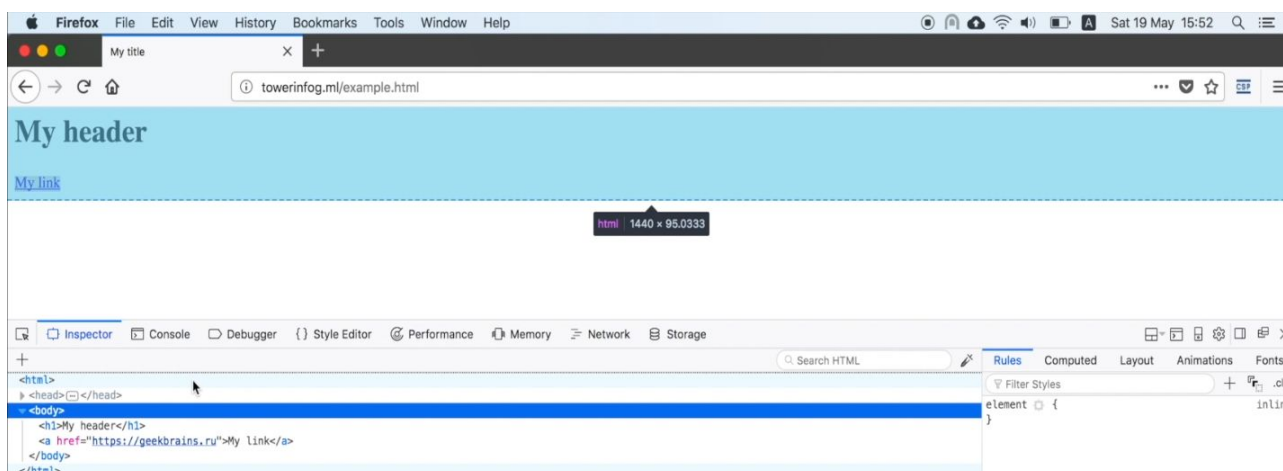
На странице 3 основных элемента:

- 1) Написанный жирным шрифтом текст **My header**.
- 2) Также у страницы есть заголовок — **My title**.
- 3) И есть одна ссылка — **My link**. Как мы видим в левом нижнем углу, это ссылка на <https://geekbrains.ru>.

Посмотрим подробнее на код страницы. Для этого откроем в браузере инструменты разработчика — в Firefox это можно сделать, нажав правой кнопкой мыши и выбрав **Inspect Element**:



В этом окне мы видим, как устроен этот HTML-документ:



Мы видим, что здесь есть теги `<html>`, `<head>`, `<body>`.

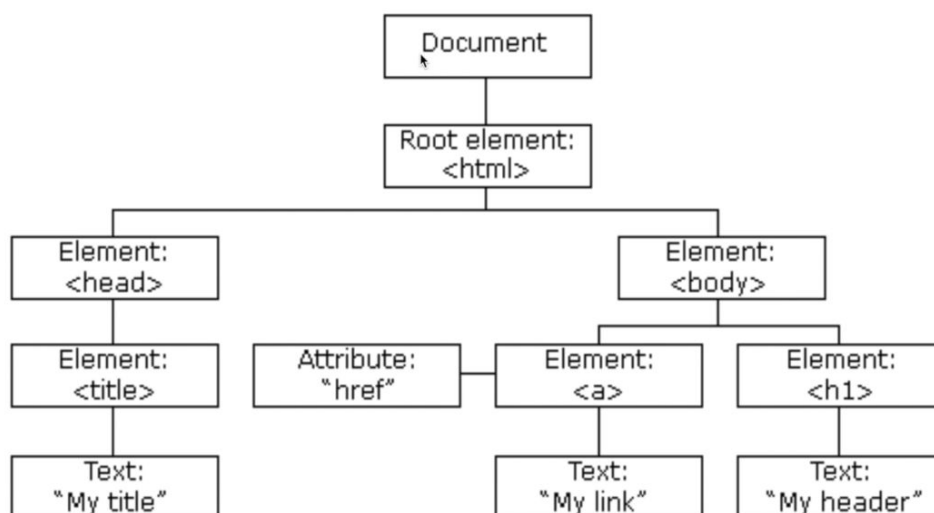
Теги в HTML

То, что заключено в угловые скобки (`<..>`) — это HTML-теги. Тег — объект в языке разметки HTML, который представляет отдельную сущность в языке. Если сравнивать, например, с реальным миром, тегом можно назвать любой предмет — тег дивана или тег стены, и т. п.

Также у тегов есть так называемые атрибуты. Например, у тега `<a>` есть атрибут ``. Атрибуты — это уже знакомые нам «ключ=значение». Ключ — имя атрибута, по которому мы к нему обращаемся, а значение — то, что он содержит. Если сравнить с реальным миром, то атрибут дивана и стены — это, например, их цвет, т. е. белый диван или красная стена. Еще пример атрибута — стена кирпичная, то есть материал — кирпич.

Основа HTML — теги, у тегов есть атрибуты, также есть просто текст и все это выстраивается в определенную древовидную структуру. Рассмотрим древовидную структуру на схеме, чтобы вспомнить, как это выглядит.

Структура HTML-документа



На схеме есть корень дерева — в данном случае это сам документ. Это та вершина, у которой нет родителя, значит она ниоткуда не начинается и сама начинает все дерево.

Дальше у HTML-документа есть корневой элемент `<html>` — он только обозначает начало структуры HTML-документа.

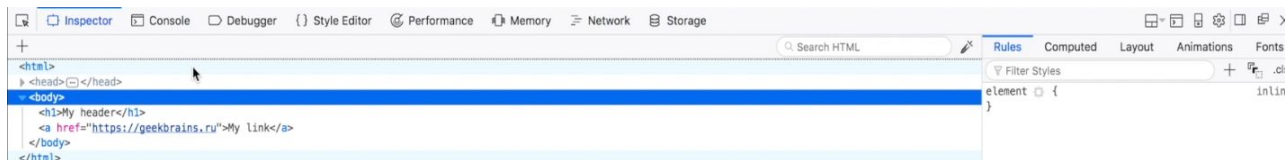
Дальше есть элемент `<head>` и заголовок `<title>` — они отвечают за то, что написано в заголовке HTML-документа.

Еще одна из ветвей дерева — `<body>`, у неё есть два потомка:

- тег `<a>` — ссылка **My link**.

- элемент `<h1>`, который обозначает текст-заголовок, выделенный крупным и жирным шрифтом — My header.

Все это — классическая модель представления дерева. Здесь в Inspector в инструментах разработчика вы тоже видите структуру документа как дерево HTML-документа:

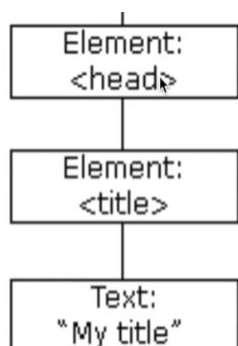


Здесь он представлен немного по-другому, но можно сопоставить предыдущую картинку с этой. Как вы видите, здесь в корне есть тег `<html>`, дальше идут его потомки `<head>` и `<body>`.

Тегов и атрибутов великое множество, часть из них мы рассмотрим в этом уроке, другую часть — по ходу курса, а какие-то из них вы можете изучить самостоятельно.

Бывают открывающие и закрывающие теги. Например, заголовок `<head>` — это открывающий тег, а `</head>` со слэшем — закрывающий.

То, что находится между открывающим и закрывающим тегами, отображается на дереве HTML-документа как потомок:



Например, у заголовка `<head>` есть потомок `<title>`. Все, что находится между `<head>` и `</head>` — потомки элемента `<head>`.

Практика

Чтобы протестировать страничку у себя на веб-сервере, создайте новый html-файл:

```
nano example.html

<html>
  <head>
    <title>My title</title>
```

```
</head>
<body>
  <h1>My header</h1>
  <a href="https://geekbrains.ru">My link</a>
</body>
</html>
```

Это обычный файл, в котором записан HTML-код тестовой странички — example.html. Вам нужно это написать и разместить у себя на сервере в директории `/var/www/html`. Потом можно будет открывать его в браузере виртуальной машины через ссылку <http://localhost/example.html>.

Вы можете написать что-то более сложное — поиграть с атрибутами и посмотреть, как это будет отображаться на странице. Попробовать это определенно стоит, хотя бы потому, что HTML-страницы вам в любом случае придется писать, чтобы понимать, как они работают, или, например, составлять вектора атак и конкретные примеры атак. Если вы чего-то не знаете, нужно уметь найти это в интернете и разобраться с неизвестными понятиями.

Итоги

Подведем итоги этого видеоурока:

- 1) Вы узнали, что такое HTML, увидели, из каких элементов он состоит, и научились использовать базовый синтаксис языка разметки веб-документов HTML.
- 2) Узнали, что такое атрибуты и теги в HTML, разобрались, для чего каждый из них нужно использовать.
- 3) Увидели, что HTML имеет древовидную структуру и поняли, как он представляется в виде дерева. В дальнейшем нам это пригодится, чтобы понимать, как работает парсер HTML — программа, которая смотрит на исходный код и понимает, как нужно его отобразить в браузере.

Видеоурок 2

В этом видеоуроке мы узнаем:

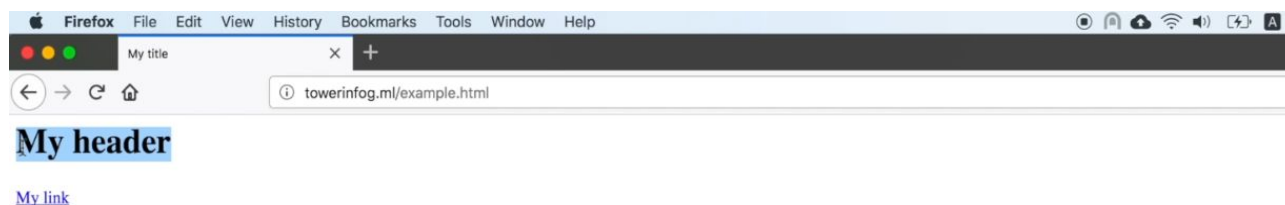
- 1) Что такое CSS, для чего он нужен.
- 2) Основной синтаксис CSS и применение CSS на практике.

Базовые концепции CSS

Вы увидели несколько разных тегов, поняли, что разные теги добавляют элементы на сайте. Но если бы на всех сайтах были одинаковые элементы, это было бы скучно. Чтобы изменить внешний вид элементов на сайте, придумали CSS.

CSS расшифровывается как каскадные таблицы стилей. Это своеобразный способ описания, как выглядят элементы на сайте, какие стили к ним применены.

Посмотрим на примере — откроем страничку example.html:



Сейчас она выглядит так же, как и до этого. Теперь попробуем сделать заголовок **My header** зеленым. Для этого откроем терминал и отредактируем файл example.html:

```
nano example.html

<html>
  <head>
    <title>My title</title>
  </head>
  <body>
    <h1>My header</h1>
    <a href="https://geekbrains.ru">My link</a>
  </body>
</html>
```

Есть несколько способов добавить CSS-стили в HTML, для начала рассмотрим первый способ — добавление стилей через атрибут тега.

Стили через атрибут тега

Так как мы хотим изменить цвет заголовка, добавим атрибут в тег `<h1>`. Вообще правильно писать значение атрибута в кавычках (""), но даже если вы их не напишете, парсер HTML все равно все правильно обработает. Об этом подробнее мы поговорим в одном из следующих видео, сейчас запомним, что парсер HTML доделывает за вас все что угодно или практически все что угодно.

Напишем `<h1 style="color: green">My header</h1>` - то есть «цвет: зеленый» и сохраним:

```
<html>
  <head>
    <title>My title</title>
  </head>
  <body>
    <h1 style="color: green">My header</h1>
    <a href="https://geekbrains.ru">My link</a>
  </body>
</html>
```

И теперь откроем Firefox и перезапустим страницу:



Заголовок стал зеленым, способ сработал.

Если на странице не один заголовок, а, допустим, 10, 100, 200, стиль не нужно прописывать в каждом атрибуте.

Стили через тег

Откроем терминал, уберем стиль в заголовке и сделаем несколько заголовков `<h1>`:

```
<html>
..
  <body>
    <h1>My header</h1>
    <h1>My header</h1>
    <h1>My header</h1>
    <h1>My header</h1>
    <h1>My header</h1>
    <h1>My header</h1>
    <h1>My header</h1>
    <h1>My header</h1>
    <a href="https://geekbrains.ru">My link</a>
  </body>
</html>
```

Можем воспользоваться другим способом — добавить тег `<style>` в начале тела документа:

```
<html>
..
  <body>
    <style>

    </style>

  </body>
</html>
```

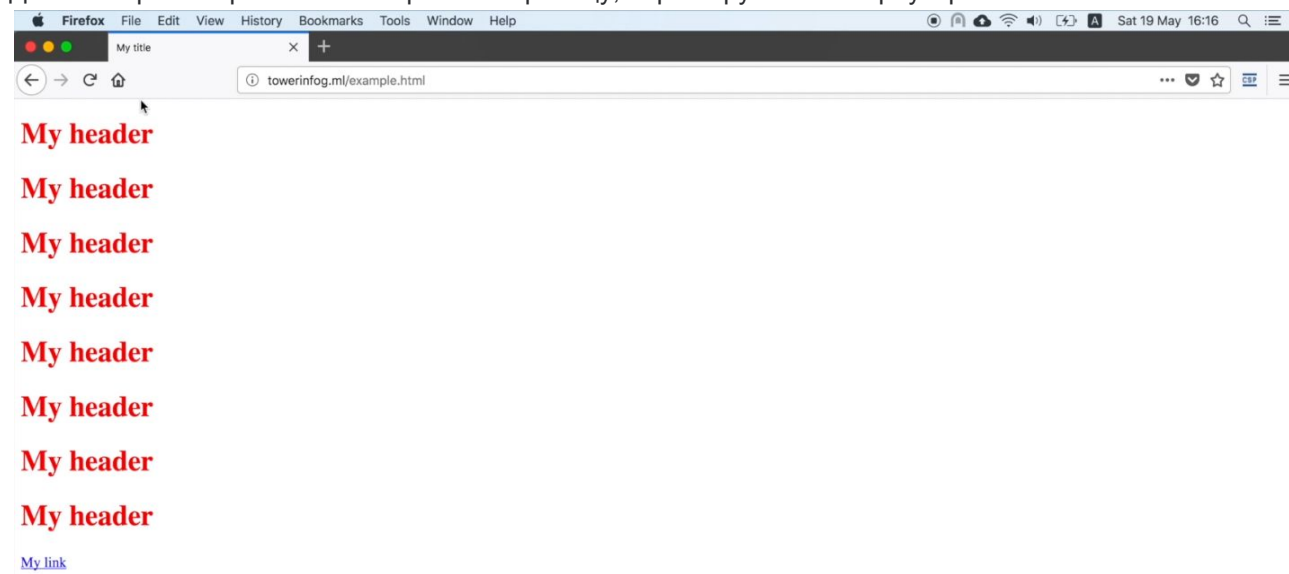
Одно и то же слово в разных контекстах может означать как атрибут, так и тег.

Вообще синтаксис CSS немного отличается от того, что мы делали в атрибуте, и выглядит следующим образом. Сначала мы говорим, к чему применить стиль. Например, к тегу <h1>. Пишем h1, ставим пробел, а дальше все стили, которые хотим применить к нему, заключаем в фигурные скобки:

```
<html>
..
    <body>
        <style>
            h1 {
                color: red;
            }
        </style>
    ..
</html>
```

Это можно было бы написать в одну строку, но чтобы было удобнее читать, пишем на нескольких строках и делаем отступы.

Далее сохраним файл и посмотрим на страницу, перезагрузим ее в браузере:



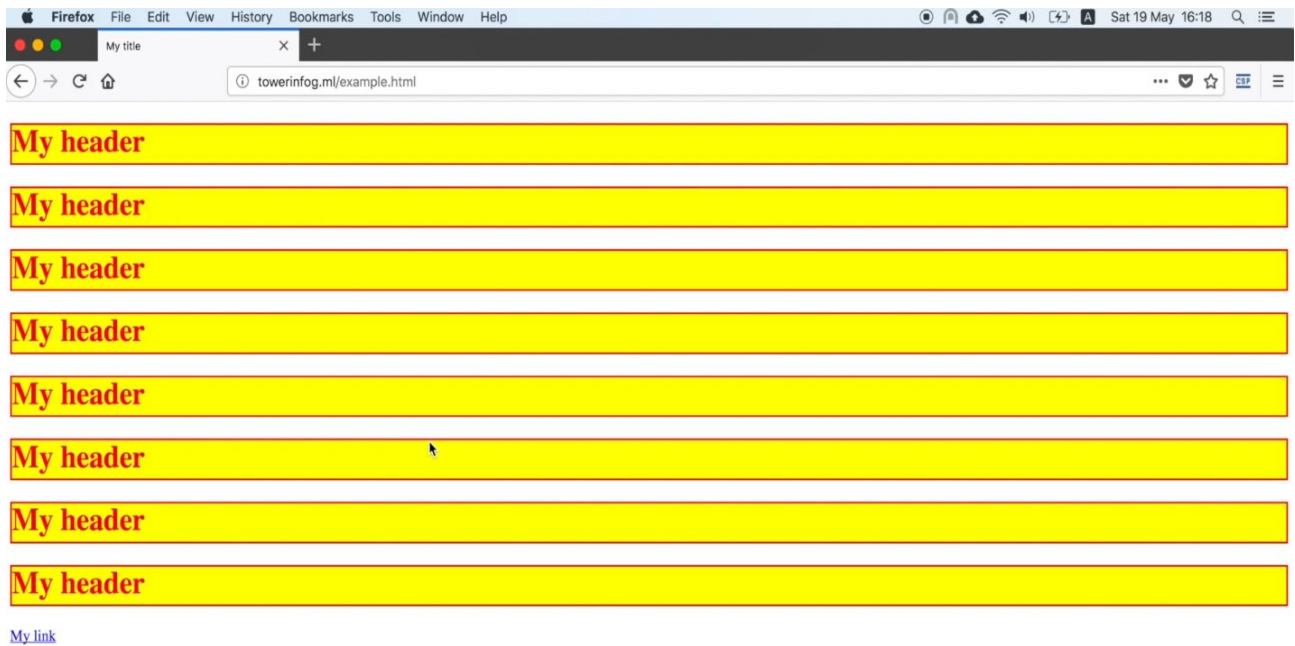
Все заголовки добавились и теперь они все красные.

Рассмотрим еще несколько CSS-атрибутов:

```
<html>
..
    <style>
        h1 {
            color: red;
            border: 2px solid;
            background-color: yellow;
        }
    </style>
..
```

```
</html>
```

Теперь у заголовков желтый фон и красная рамка толщиной в 2 пикселя, как мы и указали:



Также мы можем задавать размер элементов, шрифт внутри них, поворот и наклон — делать с ними практически все, что угодно.

Часто возникают ситуации, когда к одному элементу в разных местах нужно применить разные стили. Например, один заголовок должен быть зеленый, другой — синий, а третий должен быть в два раза больше, чем все предыдущие.

Стили через class

Можно в каждом заголовке прописать атрибут `<h1 style="...">` и добавить нужный стиль, но давно известно, что человек существо ленивое и мы не любим много ручной работы. Поэтому возьмем другой тип CSS-селектора, который использует атрибут `class`.

В CSS-стиле элемент `h1` (в данном случае то, что идет до фигурных скобок) называется селектором, потому что `select` по-английски — выбирать. В данном случае выбирается тег `<h1>` и поэтому стиль в фигурных скобках применяется к `<h1>`:

```
<body>
  <style>
    h1 {
      color: red;
      border: 2px solid;
      background-color: yellow;
    }
  </style>
```

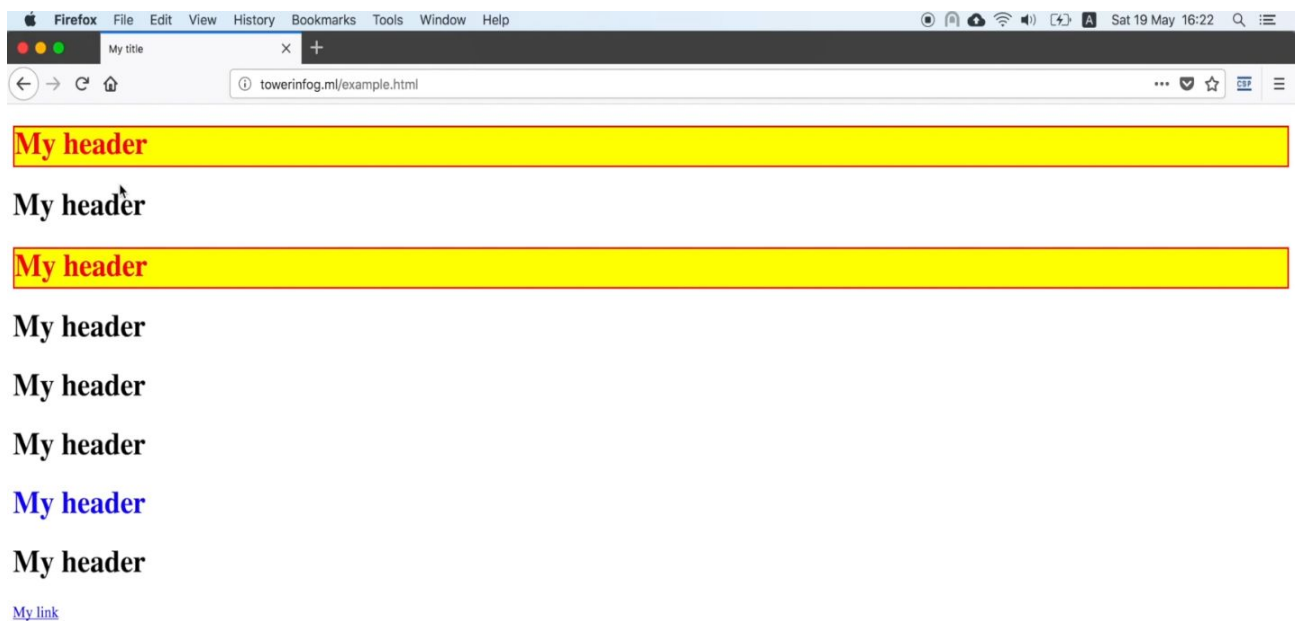
Теперь познакомимся с селекторами class (класса). Напишем здесь . (точка), т. к. селектор класса начинается с точки, и название нашего класса. Добавим 2 класса — .cl и .ss:

```
<html>
..
  <body>
    <style>
      .cl {
        color: red;
        border: 2px solid;
        background-color: yellow;
      }
      .ss {
        color: blue;
      }
    </style>
  </body>
..
```

Посмотрим, как применить к заголовкам <h1> эти классы. Если мы вернемся на страницу естественным образом, заголовки примут обычный стиль, а чтобы одна их часть была красная, а другая — синяя, мы должны применить к первым класс .cl, а ко вторым .ss — это делается с помощью атрибутов <h1 class="..">:

```
..
  </style>
  <h1 class="cl">My header</h1>
  <h1>My header</h1>
  <h1 class="cl">My header</h1>
  <h1>My header</h1>
  <h1>My header</h1>
  <h1>My header</h1>
  <h1 class="ss">My header</h1>
  <h1>My header</h1>
  <a href="https://geekbrains.ru">My link</a>
</body>
</html>
```

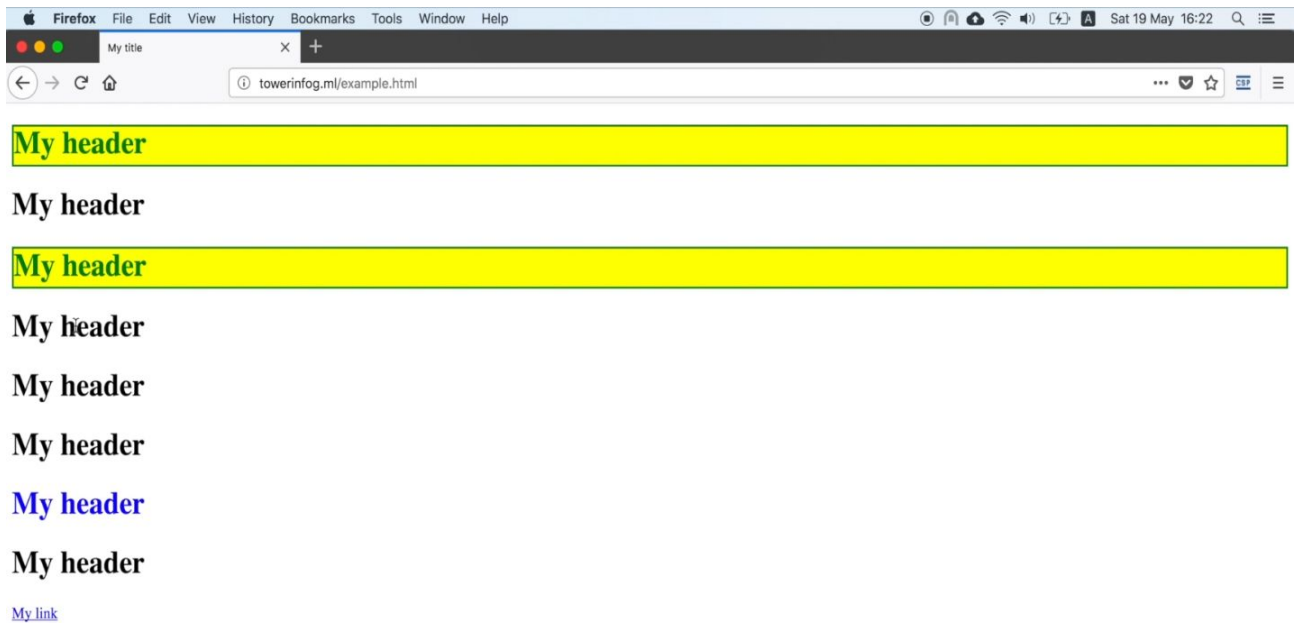
Как мы видим, класс внутри тега <h1> пишется уже без точки. Перезагрузим страницу: те заголовки, к которым мы применили класс cl, стали красными, а другие — синими.



Если нам не нравятся здесь красные заголовки, и мы хотим зеленые, переходим в файл и меняем цвет с red на green (зеленый):

```
<html>
..
  <body>
    <style>
      .cl {
        color: green;
        border: 2px solid;
        background-color: yellow;
      }
    ..
  </html>
```

Теперь перезагружаем — и ко всем заголовкам, у которых есть класс cl, применится зеленый цвет:



Стили из CSS-файла

Есть ещё один способ — когда CSS можно вынести в отдельный файл.

Создадим новый файл style.css и перенесем туда содержимое стилей:

```
nano style.css

.cl {
    color: red;
    border: 2px solid;
    background-color: yellow;
}

.ss {
    color: blue;
}
```

А в основном файле example.html сделаем ссылку на этот файл и скажем, что это стили CSS:

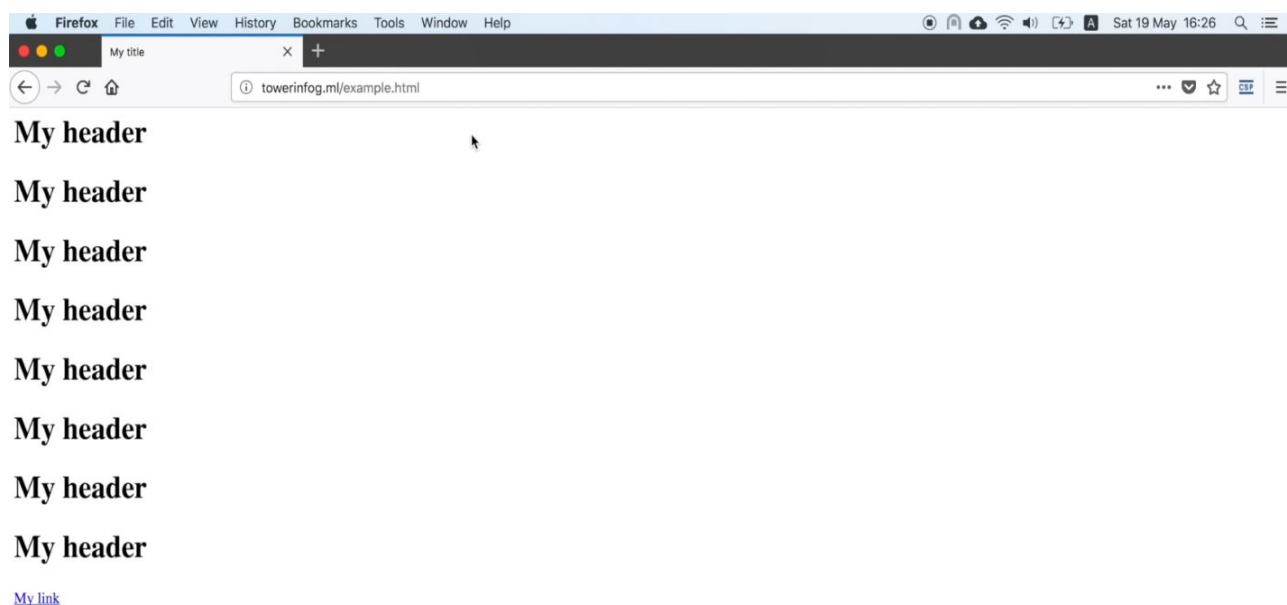
```
nano example.html

<html>
  <head>
    <title>My title</title>
    <link rel="stylesheet" href="style.css">
  ..
</html>
```

Попробуем перезапустить страницу — ничего не изменится. Если убрать тег `<link>`, то стили у заголовков тоже пропадут. Используем для этого комментарии в HTML:

```
<html>
  <head>
    <title>My title</title>
    <!--link rel="stylesheet" href="style.css"-->
    ..
  </html>
```

Перезагрузим страницу:



Как мы видим, стили больше не применяются.

Разберем, что происходит в примере. Тег `<link rel="..">` на самом деле берёт файл `style.css` и вставляет его содержимое в эту страницу, за нас это делают браузеры. Такое разделение на разные файлы удобно, когда мы пишем большие страницы: если бы всё было в одном файле, мы бы запутались, проще, когда много небольших файлов.

Итоги

В этом видеоуроке вы узнали:

- 1) Что такое CSS, для чего он нужен.
- 2) Основной синтаксис CSS, как можно его подключать разными способами.

В HTML есть два вида подключения ресурсов: можно делать это так называемым инлайн-способом, когда мы в HTML или атрибуте пишем содержимое ресурса: открываем теги `style`, добавляем как атрибут и пишем там `color="red"`. Также есть подключаемые файлы, например, через `<link rel="..">` можно подключать ресурсный файл со стилями CSS.

Видеоурок 3

В этом видео мы разберем:

- 1) Парсинг.
- 2) Парсинг HTML в браузере и особенности у HTML-парсеров.
- 3) HTML-эскейпинг, зачем он нужен, как и когда его применять.

Парсеры HTML и DOM

Парсер нужен, чтобы интерпретировать содержимое HTML и отображать его так, как задумывалось. Парсер проходит по документу и разбивает его на составные части, понимает, что каждая из частей должна сделать, отображает это определенным образом.

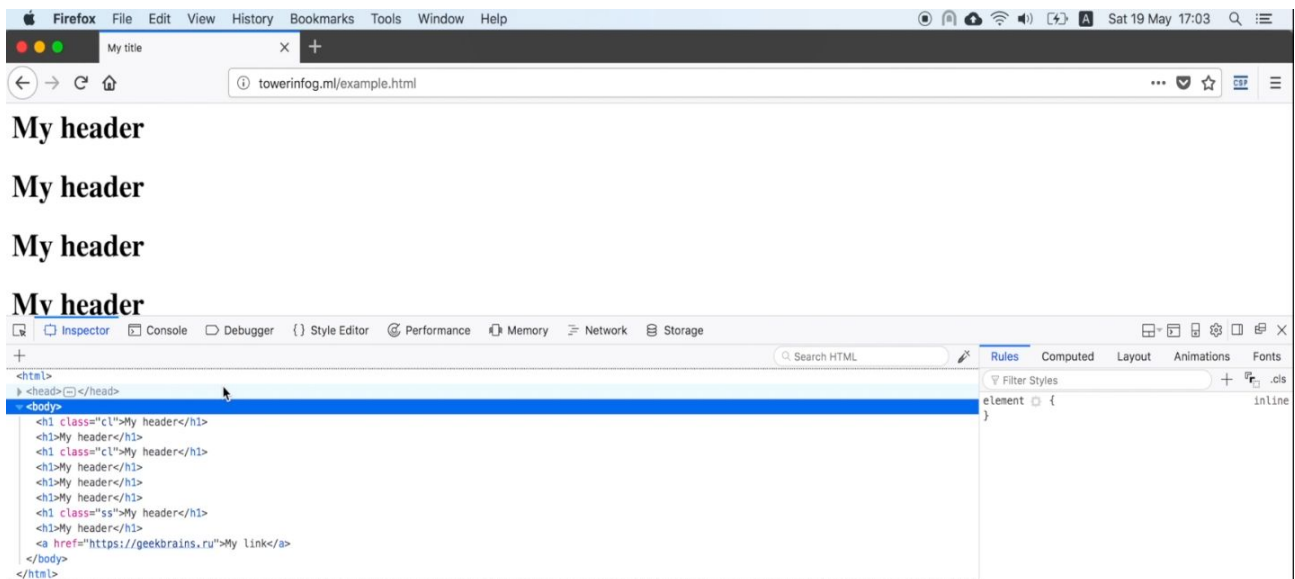
Посмотрим на примере, как происходит парсинг HTML в нестандартных ситуациях. Со стандартными мы уже разобрались и теперь нужно посмотреть, что будет, например, если HTML с ошибками.

Отредактируем файл `example.html` из предыдущего урока. Теперь для примера попробуем полностью удалить корневой тег `<html>`, затем сохраним и откроем получившийся файл в браузере:

```
<head>
  <title>My title</title>
  <!--link rel="stylesheet" href="style.css"-->
</head>
<body>
  <h1 class="cl">My header</h1>
  <h1>My header</h1>
  <h1 class="cl">My header</h1>
  <h1>My header</h1>
  <h1>My header</h1>
  <h1>My header</h1>
  <h1 class="ss">My header</h1>
  <h1>My header</h1>
  <a href="https://geekbrains.ru">My link</a>
</body>
```

По сути ничего не поменялось, все отображается, как и прежде, — правильно.

Теперь откроем в браузере исходный код страницы и посмотрим, как он выглядит:

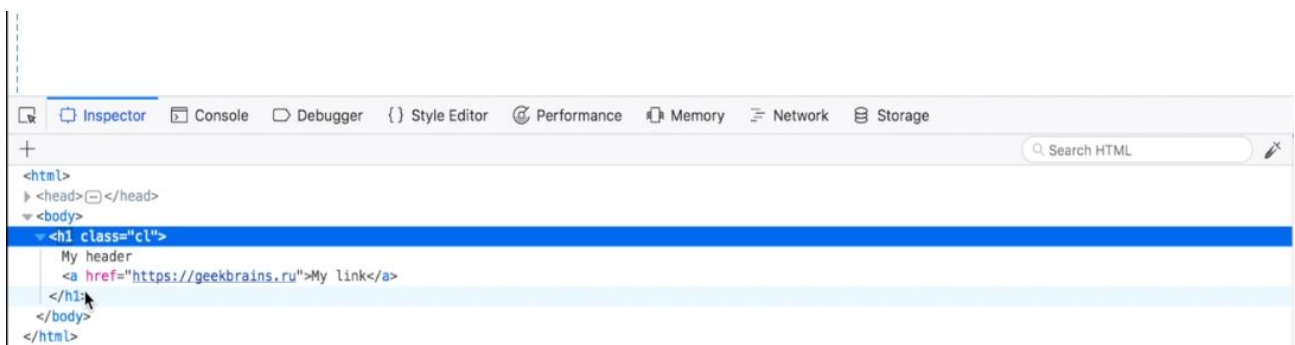


Таким образом, даже несмотря на то, что мы целиком удалили теги `<html>` и `</html>`, здесь они все равно присутствуют. Это открывает нам одну из особенностей HTML-парсеров — они умеют достраивать HTML-код документа или, по-другому, DOM-дерево, самостоятельно. DOM — это акроним от Document Object Model и вся древовидная структура HTML-документа называется DOM-деревом.

Теперь попробуем удалить закрывающий тег `</h1>` и затем перезагрузим страницу:



Тег `</h1>` успешно закрылся, но не там, где он был до этого, а ниже, сразу после ссылки:

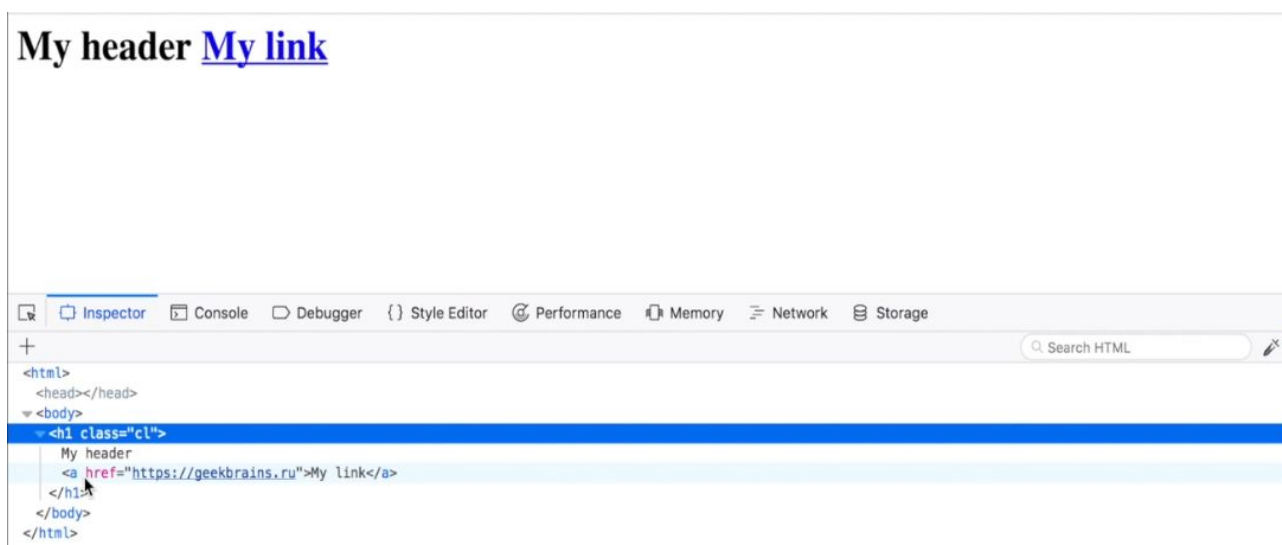


HTML-парсер в браузере сам додумал, куда ему поставить тег `</h1>`, затем закрыл его и отобразил страницу. Политика парсинга HTML такова, что парсер старается максимально достроить DOM-дерево до валидного состояния, когда его можно будет отобразить, и сделать это.

Более того, даже если мы удалим и все остальные теги, даже, например, `<body>`:

```
<h1 class="cl">My header
<a href="https://geekbrains.ru">My link</a>
```

HTML все равно успешно отобразится, хоть и снова не так, как нам было нужно:



Парсер самостоятельно добавил недостающие теги: `<html></html>`, `<head></head>`, `<body></body>` и положил внутрь тега `<body>` HTML, хоть и не совсем корректно.

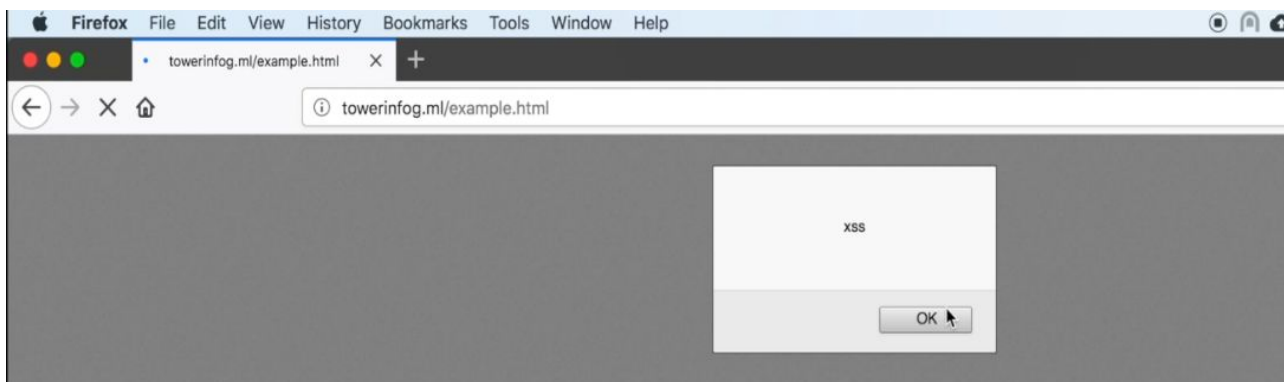
Парсинг и проблемы безопасности

Также, например, если мы закроем тег `<h1>` при помощи `</h1>`, парсер HTML тоже воспримет это как валидное закрытие. Такая лояльность парсера приводит к тому, что когда разработчики пишут фильтры безопасности, чтобы не допустить вредоносный код или нежелательный и опасный ввод, они не могут учесть все ситуации, в которых HTML сможет достроить DOM-дерево до валидного.

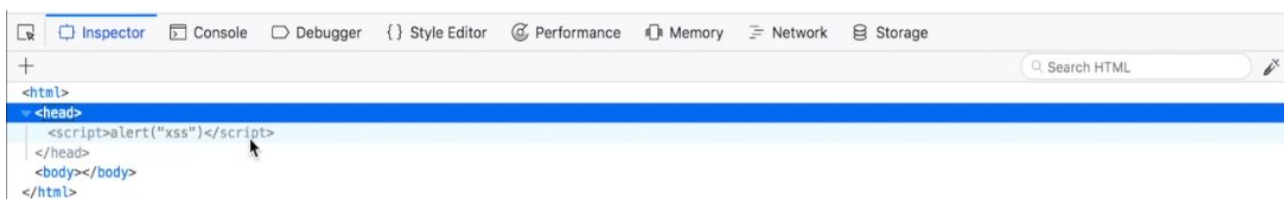
Например, есть тег `<script>` и, допустим, что фильтр запрещает закрывающий тег `</script>`. Но если злоумышленник допишет один слеш — `</script/>`, то фильтр может уже не сработать, потому что он настроен запрещать код `</script>`, а не `</script/>`:

```
<script>alert("xss")</script/>
```

Давайте проверим, что будет при попытке выполнить этот HTML-код в браузере:

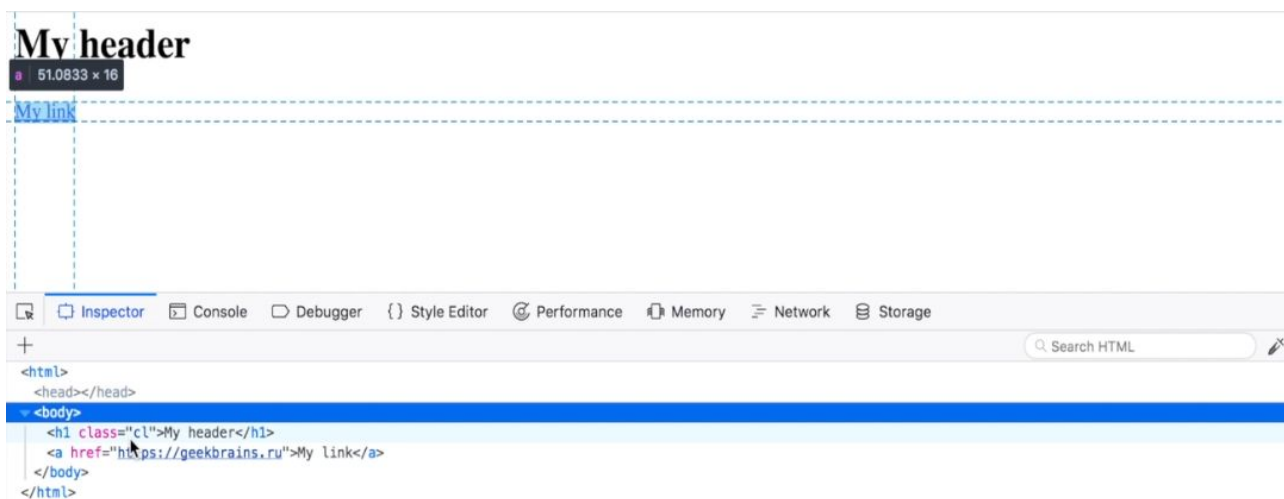


Несмотря на то, что HTML-тег `</script>` не выглядит как валидный, парсер в браузере Firefox его спокойно разобрал и додумал за нас, что необходимо это исправить на `</script>`:



HTML-парсер до последнего пытается отобразить требуемую HTML-страницу, как бы разработчик ее ни написал и какие бы теги он ни пропустил.

Также возможно опускать кавычки, например в ссылках, или заменить их на одинарные — и HTML-парсер это нормально отобразит:

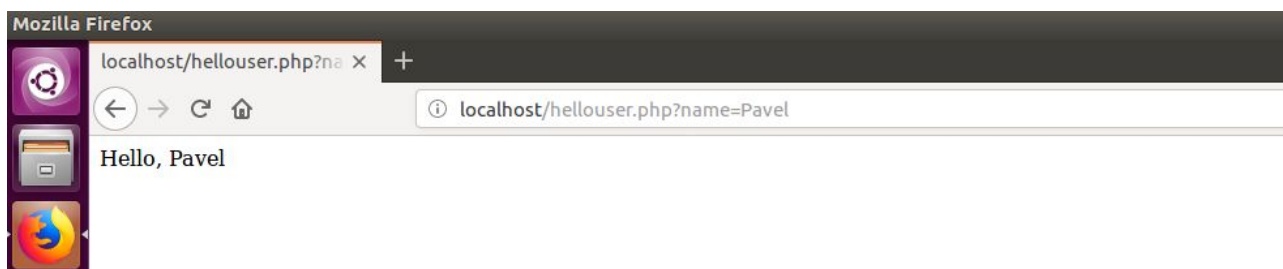


Он заменил одинарные кавычки в классе на двойные и добавил недостающие.

Инъекции в HTML

Рассмотрим такую проблему парсинга HTML, как интерпретацию пользовательских данных браузером как часть HTML-документа.

Откроем виртуальную машину с Ubuntu, откроем браузер и вспомним, как работает скрипт `hellouser.php`. Для этого откроем ссылку <http://localhost/hellouser.php?name=Pavel>:



Как мы помним, скрипт берет GET-параметр `name`, достает из него значение и отображает его на экране в составе тестовой строки приветствия.

Теперь посмотрим, как поведет себя этот скрипт, если мы добавим в него HTML-теги. Например, если мы добавим в адресную строку тег `<h2>` вот так — <http://localhost/hellouser.php?name=<h2>Pavel>:



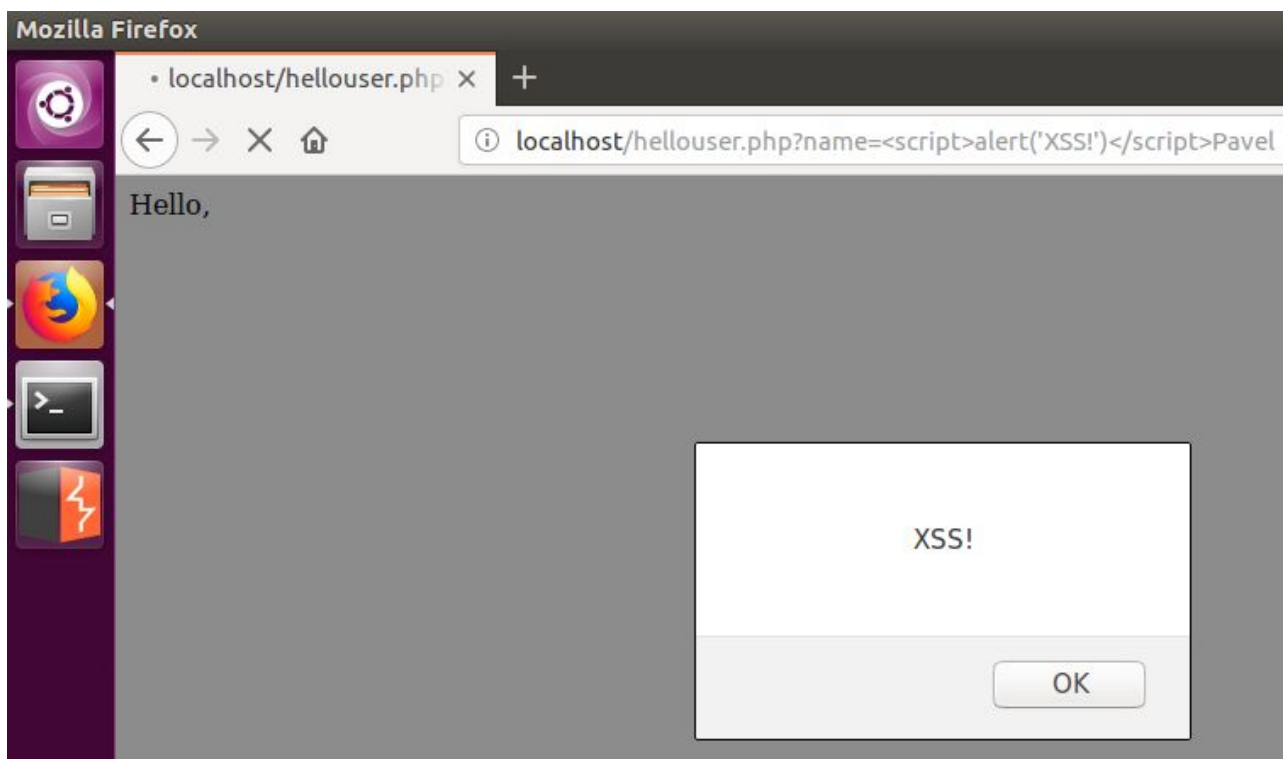
Видим, что браузер воспринял тег `<h2>` именно как HTML-тег, а не как текст.

Это основополагающая проблема возникновения целого класса атак, называемых инъекциями. HTML-инъекция появляется, когда мы вставляем произвольный HTML-код.

XSS-инъекции

Возможен и другой, более серьезный тип атак на браузер жертвы — так называемая XSS-инъекция. С ее помощью мы можем вставить вредоносный скрипт, который будет выполнен в браузере жертвы, открывшей такую ссылку.

Например, напишем такой ввод — [http://localhost/hellouser.php?name=<script>alert\('XSS!'\)</script>Pavel](http://localhost/hellouser.php?name=<script>alert('XSS!')</script>Pavel) :



И видим, что после этого скрипт отработывает, а это небезопасно, потому что при таком сценарии злоумышленник может, например, отослать на почту в письме или как-то передать пользователю вредоносную ссылку, а пользователь по ней перейдет. После этого злоумышленник сможет, например, украсть из браузера сохраненные там Cookie, сделать скриншот экрана и нарисовать любую форму на странице, или еще хуже — считать то, что печатает пользователь в браузере. Так он получит данные пользователя, доступ к его аккаунту и информацию, которая находится в браузере или личном кабинете.

Эскейпинг HTML

Отсюда возникает вопрос: а что же делать с тем, что браузер и парсер HTML воспринимают пользовательский ввод как часть HTML-синтаксиса? Для этого придумали HTML-эскейпинг (от англ. *escape* — спастись, убежать), то есть экранировать ввод пользователя, не допуская выполнения спецсимволов и тегов.

Откроем терминал и запустим скрипт `hellouser.php`:

```
nano hellouser.php

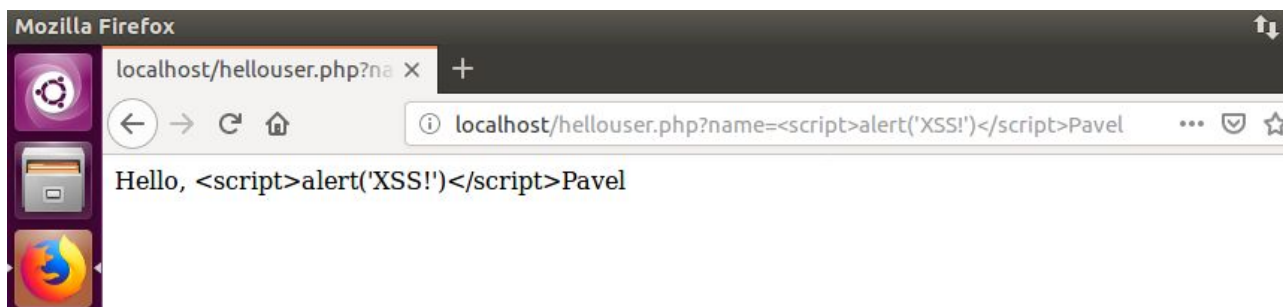
<?php
    echo "Hello, " . $_GET["name"];
?>
```

В PHP есть специальная функция `htmlspecialchars()`, добавим ее:

```
<?php
    echo "Hello, " . htmlspecialchars($_GET["name"]);
?>
```

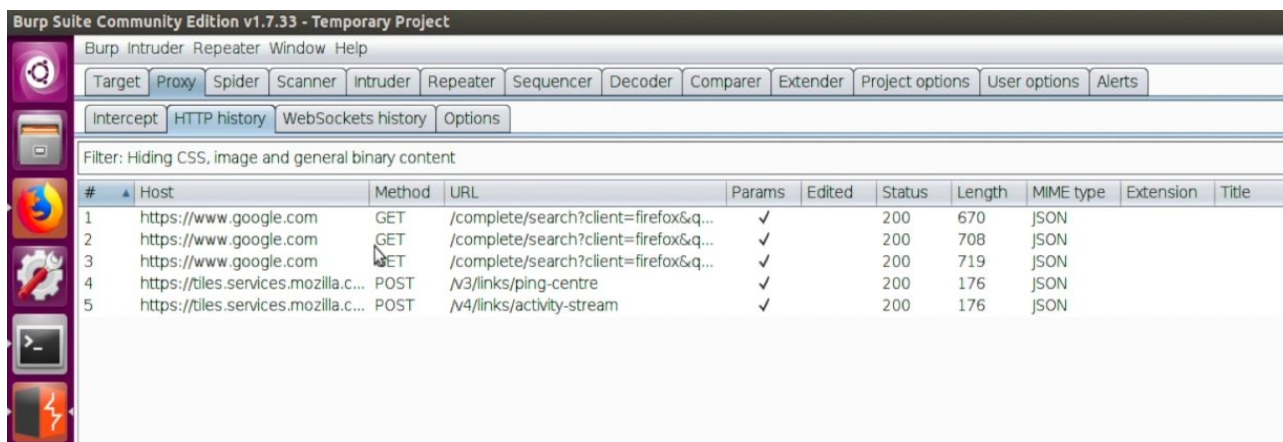
Подробнее о том, что такое функции, зачем они нужны и как работают, вы узнаете, когда мы будем проходить JavaScript.

Сейчас мы добавим эту функцию и посмотрим, какой эффект это принесет. Запишем изменения и вернемся в Firefox, чтобы перезагрузить страницу:



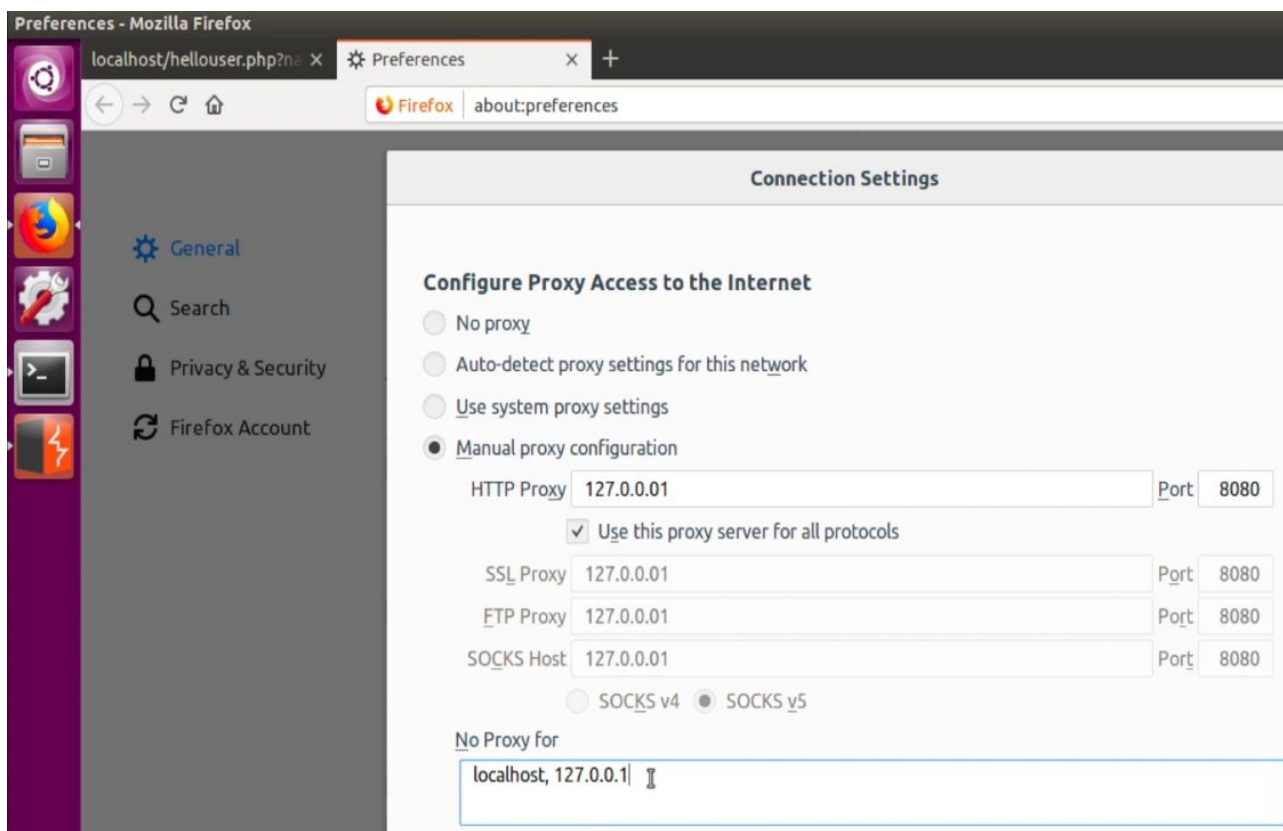
JavaScript код не исполнился, более того, все написанное отобразилось в странице как обычный текст.

Чтобы понять, что произошло, откроем Burp и перейдем в Http history:

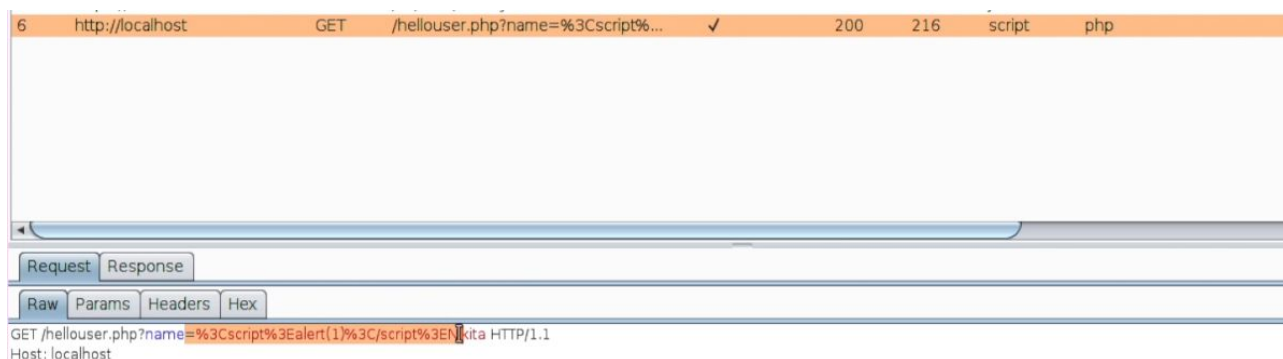


Как видим, запросов к <http://localhost> нет — это произошло из-за того, что Firefox по умолчанию не проксирует эти запросы. Изменим это.

Откроем меню настройки Firefox, и в поле No Proxy for удалим запись localhost и 127.0.0.1:



Применяем и перезагружаем страницу. Затем открываем Вирг — запрос прошел через прокси:



Вот он, причем в URL encode, т. к. специальные символы не должны появляться в URL.

HTML-сущности

Посмотрим ответ — там то, что мы написали, дополнено непонятными символами и последовательностью знаков:



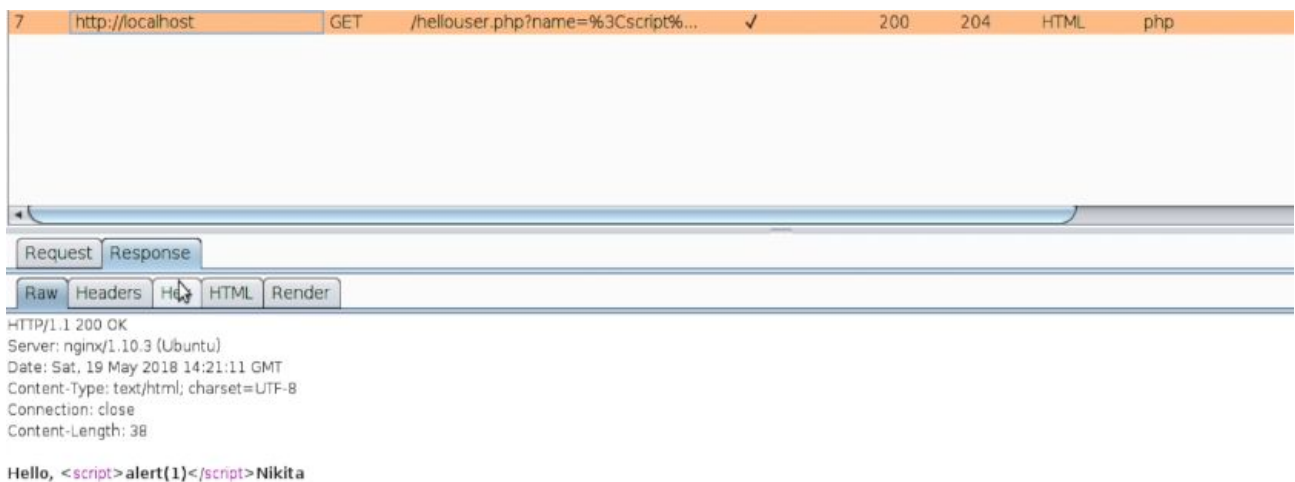
Слова, которые начинаются с амперсанда и заканчиваются точкой с запятой (здесь это `<` и `>`), называются HTML-сущностями. Это такое представление специальных символов, которое позволяет заменить их в HTML-коде. Например, спецсимвол `<` может быть заменен на `<` и после этого он будет восприниматься не как открывающая скобка тега, а именно как символ, и отображаться так, как требуется.

Такое сокращение (`lt` и `gt`) выбрано потому, что `<` по-английски называется *less than*, а `>` — *greater than* и, соответственно, это акронимы этих слов.

Посмотрим, как будет выглядеть ответ без HTML-эскейпинга. Отменим изменения в файле и сохраним его:

```
<?php
    echo "Hello, " . $_GET["name"];
?>
```

Затем перезагрузим страницу в браузере и посмотрим в Burp:




Если не применять HTML-эскейпинг, пользовательский ввод превратится в действительные HTML-теги. Даже несмотря на то, что здесь нет тегов `<html>`, `</html>`, `<body>` и ничего подобного, парсер успешно достраивает HTML до валидного и после этого выполняет скрипт.

Для безопасности не обязательно эскейпить все HTML-символы пользовательского ввода, достаточно указать только основные 5 символов:

HTML эскейпинг

&		&
"		"
'		'
<		<
>		>



На слайде выше вы видите, чему соответствуют эти символы.

Нужно запомнить и постоянно применять, будь вы разработчик либо безопасник: когда вы показываете пользовательский ввод на странице, необходимо выполнить HTML-эскейпинг. Иначе возможны различные уязвимости: HTML- или XSS-инъекции в браузере жертвы и другие.

В большинстве языков программирования уже реализованы стандартные функции для HTML-эскейпинга, например в PHP есть `htmlspecialchars()`. Если в вашем случае такой функции нет, вы всегда сможете вернуться к списку и написать свою на его основе.

Итоги

Подведем итоги этого урока:

- 1) Узнали, что такое парсинг.
- 2) Узнали, как браузер парсит HTML и какие проблемы безопасности это порождает.
- 3) Узнали, что браузер умеет достраивать HTML самостоятельно.
- 4) Научились использовать HTML-эскейпинг и поняли, зачем он нужен.

Видеоурок 4

В этом видео мы узнаем:

- 1) Гиперссылки и варианты их применения.
- 2) Атрибут `target` и для чего он может применяться.
- 3) HTML-формы и зачем они нужны.
- 4) `Iframe`.
- 5) Атрибут `src` и как с его помощью подгружать ресурсы.

Гиперссылки и загрузка контента

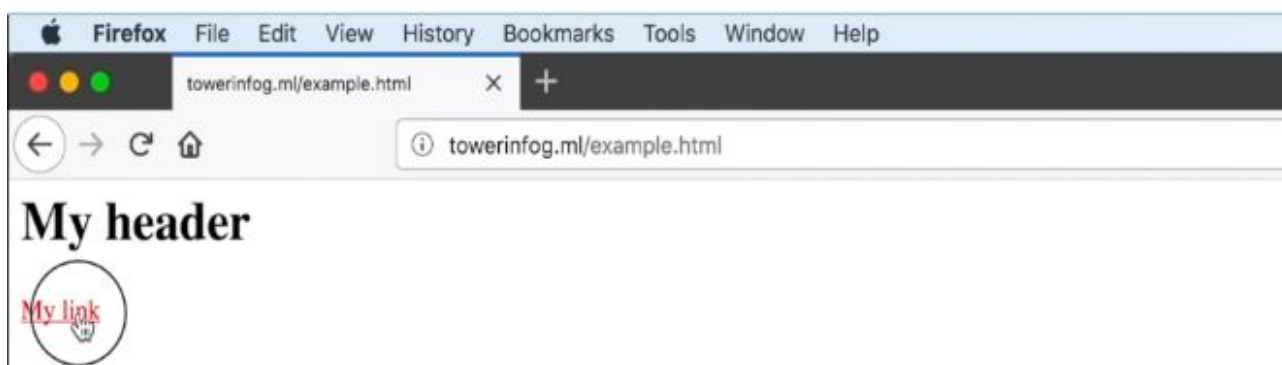
Гиперссылки нужны, чтобы пользователь мог переходить с одной страницы на другую. Мы уже видели примеры гиперссылок в теге `<a>`.

Запустим терминал в виртуальной машине Ubuntu и посмотрим на файл `example.html`:

```
cat example.html

<h1 class='cl'>My header</h1>
<a href=https://geekbrains.ru>My link</a>
```

Здесь уже есть гиперссылка на <https://geekbrains.ru>, откроем ее в Firefox, нажимаем My link:



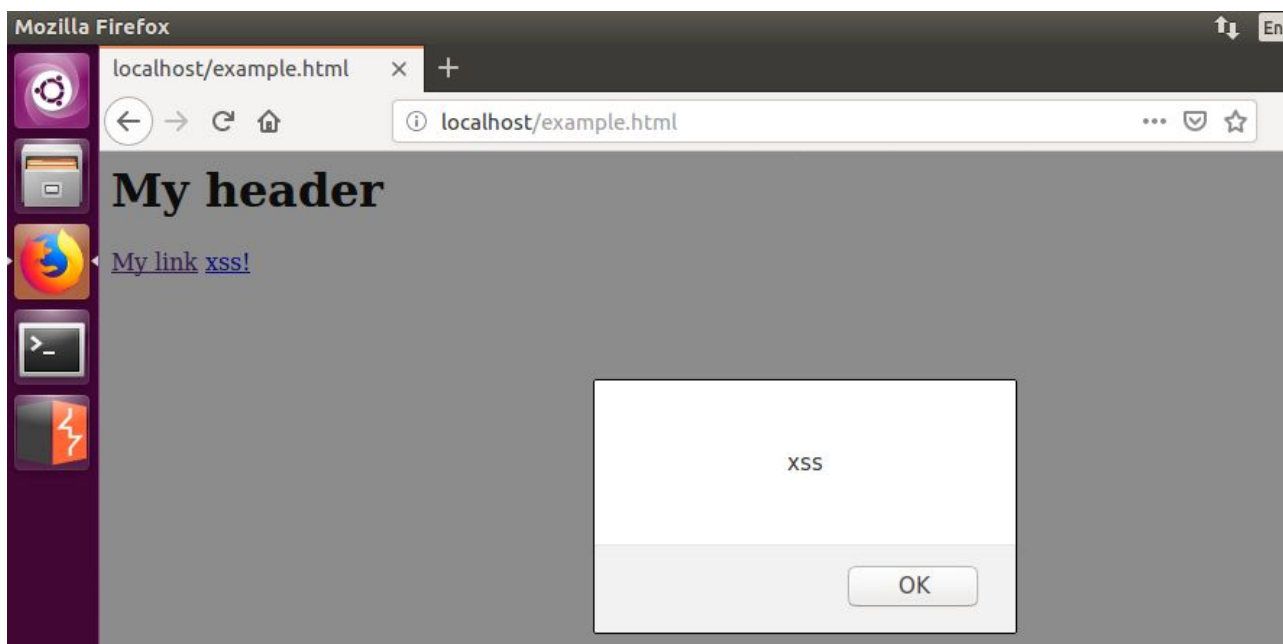
После нажатия мы перешли на <https://geekbrains.ru>.

Но на этом возможности гиперссылок не заканчиваются. Если мы добавим такую гиперссылку на страницу, то она может привести к кросс-сайт-скриптингу:

```
nano example.html

<h1 class='cl'>My header</h1>
<a href=https://geekbrains.ru>My link</a>
<a href="javascript:alert('xss')">xss!</a>
```

Перейдем по этой гиперссылке. Выполняется JavaScript-код:



Это происходит из-за того, что здесь схема не `https://` и не `http://`, а `javascript:`. Также тогда можно поставить гиперссылку на `data:` — и в ней тоже можно исполнить код, открыть картинку или любой другой тип данных.

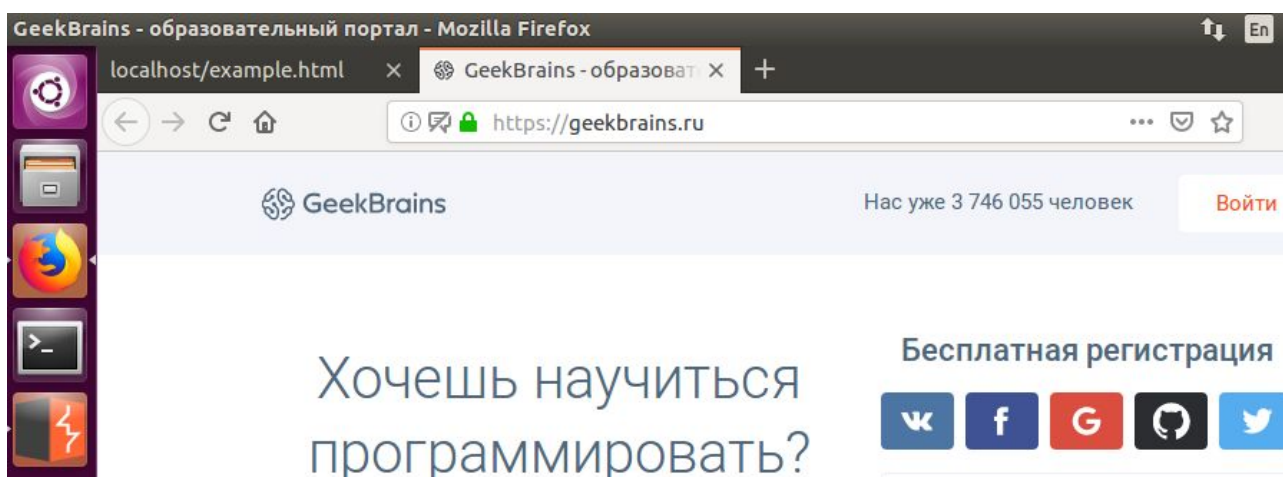
Атрибут target

Добавим атрибут `target` со значением `_blank` в тег `<a>`:

```
nano example.html

<h1 class='cl'>My header</h1>
<a href=https://geekbrains.ru target="_blank">My link</a>
<a href="javascript:alert('xss')">xss!</a>
```

Перезагрузим страницу и нажмем на ссылку `My link`:



Гиперссылка открылась не в том же окне, а в новом. Это произошло из-за атрибута `blank` — он означает открыть пустую страницу и показать в ней содержимое гиперссылки. Такое поведение можно использовать, когда мы пишем вектор атаки, чтобы пользователь не закрывал текущую страницу. Для этого есть и другие методы, но и этот часто применяется.

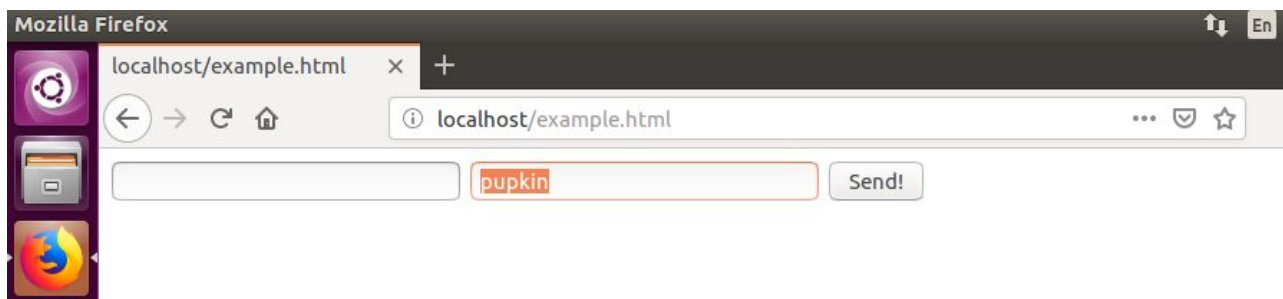
HTML-формы

Как клиенту удобно отправить данные на сервер? Мы уже знаем про GET-запросы, про заголовки HTTP, про POST-запрос. Обычный пользователь, который не хочет разбираться, как устроен веб, не будет усложнять себе жизнь. Именно для этого и придумали HTML-формы. Вы уже наверняка пользовались HTML-формами, например, на сайте <https://geekbrains.ru>. Посмотрим на пример HTML-формы:

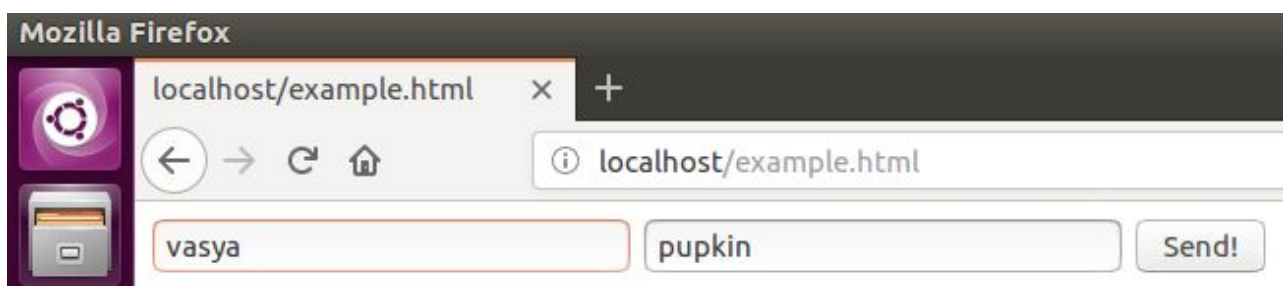
```
nano example.html

<form method="POST" action="/action.html">
  <input name="name" value="">
  <input name="last name" value="pupkin">
  <button type="submit">Send!</button>
</form>
```

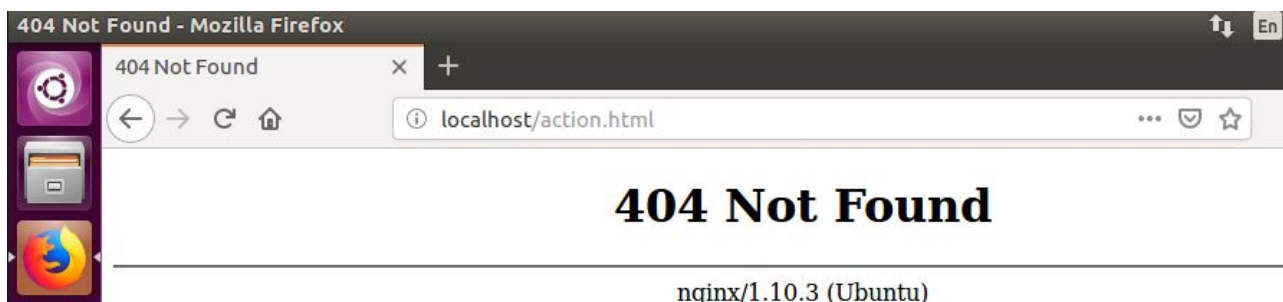
В браузере этот html отобразится так:



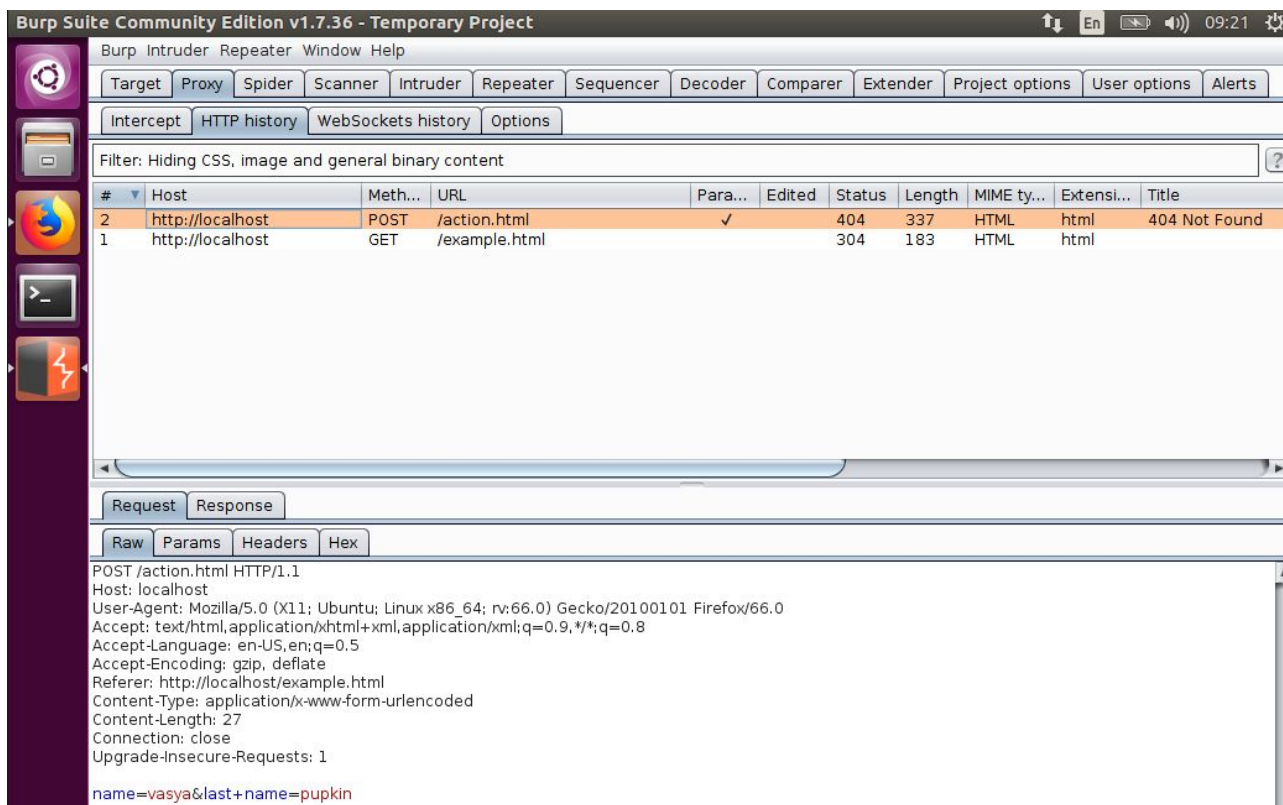
Такое уже проще воспринимать: мы видим два поля, причем одно из них предзаполнено за счет того, что мы добавили атрибут `value="pupkin"`. В них можно ввести данные, нажать **Send!** — и данные отправятся на сервер:



Но так как такого URL `/action.html` на сайте нет, мы получим ошибку 404 Not Found:



Посмотрим, какой запрос при этом отправился — откроем Burp:



Отправился POST-запрос по URL /action.html, почему так произошло, мы посмотрим, когда будем разбирать исходный код.

Откроем содержимое POST-запроса и увидим, что то, что мы добавили в имя (name=vasya) отправилось на сервер, и last name=pupkin тоже ушло на сервер как POST-параметры.

В этом и смысл форм. Форма берёт все `<input>` (тег, который превращается в поле ввода) получает их значения и отправляет их по URL, указанному в атрибуте `action` (здесь это /action.html). Наш браузер пошел по этой относительной ссылке, но здесь могла быть указана абсолютная и любая другая из типов ссылок, которые мы проходили. Здесь же можно указать метод запроса, например POST. Чаще всего это применяется именно для POST-запроса: формы не делают GET-запрос, потому что он нужен, чтобы забирать данные, а POST — чтобы отправлять их на сервер.

Также мы видим тег `<button>`, который создает кнопку. Обязательно нужно указать `type="submit"`, чтобы эта кнопка отправляла данные на сервер, а не просто нажималась или делала что-то другое.

Тег iframe и атака Click-Jacking

Разберемся с темой iframe.

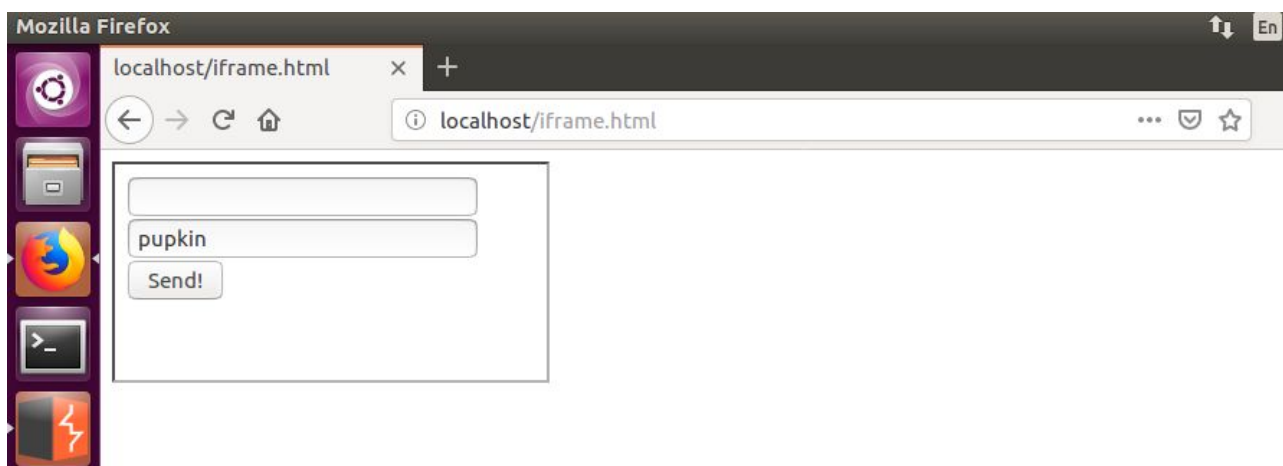
Iframe отображает другой сайт в текущем окне.

```
nano iframe.html

<iframe src="//localhost/example.html"></iframe>
```

Тег несложный в базовой конфигурации: он открывается с помощью `<iframe>`, пишется `src="источник, откуда будет загружен iframe, та страница, которая будет в него загружена"` и закрывается тегом `</iframe>`. Между открывающим и закрывающим тегами можно написать текст, который отобразится, если iframe не будет загружен.

Попробуем открыть его в браузере:

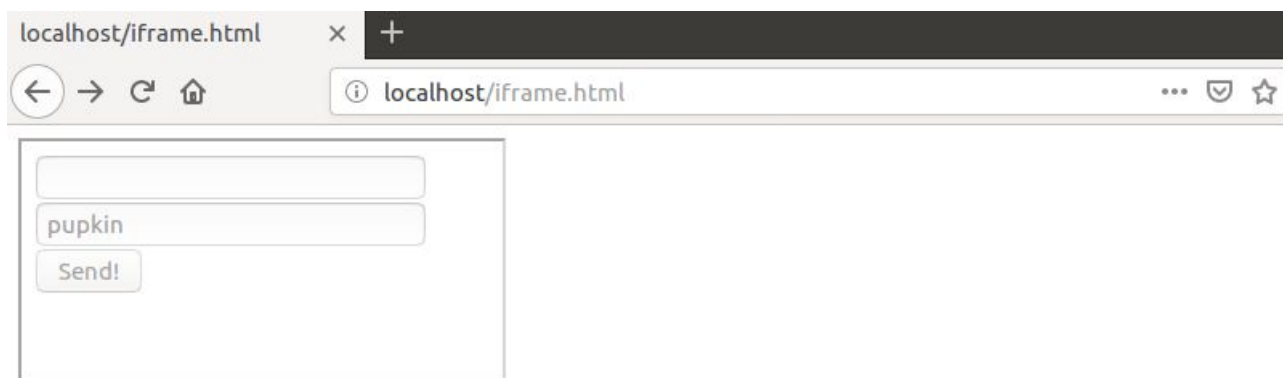


Теперь страница открылась в специальном окне. Это окно можно менять, перетаскивать и изменять его стиль. Посмотрим, как будет выглядеть полупрозрачный iframe, на примере:

```
nano iframe.html

<iframe src="//localhost/example.html" style="opacity:0.5"></iframe>
```

Как мы видим ниже, iframe стал прозрачнее:



Его можно скрыть, сделав полностью прозрачным, при этом можно будет нажимать на кнопки и взаимодействовать с его элементами. Эта особенность используется в такой атаке, как клик-джекинг (Click-Jacking), ее мы разберем на более поздних видео.

В `iframe` есть атрибут `sandbox`, который позволяет ограничить и оградить `iframe` от остальной страницы. У атрибута `sandbox` во фрейме очень много настроек: можно включить или выключить исполнение скриптов, одинаковый `origin` в `iframe` относительно родительской страницы и так далее.

Атрибут `src`

Чтобы подгружать контент с других страниц, нужно использовать атрибут `src`. Например, мы можем подгружать скрипт из других страниц так же, как мы подгружали `iframe` выше.

Итоги

Подведем итоги:

- 1) Вы узнали, что такое гиперссылки, как по-разному их можно применять.
- 2) Узнали, для чего нужен атрибут `target`.
- 3) Узнали, зачем нужны HTML-формы, как они работают.
- 4) Разобрались что такое `iframe` и как включать контент с других ресурсов.

Итоги урока

На этом мы завершили изучение HTML:

- 1) Теперь мы умеем писать на HTML и CSS.
- 2) Понимаем, как HTML-парсер разбирает HTML-страницу и превращает ее в то, что мы видим в браузере, какие проблемы безопасности с этим связаны.
- 3) Также мы узнали, как работают `iframe` и Content Inclusion.

В следующем уроке мы научимся писать программы на JavaScript, узнаем, что такое DOM и как с ним работать, освоим XHR-запросы и JSON-сериализацию.

Ссылки к уроку

1. [Справочник HTML и CSS.](#)
2. [Объектная модель HTML-документа DOM.](#)
3. [Браузеры и парсинг в HTML.](#)
4. [Эскейпинг в HTML.](#)
5. [Что такое XSS- и DOM-инъекции.](#)
6. [Атака Clickjacking и защита от неё.](#)