**Тема 9. Окна.**

1. **Открытие окон и методы window**
2. **Общение между окнами**
3. **Атака типа clickjacking**

Содержание данной темы включает материалы, доступные по адресу https://learn.javascript.ru.

1. Открытие окон и методы window

Всплывающее окно («попап» – от англ. Popup window) – один из древнейших способов показать пользователю ещё один документ. Достаточно запустить:

window.open('https://javascript.info/')

и откроется новое окно с указанным URL. Большинство современных браузеров по умолчанию будут открывать новую вкладку вместо отдельного окна.

Попапы существуют с давно. Они были придуманы для отображения нового контента поверх главного окна, оставив главное окно открытым. Но с тех пор появились другие способы сделать это: JavaScript может загрузить содержимое вызовом [fetch](https://learn.javascript.ru/fetch) и показать его в тут же созданном <div>, так что попапы используются не каждый день. Кроме того, попапы не очень хороши для мобильных устройств, которые не умеют показывать несколько окон одновременно.

Однако, для некоторых задач попапы ещё используются, например, для OAuth-авторизации (вход через Google/Facebook/…), так как:

1. Попап – это отдельное окно со своим JavaScript-окружением. Так что открытие попапа со стороннего, не доверенного сайта вполне безопасно.
2. Открыть попап очень просто.
3. Попап может производить навигацию (менять URL) и отсылать сообщения в основное окно.

В прошлом злонамеренные сайты заваливали посетителей всплывающими окнами. Такие страницы могли открывать сотни попапов с рекламой. Поэтому теперь большинство браузеров пытаются заблокировать всплывающие окна, чтобы защитить пользователя. Всплывающее окно блокируется в том случае, если вызов window.open произошёл не в результате действия посетителя (например, события onclick). Например:

// попап заблокирован

window.open('https://javascript.info');

// попап будет показан

button.onclick = () => {

window.open('https://javascript.info');

};

Таким образом браузеры могут защитить пользователя от появления нежелательных попапов, при этом не отключая попапы полностью. Допустим, попап должен открываться в результате onclick, но не сразу, а только после выполнения setTimeout:

// откроется через 3 секунды

setTimeout(() => window.open('http://google.com'), 3000);

Попап откроется в Chrome, но будет заблокирован в Firefox. Но если уменьшить тайм-аут до одной секунды, то попап откроется и в Firefox:

// откроется через 1 секунду

setTimeout(() => window.open('http://google.com'), 1000);

В итоге имеем два разных результата из-за того, что Firefox «допускает» таймаут в 2000 мс или менее, но все, что свыше этого – не вызывает его доверия, т.к. предполагается, что в таком случае открытие окна происходит без ведома пользователя. Именно поэтому попап из первого примера будет заблокирован, а из второго – нет.

Полный синтаксис открытия нового окна: window.open(url, name, params):

* url – URL для загрузки в новом окне,
* name – имя нового окна. У каждого окна есть свойство window.name, в котором можно задавать, какое окно использовать для попапа. Таким образом, если уже существует окно с заданным именем – указанный в параметрах URL откроется в нем, в противном случае откроется новое окно.
* params – строка параметров для нового окна. Содержит настройки, разделённые запятыми. Важно помнить, что в данной строке не должно быть пробелов. Например width:200,height=100.

Параметры в строке params:

* Позиция окна:
* left/top (числа) – координаты верхнего левого угла нового окна на экране. Существует ограничение: новое окно не может быть позиционировано вне видимой области экрана.
* width/height (числа) – ширина и высота нового окна. Существуют ограничение на минимальные высоту и ширину, которые делают невозможным создание невидимого окна.
* Панели окна:
* menubar (yes/no) – позволяет отобразить или скрыть меню браузера в новом окне.
* toolbar (yes/no) – позволяет отобразить или скрыть панель навигации браузера (кнопки вперёд, назад, перезагрузки страницы) нового окна.
* location (yes/no) – позволяет отобразить или скрыть адресную строку нового окна. Firefox и IE не позволяют скрывать эту панель по умолчанию.
* status (yes/no) – позволяет отобразить или скрыть строку состояния. Как и с адресной строкой, большинство браузеров будут принудительно показывать её.
* resizable (yes/no) – позволяет отключить возможность изменения размера нового окна. Не рекомендуется.
* scrollbars (yes/no) – позволяет отключить полосы прокрутки для нового окна. Не рекомендуется.

Помимо этого, существует некоторое количество не кроссбраузерных значений, которые обычно не используются.

Откроем окно с минимальным набором настроек, просто чтобы посмотреть, какие из них браузер позволит отключить:

let params = `scrollbars=no,resizable=no,status=no,location=no,toolbar=no,menubar=no,

width=0,height=0,left=-1000,top=-1000`;

open('/', 'test', params);

В этом примере большинство настроек заблокированы и само окно находится за пределами видимой области экрана. Посмотрим, что получится в результате. Большинство браузеров «исправит» странные значения – как, например, нулевые width/height и отрицательные left/top. Например, Chrome установит высоту и ширину такого окна равными высоте и ширине экрана, так что попап будет занимать весь экран.

Исправим значения и зададим нормальные координаты (left и top) и значения размеров окна (width и height):

let params = `scrollbars=no,resizable=no,status=no,location=no,toolbar=no,menubar=no,

width=600,height=300,left=100,top=100`;

open('/', 'test', params);

Большинство браузеров выведет окно с заданными настройками.

Правила для опущенных параметров:

* Если третий аргумент при вызове open отсутствует или он пустой, то будут использованы настройки окна по умолчанию.
* Если строка параметров передана, но некоторые параметры yes/no пропущены, то считается, что указано no, так что соответствующие возможности будут отключены, если на это нет ограничений со стороны браузера. Поэтому при задании параметров убедитесь, что явно указаны все необходимые yes.
* Если координаты left/top не заданы, браузер попытается открыть новое окно рядом с предыдущим открытым окном.
* Если не заданы размеры окна width/height, браузер откроет новое окно с теми же размерами, что и предыдущее открытое окно.

Вызов open возвращает ссылку на новое окно. Эта ссылка может быть использована для управления свойствами окна, например, изменения положения и др.

Например, здесь генерируется содержимое попапа из JavaScript:

let newWin = window.open("about:blank", "hello", "width=200,height=200");

newWin.document.write("Hello, world!");

А здесь содержимое окна модифицируется после загрузки:

let newWindow = open('/', 'example', 'width=300,height=300')

newWindow.focus();

alert(newWin.location.href); // (\*) about:blank, загрузка ещё не началась

newWindow.onload = function() {

let html = `<div style="font-size:30px">Добро пожаловать!</div>`;

newWindow.document.body.insertAdjacentHTML('afterbegin', html);

};

Обратите внимание: сразу после window.open новое окно ещё не загружено. Это демонстрируется в строке (\*). Так что нужно ждать onload, чтобы его изменить. Или же поставить обработчик DOMContentLoaded на newWin.document.

Окна имеют свободный доступ к содержимому друг другу только если они с одного источника (у них совпадают домен, протокол и порт (protocol://domain:port). Иначе, например, если основное окно с site.com, а попап с gmail.com, это невозможно по соображениям пользовательской безопасности.

Попап также может обратиться к открывшему его окну по ссылке window.opener. Она равна null для всех окон, кроме попапов. Код ниже заменит содержимое открывшего (текущего) окна на «Тест»:

let newWin = window.open("about:blank", "hello", "width=200,height=200");

newWin.document.write(

"<script>window.opener.document.body.innerHTML = 'Тест'<\/script>"

);

Так что связь между окнами двусторонняя: главное окно и попап имеют ссылки друг на друга.

Чтобы закрыть окно: win.close(). Для проверки, закрыто ли окно: win.closed.

Технически метод close() доступен для любого окна, но window.close() будет игнорироваться большинством браузеров, если window не было создано с помощью window.open(). Так что он сработает только для попапов.

Если окно закрыто, то его свойство closed имеет значение true. Таким образом можно легко проверить, закрыт ли попап (или главное окно) или все ещё открыт. Пользователь может закрыть его в любой момент, и код должен учитывать эту возможность. Этот код откроет и затем закроет окно:

let newWindow = open('/', 'example', 'width=300,height=300');

newWindow.onload = function() {

newWindow.close();

alert(newWindow.closed); // true

};

Методы для передвижения и изменения размеров окна:

* win.moveBy(x,y) – переместить окно относительно текущей позиции на *x* пикселей вправо и *y* пикселей вниз. Допустимы отрицательные значения (для перемещения окна влево и вверх).
* win.moveTo(x,y) – переместить окно на координаты экрана (x,y).
* win.resizeBy(width,height) – изменить размер окна на указанные значения width/height относительно текущего размера. Допустимы отрицательные значения.
* win.resizeTo(width,height) – изменить размер окна до указанных значений.

Также существует событие window.onresize.

Чтобы предотвратить возможные злоупотребления, браузер обычно блокирует эти методы. Они гарантированно работают только с попапами, которые открыл код и у которых нет дополнительных вкладок.

Методами JavaScript нельзя свернуть или развернуть («максимизировать») окно на весь экран. За это отвечают функции уровня операционной системы, и они скрыты от фронтенд-разработчиков.

Методы перемещения и изменения размера окна не работают для свернутых и развёрнутых на весь экран окон.

Методы [прокрутки](https://learn.javascript.ru/popup-windows#prokrutka-okna) окна:

* win.scrollBy(x,y) – прокрутить окно на *x* пикселей вправо и *y*пикселей вниз относительно текущей прокрутки. Допустимы отрицательные значения;
* win.scrollTo(x,y) – прокрутить окно до заданных координат (x,y);
* elem.scrollIntoView(top = true) – прокрутить окно так, чтобы elem для elem.scrollIntoView(false) появился вверху (по умолчанию) или внизу.

Также существует событие window.onscroll.

Установить попап в фокус можно с помощью метода window.focus(), а убрать из фокуса – с помощью window.blur(). Также существуют события focus/blur, которые позволяют отследить, когда фокус переводится на какое-то другое окно.

Раньше на «плохих» сайтах эти методы могли становиться средством манипуляции. Например:

window.onblur = () => window.focus();

Когда пользователь пытается перевести фокус на другое окно, этот код возвращает фокус назад. Таким образом, фокус как бы «блокируется» в попапе, который не нужен пользователю. Из-за этого в браузерах и появились ограничения, которые препятствуют такого рода поведению фокуса. Эти ограничения нужны для защиты пользователя от назойливой рекламы и «плохих» страниц, и их работа различается в зависимости от конкретного браузера.

Например, мобильный браузер обычно полностью игнорирует такие вызовы метода window.focus(). Также фокусировка не работает, когда попап открыт в отдельной вкладке (в отличие от открытия в отдельном окне).

Но все-таки иногда методы фокусировки бывают полезны. Например:

* Когда код открывает попап, можно запустить для него newWindow.focus(). Для некоторых комбинаций браузера и операционной системы это устранит неоднозначность – заметит ли пользователь – это новое окно.
* Если нужно отследить, когда посетитель использует веб-приложение, можно отслеживать window.onfocus/onblur. Это позволит ставить на паузу и продолжать выполнение анимаций и других интерактивных действий на странице. При этом важно помнить, что blur означает, что окно больше не в фокусе, но пользователь может по-прежнему видеть его.

1. Общение между окнами

Политика «Одинакового источника» (Same Origin) ограничивает доступ окон и фреймов друг к другу. Идея заключается в том, что если у пользователя открыто две страницы: john-smith.com и gmail.com, то у скрипта со страницы john-smith.com не будет возможности прочитать письма из gmail.com. Таким образом, задача политики «Одинакового источника» – защитить данные пользователя от возможной кражи.

Два URL имеют «одинаковый источник» в том случае, если они имеют совпадающие протокол, домен и порт.

Эти URL имеют одинаковый источник:

* http://site.com
* http://site.com/
* http://site.com/my/page.html

А эти – разные источники:

* http://www.site.com (другой домен: www. важен)
* http://site.org (другой домен: .org важен)
* https://site.com (другой протокол: https)
* http://site.com:8080 (другой порт: 8080)

Политика «Одинакового источника» говорит, что:

* если у нас есть ссылка на другой объект window, например, на всплывающее окно, созданное с помощью window.open или на window из <iframe> и у этого окна тот же источник, то к нему будет полный доступ;
* в противном случае, если у него другой источник, нельзя обращаться к его переменным, объекту document и так далее. Единственное исключение – объект location: его можно изменять (таким образом перенаправляя пользователя). Но нельзя читать location (нельзя узнать, где находится пользователь, чтобы не было никаких утечек информации).

Внутри <iframe> находится по сути отдельное окно, то у окно, с собственными объектами document и window. Можно обращаться к ним, используя свойства:

* iframe.contentWindow ссылка на объект window внутри <iframe>;
* iframe.contentDocument – ссылка на объект document внутри <iframe>, короткая запись для iframe.contentWindow.document.

Когда код обращается к встроенному в ифрейм окну, браузер проверяет, имеет ли ифрейм тот же источник. Если это не так, тогда доступ будет запрещён (разрешена лишь запись в location, это исключение). Для примера давайте попробуем чтение и запись в ифрейм с другим источником:

<iframe src="https://example.com" id="iframe"></iframe>

<script>

iframe.onload = function() {

// можно получить ссылку на внутренний window

let iframeWindow = iframe.contentWindow; // OK

try {

// нельзя получить ссылку на document внутри него

let doc = iframe.contentDocument; // ОШИБКА

} catch(e) {

alert(e); // Security Error

}

// нельзя прочитать URL страницы в ифрейме

try {

// Нельзя читать из объекта Location

let href = iframe.contentWindow.location.href; // ОШИБКА

} catch(e) {

alert(e); // Security Error

}

// можно писать и загружать что-то другое в ифрейм

iframe.contentWindow.location = '/'; // OK

// уберём обработчик, чтобы не срабатывал после изменения location

iframe.onload = null;

};

</script>

Код выше выведет ошибку для любых операций, кроме:

* получения ссылки на внутренний объект window из iframe.contentWindow;
* изменения location.

С другой стороны, если у ифрейма тот же источник, то с ним можно делать всё, что угодно:

<!-- ифрейм с того же сайта -->

<iframe src="/" id="iframe"></iframe>

<script>

iframe.onload = function() {

// делаем с ним что угодно

iframe.contentDocument.body.prepend("Привет, мир!");

};

</script>

Событие iframe.onload – по сути то же, что и iframe.contentWindow.onload. Оно сработает, когда встроенное окно полностью загрузится со всеми ресурсами. Но iframe.onload всегда доступно извне ифрейма, в то время как доступ к iframe.contentWindow.onload разрешён только из окна с тем же источником.

По определению, если у двух URL разный домен, то у них разный источник. Но если в окнах открыты страницы с поддоменов одного домена 2-го уровня, например, john.site.com, peter.site.com и site.com (так что их общий домен site.com), то можно заставить браузер игнорировать это отличие. Так что браузер сможет считать их пришедшими с одного источника при проверке возможности доступа друг к другу. Для этого в каждом таком окне нужно запустить:

document.domain = 'site.com';

После этого они смогут взаимодействовать без ограничений. Ещё раз заметим, что это доступно только для страниц с одинаковым доменом второго уровня.

Когда ифрейм – с того же источника, код имеет доступ к документу в нём. Но когда ифрейм создан, в нём сразу есть документ. Но этот документ – другой, не тот, который в него будет загружен. Так что если сразу же сделать что-то с этим документом, то изменения, скорее всего, пропадут. Например:

<iframe src="/" id="iframe"></iframe>

<script>

let oldDoc = iframe.contentDocument;

iframe.onload = function() {

let newDoc = iframe.contentDocument;

// загруженный document - не тот, который был в iframe при создании изначально

alert(oldDoc == newDoc); // false

};

</script>

Не следует работать с документом ещё не загруженного ифрейма, так как это не тот документ. Если назначить ему обработчики событий – они будут проигнорированы. Чтобы поймать момент, когда появится правильный документ, можно проверять через setInterval:

<iframe src="/" id="iframe"></iframe>

<script>

let oldDoc = iframe.contentDocument;

// каждый 100 мс проверяем, не изменился ли документ

let timer = setInterval(() => {

let newDoc = iframe.contentDocument;

if (newDoc == oldDoc) return;

alert("New document is here!");

clearInterval(timer); // отключим setInterval

}, 100);

</script>

Другой способ получить объект window из <iframe> – забрать его из именованной коллекции window.frames:

* по номеру: window.frames[0] – объект window для первого фрейма в документе;
* по имени: window.frames.iframeName – объект window для фрейма со свойством name="iframeName".

Например:

<iframe src="/" style="height:80px" name="win" id="iframe"></iframe>

<script>

alert(iframe.contentWindow == frames[0]); // true

alert(iframe.contentWindow == frames.win); // true

</script>

Ифрейм может иметь другие ифреймы внутри. Таким образом, объекты window создают иерархию. Навигация по ним выглядит так:

* window.frames – коллекция «дочерних» window (для вложенных фреймов);
* window.parent – ссылка на «родительский» (внешний) window;
* window.top – ссылка на самого верхнего родителя.

Например:

window.frames[0].parent === window; // true

Можно использовать свойство top, чтобы проверять, открыт ли текущий документ внутри ифрейма или нет:

if (window == top) { // текущий window == window.top?

alert('Скрипт находится в самом верхнем объекте window, не во фрейме');

} else {

alert('Скрипт запущен во фрейме!');

}

Атрибут sandbox позволяет наложить ограничения на действия внутри <iframe>, чтобы предотвратить выполнение ненадежного кода. Атрибут помещает ифрейм в «песочницу», отмечая его как имеющий другой источник и/или накладывая на него дополнительные ограничения.

Существует список «по умолчанию» ограничений, которые накладываются на <iframe sandbox src="...">. Их можно уменьшить, если указать в атрибуте список исключений (специальными ключевыми словами), которые не нужно применять, например: <iframe sandbox="allow-forms allow-popups">. Другими словами, если у атрибута "sandbox" нет значения, то браузер применяет максимум ограничений, но через пробел можно указать те из них, которые не надо применять.

Список ограничений:

* allow-same-origin – "sandbox" принудительно устанавливает «другой источник» для ифрейма. Другими словами, он заставляет браузер воспринимать iframe, как пришедший из другого источника, даже если src содержит тот же сайт. Со всеми сопутствующими ограничениями для скриптов. Эта опция отключает это ограничение;
* allow-top-navigation – позволяет ифрейму менять parent.location;
* allow-forms – позволяет отправлять формы из ифрейма;
* allow-scripts – позволяет запускать скрипты из ифрейма;
* allow-popups – позволяет открывать всплывающие окна из ифрейма с помощью window.open.

Больше опций можно найти [в справочнике](https://developer.mozilla.org/ru/docs/Web/HTML/Element/iframe).

Атрибут "sandbox" создан только для того, чтобы добавлять ограничения. Он не может удалять их. В частности, он не может ослабить ограничения, накладываемые браузером на ифрейм, приходящий с другого источника.

Интерфейс postMessage позволяет окнам общаться между собой независимо от их происхождения. Это способ обойти политику «Одинакового источника». Он позволяет обмениваться информацией, скажем john-smith.com и gmail.com, но только в том случае, если оба сайта согласны и вызывают соответствующие JavaScript-функции. Это делает общение безопасным для пользователя.

Интерфейс имеет две части.

1. [postMessage](https://learn.javascript.ru/cross-window-communication" \l "postmessage) – окно, которое хочет отправить сообщение, должно вызвать метод [postMessage](https://developer.mozilla.org/ru/docs/Web/API/Window.postMessage) окна получателя. Другими словами, если надо отправить сообщение в окно win, тогда следует вызвать win.postMessage(data, targetOrigin).

Аргументы:

* data – данные для отправки. Может быть любым объектом, данные клонируются с использованием «алгоритма структурированного клонирования». IE поддерживает только строки, поэтому надо использовать метод JSON.stringify на сложных объектах, чтобы поддержать этот браузер.
* targetOrigin – определяет источник для окна-получателя, только окно с данного источника имеет право получить сообщение.

Например, здесь win получит сообщения только в том случае, если в нём открыт документ из источника <http://example.com>:

<iframe src="http://example.com" name="example">

<script>

let win = window.frames.example;

win.postMessage("message", "http://example.com");

</script>

Если не надо проверять, то в targetOrigin можно указать \*.

<iframe src="http://example.com" name="example">

<script>

let win = window.frames.example;

win.postMessage("message", "\*");

</script>

Пример вредоносной страницы. Для наглядности <iframe> полупрозрачный (на реальных вредоносных страницах он полностью прозрачен):

<style>

iframe { /\* ифрейм с сайта-жертвы \*/

width: 400px;

height: 100px;

position: absolute;

top:0; left:-20px;

opacity: 0.5; /\* в реальности opacity:0 \*/

z-index: 1;

}

</style>

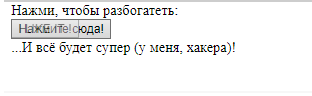
<div>Нажми, чтобы разбогатеть:</div>

<!-- Url с сайта-жертвы -->

<iframe src="/clickjacking/facebook.html"></iframe>

<button>Нажмите сюда!</button>

Полная демонстрация атаки:



// facebook.html

<!DOCTYPE HTML>

<html>

<body style="margin:10px;padding:10px">

<input type="button" onclick="alert('Вы нажали "Like" на сайте facebook.html!')" value="I LIKE IT !">

</body>

</html>

// index.html

<!doctype html>

<html>

<head>

<meta charset="UTF-8">

</head>

<body>

<style>

iframe {

width: 400px;

height: 100px;

position: absolute;

top: 5px;

left: -14px;

opacity: 0.5;

z-index: 1;

}

</style>

<div>Нажми, чтобы разбогатеть:</div>

<!-- Url с сайта-жертвы -->

<iframe src="facebook.html"></iframe>

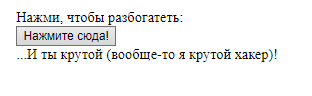
<button>Нажмите сюда!</button>

</body>

</html>

Здесь есть полупрозрачный <iframe src="facebook.html">, и в примере он располагается поверх кнопки. Клик на кнопку фактически кликает на ифрейм, но этого не видно пользователю, потому что ифрейм прозрачный. В результате, если пользователь авторизован на сайте Facebook («Запомнить меня» обычно активировано), то он добавляет «лайк», в Twitter это будет кнопка «читать», и т.п.

Вот тот же пример, но более приближенный к реальности с opacity:0 для <iframe>:



// facebook.html

<!DOCTYPE HTML>

<html>

<body style="margin:10px;padding:10px">

<input type="button" onclick="alert('Вы нажали "Like" на сайте facebook.html!')" value="I LIKE IT !">

</body>

</html>

// index.html

<!doctype html>

<html>

<head>

<meta charset="UTF-8">

</head>

<body>

<style>

iframe {

width: 400px;

height: 100px;

position: absolute;

top: 5px;

left: -14px;

opacity: 0;

z-index: 1;

}

</style>

<div>Нажми, чтобы разбогатеть:</div>

<!-- Url с сайта жертвы -->

<iframe src="facebook.html"></iframe>

<button>Нажмите сюда!</button>

</body>

</html>

Всё, что необходимо для атаки — это расположить <iframe> на вредоносной странице так, чтобы кнопка находилась прямо над ссылкой. Так что пользователь, кликающий по ссылке, на самом деле будет нажимать на кнопку в <iframe>. Обычно это можно сделать с помощью CSS-позиционирования.

Clickjacking-атака срабатывает только на действия мыши (или аналогичные, вроде нажатия пальцем на мобильном устройстве). Клавиатурный ввод гораздо сложнее перенаправить. Технически, если есть текстовое поле для взлома, то можно расположить ифрейм таким образом, чтобы текстовые поля перекрывали друг друга. Тогда посетитель при попытке сфокусироваться на текстовом поле, которое он видит на странице, фактически будет фокусироваться на текстовом поле внутри ифрейм.

Но есть одна проблема. Всё, что посетитель печатает, будет скрыто, потому что ифрейм не виден. Обычно люди перестают печатать, когда не видят на экране новых символов.

1. [onmessage](https://learn.javascript.ru/cross-window-communication#onmessage) – чтобы получать сообщения, окно-получатель должно иметь обработчик события message (сообщение). Оно срабатывает, когда был вызван метод postMessage (и проверка targetOrigin пройдена успешно).

Объект события имеет специфичные свойства:

* data – данные из postMessage;
* origin – источник отправителя, например, http://javascript.info;
* source – ссылка на окно-отправитель. Можно сразу отправить что-то в ответ, вызвав source.postMessage(...).

Чтобы добавить обработчик, следует использовать метод addEventListener, короткий синтаксис window.onmessage не работает. Вот пример:

window.addEventListener("message", function(event) {

if (event.origin != 'http://javascript.info') {

// данные с неизвестного домена игнорируем

return;

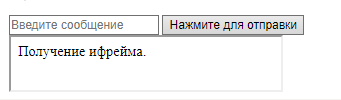
}

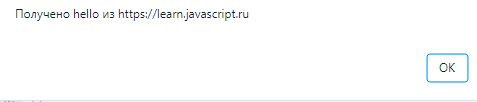
alert( "received: " + event.data );

// can message back using event.source.postMessage(...)

});

Полный пример:





// iframe.html

<!doctype html>

<html>

<head>

<meta charset="UTF-8">

</head>

<body>

Получение ифрейма.

<script>

window.addEventListener('message', function(event) {

alert(`Получено ${event.data} из ${event.origin}`);

});

</script>

</body>

</html>

// index.html

<!doctype html>

<html>

<head>

<meta charset="UTF-8">

</head>

<body>

<form id="form">

<input type="text" placeholder="Введите сообщение" name="message">

<input type="submit" value="Нажмите для отправки">

</form>

<iframe src="iframe.html" id="iframe" style="display:block;height:60px"></iframe>

<script>

form.onsubmit = function() {

iframe.contentWindow.postMessage(this.message.value, '\*');

return false;

};

</script>

</body>

</html>

Между postMessage и событием message не существует задержки. Событие происходит синхронно, быстрее, чем setTimeout(...,0).

1. Атака типа clickjacking

Атака типа clickjacking (англ. «захват клика») позволяет вредоносной странице кликнуть по сайту-жертве от имени посетителя. Многие сайты были взломаны подобным способом, включая Twitter, Facebook, Paypal и другие.

Идея этой атаки очень проста. Вот как clickjacking-атака была проведена на Facebook:

1. Посетителя заманивают на вредоносную страницу (неважно как).
2. На странице есть ссылка, которая выглядит безобидно (например, «Разбогатей прямо сейчас» или «Нажми здесь, это очень смешно»).
3. Поверх этой ссылки вредоносная страница размещает прозрачный <iframe> с src с сайта facebook.com таким образом, что кнопка «like» находится прямо над этой ссылкой. Обычно это делается с помощью z-index в CSS.
4. При попытке клика на эту ссылку посетитель на самом деле нажимает на кнопку.

Самым старым и слабым вариантом защиты является код JavaScript, запрещающий открытие страницы во фрейме (это называют «framebusting»). Выглядит он вот так:

if (top != window) {

top.location = window.location;

}

В этом случае, если окно обнаруживает, что оно открыто во фрейме, оно автоматически располагает себя сверху. Этот метод не является надёжной защитой, поскольку появилось множество способов его обойти. Рассмотрим некоторые из них.

Можно заблокировать переход, вызванный сменой top.location в обработчике события beforeunload. Внешняя страница (принадлежащая хакеру) устанавливает обработчик на это событие, отменяющий его, например, такой:

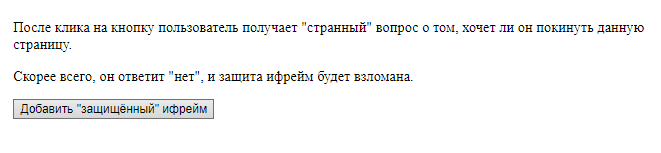
window.onbeforeunload = function() {

return false;

};

Когда iframe пытается изменить top.location, посетитель увидит сообщение с вопросом действительно ли он хочет покинуть эту страницу. В большинстве случаев посетитель ответит отрицательно, поскольку он не знает об ифрейме: всё, что он видит – это верхнюю страницу, которую нет причин покидать. Поэтому top.location не изменится. В действии:

Результат



// iframe.html

<!doctype html>

<html>

<head>

<meta charset="UTF-8">

</head>

<body>

<div>Замена top.location на javascript.info</div>

<script>

top.location = 'https://javascript.info';

</script>

</body>

</html>

// index.html

<!doctype html>

<html>

<head>

<meta charset="UTF-8">

<style>

iframe {

width: 400px;

height: 100px;

position: absolute;

top: 0;

left: -20px;

opacity: 0;

z-index: 1;

}

</style>

<script>

function attack() {

window.onbeforeunload = function() {

window.onbeforeunload = null;

return false;

};

document.body.insertAdjacentHTML('beforeend', '<iframe src="iframe.html">');

}

</script>

</head>

<body>

<p>После клика на кнопку пользователь получает "странный" вопрос о том, хочет ли он покинуть данную страницу.</p>

<p>Скорее всего, он ответит "нет", и защита ифрейм будет взломана.</p>

<button onclick="attack()">Добавить "защищённый" ифрейм</button>

</body>

</html>

Одним из действий, которые можно ограничить атрибутом sandbox, является навигация. Соответственно ифрейм внутри sandbox не изменит top.location. Поэтому можно добавить ифрейм с sandbox="allow-scripts allow-forms". Это снимет некоторые ограничения, разрешая при этом использование скриптов и форм. Но надо опустить allow-top-navigation, чтобы изменение top.location было запрещено. Вот код этого примера:

<iframe sandbox="allow-scripts allow-forms" src="facebook.html"></iframe>

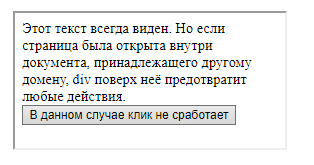
Есть и другие способы обойти эту простую защиту.

Заголовок X-Frame-Options со стороны сервера может разрешать или запрещать отображение страницы внутри фрейма. Это должен быть именно HTTP-заголовок: браузер проигнорирует его, если найдёт в HTML-теге <meta>. Поэтому при <meta http-equiv="X-Frame-Options"...> ничего не произойдёт.

Заголовок может иметь 3 значения:

* DENY – никогда не показывать страницу внутри фрейма;
* SAMEORIGIN – разрешить открытие страницы внутри фрейма только в том случае, если родительский документ имеет тот же источник.
* ALLOW-FROM domain – разрешить открытие страницы внутри фрейма только в том случае, если родительский документ находится на указанном в заголовке домене.

У заголовка X-Frame-Options есть побочный эффект. Другие сайты не смогут отобразить страницу во фрейме, даже если у них будут на то веские причины. Поэтому есть другие решения. Например, можно «накрыть» страницу блоком <div> со стилями height: 100%; width: 100%;, чтобы он перехватывал все клики. Этот <div> можно убрать, если window == top или если защита не нужна. Примерно так:



// iframe.html

<!doctype html>

<html>

<head>

<meta charset="UTF-8">

<style>

#protector {

height: 100%;

width: 100%;

position: absolute;

left: 0;

top: 0;

z-index: 99999999;

}

</style>

</head>

<body>

<div id="protector">

<a href="/" target="\_blank">Перейти к сайту</a>

</div>

<script>

if (top.document.domain == document.domain) {

protector.remove();

}

</script>

Этот текст всегда виден.

Но если страница была открыта внутри документа, принадлежащего другому домену, div поверх неё предотвратит любые действия.

<button onclick="alert(1)">В данном случае клик не сработает</button>

</body>

</html>

// index.html

<!doctype html>

<html>

<head>

<meta charset="UTF-8">

</head>

<body>

<iframe src="iframe.html"></iframe>

</body>

</html>

Атрибут samesite также может помочь избежать clickjacking-атаки. Файл куки с таким атрибутом отправляется на сайт только в том случае, если он открыт напрямую, не через фрейм или каким-либо другим способом. Если сайт, такой как Facebook, при установке авторизующего куки ставит атрибут samesite:

Set-Cookie: authorization=secret; samesite

тогда такие куки не будут отправляться, когда Facebook будет открыт в ифрейме с другого сайта. Так что атака не удастся.

Атрибут samesite не играет никакой роли, если куки не используются. Так что другие веб-сайты смогут отображать публичные, не требующие авторизации, страницы в ифрейме. Однако, это даёт возможность в некоторых ситуациях осуществить clickjacking-атаку, например, на сайт для анонимных опросов, который предотвращает повторное голосование пользователя путём проверки IP-адреса. Он останется уязвимым к атаке, потому что не аутентифицирует пользователей с помощью куки.