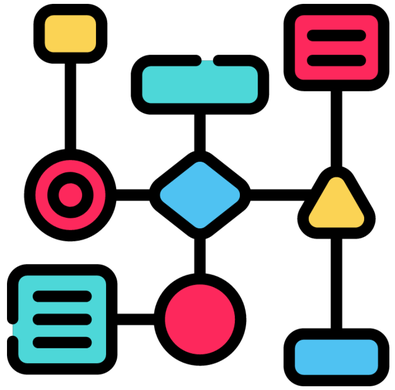
конференция
РусКрипто

21 марта 2019, Солнечногорск



Динамический анализ потоков данных JavaScript-кода

Хашаев Артур Акрамович

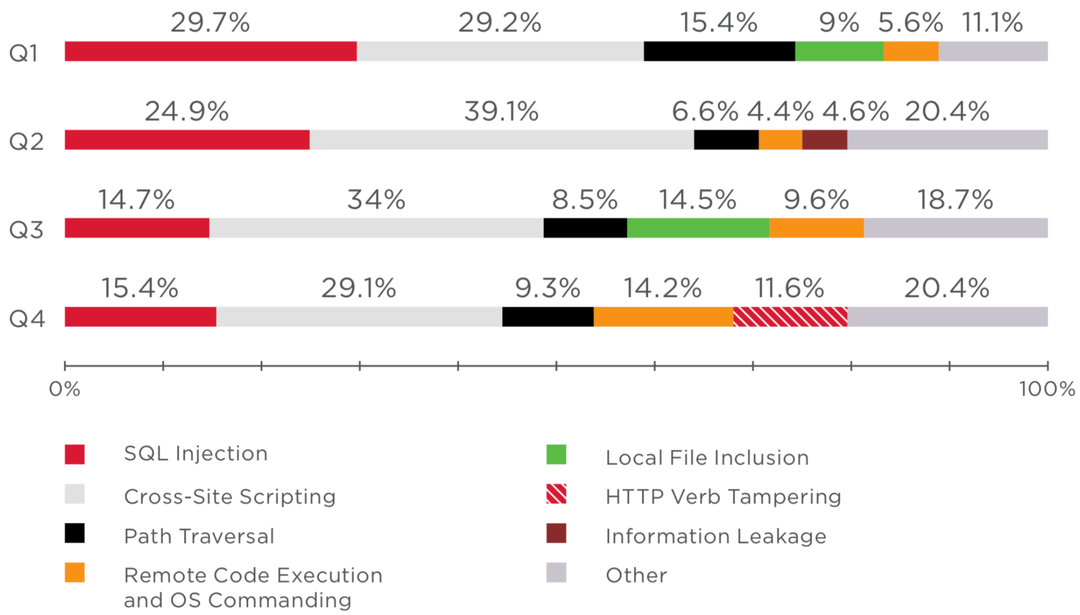
arthur@khashaev.ru

МГУ им. М.В. Ломоносова
Лаборатория интеллектуальных систем кибербезопасности

Распространенность XSS

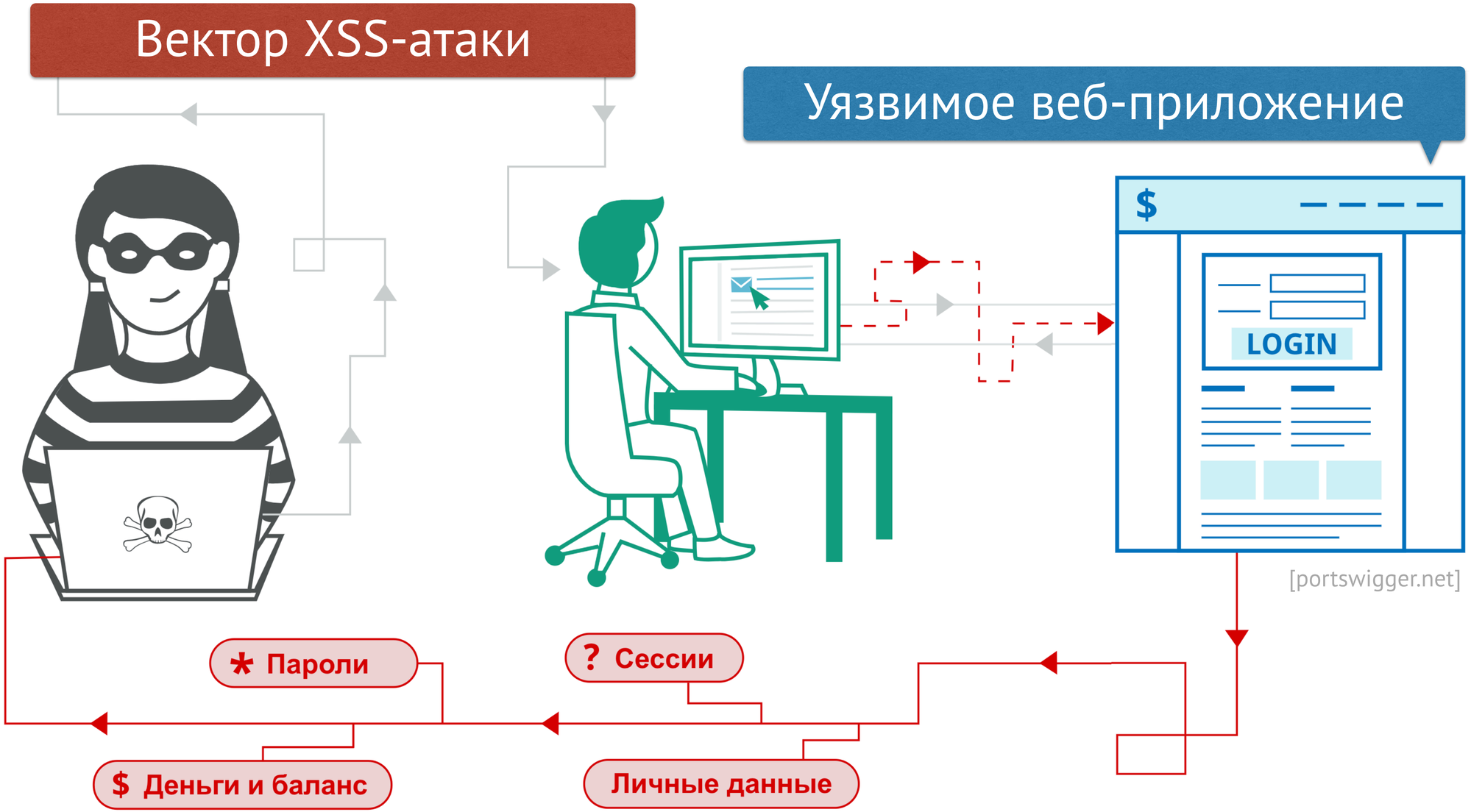
- По различным оценкам, атаки класса XSS составляют ≈30–40% от всех атак на веб-приложения [PT, 2017; WhiteHat, 2017]
- Входят в Top-10 наиболее критичных рисков [OWASP, 2017]

[Positive Technologies, 2017]



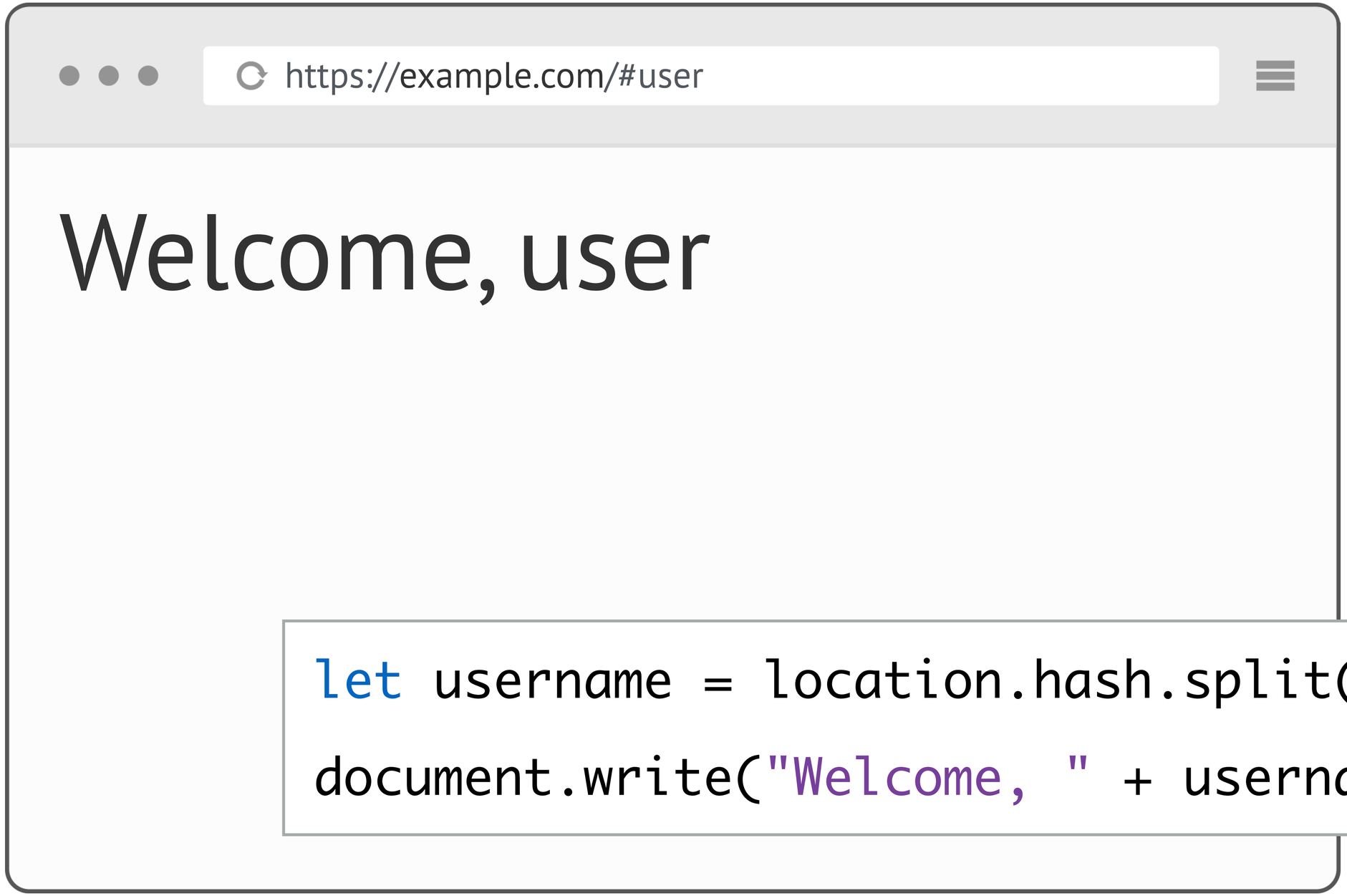
Распространенность DOM XSS

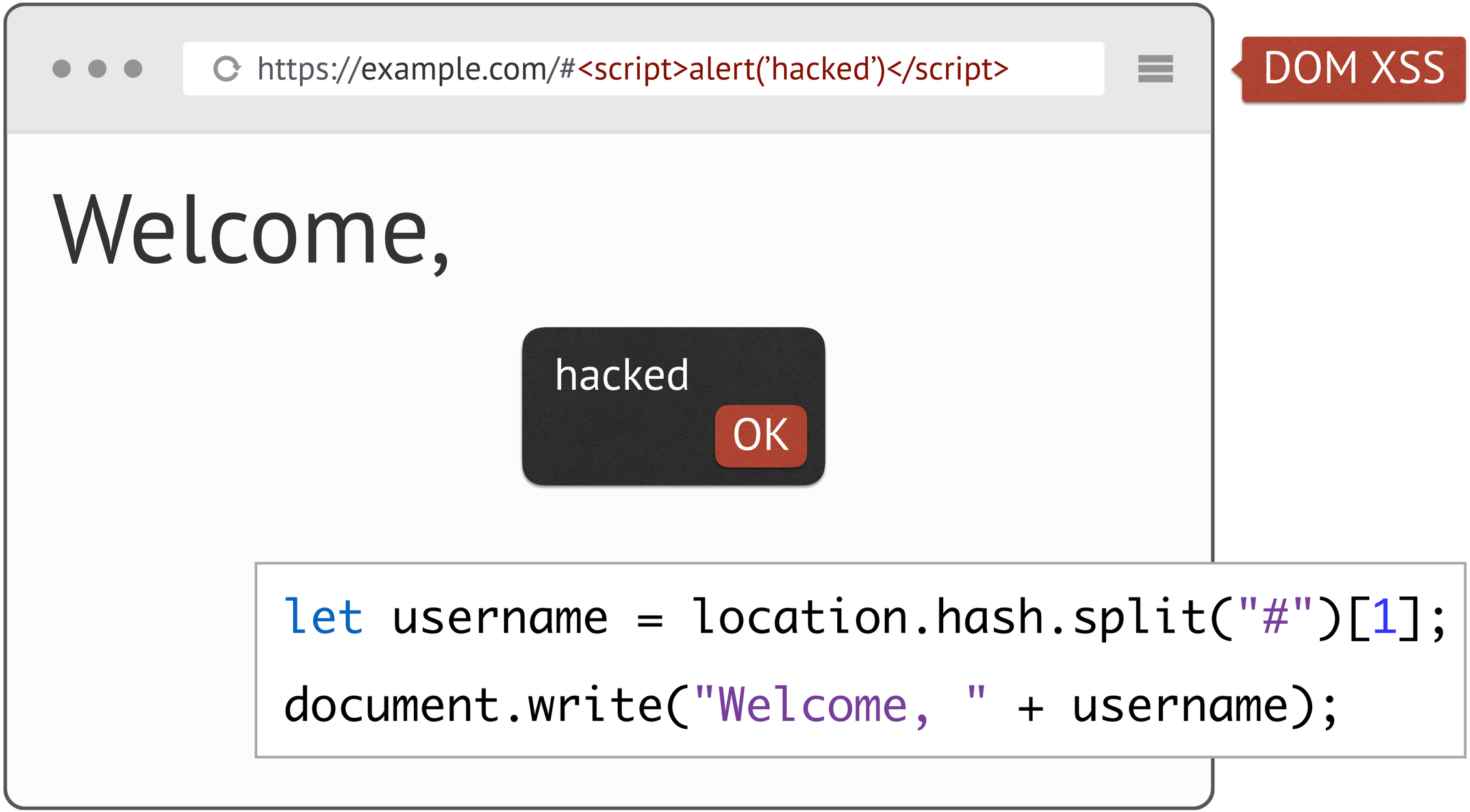
- DOM XSS — крупный подкласс XSS, ≈40%
  - Document Object Model — объектная модель документа
  - Модификация DOM производится языком программирования JavaScript в браузерном окружении
- Набирает популярность в связи с развитием концепций насыщенных (RIA) и одностраничных (SPA) приложений
- Являются довольно сложными для обнаружения, поскольку производятся исключительно на клиентской стороне приложений


Вектор XSS-атаки
Уязвимое веб-приложение
$
LOGIN
[portswigger.net]
* Пароли
? Сессии
$ Деньги и баланс
Личные данные

Риски

- XSS — это в первую очередь атака на пользователя
- В основном популярны:
  - Кражи пользовательских данных и сессий
  - Совершение нелегитимных действий от лица пользователя в контексте веб-приложения
- Атаки могут носить массовый характер


https://example.com/#user
Welcome, user
let username = location.hash.split("#")[1];
document.write("Welcome, " + username);


https://example.com/#<script>alert('hacked')</script>
DOM XSS
Welcome,
hacked
OK
let username = location.hash.split("#")[1];
document.write("Welcome, " + username);

Анализ потоков данных

- Определение уязвимости — модель невмешательства [Goguen & Meseguer, 1982]
- sources → taint propagation | filters → sinks
- flow = ⟨source, sink⟩

```
let username = location.hash.split("#")[1];
document.write("Welcome, " + username);
```

Типичные источники DOM XSS

- URL-адрес и его составные части
- Cookie
- HTTP-заголовок Referrer
- Заголовок окна, контролируемый открывающим
- История браузера — HTML5 History API
- Локальные и сессионные хранилища — HTML5 Storage Objects
- Хранимые на сервере и контролируемые пользователем данные
- PostMessage API

Типичные стоки DOM XSS

- Прямое исполнение кода — `eval`, конструктор `Function`, таймеры `setTimeout`, атрибут `src`, обработчики событий (например, `onload`), …
- Манипуляции с HTML — `document.write`, свойство `innerHTML`, …
- Стилевые CSS-стоки, которые обычно приводят к UI Redressing
- Взаимодействие с сервером через `XMLHttpRequest` и Fetch API
- Cookie
- URL-адрес
- Стоки различных библиотек, например, функция $ в jQuery

API тейнт-анализатора

тейнтирование значения

```
let username = Flow.tainted(location).hash.split("#")[1];
document.write(Flow.detect("Welcome, " + username));
```

логгирует путь вычисления в случае наличия тейнтирования

Примитивы:

- Тейнтирование — taint
- Снятие тейнтирования — release
- Проверка тейнтирования — isTainted

Возможные подходы

- Использовать доступные средства:
  - ES6 Proxy, Reflect, Debugger API, …
  - Поведение изменяется
  - Множество ограничений
- Модификация среды выполнения
- Инструментация кода

Возможные подходы

- Использовать доступные средства
- Модификация среды выполнения:
  - Создает жесткую зависимость
  - Проблема с обновлениями, трудности в поддержке
  - Среда выполнения плохо на это рассчитана
  - Большая кодовая база (например, в V8 более 10^6 строк кода)
- Инструментация кода

Возможные подходы

- Использовать доступные средства
- Модификация среды выполнения
- Инструментация кода:
  - Необходимо сохранить наблюдаемое поведение
  - JavaScript — мультипарадигменный язык программирования со слабой динамической типизацией; есть еще BOM и DOM
  - Множество особенностей и краевых случаев

Инструментация кода

```
let foo = bar();
```

Инструментация кода

```
let foo = bar();
```

функция-перехватчик

```
let foo = taintflow.intercept({
    type: "CallExpression",
    callee: () => new taintflow.Identifier(() => bar),
    arguments: () => [],
}).value;
```

Инструментация кода

```
let foo = bar("Welcome, " + username);
```

Инструментация кода

```
let foo = bar("Welcome, " + username);
```

```
let foo = taintflow.intercept({
    type: "CallExpression",
    callee: () => new taintflow.Identifier(() => bar),
    arguments: () => [taintflow.intercept({
        type: "BinaryExpression",
        operator: "+",
        left: () => new taintflow.RValue("Welcome, "),
        right: () => new taintflow.Identifier(() => username),
    })],
}).value;
```

Общий подход

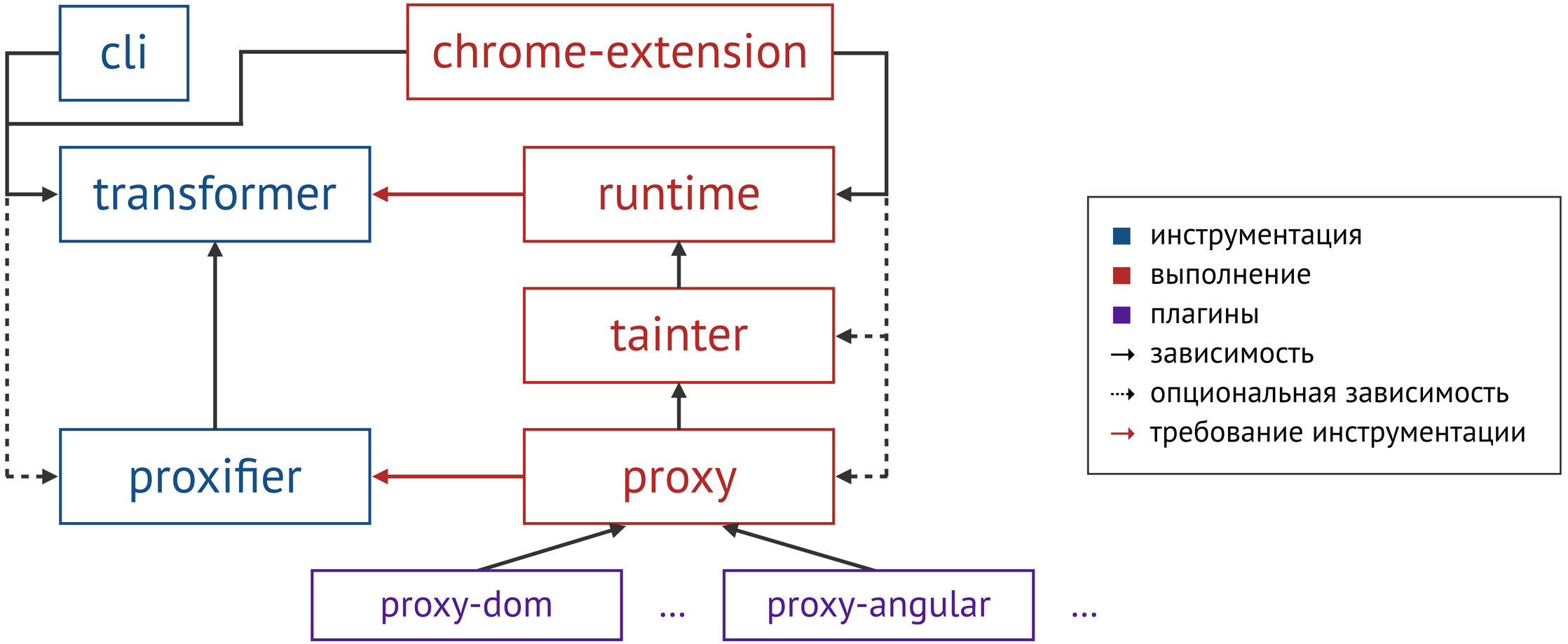
- Инструментируем AST с помощью унифицированного метода
- Подключаем специальную библиотеку — среду времени выполнения, которая работает только в инструментированном коде
- Стандартный перехватчик сохраняет семантику кода
- Является расширяемым, что позволяет разрабатывать произвольные анализаторы
- Можно подключить анализ распространения меток

tf TaintFlow

taintflow.org

- ПО для инструментации
  - Генерация карт кода (*source maps*) для удобства отладки
  - Поддержка стандартов ECMAScript 5–9
- Расширяемая среда выполнения
- Тейнт-анализатор потоков данных
- Консольная утилита
- Расширение для браузера Google Chrome

Архитектура



Тестирование

- Синтетические тесты:
  - Более 50 автоматизированных тестовых сценариев
  - Покрываются различные конструкции языка JavaScript
  - Реализована песочница
  - Обеспечено покрытие кода на уровне 98%
- Тестирование на реальных сайтах и веб-приложениях

Тестирование

- Синтетические тесты

- Тестирование на реальных сайтах и веб-приложениях:
  - Список [Alexa], а также отобранные приложения на популярных клиентских фреймворках React и Angular:
    - Сохранение семантики
    - Производительность
  - Корпус уязвимых приложений [DV]:
    - Распространение меток

Тестирование

#	Веб-сайт	Σ, Кб	Σ′, Кб	Σ′/Σ	(К1)	(К2)
1	google.com	128	1 639	12.76	✓	✓
2	facebook.com	2 742	16 764	6.11	✓	✓
3	en.wikipedia.org	268	1 488	5.54	✓	✓
4	yahoo.com	911	8 952	9.82	✓	✓
5	reddit.com	1 330	14 014	10.53	✓	✓
6	twitter.com	687	5 855	8.51	✓	✓
7	amazon.com	995	9 265	9.30	✓	✓
8	vk.com	1 777	12 048	6.78	✓	✓

…

- Top 25 из списка [Alexa]
- Σ — суммарный размер кода до инструментации
- Σ’ — после

- (К1) — критерий отсутствия исключений
- (К2) — критерий сохранения наблюдаемого поведения

Кратко о накладных расходах

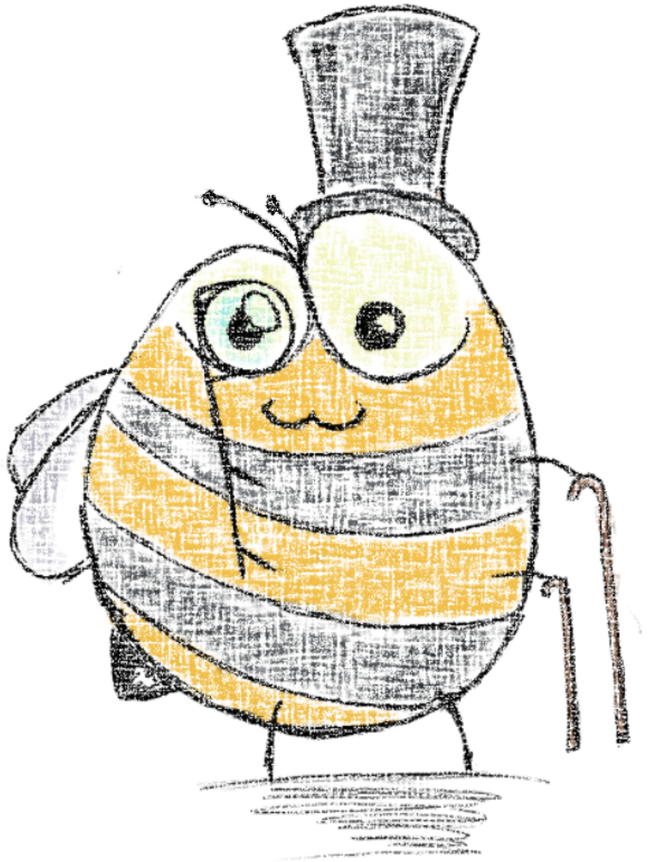
- Вычислительная сложность метода инструментации — O(n)
- Линейный рост размера инструментированного кода,
  в среднем увеличивается в 8.95 раз
- Потребление памяти во время выполнения возрастает в 4 раза,
  процессорного времени — в 5 раз ($\sigma = 0.76\%$)

Текущие результаты

- Рассмотрены возможные подходы к решению задачи динамического анализа потоков данных JavaScript-кода в контексте автоматизированного поиска уязвимостей класса DOM XSS в клиентской части современных веб-приложений
- Собрана коллекция источников и стоков DOM XSS
- Предложен и реализован метод решения задачи распространения меток путем инструментации кода с помощью унифицированного переписывания абстрактного синтаксического дерева
- Продемонстрирован способ разметки объектов и значений в памяти программы с последующим распространением меток по ходу ее выполнения
- Предложен способ эволюции предложенного решения в динамический анализатор, нацеленный на поиск уязвимостей в клиентской части веб-приложений

Дальнейшие исследования

- Исследование и создание комплексных анализаторов, учитывающих наличие фильтров и отсеивающих ложноположительные срабатывания
- Разработка библиотеки для описания источников, фильтров и стоков с целью автоматического перехвата обращений к ним
- Автоматический фаззинг



Контакты

Хашаев Артур Акрамович

arthur@khashaev.ru · khashaev.ru · telegram: inviz

МГУ им. М.В. Ломоносова
Лаборатория интеллектуальных систем кибербезопасности