POSITIVE DEVELOPMENT USER GROUP

# Итак, в вашем коде нашли инъекцию…

Positive Technologies
Владимир Кочетков

# :~$ whoami

Руководитель отдела исследований по анализу защищённости приложений Positive Technologies

AppSec- и CS-исследователь (формальные методы анализа и защиты кода)

Организатор Positive Development User Group

https://about.me/vladimir.kochetkov

https://kochetkov.github.io/

vkochetkov@ptsecurity.com

# Positive Development User Group

Открытое сообщество разработчиков и IT-специалистов, которые стремятся создавать безопасные приложения.

https://t.me/ru_appsec

# Intro (1/3)

Ещё один унылый доклад про параметризацию SQL-запросов?

# Intro (2/3)

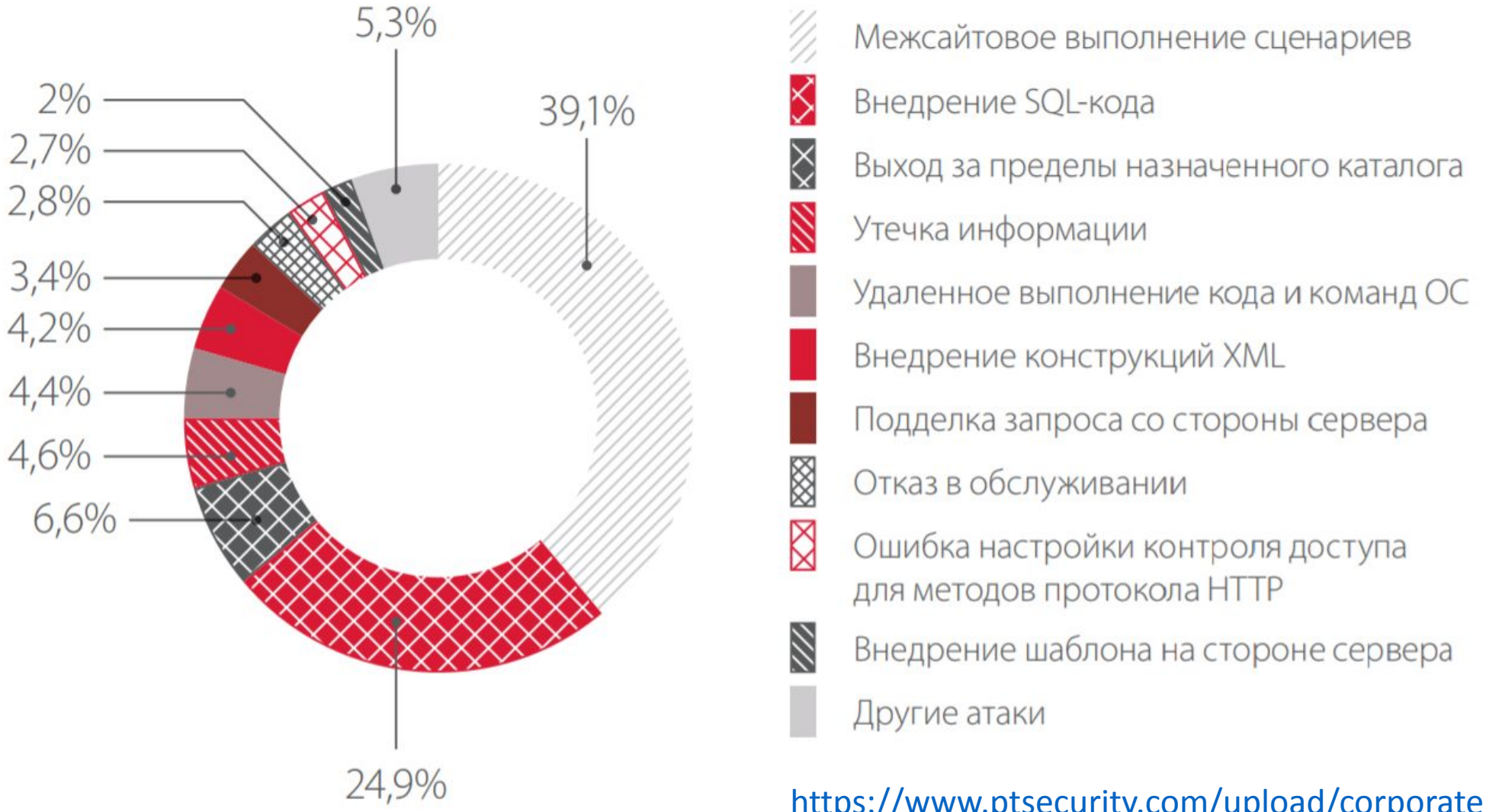
Существует ли конечное множество правил разработки защищённых приложений?

# Intro (3/3)

Если ты хочешь построить корабль, не надо созывать людей, планировать, делить работу, доставать инструменты.

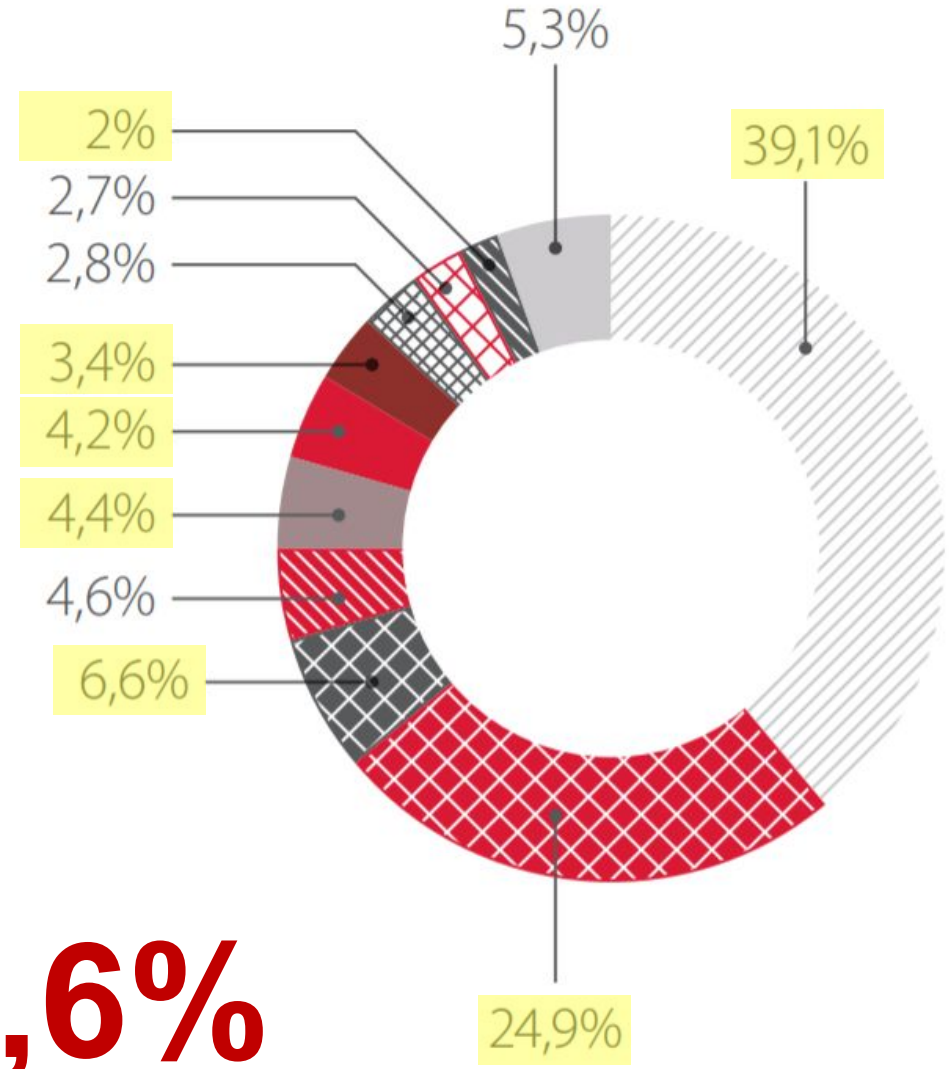
Надо заразить людей стремлением к бесконечному морю…

# Инъекции – самый распространённый тип атак (1/2)




https://www.ptsecurity.com/upload/corporate/ru-ru/analytics/WebApp-Vulnerabilities-2017-rus.pdf

# Инъекции – самый распространённый тип атак (2/2)




- Межсайтовое выполнение сценариев
- Внедрение SQL-кода
- Выход за пределы назначенного каталога
- Утечка информации
- Удаленное выполнение кода и команд ОС
- Внедрение конструкций XML
- Подделка запроса со стороны сервера
- Отказ в обслуживании
- Ошибка настройки контроля доступа для методов протокола HTTP
- Внедрение шаблона на стороне сервера
- Другие атаки

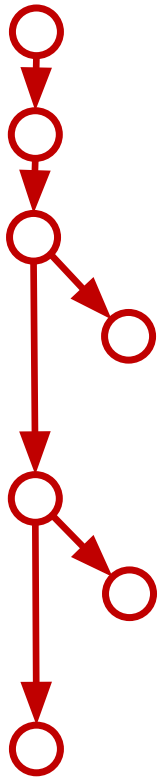
**84,6%**

https://www.ptsecurity.com/upload/corporate/ru-ru/analytics/WebApp-Vulnerabilities-2017-rus.pdf

# Формально об инъекциях

# Модель инъекции (1/7)

Граница окружения

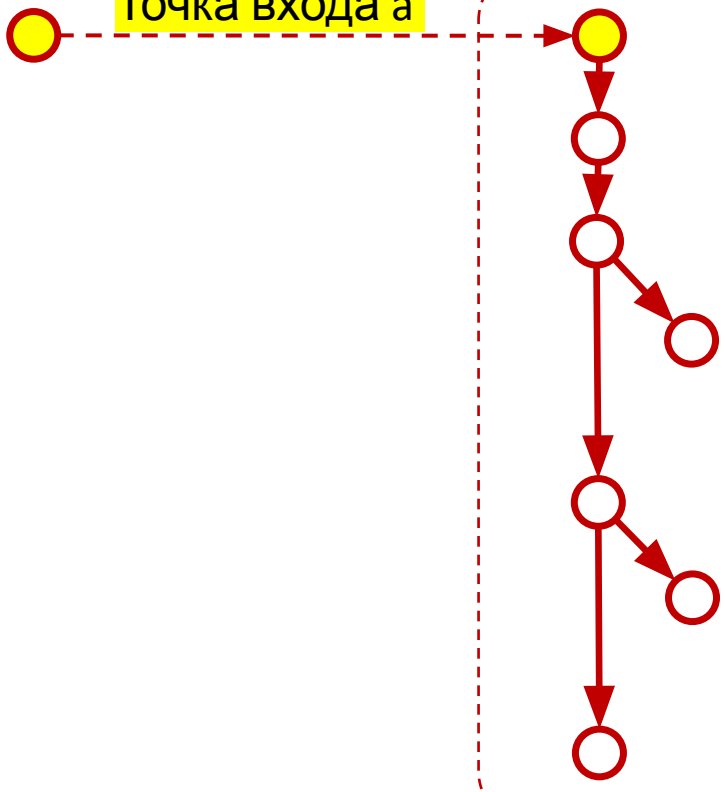


```
01 var a = Request.Params["a"];
02
03 var b = Request.Params["b"];
04
05 if (a == null)
06 {
07   return;
08 }
09
10 if (b == null)
11 {
12   return;
13 }
14
15 Response.Write($"<img src='//host/1/{a}/' onclick='f({b})'/>");
```

# Модель инъекции (2/7)

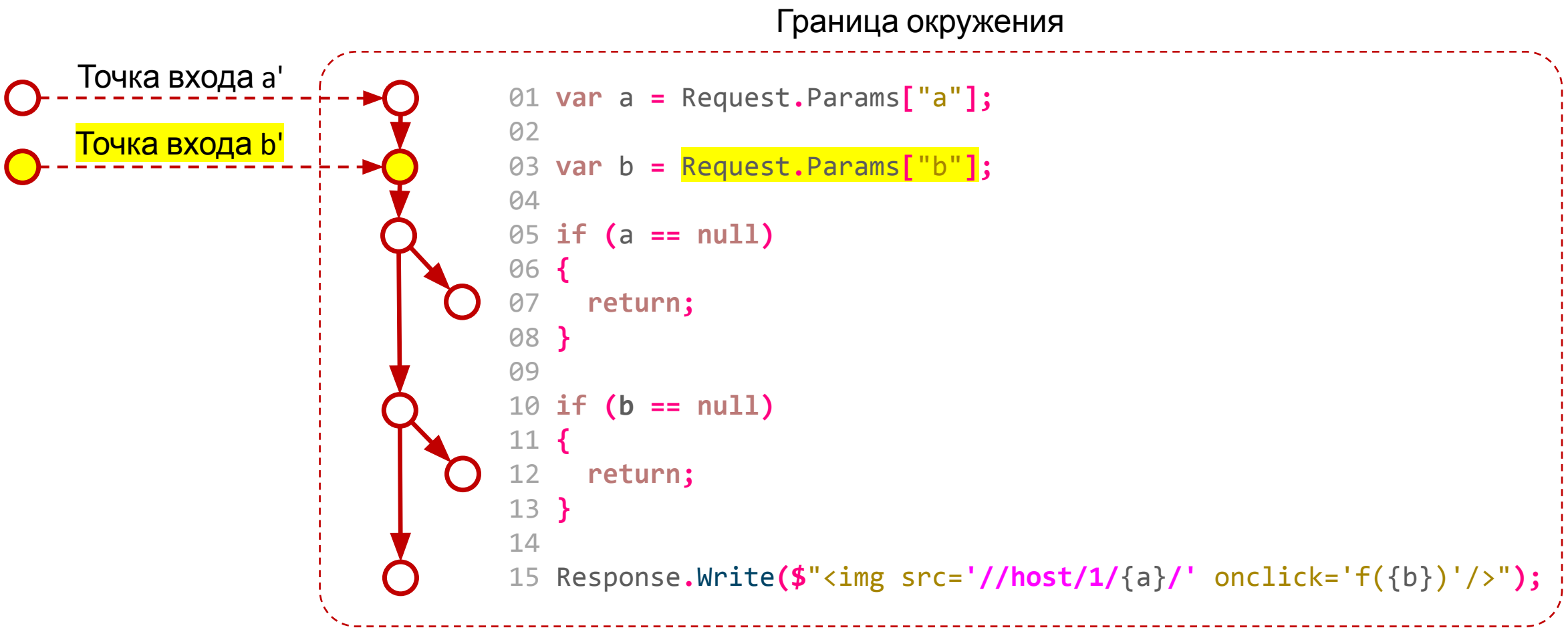
Граница окружения

Точка входа a'

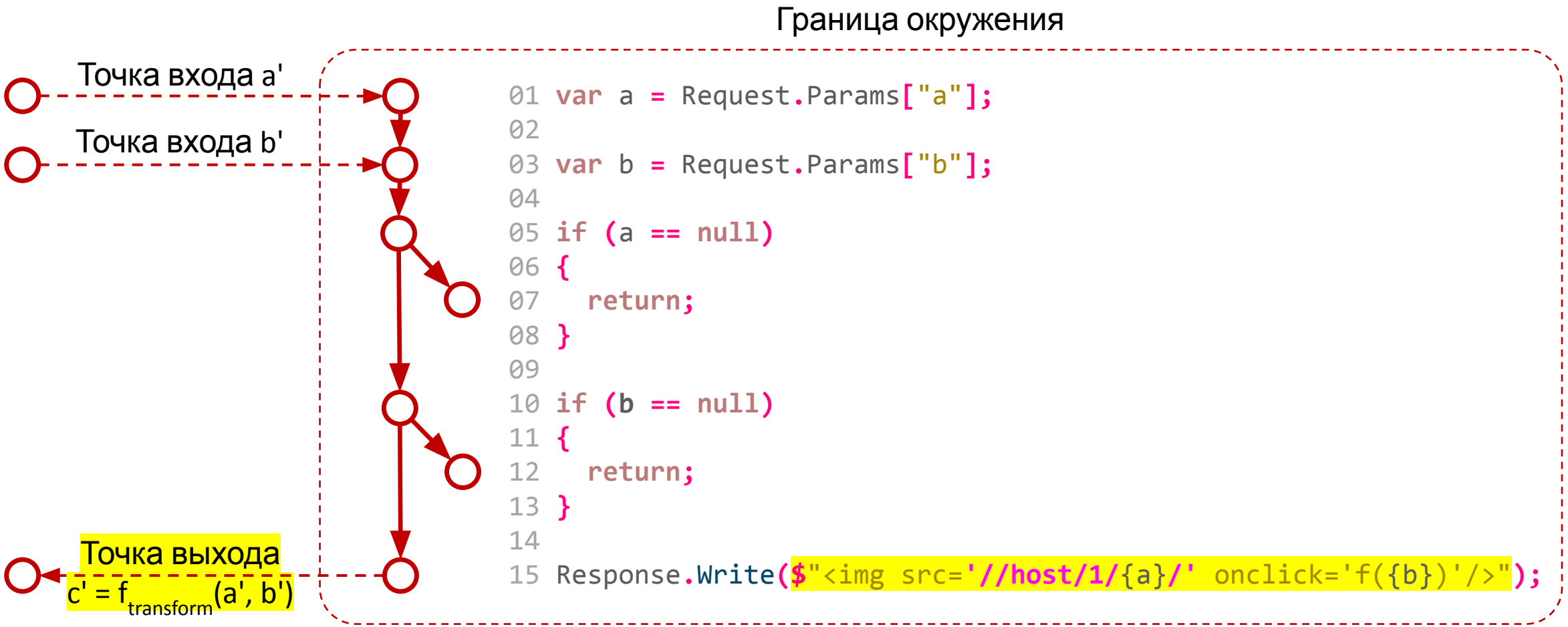


```
01 var a = Request.Params["a"];
02
03 var b = Request.Params["b"];
04
05 if (a == null)
06 {
07   return;
08 }
09
10 if (b == null)
11 {
12   return;
13 }
14
15 Response.Write($"<img src='//host/1/{a}/' onclick='f({b})'/>");
```

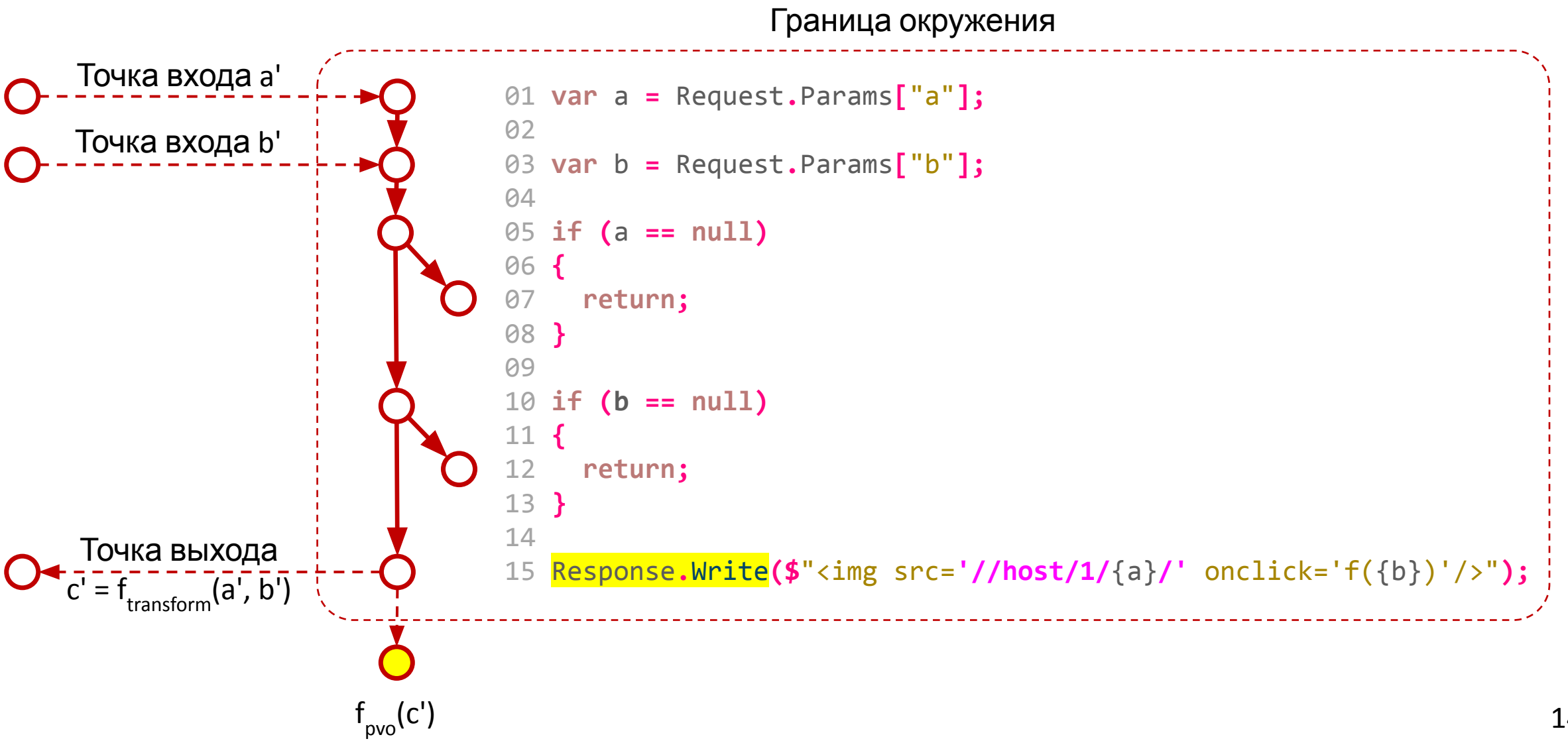
# Модель инъекции (3/7)



# Модель инъекции (4/7)

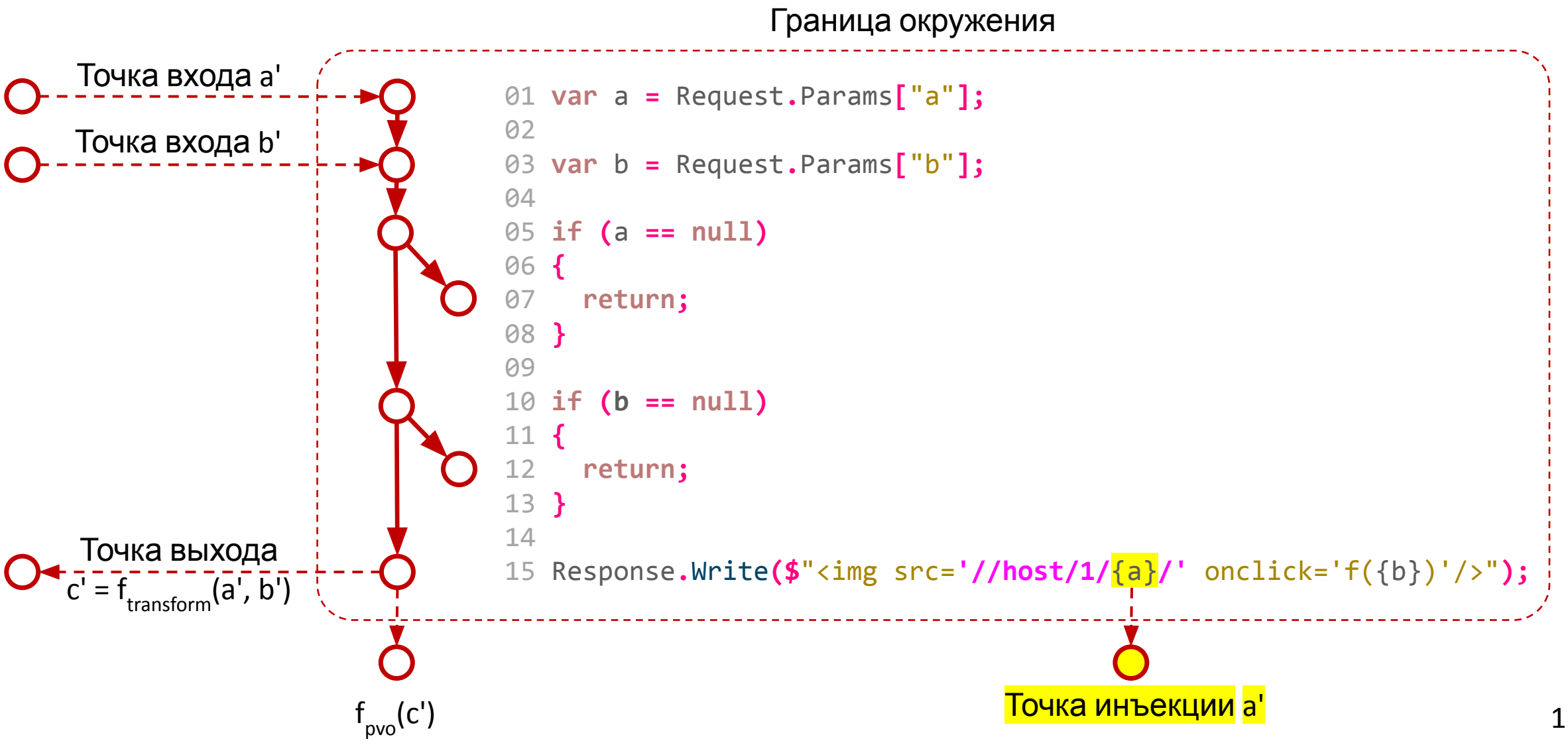


# Модель инъекции (5/7)

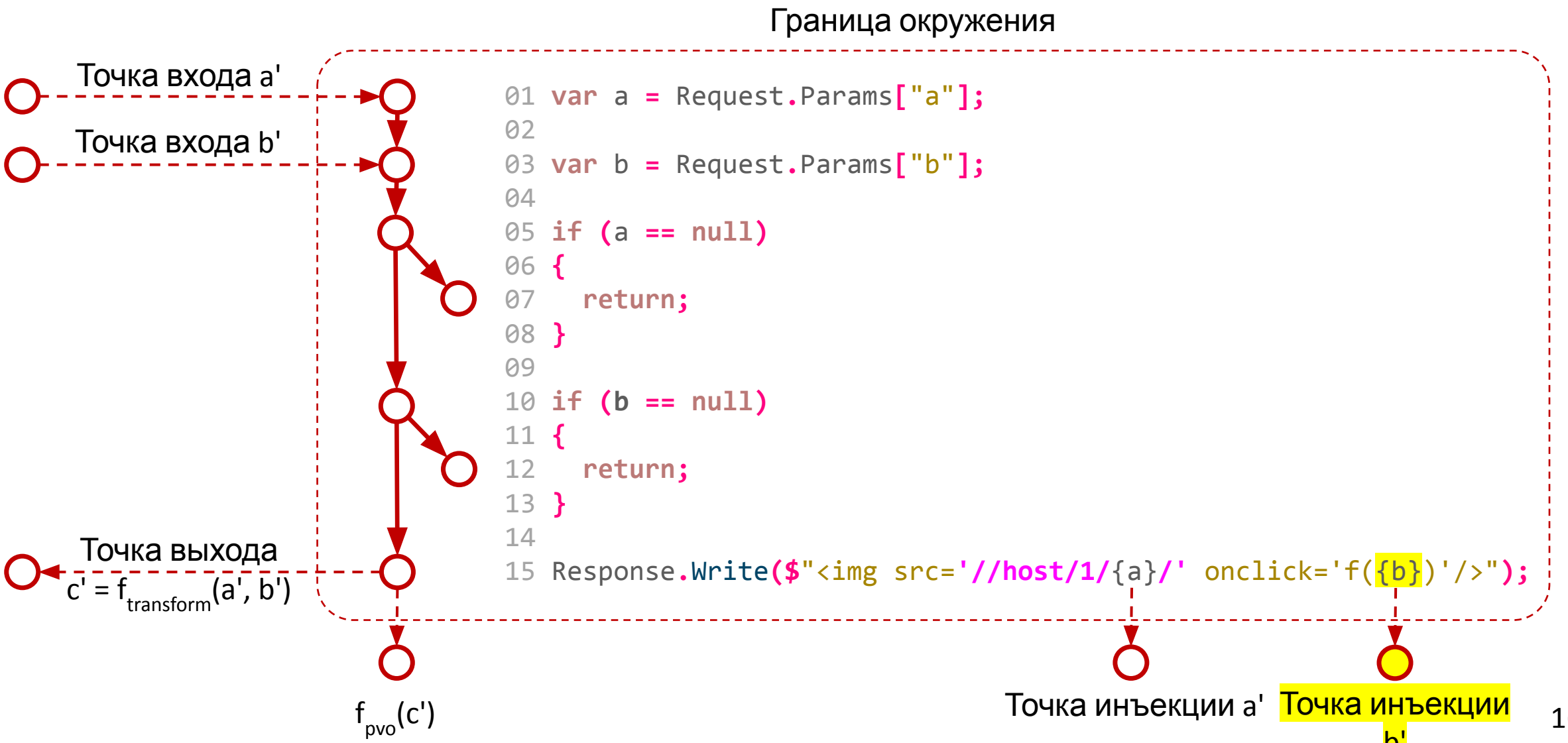



```
01 var a = Request.Params["a"];
02
03 var b = Request.Params["b"];
04
05 if (a == null)
06 {
07   return;
08 }
09
10 if (b == null)
11 {
12   return;
13 }
14
15 Response.Write($"<img src='//host/1/{a}/' onclick='f({b})'/>");
```

# Модель инъекции (6/7)

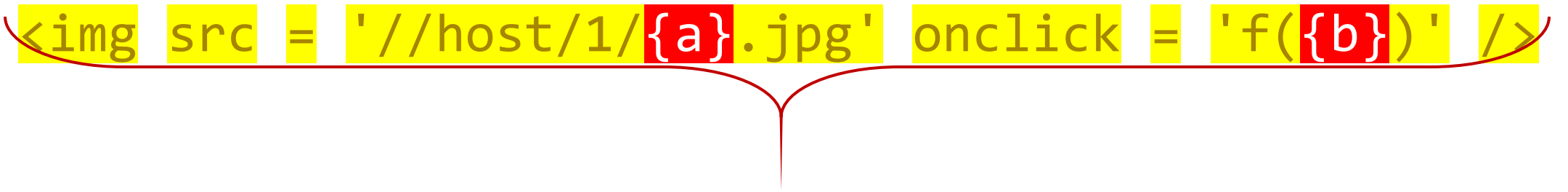


# Модель инъекции (7/7)



# Достаточный критерий защищённости от атак инъекций

Приложение защищено от атак инъекций тогда, когда в результате лексического разбора любого возможного c', количество токенов, приходящихся на точки инъекции, является константой:

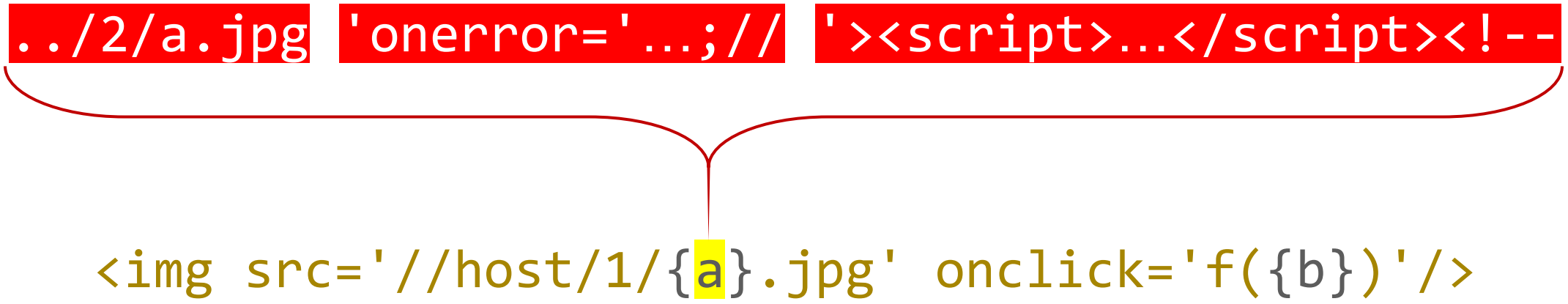



2 токена

# Необходимый критерий защищённости от атак инъекций

Приложение защищено от атак инъекций тогда, когда в результате лексического разбора любого возможного c', множества токенов, приходящихся на точки инъекции, являются авторизованными.

# Что может атакующий? (1/2)



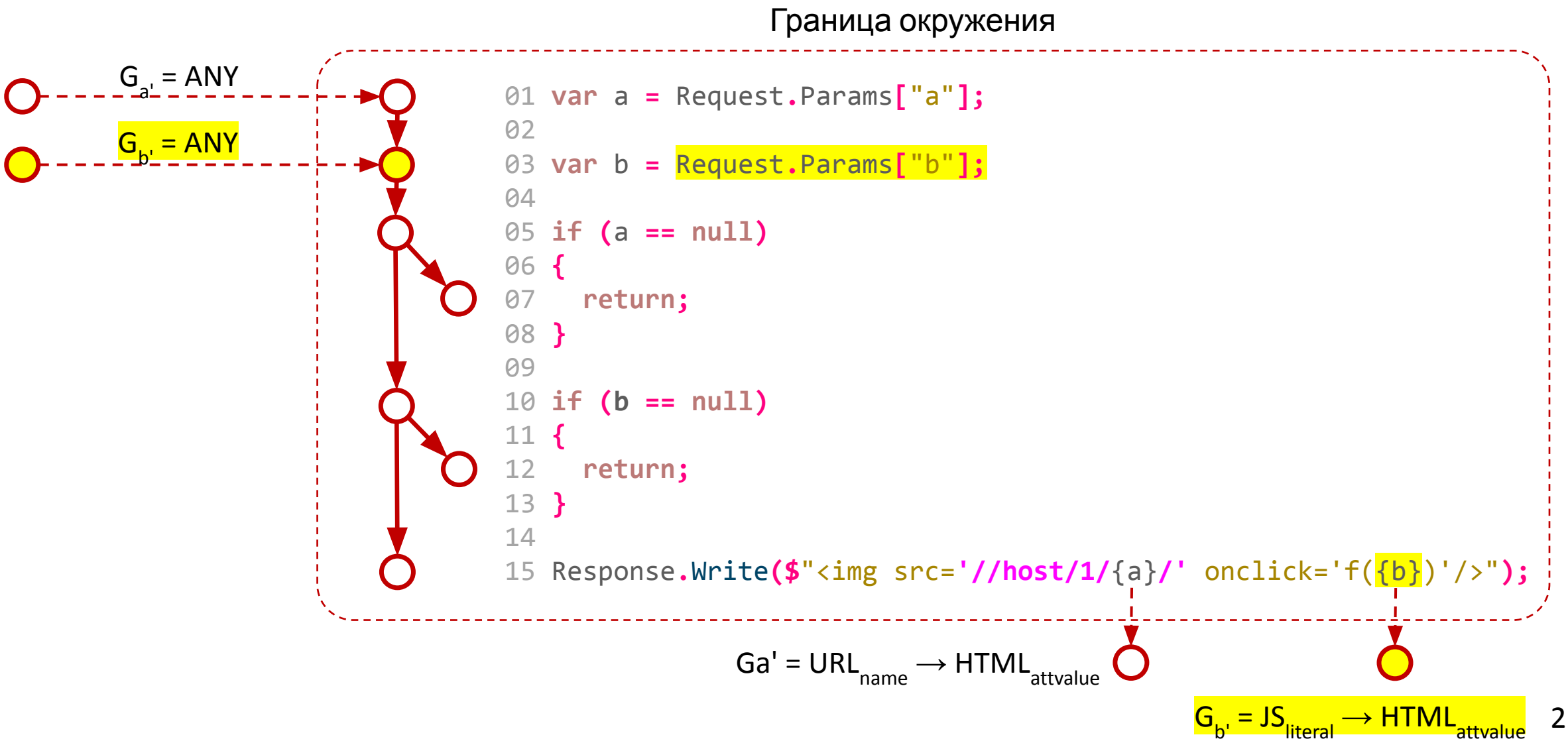
# Что может атакующий? (2/2)

```
... 0);... 0)'onload='... 0)'><script>...</script><!--
```

```
<img src='//host/1/{a}.jpg' onclick='f({b})'/>
```

# Причина инъекции – несогласованность грамматик (1/2)

# Причина инъекции – несогласованность грамматик (2/2)



# Инъекция возможна в любые нетривиальные грамматики

- Интерпретируемый код
- Структурированные данные
- Код разметки
- ID внешних ресурсов
- Любые грамматики, с числом токенов > 1

# Как защититься?

# Подходы к согласованию грамматик входных данных

Входные данные должны согласовываться с бизнес-логикой приложения за счёт:

- Типизации
- Валидации
  - Синтаксической
  - Семантической

# Типизация

**Типизация** – приведение строковых данных к конкретному типу:

```
var typed_url = Url.Parse(Request.Params["url"])
```

# Синтаксическая валидация

**Синтаксическая валидация** – проверка строковых данных на соответствие какой-либо грамматике:

```
01 var url_regex =
02   "^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?";
03
04 if (!Regex.IsMatch(Request.Params["url"], url_regex))
05 {
06   throw new ValidationException();
07 }
```

# Семантическая валидация

**Семантическая валидация** – проверка строковых данных на корректность с точки зрения логики приложения:

```
01 var request = WebRequest.Create(Request.Params["url"])
02  { Method = "HEAD" };
03
04 if (request.GetResponse().StatusCode != HttpStatusCode.OK)
05 {
06   throw new ValidationException();
07 }
```

# Подходы к согласованию грамматик выходных данных

Выходные данные должны согласовываться с грамматикой принимающей стороны за счёт санитизации

# Санитизация

**Санитизация** – преобразование строковых данных к синтаксису какого-либо токена заданной грамматики

```
01 var parm_text = Request.Params["parm"];
02
03 var parm_1 = HtmlEncode(UrlEncode(parm_text));
04 var parm_2 = HtmlEncode(parm_text);
05
06 Response.Write(
07   $"<a href='//host/a/b/{parm_1}'>{parm_2}</a>"
08 );
```

# Санитизация вложенных грамматик

В случае вложенных грамматик, например:

$G_t$ = `JavaScript` → `HTML`, t ∈ $D_e$

санитизацию необходимо проводить последовательно, в обратном порядке:

$\texttt{Encode}_{\texttt{Html}}(\texttt{Encode}_{\texttt{JavaScript}}(\texttt{t}))$,

с учётом синтаксических контекстов обоих грамматик.

# Точки согласования грамматик (1/2)

- Входные данные – как можно ближе к точке их входа, с учётом:
  - принципа необходимости и достаточности их грамматик;
  - приоритетности подходов:
    1. типизация;
    2. семантическая валидация;
    3. синтаксическая валидация.

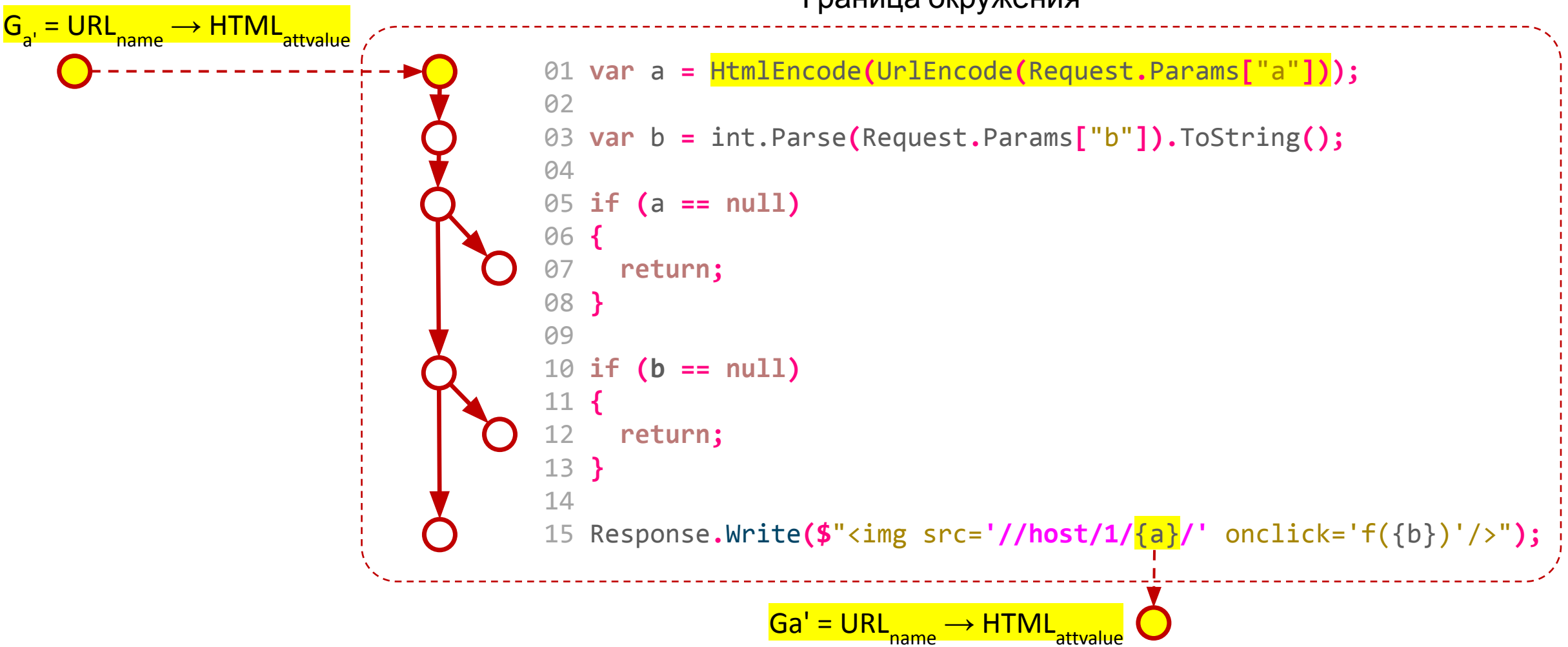
- Выходные данных – как можно ближе к точке их выхода, с учётом:
  - грамматики принимающей стороны;
  - возможной вариативности их грамматик в различных точках выполнения;
  - минимального (в идеале – нулевого) влияния согласования на прочие ветки потока вычисления.

# Точки согласования грамматик (2/2)

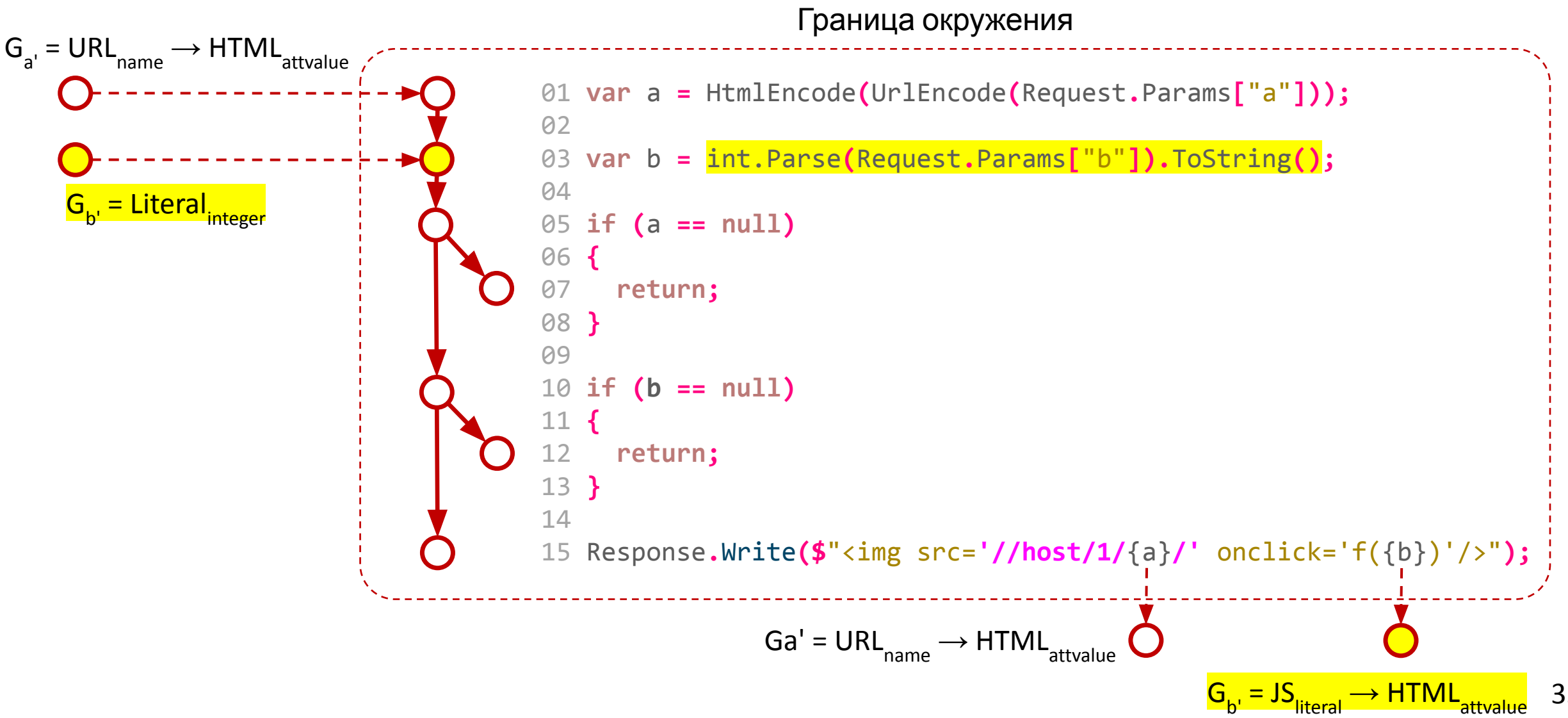
Из этих правил есть два исключения:

- Параметризация (типизация данных в точке выхода)
- Использование встраиваемых средств RASP (санитизация и валидация в точке выхода)

# Согласованные грамматики (1/2)



# Согласованные грамматики (1/2)



# Мечтать – не вредно (1/2)

А если бы существовала библиотека, принимающая на себя всю рутину по защите приложения от атак инъекций?

# Например, вот так? (2/2)

```
Response.Write(
  SafeFormat.Html(
    $"<img src='//host/1/{a}.jpg' onclick='f({b})'/>"
));
```

# Как протестировать?

# Подход на базе ошибок разбора

- Векторы от пентестеров и багхантеров – не использовать!

```
http://host/entry_point/?name=%2f..%2fWeb.Config
```

```
File.Delete($"\\temp_data\\{name}");
```

- Использовать набор векторов атаки, приводящих к ошибке парсинга:

"\..\filename?", “<filename>" и т.п.

- Каждый вектор – отдельный тест-кейс на отсутствие исключений при передаче модулю в качестве входных данных

# Подход на базе токенизации

- Используется любой набор векторов атаки и токенизатор соответствующей грамматики

- Перед каждой PVF устанавливается утверждение на количество токенов в её аргументе:

```
01 var pvfArgument = $"\\temp_data\\{name}";
02 Debug.Assert(Tokenize(pvfArgument).Count() == 4);
03 File.Delete(pvfArgument);
```

- Каждый вектор – тест-кейс на отсутствие нарушения утверждений.

# Мечтать – не вредно (2/2)

А если бы существовала библиотека, принимающая на себя всю рутину по тестированию защищённости от атак инъекций ?

# Например, вот так? (1/2)

```
01 var pvfArgument = $"\\temp_data\\{name}";
02 DebugFormat.Path(pvfArgument);
03 File.Delete(pvfArgument);
```

# LibProtection

# Что такое LibProtection? (1/4)

**LibProtection** – библиотека с открытым кодом (под MIT-лицензией), позволяющая разработчикам встраивать в приложение автоматизированную защиту от атак инъекций.

# Что такое LibProtection? (2/4)

Для защиты от атак в LibProtection реализованы:

- автоматическая санитизация входных данных там, где это возможно;
- автоматическая валидация входных данных относительно выходной грамматики (детектирование атаки) там, где невозможна санитизация.

# Что такое LibProtection? (3/4)

Детектирование осуществляется по формальным признакам: факт атаки не предполагается, а доказывается.

# Что такое LibProtection? (4/4)

Поддерживаемые языки:

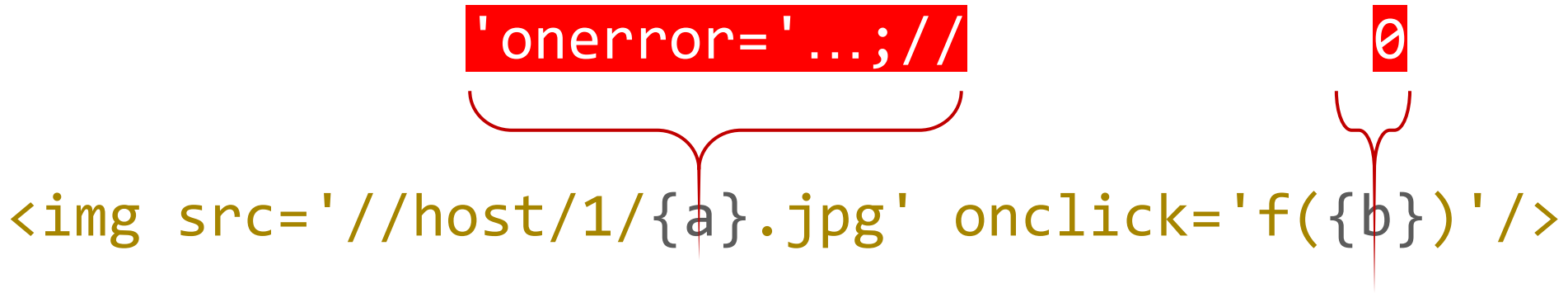
- .NET, C++ (Q4 2017)
- PHP, JVM (H1 2018)
- Python, Ruby (H2 2018)

Поддерживаемые грамматики: SQL (Microsoft SQL Server, Oracle, Database, Oracle MySQL), HTML, EcmaScript, CSS, URL, Path, XML

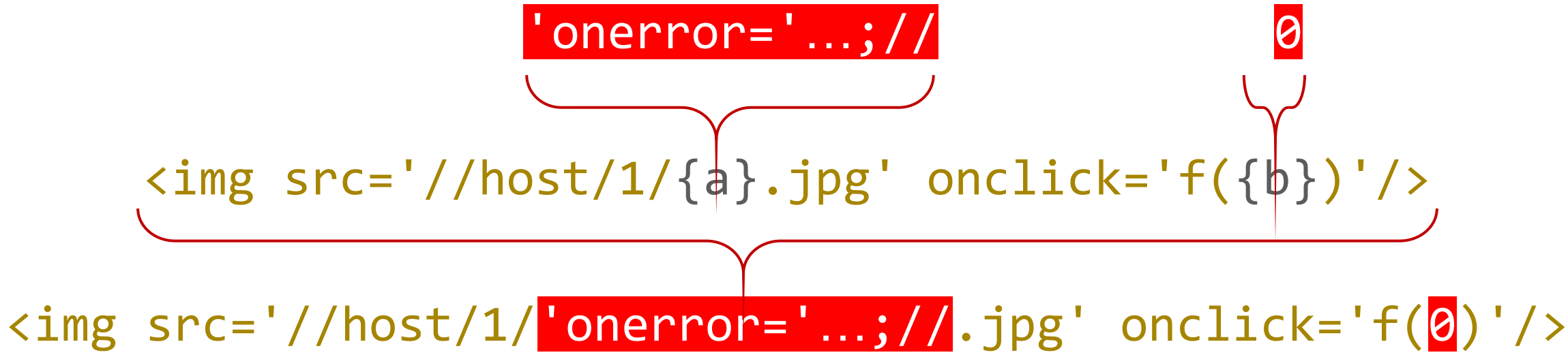
# Защита от инъекций с помощью LibProtection

```
01 Response.Write(
02   SafeFormat.Html(
03     $"<img src='//host/1/{a}.jpg' onclick='f({b})'/>"
04 ));
```

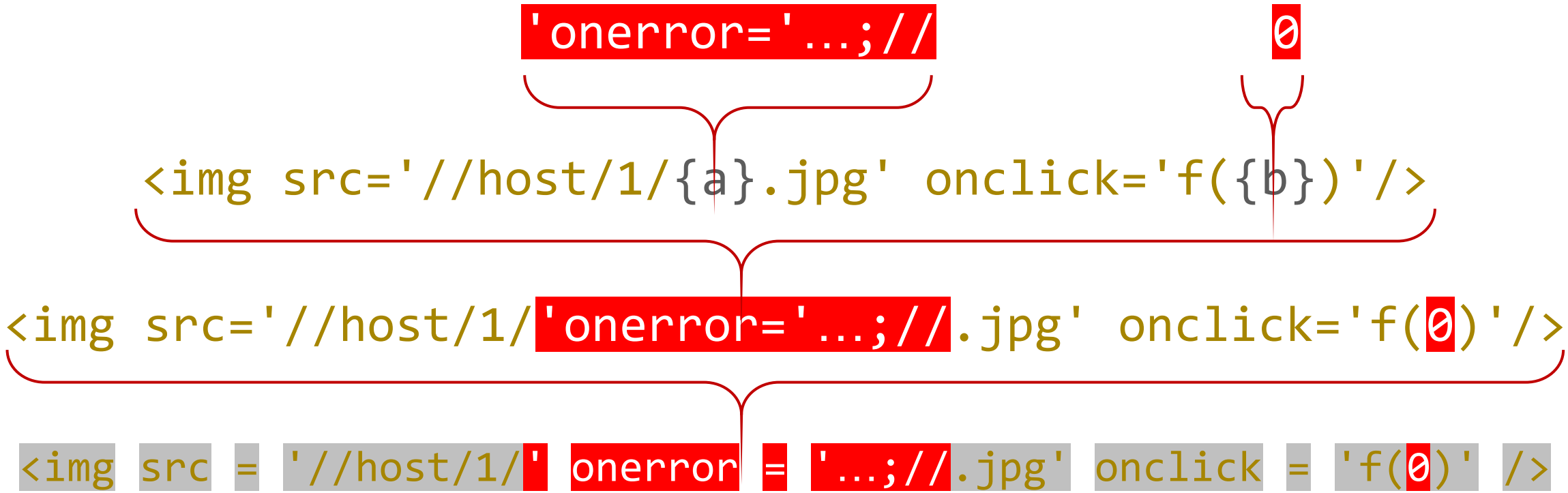
# Как это работает? (1/10)



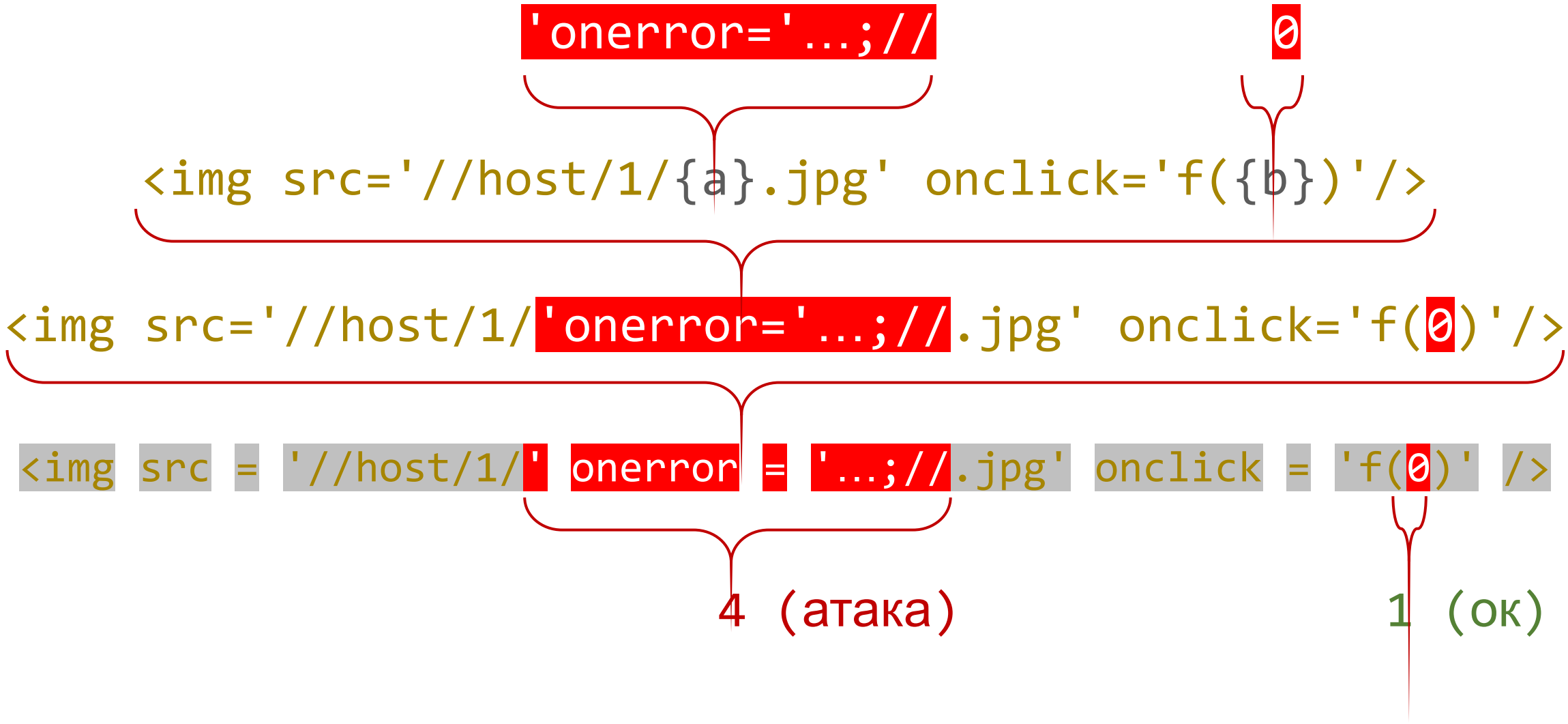
# Как это работает? (2/10)



# Как это работает? (3/10)



# Как это работает? (4/10)



# Как это работает? (5/10)

```
<img src = '//host/1/' onerror = '...;//.jpg' onclick = 'f(0)' />
```

4 (атака)

# Как это работает? (6/10)

```
<img src = '//host/1/' onerror = '...;//.jpg' onclick = 'f(0)' />
```

4 (атака)

$G_{a1}$ = URL/HTML

# Как это работает? (7/10)

```
<img src = '//host/1/' onerror = '...;//.jpg' onclick = 'f(0)' />
```

4 (атака)

$G_{a1}$ = URL/HTML

```
WU.HtmlEncode(WU.UrlEncode("'onerror='...;//"))
```

# Как это работает? (8/10)

```
<img src = '//host/1/' onerror = '...;//.jpg' onclick = 'f(0)' />
```

4 (атака)

$G_{a1}$ = URL/HTML

```
WU.HtmlEncode(WU.UrlEncode("'onerror='...;//"))
```

```
<img src='//host/1/%27onerror%3D%27%E2%80%A6%3B%2F%2F.jpg' onclick='f(0)'/>
```

# Как это работает? (9/10)

```
<img src = '//host/1/' onerror = '...;//.jpg' onclick = 'f(0)' />
```

4 (атака)

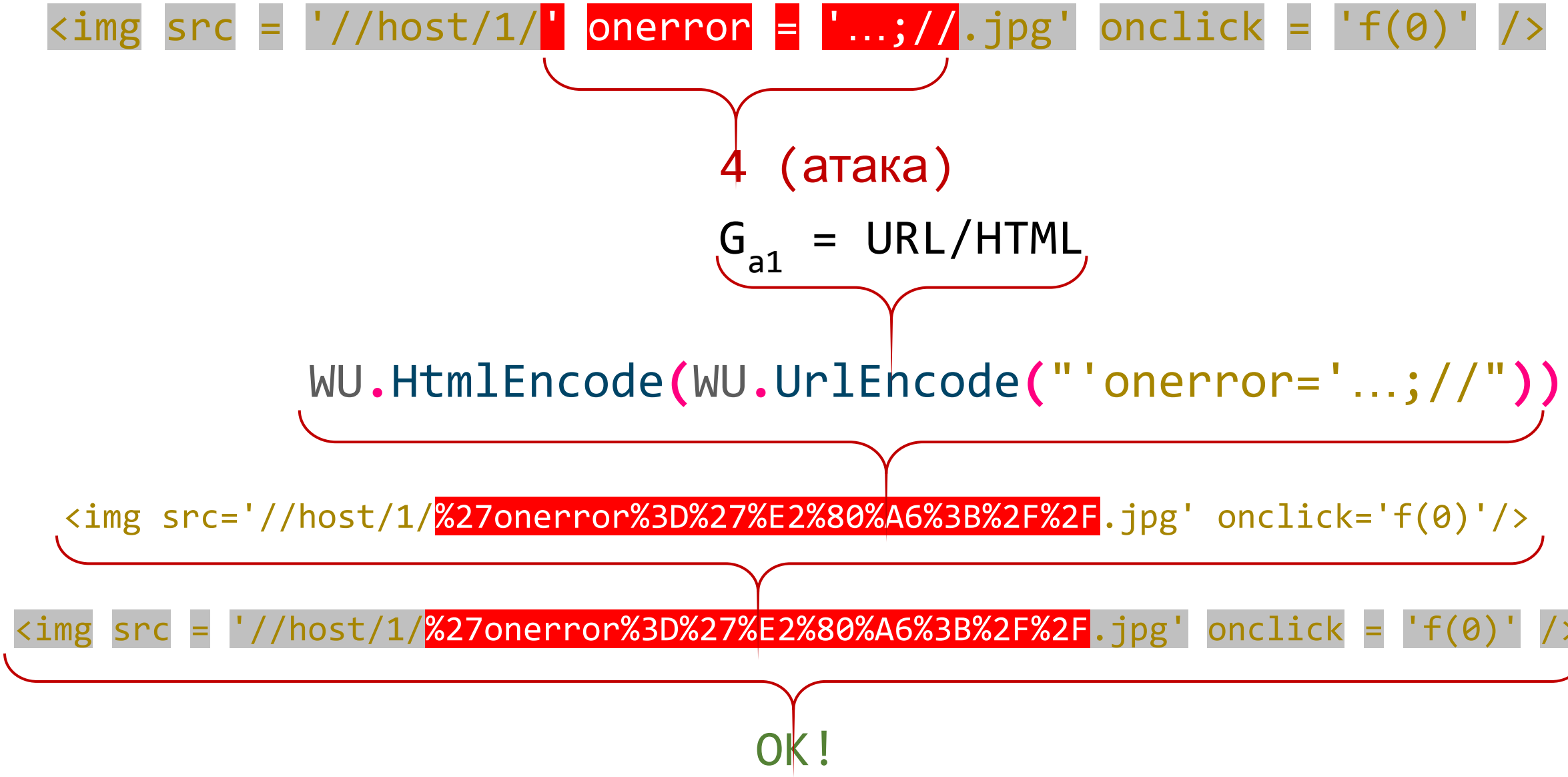
$G_{a1}$ = URL/HTML

```
WU.HtmlEncode(WU.UrlEncode("'onerror='...;//"))
```

```
<img src='//host/1/%27onerror%3D%27%E2%80%A6%3B%2F%2F.jpg' onclick='f(0)'/>
```

```
<img src = '//host/1/%27onerror%3D%27%E2%80%A6%3B%2F%2F.jpg' onclick = 'f(0)' />
```

# Как это работает? (10/10)



# LibProtection API: методы

```
// Grammar ∈ { Sql, Html, EcmaScript, Css, Url, Path, Xml }


string SafeFormat.Grammar(FormattableString formattable)
string SafeFormat.Grammar(string format, params object[] args)

bool TrySafeFormat.Grammar(FormattableString formattable, out formatted)
bool TrySafeFormat.Grammar(string format, out formatted, params object[] args)


string DebugFormat.Grammar(FormattableString formattable)
string DebugFormat.Grammar(string format, params object[] args)
```

# LibProtection API: форматные модификаторы

По умолчанию, LibProtection рассматривает все переменные, как опасные значения. Это поведение можно переопределить с помощью форматных модификаторов:

```
:safe

:validate(method-name)
    method-name ∈ Func<string,bool>

:sanitize(method-name)
    method-name ∈ Func<string, string>
```


# Спасибо!

PDUG

POSITIVE DEVELOPMENT USER GROUP