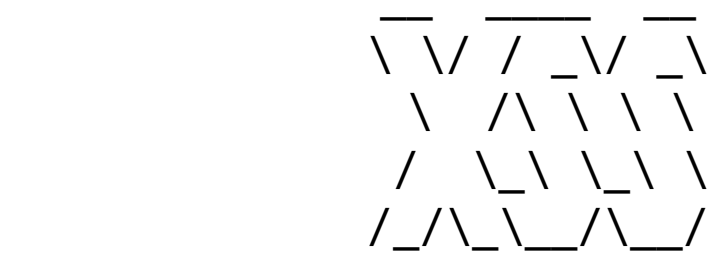


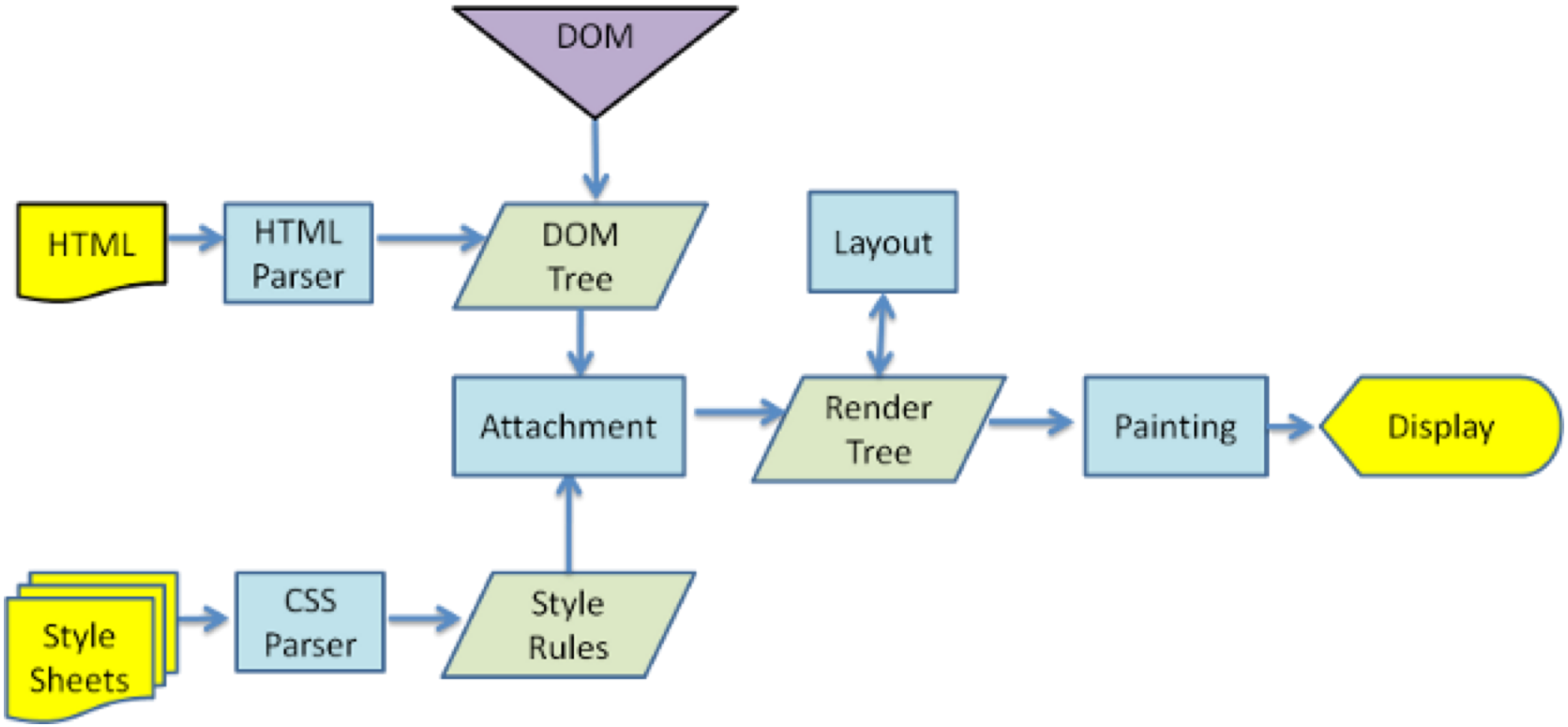
Description

- The classic code vs. data problem
- User input (data) is interpreted as code and executed
- Never trust user input!!
- Types:
  - Reflected
  - Stored/Persistent
  - DOM based

First a Step Back – HTML Rendering

- Overview
  - A browser sends a request to the server
  - The server responds and the response body contains HTML
  - The HTML is parsed by the browser and rendered and displayed on the screen
  - Note: we are intentionally breezing over CSS, JS, images, etc. for the time being

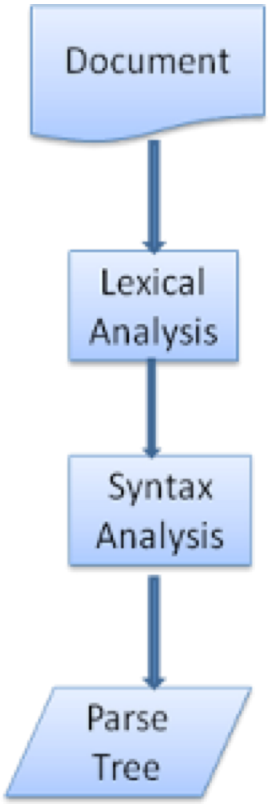
HTML Rendering



Webkit rendering flow

HTML Parsing

- Parsing consists of two functions
  - Lexical Analysis (Lexer)
    - Break input into tokens
  - Syntax Analysis (Parser)
    - Applying language syntax rules



Lexical Analysis

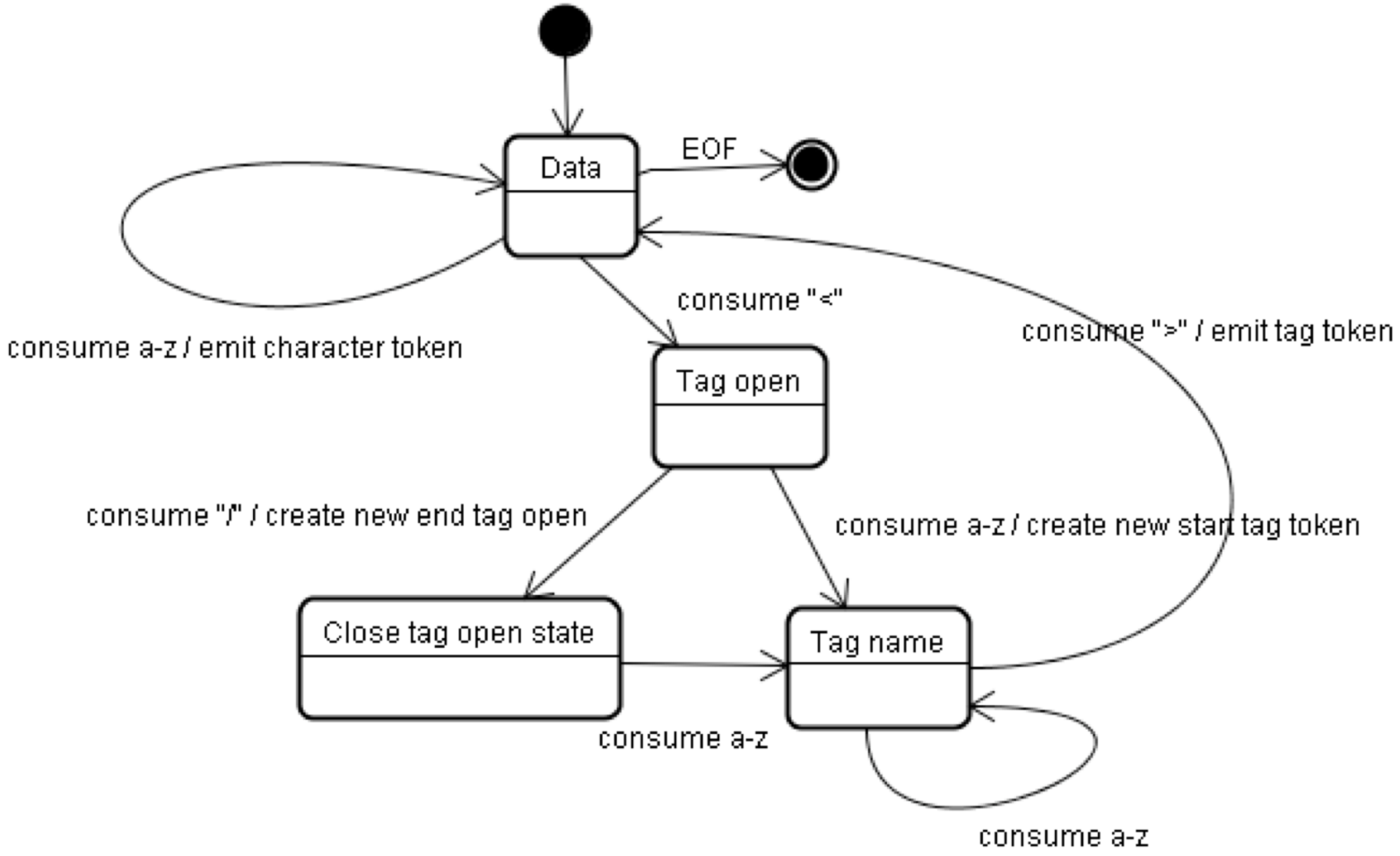
- Sometimes called “tokenization”
- The basic algorithm is a state machine
  - Each state leads to the next state
  - Each character affects the processing of the next character
  - The same character can be interpreted differently depending on the state of the state machine

Lexical Analysis - Example

- Given the following HTML

```
<html>
    <body>
        Hello world
    </body>
</html>
```

- The following Lexical Analysis occurs
  1. The initial state is the “data state”
  2. When the “<“ character is encountered the state changes to “tag open state”
  3. Consuming an “a-z” character causes creation of a “start tag token” and the state is changed to “tag name state”
  4. The “tag name state” occurs until the “>” character is hit. Each character before that is part of the tag’s name
  5. When the “>” is hit the state reverts to the “data state”
  6. “<body>” tag is treated the same
  7. “Hello world” will be processed as character tokens until the “<“ is hit
  8. “<“ will cause a “tag open state”
  9. The “/” will cause the creation of an “end tag token” and the state will enter “tag name state” until the “>” is reached, then state will become “data state”
  10. “</html>” is treated the same


Data
EOF
consume a-z / emit character token
consume "<"
consume ">" / emit tag token
Tag open
consume "/" / create new end tag open
consume a-z / create new start tag token
Close tag open state
Tag name
consume a-z
consume a-z

Syntax Analysis

- The Document Object Model (DOM) tree is constructed
- The Document is the root of the tree, elements are added
- Each node that comes out of the tokenizer is processed by the tree constructor
- Also a state machine

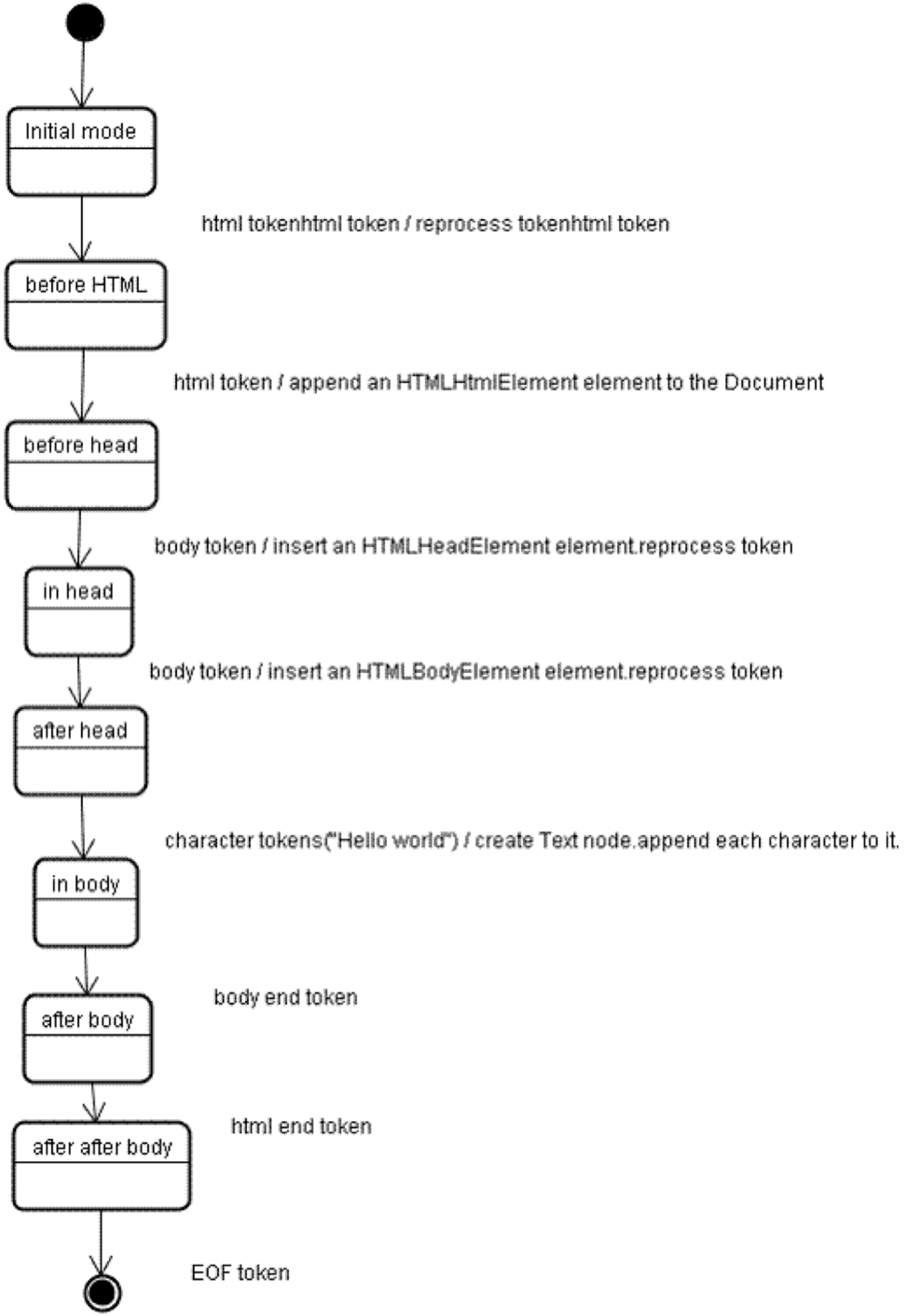
Syntax Analysis – Example

- Given the following HTML

```
<html>
    <body>
        Hello world
    </body>
</html>
```

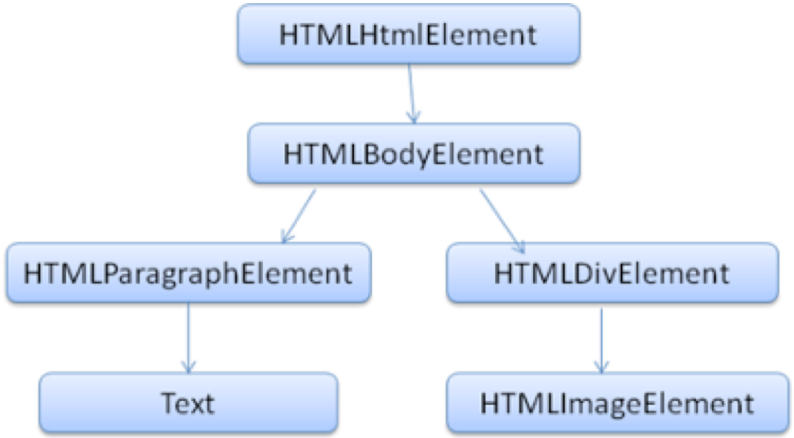
- Given the following tokens
  - html
  - body
  - “H” through “d” character tokens
  - body end
  - html end
  - end of file

- The following tree construction occurs
  1. First mode is the “initial mode”
  2. “html” token is received and leads to a “before html” state which results in a reprocessing of the token in that state
  3. An HTMLHtmlElement element is created and appended to the DOM tree and state changed to “before head”
  4. “body” token is received and an HTMLHeadElement is created and is added to the tree automatically (notice it handles the lack of expected “head” token). State changes to “in head” and then “after head”
  5. “body” token is reprocessed, HTMLBodyElement is created and inserted, state becomes “in body”
  6. “H” is received, Text node is created and other characters are appended
  7. “body end” token received and state changes to “after body” state
  8. “html end” token received and state changes to “after after body” state
  9. “end of file” token received and parsing is finished


Initial mode
html tokenhtml token / reprocess tokenhtml token
before HTML
html token / append an HTMLHtmlElement element to the Document
before head
body token / insert an HTMLHeadElement element.reprocess token
in head
body token / insert an HTMLBodyElement element.reprocess token
after head
character tokens("Hello world") / create Text node.append each character to it.
in body
body end token
after body
html end token
after after body
EOF token

HTML Parsing - Example

```
<html>
      <body>
            <p>
                  Hello World
            </p>
            <div> <img src="example.png"/></div>
      </body>
</html>
```



Injecting HTML

- Imagine a situation where the HTML is rendered dynamically server-side:

```
<?php
    <html>
        <body>
            Hello $_GET[“name”]
        </body>
    </html>
?>
```

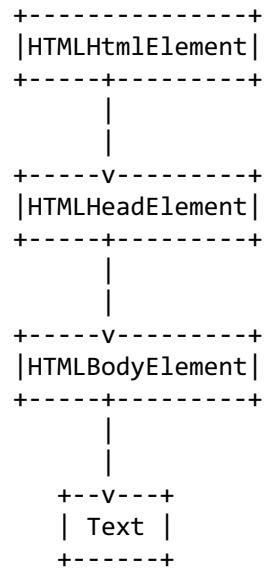
Injecting HTML

- If you go to http://example.com/index.php?name=bob the response is:

```
<html>
    <body>
        Hello bob
    </body>
</html>
```

- The resulting tokens looks like this:
  - html
  - body
  - “H” through “b” characters
  - body end
  - html end
  - end of file

- The resulting tree looks like this:



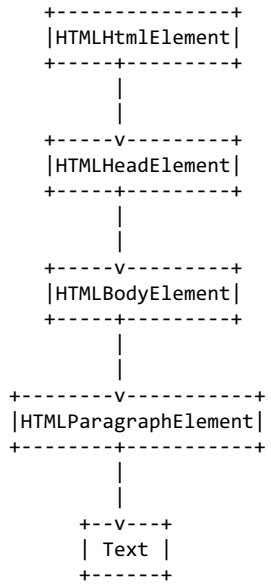
Injecting HTML

- If you go to http://example.com/index.php?name= steve the response is:

```
<html>
    <body>
        Hello <p>steve</p>
    </body>
</html>
```

- The resulting tokens looks like this:
  - html
  - body
  - paragraph
  - “s” through “e” characters
  - paragraph end
  - body end
  - html end
  - end of file

- The resulting tree looks like this:



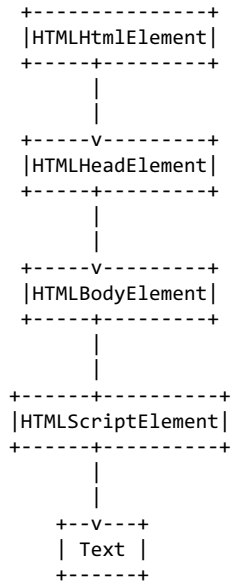
Injecting HTML

- If you go to http://example.com/index.php?name=<script>alert(0)</script> the response is:

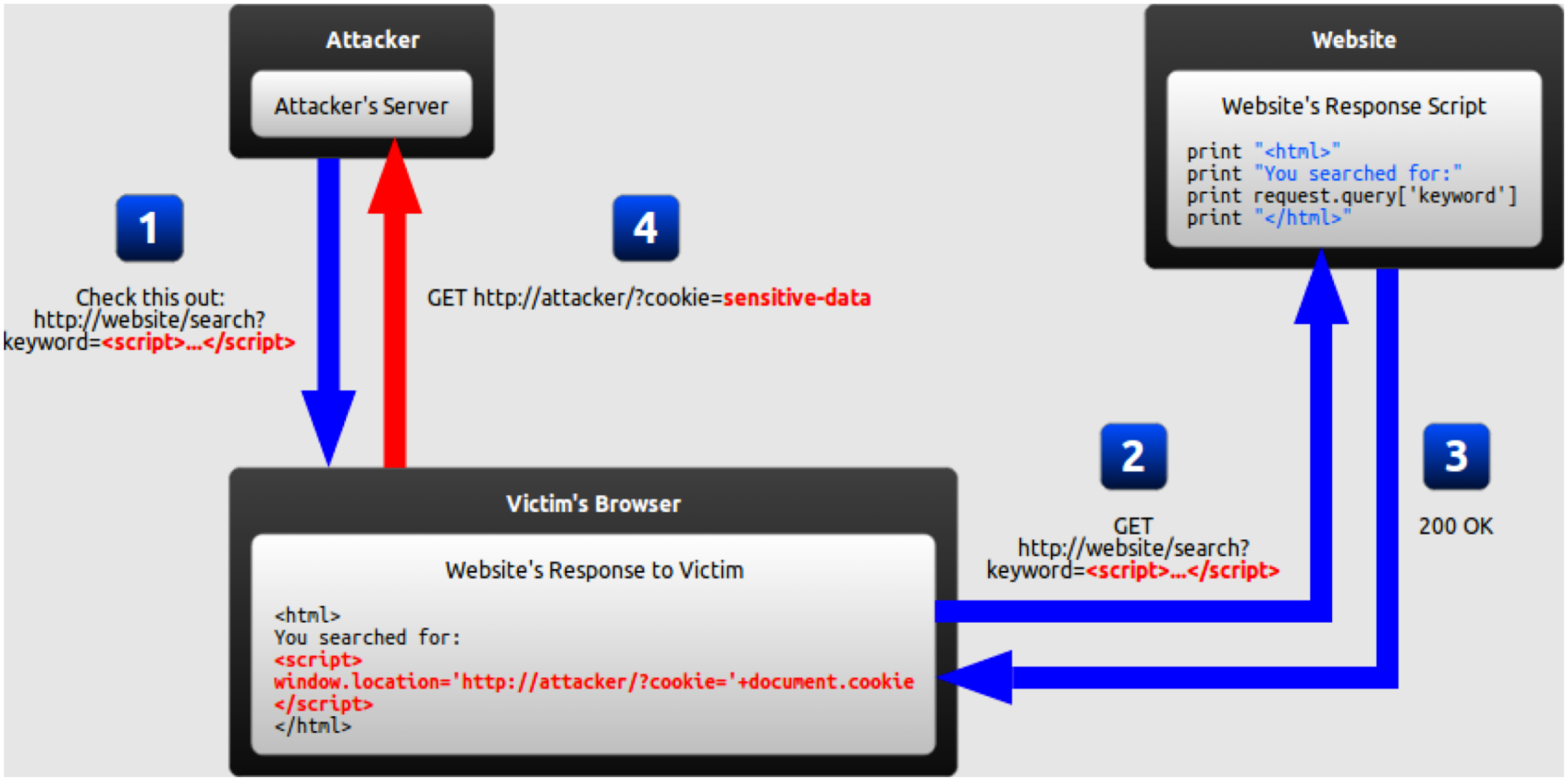
```
<html>
    <body>
        Hello <script>alert(0)</script>
    </body>
</html>
```

- The resulting tokens looks like this:
  - html
  - body
  - “H” through “o” character tokens
  - script
  - “a” through “)” character tokens
  - script end
  - body end
  - html end
  - end of file

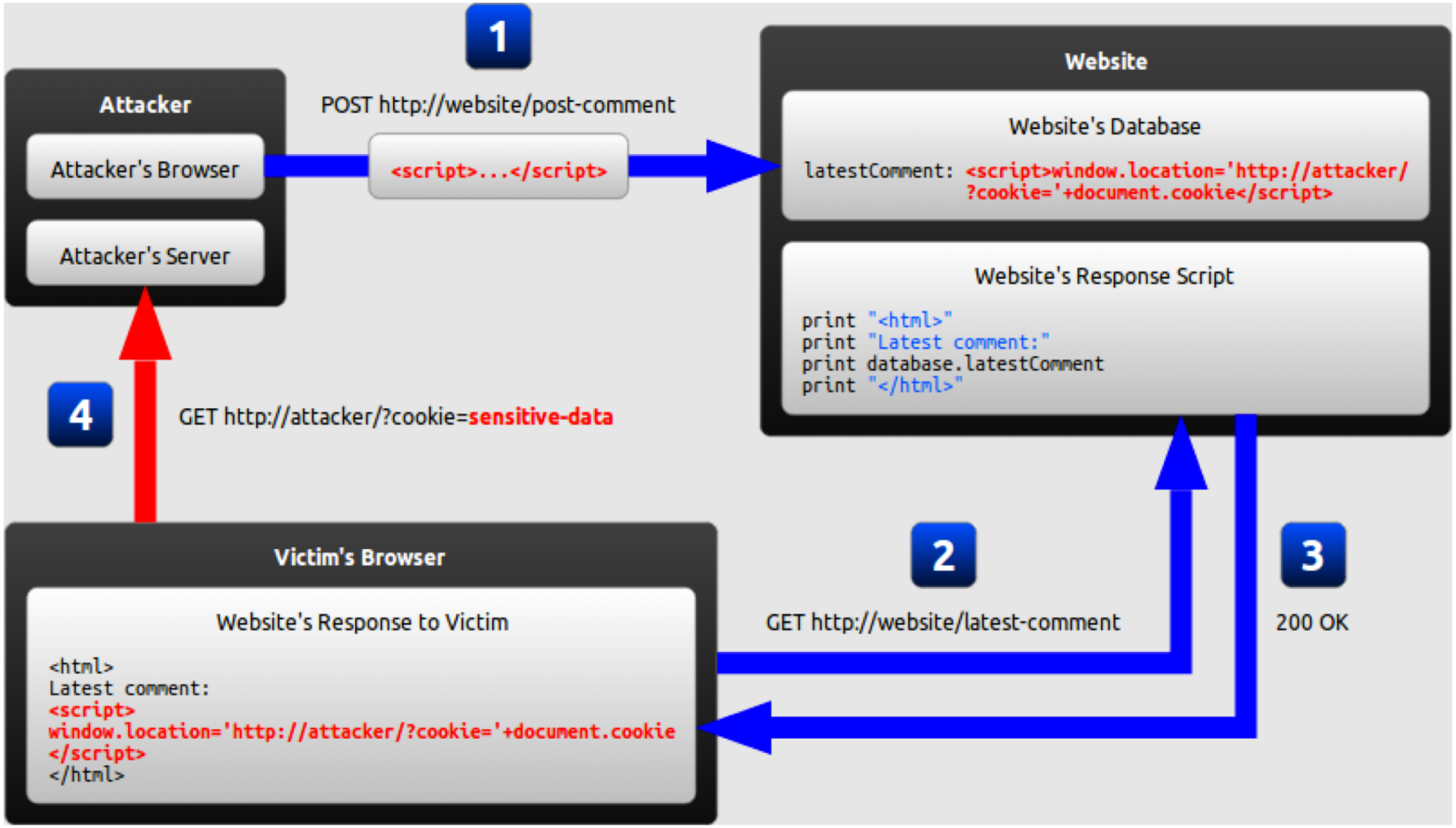
- The resulting tree looks like this:



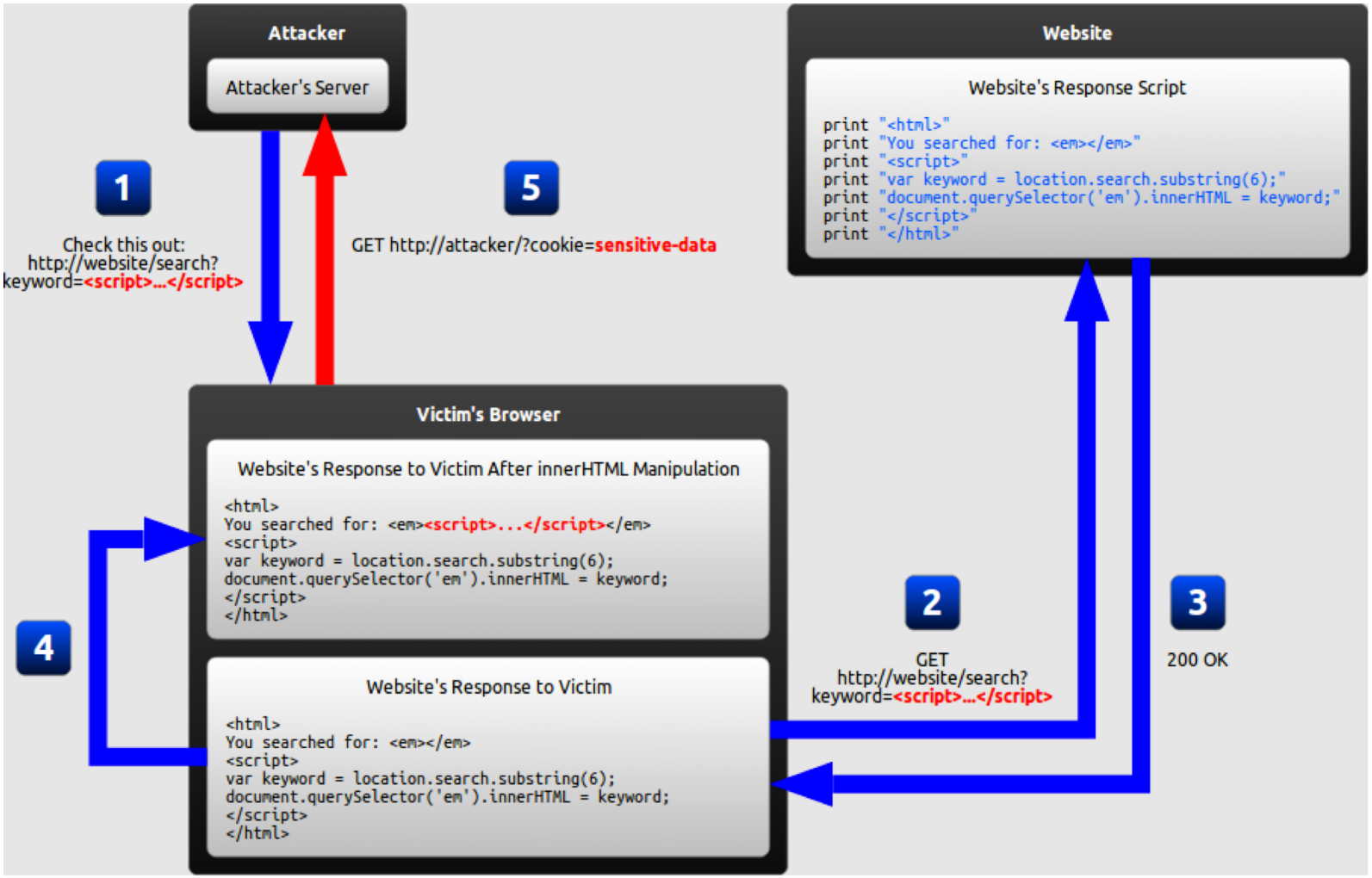
How it works (reflected)



How it works (stored/persistent)



How it works (DOM based)



demo

Why it's so common

- Most sites consume a lot of user input
- Web 2.0 means web *applications*
- Used to be extra work to sanitize everything
- Now it happens by default for newer frameworks
  - A lot of old code out there
  - Devs doing weird things
  - Functionality > security for most

Why it’s bad

- 5 major attacks
  - Stealing session tokens
  - Faking login form
  - Inducing user action
  - Keylogging
  - Client-side exploits

How to find it

- Place input in all the variables you can
  - Use something sufficiently unique (farm animals)
- See what is reflected back onto the page
- See what characters you can get through
- Figure out what you can do with the space you have and the characters you can get through
- Filter evasion

How to mitigate it

- HTML Encoding
  - < becomes <
  - > becomes >
  - “ becomes &quote;
  - ‘ becomes '
  - Things look okay in browser
  - This will neutralize the attack
- Browser mitigations (good job Chrome!)
- Validate input: regex [A-Za-z0-9+\s]+
- Use mature frameworks and don’t be “clever”

References

- Big thanks to: http://taligarsiel.com/Projects/howbrowserswork1.htm