JOE ROZNER | @JROZNER

WIPING OUT CSRF

IT'S 2017

WHAT IS CSRF?

WHEN AN ATTACKER FORCES A VICTIM TO EXECUTE UNWANTED OR UNINTENTIONAL HTTP REQUESTS

WHERE DOES CSRF COME FROM?

LET'S TALK HTTP

SAFE VS. UNSAFE

SAFE VS. UNSAFE

- Safe
  - GET
  - HEAD
  - OPTIONS
- Unsafe
  - PUT
  - POST
  - DELETE

LET'S TALK COOKIES

COOKIES

- Cookies typically used to specify session identifier for server
- Users depend on user agents to correctly control access to cookies
- User agents only but always send cookies with matching domain to hosts
  - This is done regardless of matching origin
- Cookies are user agent global (work cross tab)

SESSION=298zf09hf012fh2; Domain=example.com; Secure; HttpOnly

LET'S TALK XSS

XSS

- Attackers use XSS to inject CSRF payloads into the DOM
- With sufficient XSS all counter measures can be bypassed

HOW DOES CSRF WORK?

FORM BASED

- Normal HTML form that a victim is forced to submit
- Either genuinely supplied or attacker supplied (via XSS)
- Typically performed with JavaScript via XSS or attacker controlled page
- Good option for bypassing same origin without CORS
- Good option where safe verbs are used correctly
- Useful for phishing links as long as form click is fast

```html
<html>
  <head>
  </head>
  <body>
    <form action="http://bank.lol/transfer" method="POST">
      <input type="hidden" name="account" value="12345" />
      <input type="hidden" name="amount" value="100000000" />
      <input type="submit">Submit</input>
    </form>
  </body>
  <script>document.querySelector('form').submit();</script>
</html>
```

XHR

- Typically comes from an XSS payload
- Limited to the origin unless CORS is enabled
- No page reload required
- More difficult for victim to detect

```
<script>
  var xhr = new XMLHttpRequest();
  xhr.open('POST', '/transfer', true);
  xhr.onreadystatechange=function() {
    if (xhr.readyState === 4) {
      // request made
    }
  };
  xhr.send('account=12345&amount=1000000');
</script>
```

RESOURCE INCLUSION

- Doesn't depend on XSS or attacker supplied pages
- Requires an attacker to have control over resources on the page
- Depends on using safe verbs unsafely
- Limited to GET requests
- Possible with any HTML tags for remote resources
    - img, audio, video, object, etc.

```html
<img src="http://bank.lol/transfer?account=12345&amount=100000"></img>
```

CURRENT SOLUTIONS

- Using safe verbs correctly
- Verifying origin
- Synchronizer/crypto tokens

DISCLAIMER

WHY?

- Inability to modify the application
- Bulk support across many applications
- Providing protection to customers as a service

WHAT ARE WE LOOKING FOR?

- Easily added to apps without CSRF protections present
- Works across browsers
- Can work for xhr, forms, and resources
- Minimal performance impact (cpu, memory, network)
- No additional requests needed (updating tokens)

CORRECT SAFE VERB USE

- If you’re changing state don’t respond to safe methods
- Utilize mature and well designed frameworks and routers to help
- Be specific with your verbs and paths
- Not easy to fix after the fact but makes it much easier to solve
- If it’s not an option there are work arounds

VERIFYING REFERER/ ORIGIN

REFERER/ORIGIN OVERVIEW

- Check origin/referer in request against current address
- Not strictly required but adds some additional protection layers
- Probably what you want if you're dependent on CORS
- Possibly sufficient with safe methods used correctly
- Not fool proof because of header conditions
- For CORS read https://mixmax.com/blog/modern-csrf

```
GET /transfer HTTP/1.1

Host: bank.lol

User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_3)

Referer: http://shady.attacker/csrf-form.html
```

```
GET /transfer HTTP/1.1
Host: bank.lol
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_3)
X-Requested-With: XMLHttpRequest
Origin: shady.attacker:80
Referer: http://shady.attacker/csrf-form.html
```

```java
URL url = null;
String originHeader = request.getHeader("Origin");
if (originHeader != null) {
    url = new URL(originHeader);
} else {
    String refererHeader = request.getHeader("Referer");
    if (refererHeader != null) {
        url = new URL(refererHeader);
    }
}
String origin = url.getAuthority();
String host = request.getHeader("Host");
if (origin == null || origin.equalsIgnoreCase(host)) {
    return true;
}
return false;
```

TOKENS

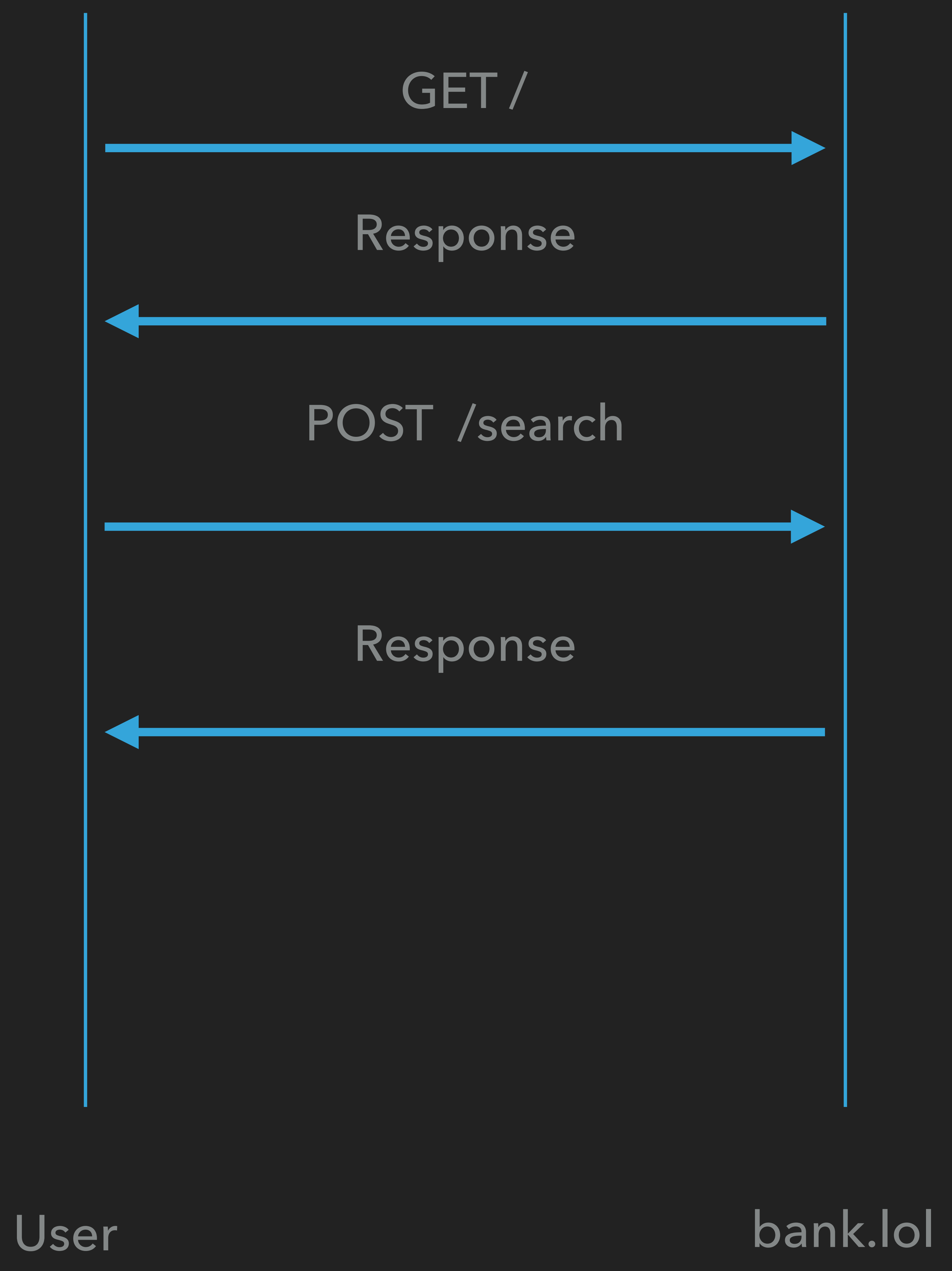
TOKEN OVERVIEW

- Come in two types: synchronizer and crypto
- Designed to make each request unique and tie it to a specific user and action
- Required for all state changing actions
- Traditionally only used for logged in users but can be used unauthenticated
- Additional benefits such as stopping replays

COMPOSITION

- Essentially composed of three components:
  - Random Value
  - User ID
  - Expiration
  - Authenticity Verification

- If one is missing security of tokens is severely compromised



```
GET / HTTP/1.1
Host: bank.lol

HTTP/1.1 200 OK
Set-Cookie: token=1234567890abcdef
...
POST /search
Host: bank.lol
Cookies: token=1234567890abcdef

HTTP/1.1 200 OK
Set-Cookie: token=123abc
```

CRYPTO VS SYNCHRONIZER

- Crypto requires no server side storage or deployment changes
- Synchronizer tokens are just random data
- Basically never use crypto tokens
- We're going to introduce a hybrid solution that provides the best of both worlds (mostly)

GENERATION

```
String generateToken(int userId, int key) {
  byte[16] data = random()
  expires = time() + 3600
  raw = hex(data) + "-" + userId + "-" + expires
  signature = hmac(sha256, raw, key)
  return raw + "-" + signature
}
```

VALIDATION

```
bool validateToken(token, user) {
  parts = token.split("-")
  str = parts[0] + "-" + parts[1] + "-" + parts[2]
  generated = hmac(sha256, str, key)
  if !constantCompare(generated, parts[3]) {
    return false
  }

  if parts[2] < time() {
    return false
  }

  if parts[1] != user {
    return false
  }

  return true
}
```

GIVING THE USER AGENT TOKENS

1. Intercept response on the way out after processing
2. If token is validated for request or doesn’t exist generate one
3. If generated create cookie and add to response

SENDING TOKENS BACK

FORMS

1. Attach an event listener to the document for “click” and delegate
2. Walk up the DOM to the form
3. Create new element and append to form
4. Return and allow browser to do it’s thing

```
var target = evt.target;

while (target !== null) {
  if (target.nodeName === 'A' || target.nodeName === 'INPUT' || target.nodeName === 'BUTTON') {
    break;
  }

  target = target.parentNode;
}

// We didn't find any of the delegates, bail out
if (target === null) {
  return;
}
```

```
// If it's an input element make sure it's of type submit
var type = target.getAttribute('type');
if (target.nodeName === 'INPUT' && (type === null || !type.match(/^submit$/i))) {
  return;
}

// Walk up the DOM to find the form
var form;
for (var node = target; node !== null; node = node.parentNode) {
  if (node.nodeName === 'FORM') {
    form = node;
    break;
  }
}

if (form === undefined) {
  return;
}
```

```
var token = form.querySelector('input[name="csrf_token"]');

var tokenValue = getCookieValue('CSRF-TOKEN');
if (token !== undefined && token !== null) {
  if (token.value !== tokenValue) {
    token.value = tokenValue;
  }

  return;
}

var newToken = document.createElement('input');
newToken.setAttribute('type', 'hidden');
newToken.setAttribute('name', 'csrf_token');
newToken.setAttribute('value', tokenValue);
form.appendChild(newToken);
```

XHR

1. Save reference to XMLHttpRequest.prototype.send
2. Overwrite XMLHttpRequest.prototype.send with new function
3. Retrieve and append token from cookie into request header
4. Call original send method

```javascript
XMLHttpRequest.prototype._send = XMLHttpRequest.prototype.send;
XMLHttpRequest.prototype.send = function() {
  if (!this.isRequestHeaderSet('X-Requested-With')) {
    this.setRequestHeader('X-Requested-With', 'XMLHttpRequest');
  }
  var tokenValue = getCookieValue('CSRF-TOKEN');
  if (tokenValue !== null) {
    this.setRequestHeader('X-CSRF-Header', tokenValue);
  }
  this._send.apply(this, arguments);
};
```

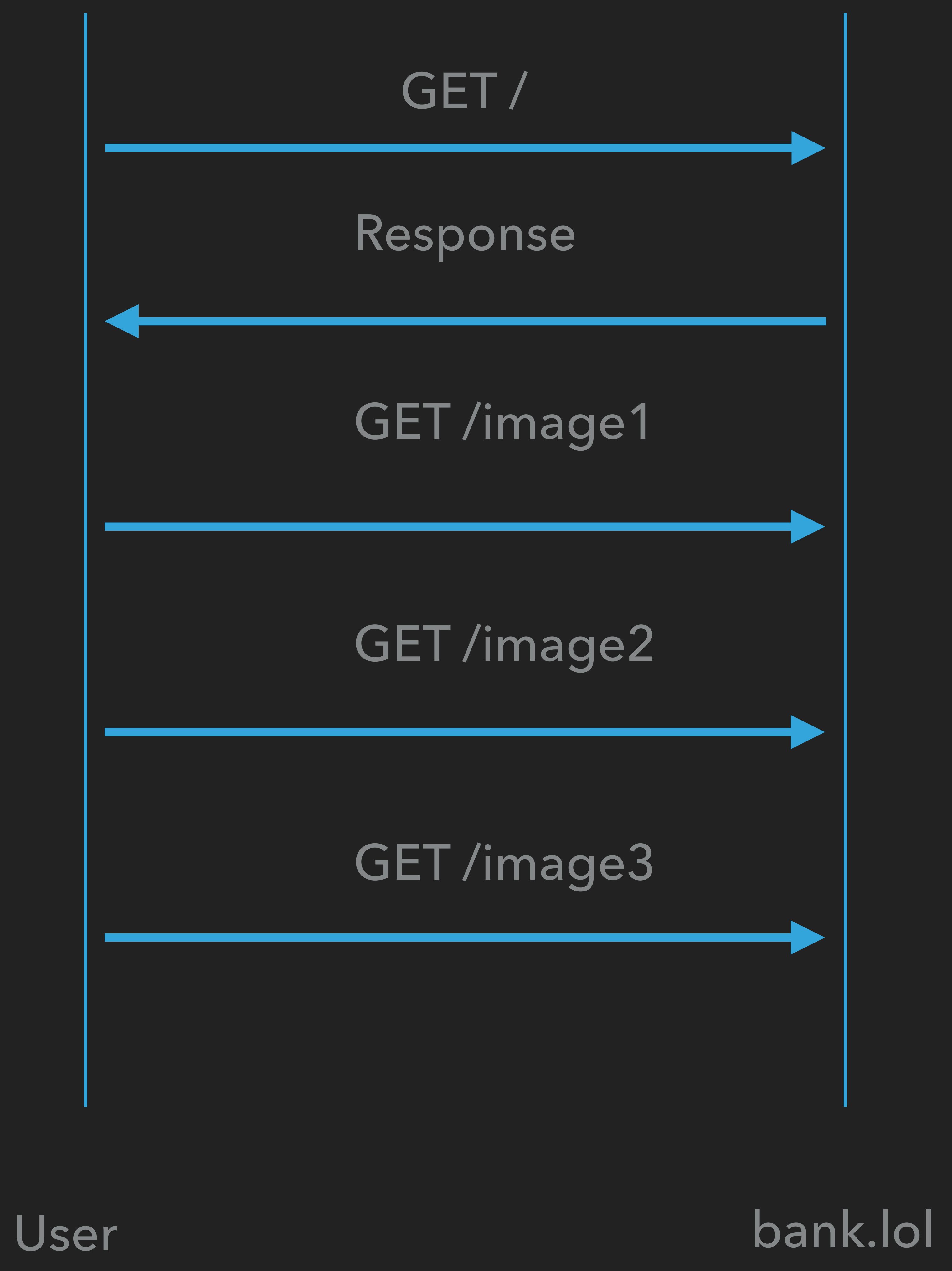
BROWSER SUPPORT

Browser	Supported
IE	8+
Edge	Yes
Firefox	6+
Chrome	Yes
Safari	4+
Opera	11.6+
iOS Safari	3+
Android	2.3+

THE FUTURE

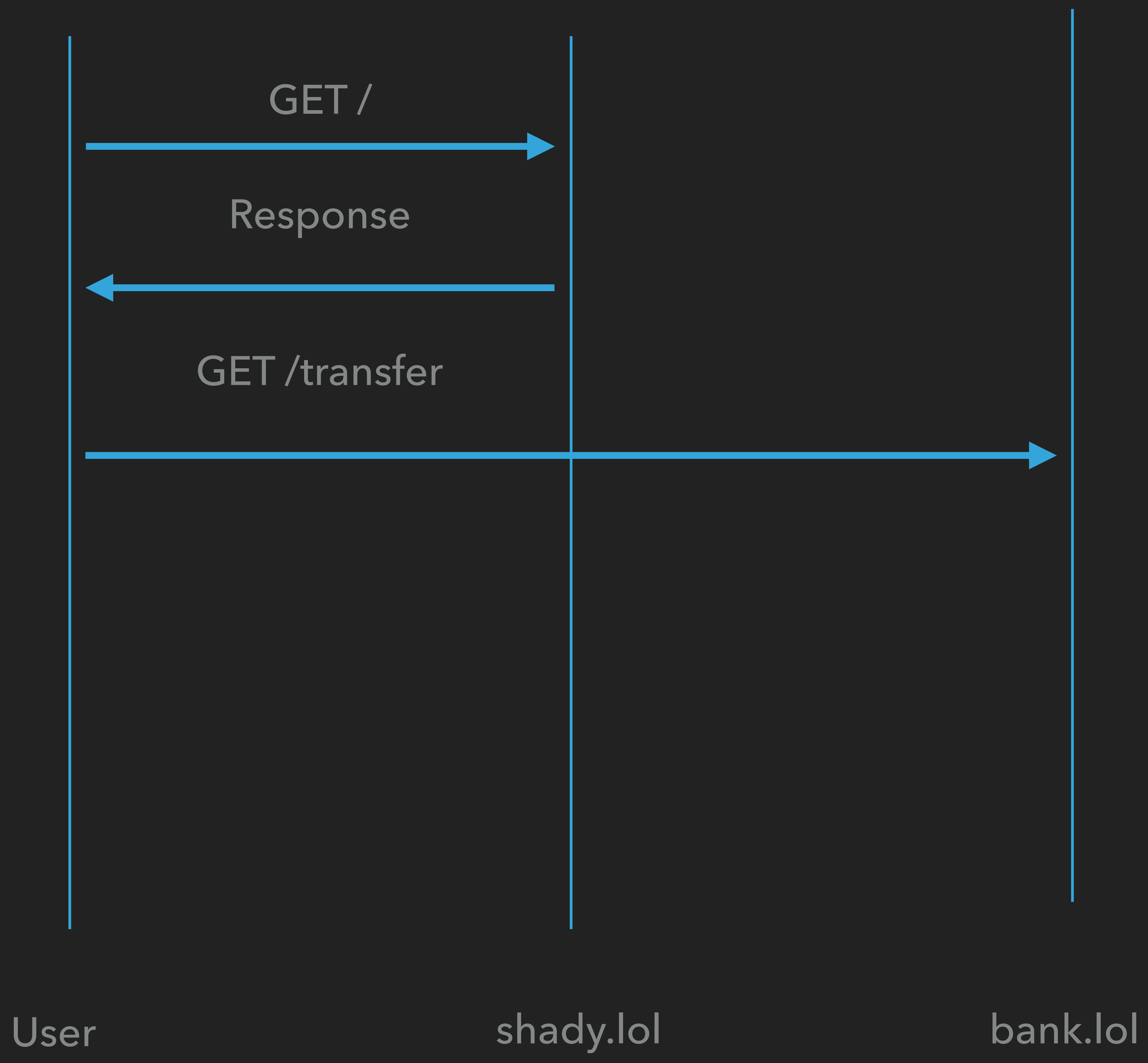
SAMESITE COOKIES

- Extension to browser cookies
- Largely replace the need for synchronizer tokens
- Correct use of safe methods is still important
- Fully client side implemented (no sever side component except cookie gen)
- Stops cookies from being sent with requests originating from a different origin



```
GET / HTTP/1.1
Host: bank.lol

HTTP/1.1 200 OK
Set-Cookie: session=1234567890abcdef
...
GET /image1
Host: bank.lol
Cookies: session=1234567890abcdef
...
GET /image2
Host: bank.lol
Cookies: session=1234567890abcdef
```



```
GET / HTTP/1.1
Host: shady.lol

HTTP/1.1 200 OK
...
GET /image1
Host: bank.lol
Cookies: session=1234567890abcdef
```

```
GET / HTTP/1.1

Host: bank.lol

...

HTTP/1.1 200 OK

Set-Cookie: session=1234567890abcdef; SameSite=Lax

...
```

STRICT VS. LAX

STRICT VS LAX

- Strict enforces for safe methods while Lax does not
- Strict can break for initial page load if cookies are expected present
- You can fix with some creative redirect magic
- Lax is probably sufficient in most cases

HTTPS://TOOLS.IETF.ORG/HTML/DRAFT-IETF-HTTPBIS-COOKIE-SAME-SITE-00

BROWSER SUPPORT

Browser	Supported
IE	X
Edge	X
Firefox	X
Chrome	55+
Safari	X
Opera	43+
iOS Safari	X
Android Chrome	56

http://caniuse.com/#feat=same-site-cookie-attribute

IMPLEMENTATION

1. Intercept responses on the way out
2. Parse Set-Cookie headers
3. Identify if cookie should have the SameSite attribute
4. Identify if SameSite attribute is present
5. Add SameSite attribute if not present

CONCLUSION

- We have current flexible solutions for CSRF that solve most cases
- These can be deployed retroactively to apps without support
- If you're building new apps use framework support
- We need to get users off old broken browsers
- SameSite looks like a possible end in many cases but too soon to tell