Web Security Checklist (v1.11)

1. Introduction

1.1. Purpose

This checklist is intended to help adopt new web frameworks (or help bootstrap new projects using a subset of our existing frameworks).

This checklist provides guidance to avoid common mistakes, and provides best practices to ensure we have a solid web security baseline. Going through this list might help standardize some APIs. If you want to assess if your web frameworks conforms to this checklist please consult with appsec@. Complete compliance with all of these guidelines is not required and documented exceptions are possible.

1.2. Why care about web security?

Web security problems can lead to undesirable consequences. A technically insignificant security bug (e.g., a XSS hole on a minor subdomain) can be misrepresented in the press, hurting our reputation. Other significant bugs, such as a privacy bug which gives access to arbitrary accounts, may not only damage our reputation but may also lead to financial losses. In the worst case, poorly written web applications can lead to remote code execution on internal services.

We should therefore strive to build web applications in the same way as we have built our backend services, specifically to the highest possible standard.

1.3. Resources

We currently have two wiki pages, which can be merged with this document:

- https://wiki.corp.squareup.com/display/SEC/Application+Security+Checklist
- https://wiki.corp.squareup.com/display/SEC/Webapp+security+requirements

Other good resources include:

- https://code.google.com/p/browsersec/wiki/Main
- https://www.owasp.org/index.php/Top10#OWASP_Top_10_for_2013
- http://www.nostarch.com/tangledweb
- https://www.cs.auckland.ac.nz/~pgut001/pubs/book.pdf
- http://www.cl.cam.ac.uk/~rja14/book.html

2. Documentation

2.1. Inform infosec about the use of a new framework + list of external libraries.
Infosec will then be able to track security vulnerabilities. (OWASP A9)

2.2. Maintain a document with best practices & common gotchas.

3. Configuration options (OWASP A5)

3.1. In production, disable rendering of error messages, exception, stack traces, etc.

3.2. Prevent accidentally serving configuration files or source code.
(If applicable) can be achieved by having the code live in a different folder than the web server's html/ folder.

3.3. Configure Keywhiz for secrets (passwords, API keys, etc.).

3.4. Ensure process is running in isolation
setuid, etc.

3.5. Ensure logs are sent over https to proper syslog service.

3.6. Disable multiple SQL statements.
Makes exploiting some forms of SQLi much harder.

4. Cookies & Session handling (OWASP A2)

4.1. By default, cookies should have the httponly flag.
This will ensure the cookies are unreadable by javascript. It is almost never ok to disable the flag on specific cookies.

Authentication cookies must always have the httponly flag to prevent session theft in the presence of XSS bugs.

4.2. All cookies should have the secure flag.
This forces the cookies to only get sent when communicating over https.

The only valid exception to this rule would be a "hsts" cookie for older browsers. No other cookie should be exempt from having the secure flag.

4.3. User session handling

4.3.1. For subdomains of squareup.com: https://docs.google.com/a/squareup.com/document/d/1IsqJzm-Qmn0E7DKybTVsU0Kbsz8SrEVLLvpD5sw6pF8/

4.4. Do not CNAME any subdomain to third party servers. This would leak our users' cookies to a third party. Using HTTP layer redirects is safe.

Note: in retrospect, it's fine to CNAME subdomains as long as we don't share/issue TLS certs for the subdomain.

5. HTTP Headers

5.1. By default, responses must contain "X-Frame-Options: DENY".
Frame Options restricts whether the content can be in a frame or iframe of another page. This should be disabled to prevent clickjacking. If you must, relax the setting to SAMEORIGIN.

Another attack this prevents is loading the page with "view-source:" and stealing users' tokens by tricking them into copy-pasting the content. When opting out of this header, the page should therefore not contain any CSRF tokens. Web widgets are usually the only valid case.

Optional: turning off the X-Frame-Options should disable setting the CSRF token on the page.

5.2. Responses must contain "Strict-Transport-Security: max-age=2592000; includeSubDomains".

HSTS tells browsers to only use HTTPS. It defends against things like sslstrip. The number is how many seconds the setting is good for (2592000 = 30 days).

You must append "includeSubDomains" to protect against a mitm csrf attack.

Specific pages may not opt-out of this header, the header must be sent in every response.

5.3. Responses must contain "Content-Type: …; charset=utf-8".
Failing to provide the right content type can lead to attacks where the browser is tricked to re-interpret the page as a malicious Flash, Java or PDF file.

Failing to provide the charset can lead to subtle xss exploits, where an attacker encodes the xss as utf-7 (+ADw- is '<') and tricks the browser into running

javascript.

Common content types:
for html: “text/html”
for json: “application/json”
for jsonp: “text/javascript”

5.4. Responses must contain “X-Content-Type-Options: nosniff”
Short version: Because Internet Explorer.

Long version: In some cases, Internet Explorer could be tricked to ignore the Content-Type header. Instead of fixing the core bug, Microsoft decided to add a header (they didn’t want to break existing websites). A decade later, people discovered two ways to bypass X-Content-Type-Options. This time, Microsoft fixed the core issue instead of adding X-Content-Type-Options-For-Reals.

Optional: it’s ok to only send X-Content-Type-Options to Internet Explorer users only. Be careful with IE11, where MSIE was dropped in the user agent string.

5.5. Responses must contain “X-XSS-Protection: 1; mode=block”

Tells browsers to detect a subset of reflected XSS bugs. In the past, this header has caused other security issues in correct pages, which is why some sites (e.g. www.facebook.com) don’t set this header.

5.6. Optional: enable Content-Security-Policy
(and ~~X-Content-Security-Policy~~, ~~X-WebKit-CSP~~)

CSP lets you tell the browser where content in the page can come from. For example, if only scripts from squareup.com are allowed, the browser will not execute javascript injected into a badly written login form (since it did not come from squareup.com).

The simplest policy is:
Content-Security-Policy: default-src 'self'; report-uri /csp-report

DO NOT use *, 'unsafe-eval', 'unsafe-inline', or javascript:.
DO use 'self' and set a report-uri. Please add a code comment to discuss further policy changes with infosec@.

While working out kinks, instead use the header
Content-Security-Policy-Report-Only to get messages about CSP violations without affecting behavior. Browser plugins might generate lots of useless

messages.

Keep in mind that Safari is buggy. Internet Explorer has a partial (and completely useless) implementation of X-Content-Security-Policy.

CSP when configured to disallow inline scripts does not allow event handlers.

CSP supports a nonce mode, where inline scripts are supported only if they carry a random value. This can provide a good tradeoff between no security and something usable. Some frameworks however require the ability the eval() which defeats the point of CSP.

Ideal CSP policy:
`Content-Security-Policy: default-src 'self' cdn1.squareup.com cdn2.cloudfront.com ...other cdns are OK...`
`X-Content-Security-Policy: default-src 'self' cdn1.squareup.com cdn2.cloudfront.com ...other cdns are OK...`

5.7. Configure meta tags
<meta name="referrer" content="origin"> or "never".
<meta name="robots" content="noindex"> if the page contains sensitive information and is public (e.g. receipts). Default should be decided on case by case basis. (Setting noindex should probably also imply sending the right Cache-Control header, so that the browser doesn't cache any information?)

5.8. CORS headers
If you are sending "Access-Control-Allow-Origin: *" to provide a read-only access to data, you should disable setting the CSRF token on the page (and/or use a subdomain). (ask Alok about pgp.mit.edu flaw).

If you are providing a cross origin read/write access, it makes even more sense to use a subdomain. The goal is to keep the surface area for potential bugs to a minimum).

5.9. Deny requests with "Service-Worker: script"
See Erling's wonderful bug report:
https://code.google.com/p/chromium/issues/detail?id=439730

6. Preventing code injection (OWASP A1)

6.1. Database access / SQL

The framework must provide some level of parameterized query construction. This can be either an ORM style (e.g. RoR) or printf/prepared statements (e.g. vanilla PHP).

6.2. Html / templating (OWASP A3)

The framework must provide a way to emit HTML not prone to XSS bugs. Variables should be escaped by default. The framework should know about attributes which can lead to javascript execution (onLoad, onClick, etc.).

The need to manually mark a variable as html or plain text should be avoided.

Ideally, the framework should have enough context to:

- correctly escape tags vs attributes
- tell which attributes lead to javascript execution (onLoad, onClick, etc.)
- avoid meta tag open redirects
- avoid javascript:// in href tags
- avoid javascript in css context
- avoid javascript in svg tags
- etc.

The framework should be able to compose HTML from snippets of HTML:
var1 = render("some_template", {some variables})
var2 = render("other_template", {... var1 …})
Should not lead to double escaping issues. The API should not require var1 to be explicitly flagged as html, safe or unsafe.

6.3. Header key/values should be sanitized

Parsing http headers correctly can be tricky. The minimal requirement should be handling newlines. It's best to avoid user controlled values in there.

6.4. Command (shell) injection

In general, it's better to build a specific backend service instead of running a shell command.

(If applicable) The framework should provide a way to safely escape command parameters.

An alternative to calling a backend service is to extend the framework and expose the shell command as a native function. The worst case should be to use the 2 parameter version of system (as opposed to the single parameter one): system('cmd', 'params').

6.5. File / path injection

6.6. Self-xss
Some users can be tricked to run code in the developer console/URL bar. I successfully convinced browser vendors to drop “javascript:” when copy-pasting things in the URL bar. For the developer console, you can display a “STOP” sign (using colorful ascii art) and mitigate self-xss to some degree.
TODO: figure out if browsers ever fixed this at the CSP level.

6.7. CSV injection
Some web applications expose data export functionality. If the data is being exported as CSV, the code must escape “,”, as well as “=” which can be used to execute commands in excel (e.g. foo,bar,=A1+A2).
See also https://hackerone.com/reports/72785

7. Forms and CSRF (OWASP A8)

7.1. Provide logged-out CSRF protection
http://homakov.blogspot.com/2014/01/two-severe-wontfix-vulnerabilities-in.html

7.2. Provide Logged-in CSRF protection
The protection should guard against mitm over https. I.e. the CSRF token needs to be tied to the user’s session.

7.3. Token must be different on each page load.
Protects against analysis of compression ratio over encrypted streams. (see BEAST, CRIME, BREACH class of attacks). Also protects against things like http://victim.website/cssp/, where a CSS injection leads to token theft.
For an example on how rails 4.2.0 and above handle this, see:
https://github.com/rails/rails/pull/16570/files

7.4. Token must to be automatically validated on every POST request, including AJAX requests.

One way to handle the Ajax case is to send the token in a custom header, e.g. “X-CSRF-TOKEN”. Keep in mind that checking the presence of the header is not enough, the actual value needs to be validated.

7.5. Token can be available to Javascript (for Ajax requests)

The ideal way to pass the token to javascript is to use a <meta> tag, e.g. <meta name=”csrf_token” value=”...”>

Putting the CSRF token in a cookie for javascript to read it is not recommended. Firefox's data URIs can steal such cookies.

7.6. Prevent state mutation on GET requests.
A possible solution is to disallow write connection to databases on GET requests and also disallow sending POST requests to backend services on GET requests. Handling state mutation to caches can be a little more tricky but we can probably live by punting on that issue.

State mutation on GET requests can be bad for three reasons: bypass of CSRF protection, request can be accidentally triggered by a pre-fetch, request can be accidentally dropped if the data is cached.

7.7. Optional: per action token.

7.8. Optional: HMAC or encryption of hidden parameters.

7.9. Validate origin header.

7.10. Provide a way to validate input.
Input validation must be performed on the server-side. Client-side input validation looks nice but cannot be trusted.

7.11. For logged out flows (common with email-based features)
- 7.11.1. Ensure sensitive URLs cannot be guessed or modified (use a token or a hmac).
- 7.11.2. Copy the token from the URL into a cookie and redirect to the same URL with the token stripped out. (to prevent referer leak).
- 7.11.3. Depending on the complexity of the logged out flow, a special csrf token might be required.

7.12. Use the Origin header
The Origin header can be used as a second layer of CSRF protection.

Currently, Chrome is the only browser to send the Origin header for XHR + form posts. The Origin header check can be enforced in Chrome and used in logging mode for other browsers.

8. URL parsing and handling

8.1. URL should always be manipulated as an object, instead of a string.

8.2. TODO: define a reference API

8.3. Prefer //[url] over specific schemes
It avoids having to care about http vs https. Also avoids the risk with javascript://.

8.4. It should be possible to express that a URL is expected to be internal vs external.

8.5. It should be possible to express that a URL is generated from a literal string or originated from user data.

8.6. Redirects should not accept a user supplied URL. (OWASP A10)
To prevent open redirects.

8.7. external links should flow through a shim.
This protects against token leaks in Referrer.

The list of protocols should be whitelisted. Things like data URIs should be forbidden (because Firefox + data uri + cookie theft, seehttps://bugzilla.mozilla.org/show_bug.cgi?id=255107, http://blog.kotowicz.net/2011/10/sad-state-of-dom-security-or-how-we-all.html).

8.8. External images must be proxied.
Ensure that all images are cached / hosted by us protects against a bunch of issues: dead images, http authentication injection, mixed content warning, slow page rendering, referrer leak, etc. Also plays nicer with CSP.

9. JSON/JSONP

In general, avoid JSONP and use CORS.

9.1. API JSON Response must start with /**/
Prevents various attacks where the first bytes of the response is used to trick the browser into running Flash code, tabular data, or other file types.

Remember, “Who controls the first bytes controls the SOP”.

9.2. (If applicable) callback parameter should be limited in character set & length: /^[a-zA-Z0-9]{0,32}$/

9.3. non-API JSON responses need an infinite loop (“for(;;);“).
Prevents a bug that was discovered in 2006. Probably no longer relevant in modern browsers, but doesn’t hurt to have it. (See JavaScript Hijacking, also Jeremiah Grossman’s work). Only the Same Origin can strip out the infinite loop.

In general, don't create dynamic javascript endpoints. It breaks our ability to enable CSP and it's a data exfiltration risk.

9.4. When setting CORS headers, disallow setting "Cache-Control: public". The combination can lead to a SOP bypass depending on what checks are happening server-side.

10. Javascript / client-side code

10.1. In general javascript should be served as standalone files
Inline <script> tags break CSP.

Data from the server-side to javascript is best passed using <meta> (or any other invisible tag) and JSON encoded data:
<meta name="for_javascript" value="{...json blob...}">

10.2. Limit the use of dynamically evaluated code. Make sure the input cannot be controlled by the user.

This includes eval(), new Function(...), jQuery's $(...), setTimeout(...), setInterval(...), etc.

10.3. Do not include scripts from other domains.
It is more reliable for us to host our own code. It also prevents having to trust third party domains from injecting malicious javascript in our page.

10.4. Do not include script sources over HTTP.
All data should be sent over https, including scripts.

11. Privacy / Access Control (OWASP A4, A6, A7)

11.1. Provide automatic privacy controls for the common case.
Provide a reasonable way to avoid trivial privacy bugs. By default, creating data should tie it to an owner, and reading, updating or deleting data should happen only after a valid ownership check.

Privacy is obviously more complicated than this; data gets shared across multiple entities, specific data mutation can happen conditionally in the future, etc.

11.2. Wrap sensitive strings in a class.
This avoids sensitive information leaks in logs, stack traces, etc.

11.3. Admin access should flow via Doorman
Doorman ensures that admin features are restricted by IP address (requires VPN) and also provides two factor authentication.

12. Cryptography

TODO: This will probably get its own doc.

13. Account takeover

The three things to keep in mind when dealing with account takeover are:

A. Prevent account take over by detecting malicious logins.
B. Reduce the monetization value of a compromised account.
C. Provide a way for compromised users to get back into their account.

13.1. Sensitive account changes should be stored in an append only way.
Storing things like passwords, emails, phone numbers, etc. in an append only format helps build the full picture when looking into compromised accounts. It also allows us to build user facing features, such as activity logs.

13.2. Provide an account recovery flow.
The recovery flow can be fully automated, fully human (i.e. the account is locked and the user has to call customer service), or a combination of both.

When applicable, the recovery flow should tell the user how their account was compromised (weak password, shared password with other services, compromised email followed by password reset, compromised phone number followed by password reset, etc.) and/or display an activity log.

In case a user's email account is compromised, you will need some form of shared secret to tell the real account holder and the attacker apart. Sites often use weak systems such as social security number, security questions, date of birth, passport scan, browser cookie, etc.

A combination of weak signals can be used to build a stronger signal. I personally prefer account guardians: each user chooses N friends who will be emailed a recovery code. The user then contacts the friends out-of-band.

Users should have the ability to flag their account in cases where we fail to detect it. Attackers often first compromise user's email accounts. The attackers can then perform a password reset. It is therefore helpful to allow users to use an old but recent password to trigger a recovery flow.

13.3. Require strong passwords.
Optional: require any password at registration time (to help user growth) but require a strong password after N days or at password reset time.

13.4. Store passwords using scrypt, bcrypt or pbkdf2.
Reduces the impact of a password database compromise.

Optional: set the number of rounds / hash parameters automatically (e.g. compute the ideal values once per day in a cron job) to offset hardware upgrades to faster servers.

Optional: prevent https://mathiasbynens.be/notes/pbkdf2-hmac because it leaks information on how the password is stored.

Optional: automatically validate against the inverse case and inverted case of the first letter, since those are common errors (caused by caps lock + auto capitalization).

13.5. Rate limit login attempts.
Ideally, you want to rate limit on a per account basis. However you want to prevent account lockout for legitimate users. I therefore recommend doing the following for each login attempt:

- let C be a per-account [counter, timestamp] pair, stored in some kind of persistent database or semi-persistent cache. When missing, C's default value should be [0, current time].
- let V = f(C.counter, C.timestamp, current time)
- if V < limit1
    - check the credentials, store [V+1, current time] if they are invalid.
- if limit1 < V < limit2
    - if a captcha response was not provided, prompt for a captcha. (don't update C)
    - if an invalid catpcha response was provided, reject the attempt (don't update C)
    - if a valid captcha response was provided, check the credentials. Store [V+1, current time] if they are invalid.
- if V > limit2
    - reject the login attempt. Display "account locked for your security, please try again later" (don't update C).
- f() is a counter function with decay. In practice, using a logarithm function makes it harder for attackers to figure out the backoff behavior, for example:

f(v, t1, t2) = v/(e^((t2-t1)/600)) reduces v's value by ~30% every 10 minutes (assuming t1 and t2 are in seconds).

Besides being hard to guess from the outside, such a function provides a way to limit the number of attempts per unit of time while only storing two values per user (counter + timestamp).

Optional: you can additionally perform per-ip address check or use other signals (such as machine-cookie, geolocation, etc.) to reduce the number of attempts that an attacker gets while avoiding legitimate users getting locked out.

13.6. Classify login attempts in real-time. Reject or limit suspicious logins.
Useful signals include user agent, ip address, asn, dns/tcp timing, etc.
When a login attempt is suspicious you can reject the user or let them access their account in a read-only way. You may want to send the user a sms/email telling them that a suspicious login was detected and that they can change their password if they don't recognize it.
Finally, you could require a second factor authentication or kick-in the account recovery flow (see 13.2).

13.7. Ensure sensitive account changes require recent password entry.
Changes to account information which can lead to an account take over (e.g. password, email, phone number, etc.) or other sensitive information change or disclosure (e.g. bank account) should require a recent password entry.

13.8. Ensure sensitive account changes trigger an email and/or sms notification.
The email should contain instructions to enable the account recovery flow (see 13.2) in case the user did not trigger the action.

Optional: use the email and/or sms as a confirmation step.

13.9. (optional) Provide users with a way to enable two factor authentication.
TOTP is a simple standard, implemented by applications such as Google Authenticator.

If the account recovery mechanism (see 13.2) does not require the second factor, think about risks of second factor bypass.

You can require two factor authentication on every login or keep a per-browser cookie and only require it on a once per browser basis.

14. Other

14.1. Commit hooks.

14.2. Unittest & code coverage.

14.3. Static analysis tools.

14.4. Denial of service protection

14.4.1. Defend against Hash-Flooding

User controlled hashes (url parameters, cookies, etc.) should not be stored in predictable hash buckets. An attacker can downgrade the hash function to a linked-list lookup.
(more info: https://131002.net/siphash/)

14.4.2. Limit size (and content type) for file uploads

Also think about file traversal (don't let users override system/existing files).
User controlled files should only be served back through a sandbox domain (something.notsquareup.com).
If the files are sensitive, it is best to serve them via: notsquare.com/<random_nonce>/
If the user controls the mimetype, you should blacklist application/x-javascript, text/javascript and application/javascript. You can also blacklist anything which contains the word "script" or build a whitelist of authorized mimetypes.

14.4.3. Protect against resource starvation (script execution limit, stack depth, #process, #fd, etc.)

In general, the container you are running in should take care of this for you via cgroups and other mechanisms.

14.4.4. Limit size of responses

It's common to have an endpoint which resizes images or returns large datasets (e.g. search results). Ensure a malicious person cannot request a gigantic dataset.

14.5. Parsers

It is recommended to avoid "parsing" non-regular languages with regular expressions. The framework should provide properly written parsers for common things like Markdown, JSON, XML, etc.

14.6. Don't try to blacklist potentially malicious things
You will never be as creative as your attacker, and you cannot predict how browsers will change.
E.g. don't try to filter onXYZ handlers, because <button form="x" formaction="javascript:...">, thanks to HTML5.

14.7. crossdomain.xml, clientaccesspolicy.xml, robots.txt, etc.

14.8. Schedule an automated scanner to run regularly
You never know, they might find something useful?

14.9. Input fields which contain sensitive info should have autocomplete="off"

14.10. Correct handling of cache invalidation at logout time.
Hitting the back button after logout should not give you back a session. Keep in mind that someone might hit the back button multiple times & try replaying the initial login request! There are also IE specific quirks that need to be taken into account.

14.11. Ensure tokens are stored and compared in a case-sensitive way.
Can we abstract out the way tokens work? I.e. have a generic nonce service?

14.12. Even if empty, serve crossdomain.xml and robots.txt.

Additional Resources

- Google Gruyere (http://google-gruyere.appspot.com/) is an interactive web application that explains common security vulnerabilities through examples.
- The Browser Security Handbook (https://code.google.com/archive/p/browsersec/wikis/Part1.wiki) is a webdoc covering the various browser inconsistencies and vulnerabilities they introduce.
- OWASP has a XSS prevention guide (https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet) . It is fairly dry and lacks clear examples.
- http://quaxio.com/login_systems/ has some useful info.
- The following twitter accounts are filled with web security jewels:
    - https://twitter.com/garethheyes
    - https://twitter.com/0x6D6172696F
- http://schd.ws/hosted_files/appseccalifornia2016/61/AppSec-PreventingSecurityBugsThroughSoftwareDesign-ChristophKern.pdf

- Valid ways to build CSRF protection (https://docs.google.com/a/squareup.com/document/d/1-yIL_UcnGzgA02Ywy3t9wz8lQgTEueAmHPu0PuhZTDA/edit?usp=sharing)
- Some info about our OAuth flow (connect.squareup.com) (https://docs.google.com/document/d/1xgOPw6mGfuIWAyY74UKa48BVBDB4nNfUKhPxGqqCDz0/edit)
- Attacking Ruby on Rails Applications (Phrack issue 69) (http://phrack.org/issues/69/12.html#article)
- Web security workshop for interns: https://docs.google.com/a/squareup.com/presentation/d/1RMFPvzgR1aQEpFaNNpCEhMN46Sf6-dJqmwENwAFRPOw/edit?usp=sharing
- Browser security SOP slides: https://docs.google.com/a/squareup.com/presentation/d/1QxfLAvuciDcN9bDL4GJ3uCziapk4aTMu5Ucmi3Thu1Q/edit?usp=sharing
- Web app security slides (really old): https://docs.google.com/a/squareup.com/presentation/d/1FXiucAtSnfN6KSDiE6suSmH424gLJZ9oH0CDM2lDfmI/edit?usp=sharing
- Facebook's secure dev (onboarding for new hires): https://www.owasp.org/images/b/ba/Fb_secure_dev.pdf

Change log

v0.9	Sep 4, 2014	Draft
v1.0	Dec 22, 2014	Initial version
v1.1	Jan 5, 2015	Added 5.8 (CORS headers)
v1.2	Jan 13, 2015	Incorporated elements from Webapp security requirements and Secure web programming.
v1.3	Feb 26, 2015	Added section 13 to prevent account takeover.
v1.4	March 20, 2015	Added 7.11 and 14.4.4
v1.5	March 31, 2015	Added 5.9
v1.6	June 15, 2015	Added 7.12
v1.7	Aug 31, 2015	Added 9.4
v1.8	Oct 26, 2015	Added 6.6
v1.9	Jan 11, 2016	Added 4.4
v1.10	Feb 23, 2016	Added 6.7
v1.11	Oct 3, 2016	Updated 8.7 (re: Firefox data uri + cookie theft).