HITBSecConf
Keeping Knowledge Free for Over a Decade

Dissecting CSRF Attacks & Defenses

Mike Shema

October 16, 2013

QUALYS

Cross Site Request Forgery

Identifying the confused, session-riding deputy. WHAT

Putting the attack in context. WHY

Analyzing & implementing countermeasures. HOW

Defending the browser. WHEN

User Agent

Double Agent

Secret Agent

```html
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv="refresh" content="0;url=https://one.origin/">
  <link ref="prefetch" href="https://two.origin/resource">
</head>
<body>
  <img src="https://three.origin/image" alt="">
  <iframe sandbox src="https://four.origin/content"></iframe>
  <a href="https://five.origin/something">click here<a>
</body>
</html>
```

Cross-origin requests are an integral design and expected behavior of HTML.

CSRF Mechanism vs. Exploit

Force a **victim's browser** to request a resource of the attacker's choosing.

```
<img src="https://target.site/icon.png" alt="">
<img src="https://another.site/images/button.png" alt="">
<iframe src="https://web.site/article/comments/a/b/c/"></iframe>
```

The request affects the **victim's context** with the web app in a way that either benefits the attacker or is detrimental to the victim.

```
https://target.site/changePassword?newPass=kar120c
```

Request Context

The **attacker chooses** an action to be performed.

```
https://target.site/changePassword?newPw=kar120c
```

The **browser includes cookies** to perform that action against the target app under the victim's session context.

Two Senses of Forgery

Creation

SOP restricts reading the response from a cross-origin request, not making the request.

Many elements automatically initiate a request.

XHR object can compose complex requests.

Counterfeit

Compose request with attacker's choice of values.

The request triggers a behavior of the attacker's choice made under the victim's context.

Request Creation

```
<form method="POST" action="changePassword">
<input type="password" name="newPass" value="">
<input type="password" name="confirmPass" value="">
<input type="submit">
</form>
```

```
POST /changePassword HTTP/1.1
Host: web.site
User-Agent: Mozilla/5.0 ...
...
Cookie: sessid=12345
Connection: keep-alive

newPass=kar120c&confirmPass=kar120c
```

```
https://website/changePassword?newPass=kar120c&confirmPass=kar120c
```

```
GET /changePassword?newPass=kar120c&confirmPass=kar120c HTTP/1.1
Host: web.site
User-Agent: Mozilla/5.0 ...
...
Cookie: sessid=12345
Connection: keep-alive
```

Request Subterfuge

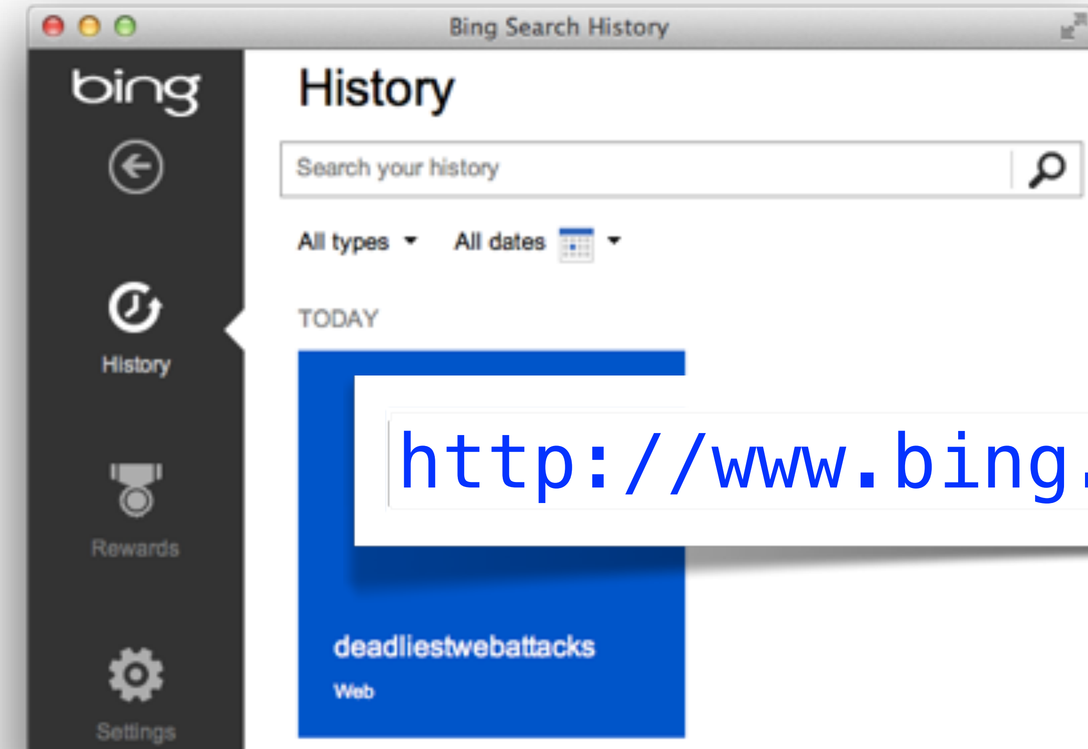
```
<img style="visibility:hidden"...
```

```
<iframe frameborder="0" height="0" width="0"...
```

```
<iframe seamless height="0" width="0"...
```

```
<iframe style="position:absolute; left:-1000px; top:-1000px"...
```

Risk Considerations




```
http://www.bing.com/search?q=deadliestwebattacks
```

```
http://192.168.1.1/apply.cgi
current_page=Main_Analysis_Content.asp&
next_page=cmdRet_check.htm&next_host=192.168.1.1&
group_id=&modified=0&action_mode=+Refresh+&
action_script=&action_wait=&first_time=&preferred_lang=EN&
SystemCmd=nvram%20show&
firmver=3.0.0.4&cmdMethod=ping&destIP=localhost&pingCNT=5
```

http://www.exploit-db.com/exploits/28652/

Are You ~~Experienced~~ Authorized?

Fundamentally, we want to distinguish between a user-intended action and a browser-initiated one.

Cross-origin requests that assume the victim's authorization are the problem (i.e. session riding).

Hence, a countermeasure might try to
...prevent the initiation of the request
...make it difficult to correctly compose the request
...separate the user's context from the request

Castles Made of Sand

Make requests harder to create.

CORS isolation

Make requests harder to counterfeit by including entropy or secrets.

Double submit cookie

Anti-CSRF token (nonce)

Tie the request to the user's session.

Separate authorization & authentication tokens

Secrets & Entropy

PRNG sizeof(PRNG)

hash(hash(hash(...(PRNG)...))) sizeof(PRNG)

hash(PRNG, salt) sizeof(PRNG + salt)

HMAC-SHA256(PRNG, secret) sizeof(PRNG + secret)

- HMAC-MD5
- HMAC-SHA512

PRNG & Entropy

“Deterministic”

- Poor seeding
- Poor algorithm
- Exposed state

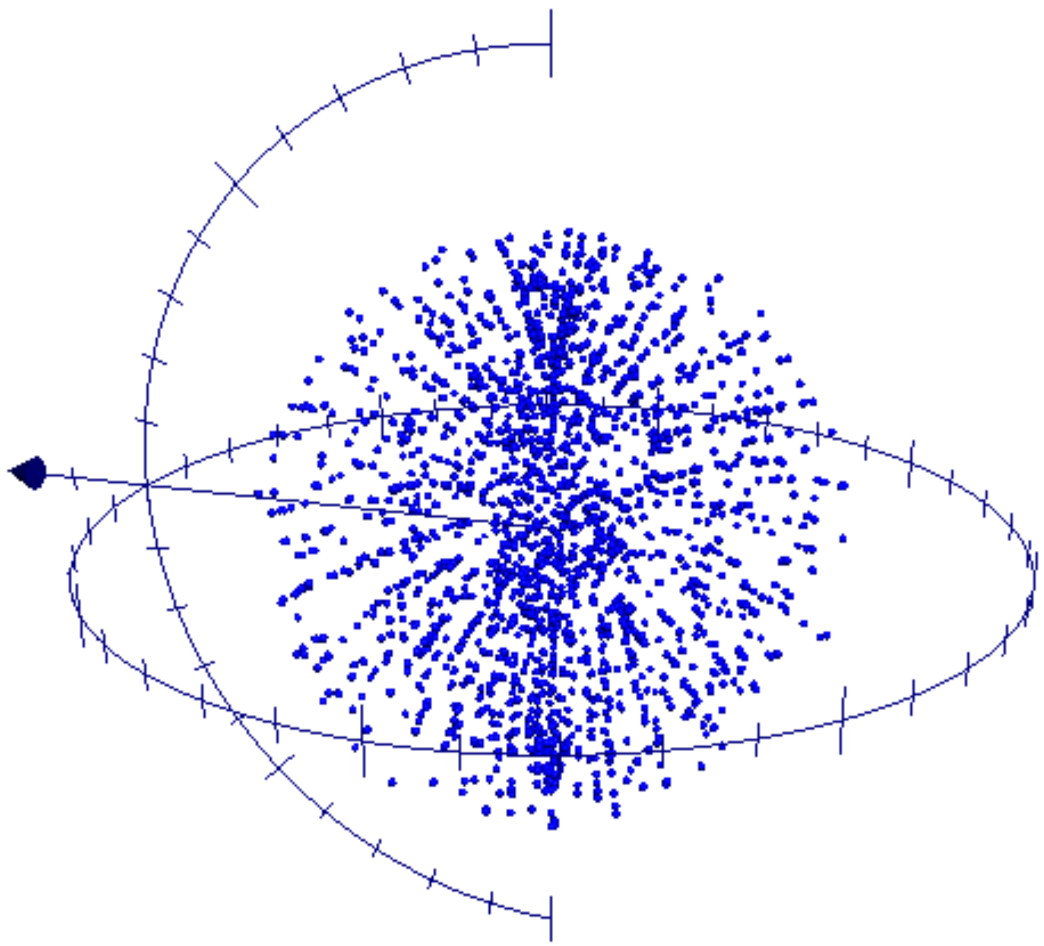
```
srand(1);
$x = rand();
```

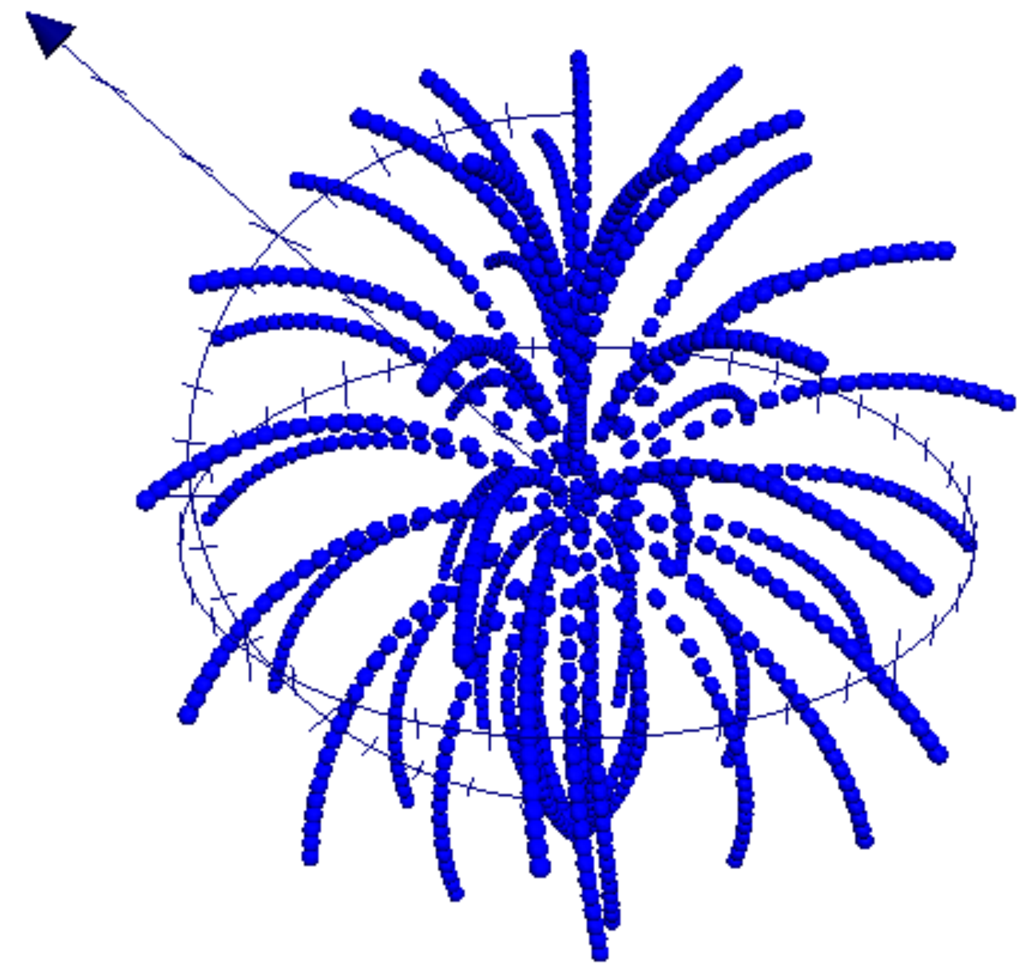
```
$x = sha256(rand());
```

Cryptographically secure algorithms designed to

- ...self-measure entropy to improve seeding
- ...resist prediction, bias
- ...resist compromise in case of state exposure

Heuristics





$\theta=2\pi X_n$

$\phi=\pi X_{n+1}$

$\rho=\sqrt{X_{n+2}}$

http://mathworld.wolfram.com/NoiseSphere.html

Entropic Horror

BH2012 -- PRNG: Pwning Random Number Generators

`sjcl.random`

`openssl rand 32 -hex`

HMAC & Secrets

Something other than the default value

keyboard cat

Something outside a dictionary

1

123

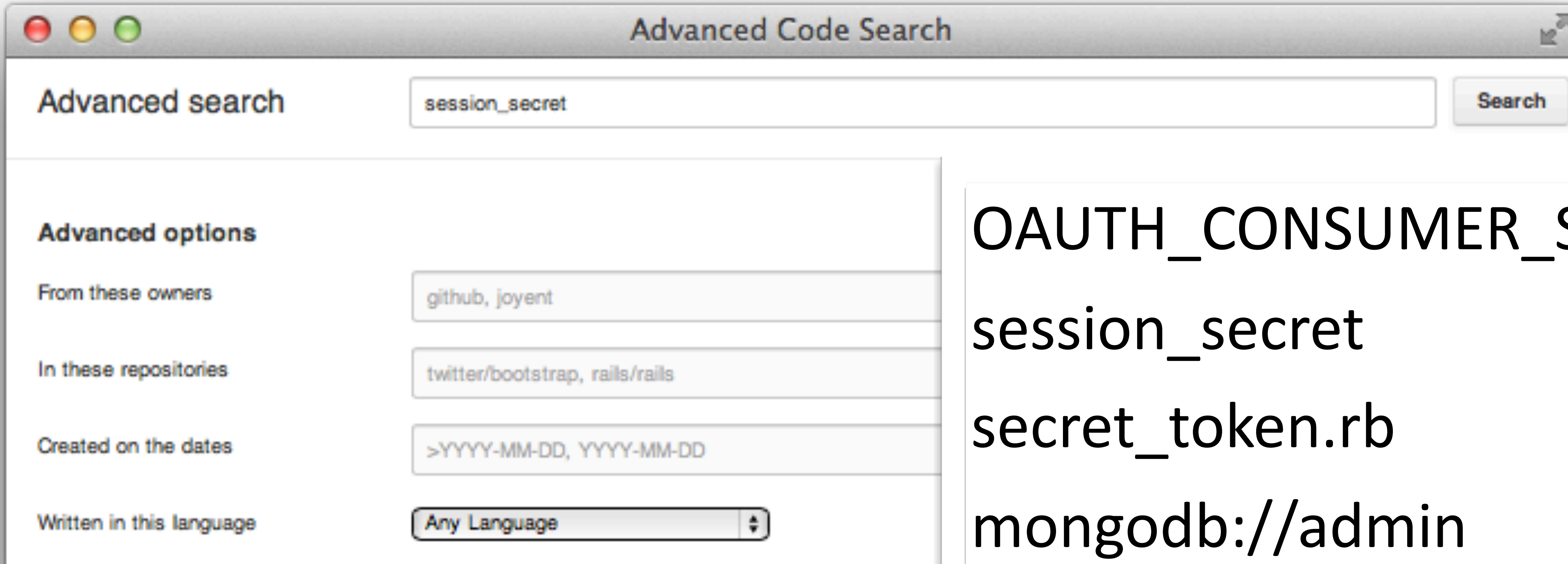
secret

Shad0wfax

```
$ ./john --format=hmac-sha256 --wordlist=words.txt sids.john
```

```
$ ./hashcat-cli64.app -a 0 -m 1450 sids.hashcat words.txt
```

.git~~ignore~~ explore



OAUTH_CONSUMER_SECRET

session_secret

secret_token.rb

mongodb://admin

ssh://root@

hmac-sha256

...

http://www.phenoelit.org/blog/archives/2012/12/21/let_me_github_that_for_you/

http://nakedsecurity.sophos.com/2013/01/25/do-programmers-understand-private/

CSRF Exposes Weak Design

Password change mechanisms that don't require current password.

Missing authentication barriers for sensitive actions.

- e.g. check-out and shipping to known vs. new address

Loose coupling of authentication, authorization, and session.

Dangerous Design

GET/POST negligence and mismatch

- form method modification
- PHP $_GET vs. $_POST vs. $_REQUEST

Unrestricted redirection

- e.g. https://web.site/page?returnUrl=https://CSRF/

"Link-based links"

- e.g. https://web.site/page?resource=CSRF.html

Attack Payloads

Griefing

Actions detrimental to user

http://justdelete.me/

Manipulation

Upvotes/downvotes

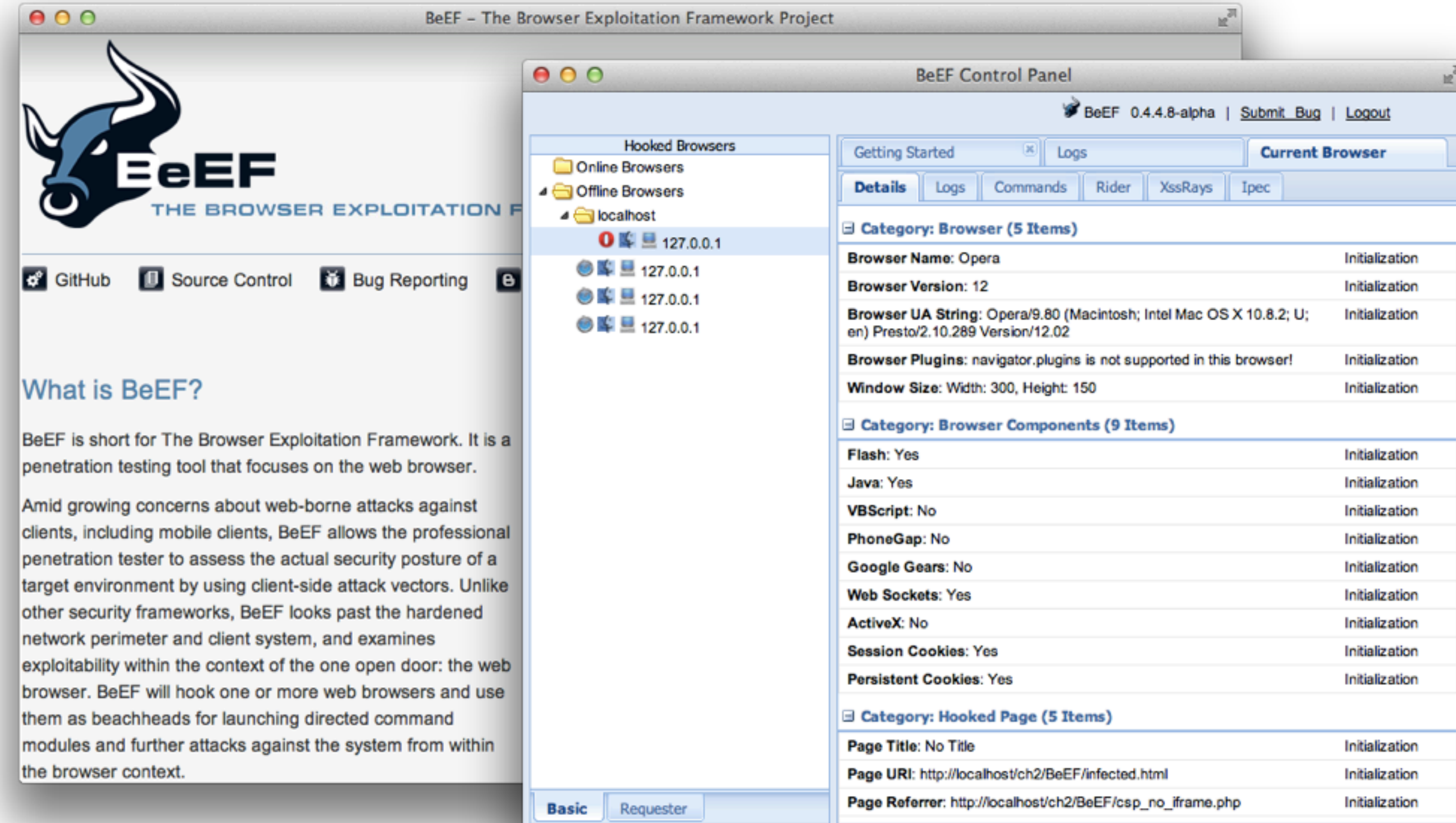
```
POST http://stackoverflow.com/posts/6655321/vote/2 HTTP/1.1
Host: stackoverflow.com

fkey=d2aad1a4a5e8326b26eb82307f25a072
```

Spamming

Messages from the user without authorization of user

(press control+c to stop)



Detection Methodologies

Pattern-based detection of token names

- Security by regexity
- Checks for presence, not effectiveness or implementation

Active test

- “Cookie Swap” between user session contexts
- Determine enforcement, not predictability

Mobile Apps

Recreating vulns from first principles

- Using HTTP instead of HTTPS
- Not verifying HTTPS certs
- But at least the apps are signed...

More areas to explore

- Not a browser, but making HTTP requests
- CSRF potential of malevolent ad banners

Wherever Browsers Roam

Does it speak HTTP(S)?

- Gaming systems
- Televisions
- Embedded devices

Does it have a user context?

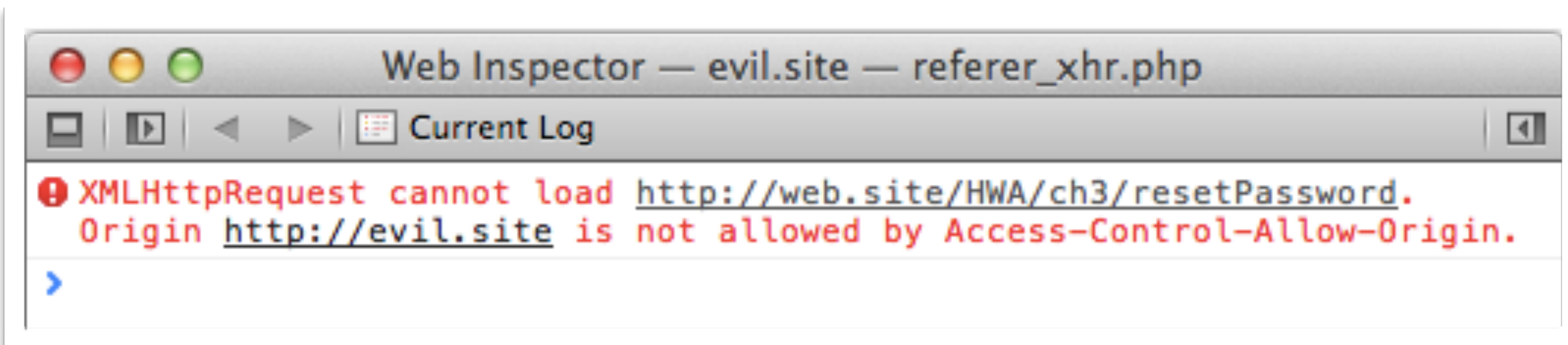
- ...or integration with social media?
- ...or control a security barrier?

Cross Origin Resource Sharing

Control the forgery (i.e. creation) of "non-simple", cross-origin requests

X-CSRF: 1

XCSRF /foo HTTP/1.1



CORS Isolation

Guarantees same Origin (or allowed cross-Origin)

- But only for "non-simple" XHR requests
- Must start inspecting the Origin header

Limitations

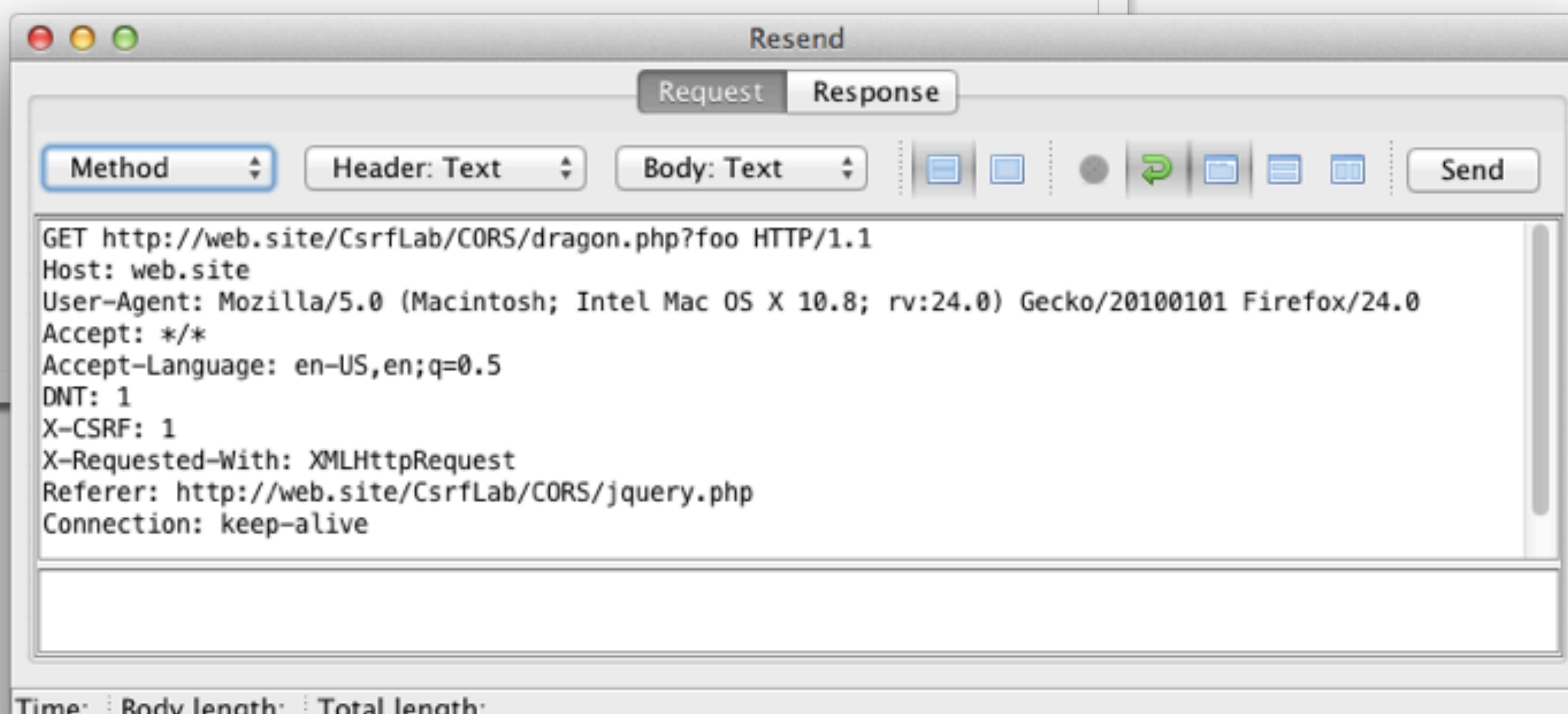
- Must be part of app's design and implementation
- Breaks "simple" cross-origin requests

<form id=”dragon”>

```
(function(){
"use strict";
$(document).ready(function() {
$("#dragon").submit(function(event) {
    $.ajax({
        url: "dragon.php",
        data: "foo",
        error: function(jqXHR, textStatus, errorThrown) {
            $("#results").html(textStatus + ", " + errorThrown);
        },
        headers: { "X-CSRF" : "1" },
        success: function(data) {
            $("#results").html(data);
        }
    });
    return false;
});

});

})();
```



Pre-Flight

```
OPTIONS http://web.site/CsrfLab/CORS/dragon.php?act=increase&gems=1 HTTP/1.1
Host: web.site
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.8; rv:24.0) Gecko/
20100101 Firefox/24.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Origin: http://evil.site
Access-Control-Request-Method: GET
Access-Control-Request-Headers: x-csrf
Connection: keep-alive
```

```
HTTP/1.1 200 OK
Date: Wed, 16 Oct 2013 07:13:31 GMT
Server: Apache/2.2.25 (Unix)
X-Powered-By: PHP/5.3.27
Set-Cookie: PHPSESSID=mkpb5bn4cbp86orsjekmp6asb7; path=/
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Access-Control-Allow-Origin: http://web.site
Access-Control-Allow-Headers: X-CSRF
Access-Control-Max-Age: 10
Content-Length: 0
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=utf-8
```

Content Security Policy

```
CSP: default-src 'self'
```

```
<input type="text" name="q" value="foo"
autofocus/onfocus=alert(9)//"">
```

```
CSP: default-src 'self' 'unsafe-inline'
```

```
<input type="text" name="q" value="foo"
autofocus/onfocus=alert(9)//"">
```

Speaking of CSP

```
<!doctype html>
<html>
<head>
<meta http-equiv="X-WebKit-CSP"
      content="img-src 'none'; report-uri
'https://csrf.target/page?a=1&b=2&c=3'">
</head>
<body>
<img alt="" src="whatever">
</body>
</html>
```

Partial POST Forgery

```
POST /page?a=1&b=2&c=3 HTTP/1.1
Host: csrf.target
User-Agent: Mozilla/5.0 ...
Content-Length: 116
Accept: */*
Origin: null
Content-Type: application/x-www-form-urlencoded
Referer: http://web.site/HWA/ch3/csrf.html
Cookie: sessid=12345
Connection: keep-alive

document-url=http%3A%2F%2Fcsrf.target%2FHWA
%2Fch3%2Fcsrf.html&violated-directive=default-
src+%27none%27
```

One Attack Among Many

Crosstown Traffic

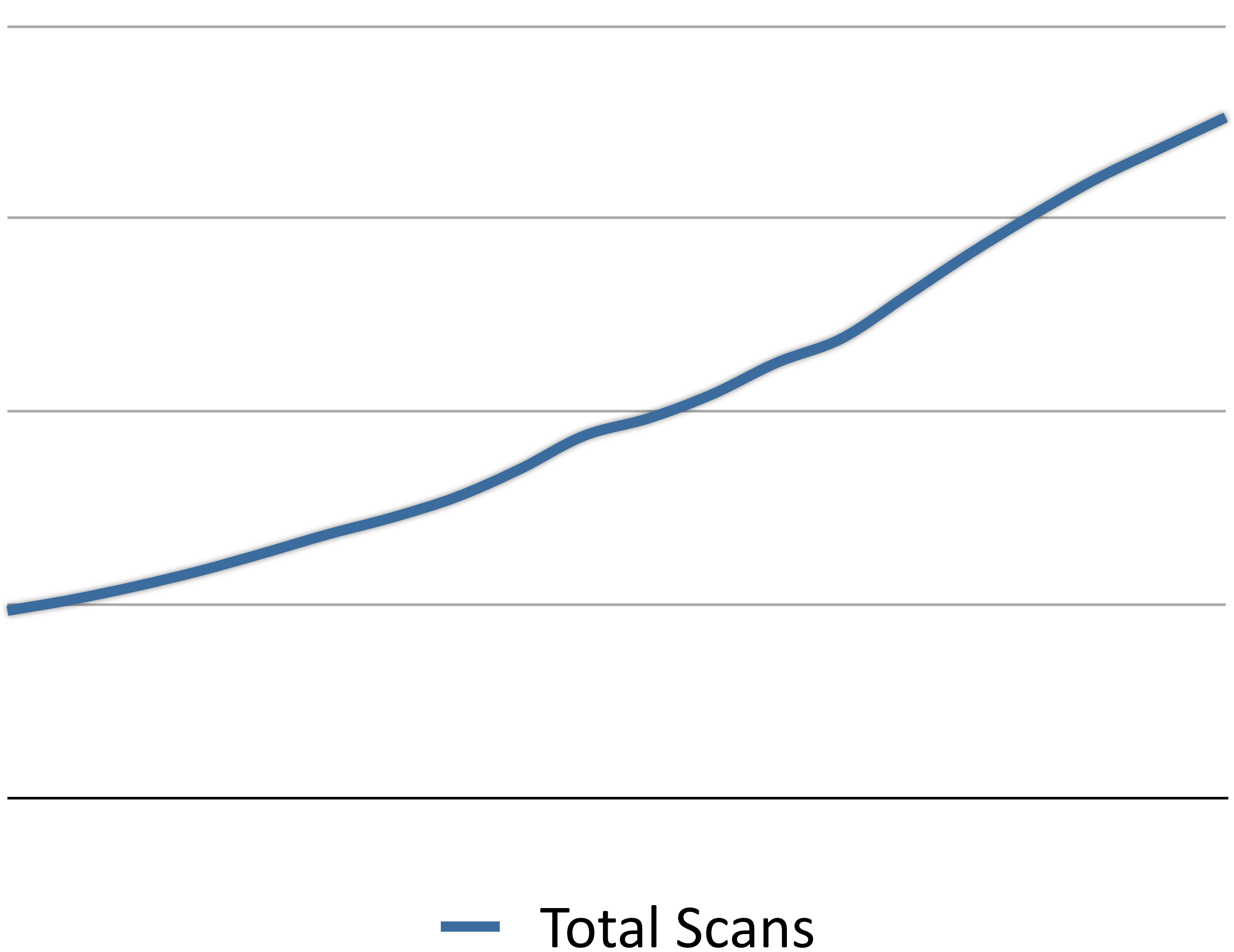
HTML injection, cross-site scripting

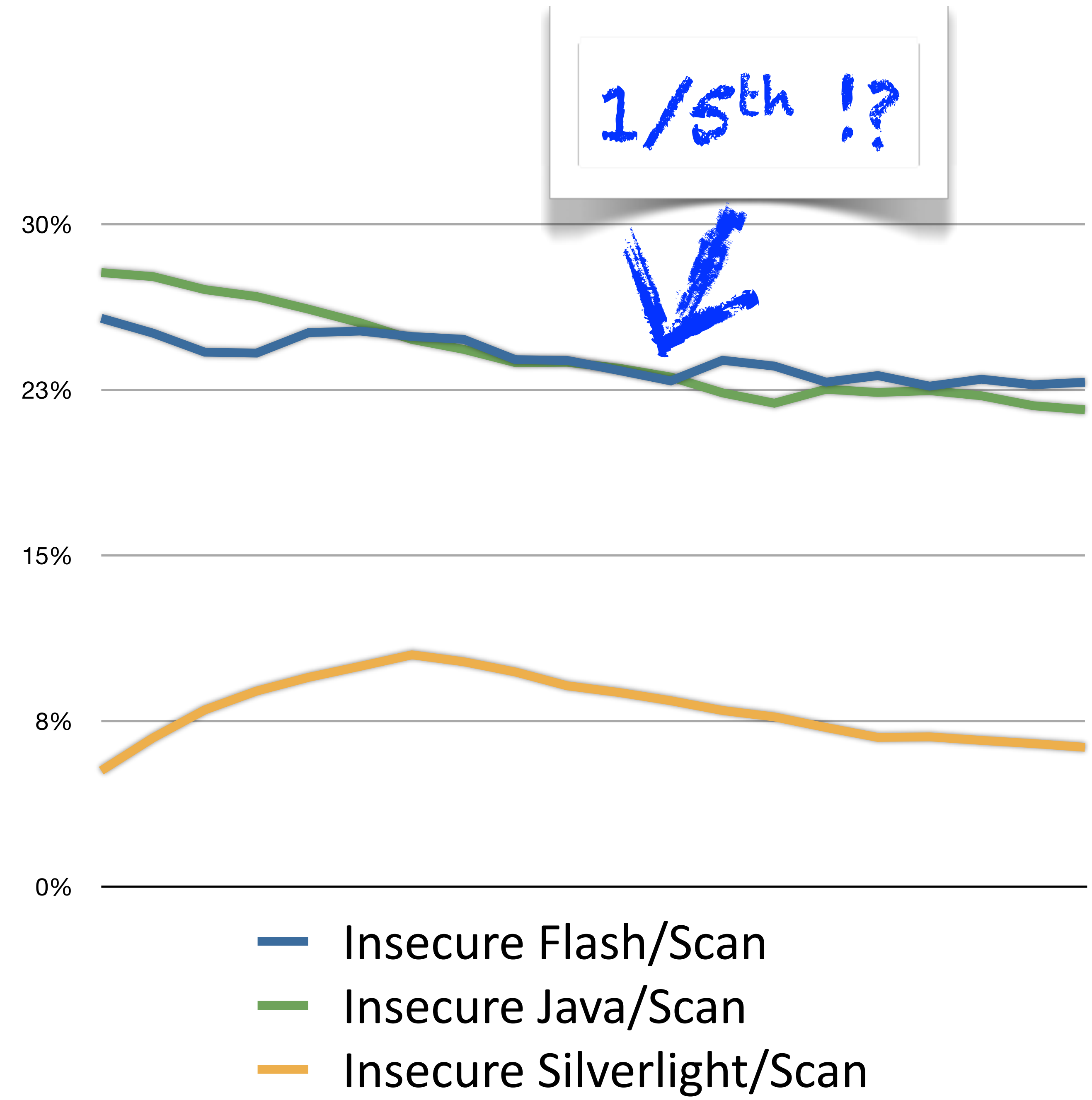
- It's executing in Same Origin
- CSRF countermeasures are intended to prevent cross-origin attacks
- Start using Content Security Policy

DNS, cache poisoning, sniffing, ...

- Start using HSTS
- Where did DNSSEC go?

Vuln Background Radiation





20 months starting November 2011

Plugins

Outside of SOP

- Outside of privacy settings

Compose requests

- Unrestricted header creation
- Raw packets

Eternally insecure

- To be replaced by HTML5, <canvas>, <audio>, <video>

••• ▬ ▬ ▬ •••

AND THEY HAVE A PLAN.

Security of Sessions

Focus on the abuse of session context

- Session-riding, confused deputy

Control when cookies accompany requests initiated from a cross-origin resource

- Similar to CORS enforcement of "non-simple" requests
- Isolate the user's session context

Simplicity of Settings

Syntax like CSP, behavior like CORS

- Simple behavior with fewer chances of mistakes
- Leverage pre-flight as a permission check for context

Don't require changes to application code

- Add headers via WAF
- Provide more flexibility by opt-in to exceptions

Should Often Succeed

Don't break the web, ease adoption

- Ad banners
- "first visit", blank browsing context
- Deal with domains & subdomains vs. Origins

Browsers have to support it

- Old, unpatched browsers forsaken to the demons of insecurity anyway

Some Ordinary Syntax

On the web application, define a policy:

```
Set-Cookie: cookieName=...
Content-Security-Policy:

  sos-apply=cookieName ‘self‘
  sos-apply=cookieName ‘any‘
  sos-apply=cookieName ‘isolate‘
  sos-apply=* ‘self’
```

Policies

self -- trigger pre-flight, cookie included only from same origin unless given exception

any -- trigger pre-flight, cookie included unless given exception

isolate -- no pre-flight, no exceptions. Cookie only included from same Origin.

(?) `sos−remove=`*`cookieName`* to remove policy

Some Ordinary Syntax

If a cookie has a policy (or no policy), and a request is generated by a resource from the same Origin.

...work like the web works today.

If a cookie has a policy of ‘isolate’, and a request is generated by a cross-origin resource.

...never include the cookie.

If a cookie has a policy of ‘any’ or ‘self’, and a request is generated by a cross-origin resource.

...make a pre-flight check

Why Pre-Flight?

Cookies apply site-wide (including subdomains!), without granularity of resources.

The /path attribute is not a security boundary

An SOS policy instructs the browser for **default** handling of a cookie.

A particular resource can declare an **exception** by responding to the pre-flight.

Pre-Flight Request

[prereq] A policy of ‘any’ or ‘self’

[prereq] Cross-origin resource initiates request

Browser makes CORS-like request:

```
OPTIONS http://web.site/resource?a=1&b=2 HTTP/1.1
Host: web.site
User-Agent: ...
Origin: http://evil.site
Access-Control-SOS: cookiename cookiename2
Connection: keep-alive
Content-Length: 0
```

Pre-Flight Response

Web app receives a pre-flight request.

Supply an expires value so the browser can cache the response.

...if a policy should be enforced for the specific resource:

```
HTTP 200 OK
Access-Control-SOS-reply: ‘allow’ | ‘deny’; expires=seconds
```

Pre-Flight Response

...if the resource is not exceptional, browser follows established policy

- ‘any’ would include the cookie for cross-origin
- ‘self’ would exclude the cookie for cross-origin

Benefits

- Web app can enforce per resource, per cookie
- Sees the Origin header
- Expiration eases performance with caching

Two Sets

- Policy applies to cookies for all resources (entire Origin)
- Policy can be adjusted by a resource
- Pre-flight response shouldn't leak information about cookies for which it has a policy
  - If the client can't ask for the right cookie, then no response.
  - Respond with 'deny' if the cookie doesn't exist

Remember

Browser tracks...

- Cookies for which a policy has been applied.
- Resources that respond to cross-origin requests with exceptions to the policy.
- Cookies and destination origin, source origin doesn't matter

Web App

- Applies a policy at each Set-Cookie
- Applies a policy at a bottleneck

Goals

Ease adoption

- Familiar syntax
- Small command set

Acknowledge performance

- Cache pre-flight responses
- Only track "all other origins" to origin, not pairs of origins

The “WordPress Problem”

Strong anti-CSRF token is present in WordPress trunk

WP plugins keep forgetting to use it

../wp-admin/admin.php?page=...

Must continually protect every new action

...or protect the /wp-admin/ directory

sos-apply=*cookieName*; ‘self’

Mitigate Social Engineering

Should prevent situations where user is tricked onto clicking a link/submitting a form on attacker's page (i.e. different origin) that submits to targeted origin

Use X-Frame-Options to deal with clickjacking

If 6 Was 9

- No secrets, no entropy
  - Easier on embedded devices, fewer mistakes
- Enforcement by origin
  - Exception-based for flexibility
  - Shift state tracking from server to browser
- Pre-flight can be handled by WAF
- ‘isolate’ and expire deal with overhead of pre-flight
  - (Which is only for cross-origin anyway)

Imperfect

Much easier to isolate an origin than work with cross-origin requests.

Decorates resources instead of decorating the cookie.

When Old Becomes New

- Update browsers
  - Still have to support legacy, although the window to the past is shrinking
  - People still use old browsers for good reasons, TorBrowser using FireFox ESR
- Fix frameworks
  - Use cryptographically secure PRNG
  - Don’t reuse example passphrases
  - Use XHR brokering with custom headers
  - Separate authentication and authorization

Strong Foundations

Use HSTS

Use CORS isolation (i.e. “non-simple” requests)

Send an SOS

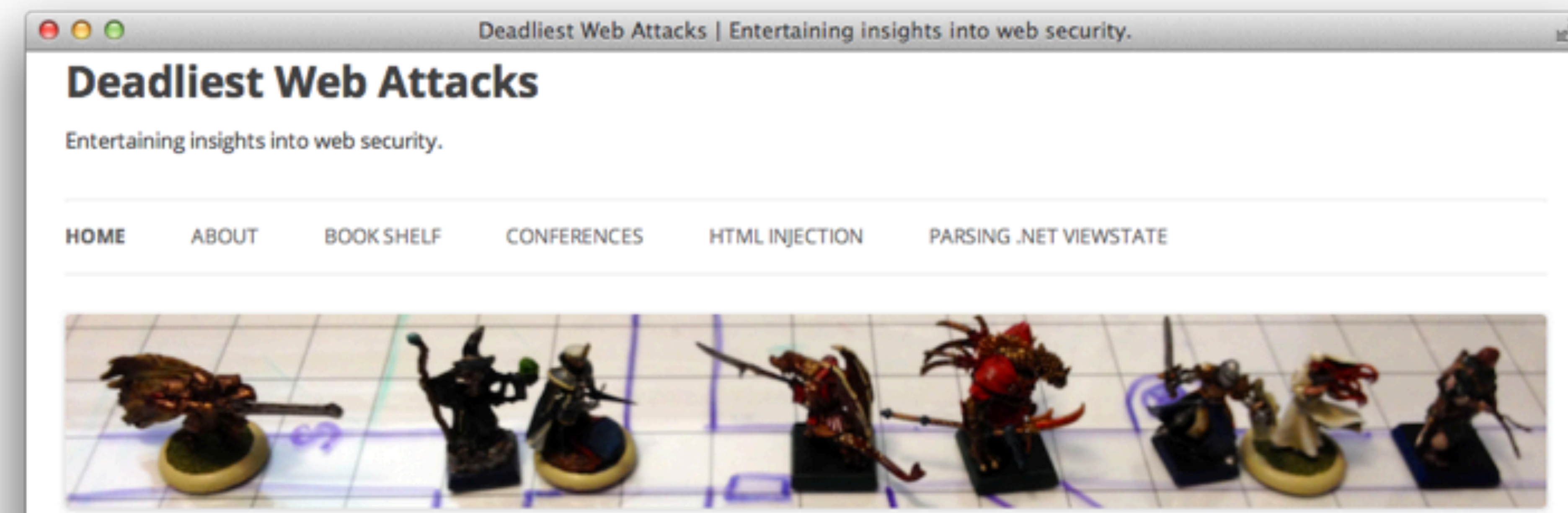
SIX: ALL OF THIS HAS HAPPENED BEFORE.

BALTAR: BUT THE QUESTION REMAINS, DOES ALL OF THIS HAVE TO HAPPEN AGAIN?

Thank You!

Contact @CodexWebSecurum

Content http://deadliestwebattacks.com



References

beefproject.com

crypto.stanford.edu/sjcl/

github.com/mutantzombie/SessionOriginSecurity

hashcat.net

media.blackhat.com/bh-us-12/Briefings/Argyros/BH_US_12_Argyros_PRNG_WP.pdf

research.microsoft.com/en-us/um/people/helenw/papers/racl.pdf

www.adambarth.com/papers/2008/barth-jackson-mitchell-b.pdf

www.openwall.com/john/

www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project