

# Website security

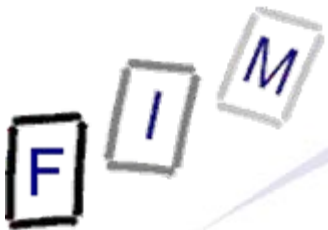
**Security and Privacy**  
**VSE Prag, 7 - 11.6.2010**

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

E-Mail: [sonntag@fim.uni-linz.ac.at](mailto:sonntag@fim.uni-linz.ac.at)  
<http://www.fim.uni-linz.ac.at/staff/sonntag.htm>



- Typical vulnerabilities
  - » How do they work?
  - » How to prevent them?
  - Cross-site-scripting
  - SQL injection
  - Buffer overflows
- Google hacking
- Error messages
- Input validation
- Access control
- Ajax security



# 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>

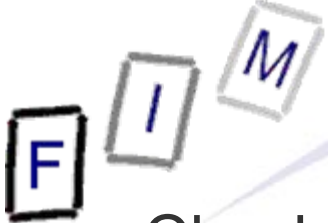


# SQL injection

- User input is used as part of the input to a database
  - Typically these are SQL databases today
    - » Problem applies to all kinds of database languages and input!
  - Typical instances: Login forms, other forms
- Example: Search form
  - `SELECT * FROM Articles WHERE Text LIKE '%" + searchword + "%';`
  - What if someone enters the following search term:  
`'; DROP TABLE Articles;--`
    - » "--" at the end → Rest of line is comment!
  - Result: `SELECT * FROM Articles WHERE Text LIKE '% '; DROP TABLE Articles;-- %';`
    - » Selects all articles and then deletes the table!
  - Note: You can obviously also insert any data, which is interesting for XSS attacks, as input verification is subverted!
- More data can be elicited through illegal SQL



- 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 often 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`, ...)!



- Checking a table exists:
  - `IF (SELECT * FROM login) BENCHMARK(1000000,MD5(1))`
- 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 --`
  - `' UNION SELECT 1, 'user','xyz',1 --`
    - » Note: Requires previous knowledge of the query structure!
- Shutdown server: `';shutdown --`

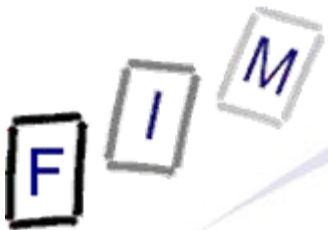


- Escaping ' and ; are good, but insufficient!
  - Techniques exist to "live without" or use other options
    - » Just removing them? → uni'on sel'ect @@version-'
  - Better: Live without them completely
- Verify all input data according to a whitelist
  - And strictly enforce length limits → SQL injection is usually (but not always!) a long string to be of use
- Limit database permissions
  - Should always be separate user with least privileges possible
- 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: XSS will not be prevented by this, only DB modifications!



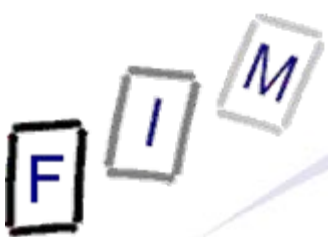
# Cross-site-scripting (XSS)

- 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 (SSL) 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



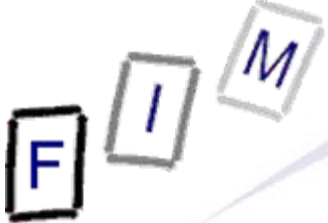
# Cross-site-scripting: Main types

- Non-persistent: "Local" or "Reflected" XSS
  - JavaScript code on the page reads user input (or the URL – also provided from site-externally!) and displays it
  - Data entered by the user is "reflected back", i.e. used to construct the response (dynamic page generation)
    - » Note: This input can be encoded in an URL, which can be obfuscated to be not recognizable as program code!
      - Example: Link in Spam messages
    - » Could also be reflected by an error message
- Persistent: "Stored" XSS
  - Data is entered by users and stored persistently; it is later used as input for generating the pages
    - » Data can come from any source: forms, log files, E-Mails, ...
  - This can directly affect a huge number of users!
    - » No need to convince someone to click on a special link
  - Viruses and even worms are possible in this way



# Cross-site-scripting: Examples

- 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 third-party sites
- Encoding possibilities to hide the program:
  - 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!



- Never try to filter out offending content
  - E.g. MySpace worm: "javascript" was filtered
  - Encoding the word in two separate lines → Slipped through!
- Always escape everything you write to the user
  - Escaping <, >, (, ), #, &, ", ' significantly increases security!
    - » Result: No HTML can be embedded at all!
    - » Use Wiki technologies → Customs "tags" which are converted to explicit and known HTML tags on output
  - "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
  - Con't click on links in spam messages/message boards
  - Turn off JavaScript and disable plugins



# 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
  - Usually the Administrator/root
- Approximately 60 % of all application vulnerabilities
  - Web servers and their programs (plugins) are affected too!



# Buffer overflows

## Original state

Return address = 0x1234
Local variable A = 17
Local variable B = FALSE
Local array[3] = '\0'
Local array[2] = 'T'
Local array[1] = 'E'
Local array[0] = 'G'

## Normal program

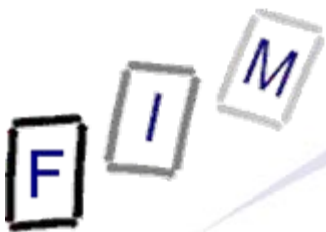
Return address = 0x1234
Local variable A = 17
Local variable B = FALSE
Local array[3] = '\0'
Local array[2] = 'T'
Local array[1] = 'U'
Local array[0] = 'P'

## Buffer overflow

Return address = 0xFFFF
Local variable A = 0x0000
Local variable B = 0xFF3C
Local array[3] = '0x0355'
Local array[2] = '0x06D0'
Local array[1] = '0xE512'
Local array[0] = '0xFA34'

Jump to ...

- The stack grows from high address down towards low ones
- Local variables are used from low addresses up to high ones
  - This is one of the main problems!
  - 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 ...



# 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/EPROM/...) as well
    - » Note: Self-modifying programs are extremely rarely useful!
- Another reason: Compilation
  - Interpreted programs are usually safe (They check bounds)
    - » As long as the interpreter is correct!
- 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 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!`)

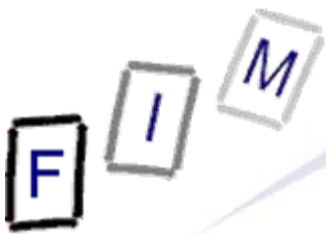
# Buffer overflows: Prevention



- Run servers under lesser permissions → chroot, ...
  - Successful attacks can then "only" affect this one application
- Always check the length of input data
  - Never ever use gets, strcpy, strcat, scanf, sprintf (and others!)
    - » Use fgets, (strncpy, strncat), sscanf, snprintf
    - » AND take care of null-termination (parameters and result!)
  - 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!
  - Duplicate return address after all local variables
- Use programming languages with automatic boundary checking: Java, C#
  - Attention: C# → Procedures can be marked as "unsafe"
  - No overflow protection then!

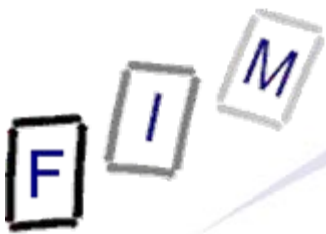


- Data execution prevention
  - Mark the stack as "non-executable" → The overflow happens and the wrong return address is used, but the code must come from somewhere else (e.g. the heap)
    - » Hardware support for this in modern processors!
    - » Not foolproof: Load stack with "fake stack data" for calling system functions to disable the execution prevention
- Safe libraries: Replacement libraries with integrated checking of bounds for those functions, which do not check them
  - Complex → Must monitor other functions as well
- Split stack: Separate stack for local variables and control information (return address)
- Double stack: Execute program twice simultaneously with the stack going in different directions
  - Stack overflows can only compromise of the two!



# Google hacking

- Not an attack as such, but the preliminaries: Searching for vulnerable systems
  - 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
    - » `/* TODO: Fix security problems */`
- Note: The subject of the attack has no chance at all of noticing this, as his server is not touched at all!
  - Attacks come "out of the blue"
    - » But not unprepared: Only pages existing for a long time can be found; usually the vulnerability is older too
  - Pages might be cached by Google → Further possibilities!



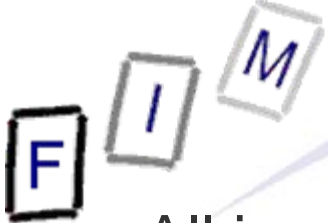
# Google hacking

- Requires advanced Google operators:
  - link: Search within hyperlinks
  - 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
  - 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
- Prevention:
  - Automated tools available: E.g. SiteDigger
    - » Can also be used on your own pages to look for "weaknesses"!
  - Use "robots.txt" to limit web crawlers to "relevant" pages
  - Captchas/Remove from Google index (→ Desirable?)



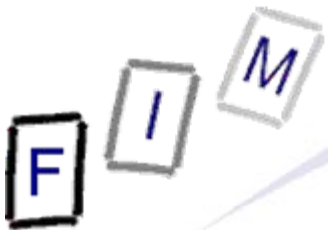
# Error messages

- 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"
- What to do:
  - For debugging → Turn all output options on
  - For release → Turn **all** output options **off!**
- Just return a page stating "An error occurred"
  - Detailed information should be logged, sent to the admin, ... but **never be published!**
  - May not include any "offending" user input
    - » **XSS reflection vulnerability!**

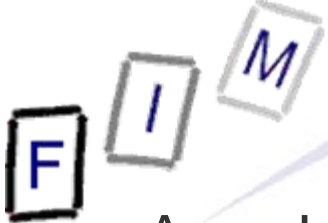


# Input validation

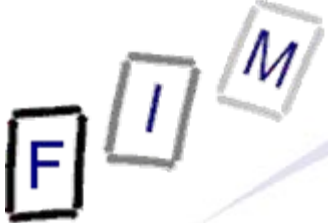
- All input into a web application must be strictly validated
  - **Syntax: Does it look correct?**
    - » Example: (ASCII) Strings may only contain one \0 at the very end
  - **Semantics: Does it have the correct meaning**
    - » Usually not a security problem, but more whether the application will perform the intended work
      - But may open vulnerabilities as well!
- Always use a positive specification
  - Exploits can use nearly unlimited possibilities for hiding!
  - Validation against:
    - » Data type, allowed character set/range, minimum/maximum length, “Null”/0/... allowed?, required/optional
    - » Good method: Regular expressions strictly defining the syntax!
- Attention: Generic security devices (content inspection on firewall) can typically use **negative** specifications only!
  - **Insufficient; only the application know exactly what it expects!**



- Authentication: Who is someone?
  - Authorization: What is this person allowed to access?
- Authentication must always be server-based
  - Cookies may never contain the user ID
    - » Always assign a random value and look this up in the local database to get the user ID; expire it after some time!
  - User **MUST** pass the login page
    - » Not having a link to further pages is insecure!
    - » Use redirection and path canonicalization (→ traversal attacks!)
- Use restrictive file permissions on the server
  - Configuration files, .htaccess, ... may not be read by the application (disallow directory browsing as well)
- Make sure caching is turned off
  - Various browsers/version support different possibilities!
- Authorization: Check in every file and always!



- 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!

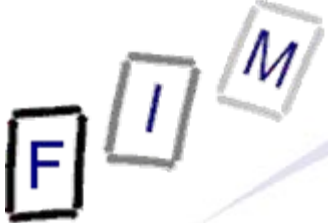


- Applications are vulnerable, but web applications
  - are more secure, as their source code is often not available
  - are more insecure, as they exist in numerous instances on powerful servers and can be tested for as long as desired
- Especially important: Input validation
  - Else: SQL injection, buffer overflow, cross-site-scripting, ...
  - Do not **ever** trust **anything** from the **user**!
- Don't just integrate scripts, data, programs (Web 2.0!) from other sources → Might be dangerous or just vulnerable!
- 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!

F I M

# Questions?

Thank you for your attention!



- **SWAT: Top 10 Web Application Security Vulnerabilities**  
[http://www.upenn.edu/computing/security/swat/SWAT\\_Top\\_Ten.php](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](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](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-ok/>
- **Google Hacking Database (no longer active):**  
<http://johnny.ihackstuff.com/ghdb.php>