Web Application Security (Part 1)

Richard Ostertág

Department of Computer Science
Comenius University, Bratislava
ostertag@dcs.fmph.uniba.sk

2016/17

The web sites era (Web 1.0)

- static documents
- browsers used only for retrieving and displaying those documents
- one-way flow of "interesting" information (server $\rightarrow$ browser)
- typically without user authentication
- every user: equal treatment, the same content
- users do not create presented content
- security threats abused mainly vulnerabilities in the web server software
- compromised server
  - no leakage of sensitive information (all info. already open to public)
  - modification of content
  - server's storage and bandwidth used to distribute "warez"

The web applications era (Web 2.0)

- interactive dynamic applications
- the browser becomes the operating system, which runs the web application
- two-way flow of "interesting" information (server $\leftrightarrow$ browser)
- usually there is users authentication (registration, login)
- each user: different treatment, different personalized content
- users are creating presented content
- security threats are abusing also vulnerabilities in web application
- compromised server / web application
  - leakage of sensitive information (personal data, credit card numbers)
  - modification of content (money defrauding, attacks on other users)
  - use of bandwidth, processing power or storage capacity
    - e.g. for creating botnets to send spam or DDoS attacks

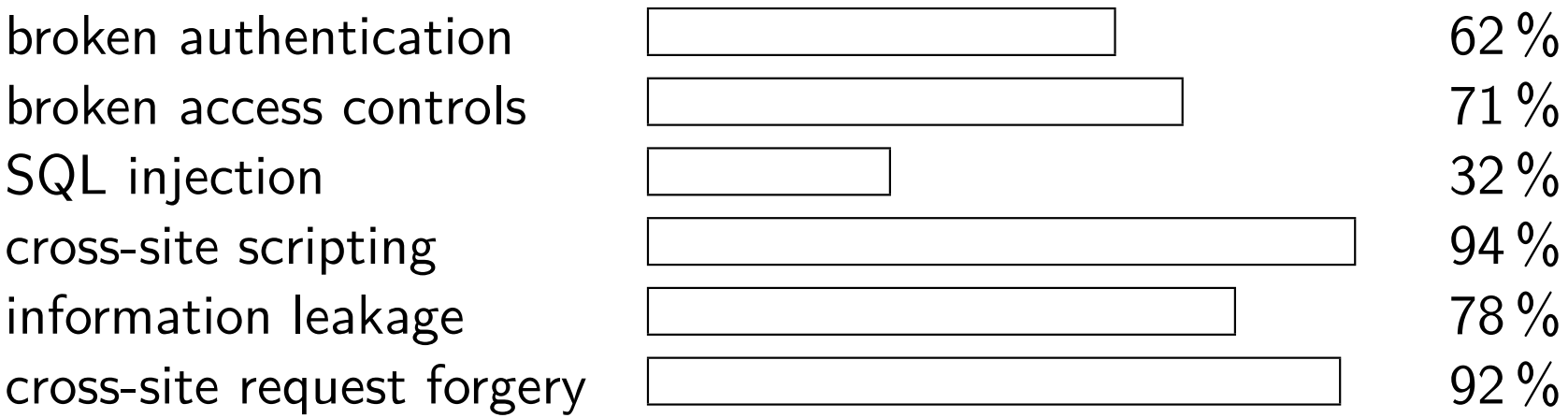
Common internet web application functions

- Internet stores
  - Amazon
  - Hej
  - Alza
- Social networks
  - Facebook
  - Twitter
  - Second life
- Internet banking
  - Tatrabanka
  - VÚB
  - Slovenská sporiteľna
- Search engines
  - Google
  - Bing
  - Baidu
- Internet auctions
  - eBay
  - Aukro
- Internet casinos
  - bwin
- Web logs
  - Blogger
- Web mail
  - Gmail
  - Hotmail
- Internet media
  - CNN
  - SME
- Internet encyclopedia
  - Wikipédia

Common intranet web application functions

- HR applications
  - payroll information
  - recruitment
  - disciplinary procedures
- Key infrastructure administrative interfaces
  - web server
  - mail server
  - virtualization server
- Collaboration software
  - document sharing
  - project management
- Business applications
  - Enterprise Resource Planning
  - Customer Rel. Mangement
- Software services
  - e-mail web interface
- Traditional desktop applications migrated to the web
  - word processors
  - spreadsheets

Common web application vulnerabilities



Source: Dafydd Stuttard, Marcus Pinto: The Web Application Hacker's Handbook

SSL, PCI – false sense of security

- users are aware of security issues in web applications
- most applications state that they are secure because they use SSL[1]:
  This site is absolutely secure. It has been designed to use 128-bit Secure Socket Layer (SSL) technology to prevent unauthorized users from viewing any of your information. You may use this site with peace of mind that your data is safe with us.
- organizations also cite their compliance with Payment Card Industry (PCI) standards:
  We take security very seriously. Our web site is scanned daily to ensure that we remain PCI compliant and safe from hackers.
  - use and maintain a firewall
  - do not use default passwords
  - encrypt data transfered over public networks
  - maintain security policy
  - ...

[1] Source: Dafydd Stuttard, Marcus Pinto: The Web Application Hacker's Handbook

The core security problem

- attacker can submit arbitrary response
  - client side safety checks can be circumvented
  - attacker does not need to use a web browser to access the application
  - attacker (on the client side) can:
    - read, modify, delete, or reuse any data that received from the server or web-browser is sending to the server
    - generate new data and inject them into the communication
    - manipulate (suppress, modify, repeat) any operations performed on the client side
  - an attacker can manipulate all kinds of data the client operates with:
    - URL (path parts, GET parameters)
    - form fields (even hidden)
    - commented out parts of HTML documents
    - scripts (review, change)
    - cookies and other information in HTTP headers (e.g. session ID)
- it must be assumed that all inputs are potentially dangerous

Key problem factors

- underdeveloped security awareness
  - field of web application security is relatively young (vs. networks, OS)
- custom development
  - many web applications are developed in-house by an organization's own staff with different skills
  - every application is different and may contain its own unique defects
- deceptive simplicity
  - a novice programmer can create a powerful application from scratch
  - difference between producing functional code and secure code
- new threats for web applications are conceived at a faster rate than is now the case for older technologies
- resource constraints (time, money, developers, . . .)
  - functionality takes precedence over the security
- overextended technologies
  - many of the core technologies employed in web applications have been pushed far beyond the purposes for which they were originally conceived and unforeseen side effects emerge

Core defense mechanisms

- user access control
  - prevent users from gaining unauthorized access
- user input validation
  - prevent unwanted behavior of application even for malicious entry
- handling attack
  - correct functionality even in the event of a direct attack
  - defensive and offensive measures to repel the attack
- application monitoring
  - administrator can react immediately

User access control

- basic components:
  - identification and authentication
  - session management
    web application issues a token that identifies the user session
  - access control

- fault in any component may lead to unauthorized access

Identification and authentication

- identification
  - most often by login name
- authentication
  - most often with a password
- the attacker can
  - obtain login names
  - obtain passwords
  - bypass the authentication function
    - due design flaw

Identification and authentication (the classic problems)

- weak passwords
  - short
  - small alphabet
  - from dictionary
  - guessable (date of birth)
    - quality control (e.g. a dictionary attack)
- the same password to different systems
- no time limit for password age
  - password age checks
- keylogger
  - virtual keyboard
  - one-time passwords
  - zero knowledge proofs
- late transition from HTTP to HTTPS
- failed login should not differentiate bad name from bad password

Session management – session hijacking[2]

- disclosure of session identifier
  - interception of communication
  - "Referer" header when switching to other sites
  - browser history
  - access to the user's computer – extraction of stored cookies, ...
- session ID guessing
  - identifier is generated in a predictable manner (for example simple arithmetic progression)
  - identifier has a small range of possible values
- possible protection:
  - protection is not easy and 100 % effective
  - identifier is not just a random number
  - concatenated with the hash of the IP address of the server and the client, User-Agent header of the client and some secret value
  - then the attacker can not simply use a stolen ID

[2] attacker finds ID of another session and through this ID he is able to join that session and works under another identity

Session management – cross-site request forgery (CSRF)

- cross-site request forgery, aka. a one-click attack or session riding
- session hijacking requires that the attacker has stolen or guessed ID
- session riding does not require knowledge of the session ID

- attacker convinces a user to send him constructed request
  - convinces him to click on link he created (e.g. in discussion)
  - images can send requests too (automatically)

- possible protection:
  - not using cookies to store session ID
  - place session ID directly into the URL
  - randomly generate and embed authentication tokens to each action URL

Access control

- authenticated users can have access only to certain parts of the site
- attacker can gain unauthorized access using programmers wrong assumption about how users will interact with the application
- URL tampering – altering parts (especially the GET parameters) of existing URL
  - if the attacker sees URL in the form:
    https://www.app.sk/zaznam.php?id=1234
  - he can try to enter a URL in the form:
    https://www.app.sk/zaznam.php?id=1235
- Forceful browsing – creation of new URLs
  - attacker can try to enter a URL like:
    https://www.app.sk/zaznam.php.old
  - or it may try to enter, for example:
    https://www.app.sk/admin.php
  - server relies on fact: client can request only URL sent to him

Input validation – different approaches

- reject dangerous inputs
  search for known patterns used in attacks
- accept safe inputs (allow specified harmless inputs)
- input sanitization
  - eliminate potential dangerous character sequences (e.g. <script>)
  - '<scr<script>ipt>' ?
  - '+ADw-script+AD4-' ?
- secure input processing
  - parameterized database queries
- semantic checks
  - input data are syntactically correct (eg account number)
  - but not semantically (not my account number)
- input must be validated on the server side
  - although for a better UX checks are done also on the client side
  - best to revalidate in each part of the application

Code injection

- each language has a specific syntax and specific control characters
- web applications often use different languages
  (e.g.: SQL, HTML, JavaScript, XML, HTTP, . . .)
- unexpected side effects can emerge if untreated data from the client are inserted into the program in any of these languages
  - SQL injection
  - XPath injection
  - HTML injection (markup injection)
    CSRF, XSRF: cross-site request forgery
  - JavaScript injection
    XSS: cross-site scripting

SQL injection

- select * from Pouzivatelia where Meno='Janko Hrasko'
- input: ' or ''='

  select * from Pouzivatelia where Meno='' or ''=''
- input: ' and 1=0 union all select * from Tabulka--

  select * from Pouzivatelia where Meno='' and 1=0 union all select * from Tabulka--'
- sometimes the application returns only part of the result or return information only indirectly
  - error message
  - request processing time

SQL injection – real example :-)

Gijs in 't Veld
@gintveld

Sledovať

Great to see that my name still causes SQL errors and that errors thrown are so hacker friendly. ;-) #integrate2016

Invalid query: You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near 't Veld', NOW()), ('cc89fdd01ea0', 'User-Profile', ':=', 'free750', '', '', NOW(' at line 1 Whole query: INSERT INTO newusers (username, attribute, op, value, callingstationid, displayname, created_at) VALUES ('cc89fdd01ea0', 'Cleartext-Password', ':=', '70358542', 'cc-89-fd-d0-1e-a0', 'Gijs in 't Veld', NOW()),('cc89fdd01ea0', 'User-Profile', ':=', 'free750', '', '', NOW())

XPath injection

- //user[name/text()='Janko Hrasko' and password/text()='passwd']/account/text()
- input: ' or 1=1 or ''='

  //user[name/text()='' or 1=1 or ''='' and password/text()='passwd']/account/text()
- input: NoSuchUser'] | P | //user[name/text()='NoSuchUser

  //user[name/text()='NoSuchUser'] | P | //user[name/text()='NoSuchUser' and password/text()='passwd']/account/text()

  first and last part return nothing $\Longrightarrow$ result is the evaluation of P
- in XPath it is not possible to apply access rights (within tags)
  - therefore, the second way can get the entire XML document
  - on the other hand, during the SQL injection, attacker will be restricted solely to the tables where a given application has access

Session cookie

- HttpOnly not set
  - this property prevents client-side scripts from accessing the cookie
- Overly broad session cookie domain/path
  - attack through other applications sharing the same domain/path
- Persistent session cookie
  - remains valid even after a user closes browser
  - often used as part of a "Remember Me" feature
- not sent over SSL
  - Modern web browsers support a secure flag for each cookie.
    - If the flag is set, the browser will only send the cookie over HTTPS.
  - If an application uses both HTTPS and HTTP without the secure flag,
    - then cookies set during an HTTPS request will also be sent during subsequent HTTP requests.
- Disabled
  - If the program does not use cookies to transmit session ID,
    - then ID is transmitted as an request parameter or as part of the URL.
  - Leaves the door open to session fixation and session hijacking attacks.

System misconfiguration

- Least privilege violation
- Password management
  - Empty, hard-coded or default password
  - Password in config file
    - Unprotected
    - Protected by weak cryptography
- System information leak
- Missing custom error page
- Improper use of SSL
  - Weak ciphers or protocols
  - Self-signed certificates

Insecure programming

- Buffer overflow, integer overflow, format string
- Exception handling
  - Empty catch block, i.e. ignoring exceptions
  - Overly-broad catch block (promotes complex error handling code)
- Insecure randomness
- Failure to begin a new session upon authentication
- File access race condition (TOCTOU)
- Pointers
  - Double free
  - Use after free
  - Memory leak
  - Null dereference
- Leftover debug code

Insecure compiler optimization – memset

- Secret data is stored in memory.
- The secret data is cleared from memory by overwriting its contents.
- The source code is compiled using an optimizing compiler.
- The compiler identifies and removes clearing as unnecessary because the memory is not used subsequently.

Insecure compiler optimization – memset example

```
1 #include <stdio.h>
2 #include <string.h>
3
4 void clear(void) {
5   char pwd[257];
6   gets(pwd);
7   memset(pwd, 0, sizeof(pwd));
8   puts(pwd); // we will comment this line later
9 }
```

Insecure compiler optimization – with puts

```
1  clear():                                    # @clear()
2          pushq   %rbx
3          subq    $272, %rsp                  # imm = 0x110
4          leaq    (%rsp), %rbx
5          movq    %rbx, %rdi
6          callq   gets
7          xorl    %esi, %esi
8          movl    $257, %edx                  # imm = 0x101
9          movq    %rbx, %rdi
10         callq   memset
11         movq    %rbx, %rdi
12         callq   puts
13         addq    $272, %rsp                  # imm = 0x110
14         popq    %rbx
15         retq
```

Insecure compiler optimization – without puts

```
1  clear():                                    # @clear()
2
3          subq    $264, %rsp                    # imm = 0x108
4          leaq    (%rsp), %rdi
5
6          callq   gets
7
8
9
10
11
12
13         addq    $264, %rsp                    # imm = 0x108
14
15         retq
```

Insecure compiler optimization – pointer arithmetic

- An arithmetic overflow on a pointer is undefined behavior.
- Array bounds check could be mistakenly optimized out.
- If an array bounds check involves computing an illegal pointer and then determining that the pointer is out of bounds, some compilers will optimize the check away.

Insecure compiler optimization – pointer arithmetic

- Following C code:

```
1 #include <stdio.h>
2
3 void wrap(unsigned long len) {
4   char *buf;
5
6   if (buf + len < buf)
7     puts("OK");
8 }
```

- translates to:

```
1 wrap(unsigned long):                # @wrap(unsigned long)
2         retq
```

- Even if $buf + len < buf$ is possible (check for arithmetic overflow) call to puts is optimized away.

Handling attacks

- developers must assume that application will be directly targeted by dedicated and skilled attackers
- handling attacks requires:
  - handling errors
  - audit logs
  - alerting administrators
  - reacting to attacks

Handling attacks – handling errors

- it is virtually inevitable that some unanticipated errors will occur
  - it is difficult to anticipate every possible way in which a malicious user may interact with the application
- application should handle unexpected errors
  - recovering from error
  - presenting an appropriate error message
    - administrator – detailed
    - user – brief

Handling attacks – audit logs

- invaluable source of information when investigating intrusion attempts
- logs help to detect:
  - what happened
  - what vulnerabilities were exploited
  - which data have been stolen
  - the attacker's identity

- the following events should always be recorded:
  - successful and failed login
  - change of password
  - important transactions (e.g. funds transfers)
  - access attempts that are blocked by the access control mechanisms
  - requests containing known attack strings
    - indicate overtly malicious intentions

Handling attacks – alerting administrators

- administrators can solve the problem online (instead of retrospective offline analysis)
  - block attackers IP addresses
  - set up a trap
- warning should come if:
  - untypical usage
    - a large number of requests from a single IP in a short time
    - a large number of transfers of funds from different accounts into a single account
  - client responses / requests contains
    - known attacks patterns
    - changed hidden data (from normal users)

Handling attacks – reacting to attacks

- system can react automatically by
  - attackers session termination
  - slowing the response to the attacker's IP address
  - blocking the attacker's IP address
  - warning attacked user
- more time for administrators