Generic security improvements

Michael Sonntag
Institute of Networks and Security

Check for evasion

- "Be liberal in what you accept" might be a practical advice for functionality, but is very bad for security!
- Try not to accept common evasion techniques, i.e. do not filter such things out or simply stop processing there… (but see later)
- Examples:
  - □ Special characters: \0, NULL, ', ", \, /, #, TAB…
  - □ Encoding: use "as-is" or decode **once** according to the **exact** method it **should** be encoded in
  - □ Superfluous data: canonicalization of all things
    - ● If users have entered it, it might be "strange"; but everything created automatically should be identical to the canonical version
    - ● Additional parameters which are not expected → don't just ignore them, log/create error
  - □ Premature ending/shuffling
    - ● "HTTP/1.1" at the beginning instead of at the end
    - ● Missing parts filled by default values (unless explicitly optional!)

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 an attacker, who is connecting to your server
- Take care: unless the 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**!

State management

- 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”!
  - If using client-side state: encrypt&”sign” and verify&validate it!
- Do not put any data into cookies or session IDs
  - These should only be a single large random number, nothing else
  - Verify it is a valid session ID
    - If unknown → ignore it and create a new session with a different ID
- Ensure state data is deleted on the server soon after timeout

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!
  - Always 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!”
- “Zero trust” principle: security = don’t trust **anyone**
- This includes:
  - Web requests (=browser input; GET and POST)
    - Including HTTP headers!
  - Environment variables (often used for configuration!)
  - Cookie data
  - Configuration data (from files, databases etc)
  - 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
  - Especially frameworks might store **all** input→ but you usually **verify** only what is **expected** to be there!

Input validation: Black- or Whitelists?

- Always use a **positive** specification (=whitelist) if possible at all
  - 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
    - Can be static lists or regular expressions (check them to see they only match what you expect them to!)
- Attention: generic security devices (e.g. content inspection on firewall) can typically use **negative** specifications only!
  - Insufficient alone; only application knows exactly what it expects!

Sanitizing input

- Change user input into an acceptable form

1. Sanitizing: remove any forbidden characters/all characters not explicitly allowed (black-/whitelisting on character level)
   - Result: all “problems” have been removed (=blacklisting)…
     - Eliminate, translate, encode
   - See before: ideally two sets
     - “Dangerous” things → Stop and produce error
     - “Strange” things → Remove, log, and continue
2. Canonicalization: change into the single “standard” form
3. Black-/Whitelisting: check if it is as a whole “acceptable”

Sanitizing input: Telephone number

- ■ Sanitizing
  - □ +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!
- ■ Canonicalization (note: very difficult here!)
  - □ Result (all of them!): +4373281547 (according to ITU-T E.164)
    - ● Note: DID (“Durchwahl”) got lost here (ÖNORM A1080: “…815-47”!)
- ■ Black-/Whitelisting
  - □ Check whether it is an acceptable country (e.g. +43) and “region” (e.g. no 0810, 0820, 0821, 0900, 0901, 0930, 0931, 0939, 118)

https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-E.164-201011-I!!PDF-E&type=items
https://www.rtr.at/de/tk/TKKS_MWD

Sanitizing input

- Take care of “sanitation loops” → they might be needed!
- Example: “Just remove all ‘<script’ once to be secure!”
  - Input: “<scri<scriptpt” → Output after one step: “<script”
- The loop is required to ensure that after removal the remaining text is still acceptable
  - But take care of double decoding vulnerabilities
  - So only **filter** in the loop, but do not **decode** in the loop
    - **Except** where you cut something out: “%<script03” → “%03”

Input validation: Some rules

- Hidden fields: should not be used
  - State should be on server!
  - Use hidden fields only for better user interface!
- URLs: don‘t send data with it, except for navigation
  - If you must, use URL en-/decoding
- HTML: **always** encode **all** data on output
  - <? print …?>, <%= (JSP), <%- (EJS)… → dangerous!
    - Careful: <%= is harmless (=escaped) in EJS, but dangerous in JSP (raw)
- Validation patterns should always stem from you
  - XSD, DTD, RegEx etc → 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!
  - Take care of different encodings, e.g. URLEncoding or Unicode (=non ASCII) characters (but: internationalization!)

“Dangerous” characters

- Characters which, when occurring in user input, should always be viewed with suspicion (see already at evasion!):
    - Note: they might be completely legal and/or completely harmless in that context, but they must be investigated!
  - Null: NULL, \0, %00, \0x00, 0xff
  - Linebreaks and whitespaces: LF, CR, CRLF, TAB, SPACE, other whitespaces (VTAB, NBSP…)
  - Quotation marks: “ ‘ ´ `
  - Brackets: < > ( ) [ ] { }
  - Characters with special meaning: + & * ? $ % @ . , ; : / \ | =
- Usually harmless: a-z, A-Z, 0-9, umlauts (äöüß…), characters with accents (áèû…)

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 is simpler to implement!
- Exception: when the verification requires “secret” data
  - E.g. username and password
    - Length, presence… → client side
    - Length, presence… **+ validity** → server side

What to look out for

- ■ Common attack attempts for URL parameters/form input
  - □ Existing filename: dumping source code, configuration 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
    - But: users don't need local/domain accounts
  - Impersonating: create a new thread with the identity of the authenticated user and server request from there
    - Can access the file system etc as if he/she were logged on directly
    - Subverting 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!
  - Impersonation of a “no-rights” user can be helpful

Security logs

- Similar as errors also often normal actions should be logged
  - To have evidence later: normal use **and** attacks
  - As input for detecting anomalies
- What should be logged? All security-relevant activities
  - Profile changes: change of username, password, other important fields, permissions, recovery-information (e.g. E-Mail address) etc
  - Modifying other users – if not included above!
  - Add/delete users
- Detail of logging: as much as possible:
  - At least date, time, user who originated the change (source IP, username, token identifier etc)
  - But:
    - Denial of service, attack target (e.g. if old&new password stored!)
    - Legal requirements: might be forbidden/require special protection, time limits (GDPR – deletion is mandatory) etc
      - Other way: if history is required by application, do **NOT** misuse logs for this!

Web Application Firewall - WAF

- Special case of Application Level Firewall/Gateway:
  - Traffic is inspected on the application layer, not on the network/transport layer
- Works independently from the application
- Usually a separate computer/appliance, a plugin to a “normal” firewall, or “module” on the web server
  - Can also be cloud-based: all traffic goes to the cloud first, and only then to your company/servers
- Requires:
  - Complete understanding of HTTP (and ideally all related specifications, e.g. Flash, PDF, Java)
  - Unencrypted content: must work on/as a proxy or on webserver (or knows private server key)
  - Stateful inspection: not only on network but also on application layer. It must therefore “store” cookies to recognize sessions.

Web Application Firewall - WAF

- Three main approaches:
  - Signature-based: known attacks are identified
  - Heuristics: “Looks like SQL code → block it (perhaps injection)!”
  - Scanner: manually/automatically crawling and scanning the application and storing the requests and responses.
    - Everything not matching these will be marked as “attack” after this learning phase
- Drawbacks:
  - Significant number of false alarms
  - Requires extensive customization of the rules to be effective
  - Works on layer 6.5: HTML/HTTP. The actual application logic is typically **not** part of the rules.
    - Statically possible: learning through observation
    - Dynamically (“ways through the site”): theoretically possible, but doesn’t seem to be used (yet) by commercial products

Attacks against WAFs

- Fingerprinting: **is** there a WAF?
  - Passive: listening for traces
    - Many WAFs introduce additional headers or cookies, or change the server header
      - Other options: special status codes, or very fast termination of connection if request is not sent immediately
  - Active: sending probes and identify WAF through its activities
    - E.g. send something "slightly" wrong → No block, but potentially an internal alert; then send something "very" wrong and look for a different response
    - Timing between success/block may also provide information
- Direct attacks are very rare, but circumvention is common
  - Add additional data to get around/through
    - E.g. "DELETE /* */ FROM"
  - Character encodings
  - Inspect signatures and modify attacks slightly

WAFs: Additional protection

- Removing various headers
- Rewriting error messages
- Signing/encrypting cookies
- Replacing multiple cookies with a single unique ID
- Tying cookies to TLS parameters (=TLS sessions)
- Brute force/DDoS protection (e.g. size/rate limiting)
- Signing/encrypting URLs (to avoid manipulation/param use)
- Request flow validation (only allow links appearing on previous pages)
- Additional logging
- Response filtering: credit card/social security numbers…

Problem areas of WAFs

- Client-side JavaScript code
- Integrating data from third sources
- AJAX/REST communication may be “difficult”: this is not HTML, but can be anything (content) and its format is vary (XML, JSON, text…)
- Dynamically changing content
- No protection against logic flaws in the application
- Web application = outside perimeter → what about insiders?
- One more single point of failure
- Another device/software to manage
  - □ May have its own vulnerabilities
- Understandability: compare these two

  -
    ```
    iptables -A INPUT -p tcp -s 0/0 -d 0/0 --destination-port 80 --syn -j ACCEPT
    ```

  -
    ```
    SecRule REQUEST_COOKIES|REQUEST_COOKIES_NAMES|REQUEST_FILENAME|ARGS_NAMES|ARGS|XML:/* "(?i:(?i:\d[\"'`´’‘]\s+[\"'`´’‘]\s+\d)|(?:^admin\s*?[\"'`´’‘]|(\/\*)+[\"'`´’‘]+\s?(?:--|#|\/\*|{)?)|(?:[\"'`´’‘]\s*?(x?or|div|like|between|and)[\w\s-]+\s*?[+<>=(),-]\s*?[\d\"'`´’‘])|(?:[\"'`´’‘]\s*?[^\w\s]?=\s*?[\"'`´’‘])|(?:[\"'`´’‘]\W*?[+=]+\W*?[\"'`´’‘])|(?:[\"'`´’‘]\s*?[!=|][\d\s!=+-]+.*?[\"'`´’‘(].*?$)|(?:[\"'`´’‘]\s*?[!=|][\d\s!=]+.*?\d+$)|(?:[\"'`´’‘]\s*?like\W+[\w\"'`´’‘(])|(?:\sis\s*?0\W)|(?:where\s[\s\w\.,-]+\s=)|(?:[\"'`´’‘][<>~]+[\"'`´’‘]))"
    ```

Example: http://blog.sec-consult.com/2012/10/are-web-application-firewalls-useful.html

When to use WAFs

- The web application is of low quality/insecure
- PCI DSS: penetration testing or WAF - select one (mandatory)!
- Changes to the web application are slow (long cycles)
- Developers or source code are unavailable
- No automated tests for the application
- Time and funds available for tuning the WAF by experts
- Useful (common?) scenario:
  - Old application, no source code, problem detected
  - Use the WAF to protect the application by preventing the attacks for this specific vulnerability
  - Used as an “interception device”, not really as a “guard”

JOHANNES KEPLER
UNIVERSITÄT LINZ

INSTITUTE
OF NETWORKS
AND SECURITY

https://www.ins.jku.at

THANK YOU FOR YOUR ATTENTION!

Michael Sonntag
michael.sonntag@ins.jku.at
+43 (732) 2468 - 4137
S3 235 (Science park 3, 2nd floor)