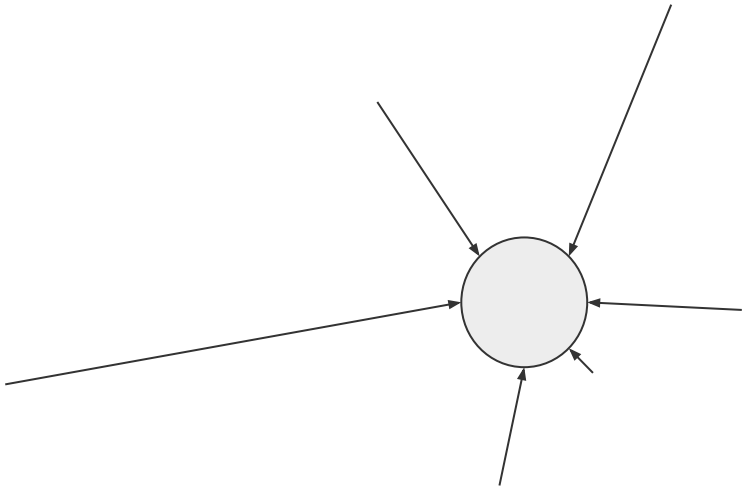
Input/Output Management

Was ist Input?

- Daten von “außen”
  - Jede Form von nicht selbst erzeugten Daten
- Z.B.
  - Nutzereingaben in einem Formular (z.B. Login)
  - Inhalt einer gelesenen Datei auf dem Dateisystem
  - Response einer Webservice Abfrage
  - Metadaten des Requests
    - Angefragte URL
    - Header Felder und ihr Inhalt

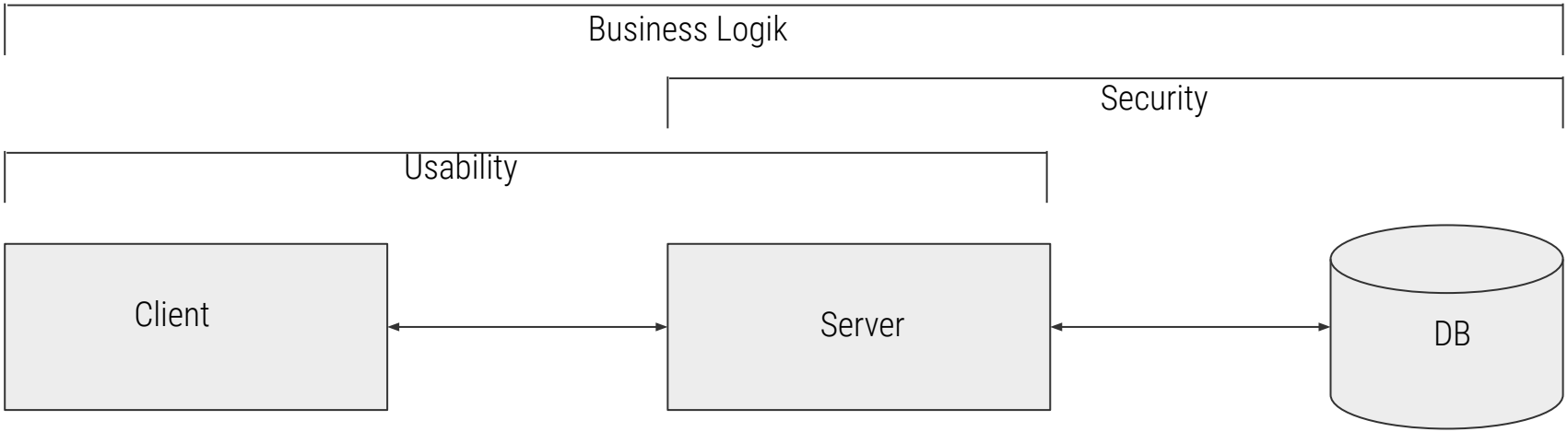


Warum Input Management?

- Eingaben müssen einer gewissen Form genügen -> **Können wir sie nutzen?**
  - Bestimmte Menge an Daten
  - Bestimmte Semantik an Daten (passend zum Use Case)
  - Klassiker: Passwort-Regeln
- Homogenität von gespeicherten Daten -> **Sind sie aufwändig zu nutzen?**
  - Bestimmte Syntax von Daten (passend zum Datentyp)
  - Erlaubt lange Nutzung von Daten
  - Erleichtert Ausgabe und weitere Verwendung
- Sicherheitsaspekte -> **Können sie uns schaden?**
  - Angriffe durch Hacker/etc. hauptsächlich über Fehler in Webanwendungen

Wo Input Management?

- Abwägen: Usability, Business Logik, Security oder alles drei?
- Mehrstufiges System

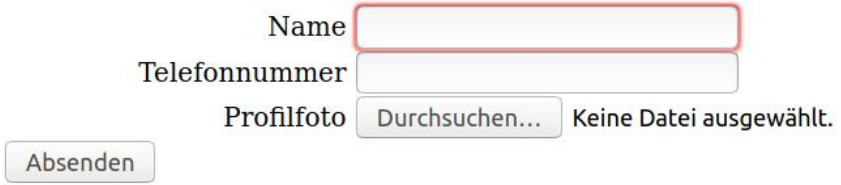


Arten von Input - Formulareingaben

- Hauptquelle von Nutzereingaben für klassische Webanwendungen
- Nativ in HTML und PHP verankert
  - Aber auch über Javascript/AJAX nutzbar
- Kombination aus umspannenden *form* Element und beliebig vielen *input* Elementen
- Vielzahl unterschiedlicher *input* Elemente
  - Repräsentieren Datentypen im Sinne der Businesslogik
- Daten über GET oder POST Request an Serverseite
  - Über $_POST/$_GET/$_REQUEST auslesbar (Kapselung in Request Objekt sinnvoll)

Arten von Input - Formulareingaben

```
<form action="contact/add" method="post">
    <label for="form_name">Name</label>
    <input type="text" id="form_name" name="name" required>
    <br/>
    <label for="form_phone">Telefonnummer</label>
    <input type="tel" id="form_phone" name="phone">
    <br/>
    <label for="form_avatar">Profilfoto</label>
    <input type="file" id="form_avatar" name="avatar">
    <br/>
    <input type="submit" value="Absenden">
</form>
```




```
array (
    'name' => 'Bernhard Wick',
    'phone' => '123456789',
    'avatar' => 'ich.jpg',
    );
```

Wie Input Management? - Validieren

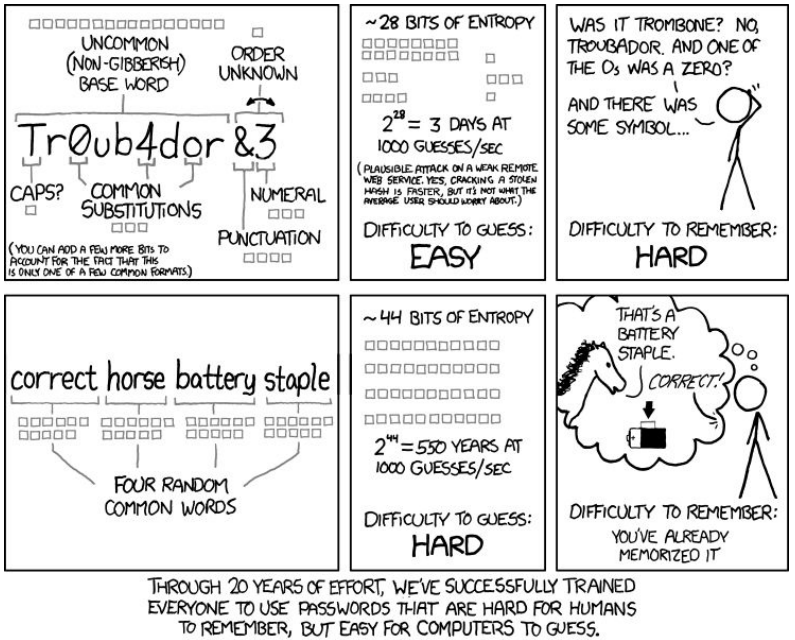
- Prüfung ob Inhalte semantisch und syntaktisch korrekt sind
  - Z.B. Format einer E-Mail Adresse, Zahlen von 1 bis 10
- Clientseite über natives HTML5 und/oder Javascript (jQuery, validate.js, Bouncer.js)
- Gute PHP Bibliotheken: respect/validation, beberlei/assert, ronanguilloux/IsoCodes

```html
<form action="contact/add" method="post">
    <label for="form_name">Name</label>
    <input type="text" id="form_name" name="name" required>
    <br/>
    <label for="form_email">Profilfoto</label>
    <input type="email" id="form_email" name="email" required placeholder="Gültige E-Mail Adresse">
    <br/>
    <label for="form_age">Alter</label>
    <input type="number" size="6" min="1" max="160" value="18" id="form_age" name="age">
    <br>
    <label for="form_site">Website</label>
    <input type="url" id="form_site" name="website" pattern="https?://.+">
    <br/>

    <input type="submit" value="Absenden">
</form>
```

```php
v::key( reference: 'name', v::allOf(v::notBlank(),v::stringType()))
    ->key( reference: 'email', v::email())
    ->key( reference: 'age', v::allOf(v::minAge( age: 1),v::minAge( age: 160)))
    ->key( reference: 'website', v::url())
    ->validate($request->getQueryParams());
```

Spezialfall - Passwörter

- Viele Passwortregeln sind wenig effizient
- Wichtig ist vor allem Entropie
  - Länge gewinnt vor Zeichensatz
- Kein Sonderzeichen- und Zahlen-Zwang
  - Resultiert in Leet-Speak (maschinell erzeugbar)
- Passwörter sollten leicht zu merken sein
  - Ansonsten droht Mehrfachnutzung von Passwörtern
- Keine Einschränkung des Zeichensatzes
  - Auch ein Leerzeichen ist ein Zeichen
- Eine Mindestlänge (>= 8) ist Pflicht
  - Aber nicht zu lang (Verarbeitungsdauer)



Wie Input Management? - Sicherer Aufbau

- **Man kann nie alles filtern**
- Jede Implementierung soll davon ausgehen mit unsicheren Daten arbeiten zu müssen
- Vermeiden Eingaben in Ausgaben zu nutzen
- Per default sichere Implementierungsweise wählen
- Z.B.
  - Stored Procedures für Datenbankabfragen (z.B. gegen Injection)
  - Kein HTML aus Eingaben generieren
  - Request-Restriktionen wie same-origin policy/Cross-Origin Resource Sharing sinnvoll nutzen
  - Mehrschichtige Authentifizierung und Autorisierungsmodelle nutzen
  - Interne Code Execution verhindern (kein eval(), saubere Deserialisierung, …)
  - *Public* Ordner als Document Root

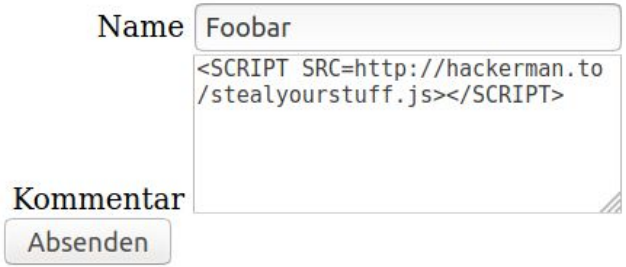
Threats - Injection

- Oft in Form von SQL-Injection
  - Aber auch andere Injections
- Manipulation von Abfragen durch ungefilterte Eingaben
  - Z.B. Verändern eines Datenbankqueries
- Erlaubt je nach Abfrage fast beliebige Aktion des Angreifers
  - Abfrage von zusätzlichen Daten
  - Manipulation von Daten
- **#1 auf der OWASP Top 10**

```
$query = 'SELECT * FROM users WHERE id="' . $_REQUEST['id'] . '"';

// http://example.com/app/users?id=" or "1"="1
// => SELECT * FROM users WHERE id="" or "1"="1"
```

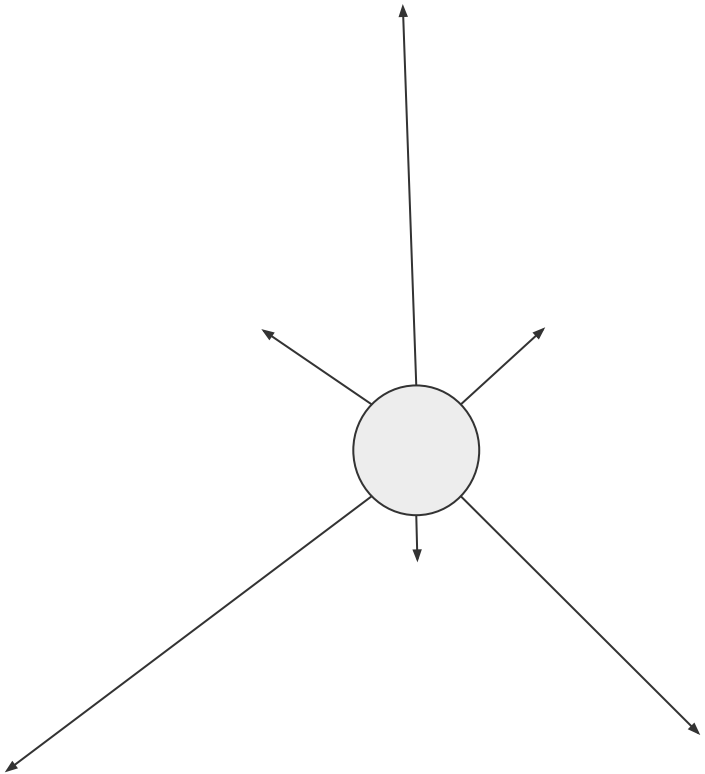
Threats - XSS

- **Cross S**ite **S**cripting
- Einfügen von manipulierten Elementen in den DOM
- Erlaubt Angreifer beliebige Aktion mit DOM
  - JS Einbinden um Daten zu stehlen
  - Verändern von Informationen
  - ...
- Kann passieren wenn Input zum Dynamisieren der Seite verwendet wird
  - Z.B. Kommentar-Funktion einer Anwendung
- Lösung: Ausgabe filtern
- Mehr in OWASP Cheatsheets



Was ist Output?

- Alle Daten die unsere Anwendung verlassen
- Z.B.
  - Statische Dateien wie CSS und Javascript
  - Dynamische Inhalte im DOM
  - Log Dateien
  - Webservice Ausgaben
  - Status Codes und Header Felder
- Bei der Behandlung muss unterschieden werden
  - Daten die in DOM eingebettet werden
  - Statischem Output



Warum Output Management?

- Ausgabe muss einer gewissen Form genügen -> **Können wir sie nutzen?**
  - Passt Syntax zum angedachten Verwendungszweck
- Bestimmungszweck von Daten -> **Ist es angebracht sie zu nutzen?**
  - Bestimmte Daten sollten nur in bestimmten Situationen ausgegeben werden
  - Manche Daten sollten Anwendung nie verlassen (z.B. Passwort Hashes)
- Sicherheitsaspekte -> **Können sie uns schaden?**
  - Woher stammen die Daten initial?
  - Enthalten sie je nach Kontext schädlichen Inhalt?

Wie Output Management? - Media (MIME) Types

- Standardisierte Notation zur Beschreibung von Request/Response Inhalten
  - Z.B. *text/css*, *video/mp4*, *image/png*, ...
- Client- und Anwendungs-Äquivalent von Dateiendungen
  - Erklären Browser wie Dateien zu interpretieren sind (**Achtung: Fehlerpotential**)
- Nutzung meistens in Headern als *Content-Type*
  - Webserver erledigt das für statische Dateien (übersetzt Dateiendung in Media Type)
  - Dynamische Inhalte muss Webentwickler richtig ausweisen
- Können Metainformationen enthalten
  - Z.B. Charset: *text/html; charset=UTF-8*
- Request gibt Hinweis auf akzeptierte Response Media Types in *Accept* Header

Wie Output Management? - Content Filter

- Betrifft sensible Informationen
  - Passwort Hashes
  - Datenbank IDs
  - Sensible Businessdaten (Bankverbindung, Wohnort, …)
- Oft ungewollt
  - Nicht gefangen Exceptions/Fehlermeldungen
  - Log Ausgaben
- Nur schwer generisch lösbar
  - Abwägen von Fall zu Fall
- Händisches Filtern so früh wie möglich
  - Im Idealfall nicht aus DB abrufen
    - Nur bestimmte Felder anfragen
    - Serializer mit Access Groups im ORM (z.B. jms/serializer) schaffen gute Basis

```php
try {
    $controller->execute($request, $response);
} catch (Exception $e) {
    $response->setStatusCode($e->getCode());
    if (APP_DEV_MODE) {
        $response->setBody($e->getMessage());
    } else {
        $view = new DynamicErrorView();
        $response->setBody($view->render(['reason' => $e->getCode()]));
    }
}

http_response_code($response->getStatusCode());
echo $response->getBody();
```

Wie Output Management? - Encodieren

- Erlaubt das Entfernen von eventuell unerwünschten und schädlichen Zeichen
- Zur Unterstützung der richtigen Darstellung/Verarbeitung von Inhalten
- Zum Verhindern der Interpretation durch Browser
- Stark abhängig vom Kontext!
  - Je nach Ziel innerhalb des Clients unterschiedliches Encoding
  - Z.B.
    - *json_encode()* zur Ausgabe in JSON
    - *htmlentities()* und *htmlspecialchars()* zur Darstellung von HTML Code (wird nicht interpretiert)
    - *urlencode()* zur Darstellung von URLs in URL Kontext

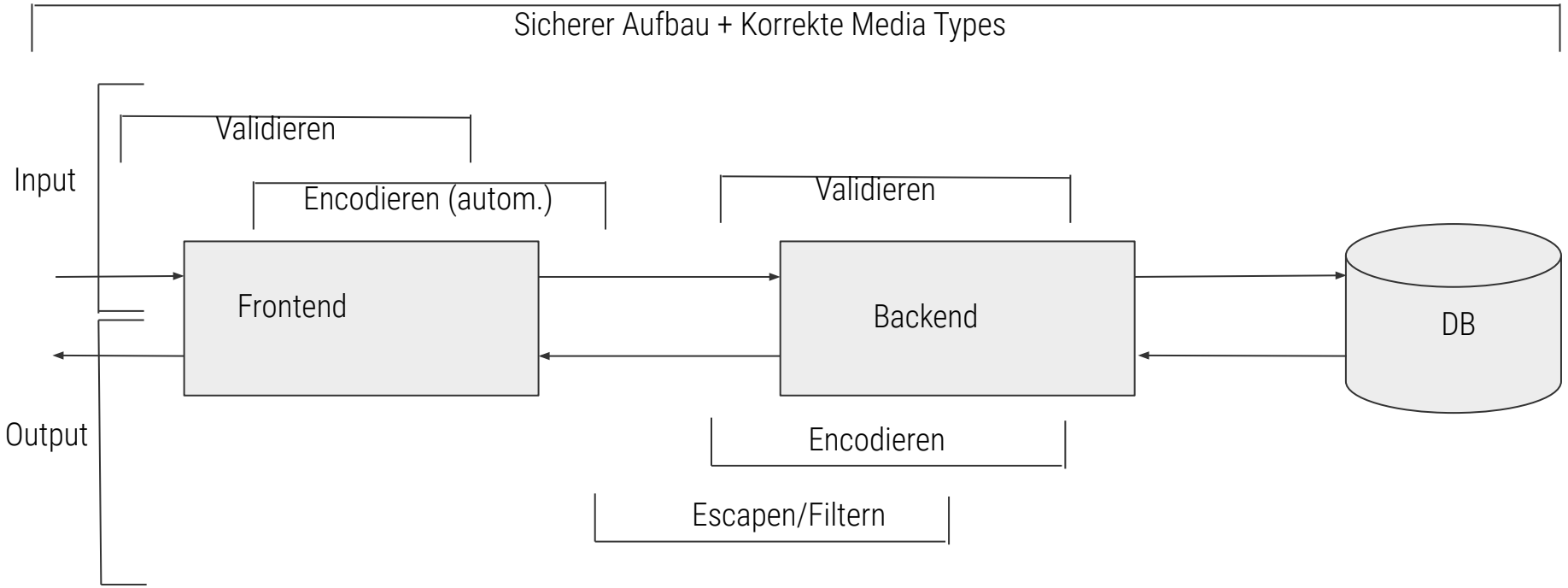
```php
$str = "Ein 'Anführungszeichen' ist <b>fett</b>";

// Ein 'Anf&uuml;hrungszeichen' ist &lt;b&gt;fett&lt;/b&gt;
echo htmlentities($str);
```

Wie Output Management? - Escaping

- Erlaubt das Bereinigen von Daten um schädliche Elemente zu entfernen
- Oft im Zusammenspiel mit Encoding
- Wirksamer Schutz gegen XSS
- Auch von Kontext abhängig
  - Daher oft in View oder direkt im Template
- Whitelisting/Blacklisting möglich
  - Z.B. von HTML Elementen
- Wenn möglich als Default nutzen und explizit deaktivieren
- Kompliziert, aber als Bibliothek etabliert z.B. über ezyang/htmlpurifier

```
<p>
    Hello {{ user.username|escape('html') }}
</p>
<script type="text/javascript">
    alert("It's me {{ user.username|escape('js') }}");
</script>
```

Was jetzt wo?



Quellen:

- https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_A1-Injection
- https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_A7-Cross-Site_Scripting_(XSS)
- https://owasp.org/www-project-proactive-controls/v3/en/c5-validate-inputs
- https://owasp.org/www-project-proactive-controls/v3/en/c4-encode-escape-data
- https://owasp.org/www-project-top-ten/
- https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html
- https://www.the-art-of-web.com/html/html5-form-validation/
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types
- https://cheatsheetseries.owasp.org/
- https://github.com/ziadoz/awesome-php
- https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

Bildquellen:

- Password Strength xkcd Comic https://xkcd.com/936/