

Verarbeitung von GET- und POST-Daten

- **Wie verarbeitet man die Query-String-Daten?**

- Zum Glück dekodiert PHP die per GET oder POST übergebenen Daten für uns. Sie liegen in
 - `$_GET` alle per **GET** übertragenen Name-Wert-Paare
 - `$_POST` alle per **POST** übertragenen Name-Wert-Paare
 - `$_REQUEST` = `array_merge($_GET , $_POST)` // Vereinigung der Mengen
- Wenn gleichgültig ist, wie die Daten übertragen wurden, kann man `$_REQUEST` benutzen.
- Alle drei sind assoziative Arrays
 - um z.B. auf den GET-Parameter „**name**“ zuzugreifen, dient der Ausdruck
`$_GET[ 'name' ]`
 - Siehe <https://www.php.net/manual/de/language.types.array.php>
- Alle drei Variablen sind **superglobal**
 - d.h. man von überall auf sie zugreifen (also ohne „**global** \$_GET;“)
 - Siehe <https://www.php.net/manual/de/language.variables.superglobals.php>

Verarbeitung von GET- und POST-Daten

- **Sicherheit** (aus Sicht des Servers)
 - Die in \$_GET und \$_POST enthaltenen Werte stammen **von außen** (vom Benutzer des Webdienstes)
 - Es können **Fehler** passieren oder Sie können sogar zu **Angriffen** dienen
 - Man kann ihren **Inhalten** allgemein **nicht vertrauen!**
 - Parameter **könnten ungesetzt** sein, obwohl wir sie gesetzt erwarten
 - Sie könnten Inhalte enthalten, die zu **unerwünschten Effekten** führen
- **Also ...**
 - Benutzergenerierte Daten müssen allgemein immer so behandelt werden, als würden sie von einem **Angreifer** stammen.
 - Ausnahme: Daten die von einem vertrauenswürdigen Nutzer (z.B. Admin).
 - Angriffe über benutzergenerierte Daten können **auch indirekt** erfolgen
 - Z.B. durch Daten, die zwischenzeitlich in der Datenbank abgelegt wurden.

Verarbeitung von GET- und POST-Daten

- **Wie prüfen, ob Parameter (nicht) gesetzt sind?**
 - Wir erwarten, dass ein Parameter gesetzt wurde
 - ... z.B. da wir die URL `/login.php` nur aus einem Formular aufrufen, in dem Name und Passwort abgefragt werden:

```
<form action="/login.php" method=post>
  <input type=text      name=name      >
  <input type=password  name=password  >
  <input type=submit    name=login value="Login" >
</form>
```
 - In `/login.php` werten wir diese Daten aus:
 - ```
<?php
 if ($_POST['name'] == 'Tom' &&
 $_POST['password'] == '1234') {
 $user = 'Tom'; // erfolgreich eingeloggt ...
 }
?>
```
    - Der Benutzer hat aber die URL manuell eingegeben oder das Formular manipuliert. So sei z.B. `$_POST['password']` undefiniert.
      - Der Zugriff auf `$_POST['password']` erzeugt eine Warnung

# Verarbeitung von GET- und POST-Daten

- **Lösung 1:** Vorher prüfen

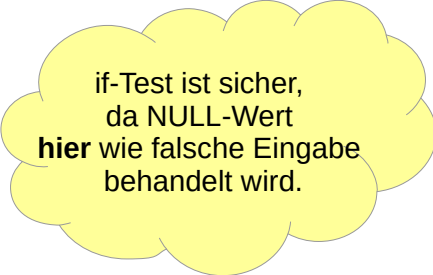
- mit `isset()` vorher prüfen

```
if (isset($_POST['name'], $_POST['password']) &&
 $_POST['name'] == 'Tom' &&
 $_POST['password'] == '1234')
{
 $user = 'Tom'; // erfolgreich eingeloggt ...
}
```

- **Lösung 2:** Sicherer Default-Wert

- Ist der Wert nicht gesetzt ist das Ergebnis NULL → *ist das sicher?*
- Optional: `@`-Präfix → Keine Warnung wenn nicht gesetzt

```
• if (@$_POST['name'] == 'Tom' &&
 @$_POST['password'] == '1234')
{
 $user = 'Tom'; // erfolgreich eingeloggt ...
}
```



if-Test ist sicher,  
da NULL-Wert  
hier wie falsche Eingabe  
behandelt wird.

# Schutz vor gefährlichen Inhalten

---

- Wie mit **gefährlichen Inhalten** von Daten umgehen?
  - z.B. werden übergebene Parameter oft in Ausgaben verwendet ...
    - Als Parameter für URLs (im HTML-Text)
      - Zum Schutz gegen unerwünschte Effekte kennen wir ja schon **urlencode** (s.o.)
    - Als Text-Ausgabe im HTML-Text
    - Als Teil einer Datenbank-Anfrage („SQL-Injection“ → später)
  - Die Idee beim Angriff ist immer, einen Parameter zu übergeben, der beim Einbau in eine Ausgabe zu **unerwünschten Effekten** führt.
  - Beispiel:
    - Unser **/login.php** soll bei einem erfolglosen Login-Versuch schon einmal den Benutzernamen wieder ins Formular übernehmen:

```
<form action="/login.php" method=post>
 <input type=text name=name
 value="<?php echo @$_POST['name']; ?>" >
 <!-- -->
</form>
```

# Schutz vor gefährlichen Inhalten

---

- **Unser (böser?) Nutzer versucht sich einzuloggen ...**

- 1. Versuch, Eintippen der URL (keine POST-Daten)

- `@$_POST['name']` liefert **NULL**, also konvertiert zu String den **leeren String**

```
<input type=text name=name value="" >
```

- 2. Versuch: Name „Peter“, (falsches) Password „4321“

- Wir liefern das Login-Formular mit `value="Peter"` zurück

```
<input type=text name=name value="Peter" >
```

- 3. Versuch: Name „**></html>**“

- Wir bauen das unbesehen in das Formular ein ...

```
<input type=text name=name value="></html>" >
```

„HTML-Injection“  
(später mehr dazu)

- Das wollten wir sicher nicht!!!

- Derartiges passiert überall, wo *unsichere Inhalte* in HTML eingebaut werden

- `<p>Hallo <?php echo $name; ?>, dein Passwort war nicht richtig.</p>`

# Schutz vor gefährlichen Inhalten

---

- **Wir wollen alle für uns gefährlichen Zeichen los werden ...**
  - „**<**“ und „**>**“, um zu verhindern dass Tags erzeugt werden (s.o.)
  - **Anführungszeichen**, um zu verhindern dass man HTML-Tag-Parameter-Strings vorzeitig beendet (s.o.)
    - u.U. genügt es, nur Doppelte „**“** zu entfernen, wenn man in für Tag-Parameter-Strings nur solche benutzt. Einfache „**'**“ können dann bleiben.
  - **&**-Zeichen, um zu verhindern dass diese als HTML-Zeichencode interpretiert werden.
    - z.B. in der Eingabe „ich messe volt**&**“, die in HTML anders interpretiert würde als vom Formular-Benutzer erwartet.
- **Genau das tun **htmlspecialchars()** bzw. **htmlentities()****
  - „**&**“ (Ampersand/kaufmännisches UND) wird zu '**&amp;**'.
  - „**“** (doppeltes Anführungszeichen) wird zu '**&quot;**'
  - „**<**“ (kleiner als) wird zu '**&lt;**'
  - „**>**“ (größer als) wird zu '**&gt;**'

# Schutz vor gefährlichen Inhalten

---

- Wirkung von **htmlspecialchars()**

- Wandelt die o.g. HTML-Sonderzeichen um in HTML-Codes

- `<?php`

- ```
$dangerous = "<a href='test'>Test</a>";  
$safe = htmlspecialchars($dangerous);  
echo $safe;
```

- ```
?>
```

- Ergebnis: `&lt;a href='test'&gt;Test&lt;/a&gt;`

- Bei Aufruf `htmlspecialchars($dangerous, ENT_QUOTES)` werden **auch einfache** Anführungszeichen umgewandelt

- Ergebnis: `&lt;a href=&#039;test&#039;&gt;Test&lt;/a&gt;`

- Diverse weitere Optionen (siehe php.net)

- Wirkung von **htmlentities()**

- Wandelt darüber hinaus z.B. Umlaute (,ä' → ,&auml;' ) um.

- Vorteilhaft bzgl. robustem Encoding – aber kein echter Sicherheitsgewinn
  - Achtung: Auch hier ist ggf. `htmlentities(..., ENT_QUOTES)` nötig (s.o.)

# Schutz vor gefährlichen Inhalten

- **Anwendung**

- Diese sichere Kodierung sollte **immer** angewandt werden, wenn **potentiell gefährliche Daten** in HTML-Seiten eingebaut werden.

- **Probleme**

- **Mehrfachanwendung** führt zu unerwünschten Effekten:  
Die HTML-Codes werden in der Webseite nur einmal dekodiert.

- ```
<?php
    $dangerous = "&";
    $safe      = htmlspecialchars($dangerous);
    $doublesafe= htmlspecialchars($safe);
    echo $dangerous . "\n" . $safe . "\n" . $doublesafe;
?>
```

- **Ergebnis:** &
 &
 & ; amp;

Browser-Anzeige: ☐ &
 &

Fehler (-Toleranz)

Schutz vor gefährlichen Inhalten

- Man kann auch die HTML-Tag-Typen beschränken

- `strip_tags( $str [ , $allowable_tags ] )`
- Entfernt alle Tags, außer sie sind explizit erlaubt
 - ```
<?php
$text = 'Das "A&0" ist <i>alles</i>.';
echo strip_tags($text) . "\n";
echo strip_tags($text, '<p>') . "\n";
?>
```
  - Ergebnis: `Das "A&0" ist alles.`  
`Das <b>"A&0"</b> ist alles.`
- Nützlich, wenn man z.B. in Blog-Beiträgen Fettschreibung (also b-Tags) erlauben will, aber nicht z.B. Bilder (img-Elemente) oder Links (a-Elemente).
- **Achtung:** Das ist nicht in allen Situationen ausreichend um Stabilität und Sicherheit zu garantieren

# (Debug-) Ausgabe von PHP-Variablen

- **print und echo**

- ... geben einen (**print** x) oder mehrere (**echo** x, y, z) Strings aus.  
→ ggf. Stringkonvertierung → strukturierte Daten werden nicht ausgegeben

```
$a = array('rot', 3=>'grün', 'b'=>'blau');
print $a;
```

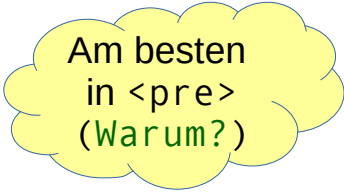
```
Array
```

- **print\_r** (→ [php.net](http://php.net))

- ... gibt strukturierte Daten **rekursiv** aus

```
print_r($a);
```

```
Array
(
 [0] => rot
 [3] => grün
 [b] => blau
)
```



Am besten  
in <pre>  
(Warum?)

- Optional auch zum Abspeichern in eine Variable

```
$a_as_text = print_r($a, true); // keine Ausgabe
```

# (Debug-) Ausgabe von PHP-Variablen

- **var\_dump** (→ [php.net](http://php.net))
  - Gibt strukturierte Daten **rekursiv** und mit **Typangaben** aus

```
$a = array('rot', 3=>'grün', 'b'=>'blau');
var_dump($a);
```

```
array(3) {
 [0]=>
 string(3) "rot"
 [3]=>
 string(5) "grün"
 ["b"]=>
 string(4) "blau"
}
```

- Tipp: Abfangen der Ausgabe von **var\_dump** als Wert
  - **Trick:** Ausgabe abfangen mit **ob\_start** + **ob\_get\_clean** (→ [php.net](http://php.net))

```
$a = array('rot', 3=>'grün', 'b'=>'blau');
ob_start();
var_dump($a);
$a_as_text = ob_get_clean();
```

} // Ausgabe der 3  
// Zeilen abfangen  
// und zurück liefern

# (Debug-) Ausgabe von PHP-Variablen

- **print\_r** oder **var\_dump** in HTML-Ausgaben benutzen
  - Wunsch:
    - Ausgabestruktur zu erhalten (Einrückung, Umbruch)
    - Sichere Ausgabe (keine Interpretation von HTML-Steuerzeichen)
  - Lösung:

```
<pre>
<?php
 $a = array('rot', 3=>'grün', 'b'=>'bold');
 echo htmlspecialchars(print_r($a, true));
?>
</pre>
```

```
<pre>
Array
(
 [0] => rot
 [3] => grün
 [b] => bold
)
</pre>
```

# Ausgabe aller PHP-System-Variablen

- Funktion **phpinfo()**

- Gibt diverse Systemvariablen von PHP in Tabellenform aus
  - → [php.net](http://php.net)
- Ausgabe ist sicher bzgl. Injections
- Vorsicht aber vor öffentlicher Ausgabe
  - da viele Informationen Angreifern helfen können



System	Linux scilab-0100 5.15.152-1-pve #1 SMP PVE 5.15.152-1 (2024-04-29T07:31Z) x86_64
Build Date	Feb 23 2023 12:43:23
Server API	Apache 2.0 Handler
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/etc/php/7.4/apache2
Loaded Configuration File	/etc/php/7.4/apache2/php.ini

.....

- **Demo:** 5... in [http://scilab-0100.cs.uni-kl.de/1\\_basics/](http://scilab-0100.cs.uni-kl.de/1_basics/)

# Sichere Ausgabe von PHP-System-Variablen

---

- **Beispiel:** Wir wollen zum Testen Variablen anzeigen
  - Sie sollen auf der Webseite sicher angezeigt werden
  - Beispiel: `$_GET`
    - ```
<?php
    echo "<h2>_GET</h2>\n";
    $printed = print_r($_GET, true);
    echo '<pre>' . htmlspecialchars($printed) . '</pre>';
?>
```
 - Wir benutzen `print_r($var, true)` um die **Ausgabe als String** zu erzeugen
 - Wir benutzen `htmlspecialchars()` um Sonderzeichen abzufangen (s.o.)
 - Analog: `$_POST`, `$_SERVER`, `$_COOKIES`, ... (s.u.)

Sichere Ausgabe von PHP-System-Variablen

- Verbesserung: **Test-Ausgabe der Variablen in Schleife**

- Diverse Variablen sollen sicher angezeigt werden

- ```
<?php
foreach (array('_GET', '_POST', '_COOKIE',
 '_SESSION', '_SERVER') as $varname) {
 if (isset($$varname)) {
 echo "<h2>" . $varname . "</h2>\n";
 $printed = print_r($$varname, true);
 echo '<pre>' . htmlspecialchars($printed) . '</pre>';
 }
}
?>
```

- Mit **`$$varname`** bekommen wir den Inhalt der Variablen mit Namen **`$varname`**

- *Wenn einige Teile nicht erscheinen ist vielleicht folgender Workaround nötig:*

- Vor die foreach-Schleife die folgende Zeile einfügen:

- `@$_SERVER; @$_REQUEST; // Workaround, macht ggf. Variablen sichtbar`

- **Demo:** 6... + 7... in [http://scilab-0100.cs.uni-kl.de/1\\_basics/](http://scilab-0100.cs.uni-kl.de/1_basics/)

# Applikationsstruktur

---

- **Struktur einer PHP-Applikation**

- Eine HTML- oder PHP-Seite anlegen, die
  - Inhalte und Daten ausgibt
  - **Requests** (ggf. mit Parametern) **erzeugt**
    - Links auf andere URLs (GET-Requests)
    - **Formulare** (GET- oder POST-Requests)
- Eine PHP-Seite anlegen, die solche **Requests verarbeitet**
  - Prüft ob Daten übergeben wurden
  - Daten sicher verwendet,
    - z.B. in eine Webseite ausgibt, ohne dass sie Schaden verursachen können

- **Günstige Applikationsstruktur**

- Ziel: Lösungen **kompakt** und **übersichtlich** realisieren
- Jeweilige Funktionen als separate PHP-Dateien realisieren
- Idee: Formular und verarbeitenden Code bündeln (**Postback**)

# Applikationsstruktur

---

- **Beispiel: Login-Seite**

- Funktion:
  - Bei erstem Aufruf: Login-Formular (POST) anbieten
  - Nach Abschicken des Formulars:
    - Einloggen (wenn Daten korrekt) *oder*
    - erneut Formular anbieten (wenn Daten nicht korrekt)
- Grober Ablauf:
  - Aufruf **mit** POST-Daten: Login-Daten prüfen
    - Wenn erfolgreich: \$user setzen
    - Wenn nicht erfolgreich: Fehler anzeigen
  - Aufruf **ohne** POST-Daten (*oder* Login-Versuch nicht erfolgreich)
    - Login-Formular ausgeben (ggf. vor-ausgefüllt)
- Beides wird **von der selben PHP-Seite** realisiert
  - **Postback**, das POST des Formulars geht also an die selbe URL
    - im Form-Tag gilt **action="" method="post"**

# Beispiel: Login-Seite

- Prüfen, ob Login erfolgreich

- ```
<?php
    $user_id = NULL;
    if ( @$_POST['name']      == 'Tom'  &&
        @$_POST['password'] == '1234' ) {
        $user_id = 'Tom'; // erfolgreich eingeloggt ...
    }
?>
```

- Login-Formular ausgeben, wenn kein Login erfolgt ist

```
<?php if (!$user_id) {      ?>
    <h1>Login</h1>
    <form action="" method=post>
        Name:      <input type=text      name=name      > <br>
        Passwd.:   <input type=password  name=password  > <br>
                  <input type=submit    name=login value="Login" >
    </form>
<?php } ?>
```

HTML,
wird nur
ausgegeben
wenn
if-Bedingung
erfüllt.

- Willkommensmeldung ausgeben, wenn Login erfolgreich war

```
<?php if ($user_id) {      ?>
    Willkommen, <?php echo htmlspecialchars($user_id); ?>!
<?php } ?>
```

Beispiel: Login-Seite

- **Verbesserung: Login flexibler realisieren**

- Login-Test flexibler und in separate Funktion

- ```
<?php
function get_userdata($id) {
 $user_list = array(
 'Tom' => array('password' => '1234', 'name' => 'Tom Jones'),
 'Peter' => array('password' => '2345', 'name' => 'Peter Deng'),
 'John' => array('password' => '3456', 'name' => 'John Doe'),
);
 return @$user_list[$id]; // liefert NULL bei unbekanntem User
}

function get_login($id, $password) {
 $u = get_userdata($id);
 if ($u && $u['password'] == $password)
 return $u; // erfolgreich eingeloggt
 return NULL; // nicht erfolgreich eingeloggt
}
?>
```

- Prüfen, ob Login erfolgreich

- ```
<?php
$user_data = get_login(@$_POST['name'], @$_POST['password']);
?>
```

Beispiel: Login-Seite

- **Verbesserung: Login separat, setzt selbst Variablen**
 - Login-Test in separater Include-Datei, setzt globale Variablen

In externe-Datei 'inc__get_login.php' auslagern

```
• <?php                                     // This is the Include-File 'inc__get_login.php'
    function get_userdata($id) {
        // ... wie bisher ...
    }

    function get_login($id = NULL, $password = NULL) {
        global $user_data, $user_id;
        $user_data = $user_id = NULL;           // Fallback-Werte
        $u = get_userdata($id);
        if ($u && @$u['password'] == $password )
            $user_data = $u;
            $user_id    = $id;
        }
    }
?>
```

- Prüfen, ob Login erfolgreich

```
• <?php include 'inc__get_login.php';
    get_login(@$_POST['name'], @$_POST['password']);
?>
```

- Die Variablen `$user_data` und `$user_id` werden dadurch gesetzt

Include bindet externe
Dateien ein (→ php.net)

Beispiel: Login-Seite

- **Verbesserung: Benutzer-Daten verwenden**

- Willkommensmeldung mit dem Klartext-Namen ausgeben

```
<?php if ($user_data) {    ?>
    Willkommen
    <?php echo htmlspecialchars(@$user_data['name']); ?>
    !
<?php }    ?>
```

- Besser: Von separater Login-Seite bei Erfolg **weiterleiten** auf Inhaltsseite für Nutzer (z.B. auf persönliche „Homepage“)

- ```
<?php
 if ($user_data) {
 header("Location: /user_home.php", TRUE, 307);
 // Setzt Location Header
 // Setzt Status-Code 307: Temporary Redirect
 }
?>
```

- `header()` muss vor allen Ausgaben aufgerufen werden (→ [php.net](http://php.net))

# Beispiel: Login-Seite

---

- **Verbesserung: Passwörter *gehasht* speichern (1)**

- Dadurch kann ein Angreifer die originalen Passwörter nicht erhalten, wenn er an die Benutzer-Datenbank bekommt.

- **Hash-Funktionen**

- Berechnen zu einem beliebigen String einen **Hash** (ein String fester Länge)
- Ein Hash ist eine Art **Prüfsumme**
- Kleine Änderungen an der Eingabe erzeugen große Änderungen am Hash

- Beispiele für verbreitete Hash-Funktionen: *md5*, *sha1*, *sha256*

- In php berechnen *md5(\$x)* und *sha1(\$x)* Hashes zu Strings

```
<?php
 foreach (array('1234', '1235') as $x)
 echo "md5('$x') = " . md5($x) . "\n";
?>
```

- Ergebnis:

```
md5('1234') = '81dc9bdb52d04dc20036dbd8313ed055'
md5('1235') = '9996535e07258a7bbfd8b132435c5962'
```

# Beispiel: Login-Seite

- **Verbesserung: Passwörter *gehasht* speichern (2)**

- Wir legen die gespeicherten Passwörter im Server nur *gehasht* ab

```
function get_userdata($id) {
 $user_list = array(
 'Tom' => array('pw_md5' => 'e7df7cd2ca07f4f1ab415d457a6e1c13',
 'name' => 'Tom Jones'),
 'Peter' => array('pw_md5' => 'c47abe049e90cd2d285fd697ca4a8c6a',
 'name' => 'Peter Deng'),
);
 return @$user_list[$id];
}
```

- Beim Prüfen des übergebenen Passworts hashen wir dieses und vergleichen

```
function get_login($id, $password) {
 $u = get_userdata($id);
 if ($u && @$u['pw_md5'] == md5($password))
 // ... erfolgreich eingeloggt
 // ... sonst nicht erfolgreich eingeloggt
}
```

- Sind die Hashes gleich, sind auch die Passwörter gleich.
- Der Server kennt (außer während des Prüfens) nur Passwörter-Hashes.
- *Am Rande:* Idealerweise sollte man den Hash noch mit einem *Salt* versehen.

# Beispiel: Login-Seite

## • Hintergrund: Kryptographische Hash-Funktionen (1)

Wozu?

### 1) Anforderung: Falltüreigenschaft

Vorsicht: Für eine **Teilmenge** der Eingaben könnte man aber die Hashwerte voraus berechnen und wiedererkennen (*Rainbow-Tables*).

- aus dem Hashwert kann die Eingabe nicht (effizient) berechnet werden
  - **Einsatz-Beispiel:** Ein Angreifer kann aus dem obigen gehashten Passwort nicht (effektiv) das originale Passwort bestimmen.

Wozu?

### 2) Anforderung: Hash-Kollisionen (praktisch) nicht zu finden

- Zu einer Hashfunktion haben alle Hash-Werte eine feste Länge
- Die Eingabe kann eine beliebige Länge haben
  - Es gibt mehr Eingabe-Werte als Hash-Werte
  - Es gibt daher mehrere Eingaben, die den selben Hash haben ([Hash-Kollision](#))
- **Aber:** Hash-Kollisionen können nicht (effizient) gefunden werden
  - Zu einer Eingabe kann man nicht (effektiv) andere mit gleichem Hashwert finden.
  - **Einsatz-Beispiel:** Wenn der Hash eines Strings unverändert ist, dann ist auch der String (höchstwahrscheinlich) unverändert.
- Und: Hash-Kollisionen sind allgemein astronomisch selten
  - Bei einer **sicheren** Hash-Funktion sind sie praktisch auch nicht gezielt zu finden

**MD5** und **SHA-1** gelten insbes. bzgl. provozierter Kollisionen nicht mehr für alle Anwendungen als ausreichend sicher.

# Beispiel: Login-Seite

---

- **Hintergrund: Kryptographische Hash-Funktionen (2)**

- MD5 und SHA1 erfüllen mittlerweile die o.g. Anforderungen nicht mehr ausreichend sicher.
  - Es sind u.a. Verfahren entdeckt worden, mit denen **Hash-Kollisionen** unter bestimmten Bedingungen gezielt gefunden werden können.
  - Sie sollten daher für kritische Anwendungen nicht mehr eingesetzt werden.

- **Bessere Alternativen: z.B. SHA256**

```
<?php
 foreach (array('1234', '1235') as $x)
 echo "sha265('$x') = " . hash('sha256', $x) . "\n";
?>
```

Ergebnis:

```
sha265('1234') = '03ac674216f3e15c761ee1a5e255f067953623c8b388b4459e13f978d7c846f4'
sha265('1235') = '310ced37200b1a0dae25edb263fe52c491f6e467268acab0ffec06666e2ed959'
```

- Die PHP-**hash**-Funktion unterstützt noch weitere Hashes
  - Siehe <https://www.php.net/manual/de/function.hash.php>

# Beispiel: Login-Seite

- **Hintergrund: Kryptographische Hash-Funktionen (3)**

- Alle (effizient) umkehrbaren Funktionen sind **keine** Hash-Funktionen

- Beispiel: **base64-Kodierung** ist keine Hash-Funktion

- ```
<?php
    $s = '1234';
    $a = base64_encode($s);
    echo "base64_encode('$s') = '$a'\n";
    unset($s);           // original-String zur Demonstration gelöscht
    $b = base64_decode($a);
    echo "base64_decode('$a') = '$b'\n";
?>
```

- Ergebnis:

```
base64_encode('1234')      = 'MTIzNA=='
base64_decode('MTIzNA==') = '1234'
```

Zudem: lokale Änderung
→ lokaler Effekt:

```
'1234' → 'MTIzNA=='
'1235' → 'MTIzNQ=='
```

- Zur Erinnerung (Kapitel 1):

- Die Base64-Kodierung wurde zur Verschlüsselung des Passworts in der **Basic-Authentication** eingesetzt
 - Diese kann aber leicht dekodiert werden → Schein-Sicherheit („**Snakeoil**“)

Beispiel: Login-Seite

- **Ausblick: Sichere Passwort-Hash-Verfahren**

- Problem: Passwörter oft **kombinatorisch schwach**
 - Vom Benutzer gewählte Passwörter sind oft **relativ kurz** (6-10 Zeichen)
 - Es werden oft nur wenige Zeichen genutzt (nur Kleinbuchstaben, nur Ziffern)
 - 10^6 (6 Ziffern) ... $26^{10} \approx 1,4 \cdot 10^{14}$ (10 Kleinbuchstaben) Varianten
- Angriff: Man berechnet alle Hashes voraus („**Rainbow-Table**“)
 - Eine Festplatte mit 8TB ($8 \cdot 10^{12}$ Zeichen) kostet ca. 100€
 - Für einige 10.000€ kann man alle Hashes für erwartete Passwörter speichern
- Lösung: **Salt** (oder **Pepper**)
 - Das Passwort wird um einen Zufalls-Zeichenkette **erweitert** (**Salt**)
 - Der Salt wird zusammen mit dem Passworthash abgelegt

\$salt **hash(\$salt . \$passwd)**

\$a34df2d\$81dc9bdb52d04dc22036dbd8313ed055

Und/oder **Pepper**:
Vom Server fest gewählte
(geheime) Erweiterung
hinzufügen

Beispiel: Login-Seite

- **Ausblick: Sichere Passwort-Hash-Verfahren**

- Problem: Passwörter haben oft **geringe Entropie** oder sind **errratbar**
 - Entropie ist ein Maß, wie ungeordnet ein System ist
 - Konkret: Bestimmte Passwörter sind beliebt
 - Angreifer sammeln „beliebte“ **Passwörter** und probierten diese (**Wörterbuch**)
 - Aber auch: Passwörter sind nicht völlig gleichverteilt
 - Worte, Silben oder Zifferngruppen kommen oft vor
 - Häufig Muster in Passwörtern (z.B. „xyz123#“)
 - Erfolgswahrscheinlichkeit bei **geschicktem Ausprobieren** besser als erwartet
- Angriffsform: **Wörterbuchattacke**, geschickte **Brute-Force-Attacke**
- Lösung: Das **Ausprobieren** eines Passworts „teuer“ machen
 - Hash-Funktion mehrmals (z.B. 10.000 **Runden**) hintereinander anwenden
 - Dadurch kostet ein Passwort-Test z.B. eine Sekunde → Massentests unmöglich
 - Das erschwert auch den Aufbau einer Rainbow-Table extrem
 - Beispiel: **Bcrypt** (<https://de.wikipedia.org/wiki/Bcrypt>)
 - Salting, parametrierbarer Rechenaufwand, nicht ASIC-/SIMD-optimierbar

Beispiel: Login-Seite

- **Ausblick: Sichere Passwort-Hash-Verfahren**

- Passwort-Hashing ist ein komplexes Thema
 - Man kann leicht Fehler bzgl. Verwundbarkeiten verursachen
- Besser eine dafür vorgesehene und überprüfte Lösung nutzen
 - Erweiterte Passwort-Hashing-Funktionen von PHP
 - <https://www.php.net/manual/de/book.password.php>
 - `$hash = password_hash($password, $algo [, $options ])`
 - `$algo` z.B. `PASSWORD_BCRYPT` oder `PASSWORD_ARGON2I`
 - Liefert so etwas wie
`$argon2i$v=19$m=1024,t=2,p=2$YzJBMzc3d3laeg$zqUsdWLaw3sYY2i2jT0 ...`
 - Incl. Algorithmus, Anzahl Runden, Parameter, Salt, Hash
 - `$ok = password_verify($password, $hash)`
- Siehe
 - <https://www.php.net/manual/de/faq.passwords.php>