PHP -Verschlüsslungen Teil 1

Zum Autor

Tach zusammen, mein Pseudonym ist „Know v3.0“. Mein Interessengebiet beinhaltet die Programmierung – PHP, Perl, Python, Ruby, Qt-C++, C#, VB.Net, ASM, Websec. ein wenig Reverse und Social-Engineering.

Ich habe mich dazu entschlossen dieses Tutorial zu schreiben, weil ich in diesem Bereich, nichts brauchbares für anständige Verschlüsslungen mit PHP gefunden habe.

Ich möchte hier kurz und knapp einen Einblick in die Verschlüsslung, deren Anwendung und Einsetzung geben.

Was dieses Tutorial nicht ist

Dieses Tutorial ist nicht ohne Rechtschreibfehler an zu finden.
Ich habe grob drüber geschaut, denn es geht um den Inhalt, solange dieser verständlich ist, sollte es easy sein. Sonst bitte via Mail „knolegend@yahoo.com“ flamen ;)
Dieses Tutorial soll keine StepbyStep Beschreibung, sein um eine Verschlüsslungs-Klasse zu Erstellung und sinnvoll zu nutzen! Es soll lediglich einen Einblick in die (Richtige)-Verschlüsslung unter PHP geben.

Ja! Das „Richtig“ steht in Klammern, weil ich für nichts was ich hier von mir gebe eine Gewehr gebe! Schaden o.ä. der durch dieses Tutorial entsteht, habe ich nicht zu verantworten und ich distanziere mich von dieser Person und dem Sachverhalt!

Einleitung

Voraussetzung für das Verständnis dieses Tutorials sind **gute** PHP Kenntnisse.
D.h. Erfahrung mit PHP + OOP und eventuell Vorkenntnisse im Zusammenhang mit den PHP-Funktionen *md5*, *sha1* und allgemeiner Verschlüsslung.

Blabla genug Gelaber...

Der Anfang

Fangen wir mit den Funktionen *md5()*, *sha1()*, *base64_encode()* und *base64_decode()*

Okay! Wo ist jetzt der Unterschied zwischen *md5()*, *sha1()*, *base64_encode()* und *base64_decode()* ?

Ganz einfach! *md5()* sowie *sha1()* sind **Hash**-Funktionen. D.h. Dass man die verschlüsselten Zeichenketten die via *md5()* oder *sha1()* erstellt wurden, ***nicht*** wieder entschlüsseln kann, weil sie auf zufalls-gesteuerten Rechenalgorithmen basieren - theoretisch!

base64_encode() und *base64_decode()* sind lediglich Kodierungen! Sie kodieren die Zeichenketten nur, deswegen kann man sie auch wieder dekodieren(*base64_decode()*) und erhält so den Klartext(Plaintext).

MD5 und SHA1

Schauen wir uns mal das Grundgerüst von der Funktion *md5()* an:

Beschreibung Report a bug

string **md5** (string *$str* [, bool *$raw_output* = *false*])

Berechnet den MD5-Hash von ***str*** unter Verwendung des » RSA Data Security, Inc. MD5 Message-Digest Algorithm und gibt das Ergebnis zurück.

Parameter-Liste Report a bug

str

Die Zeichenkette.

raw_output

Wurde der optionale Parameter ***raw_output*** mit **TRUE** angegeben, wird der MD5-Wert im Raw Binary Format mit einer Länge von 16 Zeichen zurückgegeben.

Wichtig für uns ist nur:

string ***md5*** *(string $str)*

plus Salt, der komischer Weise nicht im Manuel aufgelistet wird.

Da die Variable *raw_output* Standardmäßig auf *FALSE* steht, brauchen wir uns keine Gedanken um den Parameter machen.

Ein kleiner Exkurs:

Ein Salt dient dazu, den **Hash** den man nach der Verschlüsslung mit *md5()* erhält, davor, zu „salzen“.

Erläuterung:

„Salt (engl. ‚Salz‘) bezeichnet in der Kryptographie eine zufällig gewählte Zeichenfolge, die an einen gegebenen Klartext vor der Verwendung als Eingabe einer Hashfunktion angehängt wird, um die Entropie der Eingabe zu erhöhen“

Quelle: http://de.wikipedia.org/wiki/Salt_(Kryptologie)

Back2Topic:

Die *md5()* Funktion sieht mit **Salt** dann so aus:

string ***md5*** *(string $str* **.** *string $salt)*

Schauen wir uns mal das Grundgerüst von der Funktion *sha1()* an:

Beschreibung Report a bug

string **sha1** (string *$str* [, bool *$raw_output* = *false*])

Berechnet den SHA1 Hash von ***str*** unter Verwendung des » US Secure Hash Algorithmus 1.

Parameter-Liste Report a bug

str

Die Eingabezeichenkette.

raw_output

Ist der optionale Parameter ***raw_output*** **TRUE**, wird der SHA1-Extrakt im Raw-Binary-Format mit einer Länge von 20 Zeichen zurückgegeben. Ansonsten ist der Rückgabewert ein 40 Zeichen langer Hexadezimalwert.

Auch hier ist für uns nur die Formation von belang:

string ***sha1*** *(string $str)*

natürlich nur in Kombination mit einem *Salt* benutzen – ***mindestens!***

Mit *Salt*:

string ***sha1*** *(string $str* **.** *string $salt)*

Ich denke mal, dass sich jeder, der sich Entschlossen hat, dieses Tutorial zu lesen, im Stande ist sich die Nutzung von *md5()* sowie *sha1()* vorzustellen.

Falls nicht, hier die Links zu dem Manuel der beiden Funktionen:

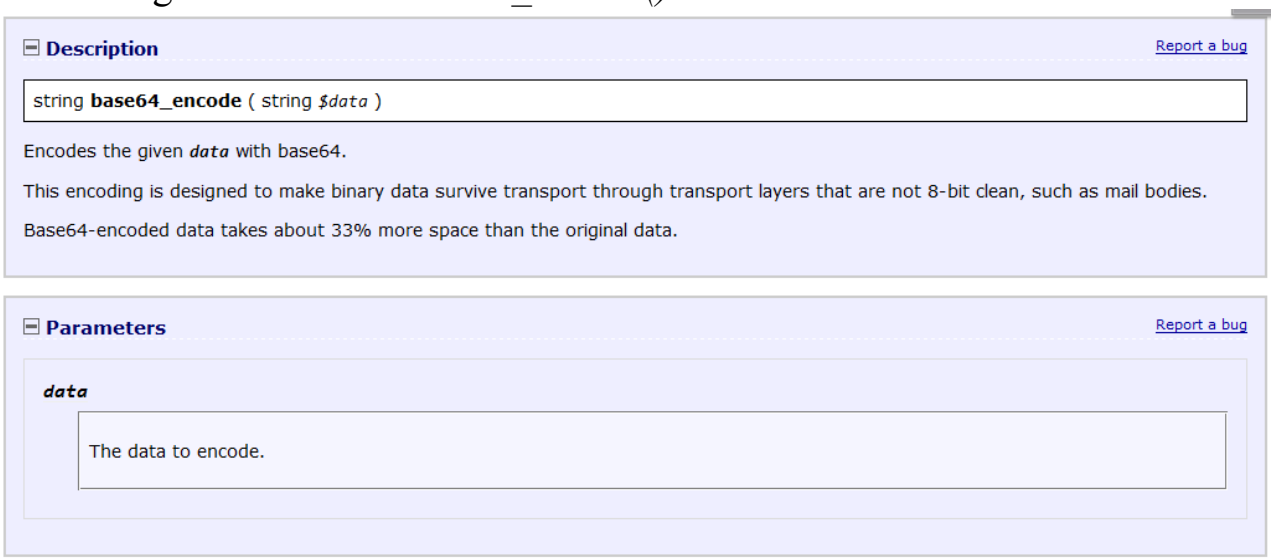
md5(): http://de2.php.net/manual/de/function.md5.php
sha1(): http://de2.php.net/manual/de/function.sha1.php

TIPP:
Im unteren Teil des Manuals sind Beispiele zu den Funktionen.

BASE64_ENCODE und BASE64_DECODE

Hier veranschauliche ich auch „noch“ die Beiden Funktionen *base64_encode()* und *base64_decode()*.

Benutzung der Funktion *base64_encode()*:



Description Report a bug

string **base64_encode** (string *$data*)

Encodes the given ***data*** with base64.

This encoding is designed to make binary data survive transport through transport layers that are not 8-bit clean, such as mail bodies.

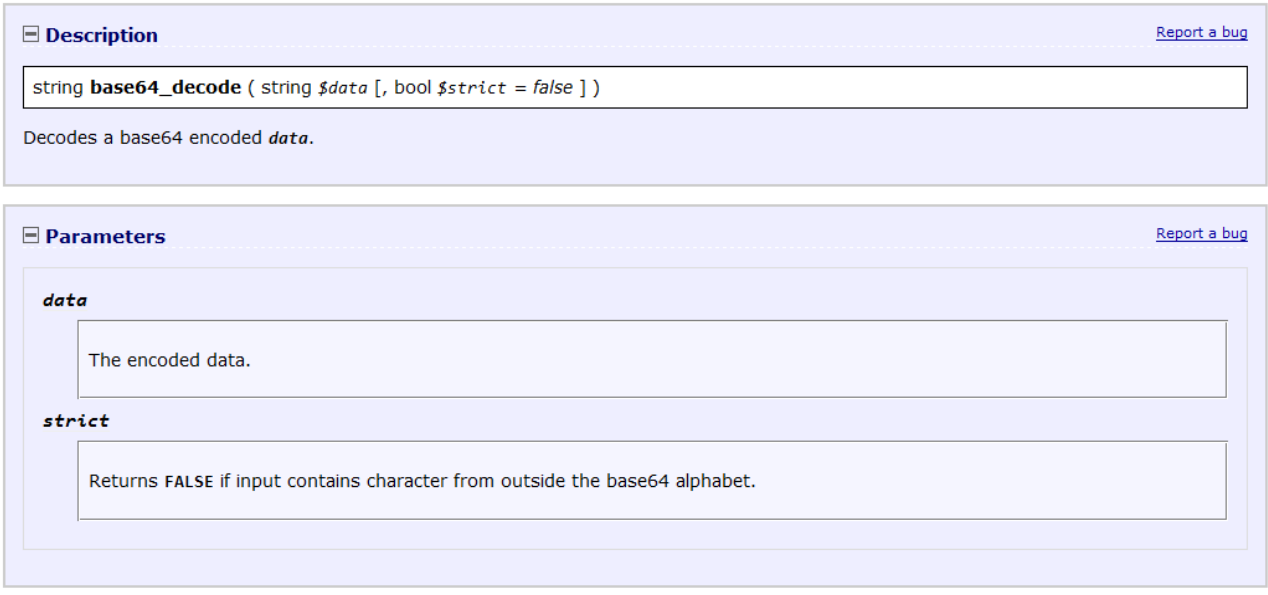
Base64-encoded data takes about 33% more space than the original data.

Parameters Report a bug

data

The data to encode.

Benutzung der Funktion *base64_decode()*:



Description Report a bug

string **base64_decode** (string *$data* [, bool *$strict* = *false*])

Decodes a base64 encoded ***data***.

Parameters Report a bug

data

The encoded data.

strict

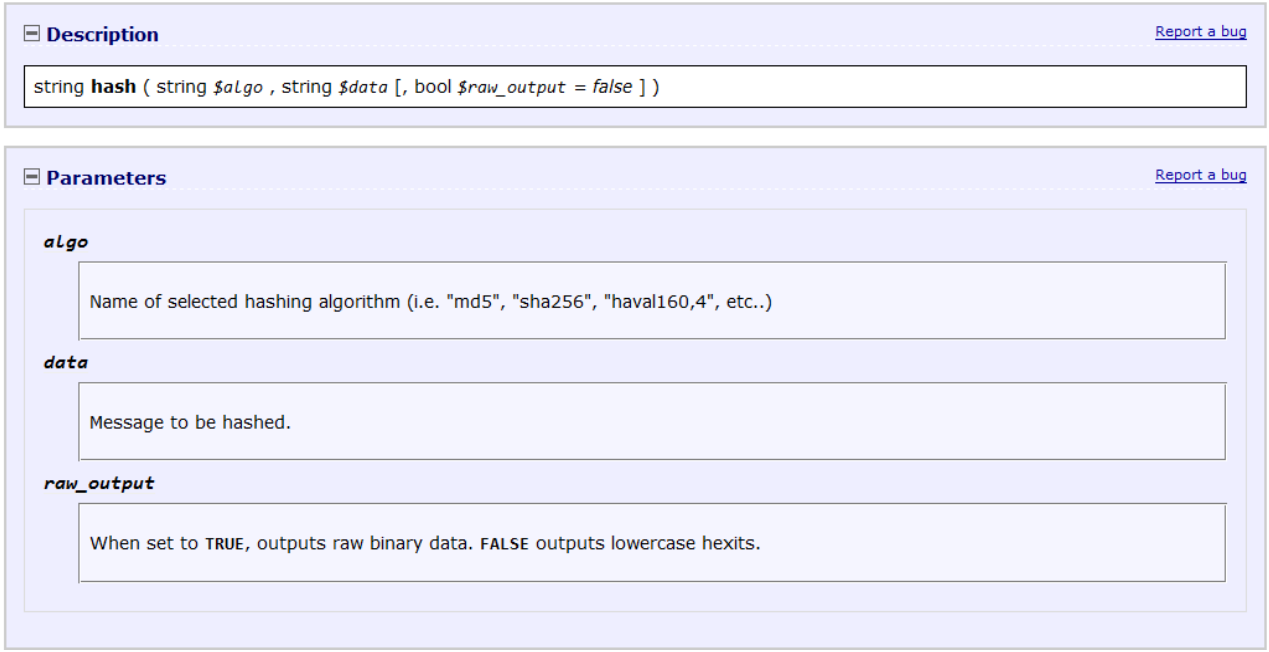
Returns **FALSE** if input contains character from outside the base64 alphabet.

Ich denke mal, dass ich dazu nicht mehr sagen brauch ;)

Andere nützliche Funktionen

Hier liste ich noch weitere für die Verschlüsslung wichtige Funktionen auf.

Eine der wichtigsten Funktionen die wir noch gebrauchen können, ist *hash()*:



Description Report a bug

string **hash** (string *$algo* , string *$data* [, bool *$raw_output* = *false*])

Parameters Report a bug

algo

Name of selected hashing algorithm (i.e. "md5", "sha256", "haval160,4", etc..)

data

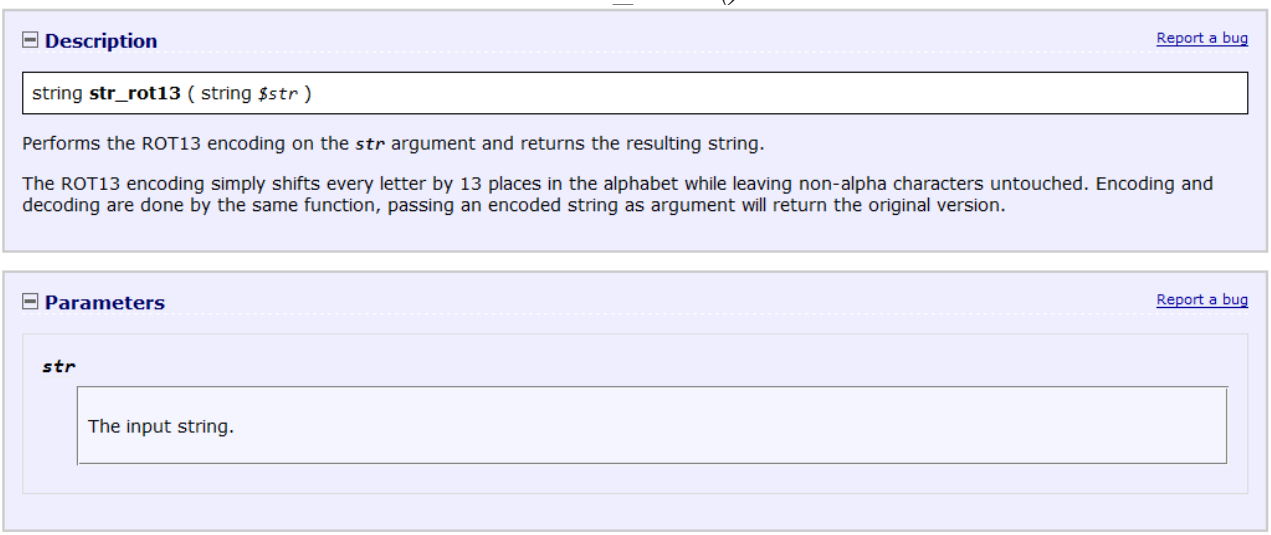
Message to be hashed.

raw_output

When set to **TRUE**, outputs raw binary data. **FALSE** outputs lowercase hexits.

Diese Funktionen ermöglicht uns den Algorithmus den wir zum Verschlüsseln benutzen wollen auszusuchen. Sehr nützlich!

Eine weitere hilfreiche Funktion ist *str_rot13()*:



Description Report a bug

string **str_rot13** (string *$str*)

Performs the ROT13 encoding on the ***str*** argument and returns the resulting string.

The ROT13 encoding simply shifts every letter by 13 places in the alphabet while leaving non-alpha characters untouched. Encoding and decoding are done by the same function, passing an encoded string as argument will return the original version.

Parameters Report a bug

str

The input string.

Die etwas bewanderten unter euch, werde sich wohl denken „Dafuq! Was soll diese scheiß Funktion uns nützen ?! “

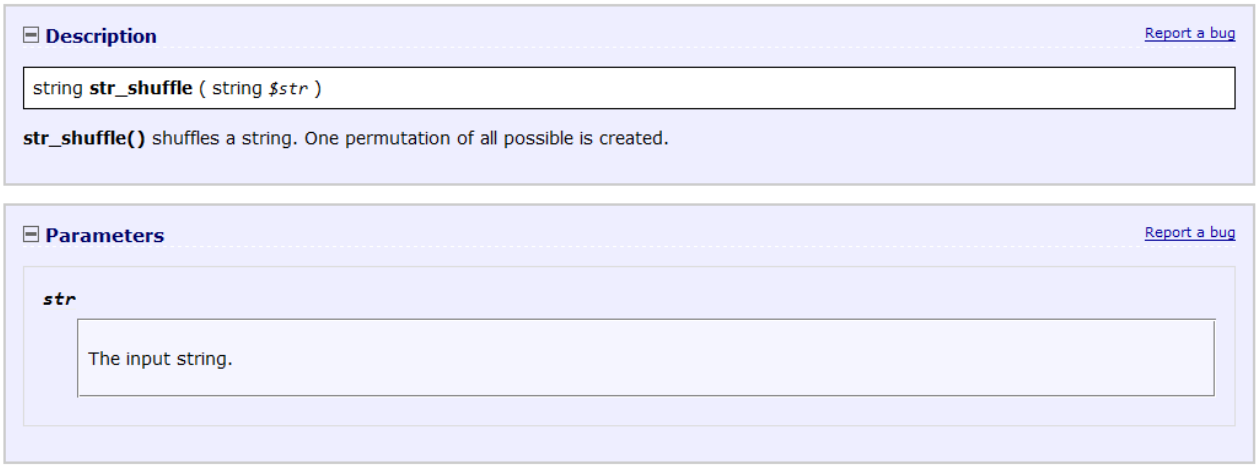
Ganz einfach, wenn wir einen **Hash** „generieren“ ist schnell einsehbar (Länge und Aussehen) worum es sich handelt. Aber wenn man den **Hash-String** mit *str_rot13()* bearbeiten, oder den **Hash-String** „splittet“ und nur ein Teil mit *str_rot13()* bearbeiten und dann noch eventuell *strrev()* benutzen, wird schnell klar, das es erst etwas an Arbeit verlangt um dann den „wirklichen“ **Hash** zu „bruten“.

TIPP:
Du weißt nicht was „bruten“ oder „BruteForce“ bedeutet?
Hier findest du eine einfache Erklärung zu diesem Mysteriösen Wort.

Erklärung der Bedeutung des Wortes: http://de.wikipedia.org/wiki/Brute-Force-Methode

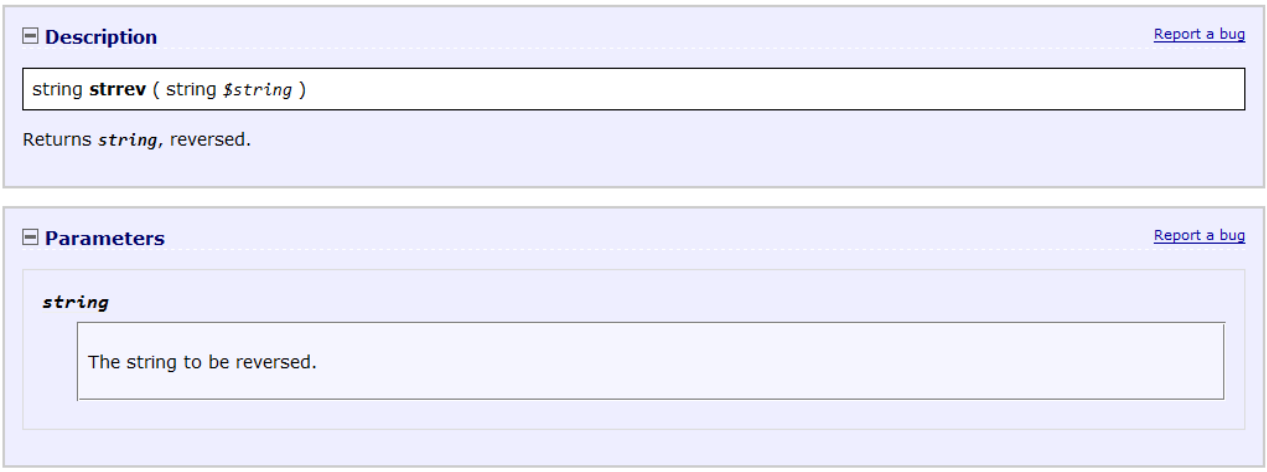
Erklärung des Zusammenhangs mit der Kryptologie: http://de.wikipedia.org/wiki/Brute-Force-Methode#Kryptologie

str_shuffle() ist mit Vorsicht zu genießen! Diese Funktion erstellt aus einem **String** einen neuen **String** in dem die Zeichen aus dem Ersten zufällig neu zusammengesetzt wurden. Diese Funktion ist nicht für (sehr)lange Passwörter geeignet! Und schon gar nicht, um damit Passwörter von Usern zu „Verschlüsseln“!

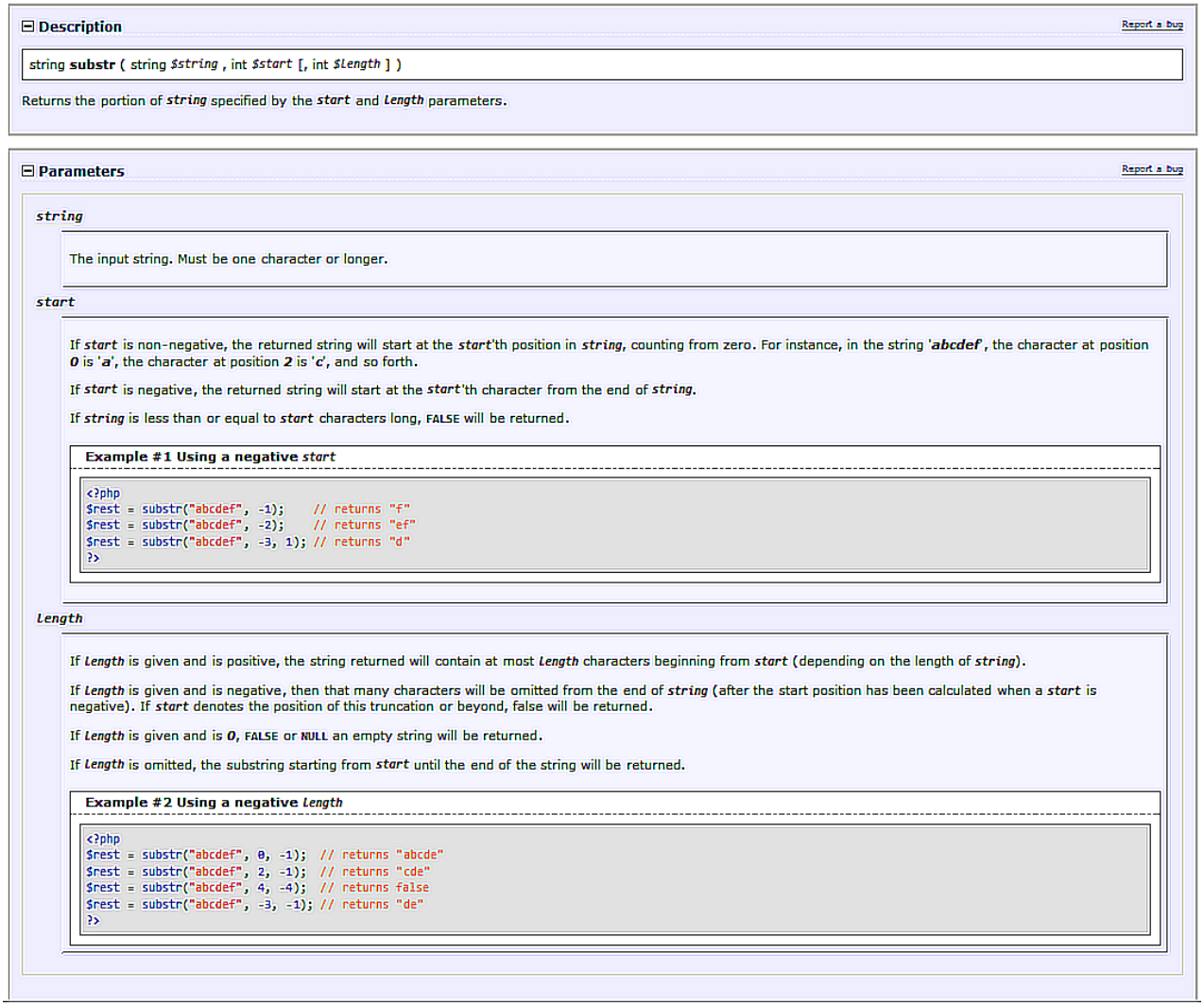


Als vorletzte Funktion, stelle ich *strrev()* vor, sie gibt den übergebenen **String** rückwärts zurück.

Format:



Nun als letzte Funktion, zeige ich euch substr().



Description Report a bug

string **substr** (string *$string* , int *$start* [, int *$length*])

Returns the portion of ***string*** specified by the ***start*** and ***length*** parameters.

Parameters Report a bug

string

The input string. Must be one character or longer.

start

If ***start*** is non-negative, the returned string will start at the ***start***'th position in ***string***, counting from zero. For instance, in the string '***abcdef***', the character at position ***0*** is '**a**', the character at position ***2*** is '**c**', and so forth.

If ***start*** is negative, the returned string will start at the ***start***'th character from the end of ***string***.

If ***string*** is less than or equal to ***start*** characters long, FALSE will be returned.

Example #1 Using a negative *start*

```
<?php
$rest = substr("abcdef", -1);    // returns "f"
$rest = substr("abcdef", -2);    // returns "ef"
$rest = substr("abcdef", -3, 1); // returns "d"
?>
```

Length

If ***Length*** is given and is positive, the string returned will contain at most ***Length*** characters beginning from ***start*** (depending on the length of ***string***).

If ***Length*** is given and is negative, then that many characters will be omitted from the end of ***string*** (after the start position has been calculated when a ***start*** is negative). If ***start*** denotes the position of this truncation or beyond, false will be returned.

If ***Length*** is given and is ***0***, FALSE or NULL an empty string will be returned.

If ***Length*** is omitted, the substring starting from ***start*** until the end of the string will be returned.

Example #2 Using a negative *Length*

```
<?php
$rest = substr("abcdef", 0, -1);  // returns "abcde"
$rest = substr("abcdef", 2, -1);  // returns "cde"
$rest = substr("abcdef", 4, -4);  // returns false
$rest = substr("abcdef", -3, -1); // returns "de"
?>
```

Sie dient dazu, um den Teil des übergebenen **Strings** zurückzugeben.

Die Praktik und ein wenig Theorie ^^

Ich möchte euch hier mal einen kleinen Benchmark-Test der Verschlüsselungsmethoden zeigen:

- *md5()* = **100%**
- *md5()* + *salt* = **200%**
- *md5()* + *salt* + *Passwortlänge* = **400 – 700%** (je nach Passwortlänge)
- *md5()* + *salt* + *Passwortlänge* + *crypt()* = **2500 – 4500%**

So die Theorie! Wir übertreiben es natürlich in der Praktik ;)

Ich zeige euch hier eben Beispiele für den obigen Benchmark-Test.
Dafür benötigen wir einen User, nennen wir ihn mal: *1337-leet-ultra-k!ll4*
Da wir noch ein richtiges l33t Passwort brauchen, kriegt unser User noch ein Passwort: „12334567“ ;-)

Beispiel-1:

```php
<?php

Class Examples
{

    private $username   = "1337-leet-ultra-k!ll4";

    private $password   = "12334567";

    private $cypher     = NULL;


    public function __construct() {}


    public function  exOne()
    {

        $this->cypher = md5( $this->password );

        print "<h1>Example-1: </h1> <br /> " . $this->cypher;
```

```
    }
}



$instance = new Examples();


/* Look at the first example */
$instance->exOne();


?>
```

Ausgabe:

Example-1:

32135a337f8dc8e2bb9a9b80d86bdfd0

Hier sehen wir den **MD5-Hash** von dem Passwort „12334567“.

Um sicherzustellen, dass der Angreifer es nicht so einfach hat den **Hash** zu bruten, benutzen wir jetzt einen **Salt**.

Beispiel-2:

Wir ergänzen unser Klasse „Examples“ mir der Funktion:
In diesem Beispiel, dient der Benutzername als **Salt**.

```
public function exTwo()
{

    $this->cypher = md5( $this->password . $this->username );

    print "<h1>Example-2: </h1> <br />" . $this->cypher;

}
```

Und aufrufen müssen wir die Funktion auch noch:

```
/* Look at the second example */
$instance->exTwo();
```

Ausgabe:

Example-2:

269c45bc69e7391229455de0d3360dc8

Um die Ausgabe vom 2ten Beispiel zu bruten, würde der Angreifer schon erheblich mehr Zeit benötigen. Aber da dies immer noch nicht ausreicht, programmieren wir sofort weiter.

Da ich aber denke, dass Ihr euch nun die anderen Beispiele auch vorstellen könnt, überspringe ich mal den Part wo ich die Funktionen dazu code und gehe direkt an ein Beispiel, wie man das Passwort möglicherweise verschlüsseln könnte.

Die Idee hinter dem folgenden Algorithmus ist, dass man anhand der Passwortlänge und einem Array aus **Hashes,** die aus dem Passwort und immer einem anderen **Salt** bestehen, einen neuen **Hash** „generiert".

Also, dass man dann zB. ein **Array** hat, in dem 8(die Länge des Passwortes) **Hashes** gespeichert sind, die alle aus dem **Benutzerpasswort** und einem immer anderen **Salt** bestehen. Dann wird immer ein Zeichen aus jeweils einem **Hash** des **Arrays** genommen und in einer Variable gespeichert, bis man ein 32 Zeichen langen String hat.

Ergo, der Angreifer denkt, dass es der **Hash** des Passwortes des Benutzers ist, aber wenn er den **Hash** erfolgreich brutet und dann die ergebene Zeichenfolge als **Passwort** benutzen will, wird ihm „gezeigt" das es das falsche Passwort sei.

Weitere Vorteile sind, dass kein **Salt** in der Datenbank gespeichert werden muss – wie bei vBulletin, wenn mich nicht alles irre – und dass der Angreifer durch eine erfolgreiche SQL-Injection nicht an den Algorithmus kommt der in der Verschlüsslungs-Klasse, „sicher „auf dem Server liegt ;)

Hier der Code zu der Idee:

```
class Crypt
{

    private $username   = "1337-leet-ultra-k!ll4";

    private $firstname  = "Max";
```

```
        private $lastname   = "Mustermann";

        private $email      = "mAxKiLl4@hotmail.de";

        private $password   = "12334567";

        private $hashes     = array();

        private $salts      = array();

        private $cypher     = NULL;

        private $count      = 0;

        private $hcount     = 0;

        public function __construct()
        {
                    /* Die bekannten Benutzerdaten in einem Array speichern. */
                $this->salts = array($this->username, $this->firstname, $this->lastname,
$this->email, $this->password);

                    /* Die Schleife sooft durchfuehren, wie die Laenge des Passwortes
ist. */
                for($i = 0;$i < strlen($this->password);++$i)
                 {
                                /* Wenn die Variable "count"(int) die gleiche Zahl
beinhaltet, wie es Elemente im Array "salts" gibt,
                                    * dann "count" auf 0 setzen, sonst einfach
weitermachen. */
                       ($this->count == count($this->salts)) ? $this->count = 0 : NULL ;

                                /* Im Array "hashes" den MD5-Wert vom Benutzerpasswort und
dem Element aus dem Array "salts"
                                    * mit dem Index von "count" speichern. */
                     $this->hashes[$i] = md5($this->password.$this->salts[$this->count]);

                                /* "count" um 1 erhoehen. */
                     ++$this->count;

                 }

                    /* Die Funtkion "crypt" aufrufen. */
                $this->crypt();

        }

        private function crypt()
        {
                    /* "count" den Wert 1 zuweien. */
                $this->count = 1;

                    /* Die Schleife 33 mal durchlaufen, um den 32 Zeichen langen
"Schein-Hash" zu erstellen.
                     * Glaubt mir es muessen 33 Durchlaeufe sein xD Ich bin nicht
bloed :D Es geht auch anders,
                     * aber ich habe es nunmal so gemacht ;) */
                for($i = 0;$i <= 33;$i++)
                {
                    /* Das Element aus dem Array "hashes" mit dem Index von "count - 1" der
Variable "hash" zuweisen. */
                    $hash = $this->hashes[$this->count-1];
```

```
                        /* Wenn die Variable "count"(int) die gleiche Zahl
beinhaltet, wie es Elemnte im Array "hashes" gibt,
                         * dann "count" auf 0 setzen und die Variable "hcount" um 1
erhoeren. */
                if($this->count == count($this->hashes))
                {
                     $this->count = 0;
                     ++$this->hcount;

                }

                        /* Der Variable "cypher" den Buchstaben an der Stelle "count
+ hcount" aus dem String "hash" HINZUFUEGEN. */
                $this->cypher .= $hash[$this->count+$this->hcount];

                        /* "count" um 1 erhoehen. */
                ++$this->count;

            }

                    /* Den "Schein-Hash" ausgeben! */
            print "<br />Der neue Hash: ".$this->cypher;
        }


    }

   /* Eine Instanz der Klasse "Crypt" erzeugen. */
   $instance = new Crypt();
```

Das war nur eine kleine Idee. Es gibt weitaus kompliziertere Möglichkeiten, aber diese – von mir gezeigte – ist schon gut ^^

Aber was wichtig ist, diese Klasse reicht nicht um eine größere Webseite vor angriffen – Sql-injections – zu schützen!

Lange und komplizierte DB(Datenbank) Passwörter und Benutzernamen sind ein weiterer Faktor um die Sicherheit zu erhöhen. Natürlich müssen auch alle Parameter die von der Seite entgegengenommen werden, gefiltert werde bzw. auf mögliche Angriffe überprüft werden. Bei allen SQL-Anweisungen die „erzeugt“ werden, müssen die PHP-Variablen auf mögliche Sql-injections prüfen und/oder alle Variablen, die direkt in die SQL-Anweisung gesetzt werden mit zb. *mysql_real_escape_string()* etc „sichern“ lassen.

TIPP: Die oben genannten Funktionen, reichen nicht um „komplette“ Sicherheit zu erreichen. Es muss auch der Server auf dem du deine Webpräsenz hast, vernünftig Konfiguriert und abgesichert sein.

Weiter geht es demnächsht mit Teil 2!

Email: knowlegend@yahoo.com
Board: https://new-crew.biz
Blog: http://nn-fraktion.blogspot.de/
ICQ: DafuQ! Frag via Email ;)

Beste Grüße!