

Kryptographie mit PHP

Inhalt:

1. Einführung	S. 1
2. Eingebaute Hash-Funktionen	S. 2
3. Hash-Algorithmen der Bibliothek „mhash“	S. 3
4. Erzeugung von Schlüsseln	S. 4
5. Symmetrische Verschlüsselung mit „mcrypt“	S. 5
6. Wichtige Hinweise	S. 7

1. Einführung

Zunächst eine kleine Erklärung der Begriffe:

- **Message Digests:** auch kurz als **Hash** bezeichnet. Ihre Aufgabe in der Kryptographie ist es, die *Datenintegrität* ähnl. Prüfsummen (z.B. CRC32 und Adler32, häufig verwendet in ZIP-Files) sicherzustellen. Im Gegensatz zu Prüfsummen muss es bei einem Message Digest unmöglich sein, zwei Eingaben zu finden, die die selbe Ausgabe erzeugen, oder eine bestimmte Ausgabe bewusst erzeugen zu können. Die Anwendungsmöglichkeiten reichen von einer Verwendung als *Prüfsumme* über die *Verschlüsselung von Passwörtern* und *Fingerprints von öffentlichen Schlüsseln* bis hin zum sog. "*Zero-Knowledge-Proof*", bei dem durch den Message Digest zu einem späteren Zeitpunkt die Existenz einer Nachricht bewiesen werden kann, ohne sofort deren Inhalt preiszugeben.
- **Block Cipher:** gehören zu den *symmetrischen Verschlüsselungsverfahren*, d.h. zum Verschlüsseln und Entschlüsseln einer Nachricht wird derselbe Schlüssel verwendet. Die Nachricht wird dabei anhand des gegebenen Schlüssels jeweils in Blöcken zu meist 8 oder 16 Bytes verschlüsselt. Die Tatsache, dass nicht jedes Byte einzeln verschlüsselt wird, erhöht die Sicherheit.
- **Stream Cipher:** gehören ebenfalls zu den symmetrischen Verschlüsselungsverfahren. Hier wird jedes Byte einzeln und isoliert verschlüsselt (mittels einer bitweisen XOR-Operation auf den Key-Stream, der nach einer gewissen Anzahl von Bytes intern neu berechnet wird). Stream Cipher gelten im Allgemeinen als nicht so sicher wie Block Cipher (die XOR-Operation erlaubt zudem die Rekonstruktion des Schlüssels, sofern ursprüngliche und verschlüsselte Nachricht bekannt sind), arbeiten jedoch oft etwas schneller und sind einfacher in der Handhabung.
- **Public-Key-Kryptosysteme:** auch *asymmetrische Verschlüsselungsverfahren* genannt, bekannte Vertreter sind *RSA* oder *ElGamal*. Im Gegensatz zu symmetrischen Verfahren gibt es hier ein Schlüsselpaar: ein Schlüssel wird geheim gehalten (*private key*), ein anderer veröffentlicht (*public key*). Was mit einem public key verschlüsselt wurde, kann nur mit dem zugehörigen private key entschlüsselt werden und umgekehrt. Somit entfällt das Problem der Schlüsselübertragung von symmetrischen Verfahren. Jedoch sind asymmetrische Verfahren sehr langsam, so dass nicht die gesamte Nachricht mit einem Public-Key-System verschlüsselt wird. Vielmehr wird die Nachricht mit einem (schnellen) symmetrischen Verfahren verschlüsselt, deren Schlüssel wiederum mit einem asymmetrischen Verfahren, um die Übertragung sicher zu machen (*hybride Systeme*).
- **Digitale Unterschriften:** haben die Aufgabe, *Herkunft und Integrität* einer Nachricht sicherzustellen. In der Praxis wird dazu der Message Digest einer Nachricht berechnet

und mit einem asymmetrischen Verfahren verschlüsselt. Der Empfänger kann mit dem public key den mitgeschickten Message Digest entschlüsseln und mit dem selbst berechneten Message Digest abgleichen. Stimmen sie überein, sind Herkunft und Integrität gesichert.

- **Zertifikate:** Sollte ein privater Schlüssel in falsche Hände fallen, wären die Auswirkungen verheerend (bis zum Identitätsdiebstahl). Hier setzen Zertifikate an: sie *garantieren die Echtheit und die Herkunft* eines publizierten öffentlichen Schlüssels. Sog. X.509-Zertifikate werden, um die Glaubwürdigkeit abzusichern, von CAs (*Certification Authorities*) ausgestellt. Das sind unabhängige Institute, die staatlich autorisiert und geprüft werden, wie z.B. VeriSign oder Thawte. Ein Zertifikat enthält in der Regel den öffentlichen Schlüssel des Inhabers, seinen Identitätsmerkmalen (Name, Wohnort, Firma, Abteilung, E-Mail etc.), der Bezeichnung des CA und den digitalen Signaturen von Inhaber und CA. Dieses Verfahren wird auch bei *SSL-Verbindungen* benutzt.

2. Eingebaute Hash-Funktionen

Die bekanntesten und am häufigsten verwendeten Message Digests sind **MD5** und **SHA1**. MD5 ist ein Digest der Länge 128 Bit, zurückgegeben werden 16 Bytes (Zahlenbereich jeweils 0-255), die den MD5-Wert der Eingabe enthalten. SHA1 ist 160 Bit lang, das entspricht ergo 20 Bytes.

Diese beiden Hash-Funktionen sind im Standard-Umfang von PHP enthalten (SHA1 allerdings erst ab PHP 4.3.0), ihre Benutzung ist denkbar einfach, da das Ergebnis bereits als Hex-String formatiert zurückgegeben wird (d.h. jedes Byte wird in hexadezimale Schreibweise umgewandelt). Datei **hash1.php**:

```
echo md5("message digest") . "\n";
echo md5_file("test.txt") . "\n";
echo sha1("message digest") . "\n";
echo sha1_file("test.txt");
```

Die Funktionen **md5()** und **sha1()** nehmen als Argument die Daten (in Form eines Strings), aus denen der MD5 bzw. SHA1-Wert berechnet werden soll. **md5_file()** und **sha1_file()** berechnen den Digest zum Inhalt der angegebenen Datei. Seit PHP 5.0.0 nehmen all diese Funktionen auch einen zweiten Parameter, welcher vom Typ Boolean (TRUE oder FALSE) sein muss. Wird TRUE übergeben, wird kein Hex-String zurückgegeben, sondern die "rohe" Ausgabe in Bytes.

Dies kann dann sinnvoll sein, wenn man mit den Digest-Werten andere Berechnungen durchführen will. Datei **hash2.php**:

```
$sha1bytes = sha1("message digest", TRUE);
echo "Formatiert in Hexadezimal-Darstellung mittels bin2hex(): " . bin2hex($sha1bytes);
```

3. Hash-Algorithmen der Bibliothek „mhash“

Auf weitere Message Digests kann mit der Bibliothek **mhash** zugegriffen werden.

Unter *Windows* muss in der Datei php.ini die Zeile ;extension=php_mhash.dll auskommentieren (d.h. den Strichpunkt entfernen) und (für PHP 4) die Datei libmhash.dll aus dem Verzeichnis C:\php\dlls in das Verzeichnis C:\WINDOWS\SYSTEM (bzw. C:\WINDOWS\SYSTEM32 unter NT/XP) kopiert werden.

Unter *Linux* muss zunächst mhash kompiliert und installiert werden (zu beziehen unter <http://mhash.sourceforge.net/>), dann PHP zusätzlich mit der Option **--with-mhash** neu übersetzt werden. Die Anwendung ist ebenfalls recht einfach, Rückgabe erfolgt unformatiert in Bytes. Datei **hash3.php**:

```
$txt = "message digest";
$sha1hash = mhash(MHASH_SHA1, $txt);
$sha1hmac = mhash(MHASH_SHA1, $txt, "Chrstph");
echo "Der sha1-Hash von \"message digest\": \" . bin2hex($sha1hash) . \"\n\";
echo "Der sha1-HMac von \"message digest\": \" . bin2hex($sha1hmac);
```

Die Funktion **mhash()** nimmt wahlweise 2 oder 3 Parameter: der erste Parameter ist eine Konstante und definiert den zu verwendenden Algorithmus. Folgende Konstanten stehen zur Verfügung (mhash-Version 0.9.1, berücksichtigt werden nicht Varianten in der Länge der Digests und die nicht-kryptographischen Prüfsummen CRC32, CRC32B und Adler32):

Konstante	Kryptographische Würdigung
MHASH_MD2	MD2: 128 Bit, langsam, unsicher, entworfen für 8Bit-Systeme
MHASH_MD4	MD4: 128 Bit, schnell, extrem unsicher, früher häufig gebraucht
MHASH_MD5	MD5: 128 Bit, schnell, unsicher, häufig gebraucht
MHASH_SHA1	SHA1: 160 Bit, durchschnittliche Geschwindigkeit, unsicher, häufig gebraucht
MHASH_SHA256	SHA256: 256 Bit, eher langsam, sicher, (noch) selten gebraucht
MHASH_SHA384	SHA384: 384 Bit, langsam, sehr sicher, (noch) selten gebraucht
MHASH_SHA512	SHA512: 512 Bit, langsam, sehr sicher, (noch) selten gebraucht
MHASH_RIPEMD160	RIPEMD160: 160 Bit, eher langsam, sicher, eher selten gebraucht
MHASH_HAVAL256	Haval: 256 Bit, schnell, sicher mit Einschränkung, selten gebraucht
MHASH_TIGER	Tiger: 192 Bit, schnell (besonders auf 64-Bit-Rechnern), sicher, selten gebraucht
MHASH_GOST	Gost: 256 Bit, langsam, unsicher, selten gebraucht
MHASH_WHIRLPOOL	Whirlpool: 512 Bit, langsam, sehr sicher, selten gebraucht

Der zweite Parameter ist der String, zu dem der Hash-Wert berechnet werden soll. Der dritte Parameter ist optional: wird er angegeben, so wird nicht der Message Digest, sondern der **HMac** zum zugehörigen Algorithmus berechnet (vereinfacht gesagt ein Message Digest in Abhängigkeit zum angegebenen Schlüssel).

Hinweis: die oben aufgeführten Algorithmen sind die gebräuchlichsten, es gibt noch mehr (bzw. sind weitere in Planung). Der aktuelle Stand ist auf der mhash-Homepage abzufragen. Um herauszufinden, welche Algorithmen in der eigenen PHP-Distribution zur Verfügung stehen, dient dieser Code (Datei **hash4.php**):

```
for ($i = 0; $i <= mhash_count(); $i++)
    printf ("Hash: %s, Blockgroesse: %d\n", mhash_get_hash_name($i),
    mhash_get_block_size($i));
```

Empfehlung:

Im Regelfall sollte man zu SHA256 greifen. MD5 ist zwar ebenfalls sehr populär, gilt allerdings als nicht mehr sicher, da der Algorithmus vor kurzem „geknackt“ wurde (d.h. es gelang, in kürzester Zeit eine Eingabe zu finden, die einen bestimmten Output erzeugt) und mit 128-Bit nicht resistent genug gegen bestimmte Attacken (z.B. birthday attack) ist. Selbiges gilt mittlerweile auch für SHA1. Ausreichenden Schutz bieten derzeit nur noch Tiger, SHA2 (und höher) und Whirlpool.

4. Erzeugung von Schlüsseln

Interessant ist aus der mhash-Bibliothek noch die Funktion **mhash_keygen_s2k()**: mit ihr ist es möglich, aus einem Benutzer-Passwort oder Benutzer-Schlüssel einen solchen Schlüssel herzustellen, der auch als Schlüssel für kryptographische Anwendungen verwendet werden kann (Ein Benutzer verwendet i.d.R. bei der Eingabe nur Zeichen von 6 oder 7 Bit Länge und nicht die vollen 8 Bit). Dies geschieht über den Algorithmus **Salted S2K** (definiert in RFC 2440). Datei **keygen.php**:

```
$salt = substr(mhash(MHASH_SHA1, mt_rand()), 0, 8);  
$secretKey = mhash_keygen_s2k(MHASH_SHA1, "password", $salt, 16);
```

Der erste Parameter spezifiziert wieder den Hash-Algorithmus. Der zweite Parameter gibt das Benutzer-Passwort an. Der dritte Parameter ist ebenfalls ein String und der Salt zur Schlüsselerzeugung. Dieser sollte, wie hier gezeigt, ebenfalls aus einer Random-Quelle kommen und 8 Zeichen lang sein. Die Rückgabe ist ebenfalls ein String aus Bytes und nicht ohne Formatierung darstellbar.

5. Symmetrische Verschlüsselung mit „mcrypt“

Die Bibliothek **mcrypt** bietet uns verschiedene Block Cipher und Stream Cipher zur symmetrischen Verschlüsselung von Daten. Die Installation läuft so ab:

Unter *Windows* muss in der Datei `php.ini` die Zeile `;extension=php_mcrypt.dll` auskommentiert werden. Zusätzlich muss die Datei `libmcrypt.dll` von <http://ftp.emini.dk/pub/php/win32/mcrypt> heruntergeladen werden und ins Windows-Systemverzeichnis kopiert werden.

Unter *Linux* muss zunächst `mcrypt` kompiliert und installiert werden (zu beziehen unter <http://mcrypt.sourceforge.net>), dann PHP zusätzlich mit der Option **--with-mcrypt** neu übersetzt werden.

Wir gehen davon aus, dass `mcrypt` in der Version 2.4.0 oder höher vorliegt.

Sehen wir uns gleich ein komplettes Beispiel an, in dem ein Text zuerst ver- und anschließend entschlüsselt wird. Erklärungen folgen anschließend. Datei **crypt1.php**:

```

// Handler erzeugen
$handler = mcrypt_module_open('blowfish', '', 'cbc', '');

// IV erzeugen
$iv = mcrypt_create_iv(mcrypt_enc_get_iv_size($handler), MCRYPT_RAND);

// Schlüssel erzeugen
$salt = substr(mhash(MHASH_SHA1, mt_rand()), 0, 8);
$secretKey = mhash_keygen_s2k(MHASH_SHA1, "password", $salt,
mcrypt_enc_get_key_size($handler));

// Verschlüsselung
mcrypt_generic_init($handler, $secretKey, $iv);
$encrypted = mcrypt_generic($handler, "wichtige Daten...");
mcrypt_generic_deinit($handler);
printf ("verschluesselt: %s\n", bin2hex($encrypted));

//Entschlüsselung
mcrypt_generic_init($handler, $secretKey, $iv);
$decrypted = mdecrypt_generic($handler, $encrypted);
mcrypt_generic_deinit($handler);
printf ("entschluesselt: %s\n", trim($decrypted));

// Handler freigeben
mcrypt_module_close($handler);

```

Zunächst muss der entsprechende Cipher mit der Funktion **mcrypt_module_open()** ausgewählt werden. Der erste Parameter gibt den gewünschten Algorithmus an, der dritte den gewünschten Modus (dazu jeweils später mehr). Parameter 2 und 4 interessieren uns jetzt nicht, wir übergeben nur einen Leer-String. Die Funktion liefert einen Handler zurück, der für die folgenden Funktionen gebraucht wird.

Anschließend muss ein IV (initial vector) mit der Funktion **mcrypt_create_iv()** passend zum jeweiligen Algorithmus erzeugt werden. Dann fragen wir mit **mcrypt_enc_get_key_size()** die erforderliche Länge des Schlüssels ab und erzeugen diesen. Bevor nun verschlüsselt werden kann, muss der Cipher initialisiert werden. Dies geht über die Funktion **mcrypt_generic_init()**, die außer dem Handler den Schlüssel und den IV erhält. Die Verschlüsselung selbst geschieht mit der Funktion **mcrypt_generic**. Zurückgeliefert wird der verschlüsselte Text (ebenfalls ein Byte-String). Zuletzt muss mit **mcrypt_generic_deinit()** die Verschlüsselung beendet werden. Die Entschlüsselung funktioniert genauso, es wird lediglich anstelle von **mcrypt_generic()** die Funktion **mdecrypt_generic()** aufgerufen. Wird der Cipher nicht mehr gebraucht, muss der Handler mit **mcrypt_module_close** freigegeben werden. Weitere Funktionen zu diesem Modul sind unter <http://www.php.net/manual/de/ref.mcrypt.php> zu finden.

Mit folgendem Snippet können alle verfügbaren Cipher abgefragt werden (der Name muss als erstes Argument an **mcrypt_module_open()** übergeben werden, wir erinnern uns). Datei **crypt2.php**:

```

$algos = mcrypt_list_algorithms();
foreach ($algos as $algo)
    echo $algo . "\n";

```

Bei mir waren folgende *Block Cipher* definiert (mcrypt-Version 2.6.4):
blowfish, blowfish-compatible, twofish, cast-128, cast-256, des, tripledes, serpent, xtea, rijndael-128, rijndael-192, rijndael-256, rc2, gost, saferplus, loki97
 Und folgende *Stream Cipher*:
arcfour, enigma, wake

Empfehlung:

Im Regelfall Blowfish verwenden, ist pfeilschnell und sehr sicher. Für maximale Sicherheit sollte man TripleDES verwenden (allerdings sehr langsam). Als sicher kann nach dem derzeitigen Stand ebenfalls gelten: Serpent, Twofish, CAST-128, CAST-256. Rijndael (auch bekannt als AES) zeigt in der Kryptoanalyse Schwächen, wurde aber noch nicht gebrochen. Alle anderen Algorithmen zeigen zu große Schwächen, als dass ihr Einsatz ernsthaft in Betracht gezogen werden sollte.

Stream Cipher sind generell mit Vorsicht zu genießen (*Bit-Flip-Attacken* sowie *Rekonstruktion des Schlüssels*). Wenn es wirklich nötig ist, sollte man zu Wake greifen.

Kommen wir nochmal auf den zu verwendenden Modus zurück (3. Parameter von `mcrypt_module_open()`). Block Cipher verschlüsseln Daten ja blockweise. Dies ist gegenüber Stream Cipher ein genereller Vorteil, jedoch wird der verschlüsselte Text gegenüber sog. *Copy&Paste-Attacken* verwundbar. Es gibt nun verschiedene Modi, um die Ausgabe eines Blocks auf den nächsten rückwirken zu lassen, um eben diese Attacken zu vereiteln. mcrypt kennt folgende Modi:

Modus	Bedeutung
ecb	electronic codebook (normale Verschlüsselung ohne Feedback, geeignet für Schlüssel)
cbc	cipher block chaining (blockweises Feedback, beste Sicherheit bei der Verschlüsselung von Dateien)
cfb	cipher feedback (macht einen Block Cipher Stream-fähig, d.h. einzelne Verschlüsselung von Bytes)
ofb	output feedback (8 Bit (unsicher), wie cfb, aber geeignet, um "Duchschlagen" eines Fehler wegen des Feedback zu unterbinden)
nofb	output feedback (allerdings in der Blocklänge des Ciphers, daher sicherer)
stream	Modus für Stream Cipher

6. Wichtige Hinweise

Das größte Problem in der Kryptographie ist nicht das mathematische Verfahren an sich, sondern der Umgang mit den Schlüsseln. Dafür ist jeder einzelne verantwortlich, und generell kann man sagen: die Paranoia kann nicht groß genug sein, wenn es sich um wichtige Daten handelt. *Niemand anderes darf den Schlüssel in die Hand bekommen!*

Generell gilt: den Schlüssel durch einen **kryptographischen Zufallszahlengenerator** (PRNG) erzeugen, *nur einmal (!) verwenden und nicht auf einem Computer speichern!* Das größte Problem ist die Schlüsselübertragung, da der Empfänger den selben Schlüssel zum Entschlüsseln braucht. Deswegen muss der Schlüssel selbst über einen sicheren Kanal übertragen werden. Bei PHP als Sprache, die im Normalfall serverseitig abläuft und dabei stark auf gespeicherte Daten angewiesen ist, ist das Problem ganz akut. Es stellt sich daher kritisch die Frage, ob solche Mechanismen (symmetrische Verschlüsselung) für eine Homepage, ein Forum o.Ä. geeignet sind. Eher macht es Sinn, damit Daten auf dem heimischen Rechner zu verschlüsseln und den Schlüssel anderweitig aufzubewahren.

Wie eingangs schon erwähnt, kann das Problem der Schlüsselübertragung durch asymmetrische Verfahren gelöst werden. Glücklicherweise steht für PHP auch eine Schnittstelle zur OpenSSL-Bibliothek bereit, so dass diese Verfahren auch von PHP aus genutzt werden können. Dieser Bereich ist allerdings eine Wissenschaft für sich und würde den Rahmen zumindest dieses Tutorials sprengen. Wenn das Interesse an der Kryptographie groß ist, werde ich gerne in einem anderen Tutorial auf Public-Key-Systeme bzw. die OpenSSL-Funktionen in PHP eingehen.

Wer weitere Informationen und sicherheitstechnische Würdigungen zu einzelnen Algorithmen sucht, sollte einen Blick auf diese sehr gelungene Seite werfen: <http://www.cipherbox.de>

Wer sich für das Innenleben kryptographischer Algorithmen interessiert, sollte eine gehörige Portion an mathematischem Verständnis mitbringen und in der jeweiligen Programmiersprache sicher sein. Viele Bibliotheken werden allerdings mit Quelltexten ausgeliefert, so bspw. in C++ die Bibliotheken *Crypto++* oder *Botan*, in C das schon genannte *mcrypt*, *LibTomCrypt* sowie *OpenSSL*, in Java die Bibliotheken *GNU-Crypto*, *Bouncycastle* und *FlexiProvider*. Auch ich publiziere Quelltexte (in C++, Java, JavaScript). Übrigens finden sich auch bei [tutorials.de](http://www.tutorials.de) viele Links zu diesem Thema unter http://www.tutorials.de/webverzeichnis/Science/Math/Applications/Communication_Theory/Cryptography/Programming_Libraries.

Links zu weiteren Bibliotheken:

Crypto++	C++	http://www.eskimo.com/~weidai/cryptlib.html
Botan	C++	http://botan.randombit.net
LibTomCrypt	C	http://www.bizfactory4u.de/gnu/directory/security/crypt/libtomcrypt.html
OpenSSL	C	http://www.openssl.org
GNU-Crypto	Java	http://www.gnu.org/software/gnu-crypto
FlexiProvider	Java	http://www.flexiprovider.de
Bouncycastle	Java	http://www.bouncycastle.org

Christoph Bichlmeier
chris (at) mad-teaparty (dot) com
<http://www.mad-teaparty.com>
erstellt am: 17.10.2004, letztes Update: 6.4.2006