universität wien

AUSZUG DER DIPLOMARBEIT / EXCERPT OF THE DIPLOMA THESIS

Titel der Diplomarbeit / Title of the Diploma Thesis

„Kekse ohne Salz schmecken nicht“
Ein innovatives Unterrichtskonzept zur Vermittlung von Security im Webdatenbereich

verfasst von / submitted by

Simon Marik

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Magister der Naturwissenschaften (Mag.rer.nat.)

Wien, 2017 / Vienna, 2017

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	A 190 884 313
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Lehramtsstudium UniStG UF Informatik und Informatikmanagement UniStG - 10/2003 UF Geschichte, Sozialkunde und Polit. Bildg. UniStG - 10/2008
Betreut von / Supervisor:	ao. Univ.-Prof. Dipl.-Ing. Dr. Renate Motschnig
Mitbetreut von / Co-Supervisor:	Ass.-Prof. Mag. Dr. Christian Cenker

Standards in der Webentwicklung

0.0.0.1 Salted-Hash-Cookies

Ab der PHP Version 5.6 existierte durch `crypt()` eine überarbeitete Passwort-API, welche mittels `password_hash()` dazu im Stande ist, aus dem Stegreif sehr sichere Hashwerte für sensible Daten innerhalb von Cookies oder Passwörtern zu generieren. Diese API war von langer Hand geplant, da jeder nur erdenkliche Hashwert herkömmlicher Hashalgorithmen, wie z.B. von MD5, mittlerweile allesamt durch Brute Force (d.h. Ausprobieren aller möglichen Kombinationen) kompromittiert wurden und die entsprechenden Rainbow- bzw. Lookup-Tables (Datenstrukturen, die vorberechnete Hashpaare enthalten, um das Performanceproblem von Brute Force Attacken zu umgehen) sogar mit Google gefunden werden können.[1] Selbstverständlich gibt es solche Tabellen auch für komplexere Algorithmen, da durch den ewig wachsenden Bestand an Cloudfarmen auch immer mehr Rechenleistung zum Knacken solcher Algorithmen vorhanden ist. Um die selbst generierten Hashwerte nicht innerhalb diversester Suchmaschinen wiederzufinden, glauben manche Hobbyentwickler, dass das Problem durch eine krude Vermischung von MD5 mit Base64 gelöst wäre. Dem ist natürlich nicht so, da selbst für solche Kombinationsmöglichkeiten Rainbow-Tables existieren. Das einzige was hilfreich ist, sind sogenannte Salted-Hash-Verfahren, welche einen zufällig generierten Salt (d.h. Salz) in den Hashwert miteinberechnen, sodass dieser nicht mehr durch normale Tabellen auffindbar wird. Genauer gesagt wird der zufällig generierte Salt an das Ende des Strings angeheftet und muss somit bei jeder Validierung genau dort vorkommen. Dadurch, dass jeder Hashwert einen eigenen Salt besitzt, wäre der zu investierende Aufwand, den Hashwert zu knacken, enorm hoch und mit aktuellen CPU-Leistungen gar nicht realisierbar. Aktuell wird in PHP zur Verschlüsselung automatisch „BCrypt“ eingesetzt, da er zurzeit zu den sichersten Hashalgorithmen überhaupt zählt *(Abb. 2.3.13)*.[2]

```php
/* Hashwerterstellung mit BCrypt & Salt */
password_hash($cookiedata, PASSWORD_BCRYPT);

/* Überprüfung des Hashwerts mit Datenbankeintrag */
password_verify($cookiehash, $fetch->dbdata);
```

Abbildung 0.0.1: Minimalbeispiel zur Erstellung und Überprüfung eines „salted“ Hashwerts

Obwohl die sogenannte „Salt & Pepper“ Methode an und für sich sehr einfach umzusetzen ist, machen es selbst hochrangige WebanwendungsherstellerInnen nach wie vor nicht richtig. So speichert beispielsweise die allseits bekannte Blogsoftware „WordPress“ die Passwörter seiner NutzerInnen im einfachen Hashformat ohne jeglichem Salz. Wenn ein Angreifer nun Zugang zur

[1] s. *Code-Bude*: Sicheres Passwort Hashing mit Salts, (Stand: 12.10.2016).
[2] vgl. *Zander* (2014), S.420 f.

Passwortdatei eines WordPress-Blogs erhält, kann dieser schwache Passwörter mit vorkompromittierten Listen abgleichen, um sich somit Zugang zum jeweiligen Account zu verschaffen. Viel schlimmer ist es allerdings, dass WordPress einen Hash von diesem unsicheren Hashwert nutzt, um daraus jenes Cookie zu generieren, welches schlussendlich am Nutzerrechner abgespeichert wird. Somit können auch Cookies repliziert werden, um sich Zugang in einen WordPress-Blog Account zu verschaffen. Wie man sieht, ist das eines der Negativbeispiele in Sachen schlecht generierter Hashwerte. Aus diesem Grund wird die korrekte Vorgehensweise eines Salted-Hash-Verfahrens nochmals in der folgenden Abbildung *(Abb. 2.3.14)* schematisch dargestellt.[3]

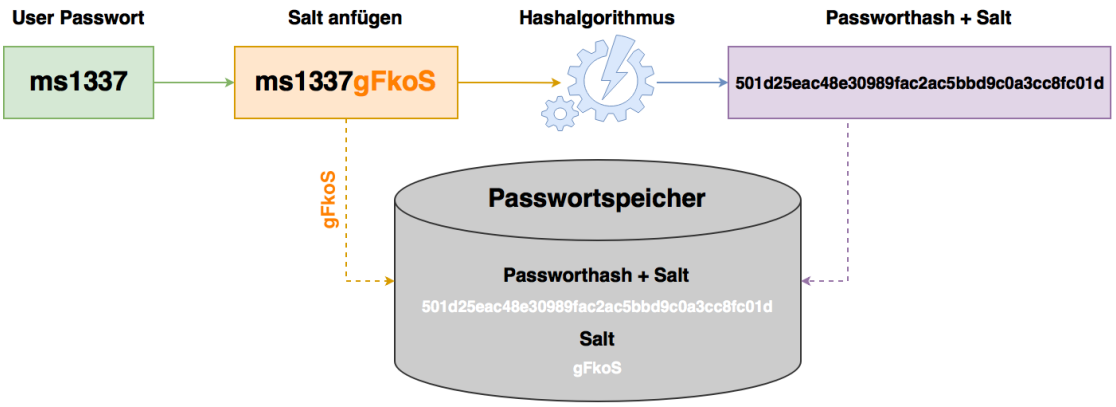

Abbildung 0.0.2: Salted-Hash-Verfahren bei gespeicherten Benutzerpasswörtern

Natürlich gibt es auch weitere häufig auftretende Fehler sowie falsche Mythen rund um Salt basierte Hashverfahren, welche nun in der folgenden Liste näher erläutert werden:

- Hashvorgang erfolgt clientseitig: Die Validierung sensibler Daten muss immer serverseitig geschehen und darf niemals benutzerseitig mittels JavaScript vorgenommen werden. Dabei könnten Angreifer Hashwerte auslesen und somit Vollzugriff auf eine Anwendung erlangen. Außerdem teilt man bei der clientseitigen Verarbeitung den Wert des Salt bzw. des verwendeten Hashalgorithmus mit.
- Doppelte Verwendung von Hashalgorithmen wie vorhin bereits erläutert
- Mehrfachnutzung des Salt: Sehr häufig verwenden Entwickler in ihren Webprojekten immer den gleichen Salt. Vor allem dann, wenn er einfach hart in eine PHP-Datei hinein programmiert bzw. nur einmal generiert und anschließend immer wieder verwendet wird. Angreifer müssen hier lediglich einen einzigen Salt an jeden Hashwert anfügen, um auf alle Daten zeitgleich zugreifen zu können.
- Wahl eines zu kurzen Salt: Wird ein zu kleiner Salt gewählt, werden die zu verschlüsselnden sensiblen Daten nur sehr geringfügig verändert. Auf diese Art und Weise lassen sich ebenfalls sehr leicht Lookup- oder Rainbow-Tables erstellen. Ein Salt sollte immer zumindest halb so groß wie der eigentliche Hashwert sein bzw. im Idealfall sogar gleich groß.[4]

0.0.0.1.1 SHA1 En-/Decryption

Der „Secure Hash Algorithm“ (SHA) beschreibt einen äußerst sicheren Verschlüsselungsalgorithmus, welcher vom „National Institute of Standards and Technology“ (NIST) in Verbindung mit der „National Security Agency“ (NSA) entwickelt wurde und innerhalb des „Digital Signature Algorithm“ (DSA) für digitale Signaturen verwendet wird. Dieser wurde im

[3] vgl. *Anderson* (2008), S.57.
[4] vgl. *Schäfers* (2016), S.224 f.

Jahr 1993 als ein sogenannter „Federal Information Processing Standard“ namens „FIPS 180“ erstmalig herausgegeben und zwei Jahre später überarbeitet (FIPS 180-1), wodurch sich einige Eckpfeiler rund um SHA1 entwickeln konnten:

- Funktioniert bei Nachrichtenlängen mit bis zu 2^{64} Zeichen
- In PHP über die Methode `sha1()` aufrufbar
- Erzeugt einen 160-Bit Hashwert
- Füllt eine Nachricht um ein Vielfaches von 512 Bits auf (Länge = 448 mod 512)
- Arbeitet achtzig verschiedene Operationen alleine im Hauptalgorithmus ab
- Führt je eine nichtlineare Operation an drei der fünf Variablen A, B, C, D und E pro Operation aus
- Nutzt eine Sequenz logischer Funktionen $f_{0'}, f_{1'}, ..., f_{79'}$, in der jede Funktion $f_{t'}$, bei der 0 $\leq$ t $\leq$ 79 gilt, an drei 32-Bit Wörtern arbeitet und ein neues 32-Bit Wort ausgibt[5]

Wie im FIPS 180-1 bzw. im RFC 3174[6] beschrieben, verarbeitet der SHA-Algorithmus Nachrichten mit beinahe unendlicher Länge, wobei die einzelnen Blöcke eine maximale Länge von 512 Bit zählen. Um die maximale Blocklänge zu erreichen, füllt SHA1 genau gleich wie Base64 die Leerstellen mit Füllbits oder sogenannten „Paddingbits“ auf, damit die Eingabedaten ein Vielfaches von 512 Bit betragen. Daraufhin folgt eine Zusammenkettung der einzelnen Blöcke, gefolgt von einer Aufteilung in 32-Bit Teile und einer anschließenden neuen Verkettung *(Abb. 2.3.11)*. SHA1 operiert mittels eines Eingabeblocks von 512 Bit Länge, einer Prüfsumme von 160 Bit und 80 Schritten pro Datenblock. Dazu kommen noch einige nichtlineare Funktionen sowie mehrere Additionskonstanten hinzu. Als Nachfolger von SHA1 hat die NIST SHA2 standardisiert. Unter diese Bezeichnung fallen Algorithmen wie SHA224, SHA256, SHA384 oder SHA512, welche zu den sichersten Hash-Algorithmen überhaupt gehören und allesamt von PHP unterstützt werden.[7]

Das Auffüllen der Nachrichten passiert im Wesentlichen deswegen, damit am Schluss eine Nachricht mit 512 Bit Länge finalisiert werden kann. Dafür wird als erstes eine 1 angeheftet, welcher im Anschluss so viele Nullen wie nötig folgen, um es auf 64 Bit eines Vielfachen von 512 Bit zu kürzen. Schlussendlich wird ein 64-Bit Integer am Ende der Nullen angehängt, um die finale Länge von n $*$ 512 Bits zu erreichen. Dabei repräsentiert der Wert der Integer die Länge der ursprünglichen Nachricht. SHA1 teilt dabei den Input sequentiell in 512-Bit Blöcke auf und arbeitet diese der Reihe nach ab, währenddessen der Hashwert abschließend erstellt wird.[8] Zuvor muss die Nachricht allerdings noch in n M-Bit Blöcke geparst werden bevor die Hashgenerierung starten kann. Bei SHA1 wird die aufgefüllte Nachricht demnach in n 512-Bit Blöcke $M^{(1)}, M^{(2)}, ..., M^{(n)}$ geparst. Durch die 512 Bit des Eingabeblocks lässt sich auf insgesamt sechzehn 32-Bit Wörter schließen, wobei die ersten 32 Bit des Nachrichtenblocks i als $M_0{}^{i}$ gekennzeichnet sind, die nächsten 32 Bit als $M_1{}^{i}$, bis hin zu $M_{15}{}^{i}$. Um den Hashwert zu berechnen, verwendet SHA1 zwei unterschiedliche Puffer, welche je aus fünf 32-Bit Wörtern und einer Sequenz von achtzig 32-Bit Wörtern besteht. Die Wörter des ersten Puffers werden als H_0, H_1, H_2, H_3 und H_4 bezeichnet, während die Wörter des zweiten Puffers a, b, c, d und e genannt werden und die Elemente der letzten achtziger Sequenz als $W_0, W_1, ..., W_{79}$ gekennzeichnet sind.[9]

0.0.0.1.2 „BCrypt“ Blowfish-Algorithmus

Im Jahr 1993 wurde der Blowfish-Verschlüsselungsalgorithmus von Bruce Schneier entwickelt, welcher erst im darauffolgenden Jahr publiziert wurde und ab diesem Zeitpunkt frei verfügbar war.

[5] vgl. *Mogollon* (2007), S.131.
[6] s. *MySQL*: Encryption Functions, (Stand: 11.10.2016).
[7] s. *ITWissen*: SHA (Secure Hash Algorithm), (Stand: 11.10.2016).
[8] vgl. *Rhee* (2003), S.149.
[9] vgl. *Mogollon* (2007), S.132.

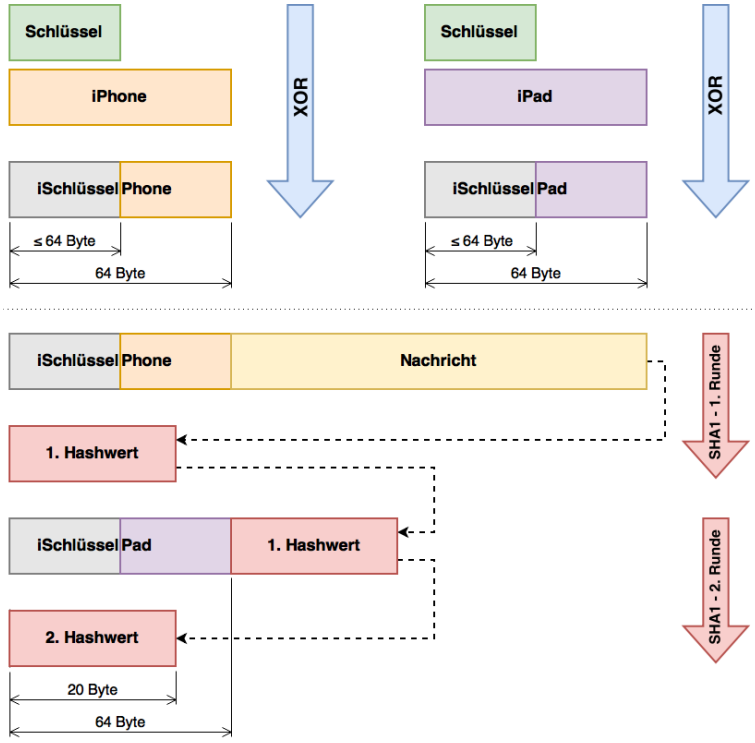

Abbildung 0.0.3: Schematische Darstellung eines SHA1 Verschlüsselungszyklus

Ursprünglich wurde der Algorithmus hauptsächlich in der Mikrocontrollerelektronik verwendet, obwohl er sich auch bestens für die Verschlüsselung sensibler Daten – z.B. in Datenbanken bzw. für sichere Kommunikationskanäle – mit seltenem Schlüsselwechsel eignet. Blowfish verwendet, im Gegensatz zu SHA, 64-Bit Blöcke und kann Schlüssel mit einer Länge von bis zu 448 Bit einsetzen. Prinzipiell besteht der Algorithmus aus zwei Teilen, wobei lediglich der zweite Teil den Verschlüsselungsprozess umfasst. Der erste Teil löst den Schlüssel in verschiedene Teilschlüssel auf, sodass sich eine Gesamtlänge von 4168 Bit ergibt. Im zweiten Teil wird dann der 64-Bit Datenblock aufgeteilt und innerhalb von 16 Iterationen jeweils schlüsselabhängige Permutationen sowie schlüssel- und datenabhängige Substitutionen unterzogen. Die einzelnen Hälften werden danach iterativ durch mehrere S-Boxen (d.h. nichtlineare Substitutionsoperationen) geschickt, wobei die Substitutionsparameter dieser Boxen durch den Schlüssel generiert werden. Das ist auch einer der Hauptgründe, weshalb Blowfish so unempfindlich gegen kryptanalytische Angriffe ist.[10]

Beim Blowfish-Algorithmus wird ein 64-Bit Eingangsblock, welcher mit Ursprungsdaten befüllt ist, zu allererst in zwei 32-Bit Blöcke aufgeteilt. Die linken 32 Bit durchlaufen einen XOR-Gatter mit dem ersten Element eines P-Arrays, um eine Variable P' zu generieren. Anschließend läuft die neu erstellte Variable P' durch eine Transformationsfunktion S, wodurch sie anschließend, mit den rechten 32-Bit der Ursprungsdaten, erneut durch ein XOR-Gatter muss und dabei eine neue Variable S' erstellt. Dann wird der linke 32-Bit Block der Ursprungsdaten durch S' und der rechte 32-Bit Block durch P' ersetzt. Dieser Prozess wird schlussendlich 15 Iterationen lang mit gültigen Elementen des P-Arrays fortgesetzt. Die daraus resultierenden Variablen S' und P' werden dann noch ein allerletztes Mal mit den beiden letzten Elementen des P-Arrays (17 & 18) durch ein XOR-Gatter geschickt, bevor sie wieder zu einem 64-Bit Block zusammengefügt werden, welcher

[10] vgl. *Lassmann* (2006), S.390 f.

den verschlüsselten Text repräsentiert *(Abb. 2.3.12)*.[11]

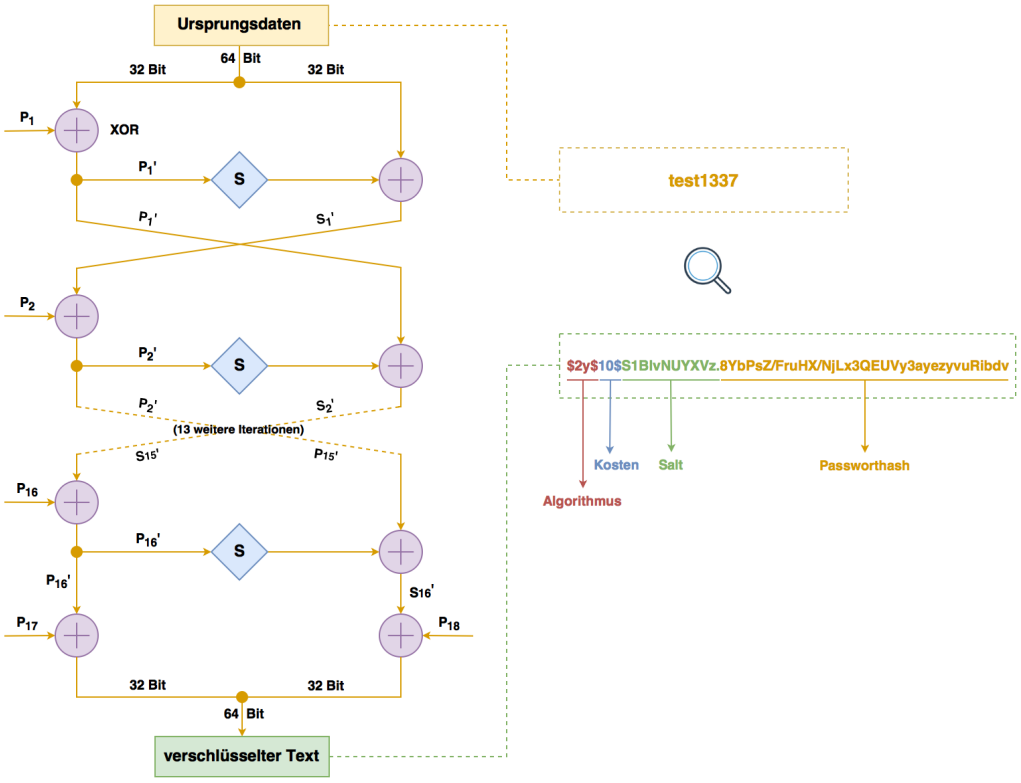

Abbildung 0.0.4: Ablaufdiagramm eines Verschlüsselungszyklus des Blowfish-Algorithmus

Herkömmliche Hashalgorithmen wie MD5 oder SHA1 sind einzig und allein für Performance optimiert, was zwar einerseits gut ist, um Datentransfers oder Zertifikate zu überprüfen, andererseits aber sehr leicht umkehrbar ist und sich somit eher weniger für die Verschlüsselung von Passwörtern eignet. Selbstverständlich kann man an dieser Stelle auch ohne Blowfish-Algorithmus mit ganz einfachen Salts arbeiten, allerdings eignet sich dieser perfekt aufgrund der datenbankseitigen Speicherung. Blowfish ist nämlich sehr langsam und das ist ideal, um Hackerangriffen vorzubeugen. Der Hashwert beinhaltet dabei einen Wert für die Iterationen – die sogenannten Kosten (Aufwand für die Berechnung) – sowie für einen individuell gewählten Salt. In PHP ist „BCrypt“ über die `crypt()` Funktion seit der Version 5.3 integriert und mittels einer der Kennungen `$2a$`, `$2x$` oder `$2y$` am Anfang jedes Hashwerts deutlich erkennbar *(Abb. 2.3.12)*.[12] Es gibt jedoch einige Funktionen, welche ärgerlicherweise im Blowfish Algorithmus nicht vorgesehen sind. So ist es z.B. nicht möglich, einen geheimen Salt innerhalb einer Webanwendung zu hinterlegen oder den Hashwert von zusätzlichen Faktoren wie Benutzernamen oder User-IDs abhängig zu machen.[13]

Einige namhafte Kryptographen wie Sarge Vaudenay oder Vincent Rijmen haben Blowfish auf etwaige Schwachstellen und Fehler untersucht und sind dabei auf zwei Erkenntnisse gestoßen. Einerseits können bei schwachen Schlüsseln, bei denen z.B. mehrere Werte gleichzeitig vorkommen, bestimmte Teile ermittelt werden, sofern weniger als 14 Iterationen durchlaufen werden. Des Weiteren wurde sogar von Rijmen mittels einer sogenannten „Second-Order-Differential-Attack“ versucht, den Blowfish-Algorithmus zu knacken. Dieser Angriff war bis zur vierten Iteration erfolgreich, schlug allerdings – und das betrifft die meisten Vorhaben in diesen

[11] s. *Gatliff*: Encrypting Data with the Blowfish Algorithm, (Stand: 12.10.2016).
[12] s. *PHP.net*: Funktion crypt, (Stand: 12.10.2016).
[13] s. *PHP Gangsta*: Schöner hashen mit BCrypt, (Stand: 12.10.2016).

Belangen – bei jeder weiteren fehl, wodurch der Blowfish-Algorithmus bis heute noch nicht geknackt wurde. Gerade deswegen erfreut sich der Blowfish-Algorithmus unter Entwicklern so großer Beliebtheit und kommt seit 1994 in zahlreichen kommerziellen Web- sowie Softwareanwendungen wie z.B. „OpenBSD“, „TrueCrypt & VeraCrypt“, oder „BestCrypt“ zum Einsatz, in denen sensible Daten verschlüsselt werden sollen.[14]

[14] s. *Doumack*: Symmetrische Verschlüsselung mit Blowfish-Algorithmus, (Stand: 12.10.2016).

Literatur- & Quellenverzeichnis

Anderson, Ross J. (2008): Security Engineering - A Guide to Building Dependable Distributed Systems; 2. Auflage, Wiley Publishing, Indianapolis.

Lassmann, Wolfgang (2006): Wirtschaftsinformatik - Nachschlagewerk für Studium und Praxis; 1. Auflage, Gabler Verlag, Wiesbaden.

Mogollon, Manuel (2007): Cryptography and Security Services - Mechanisms and Applications; 1. Auflage, Cybertech Publishing, Hershey.

Rhee, Man Young (2003): Internet Security - Cryptographic Principles, Algorithms and Protocols; 1. Auflage, Wiley Publishing, Chichester.

Schäfers, Tim Philipp (2016): Hacking im Web - Denken Sie wie ein Hacker und schließen Sie die Lücken in ihrer Webapplikation, bevor diese zum Einfallstor für Angreifer wird; 1. Auflage, Franzis Verlag, München.

Zander, Tobias (2014): Security im E-Commerce - Absicherung von Shopsystemen wie Magento, Shopware und OXID; 1. Auflage, Entwickler Press, Frankfurt.

Code-Bude: Sicheres Passwort Hashing mit Salts; siehe online unter: http://code-bude.net/2015/03/30/grundlagen-sicheres-passwort-hashing-mit-salts/ (Stand: 12.10.2016).

Doumack, Andrej: Symmetrische Verschlüsselung mit Blowfish-Algorithmus; siehe online unter: http://www.gm.fh-koeln.de/~hk/lehre/ala/ws0506/Praktikum/Projekt/C_rot/Blowfish.pdf (Stand: 12.10.2016).

Gatliff, Bill: Encrypting Data with the Blowfish Algorithm; siehe online unter: http://www.embedded.com/design/configurable-systems/4024599/Encrypting-data-with-the-Blowfish-algorithm (Stand: 12.10.2016).

ITWissen: SHA (Secure Hash Algorithm); siehe online unter: http://www.itwissen.info/definition/lexikon/secure-hash-algorithm-SHA-SHA-Algorithmus.html (Stand: 11.10.2016).

MySQL: Encryption Functions; siehe online unter: http://dev.mysql.com/doc/refman/5.7/en/encryption-functions.html#function_sha1 (Stand: 11.10.2016).

PHP Gangsta: Schöner hashen mit BCrypt; siehe online unter: http://www.phpgangsta.de/schoener-hashen-mit-bcrypt (Stand: 12.10.2016).

PHP.net: Funktion crypt; siehe online unter: http://php.net/manual/de/function.crypt.php (Stand: 12.10.2016).

Abbildungsverzeichnis