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

0.0.1 Cookies

Dadurch, dass HTTP ein zustandsloses Protokoll ist, geraten Daten spätestens dann in Vergessenheit, sobald die Website an den Client gesendet und die Verbindung wieder getrennt wurde. Im Jahr 1994 implementierte Netscape im hauseigenen Browser schlussendlich das Cookie, welches nur von jener Webseite gelesen werden konnte, von der es auch geschrieben wurde. Somit stellte es eine sichere Art dar, um Informationen dauerhaft über Seiten hinweg zu speichern, obwohl es anfänglich eher einem schlechten Ruf unterlag, da der Host darüber feststellen kann, wie oft eine BenutzerIn die jeweilige Webseite aufruft und was dort gemacht wird. Viele Menschen hatten plötzlich große Sorge um ihre Privatsphäre innerhalb des Internets, wodurch unter anderem auch zahlreiche Gerüchte entstanden. Beispielsweise wurde Cookies nachgesagt, dass sie jede Information auf der Festplatte lesen könnten, wodurch viele Zeitschriften und Blogs auf diesen Zug aufsprangen und sogar zur Anwendungsdeaktivierung von Cookies im Webbrowser rieten. Mittlerweile hat sich die vorerst angespannte Situation wieder beruhigt und Cookies werden im Allgemeinen von der Mehrheit akzeptiert.[1]
Prinzipiell stellen Cookies lediglich reine Textinformationen dar, welche auf Aufforderung eines Webservers durch den Internetbrowser auf der Festplatte des Clients abgespeichert werden.[2] Wenn die gleiche Webseite oder eine Seite der gleichen Domäne erneut aufgerufen wird, so schickt der Browser die im Cookie enthaltenen Textinformationen an den Webserver zurück. Der Webserver selbst ist aber lediglich in der Lage, ihm bereits bekannte Informationen im Cookie abzuspeichern, sodass auch nur solche Informationen vom Webbrowser an ihn zurückgesandt werden. Die oben genannte Einschränkung kann aber auch soweit technisch umgangen werden, dass andere Domänen ebenfalls Gebrauch von einem abgespeicherten Cookie machen können. In diesem Fall spricht man allerdings von Cookies von Drittanbietern *(Abb. 2.3.7)*.[3] Selbstverständlich besteht auch die Möglichkeit, in aktuellen Webbrowsern wie Safari, Chrome, Firefox oder Opera, die Speicherung von Cookies komplett zu deaktivieren. Diese Option kann aber unter Umständen dazu führen, dass der ordnungsgemäße Aufbau einer Seite, sowie ein reibungsloses Surfen an sich, nicht mehr gewährleistet werden kann.[4] Grundsätzlich kann man also sagen, dass es sich bei Cookies im weiteren Sinne um eine Art Cachespeicher handelt, durch den eine bestimmte Anzahl von Informationen gespeichert wird, wodurch eine BenutzerIn vom Webserver wiedererkannt werden kann.[5]

Als WebentwicklerIn kann man sowohl auf Cookies als auch auf Sessions problemlos zugreifen. Es ist schlussendlich mit beiden Techniken möglich, Daten einer Seite über mehrere andere hinweg zu speichern, wobei diese auch einige Unterschiede aufweisen. Cookies können nämlich – ganz im Unterschied zu Sessions – auf eine lange Lebensdauer eingestellt werden, wodurch Da-

[1] vgl. *Hudson* (2005), S.181.
[2] s. *Cookie Central*: The Cookie Concept, (Stand: 04.10.2016).
[3] vgl. *Peyton* (2002), S.53 f.
[4] vgl. *Christl* (2014), S.21.
[5] vgl. *Voss* (2003), S.229.

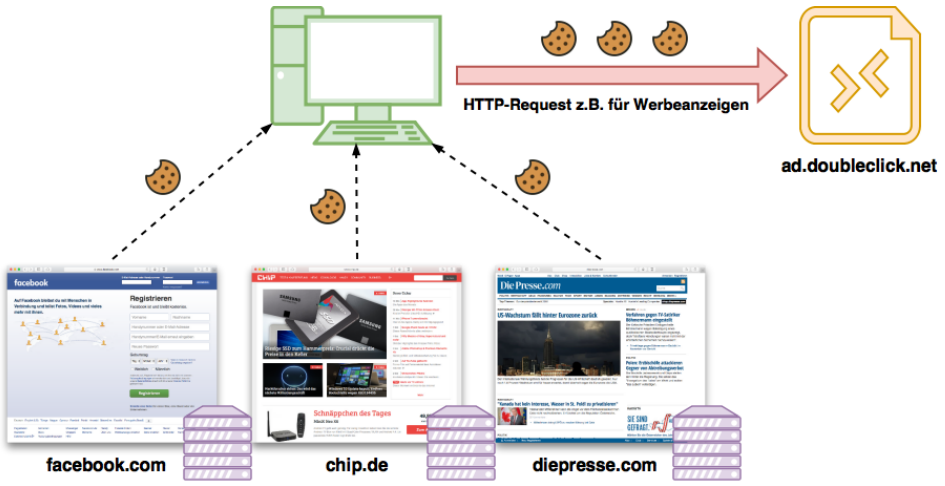

Abbildung 0.0.1: Abfrage von Cookieinformationen durch Drittanbieter anderer Domänen

ten innerhalb eines Cookies über Monate bzw. Jahre gespeichert werden können. Da Cookiedaten clientseitig gespeichert werden, ist die Speicherung sensibler Daten allerdings nur dann sinnvoll, wenn z.B. in einem Cluster mehrerer Webserver gearbeitet wird. Sessions wären hier nicht nützlich, da nach der ersten fertig abgearbeiteten Anfrage eines Webservers die Informationen im Cluster schlichtweg nicht mehr verfügbar wären. Außerdem weisen viele moderne Webbrowser eine Speicherplatzbegrenzung bei der Speicherung von Cookies auf, um schädliche Internetseiten davor zu hindern, immens viel Speicherplatz in Form von Cookies auf der Festplatte zu verschwenden. Schlussendlich beruht die Entscheidung, ob man in seinem Webprojekt nun Cookies oder doch lieber Sessions verwenden sollte darauf, ob die Daten noch am nächsten Tag für die BenutzerIn zur Verfügung stehen sollen. Falls dem so sein sollte, kommt man um Cookies nicht herum. Falls sensible Daten gespeichert werden sollen, ist es empfehlenswert, diese in einer gesicherten Datenbank abzulegen und lediglich eine Referenz zur ID im Cookie zu speichern. Dies deshalb, da Cookies dadurch, dass sie am Computer gespeichert werden, von BenutzerInnen sehr leicht bearbeitet werden können. So könnte man beispielsweise die Benutzer-ID, welche zur automatischen Anmeldung bei einer Webseite benötigt wird, umschreiben und somit in den Account der jeweils geänderten Benutzer-ID einsteigen. Genau deshalb werden heutzutage auch oft Sicherheitstoken innerhalb solcher Login-Systeme als eine Art zusätzliche Hackerbarriere implementiert, welche sich aus verschiedenen Benutzerinformationen zusammensetzen und mittels der Benutzer-ID und sonstigen Daten einen eindeutig feststellbaren Hashtag ergeben.[6]

Verlagert man nun die gesamte Thematik zurück zu PHP, muss unbedingt erwähnt werden, dass der Aufruf der Methode `setcookie()` immer vor dem HTML-Formular stehen muss, da HTTP alle Headerinformationen vor dem `<body>`-Tag sendet. In den Kopfzeilen werden Informationen wie beispielsweise der Webservertyp (z.B. Apache), die Seitengröße in Bytes sowie andere wichtige Daten übertragen. Cookies gehören ebenfalls zu diesen Headerinformationen, was heißt, dass nach jedem `setcookie()` der Webserver eine Zeile in den Header schreibt. Die `setcookie`-Funktion kann dabei drei Hauptparameter annehmen, nämlich den Cookienamen, den Wert sowie das Datum wann das Cookie wieder verfallen soll *(Abb. 2.3.8)*.[7] Im einfachen Fall wird in einem Cookie genau ein Wert gespeichert (max. 4 Kilobyte). PHP bietet aber auch die Möglichkeit, eine Arrayvariable als Cookie abzuspeichern, wodurch es ermöglicht wird, die Werte in einer Schleife abzuarbeiten. Für den Fall, dass noch komplexere Daten in einem Cookie gespeichert werden sollen, gibt es die Möglichkeit, den Variableninhalt mittels der Methode `serialize()` zu serialisieren,

[6] vgl. *Hudson* (2005), S.181 f.
[7] vgl. *ebd.*, S.182 f.

indem eine Zeichenkette aus der komplexen Variable erstellt wird.[8]

```
/* Erstellung eines Cookies namens "thesis-cookie" */
/* Ablaufdatum: aktuelle Zeit + 5000 Tage */
setcookie('thesis-cookie', $data, time() + (86400 * 5000), "/");

/* Entfernung des Cookies "thesis-cookie" */
/* Inhalt wird gelöscht / Ablaufdatum: aktuelle Zeit */
setcookie('thesis-cookie', '', time() - (86400 * 5000), "/");
```

Abbildung 0.0.2: Minimalbeispiel zur Erstellung und Entfernung eines Cookies

Der größte Nachteil von Cookies liegt sicherlich darin, dass man sich einfach nicht auf sie verlassen kann. Es lässt sich nämlich nicht mit Gewissheit sagen, ob ein Browser auch wirklich ein Cookie nach einer Sitzung gelöscht hat, oder ob sie eine BenutzerIn aufgrund von Sicherheitsbedenken eigenhändig entfernte bzw. ob sie sogar deaktiviert wurden. Darum ist es generell keine gute Idee, gerade bei sicherheitskritischen Bereichen wie z.B. Anmeldesystemen, Cookies zu verwenden, außer man rüstet sein Login-System mit einem Sicherheitstoken aus. Ein weiterer großer Nachteil von Cookies ist, dass man sie nur vor dem Ausliefern eines Seiteninhalts erstellen bzw. bearbeiten kann. Diese Einschränkung ist innerhalb der Spezifikation von HTTP geregelt und ist nicht PHP bedingt. Des Weiteren werden Cookies nach dem Aufruf der Methode `setcookie()` lediglich erstellt. Erst nach dem Laden der nächsten Seite wird das Cookie auch im Webbrowser als aktiv angezeigt, was ein häufig auftauchender „Fehler“ in so manchem Forum ist.[9]

0.0.1.1 Verschlüsselung von Cookies

Grundsätzlich sollte bei jeder Webanwendung eine Verschlüsselung zum Einsatz kommen, um die Möglichkeit eines potentiellen Angriffs zumindest reduzieren zu können. Normalerweise wird an dieser Stelle das HTTPS-Protokoll bzw. ein „HTTP Strict Transport Security“-Protokoll (HSTS) verwendet, um der NutzerIn eine direkte verschlüsselte Verbindung zum Webserver anbieten zu können, wodurch Session-Übernahmen und dergleichen verhindert werden. Sind die HTTPS-Funktionen am Webserver allerdings nicht konfiguriert, muss man zu anderen Verschlüsselungsmethoden zurückgreifen, um zumindest die abgespeicherten Cookies bestmöglich vor Fremden zu schützen. Verschlüsselung bedeutet aber im Wesentlichen nur, dass während einer Verbindung beispielsweise die Session-ID ausgelesen werden kann, was bedeuten soll, dass kein Schutz vor vorhersagbaren Session-IDs oder anderen Angriffsvektoren gegeben ist.[10] Eine Webseite kann darüber hinaus noch die sogenannte „Secure Cookie“ Option nutzen, wodurch überprüft wird, ob ein Cookie vor seiner Übertragung auch wirklich verschlüsselt ist. Diese Option ist vor allem dann sehr hilfreich, wenn ein Webprojekt aus einer Kombination von mehreren verschlüsselten sowie unverschlüsselten Seiten besteht.[11]

Möchte man jedoch keine Zeit mit SSL verschwenden, kann durch das Zusammenspiel von Cookie-Integrität und Cookie-Authentifizierung innerhalb moderner Webprojekte eine vollkommen gleichwertige Cookie-Verschlüsselung und die damit verbundene Diskretion geboten werden. Die notwendige Authentifizierung wäre dabei z.B. durch einen Salt, ein Zertifikat oder einer digitalen Signatur gegeben, wodurch in weiterer Folge „Cookie-Replay-Attacks“ keine Chance

[8] vgl. *Kofler & Öggl* (2010), S.171.
[9] vgl. *ebd.*, S.171 f.
[10] vgl. *Schäfers* (2016), S.101 f.
[11] vgl. *Splaine* (2002), S.108.

mehr hätten. Ein weiterer Vorteil davon wäre beispielsweise der Schutz beim Transfer, sowie bei der Speicherung solcher Cookies, da verschlüsselte Kanäle wie beispielsweise HTTPS lediglich die bloße Übertragung absichern. Ein nennenswerter Nachteil solcher Verschlüsselungsmethoden besteht allerdings im serverseitigen Schlüsselmanagement der einzelnen „Keys“, welche für die Entschlüsselung benötigt werden.[12] Im Wesentlichen wird zwischen zwei großen Cookie-Verschlüsselungsmethoden unterschieden:

- Serverseitig verwaltete Cookie-Verschlüsselung: Bei diesem Zugang arbeitet der Webserver mit einer Kombination aus unterschiedlichen „Secure Cookies“ wie z.B. einem Username-Cookie (speichert den Benutzernamen für die Authentifizierung), einem Key-Cookie (für den Schlüssel) und einem Sicherheitstoken, welcher die Integrität durch einen zufällig gewählten Hashwert aufrechterhält. Gestohlene „Secure Cookies“ können allerdings nach wie vor an den Webserver gesendet werden, weswegen diese Methode keinen Schutz vor „Cookie-Replay-Attacks“ bietet.
- Clientseitig verwaltete Cookie-Verschlüsselung: Hier hat die BenutzerIn die Kontrolle über sämtliche Sicherheitsmechanismen seiner Cookies und kann Verschlüsselungs- bzw. Kodierverfahren wie z.B. Base64, SHA1 oder den Blowfish Algorithmus einbauen. Demnach ist aber auch das Schlüsselmanagement Aufgabe der BenutzerIn, was im weiteren Sinne ein wesentliches Sicherheitsrisiko darstellt, da die „Keys“ clientseitig gespeichert werden müssen. Um diese weitgehend abzusichern, eigenen sich die folgenden zwei Möglichkeiten:
    - Symmetrische Verschlüsselung: Die NutzerIn aktiviert die Cookie-Verschlüsselung mittels einer Anfrage an den Webserver
    - Asymmetrische Verschlüsselung: Die NutzerIn benötigt ein gültiges Zertifikat und ein asymmetrisches Schlüsselpaar[13]

Innerhalb von PHP ist das Verschlüsseln und Überprüfen von sensiblen Cookie-Daten mittels der Methoden `password_hash()` und `password_verify()` möglich. Die Methode `password_hash()` erstellt dabei einen neuen Hashwert aus den eingefügten Stammdaten, indem ein sehr starker Einweg-Hash-Algorithmus benutzt wird. An dieser Stelle ist auf die Methode `crypt()` zu verweisen, welche prinzipiell dasselbe macht, aber weniger Konfigurationsmöglichkeiten, z.B. bei der Wahl des Hash-Algorithmus, bietet. Außerdem werden verschiedene Optionen unterstützt, wie das manuelle Einfügen eines Salt-Parameters oder die Angabe der algorithmischen Kosten, welche stark von der Hardware abhängig sind.[14] Die Methode `password_verify()` verifiziert im Anschluss den übergebenen Hashwert mit den vorliegenden sensiblen Daten. Dabei übergibt die Methode den gewählten Algorithmus, die Kosten sowie den Salt-Parameter um Serverressourcen einzusparen und ist durch das Retournieren eines herkömmlichen Booleschen Werts vollkommen vor Laufzeitangriffen geschützt.[15]

0.0.1.1.1 Base64 Kodierung

Das „Base64-Encoding“ einer Datei ermöglicht die Kodierung der darin liegenden Zeichensätze in 64 Zeichen des ASCII-Codes. Dabei werden aus drei 8-Bit Sequenzen (je 3 Bytes) vier 6-Bit Sequenzen generiert, wodurch insgesamt 256 mögliche Zustände eines Bytes in die 64 möglichen Zustände der Base64 Kodierung umgewandelt werden können *(Abb. 2.3.10)*.[16] Der Kodierungsstandard wird bis heute, vor allem im E-Mailversand, eingesetzt, da er im „Simple Mail Transfer Protocol“ (SMTP) implementiert ist. So wird praktisch jede E-Mail und auch jeder Anhang wie beispielsweise Archiv- oder Bilddateien, vor dem Versand in Base64 umgewandelt.

[12] vgl. *Khu-smith & Mitchell* (2002), S.137.
[13] vgl. *ebd.*, S.138 ff.
[14] s. *PHP.net*: Funktion password_hash, (Stand: 10.10.2016).
[15] s. *PHP.net*: Funktion password_verify, (Stand: 10.10.2016).
[16] s. *Brünner*: Der Kodierungsstandard Base64, (Stand: 11.10.2016).

Bei der Kodierung eines herkömmlichen Klartextes vergrößert sich die Menge um etwa 33%, wobei Füllbytes in Form von Gleichheitszeichen am Ende entstehen können. Base64 kommt in den meisten Webanwendungen bei Cookies oder sonstigen Parametern zum Einsatz, da der Kodierungsstandard fest in PHP eingebettet ist und mittels der Methoden `base64_encode()` und `base64_decode()` gesteuert werden kann. Es ist jedoch möglich, durch entsprechende Tools, welche auch online zu finden sind, sowie durch bloße Umkehrung der Kodierung den kompletten Inhalt zu ermitteln, weswegen die Base64 Kodierung bei sensiblen Daten nicht eingesetzt werden sollte.[17]

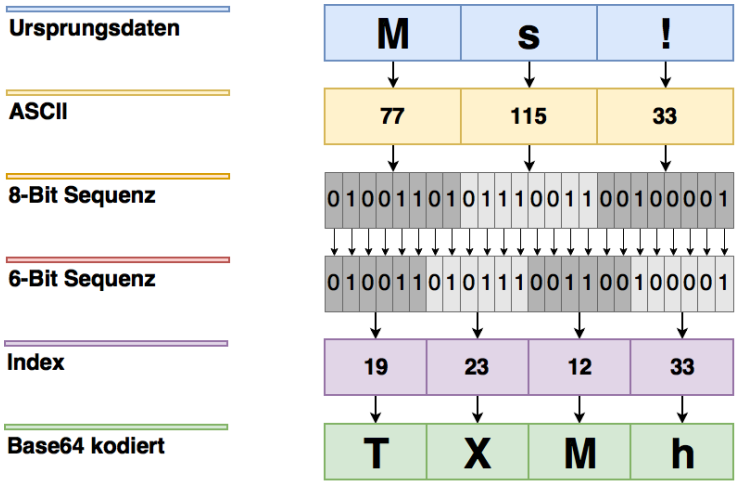

Abbildung 0.0.3: Kodierung einer Zeichenkette durch Base64

Dadurch, dass mit 6-Bit eine maximale Anzahl von 64 unterschiedlichen Werten dargestellt werden kann, wurde das Alphabet so konstruiert, dass es lediglich Zeichen des ASCII Zeichensatzes enthält und somit jederzeit ohne jeglichen Konvertierungsaufwand leicht übertragbar ist. Die folgende Tabelle *(Tab. 2.3.2)* listet das gesamte Base64 Alphabet und die dazugehörigen Indexwerte jedes Zeichens auf.[18]

Wert:	Kodierung:	Wert:	Kodierung:	Wert:	Kodierung:	Wert:	Kodierung:
0	A	**16**	Q	**32**	g	**48**	w
1	B	**17**	R	**33**	h	**49**	x
2	C	**18**	S	**34**	i	**50**	y
3	D	**19**	T	**35**	j	**51**	z
4	E	**20**	U	**36**	k	**52**	0
5	F	**21**	V	**37**	l	**53**	1
6	G	**22**	W	**38**	m	**54**	2
7	H	**23**	X	**39**	n	**55**	3
8	I	**24**	Y	**40**	o	**56**	4
9	J	**25**	Z	**41**	p	**57**	5
10	K	**26**	a	**42**	q	**58**	6
11	L	**27**	b	**43**	r	**59**	7
12	M	**28**	c	**44**	s	**60**	8
13	N	**29**	d	**45**	t	**61**	9
14	O	**30**	e	**46**	u	**62**	+
15	P	**31**	f	**47**	v	**63**	/
						64	== (Füllbits)

Tabelle 0.0.1: Das komplette Base64 Alphabet [19]

[17] vgl. *Schäfers* (2016), S.87.
[18] vgl. *Fuchs* (2001), S.109 f.

Wie man sieht, muss lediglich eine Sonderregelung für den Schluss einer jeden Zeichenkette getroffen werden, da nicht garantiert werden kann, dass am Schluss auch genau 24-Bit überbleiben. Da ein Byte immer aus 8 Bit besteht und somit kein anderer Fall existieren kann, unterscheidet man generell zwischen drei unterschiedlichen Fällen:

- Es bleiben 24 Bit über: nichts wird unternommen (bereits vier 6-Bit Sequenzen)
- Es bleiben 8 Bit über: vier 0-Bit Sequenzen werden angehängt (daraus entstehen zwei 6-Bit Sequenzen - fehlende 12 Bit werden durch „==“ Füllbits ersetzt)
- Es bleiben 16 Bit über: zwei 0-Bit Sequenzen werden angehängt (daraus entstehen zwei 3-Bit Sequenzen - fehlende 8 Bit werden durch „==“ Füllbits ersetzt)[20]

0.0.1.2 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.[21] 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)*.[22]

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

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

Abbildung 0.0.4: 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

[19] *Fuchs* (2001), S.110.
[20] vgl. *ebd.*
[21] s. *Code-Bude*: Sicheres Passwort Hashing mit Salts, (Stand: 12.10.2016).
[22] vgl. *Zander* (2014), S.420 f.

speichert beispielsweise die allseits bekannte Blogsoftware „WordPress“ die Passwörter seiner NutzerInnen im einfachen Hashformat ohne jeglichem Salz. Wenn ein Angreifer nun Zugang zur 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.[23]

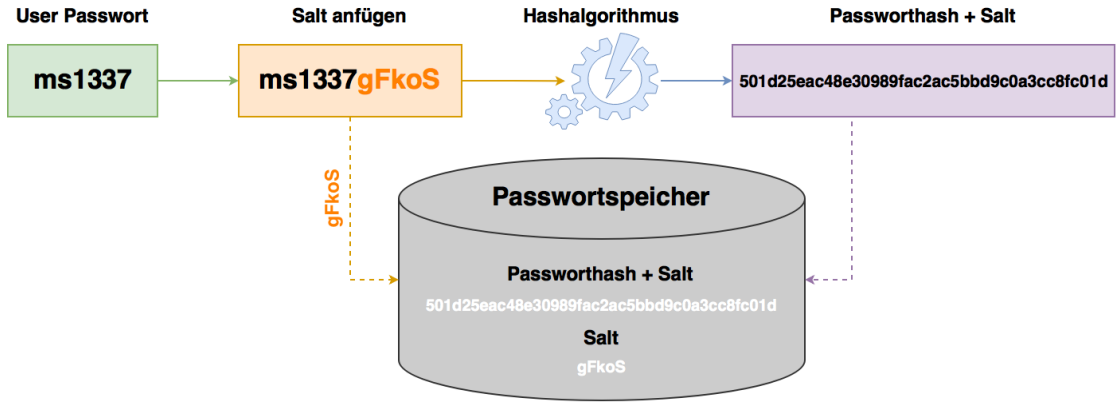

Abbildung 0.0.5: Salted-Hash-Verfahren bei gespeicherten Benutzerpasswörtern

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.[24]

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

Literatur- & Quellenverzeichnis

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

Christl, Alexander (2014): Datenschutz im Internet - Cookies, Web-Logs, Location Based Services, eMail, Webbugs, Spyware; 1. Auflage, Disserta Verlag, Hamburg.

Fuchs, Florian (2001): Sicherheit von Internet- und Intranetdiensten; Universität Klagenfurt, Klagenfurt.

Hudson, Paul (2005): PHP in a nutshell; übersetzt von: Speidel, Sigrid & Ulrich; 1. Auflage, O'Reilly Verlag, Köln.

Kofler, Michael & Öggl, Bernd (2010): PHP 5.3 & MySQL 5.4 - Programmierung, Administration, Praxisprojekte; 1. Auflage, Addison-Wesley Verlag, München.

Khu-smith, Vorapranee & Mitchell, Chris; Kim, Kwangjo (Hrsg.) **(2002):** Information Security and Cryptology - ICISC 2001; 1. Auflage, Springer Verlag, Seoul.

Peyton, Christine (2002): Das neue PC Lexikon für Alle; 1. Auflage, Sybex Verlag, Düsseldorf.

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.

Splaine, Steven (2002): Testing Web Security - Assessing the Security of Web Sites and Applications; 1. Auflage, Wiley Publishing, Indianapolis.

Voss, Andreas (2003): Das große PC & Internet Lexikon; 1. Auflage, Data Becker Verlag, Düsseldorf.

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

Brünner, Arndt: Der Kodierungsstandard Base64; siehe online unter: http://www.arndt-bruenner.de/mathe/scripts/base64.htm (Stand: 11.10.2016).

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).

Cookie Central: The Cookie Concept; siehe online unter: http://www.cookiecentral.com/c_concept.htm (Stand: 04.10.2016).

PHP.net: Funktion password_hash; siehe online unter: http://jp2.php.net/manual/en/function.password-hash.php (Stand: 10.10.2016).

PHP.net: Funktion password_verify; siehe online unter: http://jp2.php.net/manual/en/function.password-verify.php (Stand: 10.10.2016).

Abbildungsverzeichnis

Tabellenverzeichnis