February 27, 2007

Security Analysis

Hashing Credit Card Numbers: Unsafe Application Practices

Overview

Cryptographic hash functions seem to be an ideal method for protecting and securely storing credit card numbers in ecommerce and payment applications [1]. A hash function generates a secure, one-way digital fingerprint that is irreversible and meets frequent business requirements for searching and matching of card numbers. However, due to the predictability of credit card numbers and common business requirements in processing credit cards, ecommerce and payment applications may implement such hashing of card numbers in an unsafe manner that allows an attacker to obtain a large percentage of card numbers by brute forcing compromised hashes in a matter of hours.

This paper is an analysis of actual application practices for storing of credit card number hashes and a review of brute force attack methods against such hashes. The concepts presented in this paper have been broadly described prior by Kurt Seifried in 2001 [2], John Deters in 2002 [3], Branden Williams in 2006 [4], and many others, nevertheless some ecommerce and payment applications store credit card numbers in unsafe and easily brute forced ways. The impetus for this paper was identification of this issue during multiple application security assessments. The objective is to highlight the weakness of common credit card hashing techniques and to educate application architects and programmers on the issues of storing credit card numbers as hashes.

PCI, Card Numbers, and Business Requirements

1. Payment Card Industry Data Security Standard

PCI DSS Requirement 3.4 –

Render [credit card numbers], at minimum, unreadable anywhere it is stored (including data on portable digital media, backup media, in logs, and data received from or stored by wireless networks) by using any of the following approaches:

- *Strong one-way hash functions (hashed indexes)*
- *Truncation*
- *Index tokens and pads (pads must be securely stored)*
- *Strong cryptography with associated key management processes and procedures*

The Payment Card Industry (PCI) Data Security Standard (DSS) 1.1 [5] requirement is clear regarding the need to make credit card numbers unreadable. Many applications use encryption and/or hashing as primary methods to achieve compliance with Requirement 3.4. PCI DSS does allow for compensating controls for protecting card numbers, however, making card numbers unreadable is the preferred approach.

PCI DSS Requirement 3.3 –

Mask [credit card numbers] when displayed (the first six and last four digits are the maximum number of digits to be displayed).

PCI DSS Requirement 3.3 specifies that up to the first six and last four digits (Last-4) may be displayed to the user. Most applications do routinely display the Last-4 on web pages, application screens, reports, and receipts. Portions of the first six digits or a representation of the data (i.e., credit card brand) usually are displayed in back-end processing or are used for some business processes.

These two requirements map to similar requirements in the Visa U.S.A. CISP Payment Application Best Practices (Requirements 2.2 and 2.1, respectively) [6], which in the future may be a requirement for all payment processing applications.

When the entire credit card number is hashed, the application must store portions of the prefix and suffix in some manner to allow for retrieval and matching. The amount of information stored will vary based on business requirements and diligence of the application design. The PCI DSS requirements are ambiguous as to the acceptability of storing the first six (or portions of) and/or last four digits in plain-text, which is often done when the card number is hashed.

2. Predictability of Credit Card Numbers

Many of the digits of a credit card number are predictable based on card brand and other factors. For almost all card brands, the last digit is a check digit calculated using the Luhn checksum algorithm. For more information on credit card numbering, see the excellent Wikipedia entry on credit card numbering (http://en.wikipedia.org/wiki/Credit_card_numbers) [7].

For readability and for those not familiar credit card industry terms, the following simplified terms will be consistently used to describe portions of a credit card number –

Term	Description
Card Number	Full 14 to 16 digit account number, referred to as the Primary Account Number (PAN) in PCI DSS. Card numbers may be up to 19 digits, but most major brands are in the range of 14 to 16 digits.

Brand	The credit card brand – Visa, MasterCard, American Express, Discover, Diners Club, JCB, etc. Due to processing agreements between the brands, the brand will be defined as the brand indicated by the brand ID in the card number rather than the brand name on the card.
Brand ID	The first one to five digits that represents the brand – a brand may have multiple brand IDs. The Brand ID does not include the bank identifier for brands like Visa and MasterCard.
Common Prefix or Bank Prefix	The first three to six digits that are significant to a brand card number. This information is not generally available, but unauthorized lists of prefixes and bank identifiers are available on some Internet sites for the most popular card brands.
Prefix 6	The first six digits of the card number, regardless of brand or length.
Last 4	The last four digits of the card number, regardless of brand or length. This includes the last digit, which is the check digit.
Check Digit	The last digit of the card number, which for most brands is the check digit and is calculated using the Luhn checksum algorithm of the prior digits.

This paper will focus only on the most popular card brands in the United States and Europe. The five PCI Security Standards Council founding brands (American Express, Discover Financial Services, JCB, MasterCard Worldwide, and Visa International) and Diners Club are included. We also assume all remaining digits of the card number are randomly assigned by the brand and there is no algorithm or other method to pre-determine these digits – most likely this assumption is incorrect for some card brands.

3. Credit Card Processing Business Requirements

Common business requirements for processing and handling credit cards determine the amount of card data stored and what portions of the card number may be required to be retrieved after the initial transaction. These requirements will vary from merchant to merchant based on payment processor and type of retailer (i.e., brick and mortar versus ecommerce). Although in general, almost all merchants have some business requirements to handle customer returns/credits, reconciliation, chargebacks, and retrieval requests.

Ecommerce and Payment Applications

Usually ecommerce and payment applications must satisfy a number of common business requirements related to the processing and handling of customer payments using credit cards. These processes may be manual or occur through automated interfaces and batch processing. The following are common business processes that occur after the transaction –

- Chargeback – Chargeback reports only include the card number, transaction date, reference number, approval code, and/or transaction amount. The transaction and payment record must be accurately and efficiently located using the limited number of fields.
- Retrieval Request – Retrieval requests only include the card number, transaction date, reference number, and/or transaction amount. The transaction and payment record must be accurately and efficiently located using the limited number of fields.
- Customer Return or Credit – The card used in the original transaction is usually used to issue a credit. Common practice when the card number is hashed is to display the Last-4 to the employee and the full card number is re-entered for the credit transaction. Usually, the original transaction and payment record is identified using transaction ID from the receipt or customer number.
- Reconciliation – Daily or monthly credit card activity reports and payments need to be reconciled against the actual transactions to ensure all transactions were processed and the proper payment amount was credited. Activity reports often only include the card number, transaction date, reference number, approval code, and/or transaction amount. A common business requirement is to improve the reconciliation process by identifying the card brand or payment processor to facilitate the reconciliation process, therefore, portions of the prefix must be stored.

Data Mining and Data Warehousing Applications

Data mining and data warehousing applications have an entirely different set of business requirements. These applications typically are looking for trends in and summarization of transaction data, such as –

- Matching credit card use across transactions for customer profiling or fraud detection regardless of expiration date, cardholder name, or other transaction data.
- Summarization of card brand transaction volume to optimize marketing and fees (e.g., what is the average transaction amount for American Express Platinum vs. Green cards). Credit card processing fees are a significant business expense and merchants often look to manage this expense.

Common Application Practices

Credit Card Data Storage

Based on the business requirements and PCI DSS requirements, hashing is a suitable method of protecting and storing card numbers. Only hashing the card number is the minimum requirement for PCI compliance, therefore, in many applications this is the only cardholder data value protected. The card number is stored in some applications both encrypted and hashed to allow for efficient searching and matching of card numbers. In addition to the card number being hashed, some digits of the card number may be stored as plain-text to support the various business requirements outlined previously. The PCI DSS requirements 3.3 and 3.4 are ambiguous if storing in plaintext any digits is acceptable as requirement 3.3 allows displaying up to the first 6 digits and last 4 digits. Most ecommerce and payment applications that store credit card numbers hashed fall into one of the six following design patterns related to storing other digits in plaintext –

Pattern	Description
1. Card Number (hashed)	Only the card number is stored hashed and no digits are stored as plain-text. This pattern is usually in applications that also encrypt the card number. The card number is stored as a hash to allow for efficient searching and matching.
1. Card Number (hashed) **2. Brand**	The card brand (Visa, MasterCard, American Express, etc.) is stored in plain-text as a custom application value. The application will use the brand ID to programmatically determine the card brand. This pattern is usually in applications that also encrypt the card number. The brand is used for reconciliation or easy retrieve for chargebacks and retrieval requests.
1. Card Number (hashed) **2. Brand ID**	The card brand ID is stored in plaintext, which will be the first 1 to 5 digits of the card number. This pattern is usually in applications that also encrypt the card number. The brand is used for reconciliation or easy retrieve for chargebacks and retrieval requests.
1. Card Number (hashed) **2. Brand ID** **3. Last-4**	The card brand ID and last 4 digits are stored in plaintext. This seems to be a common design pattern and usually meets all necessary business requirements.
1. Card Number (hashed) **2. Last-4**	Only the last 4 digits are stored in plaintext to allow for returns and credits processing. The last-4 digits are displayed on receipts, web pages, application screens, and in reports.
1. Card Number (hashed) **2. Prefix-6** **3. Last-4**	This is the worst-case scenario where the first 6 digits and last 4 digits are stored in plaintext. The business requirement is that the brand as well as bank ID must be known.

HASH ALGORITHMS

Applications that hash credit card numbers typically use the widely accepted cryptographic hash functions MD5 and SHA-1, as these two functions are readily available on a wide variety of application platforms. The SHA-2 family of hash functions (SHA-224, SHA-256, SHA-384, and SHA-512), WHIRLPOOL (an ISO standard), and RIPEMD-160 are used to lesser degrees due to potential performance concerns, lack of availability on some application platforms, and perception that SHA-1 is sufficient for most application applications. MD4 and other older hash functions are seldom encountered in modern ecommerce and payment applications. The general opinion for PCI DSS is that industry-standard cryptographic hash functions or encryption algorithms should be used as the PCI Security Audit Procedures 1.1 [8] document specifically mentions SHA-1 by name, therefore, the use MD5 may be unacceptable as well as any custom developed hash functions.

Hash Algorithms Weaknesses

The weaknesses and issues found in the MD5 and SHA-1 hash algorithms are related to finding collisions where two strings produce the same hash ("collision-resistance") rather than any inherent flaw in the algorithm that helps to determine the original string ("preimage-resistance"). These weaknesses regarding collisions in the MD5 and SHA-1 hash algorithms are irrelevant in the discussion of hashing credit card numbers since only the original credit card number is meaningful. The only demonstrable issue is that there may be a remote possibility of a collision between two credit card numbers that would result in a false match when searching hashes.

Hash Salt

"Salt" is a value added to the original string to produce a unique hash, which may prevent the same card numbers from producing the same hash and may make brute forcing hashes more difficult. There are a number of limitations and issues related to using salt when hashing card numbers –

- Salt value can never change since the card number can never be re-hashed.
- Salt must be a value always available when searching for a card number, otherwise a match is not possible.
- If the salt value is from the transaction, then all card numbers must be searched by hashing the searched card number with the salt value of every transaction.
- Only transaction ID, transaction date, transaction amount, and/or approval code may be available for standard processes like chargebacks and retrieval requests, so hashing with any other value would require the searched card number to be hashed with the salt value of every prior transaction.
- Transactional data may not be available when the hash is created or may change after the hash is created, therefore, should not used as salt. Examples include a delayed authorization where the approval code is set at a later time or transaction amount may be updated to correct an error or include a discount.
- For data warehousing applications, a frequent business requirement is that the same card number must be matched across multiple transactions. Thus, the salt value must be fixed for the same card number, but should not be card holder name, address, or expiration date as these values may change over time.

Based on the above limitations and issues, this paper will assume no salt or a fixed salt is being added to the card number.

Brute Force Hash Attacks

Attack Methods

A number of brute force attack methods can be used to obtain valid card numbers from compromised hashes. This paper assumes the card number hash was generated using either MD5 or SHA-1 in a single pass with no salt or a fixed, known, and short salt for all card numbers – these assumptions are based on our findings during multiple application security assessments.

The optimal attack method will vary based on the card brand mix and other factors such as number of card numbers desired by the attacker and the anticipated end-use of the compromised card numbers. The card brand mix will vary based on the merchant business segment (retail vs. travel or b2c vs. b2b), sales channels (ecommerce vs. bricks and mortar), geographic location (United States vs. Europe), and the brands accepted by the merchant.

This paper will highlight three different attack methods and estimate the time to brute force card numbers –

Length Attack – As card numbers vary in length by brand, an attacker will first brute force shorter length card numbers. It is possible to compromise all 14 and 15 digit card number in less than thirteen days.

Brand Attack – The length of the brand ID varies by card brand, an attacker can attack longer brand IDs. Attacking based on bank prefix (up to the first six digits are the bank prefix for some brands) is the most optimal attack.

Known Digits Attack – The application may store from 4 to 10 digits in plain-text. If 10 digits are in plain-text, an attacker can compromise all card numbers in less than 2 hours.

For these methods to be used, an internal or external attacker must be able to compromise the application/database and obtain the necessary information including the following –

- Credit card hashes
- Card holder data including card holder name, address, and expiration date
- Credit card data stored in plain-text, which may include the last 4 digits, card brand information, or first 6 digits
- Understanding of the hash method including salt that may be added to the hash by obtaining application design documents, application code, or through testing of the application (e.g., purchase something and try different algorithms on the hash)

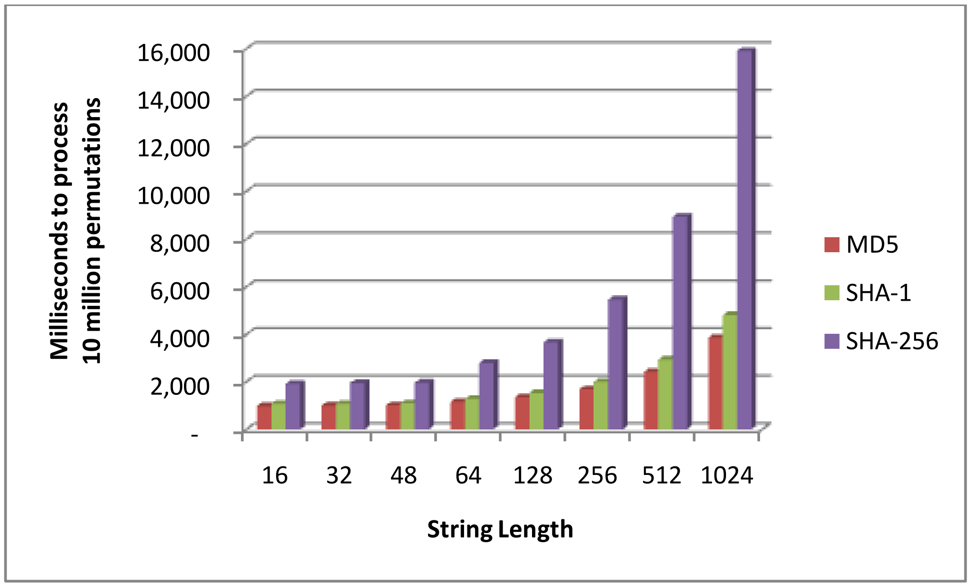
BRUTE FORCE BENCHMARK

In order to estimate and benchmark the time required to brute force hashes based on the MD5 and SHA-1 algorithms, a prototype program was developed in C using the OpenSSL 0.9.8d (www.openssl.org) assembly language MD5 and SHA-1 Windows dynamic link libraries and a custom Luhn algorithm function. The program executes sequentially in a single thread and was only minimally optimized with about a 30% performance improvement from initial tests to final benchmark. All benchmarks were executed on an Intel Core2 Duo T7200 2.0 GHz processor running the Microsoft Windows XP SP2 operating system.

All benchmarks were performed two ways: (1) all permutations with the check digit calculated for the last digit and (2) when the check digit is known (i.e., last 4 digits known), all permutations are generated and only permutations with valid check digits are hashed. When the check digit is known, only about 8-12% of permutations are valid and need to be hashed by the brute force program. Approximately 80% of the brute forcing processing time is related to generating the hash versus generating the permutation, calculating the Luhn check digit, and comparing the hashes.

Benchmark	Brute Force Benchmark
All permutations calculating last digit as check digit using SHA-1 algorithm 1. Generate Permutation 2. Calculate Luhn check digit 3. Generate SHA-1 hash 4. Compare to compromised hashes	2 million hashes per second
Only valid permutations when check digit is known (discarding about 90% of total permutations) using the SHA-1 algorithm 1. Generate Permutation 2. Calculate Luhn check digit 3. Compare to known check digit a. If not valid, discard permutation b. Otherwise, generate SHA-1 hash 4. Compare to compromised hashes	10 million hashes per second

- Only about a 5% difference exists between MD5 and SHA-1 – the small difference is most likely due to the fact that both of the OpenSSL algorithms are written in assembly language.
- The string length (card number + salt) only significantly impacted performance when string length was greater than 64 bytes due to the internal processing of the hash algorithm.
- A dramatic decrease in performance was observed for SHA-256 as compared to SHA-1. This is probably the result of the SHA-256 implementation in OpenSSL being written in C rather than assembly language.



With multi-threading, optimization, and a faster processor, we believe a brute force speed of over 50 million hashes per second is achievable for MD5 and SHA-1 with small string lengths. Using off-the-shelf specialized MD5 and SHA-1 cryptographic co-processors, it may be possible to achieve even faster brute force speeds.

Brute Force Estimation Tables

For each attack method, the possible number of permutations and estimated time to compromise card numbers has been calculated. This is not meant to be an exhaustive analysis of all possible methods, but illustrative of some basic approaches an attacker may use to compromise hashed card numbers. By intelligently approaching the problem, an attacker most likely can break 30-70% of all compromised hashes within a reasonable time period.

The estimation tables are for illustration only and may contain errors. The following assumptions were used to generate the estimates –

- Card Brands – Only the most popular credit and charge card brands for the United States and Europe are included. Debit and specialty cards are excluded.
- Brand IDs – The list of brand IDs was compiled from multiple sources. Most likely the brand IDs used do not accurately represent all current and active brand IDs.
- Random Card Numbers – Card numbers are assumed to be randomly generated and there exists no algorithm or logic to generate the digits between the brand ID, bank prefix, and check digit. This may not be true for some card brands and the distribution of card number most likely is not perfect.
- Brute Force Performance – The following benchmarks are used for time estimates –
    1. Check digit not known = 2 million hashes or permutations per second
    2. Check digit known = 10 million hashes or permutations per second

A. Length Attack

Card numbers are variable length based on the brand ID with the most popular card brands ranging from 14 to 16 digits. As with any brute forcing of hashes, encryption keys, or passwords, the shorter the value means the smaller number of possible permutations. **If only the hashed card number is available, it is actually practical to obtain all 14 and 15 digital card number hashes in less than thirteen days.**

If the Last-4 digits are stored in plaintext, each Last-4 combination (i.e., 1234) requires significantly less time as the Luhn check digit results in only 1 in 10 permutations having to be hashed. This increases the hash generation speed from 2M/sec to 10M/sec. About 0.01% of the all compromised hashes can be obtained in just under five hours.

Paradoxically due to the check digit, it is actually more efficient to perform a full length attack rather than attack all Last-4 combinations since when generating the Last-4 combination 9 of 10 permutations are discarded as invalid card numbers.

Length Attack			
Brand	**Len**	**Length Attack**	**Known Last 4**
Diners Club, Diners Carte Blanche	14	160,000,000,000	160,000,000
American Express, JCB (15)	15	2,020,000,000,000	2,020,000,000
Diners enRoute (no check digit)	15	200,000,000,000	20,000,000
Visa, MasterCard, Discover, JCB (16)	16	166,050,000,000,000	166,050,000,000
Estimated Time (h:mm:ss)		**All Hashes**	**per Last 4 Combination**
Diners Club, Diners Carte Blanche	14	22:13:20	0:00:16
American Express, JCB(15)	15	280:33:20	0:03:22
Diners enRoute (no check digit)	15	27:46:40	0:00:02
Visa, MasterCard, Discover, JCB (16)	16	23062:30:00	4:36:45

Note: (card number length)

B. Brand Attack

A more efficient method is to brute force the hashes based on brand and brand IDs. For shorter length card numbers and long brand IDs, this attack can obtain all the card numbers for a specific brand in a matter of hours.

Attacking specific common prefixes can potentially obtain a significant number of card numbers for large issuing banks in a matter of minutes. This information is not generally available, but unauthorized lists of prefixes and bank identifiers are available on some Internet sites for the most popular card brands.

If only the hashed card number is stored, an attacker can potentially obtain 30-70% of all card numbers within a matter of hours by intelligently focusing on the most popular card brands and issuing banks. The top ten commercial banks control over 90% of the credit card market as measured by total credit card debt [9], thus the number of common prefixes is a small subset of the possible 100,000 values. This attack can also be optimized by targeting regional issuers.

Brand Attack				
Brand	**Len**	**Brand Attack**	**Brand ID Attack**	**Common Prefix Attack**
Visa	16	100,000,000,000,000	100,000,000,000,000	10,000,000,000
MasterCard	16	50,000,000,000,000	10,000,000,000,000	1,000,000,000
American Express	15	2,000,000,000,000	1,000,000,000,000	100,000,000
Discover (b=65)	16	10,000,000,000,000	10,000,000,000,000	-
Discover (b=6011x)	16	50,000,000,000	10,000,000,000	-
JCB (16)	16	6,000,000,000,000	100,000,000,000	-
JCB (15)	15	20,000,000,000	10,000,000,000	-
Diners Club	14	100,000,000,000	100,000,000,000	-
Diners enRoute	15	200,000,000,000	100,000,000,000	-
Diners Carte Blanche	14	60,000,000,000	10,000,000,000	-
Estimated Time (h:mm:ss)		**All Hashes**		**per Prefix**
Visa	16	13888:53:20	13888:53:20	1:23:20
MasterCard	16	6944:26:40	1388:53:20	0:08:20
American Express	15	277:46:40	138:53:20	0:00:50
Discover (b=65)	16	1388:53:20	1388:53:20	-
Discover (b=6011x)	16	6:56:40	1:23:20	-
JCB (16)	16	833:20:00	13:53:20	-
JCB (15)	15	2:46:40	1:23:20	-
Diners Club	14	13:53:20	13:53:20	-
Diners enRoute	15	27:46:40	13:53:20	-
Diners Carte Blanche	14	8:20:00	1:23:20	-

Note: (card number length) and (b=brand ID)

C. Known Digits Attack

An application may store the brand, brand ID, or first six digits along with the last 4 digits. **If the Prefix 6 + Last 4 digits are known, all card numbers can be obtained in less than 2 hours.** The Brand ID + Last 4 attack is much less efficient than attacking using common prefixes, but still can obtain about 0.01% of card numbers quickly especially for some brands.

Known Digits Attack				
Brand	**Len**	**Known Brand + Last 4**	**Known Brand ID + Last 4**	**Known Prefix 6 + Last 4**
Visa	16	100,000,000,000	100,000,000,000	1,000,000
MasterCard	16	50,000,000,000	10,000,000,000	1,000,000
American Express	15	2,000,000,000	1,000,000,000	100,000
Discover (b=65)	16	10,000,000,000	10,000,000,000	1,000,000
Discover (b=6011x)	16	50,000,000	10,000,000	1,000,000
JCB (16)	16	6,000,000,000	100,000,000	1,000,000
JCB (15)	15	20,000,000	10,000,000	100,000
Diners Club	14	100,000,000	100,000,000	10,000
Diners enRoute	15	20,000,000	10,000,000	100,000
Diners Carte Blanche	14	60,000,000	10,000,000	10,000
Estimated Time (h:mm:ss)		**per Known Digits Combination (average 0.01%)**		
Visa	16	2:46:40	2:46:40	0:00:00
MasterCard	16	1:23:20	0:16:40	0:00:00
American Express	15	0:03:20	0:01:40	0:00:00
Discover (b=65)	16	0:16:40	0:16:40	0:00:00
Discover (b=6011x)	16	0:00:05	0:00:01	0:00:00
JCB (16)	16	0:10:00	0:00:10	0:00:00
JCB (16)	15	0:00:02	0:00:01	0:00:00
Diners Club	14	0:00:10	0:00:10	0:00:00
Diners enRoute	15	0:00:02	0:00:01	0:00:00
Diners Carte Blanche	14	0:00:06	0:00:01	0:00:00

Note: (card number length) and (b=brand ID)

Rainbow Tables

Rainbow tables are pre-generated hashes for large sets of permutations. A rainbow table requires significant processing power, time, and storage to initially generate, but can match a credit card number in milliseconds. The disadvantages of a rainbow table are that it is fixed on a specific hash algorithm, format of the credit card number, and salt. Generally, rainbow tables are most useful with hashed passwords in widely deployed applications as the hashing algorithm and salt do not change from system to system and the rainbow table can be reused repeatedly. In specific situations, use of a rainbow table could dramatically reduce the time to compromise subsets of credit card numbers, especially when card numbers are obtained over a period of time or from multiple systems using the same hashing methods. This paper will not look at rainbow tables as one-time brute forcing is usually as time efficient in most situations and does not require the massive storage of a rainbow table.

Conclusion

Storing of credit card numbers by simply hashing only the card number is unacceptable and can be easily compromised by brute force methods. An attacker who is able to compromise the application or database can obtain many card numbers in a trivial amount of time –

- If only the hashed card number is available, it is actually practical to obtain all 14 and 15 digital card number hashes in less than thirteen days.
- If only the hashed card number is stored, an attacker can potentially obtain 30-70% of all card numbers within a matter of hours by intelligently focusing on the most popular card brands and issuing banks.
- If the Prefix 6 + Last 4 digits are known, all card numbers can be obtained in less than 2 hours.

Hashing of credit card numbers is an acceptable solution if designed to protect against brute force methods, while still satisfying the basic business requirements. Ecommerce and payment applications should use the strongest available cryptographic hash algorithm (like SHA-512), always use large salt values, and perform multiple hash iterations (at least 100 iterations) in order to reduce brute force efficiency.

The use of salt in the hashing process is most problematic due to business requirements. A significant advantage to using salt in a custom application is that an attacker has to determine or obtain application code where card number is hashed. This is particularly effective in situation when the attacker can only exploit a query-only SQL injection vulnerability. In the case of an internal attacker, open source applications, or widely-deployed commercial applications, there is probably little to no advantage.

Optimally, the salt should be transaction specific (either a transaction value or a random number), thus requiring an attacker to calculate all permutations for each transaction. Although, transaction specific salt does not work with tracking card numbers across transactions nor allows for efficient searching and matching of card numbers as the search for card number must be hashed with every transaction value. Using a static salt or one derived from the card number is of limited value as the attacker still only has to make one pass through all the possible permutations. A secret key can be effectively used as a salt value, albeit this defeats some of the reasons for hashing in the first place and requires the same key management as encryption. Another option is to use an especially large salt value, like 4KB, that is computationally more expensive and will reduce brute force efficiency as the internal hash algorithm processing is based on blocks (SHA-1 uses 512-bit blocks). The advantage of this technique is that it has no affect the stored hash value size. In our benchmark, increasing the string size from 16 bytes to 1,024 bytes (15 more internal 512-bit blocks) increased brute force time by about 400%.

References

1. Roger Nebel, "Hashing for fun and profit: Demystifying encryption for PCI DSS", 6 December 2006, http://searchsecurity.techtarget.com/tip/0,289483,sid14_gci1230572,00.html
2. Kurt Seifried, "Storing credit card numbers securely", 10 September 2001, http://www.seifried.org/security/cryptography/20011009-storing-cc.html
3. John Deters, " Taylor's Law: John Deters's Rebuttal", 21 August 2002, http://www.miketaylor.org.uk/tech/law2.html
4. Branden Williams, "Eliminating Card Numbers to Minimize PCI Exposure", 2006, http://www.verisign.com/static/036133.pdf
5. PCI Security Standards Council, "Payment Card Industry Data Security Standard 1.1", September 2006, https://www.pcisecuritystandards.org/tech/download_the_pci_dss.htm
6. Visa U.S.A., "Cardholder Information Security Information Program (CISP) Payment Application Best Practices 1.3", 8 May 2006, http://usa.visa.com/download/business/accepting_visa/ops_risk_management/cisp_payment_application_best_practices.doc
7. Wikipedia, "Credit card number", 2 February 2007, http://en.wikipedia.org/wiki/Credit_card_numbers
8. PCI Security Standards Council, "Payment Card Industry Data Security Standard Security Audit Procedures 1.1", September 2006, https://www.pcisecuritystandards.org/pdfs/pci_audit_procedures_v1-1.pdf
9. Federal Reserve Bank of Philadelphia, "WORKING PAPER NO. 05-29", page 10, 5 December 2005, http://www.phil.frb.org/files/wps/2005/wp05-29.pdf

History

February 27, 2007 – Initial Version

About Integrigy

Integrigy Corporation is a leader in application security for large enterprise, mission critical applications. Our application vulnerability assessment tool, AppSentry, assists companies in securing their largest and most important applications. AppDefend is an intrusion prevention system for Oracle Applications and blocks common types of attacks against application servers. Integrigy Consulting offers security assessment services for leading ERP and CRM applications.

Integrigy Corporation
P.O. Box 81545
Chicago, Illinois 60602 USA
888/542-4802
www.integrigy.com

Author: Stephen Kost

If you have any questions, comments or suggestions regarding this document, please send them via e-mail to alerts@integrigy.com.

Integrigy's Vulnerability Disclosure Policy – Integrigy adheres to a strict disclosure policy for security vulnerabilities in order to protect our clients. We do not release detailed information regarding individual vulnerabilities and only provide information regarding vulnerabilities that is publicly available or readily discernable. We do not publish or distribute any type of exploit code. We provide verification or testing instructions for specific vulnerabilities only if the instructions do not disclose the exact vulnerability or if the information is publicly available.