



***A White Paper
Credit Card Information
Surrogate
“A Method and System for
using surrogates to integrate
PCI-level security for legacy
information systems”***

by

***Tom Arnold, CISSP, ISSMP, CFS,
QSA, QPASP***

Script contributions by

Paul Guthrie

March 1, 2007

***Derivative of work published in
May 2001¹***

¹ This work is derived as an extension to the original work prepared by Tom Arnold and referenced in footnote 2.

Abstract:

This paper presents a method and system for assignment of a consumer credit card surrogate, based on card account number issuing methods. The method offers a solution to the drawback presented by using a SHA1 message digest, specifically for credit card numbers. The proposed solution appears to have all the positive characteristics noted for a digest, and the surrogate I describe fits into a 16-digit numeric credit card field in legacy file systems or databases. I am pleased to introduce the Card Account Surrogate or Token. A national space account number² that has the random aspects of a digest, uniquely identifies a customer account, has no value to an outside individual, and passes the Luhn³ check used to validate card account numbers.

Contents

ABSTRACT:	1
CONTENTS	1
INTRODUCTION.....	1
THE PROBLEM	3
EXAMPLE OF CARD NUMBER SHA1 ENCODED IN HEX AND BASE64.....	4
USE OF SURROGATE (ALSO CALLED A TOKEN)	5
METHOD AND SYSTEM FOR USING SURROGATE CREDIT CARD NUMBERS.....	6
EMBODIMENT OF SURROGATE SYSTEM	9
SAMPLE TOKEN GENERATION SCRIPT	13
SAMPLE DE-MASKING CARD SCRIPT	15

Introduction

This paper presents a method and system for assignment of a consumer credit card surrogate, based on card account number issuing methods. Many years ago, I published a paper describing an Electronic Citadel for security cardholder information⁴ ("E-Citadel"). In the E-Citadel paper, I described the concept of an isolated vault database that could be used to store private consumer information, like a credit card. The E-Citadel addressed the use of encryption and key management techniques to protect this sensitive information. I specifically described a mechanism for producing a hash of the card information and using the hash as a unique reference to the clear card number, using a SHA1⁵. This process generates a unique hash or digest of a consumer credit card number that can be as large as 40 characters (if represented in hex). Needless to say, the use of a hash to uniquely identify card numbers stored in a secure data vault or E-Citadel provided an easy to generate value that could be computed to test the presence of a valid card number. The SHA1 has several characteristics that are very good, like:

² ISO 7812 International Standard governing magnetic stripe identification cards

³ Patent 2,950,048, Computer for Verifying Numbers, Hans P. Luhn, August 23, 1960.

⁴ Arnold, Tom "An Electronic Citadel: A method for securing credit card data and private consumer data in E-business sites" A White Paper from the Technology Working Group, eBusiness Division, Software and Information Industry Association, May 2001

⁵ FIPS 180-1, "Secure Hash Standard" Federal Information Processing Standards Publication 180-1, <http://www.itl.nist.gov/fipspubs/fip180-1.htm>, 1995

1. The value is easily computed;
2. The resulting hash appears random and is easily indexed by relational database (it's a good key);
3. Searches of databases using a SHA1 are very fast;
4. And, the SHA1 is a method approved for use according to the PCI Data Security Standard⁶ ("PCI DSS") §3.4.

That said, there are a few serious drawbacks to using a digest:

1. The size of the resulting hash is larger than fields in many legacy database systems. Some companies attempt to resolve this by using a database table identity value as a key. Unfortunately, this system-generated value may require that legacy systems undergo some level of conversion or change prior to accepting it.
2. Message digests of sensitive data are frequently not salted. As such, the resulting hash may facilitate recovery of the original value should the attacker obtain the hash value along with other facts permitted by the PCI DSS, like: the credit card type, bank identification number, and the final four digits.

This paper presents a solution to the drawback presented by using a SHA1 message digest, specifically for credit card numbers. The proposed solution appears to have all the positive characteristics noted for a digest, and the surrogate I describe fits into a 16-digit numeric credit card field in legacy file systems or databases. I am pleased to introduce the Card Account Surrogate or Token. A national space account number⁷ that has the random aspects of a digest, uniquely identifies a customer account, has no value to an outside individual, and passes the Luhn⁸ check used to validate card account numbers.

About the Author

Tom Arnold is a partner with PSC (www.paysw.com), a firm specializing in payment systems, security, and compliance for companies that accept or process consumer payments. He has experience as a VP of Engineering and Chief Software Architect for over 20 years and developed products for a variety of software companies. He is a CISSP, ISSMP and has a broad background engineering secure software applications. Over his career, he's led large engineering teams and understands the principles of engineering quality products. Tom's experience includes development of payment processing applications; high-performance credit card processing switches; and several secure data vaults for protecting sensitive information.

In May 25, 1999, Mr. Arnold gave expert testimony to the US House of Representatives, Committee on Commerce, Subcommittee on Telecommunications, Trade and Consumer Protection to assist in their deliberations on the proposed SAFE Act, related to the use of encryption technology to secure and protect Internet eBusiness trading. Then, on June 10, 1999, Mr. Arnold testified before the US Senate, Committee on Banking on the security and technology impact of the proposed Export Administration Act of 1999. Since that time, he has been consulted by numerous regulatory agencies including the Department of Commerce, Department of Treasury, Department of Justice, World Trade Organization, European Union Tax Ministers, Organization for Economic Cooperation and Development, and the US Fair Trade Commission on topics of Internet commerce, digital rights management, identity theft, fraud, consumer protection and consumer privacy.

⁶ "Payment Card Industry Data Security Standard" Payment Card Industry Security Standards Council. Version 1.1. September 2006.

⁷ ISO 7812 International Standard governing magnetic stripe identification cards

⁸ Patent 2,950,048, Computer for Verifying Numbers, Hans P. Luhn, August 23, 1960.

Corrections, Comments and Complaints

Please send communications about the contents of this paper to the author. Tom Arnold can be reached by email at tom@paysw.com

Why Release This Paper?

My intent in releasing this paper is for several reasons:

- To make software publishers and merchants aware of a mechanism that can be used to facilitate security and compliance with the Payment Card Industry Data Security Standard ("PCI DSS").
- For protection of credit card data, to help educate and expose some of the issues with card account surrogate (vendors sometimes refer to a tokenization) systems that are offered from payment gateway and software vendors.
- To describe a method and system that can be adapted so that merchants have a low-cost tool to segment card processing networks and systems with minimal impact on legacy systems (like ERP, finance applications, distribution, or manufacturing systems). There may be some merchants who came up with this exact process prior to the release of this paper and the Electronic Citadel in 2001. For someone who intimately knows credit card processing, this solution is actually quite obvious. My intent is to broaden the audience and to explain the concept to more people.
- To update the original concept presented in the Electronic Citadel paper, referenced earlier.
- To describing a couple of test embodiments using business uses cases. Word of my work leaked to one of the companies that provide encryption products. They expressed concern that I was talking about this and expressed that they might file a patent on the method. By releasing the paper, I've reminded them that the basis for my concept was rooted in both the PCI DSS and the Electronic Citadel, originally published in 2001. They weren't happy.
- I have a strong belief that this concept, the method and the system belong in the public domain, as this is a natural extension to the Electronic Citadel and PCI DSS.

The Problem

As we examine the securing of sensitive data elements that have latent processing value like credit card account numbers, UK National Health Service, or US social security numbers, the challenge comes when the attributes must be referenced for future processing.

As a review, private consumer identification information can be stolen and used to commit numerous types of fraud and crime. There is little argument about the value of this information when obtained by an attacker. Reading the popular press, one can easily understand the significance of this data as the basis of any identity theft.

This is the basis for the electronic citadel; the ubiquitous value of sensitive consumer account data is also the basis of the PCI DSS, consumer privacy laws and the focus of government cyber regulation and many emerging security standards.

The challenge to securing this information comes when an organization decides to invest in the development of a cryptographic system where the elements have a latent processing value. Here the term "latent processing" means that the sensitive element needs to be referenced by a application at some future time, recovered to its original state and finally submitted as a part of a future process. The most obvious of these can be seen in common electronic commerce sites where the same consumer returns to the site and makes a subsequent purchase. Another example is a rate-based billing system where a registered consumer opts to charge the balance due on their account to a credit card.

Using a credit card account number as an example, I describe the use of a SHA1 message digest to create a unique reference to the original account number in the original Electronic Citadel paper. The example below illustrates the use of a SHA1 as a reference to the original card number. These are illustrated as either a hex or base64-encoded representation of the account number.

Example of card number SHA1 encoded in hex and base64	
Digest length	40
Hex Digest	01af08779d5c205d5b5b1f8c462665af101a176d
Digest length	28
base64 Digest	Aa8ld51cIF1bWx+MRiZlrxAaF20=

The challenge for most implementers is the size of the resulting value. Note that the base64-encoded value is represented as a 28-character string. Although this is a suitable approach for referencing the original card number, the modification to legacy systems to accommodate the larger field size can result in significant cost increases. Databases may have to be modified and application code may have to change to accommodate the use of this kind of surrogate value.

Another potential problem with the use of a message digest as a reference appears if the computational value of the digest was improperly formed. Members of the computer security community clearly understand that a message digest needs to be salted with some other value to protect from a brute-force computation attack. A salt is a type of cryptographic nonce.⁹ This is a random value that is concatenated to the clear value, and then hashed to produce the message digest. Sadly the creation and storage of a salt in a payment system introduces significant complexity for the transaction processing system. In essence, the salt must be known to the systems that may need to test a credit card number at some future time. (Consider the refund scenario. You only want to give a refund to the same card used to make the original purchase). At the same time, the salt value should not be generally known. The secure storage and lookup of a salt can be as painful as the protection of an encryption key. Companies that have access to hardware security modules (HSM) used to protect cryptographic key material may use these devices to store a key that is used for the salt. This solution secures the resulting hash from a computational attack, but limits the ability to calculate the digest to systems that are authorized to connect to the HSM.

To further illustrate the computational problem, consider the following PCI DSS requirements:

PCI DSS §3.2: Mask PAN¹⁰ when displayed (the first six and last four digits are the maximum number of digits to be displayed).

Note: This requirement does not apply to employees and other parties with a specific need to see the full PAN; nor does the requirement supersede stricter requirements in place for displays of cardholder data (for example, for point of sale [POS] receipts).

⁹ Salt, as defined in Wikipedia. See: [http://en.wikipedia.org/wiki/Salt_\(cryptography\)](http://en.wikipedia.org/wiki/Salt_(cryptography))

¹⁰ PAN, as used in PCI DSS, stands for the Primary Account Number. This is the credit card number.

PCI DSS §3.4:

Render PAN, 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 MINIMUM account information that must be rendered unreadable is the PAN.

When implementing PCI 3.2, many companies will store the first 6 digits of the card number along with the final 4 digits of the account number. (They actually misinterpret the requirement and confuse masking with truncation – that would be the subject of yet another whitepaper). This is actually an inappropriate thing to do, since the middle 6 digits are the only portions of the account number that are truncated from the original. This requirement actually states that the account numbers are to be masked “when displayed.” In an of itself, this mistake leaves 100,000 permutations of possible cards that can be used to find the original and an attacker has little help in finding the exact card.

PCI DSS 3.4 allows for the use of SHA1¹¹ to conceal the card account number. The actual problem arises when a company stores the first 4 or 6 digits and an un-salted SHA1 of the whole account number.

Digest matched. The credit card number is: 4111111111111111	
real	0m41.422s
user	0m39.867s
sys	0m0.143s

The result above is from a small *perl* script that located the missing digits to identify the account number. As noted in the table, my old Mac Powerbook G4 with 1gb RAM and not much else running, took a little over 41 seconds to locate the actual account number. This was originally performed using a real credit card number in the FindCard script¹², the output above has been masked to a pseudo number.

This provides a significant target for an attacker when the company that constructed the solution has modified their databases and legacy systems to accommodate the use of a SHA1 as a surrogate of the credit card account number. Unfortunately, the result is a system that is no more secure than if they were storing the full card number. The only potential benefit that this company would see is that they could consider themselves exempt from various US State laws requiring notice to consumers because they encrypt the card numbers, hardly justification.

Use of Surrogate (also called a token)

The use of surrogate values to reference real data in database systems has long been a standard in the information industry. The most obvious use is enumerated primary keys in a relational database. An example being that a given row in a database contains important information and has a system-generated value as the primary key. The key value is used to uniquely identify a row and in addition, all attributes in the row are referenced as functionally dependent on the key value.

Surrogate database keys are commonly used in many information systems, but they are not actually the surrogates that I'm referencing in this paper. Using a credit card as an example, the surrogate will reference a specific account number, not a set of data. Here are some examples where surrogates can be used:

¹¹ In PCI DSS §3.4, the term “strong one-way hash” is defined as a minimum use of SHA1.

¹² Findcard.pl script is included at the end of this paper

- Credit card account number: A single surrogate can reference the single instance of a specific consumer credit card number. This would be similar to how many businesses use the calculated SHA1, except the SHA1 is an actual message digest or digital signature of the account number. The SHA1 can be used to calculate whether or not a card number is the same that was used in a prior transaction. This is not a function of the surrogates that I'm presenting and was covered in detail in the Electronic Citadel paper.
- US Social Security Number: A single surrogate can reference the single instance of a social security number and be used to perform secondary processing on this number.
- UK National Health Service number: This is a value similar to the US social security number that is frequently stored in health-provider databases.
- Passport number: A surrogate number can be assigned to reference the passport number in company databases. In many countries, passport numbers are collected by hotels, airlines, and by employers as proof of citizenship.

This is not an exhaustive list of conditions where surrogates can be used. There are many possibilities, depending on the industry and use. The prime characteristic when identifying if a surrogate should be used for a given value is whether or not the original value will need to be used for processing after the original session – where the value was submitted – was concluded.

For the purpose of this paper, the focus will be on the credit card numbers. This is a value that is highly recognized, subject to abuse and well suited for the use of surrogates.

Method and System for Using Surrogate Credit Card Numbers

In the section that follows, I will describe the embodiment of credit card surrogates in two business cases. These cases should provide clarity on the functional applicability of implementing a surrogate solution.

In this system, a business application is used to collect the credit card number. This might be a front-desk system at a hotel, a call center application used to take an order, or the checkout web page on an electronic commerce site. The application being used is actually irrelevant.

For illustrative purposes, let's assume we have a system that processes a credit card tender in two steps: authorization and settlement.

The card number is accepted for purchase of products and services and sent to a payment application. The payment application reformats the authorization request and submits to a credit card processor. This is actually a much more complex process, but I've simplified it for our purposes.

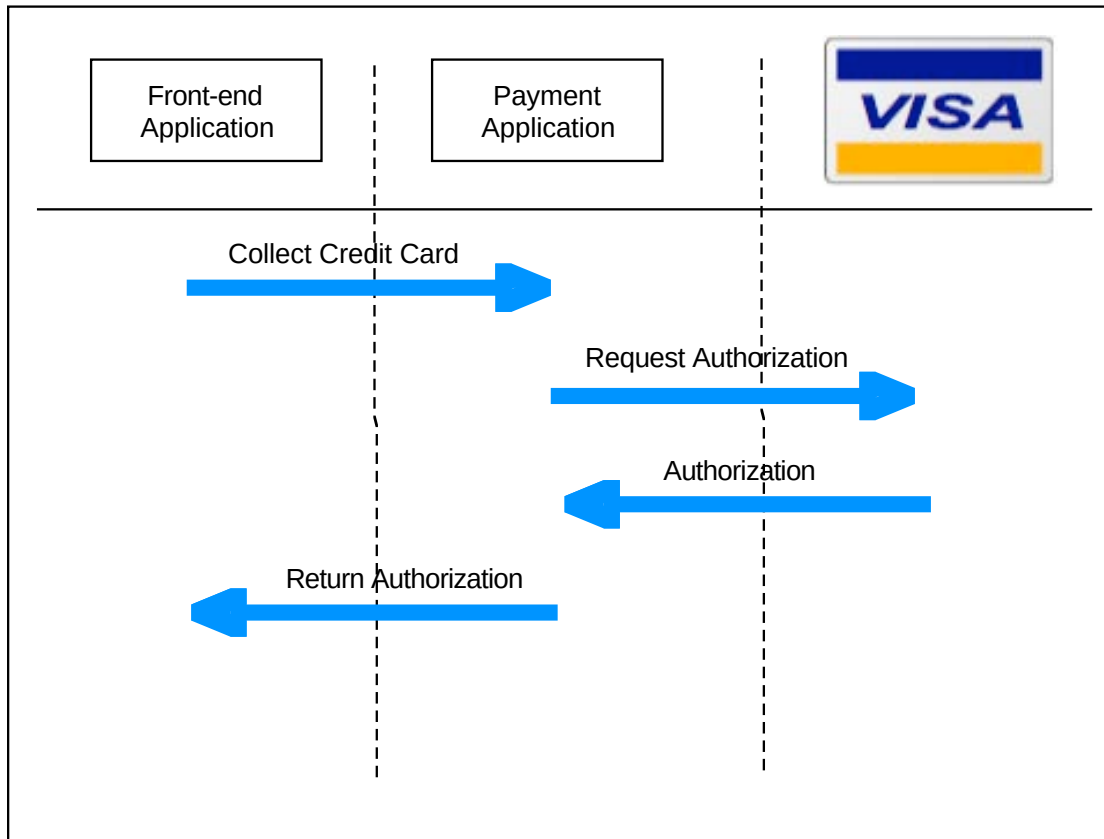


Figure 1: Simplified Payment Process

The shipping and distribution system then manages the selection, packing and shipping of the goods to the consumer. This process may actually take place several days after the original purchase transaction was placed. In this case, the warehouse application needs to notify the payment application that it's time to collect funds from the original authorization.

The payment application needs access to the original, clear credit card number along with some other attributes to prepare a batch for submission to the payment processor. Then, at a pre-ordained time, the batch is submitted, processed, and funds collected. Again in actual practice this is a much more complex process.

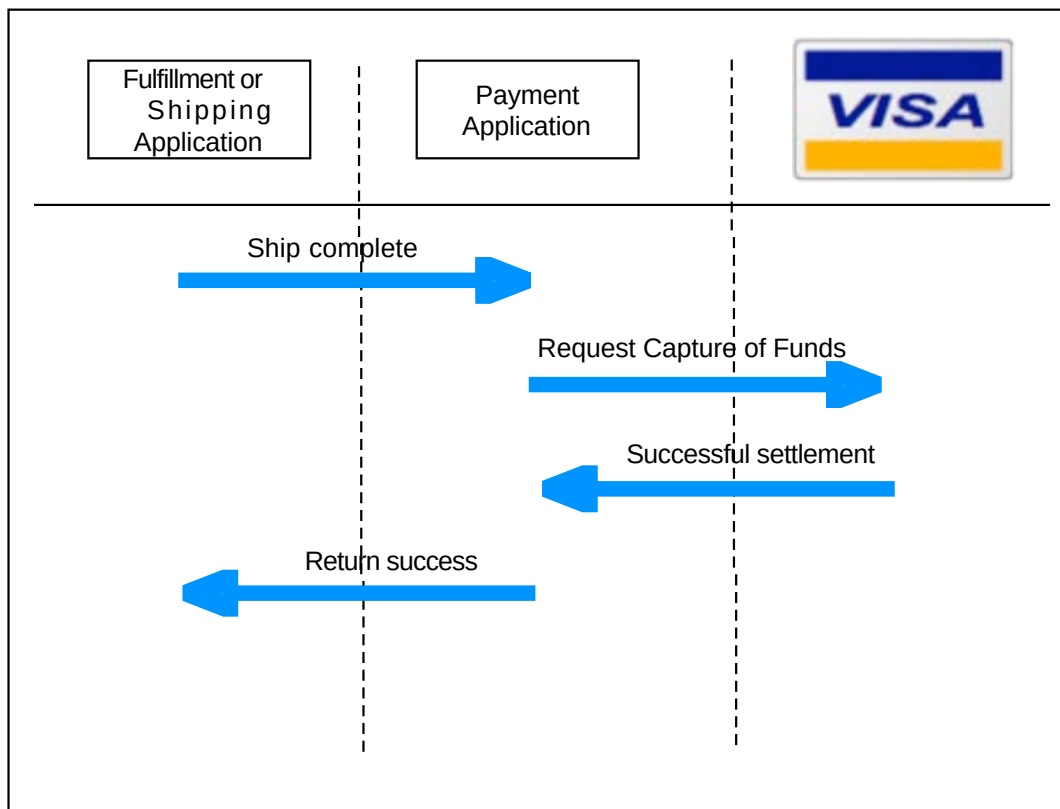


Figure 2 Submission and Settlement Process

There is an important point in this: either the payment application or the fulfillment and shipping application needs to have access to a database where the actual account numbers are stored and referenced by the order. In most cases, both of these applications will have a database associated with them.

In most companies, the applications listed are legacy systems or include software applications that were purchased from an independent software company. In the latter case, the company implementing the application may not have the source code or the right to modify the application. In any event, a surrogate value can be stored in the databases of these applications, and if that surrogate fits the ISO 7812 standard for an account number, then the applications may require less modification to adapt.

Method

The method for generation of a surrogate is to create a 16 digit number that passes the Luhn check, but uses the national account number space for the value. The national account number space is frequently used to generate account numbers for private-label or closed-loop cards, like gift cards, merchant stored-value cards, and other private charge cards.

System

As an example, when the consumer's credit card number is submitted to the payment application for the first authorization, a software sub-routine in the payment application generates a surrogate value. I've included a Perl script as an example of this. The surrogate value is a 16 digit number that passes the Luhn check and is consistent with the ISO 7812 standard for account numbers.

```
$ genToken.pl 1 P
```

```
9794843943821565
```

The 16-digit surrogate is held in the payment application database along with the encrypted original account number. This payment application may generate a message digest of the original account number for use as a signature, but this is outside the method for surrogates. After receiving the authorization response, the surrogate account number is appended to the authorization response that is sent to the front-end application.

Upon receiving this authorization response, the front-end application continues the order acceptance and processing lifecycle as implemented. As part of this lifecycle, the surrogate account number may be passed to other derivative systems, like the fulfillment and warehouse system.

An alternate process for generation of these tokens may involve further implementation of the ISO 7812 standard and include the registration of a formal Issuer Identifier Number (IIN). In this manner, the IIN or first 6-digits of the numbers generated would be unique to the company that registers. One obvious use of registering for a formal IIN would be to make the tokens identify a specific class of cards. A case where this would be useful is when a merchant has a co-branded issued MasterCard and offers a discount on its use. This IIN with the token could be used to easily identify the cards where discounts are entitled. The IIN may also be used to generate surrogates for large global companies where the first three digits after the nine (9) would be the country code. This could be used to enhance reporting and accounting by identifying the region where the original sale was collected.

Embodiment of Surrogate System

In this section, I describe the business implementation of the credit card surrogate system using two business cases. In all cases the system is implemented when an Electronic Citadel or data vault is used to protect the actual credit card number. This data vault is a separate application that handles the storage, encryption, decryption, and references to the clear credit card account number. The vault is a small, but separate application that is referenced by the payment application.

For each case, I will describe a brief background of the applications use and functionality. I'll also describe a basic deployment of the data vault and application.

Wholesale Distribution and Sales System

This is a large database and sales system implemented in a clothing manufacturer for selling products into a worldwide retail distribution channel. Many of the clothing products are also available for sale in co-branded Web sites to end consumers. Also, many of the smaller retailers are unable to qualify for a trade credit account with the manufacturer, so the system accepts orders from these small retailers using credit cards.

Environment

The core system [1] is a homegrown application that operates in an IBM AIX system environment. The application was coded in a mixture of Java and C. A large relational database [2] supports the core application processing and contains dynamic product information, pricing, as well as the artifacts of business transactions. The payment application [3] acts as a switch and handles credit card tenders from Visa, Mastercard, American Express, and Discover. This application communicates to a variety of payment processors and manages multiple merchant accounts, depending on whether the sale is wholesale or a direct to consumer retail transaction. A constellation of Web servers and Web applications [4] and [5] provide application interfaces to the core system.

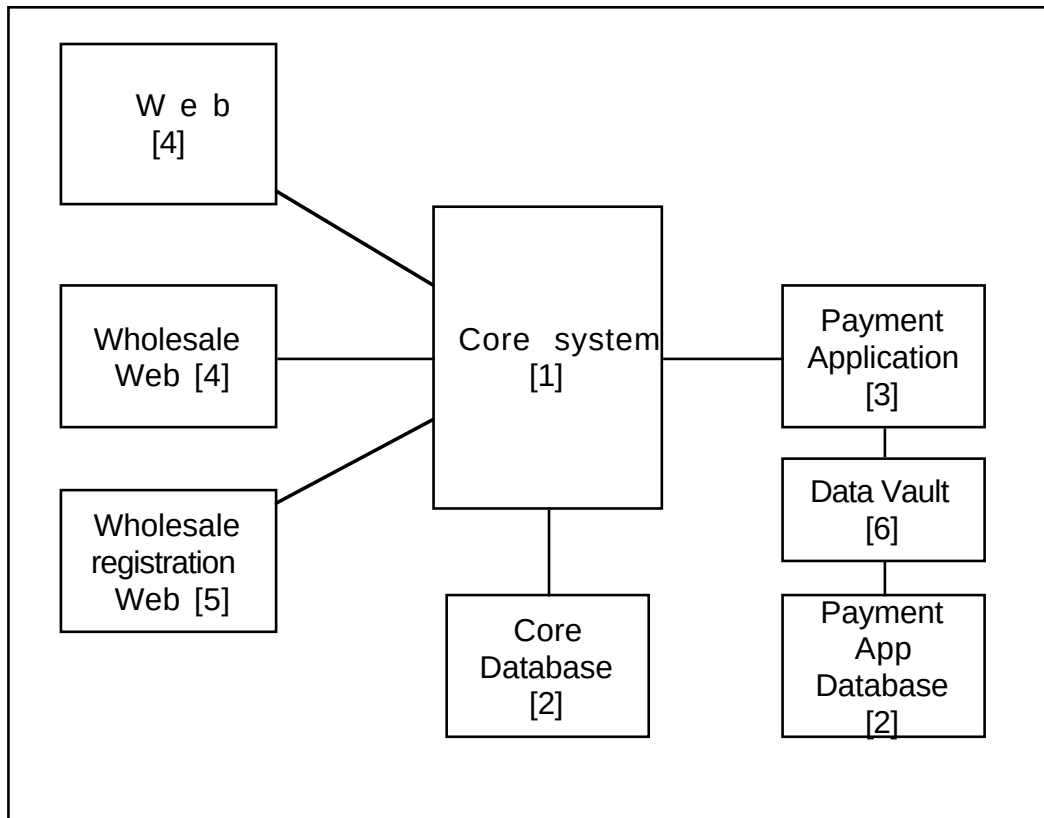


Figure 3: The Core System

The Web servers [4] support a variety of custom and co-branded Web sites that are accessed by consumers to purchase clothing. These servers operate shopping cart and order acceptance applications. The sites also support customer registration and collection of customer profiles that include credit card numbers. This facilitates purchases by return customers.

Wholesale Web servers [4] support orders placed either through company-run call centers or directly by small retailers. The site supports the selection of products and the collection of credit card data to accept the order.

Wholesale registration [5] is a Web-based application that is used to create new wholesale accounts in the core system. When an account is created for a small retailer, credit card information is collected and stored. Additionally, some medium-size retailers like to pay their monthly invoices with a credit card so as to collect cardholder loyalty points.

A Data Vault [6] has been implemented in the system to provide secure storage and cryptographic services to the payment application. This component includes the use of a HSM to securely store encryption keys and to produce salted SHA1 values as signatures of card numbers.

Surrogate Implementation

The Data Vault [6] implements the generation and storage of surrogates at the time an initial authorization is run on a credit card. The surrogate is stored in the vault with the encrypted and salted-SHA1 of the card account number. If a card transaction comes into the payment application [3] where the card account number begins in a 9, payment application [3] knows to

ask the Data Vault to exchange the surrogate for the actual account number. The payment application [3] then proceeds to process the transaction as though any normal payment request.

All other components of the system that store credit card account numbers will record the surrogate to their databases, instead of the actual account number. This was a small change to the Core System [1] to substitute the actual card number with the surrogate when an authorization response was received.

The Core System records the surrogate for any type of authorization response. In essence, if the credit card number is declined, the surrogate would be extracted from the response message and stored in the database.

The Wholesale registration site [5] needed to be modified slightly to accommodate the surrogate. In the original implementation, the credit card number was never authorized upon initial registration. This system [5] only performed a Luhn check to verify that the account number was correctly entered and then stored the number directly into the database. To accommodate the surrogate, this system [5] needed to request the Core system [1] to perform a pre-application (or pre-authorization). The Core System [1] would then request an authorization from the Payment Application [3] in the amount of \$1. The response would include the surrogate, which was stored with the wholesale account registration in database [2].

Rate Based Billing System

This is a very large system supporting the rate-based billing of mobile telephone user accounts at a very large cell-phone carrier. The system supports a wide variety of sales and distribution channels including those that are either outsourced to other companies or support independent resellers.

Environment

The core system [1] is an application licensed from SAP that operates in a large IBM environment. Oracle relational databases [2] support the core application processing as well as the storage of data from the payment application. The payment application [3] was a homegrown application that integrates several switching functions. This application and its database support the full lifecycle of credit card account tenders as well as tenders based on other forms of closed-loop payment systems. Credit card tenders include all major global cards from Visa, Mastercard, American Express, JCB, Discover, Switch, Solo, Giro, US EFT, US ACH, and others. This application communicates to a set of payment switch applications that will direct the authorization and settlement requests to the appropriate processor, depending on payment brand.

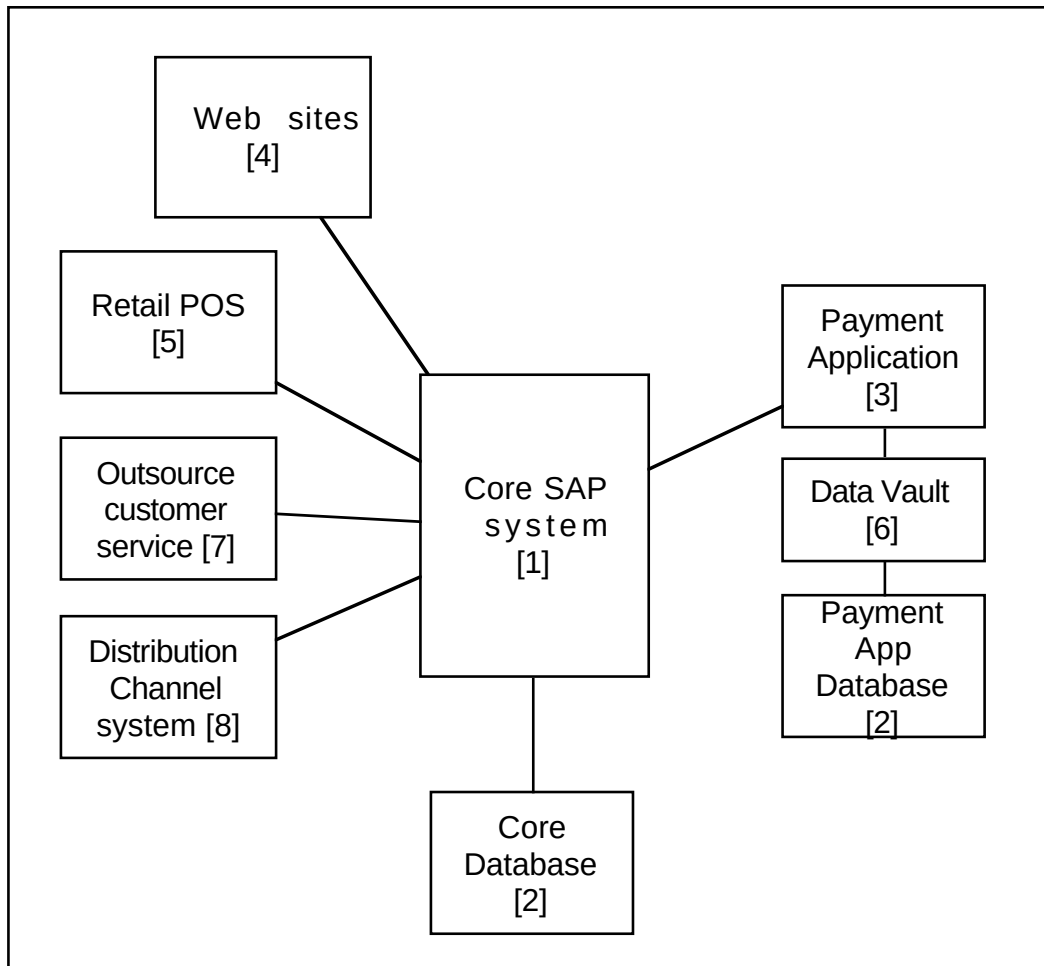


Figure 5: Legacy Core System

The Core Database [2] and Payment Application Database [2] are Oracle database schemas implemented in a very large database cluster.

The Web sites [4] support a variety of custom web applications where customers can view their mobile phone account; pay bills, select billing options, register payment types, buy phone products, and upgrade service plans. These servers operate shopping cart and order acceptance applications.

Retail POS [4] represents a network of over 1400 retail sites. Orders are placed through these systems when customers come into stores to purchase cell phones. The information collected at these sites includes credit card numbers and tender information, should the customer decide to sign up to pay usage bills by credit card.

Outsource customer service [7] is an application accessed by various call center companies to manage customer accounts. Functions include purchase of equipment, change of plans, setup of recurring billing and collections.

Distribution Channel systems [8] include a diverse set of applications that are Web-service-based supporting account setup, plans selection, recurring billing setup and bill payment. This set of applications support third-party payment kiosks as well.

A homegrown Data Vault [6] has been implemented in the system to provide secure storage and cryptographic services to the payment application. This component includes the use of a HSM to securely store encryption keys and to produce salted SHA1 values as signatures of card numbers.

Surrogate Implementation

The Data Vault [6] implements the generation and storage of surrogates at the time an initial authorization is run on a credit card. The surrogate is stored in the vault with the encrypted and salted-SHA1 of the card account number. If a card transaction comes into the payment application [3] where the card account number begins in a 9, payment application [3] knows to ask the Data Vault to exchange the surrogate for the actual account number. The payment application [3] then proceeds to process the transaction as though any normal payment request.

All other components of the system that store credit card account numbers will record the surrogate to their databases, instead of the actual account number. This was a small change to the Core System [1] to substitute the actual card number with the surrogate when an authorization response was received.

The Core System [1] and Payment Application [3] record the surrogate for any type of authorization response. In essence, if the credit card number is declined, the surrogate would be extracted from the response message and stored in the database.

Sample Token Generation Script

I've included the following script as an example of how to derive a card token. Though it should be noted that this script is only included as an example. The script was built using Perl version 5.8.8 included with the Mac OSX Leopard 10.5. There is one major function missing and that is a check for duplicates. In it's current form, this script uses the Perl `rand()` function to generate random token values. If you generate enough tokens, there will be duplicates so be attentive.

Also, these tokens do not use the concept of a BIN and are restricted only to having the first digit be 9. At the discretion of the implementer, the concept of a private BIN could be implemented. In this situation, each token would start with the same 4 or 6 digit value.

Results generating 7 tokens:

```
Usage: perl genToken.pl #-of-cards (P | R)
      /t #-of-cards -- number of card values to generate
      /t P (default) -- Use private card range
      /t R -- Generate regular cards distributed by percent
              market share
```

So, running `genToken.pl` to generate a set of 7 tokens, the results look like this:

```
$ genToken.pl 7 P
9794843943821565
9201001850634664
9078430134342168
9058690168232908
9082278313715785
9019751978887113
9414425288494024
```

Here is the script:

```
#!/usr/bin/perl
#
# Program to generate a credit card token
# The resulting value will look like a credit card but will start with
# a 9
#
# Based on Paul Guthrie ccgen.perl. Paul Guthrie contributed
# significantly to this script.
#
# Modified by Tom Arnold
# March 2007
#
use strict;
use switch;

# Global Variables
my %map = map { $_ => $_ } 0..9;

# Subroutine to perform check digit operation
sub check_digit {
    my @buf = reverse split //, shift;

    my $totalVal = 0;
    my $flip = 1;
    foreach my $r (@buf) {
        unless (exists $map{$r}) {
            my $ERROR = "Invalid character, '$r', in check_digit
calculation";
            return;
        }
        my $posVal = $map{$r};

        $posVal *= 2 unless $flip = !$flip;

        while ($posVal) {
            $totalVal += $posVal % 10;
            $posVal = int($posVal / 10);
        }
    }
    return (10 - $totalVal % 10) % 10;
}

# Begin main process
#
# Test for arguments
if ($ARGV[0] == "") {
    print "Usage: perl genToken.pl #-of-cards (P | R)
/t #-of-cards -- number of card values to generate
/t P (default) -- Use private card range
/t R -- Generate regular cards distributed by percent market
share\n";
    exit(1);
}
```

```

# Global variable declaration and assignment of first card numeral
my $cardcount=$ARGV[0];
my $cardtype="P"; # Set type of token to private card only -- first
digit will start with 9

# If second argument given, assign argument to $cardtype.
# If this argument is not R or B, this program will run as default
if ($ARGV[1]) {$cardtype = $ARGV[1];}

# Change this next line to change the distribution of cards.
# Sorry - only by 5% at a time.
# This string should always be 20 long and represent the distribution
of
# cards generated
# currently at:
#     AMEX: 15%
#     DISCOVER: 5%
#     MASTERCARD: 30%
#     VISA: 50%
#     (63335555554444444444)
#
# Original from card generator by PG my $cardstring =
"63335555554444444444";
#

# Set cardstring for all first card values to 9
my $cardstring = "99999999999999999999";

# Test $cardtype and set to Random card if value of R | r
if (($cardtype eq 'R') || ($cardtype eq 'r')) {
    $cardstring = "63335555554444444444";
}

# Generate card numbers
while ($cardcount-- > 0) {
    my $firstdigit = substr($cardstring,int(rand()*20),1);
    my $ccno = $firstdigit;
    for (my $i=0;$i< ($firstdigit == "3"?13:14);$i++) {
        $ccno = $ccno . int(rand()*10);
    }
    print $ccno . check_digit($ccno) . "\n";
}

# end

```

Sample De-Masking Card Script

I've included this script as an example of how to derive card numbers when the "masked" value is stored along with a hash. This script uses a hash representation that is in hex or base64. This is only an example and has been supplied to assist with security testing. This script is really quite simple and takes no math skills at all to figure out. You just have to understand how to setup basic permutations of a number and it's done. These permutations are taught in US 8th grade math classes.

I use this script to test data when doing PCI DSS assessments where the merchant or service provider truncates card numbers by removing the middle 6 digits and stores the PAN.

The script is written in Perl. For UNIX-based systems, we suggest calling this with the UNIX "time" command.

```
#!/usr/bin/perl
#
# Script to generate a credit card number set from a BIN and final
# 4 digits.
# The script will accept a hash of the clear card number as either an
# md5 or sha1. Digest can either be base64 or hex encoded.
#
# By Tom Arnold
# 12/5/07
# Modified: 12/8/07 to include b64 encoding.
#
use strict;
use Digest::SHA1;
use Digest::MD5;

# Global Variables
my %map = map { $_ => $_ } 0..9;

# Subroutine to perform check digit operation
sub check_digit {
    my @buf = reverse split //, shift;

    my $totalVal = 0;
    my $flip = 1;
    foreach my $r (@buf) {
        unless (exists $map{$r}) {
            my $ERROR = "Invalid character, '$r', in check_digit
calculation";
            return;
        }
        my $posVal = $map{$r};

        $posVal *= 2 unless $flip = !$flip;

        while ($posVal) {
            $totalVal += $posVal % 10;
            $posVal = int($posVal / 10);
        }
    }
    return (10 - $totalVal % 10) % 10;
}

# Begin main process
#
# Test for arguments
if ($#ARGV < 1) {
    print "Usage: time findCard.pl final-four Bin-number [digest]
/t final-four -- the final four digits of a card number
/t BIN-number -- first 1, 4 or 6 digits of the card number
/t [digest] -- this is either the hex or base64 digest (md5 or
sha1) of the credit card number \n";
    exit(1);
}
```

```

# Test the BIN to see how many digits were given
my $BIN_length = length($ARGV[1]);
my $startNumbers = $ARGV[1]; # Assign BIN to start of number
my $hashType; # hash type of either md5 or sha1
my $hashEncoding = "hex"; # set default to hex encoding for hash
my $digest; # hash or digest of credit card
my $finalFour = $ARGV[0]; # final four digits
my $finalThree = substr($finalFour,0,3);
my $numMissingDigits = 12; # Set for 16 digit visa with final four
digits known total 12
my $firstDigit = substr($startNumbers,0,1);

# Adjust number of missing digits based on card type; subtract one more
for American Express
if ($firstDigit == 3) { $numMissingDigits -= 1; }

# see if a digest was presented. If so, test and figure out what type.
if ($ARGV[2]) {
    $digest = $ARGV[2];
    my $l = length($digest);
    if (($l == 32)||($l == 24)) {
        # Must be an md5
        $hashType = "MD5";
        if ($l = 24) { $hashEncoding = "b64"; }
    } elsif (($l = 40)||($l == 28)) {
        # Must be a sha1
        $hashType = "SHA1";
        if ($l = 28) { $hashEncoding = "b64"; }
    } else {
        # Error
        print "Error: Invalid digest. Cannot determine by size if this is
either SHA1 or MD5\n";
        exit(1);
    }
}

# Test and assign variables and error out if issue

# Test BIN
if (($BIN_length == 1) || ($BIN_length == 4) || ($BIN_length == 6)) {
    $numMissingDigits -= $BIN_length;
} else {
    # Error. BIN not 1, 4 or 6 digits
    print "Error: BIN-number must be either 1, 4 or 6 digits in
length\n";
    exit(1);
}

# Get on with it
my $lastDigit = substr($finalFour,2,1);

my $factor = 10 ** $numMissingDigits;
my $counter = $factor;

```

```

# Let's do it; see if there is a hash present. If so, run until hash
matches
if ($hashType) {
    # Hash is present, do the work and calc a hash, compare the hash
    and find the card
    # Generate card numbers; test hash
    for (my $f=0;$f<$factor;$f++) {
        my $ccno = $startNumbers; # Assign start numbers
        #
        # Setup the missing middle digits using the $counter (which
        increments with each pass), given that $factor
        # is on digit larger than the number of missing digits. This will
        pad the number with 0 while counting
        # This could be slow.
        $ccno = $ccno . substr($counter++,1,$numMissingDigits);
        $ccno = $ccno . $finalThree; # put the final three digits on
        the number
        $ccno = $ccno . check_digit($ccno); # print the number with the
        check digit appended
        my $hash;
        if ($hashType eq "MD5") {
            my $md5=Digest::MD5->new();
            $md5->add($ccno);
            if ($hashEncoding eq "b64") {
                $hash = $md5->b64digest;
                $hash = $hash . "==" ;
            } else {
                $hash = $md5->hexdigest;
            }
        } else {
            # must be a sha1
            my $sha1=Digest::SHA1->new();
            $sha1->add($ccno);
            if ($hashEncoding eq "b64") {
                $hash = $sha1->b64digest;
                $hash = $hash . "=" ;
            } else {
                $hash = $sha1->hexdigest;
            }
        }
        if ($hash eq $digest) {
            # we have a match
            print "Digest matched. The credit card number is: $ccno
\n";
            exit(0); # done. Let's stop
        }
    }
    # This will only run if there is no match with the cards and hash
    print "Sorry. No match found. Digest must be wrong or other
parameter is invalid\n";
    exit(0); # done. Let's stop
} else {
    # Hash not present, count all of the possible valid card numbers
    where the last digit matches the checksum
    # Generate card numbers. Just do a account of the card numbers
    found. There will be a lot.

```

```

    my $validCards; # This will be the count of all valid card numbers
found
    print "Calculating set of valid card numbers. Please be patient...
\n";
    for (my $f=0;$f<$factor;$f++) {
        my $ccno = $startNumbers; # Assign start numbers
        #
        # Setup the missing middle digits using the $counter (which
increments with each pass), given that $factor
        # is on digit larger than the number of missing digits. This will
pad the number with 0 while counting
        # This could be slow.
        $ccno = $ccno . substr($counter++,1,$numMissingDigits);
        $ccno = $ccno . $finalThree; # put the final three digits on
the number
        my $checkno = check_digit($ccno);
        if ($lastDigit == $checkno) {
            # we have a match
            $validCards++;
        }
    }
    print "\tFinished. Found:  $validCards valid card numbers\n";
}
# end

```

Acknowledgements

The author would like to thank the following individuals for their assistance and contributions to this paper:

Ashok Misra for taking the time to review and provide technical comments on the payment processing and surrogate number issuance.

Brad Dispensa who provided invaluable technical editing to the document. He cleaned up a bunch and questioned everything. This assistance was greatly appreciated.

Tony Bates, Nigel Tranter, Paul Guthrie, Ralph Engwall, and Rafael Rosado who contributed materially and with a great amount of technical support on the topic.