PuKE-PoW

A Combined Public Key Encryption and Proof-of-Work System for Obfuscated Storage and Transmission of Credit Card Information

Aaron Hammond

I. Need

The Payment Card Industry Data Security Standard (PCI DSS) governs appropriate methods of storage and transmission of cardholder data. The strict guidelines thereof often pose a challenge in the prospective development of new technologies in the electronic commerce space. A particular challenge emerges when considering how best to enable individuals to accept and transfer information required to process payments, including credit card number, credit card verification number, expiration date, and cardholder identity. A secure and easily implementable method is therefore needed for future progress in e-commerce, especially on often-insecure mobile devices.

II. Solution

The following is a secure method for the storage and subsequent transmission of cardholder data, generalized for the entry of credit card number for the sake of simplicity. Extensions of the scheme described below to other elements of cardholder data would, however, be trivially enumerated.

Sally Sue is a sales-agent of the Breton River Cosmetics Company (henceforth referred to as BRCC). As a business, BRCC has a relationship with the Lowell Card Company (LCC), a payments processor. BRCC relies on LCC to process all payments made through its subsidiary sales-agents, and Seller will therefore transmit sales information directly to LCC. The proposed solution would thus be implemented by LCC.

At a function designed to drum up sales shamelessly disguised as a party, Sally convinces her friend Betty Bluth to purchase a bourgeois skin product (which, for the reader's interest, contains powder from post-processed diamonds). Betty is now prepared to enter her credit card information on a mobile device running an application provided by LCC.

When Sally signed up for BRCC, using her automobile as collateral, she was provided by LCC with an identification number and a unique, *n* digit application programming interface (API) key. A hashed version of this API key is associated with Sally's identification number in a database table within LCC's rather large quagmire of a processing platform. For the sake of example, Sally's identification number (ID) is 101, her unhashed API key is 11APPLES11, and her hashed API key is BEEFCAKE. To reiterate, Sally has knowledge only of her plaintext API key and her ID. After the registration process, Sally downloaded an application provided by LCC for her mobile device.

Prior to her "party," Sally, ever the dilligent sales-lady, prepared for her sales of the day, grinning and rubbing her hands together in a Mr. Burns-like fashion. In a simple user interface, Sally enters her ID and unhashed API key

(11APPLES11) into her mobile application. Her ID (101) is stored in plaintext in the application, while her unhashed API key is hashed using the same algorithm used by LCC and then stored (now BEEFCAKE). These two identifiers will remain stored in the application for the duration of her "party."

Returning to Betty (the purchaser who, after this explanation, has become rather impatient), Sally begins the transaction procedure. Her mobile application sends a request containing her ID to the LCC server, indicating that it is ready to accept a credit card number. The server responds with a public key to encrypt the transaction and a unique transaction number (henceforth referred to as TXN). In the meantime, the mapping between the TXN, ID, and corresponding private key is stored (with a time-out) on the server. After indication that the application has received the requisite information (perhaps a blinking light; people love blinking lights), Sally passes her mobile device to Betty, who is now presented with a number pad in which she can enter her credit card number, digit-by-digit, away from the prying eyes of Sally.

For the sake of nomenclature, Betty's credit card number is composed of sixteen digits, named b_0, b_1, b_2, ..., b_{15}. Betty enters the first digit of her credit card number, b_0, which is encrypted in its byte representation using the public key provided earlier by the request to LCC. When Betty enters the second digit of her credit card number, b_1, its byte representation is appended to the result of the encryption of the previous digit. The resultant string is then again encrypted using the public key provided by LCC.This process is repeated for each of the remaining 14 digits. For those keeping score at home, the resultant byte string will therefore be:

$$result = \text{PK}(\text{PK}(\text{PK}(\text{PK}(\text{PK}(\text{PK}(\text{PK}(\text{PK}(\text{PK}(\text{PK}(\text{PK}(\text{PK}(\text{PK}(\text{PK}(\text{PK}(b_0) + b_1) + b_2) + b_3) + b_4) + b_5) + b_6) + b_7) + b_8) + b_9) + b_{10}) + b_{11}) + b_{12}) + b_{13}) + b_{14}) + b_{15})$$

where PK is the public key encryption function and + indicates an byte-string append. Condensing to a recursive formula (because recursion makes recursion easier):

$$\text{result}(0) = \text{PK}(b_0)$$
$$\text{result}(n) = \text{PK}(\text{result}(n - 1) + b_n)$$

Terrific! We now have a publicly encrypted byte string of the credit card number. The more astute reader might say, however, "Hey moron! Anyone can request a public key if she has the insecure ID! This system isn't secure, you shmuck!" Well, astute reader, the system does not conclude here (you shmuck). Recall the hashed API key (BEEFCAKE), now stored on Sally's mobile device. Prior to the iteration of the public key encryption on each digit of the credit card and the resultant string, the digit can be used as a salt for successive hashes. That is:

$$hash = \text{H}(\text{H}(\text{H}(\text{H}(\text{H}(\text{H}(\text{H}(\text{H}(\text{H}(\text{H}(\text{H}(\text{H}(\text{H}(\text{BEEFCAKE} + b_0) + b_1) + b_2) + b_3) + b_4) + b_5) + b_6) + b_7) + b_8) + b_9) + b_{10}) + b_{11}) + b_{12}) + b_{13}) + b_{14}) + b_{15})$$

where H is the same hash function used by LCC and + indicates an byte-string append. Condensing to a recursive formula (because recursion makes recursion easier):

$$\text{hash}(0) = \text{hash}(\text{BEEFCAKE} + b_0)$$
$$\text{hash}(n) = \text{hash}(\text{hash}(n - 1) + b_n)$$

Now that we have a thoroughly hashed API key (henceforth referred to as KEY) and a thoroughly encrypted credit card number (henceforth referred to as CCN), a request can be created with both data, along with the original TXN issued by the server. This request can be delivered immediately or later in a batch with all other requests for the "party" over HTTP. Either way, the complete request is received by the LCC server, where processing begins.

The first step of the processing is retrieval of the private key and hashed API key (11APPLES11) for the corresponding ID which is obtained by the TXN provided with the request. At this juncture, the original credit card number provided by Betty can be obtained by iterated decryption of the CCN using the stored private key. That is, the private key is used to decrypt the raw CCN, and the final byte of the resultant byte string is the original b_{15}. Taking the remaining byte string and again applying the private key decryption, the final byte of this resultant string is the original b_{14}. This process is repeated to construct the original b_0, b_1, b_2, ..., b_{15}.

Now that the LCC server has the salts (b_n) used in construction of the KEY, the server can use a basic proof-of-work algorithm to construct the expected KEY in much the same manner as KEY was computed prior to the

request. Comparing the expected KEY with the KEY provided verifies the correctness of the initial provided hashed API key. LCC now has the complete credit card number against which it should transact, as well as the guarantee that the correct API key was provided. LCC can now charge Betty and transfer the appropos funds to BRCC and Sally, thus completing its obligations.

III. Benefits and Limitations

The principal benefit offered by this scheme is the minimal storage of data. At any given point in the card number entry stage, only one single digit exists in an unencrypted state. Barring interception of entry, potential thieves would gain access only to a thoroughly encrypted byte string representation of a credit card number. By enforcing time sensitivity of the TXN identifier and corresponding public key, LCC could further mitigate potential insecurity as a result of theft. Even given the initial hash, the proof of work system used would require access to the entirety of the unencrypted card number to generate the correct KEY. At the transmission stage, decryption of requests would therefore be prohibitively computationally expensive without access to the private key or initial hash.

On the server side, the scheme as detailed above remains secure. The issuance of private keys on a per-transaction basis limits incredibly the potential for malicious decryption of incoming requests. Assuming request data and corresponding private key/TXN mappings are purged at their expiration, would-be attackers could therefore decrypt only the incoming transactions for the period in which they are able to access the database.

The quite-obvious trade-off for this level of security is computational cost. Each encryption/decryption and hash operation is of nontrivial complexity. Partial implementation of this scheme (i.e. use every-other card number digit to salt the encryption of CCN and hashing of KEY) would thus be more feasible for organizations willing to trade slightly-less secure storage and transmission for less-intense computation. That said, the possibility of decryption varies little between 16 rounds of hash and encryption and 8 rounds of hash and encryption.