Final Individual Project Report

Kurt Schneider / iSafe / Ariel Amberden

In today’s global market, electronic transactions are becoming an increasingly used method of conducting business. In a recent study, the Diary of Consumer Payment Choice found that 27 percent of all transactions are electronic and roughly 11 percent of these transactions are done on mobile devices[[1]](#endnote-1). A few of the mobile applications that were created to meet this increasing demand include PayPal and ApplePay. Consumers should be confident that any transaction done through a mobile phone is as secure as a credit card or debit card transaction done in traditional ways. However, According to J. Gold Associates Research Report, "organizations lost an average of $92.3 million due to fraudulent mobile transactions for a 3% loss of total revenue per year."[[2]](#endnote-2) With this in mind, there is clear need of improvement in mobile security.

1. INTRODUCTION

E-commerce continues to play an increasing role in the global economy as more business transactions are done electronically. As this trend grows, more of these transactions will be made through mobile devices. Customers would like assurance that any transaction made through a mobile phone is as secure as any other electronic transaction. Many mobile applications exist to meet this demand; the two most prominent being PayPal and ApplePay. Any such application must ensure data confidentiality and data integrity. To accomplish this, we created a mobile online payment system that securely performs payment processing on behalf of customers and merchants. Any user of the application can play the role of the customer or merchant. The main assumptions and security concerns include:

1. Outside passive (eavesdropping) and active adversaries with the capabilities of the most current high-end computers. Active attacks include denial of service attacks and SQL injection attacks.
2. Inside passive adversaries accessing data they are not supposed to and inside active adversaries changing data they are not supposed to.
3. All adversaries’ knowledge will be limited to cipher text only and known plain text.
4. A successful attack for an eavesdropper requires that user data or transaction data be retrieved.
5. A successful denial of service attack requires that transactions are halted.
6. A successful SQL injection attack requires that information be retrieved from the database or changes made to the database without detection.
7. In order for insider attacks to be considered successful, they must not be identified by the system. In other words, if the system can determine which unauthorized changes were made, correct those changes and determine the adversary, then the attack was not successful.

The solutions to these problems closely follow ApplePay and the EMV payment tokenization system. User information will be encrypted at the mobile app level using an Apple style encryption before any information is sent to the server. The current Apple standard is a combination of 1280-bit RSA public encryption and128 bit AES, signed with ECDSA under a 256-bit NIST curve[[3]](#endnote-3). Client and server connections will use the OpenSSL TSL protocol. To prevent unauthorized access, users will be required to create strong login credentials. To protect these credentials, the app will require users to change passwords at regular time intervals without annual repetition and controlled password resets via email notification. After three incorrect login attempts, the user will be notified that their account has been disabled for a given amount of time. Users will only be required to add the minimum amount of information required to complete a transaction (i.e. credit card information). Emails will be sent to the users with records of purchases and a timeline of recent logins. Users can send notification of potential fraud through the link sent with the email.

1. LITERATURE RESEARCH

The workings of ApplePay are largely governed by the EMV Token Specification Framework. The EMV specification defines five different parties for any transaction: 1) Cardholder, 2) Merchant, 3) Token Service Provider 4) Token Requestor, and 5) Issuer. In the iSafe app, the user will play the role of the Cardholder or Merchant depending on the context of the transaction. We also have the Merchant act as the Token Requester and the Issuer act as the Token Service Provider.

Token Service Providers are entities that are authorized to provide payment tokens to Token Requestors. These entities are responsible for a few important functions. They must operate and maintain a Token Vault, which is the repository that holds the one-to-one mappings of credit card numbers to payment tokens. A payment token is a randomly generated surrogate value for a valid credit card number that is identical to a valid number, and acts as a reference to the actual credit card number[[4]](#endnote-4). When generating a payment token, they must insure that the token fits the same format as a credit card number and does not overlap with a currently existing valid credit card number. Token Requesters’ are required to register with the Token Provider. After registration, a Token Requester will be provided with a Token Requester ID. From this point on, the Token Requester can request tokens from the Token Requester[[5]](#endnote-5). Each token request also includes an Identification and Verification (ID&V) assurance method which is based on the agreed assurance level between the Token Requester and the Token Provider. This enables an entity to successfully validate the Cardholder identity and account information in order to establish a level of confidence in the credit card to payment token mapping. Examples of ID&V methods include 1) account verification method, 2) risk score method, and 3) one time password by the Issuer to verify the Cardholder[[6]](#endnote-6).

One of the main security aspects when using this specification is that credit card numbers are never stored on the iPhone, Apple servers or on any merchants’ server. When a user signs up to Apple Pay, encrypted credit card information is sent to the appropriate credit card network where the credit card number is validated. If the number is valid, a payment token is sent back and stored on the mobile device[[7]](#endnote-7). An adversary would be unable to distinguish a payment token from a credit card number. A payment token has no intrinsic value and is utterly useless by itself. A token issuer generates a random string and maps it to a valid credit card number. This token can be made to expire after one transaction or made for only one merchant. It is also possible to create one universal payment token per credit card number, but this practice is less secure. Since the payment token and the card number have no fundamental relationship, only the token issuer (normally the credit card issuing bank) is able to determine the card number from the payment token.

When a transaction takes place, the iPhone sends the payment token to the merchant. The merchant then sends the payment token to the credit card network, which maps it to the corresponding credit card number. The credit card number then contacts the issuing bank to approve the transaction. The issuing bank will then notify the merchant if the transaction was approved and the transaction can proceed[[8]](#endnote-8).

This process prevents all kind of security breaches because the merchants never come in contact with the credit card number. For example, it protects against man in the middle attacks and credit card skimming because an adversary will only be able to obtain a useless payment token. While we were unable to determine the life time of a payment token for ApplePay, we chose to provide each merchant with a unique payment token. Thus the same card number will map to a different payment token.

3.1 PROBLEM STATEMENT

The problem was to create a secure mobile online payment system given a few assumptions and security concerns. The adversary model we are working with is limited to cipher text only and known plain text with the computing power of the most current high-end desktop computers. Some of the kind of attacks we wish to prevent are SQL injection attacks, denial of service attacks, and man-in-the-middle attacks. The biggest concern is keeping credit card information secure since credit card fraud remains customers’ biggest concern. The last concern was the ability to isolate iPhone application processes in order to prevent third-party application attacks.

3.2 CHALLENGES

During this process, we encountered four main challenges. The first challenge was the implementation environment. We wanted to make a mobile application and we could have gone with a more easily accessible environment like Android, but chose to develop the application on the iPhone using the relatively new Swift programming language. The problem this presented was that I did not have access to a Mac and thus was extremely limited in my ability to develop an iPhone application. To resolve this problem, the work was split so that I would handle the implementation of the server and Ariel would handle the implementation of the client.

The second challenge was that neither of us had developed an iPhone application before or programmed with Swift or Xcode, which are the two main tools for iPhone development. We were able to find an online Swift programming language book to help guide us.

The third challenge was understanding the complex payment transaction process, such as how each entity in the process interacts and what each entity is responsible for. Without a clear understanding of this process, simulating the process would be impossible.

The last challenge was that I had no experience in server side coding or where to begin. Much time was spent using Google, reading articles and watching tutorials. Eventually I decided on creating databases on MySQL server along with Microsoft .NET Framework.

3.3 SOLUTIONS

The solution structure was broken up into three main class structures: 1) iSafe User (iPhone), 2) Acquirer, 3) Issuer and Token Service Provider. The iPhone application would act as the client, where the user could play the role of the customer or merchant. The basic functions of the application are to create new user, request payment (merchant role), approve amount (customer role), request token (merchant role), and confirm payment (customer role – after receiving confirmation from the Acquirer). The user provided customer data includes name, address, credit card number, credit card type, credit card cvv and the assigned data includes the ID&V assurance level and a unique user ID. The merchant user provided data includes name, address, checking account number, checking account routing number, and checking account institution. The assigned data includes the ID&V assurance level and a unique merchant ID.

The server side was broken up into the Acquirer module and Issuer module. The Acquirer module simulates the bank or credit union. To perform the simulation, the database is pre-loaded with dummy accounts for testing purposes. The data includes all of the data collected from the iSafe application along with transaction information (merchant and customer id’s, amount, date, and payment token data). The simulated functions are authorization, capturing, clearing, exception processing, and request of transaction approval. Authorization occurs when the customer’s issuing bank approves the transaction based on available credit. Capturing is the submission of a credit card transaction for processing and settlement. Clearing is where the buyer and the seller of a transaction are matched, thus confirming that both parties are in agreement with the terms of the transaction. Settlement is when the Issuer and the Acquirer exchange financial information resulting from the transaction[[9]](#endnote-9).

The Issuer module simulates the customers issuing bank and the Token Service Provider. The Issuer database is pre-loaded with dummy cardholders and pre-loaded with a payment token vault mapping credit card numbers to payment tokens. The main functions include CRUD operations (create, retrieve, update, and delete), registration of a new token requester, and the verification of the user and payment token, as well as issuing the payment token to the Token Requestor.

The iSafe application protects credit card information through the EMV Token Specification Framework and will never store credit card information. However, security breaches are still possible if the application does not safeguard user account information. To safeguard this information, salted password hashing will be used. The salt will be created using a Cryptographically Secure Psuedo-Random Number Generator (CSPRING) and the hashing done with a cryptographic hash function (SHA256). Note that a new random salt must be created for each user or when the user changes passwords. The username, salted hash of the password, and the salt (in plain text) are stored on a MySQL database. The iSafe application will also keep a history of orders and products.



Figure 1: Login UML Class Diagram

When the user creates an account with username and password, the salt is generated and prepended to the password. The password and salt is then hashed and stored on the database (never in plaintext). When the user attempts to login to their account, the hash of the password and the salt are retrieved from the database. The salt is prepended to the password the user has put in and hashed again. If it matches the hash of the stored password (the actual password) then the user is granted access[[10]](#endnote-10). It is also important to hash the MySQL server. For example:

INSERT INTO Login (username, salt, password) VALUES

(‘name’, ‘salt’, SHA2(‘password’, 256))

Tokenization and salted password hashing are not a substitute for secure client-server connections, thus it is still important to create secure connections using Secure Socket Layer (SSL). Client and server connections will use the OpenSSL TSL protocol. It is important that the most recent TLS version 1.2 is used and that previous versions (SSLv2 and SSLv3) are disabled. While this may restrict certain clients, it offers a higher security guarantee.

There are four general strategies to prevent against SQL injection attacks. The first strategy is data sanitization where data is filtered for context. For example, a credit card number will only accept a value that matches a valid credit card format. The second strategy is to place the server behind a firewall. A firewall can protect against “at least 50% of all types of exploits in any software.”[[11]](#endnote-11) The third strategy is to limit database privileges. Only the necessary privileges should be granted. The final strategy is to use prepared statements or stored procedures instead of constructing full queries with user input. This can prevent users from querying the database in unintended ways. For example, a user could query the database in Figure 1 with SELECT \* FROM Login WHERE username = '' OR 1''. This creates a logical disjunction where either the username = “” OR TRUE. Since this disjunction is always true, the query will return all the entries in the Login table.

The last concern is iPhone application processes isolation. Most of this is handled by Apple’s operating system (iOS) as explained in the iOS Security Guide[[12]](#endnote-12). iOS provides layers of protection to ensure that apps are signed, verified, and sandboxed (a sandbox is a security mechanism for separating programs) to protect user data, thus offering protection from malware and other unwanted third party attacks. The iOS kernel controls which user processes and apps can be run. iOS requires that all executable code be signed using an Apple-issued certificate. This certificate ensues that all apps come from a known and approved source. Third-party apps must also be validated and signed using an Apple-issued certificate, thus preventing third-party apps from loading unsigned code resources or using self-modifying code.

To protect the system and other apps from loading third-party code inside of their address space, the system will perform a code signature validation of all the dynamic libraries that a process links against at launch time. This verification is accomplished through a 10-character alphanumeric string (i.e. 1A2B3C4D5F) called the team identifier, which is extracted from an Apple-issued certificate. A program may link against any platform library that ships with the system or any library with the same team identifier in its code signature as the main executable. Since the executables shipping as part of the system do not have a team identifier, they can only link against libraries that ship with the system itself.

iOS does not allow users to install potentially malicious unsigned apps from websites, or run untrusted code. At runtime, code signature checks of all executable memory pages are made as they are loaded to ensure that an app has not been modified since it was installed or last updated.

Even after an application is verified and approved, iOS enforces security measures designed to prevent it from compromising other apps or the rest of the system by “sandboxing” all third-party apps. This ensures that third-party apps are restricted from accessing files stored by other apps or from making changes to the device, thus preventing one app from obtaining or changing information stored by another app. Each app has a randomly assigned and unique home directory for its files given to it when the app is installed. A third-party app can only obtain outside information by using services explicitly provided by iOS.

Since Apple makes it difficult for hackers to infect iOS apps with malware, this has generally not been a problem. In September 2015, hackers were able to get malware past Apple’s app store review by modifying some code in the XCode toolset and passing it off as Apple’s official version of XCode[[13]](#endnote-13). In order to prevent this type of attack, developers should make sure that they download XCode directly from Apple.

3.4 INDIVIDUAL WORK

Since we were developing an iPhone application and I did not have access to a Mac, the work was split so that I would handle the implementation of the server (Acquirer module and Issuer module) and Ariel would handle the implementation of the client. Much of the details are explained in the sections below.

3.5 COMPARISON OF WORK

The work load largely followed our project proposal and progress report. Initially, we both researched current industry standards for securing server transactions. Not really knowing where to begin, I read about the Payment Card Industry (PCI) Data Security Standard[[14]](#endnote-14). This turned out to be too general and too detailed for our purposes, but it did list some requirements that were applicable to our needs. Next we researched isolating iPhone app processes and securing transactions. I read the iOS Security Guide and concluded that this was pretty much handled by the operating system and out of our hands. After this, I began researching databases and servers and Ariel researched the Swift programming language and Xcode. We communicated very well during the research and design part of the problem, but seemed to lose track during the implementation process.

3.6 SHORTCOMINGS

The main shortcoming was having no experience developing anything of this nature, thus most of the time was spent trying to understand things outside of the security aspect of the project. After we thought we had all our research completed, Ariel discovered the EMV Payment Tokenization System. Originally, the plan was to have one server holding all transaction information without performing the Acquirer and Issuer roles. While this was definitely an improvement in design, it delayed our implementation progress.

At first, I just wanted to get the server up and running, so I created a payment token vault in the database without generating the payment tokens; I simply gave each credit card number a surrogate token value and each token an expiration date. This way we could get the client and servers communicating and take care of the transaction process. Ideally, I would have liked to create a CSPRING for each credit card number using the RNGCryptoServiceProvider class within Microsoft’s .NET Framework. This way, anytime a new transaction was added to the Issuer, a new random number would be generated and used as the payment token for that user and for that merchant the transaction was with. The length of the token would be slightly different for each credit card type. American Express cards always start with 37 and have 15 digits. Visa cards always start with 4 and have 16 digits. Master Card always starts with 5 and have 16 digits. Finally, Discover Card always starts with a 6 and has 16 digits.

While tokenization offers a great deal of security, it does not secure the initial part of the process. At some point credit card information must be transmitted to the Issuer. If this information is sent in plain text, an adversary could easily intercept the data. The plan was to have the iSafe application collect the credit card information of the user at the point of registration. The credit card information would be encrypted with 128 bit AES (e.g. RijndaelManaged class in Microsoft’s .NET Framework) and sent over a secure channel to the Issuer. The key and IV would be encrypted with an asymmetric encryption algorithm, such as the RSACryptoServiceProvider class provided by Microsoft’s .NET Framework. The Issuer would verify the user and account information and send a universal payment token back to the iSafe application. This universal payment token would be stored on the iSafe database in place of the actual credit card number. When the user wants to make a purchase, the universal payment token is sent to the merchant. The rest of the process continues, until a unique merchant token is returned to the merchant. This token is then stored on the merchants database for future use. Since this token is unique to the merchant, an adversary who steals them would be unable to use them.

3.7 LESSONS LEARNED

The main lesson learned is that communication is important in any team or group project. There were two distinct parts to this problem, but we never got to the point to where we could connect them. It is often hard to meet with group members in person to discuss things due to conflicting schedules. It is also important to understand that a problem can change at any time. The design and scope of the project changed after discovering tokenization. How to implement this and best simulate the process became an additional challenge. Understanding the scope of a project and your limitations are also important. Based on the initial knowledge that we had, it may have been beneficial to reduce the scope of the project or least have divided the project into smaller sub-problems. This way, even if the entire payment transaction process was not working, we could have at least implemented the storing of salted hashed passwords on the database or some other aspect of the overall project.

At times I was unsure of how much of the actual process to simulate, so I felt like I was setting my own guidelines. Creating a Token Service Provider and Token Requester exactly how it is done in the industry is far too much, but it is also possible that I did not simulate enough of the key points. In hindsight, I would have generated the payment tokens as explained in section 3.6. The scope of our simulation is something we needed to discuss.

We tried to rely on github instead of in person discussions, which I think was a problem. Github is a great tool, but it cannot replace communication. During the research and design process, github worked fine. We would check it and see what was done and read each others work. Later on, I uploaded the databases to github, but I did not see any code for the iPhone application and don’t know what progress was made. I assumed I would get an update when the time came. The key lesson here is to be more assertive instead of waiting for someone to come to you.

1. IMPLEMENTATION AND RESULTS

The first part of the implementation was the Issuer database.



Figure 2: Issuer UML class diagram

This was the first iteration of the Issuer which assumed a token vault. A later iteration would have implemented the token generator. It was also assumed that the merchant had already registered with the Token Service Provider. When a transaction is made, the Issuer can verify that the merchant is registered (in this case it is always true). Only some basic information is stored in order to simulate the transaction process. The transaction amount is checked against the accounts available limit to determine if there are enough available funds. If the transaction is approved, the account number is mapped to a unique merchant token and sent back to the Acquirer. Here is a portion of the database that was created in MySQL Workbench:

CREATE TABLE account (

accountNum VARCHAR(50),

avlLimit DOUBLE,

ID INT,

CONSTRAINT acount\_pk PRIMARY KEY (accountNum),

CONSTRAINT acount\_fk FOREIGN KEY (ID) REFERENCES cardHolder (ID)

The basic CRUD functionality for the Issuer was done on Visual Studio. A few dummy users and dummy accounts were created. Transaction information would be known at the point of sale and passed down to the Issuer, but for testing purposes, dummy transaction information was also input into the database. It displays the database on a Windows Form and has the ability to add, delete and edit users and user information and save those changes to the database. The idea here was to illustrate the simulation process.

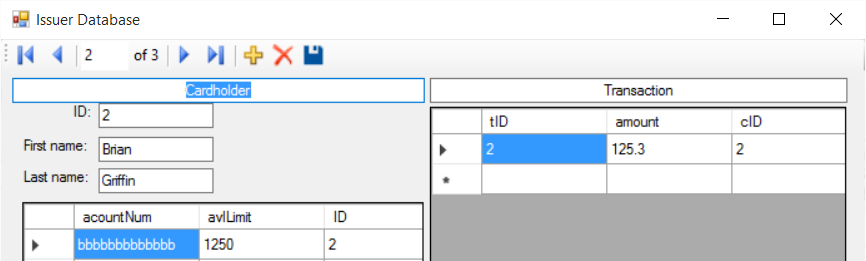


Figure 3: Windows Form with CRUD functions

The next step was to create the Acquirer database. The UML class diagram is shown below.



Figure 4: Acquirer UML class diagram.

The information in the Acquirer database includes all of the data collected from the iSafe application along with transaction information (merchant and customer id’s, amount, date, and universal payment token data). For testing purposes, the database was pre-loaded with dummy information. Functionality includes the basic CRUD functions as well as the payment transaction functions (authorization, capturing, clearing). Data is displayed on a Windows Form identical to the Issuer. The Acquirer connects to the Issuer to verify the customer. The authorization function verifies the users first and last name and checks if there is enough funds to complete the transaction. The results are displayed in the message box for simulation and testing purposes.

Console.WriteLine("Connecting to issuing bank...");

conn.Open();

string sql = "SELECT fName, lName, avlLimit FROM cardholder As c INNER JOIN account AS a ON c.ID = a.ID";

MySqlCommand cmd = new MySqlCommand(sql, conn);

MySqlDataReader rdr = cmd.ExecuteReader();

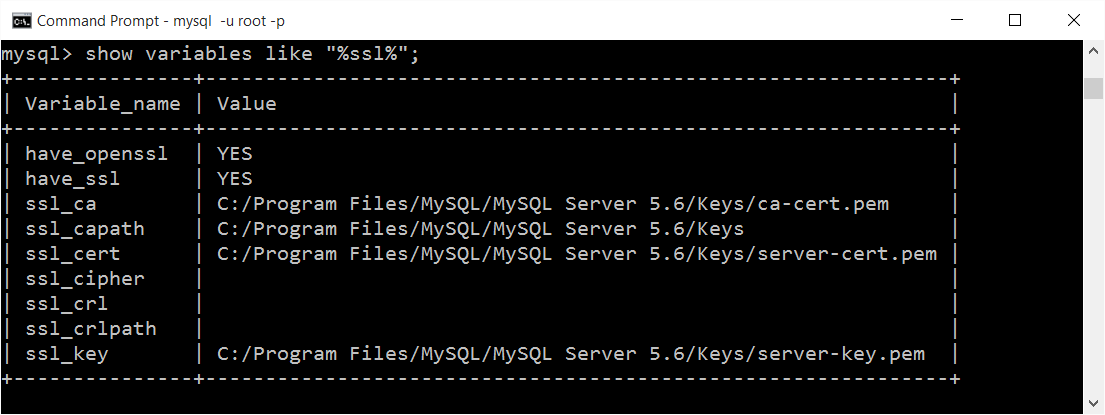
…

if ((Convert.ToDouble(amount) <= Convert.ToDouble(balance)))

MessageBox.Show("Transaction approved.")

If the transaction is approved, then the clearing function is called. This is where the Issuer matches the universal payment token it receives from the Acquirer and matches it with the actual credit card number of the customer. The Issuer assigns a unique token specific to this merchant and sends this token back to the merchants Acquirer (the Acquirer should send this back to the merchant for storage). The next step was to secure the communication between Acquirer and Issuer.

The tool used to secure communications was OpenSSLv1.0.2e for Windows. The MySQL server was configured such that an SSL connection is required. If an SSL connection is not available, then the connection will fail. The following image shows the path of the Certificate Authority file and the path to the Client Key file. It also confirms that SSL is enabled. Note that the key is self-signed.



The next figure shows a test of a secure connection.

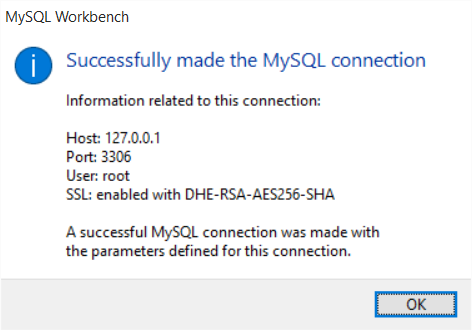


Figure 6: Secure connection example

1. CONCLUSION AND FUTURE WORK

This was a difficult and challenging problem. Neither of us had any experience or knowledge of the payment transaction process, creating server side code, or developing an iPhone. Researching and understanding the problem took much longer than expected, especially since we did not come across the concept of tokenization until after our initial research was completed. This increased the complexity of the problem and forced us to change our initial design. The benefits of doing this are that we now understand how a real world working mobile payment system works. Future work would include creating an Android application that simulates the payment tokenization project.

1. BIBLIOGRAPHY

1. <http://www.frbsf.org/cash/files/FedNotes_Evidence_from_DCPC.pdf> [↑](#endnote-ref-1)
2. <http://www.emc.com/collateral/analyst-reports/h13939-mobile-e-commerce-friend-or-foe.pdf> [↑](#endnote-ref-2)
3. <http://blog.cryptographyengineering.com/2015_09_01_archive.html> [↑](#endnote-ref-3)
4. EMV Payment Tokenization Specification, Technical Framework Version 1.0, March 2014, page 13. [↑](#endnote-ref-4)
5. EMV Payment Tokenization Specification, Technical Framework Version 1.0, March 2014, page 25-26. [↑](#endnote-ref-5)
6. EMV Payment Tokenization Specification, Technical Framework Version 1.0, March 2014, page 12. [↑](#endnote-ref-6)
7. <http://www.engadget.com/2014/10/02/apple-pay-an-in-depth-look-at-whats-behind-the-secure-payment/> [↑](#endnote-ref-7)
8. <http://www.engadget.com/2014/10/02/apple-pay-an-in-depth-look-at-whats-behind-the-secure-payment/> [↑](#endnote-ref-8)
9. <https://www.braintreepayments.com/blog/definitions-for-the-payment-processing-industry/> [↑](#endnote-ref-9)
10. <https://crackstation.net/hashing-security.htm> [↑](#endnote-ref-10)
11. <http://dev.mysql.com/doc/refman/5.7/en/security-guidelines.html> [↑](#endnote-ref-11)
12. iOS Security Guide, September 2015, page 18-20 [↑](#endnote-ref-12)
13. <http://www.zdnet.com/article/how-malware-finally-infected-apple-ios-apps-xcodeghost/> [↑](#endnote-ref-13)
14. Payment Card Industry (PCI) Data Security Standard, Requirements and Security Assessment Procedures, Version 3.1, April 2015.

    Appendix

    This is sample code of the Issuer that saves to the database after the CRUD functions are performed.

    private void accountBindingNavigatorSaveItem\_Click(object sender, EventArgs e)

    {

    try

    {

    this.Validate();

    this.cardholderBindingSource.EndEdit();

    this.accountBindingSource1.EndEdit();

    this.merchantBindingSource.EndEdit();

    this.trnsactionBindingSource.EndEdit();

    this.tableAdapterManager.UpdateAll(this.issuerDataSet);

    }

    catch (System.Exception ex)

    {

    MessageBox.Show(ex.ToString());

    }

    }

    This simulates the authorization process for the Acquirer.

    private void Authorize\_Click(object sender, EventArgs e)

    {

    //get the currently selected user from the userBindingSource.

    MessageBox.Show(((DataRowView)this.userBindingSource.Current)["fName"].ToString());

    string fname = (((DataRowView)this.userBindingSource.Current)["fName"].ToString());

    MessageBox.Show(((DataRowView)this.userBindingSource.Current)["lName"].ToString());

    string lname = (((DataRowView)this.userBindingSource.Current)["lName"].ToString());

    string amount = "";

    //customer information

    string cfname, clname, balance = "";

    try

    {

    MessageBox.Show(((DataRowView)this.trnactionBindingSource.Current)["amount"].ToString());

    amount = (((DataRowView)this.trnactionBindingSource.Current)["amount"].ToString());

    }

    catch (Exception ex)

    {

    MessageBox.Show("No transaction for this user.");

    }

    string connStr = "server=localhost;userid=root;password=iSafePW11;persistsecurityinfo=True;database=issuer";

    MySqlConnection conn = new MySqlConnection(connStr);

    try

    {

    Console.WriteLine("Connecting to issuing bank...");

    conn.Open();

    string sql = "SELECT fName, lName, avlLimit FROM cardholder As c INNER JOIN account AS a ON c.ID = a.ID";

    MySqlCommand cmd = new MySqlCommand(sql, conn);

    MySqlDataReader rdr = cmd.ExecuteReader();

    while (rdr.Read())

    { //check the first and last name of the customer

    if (fname == rdr[0].ToString() && lname == rdr[1].ToString())

    {

    Console.WriteLine(rdr[0] + " -- " + rdr[1] + " -- " + rdr[2]);

    cfname = rdr[0].ToString();

    clname = rdr[1].ToString();

    balance = rdr[2].ToString();

    }

    }

    rdr.Close();

    Console.WriteLine(Convert.ToDouble(amount));

    Console.WriteLine(balance);

    //check if customer has enough funds in account

    if ((Convert.ToDouble(amount) > Convert.ToDouble(balance)))

    MessageBox.Show("Transaction denied. Not enough funds in account.");

    else

    {

    MessageBox.Show("Transaction approved.");

    wasApproved = true;

    }

    }

    catch (Exception ex)

    {

    Console.WriteLine(ex.ToString());

    }

    conn.Close();

    Console.WriteLine("Done.");

    }

    Other Sample codes, charts, and diagrams are shown above. [↑](#endnote-ref-14)