View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by TELKOMNIKA (Telecommunication Computing Electronics and Control)

TELKOMNIKA, Vol.9, No.2, Agustus 2011, pp. 365~376
ISSN: 1693-6930
accredited by DGHE (DIKTI), Decree No: 51/Dikti/Kep/2010

Implementing the Payment Card Industry (PCI) Data Security Standard (DSS)

Enda Bonner[1], John O' Raw[1], Kevin Curran*[2]
[1]Department of Computing, Letterkenny Institute of Technology, Ireland
[2]Faculty of Computing & Engineering, University of Ulster, Derry, Northern Ireland
e-mail: kj.curran@ulster.ac.uk*

Abstrak

Didukung meningkatnya kriminalitas online, maka diperkenalkan standar keamanan data (DSS) industri kartu pembayaran (PCI) yang menguraikan sebuah subset dari prinsip inti dan persyaratan yang harus diikuti, termasuk tindakan pencegahan yang berkaitan dengan perangkat lunak yang memproses data kartu kredit. Kebutuhan untuk mengimplementasikan persyaratan ini dalam aplikasi perangkat lunak yang telah ada dapat menghadirkan pemilik dan pengembang perangkat lunak dengan berbagai isu. Makalah ini menghadirkan sebuah solusi generik untuk isu sensitif dari penyesuaian PCI, dimana pemrograman berorientasi aspek (AOP) dapat diterapkan untuk memenuhi kebutuhan penyembunyian nomor rekening utama (PAN). Arsitektur yang diusulkan memungkinkan sejumlah tertentu kode untuk ditambahkan yang memotong semua metode yang ditetapkan dalam aspek ini, tanpa memperhatikan tambahan lebih lanjut untuk sistem, sehingga mengurangi kebutuhan kerja untuk aspek pemeliharaan. Konsep ini dapat memberikan wawasan bagaimana pendekatan persyaratan PCI untuk melakukan tugas. Artifak perangkat lunak seharusnya juga harus melayani sebagai panduan untuk pengembang mencoba untuk mengimplementasikan aplikasi baru, di mana keamanan dan desain adalah elemen fundamental yang harus dipertimbangkan melalui setiap fase dari siklus pengembangan perangkat lunak dan bukan sebagai renungan.

Kata kunci*: industri kartu pembayaran, kartu kredit, nomor rekening utama, standar keamanan data*

Abstract

Underpinned by the rise in online criminality, the payment card industry (PCI) data security standards (DSS) were introduced which outlines a subset of the core principals and requirements that must be followed, including precautions relating to the software that processes credit card data. The necessity to implement these requirements in existing software applications can present software owners and developers with a range of issues. We present here a generic solution to the sensitive issue of PCI compliance where aspect orientated programming (AOP) can be applied to meet the requirement of masking the primary account number (PAN). Our architecture allows a definite amount of code to be added which intercepts all the methods specified in the aspect, regardless of future additions to the system thus reducing the amount of work required to the maintain aspect. We believe that the concepts here will provide an insight into how to approach the PCI requirements to undertake the task. The software artefact should also serve as a guide to developers attempting to implement new applications, where security and design are fundamental elements that should be considered through each phase of the software development lifecycle and not as an afterthought.

Keywords: *credit card, data security standards, payment card industry, primary account number*

1. Introduction

Credit card usage has increased and as a consequence, so has credit card fraud. The iC[3], a partnership between the FBI and National White Collar Crime Centre in the US, placed credit card fraud third on its list of reported claims [1]. In an effort to combat this, the founding brands of the Payment Card Industry Security Standards Council (PCI SSC); American Express, Discover Financial Services, JCB International, MasterCard Worldwide and Visa Inc., produced the PCI data security standard (PCI DSS). The purpose of the PCI DSS is to help facilitate the adoption of security standards and reduce card fraud [2]. To assist the various stakeholders within the payment card industry, the PCI SSC released the PCI DSS, updated to version 1.2 in October 2008 [3]. The document is a set of comprehensive requirements for enhancing payment account data security. The standard incorporates existing standards such

Received June 8th, 2011; Revised June 15th, 2011; Accepted June 18th, 2011

as Visa's Cardholder Information Security Program (CISP) and MasterCard's Site Data Protection (SDP) into a universal standard to be adopted by all stakeholders involved in the payment card industry. The PCI DSS consists of a core set of principles and requirements that must be followed to achieve PCI compliance. The PCIS DSS document expands on each of the requirements by explaining them in greater detail and outlining testing procedures to be put in place to ensure each of the requirements are met. These tests must be passed on an annual basis to retain compliance. The PCI DSS splits the stakeholders within the payment card industry into levels depending on the volume of their annual transactions [4].

The standards are then enforced by each payment card brand through their individual programs like CISP and SDP. The PCI Security Standards Council themselves do not impose penalties or deadlines; this is left to the discretion of the payment card companies to enforce the standards. To show their commitment to the requirements, it was widely reported that Master Card introduced fines of up to $375,000 per year for non-compliance. The high profile of card information thefts has also seen 38 US States adopt laws putting the onus on the merchant to contact customers whose data has been breached [5]. The requirements cover all aspects of credit card usage from the building of secure networks to the creation and maintenance of an Information Security Policy. It is important from a business's overall perspective that all of the requirements are adhered to. However for the purpose of this research only requirements pertinent to software design will be investigated.

The PCI DSS is available from the PCI security standards council website. The document lists the 12 requirements that must be followed to ensure PCI compliance [6]. Developers can use the PCI DSS as a guide when building new software to ensure it complies with the standards. The problem arises in systems that have been in existence before the PCI DSS was produced. We endeavour here in this work to find the most suitable solution to this problem. Over the years, there have been many solutions proposed to solve the problems of securing older systems. These solutions have included rebuilding or wrapping entire systems with newer technologies, using middleware technologies or 'crosscutting' technologies such as aspect orientated programming along with various cryptographic methodologies. Each of the proposed solutions has their advantages and disadvantages. This research seeks to ascertain the most viable solution to the PCI DSS problem within existing applications.

2. PCI DSS Software Compliance

There are two main areas to be addressed when applying the PCI DSS requirements to a software application. The first is the displaying of sensitive information being processed or transmitted by the application and the second is the storing of sensitive data within the application. Blackwell [7] argues that PCI DSS offers a false sense of security to clients. The argument being that despite all the technical controls, insiders provide the principal threat to card information theft. Another threat is the theft of hardware devices. A car stolen in 2006 containing the laptop of an Ernst & Young employee resulted in the loss of card information of 243,000 Hotel.com users [8]. Requirement 3 in the PCI DSS looks to avoid such scenarios – "If an intruder circumvents other network security controls and gains access to encrypted data, without the proper cryptographic keys, the data is unreadable and unusable to that person" [9]. This requirement looks to mask any occurrences of the primary account number (PAN) throughout the system, as well as encrypting any sensitive data that is stored. Table 1 illustrates how each credit card element must be treated.

Table 1. Credit Card elements [9]

	Data Element	Storage Permitted	Protection Required	PCI DSS Req. 3.4
Cardholder Data	Primary Account Number (PAN)	Yes	Yes	yes
	Cardholder Name	Yes	Yes	No
	Service Code	Yes	Yes	No
	Expiration Date	Yes	Yes	No
Sensitive Authentication Data	Full Magnetic Stripe Data	No	N/A	N/A
	CAV2/CVC2/CVV2/CID	No	N/A	N/A
	PIN/PIN Block	No	N/A	N/A

The PCI DSS requirements and security assessment procedures document contains a detailed explanation on how each requirement must be implemented. Software developers must then decide on the most appropriate method to retrofit them into the existing system. Some of the main problems facing development teams when retrofitting systems with new features include code bloat and a design structure lacking continuity, mainly due to bug fixes which hinder code readability [10]. Furthermore the lack of subject matter experts (SME's), any of the original developers or up to date documentation makes the task even more difficult. Over the years there have been many approaches to add additional functionality such as extra security to older systems. PCI DSS requirements can be applied by modifying the original code. This however is generally unrealistic due to system size and complexity. For large systems this could mean the addition of thousands of lines of code, resulting in code bloat and the mixing of application code and security code, known as code tangling. More importantly, the disruption of already working code can lead to problems such as bugs and inefficient code. This therefore is an unviable approach for implementing PCI DSS requirements within enterprise applications. Middleware technologies like common object request broker architecture (CORBA) have also been used to bridge the gap between incompatible systems. Working on the basis of [11] adapter pattern, CORBA was initially designed to allow interoperability between non compatible systems but as a result security features were limited. The need for additional features such as security has increased the complexity of middleware standards [12]. These drawbacks, along with high running costs and tight coupling make CORBA an unsuitable solution to incorporate the PCI DSS requirements [13].

The emergence of web services has also allowed developers to extend older legacy systems. Legacy systems can be 'wrapped' within a web service [14]. This approach removes some of the problems associated with middleware technology including loose coupling and a reduction in complexity. Web services also offer established security standards and mechanisms such as web service (WS) security and WS security policy [15]. While securing data transmissions, web services can fall short when implementing the PCI DSS Requirements. The issue of securing processed data is not directly addressed. To mask sensitive data, developers may have to tinker with the existing code and this can have a knock-on effect on the system. Although each of the technologies discussed have their merits, none of them produce the complete solution. More recently, developers have looked to aspect orientated programming (AOP) as a more elegant solution to such problems. AOP eradicates the problems associated with the previously proposed solutions. AOP can support separation of concerns by modularizing these crosscutting concerns into aspects [16]. This will allow the implementing of the PCI DSS requirements separate from the existing code, reducing the complexity of the system and code bloat. It also allows for a more maintainable and flexible system where changes made in the aspect can reach across all relevant areas of the system [17].

2.1 Storing Data

The PCI DSS requirements also state that credit card information should only be stored if it is absolutely necessary. Any information stored must be made unreadable by use of one-way hash functions, strong cryptography, truncation, index tokens and securely stored pad. Proper key management procedures must be followed in generating cryptographic keys and protecting the keys [9]. There is no definitive guide on how to implement the database encryption requirement of the PCI DSS. There are many factors to consider including the database in use, connections, size and the code to access the database. The easiest technique to protect the PAN would be truncation or one way hash systems. Although these secure the information, they are irreversible techniques. This leaves the information redundant as the original values cannot be retrieved. If these techniques are being employed, developers must question the need to store this redundant data. Encryption within a software system offers high levels of security and confidence. Sensitive information can be encrypted at entry into the system and decrypted upon exit. This protects the data throughout the system including outputs into log files, debug statements and database or file storage. It ensures that even an inadvertent information leak is secure as the data is encrypted at all times. When building a system from scratch this is undoubtedly one of the best techniques. However to refit this model into an already existing system is impracticable. AOP could be implemented to encrypt/decrypt information at entry and exit points. The encrypted fields however would require their data types

and sizes to be changed throughout the system along with each of the corresponding database fields [18].

Transparent database encryption (TDE) is another method of database encryption that is generally carried out at column level. The data is encrypted before it is written and decrypted after it is read from the database. This method is described as transparent as it requires no changes to the application code making it easier to implement in legacy systems. There are however trade-offs including performance hits. This is mainly attributed to how the data to be encrypted is intercepted and then written to the database. There is no one solution to employing database encryption and many methods may have to be explored before a suitable solution is discovered [19].

Determining the method to use when applying encryption is only a fragment of the overall solution. Key Management including key creation, storage, distribution, recovery and destruction are critical to cryptosystem security and all these must be addressed to ensure PCI DSS compliance. Although not specific, the PCI DSS outlines in detail the requirements of the key management process. Strong key generation is vital to the security of a cryptosystem. Weak keys are vulnerable to attack and if deciphered give attackers access to encrypted information. Encryption keys are generated out of large relatively prime numbers and it is important that the correct procedures are followed to create them properly. The PCI Glossary outlines the cryptographic methods to be used along with their corresponding key size. The requirements do however state that split knowledge or dual control of keys are applied. This requires the generation of key components which are divided between individuals. All components are required to form the key and no single component provides knowledge of the key. Split knowledge prevents the creation of a 'super user' who would have unrestricted access to any encrypted information. It can also protect systems from malicious employees and potential attacks by storing the key components in separate locations. This technique is generally used for 'master keys' which are used to provide protection to the underlying keys which encrypt the information [20]. The holders of the keys must also sign a form stating that they understand their responsibilities as key custodians [20].

Generated keys must be then stored in a secure location. Forms of storage include external hardware security module (HSM) or external software. HSM's are regarded as a high form of security as they undergo rigorous certification including FIPS 140-1/2 [20]. External software can provide similar levels of security without incurring similar costs. Requirements also recommend that keys are stored in the 'fewest possible locations and forms' [9]. A single centralized location with a backup would be the fewest storage locations possible. This may work for smaller applications but larger systems need to assess traffic and storage volumes as one centralised key store could lead to bottle necks. Another security approach is to restrict access to the secure location. Access should only be granted on a 'need-to-know' basis and must be 'managed independently of native operating system access control mechanisms' [21].

This requirement outlines the need for an extra level of security before users can get access to the keys. To complement this, access logs should also be recorded and kept in a separate location that key holders cannot access. The key management process must also allow older or compromised keys to be replaced. As with all products, encryption keys have a lifespan and given enough time a key can be cracked. For this reason the PCI DSS requires the re-keying at least once a year. Like other PCI DSS requirements, there is no set answer to this requirement. It is a topic that must be investigated and a policy put in place to outline the procedure. Re-encrypting data with new keys on a regular basis is both costly and time consuming. Alternatively adding extra keys requires more space and could complicate the process. If a key is compromised, all the corresponding information must be re-encrypted with a new key and the old key must be destroyed to prevent further impact. Policies and procedures must also be put in place to prevent the substitution of unauthorised keys. Measures already outlined such as key splitting, strong access controls, logs along with regular audits should help in the prevention of substitution attacks. It is up to the parties involved to consider all the relevant factors and to strike a balance. Following the PCI DSS may not necessarily be enough to secure a system. Underlying security threats such as the systems vulnerability to SQL Injection attacks must also be addressed. Issues like this could jeopardise the entire PCI compliance and should also be reviewed. When a company is satisfied with the key management structure, it must all be documented to adhere to PCI DSS requirements.

2.2 PCI Compliance

The PCI DSS applies to all UK Level 1 merchants (those processing over 6 million annual transactions) since September 2010. A 2010 survey indicated that 58% of these audited merchants were compliant; this however only represents 11% of the UK organisations processing credit and debit card data although most companies aspire to be PCI compliant, there are issues [21]. Despite the obvious costs incurred in becoming PCI compliant, it is not regarded as a reason for non-compliance. The costs for non-compliance are often greater than the costs of implementing it. Non-compliance can lead to substantial fines but more importantly, a potential loss of business. This is a real threat to companies as the US National Archives & Records Administration reports that 50% of businesses that lose their critical data for 10 days or more have to file for bankruptcy immediately. A possible reason for the high non-compliance may be the sheer volume of the requirements. From a high level there are only 12 core requirements. However, each requirement has a number of sub requirements resulting in a total of 214 issues to be addressed. Furthermore, there are no recommended implementations for every requirement, which can lead to ambiguity for companies who lack the technical knowledge to implement them. This ambiguity could be regarded as another possible reason for the non-compliance statistics.

Due to this lack of technical knowledge, many businesses outsource their security needs. With some of the requirements open to interpretation, a range of solutions are offered making it difficult for businesses to choose a suitable solution provider. Each of the offered solutions differs due to the ambiguity of the requirements. This can make the task of employing a suitable PCI DSS solution provider both time consuming and frustrating for small businesses. The outsourcing or buying commercial off the shelf (COTS) solutions to solve PCI compliance can also be problematic. As security controls tighten, attackers become more inventive to find ways around these new controls. These solutions are secure at a point in time but as attacks become more advanced the requirements and security controls need updating. As a result of this, the changing requirements are also regarded as a non-compliance factor.

Recent reports suggest that organisations who suffered breaches were 50% more likely to be non PCI compliant [21]. Some claim that all the PCI DSS only offers users a ‘false sense of security’ as there are ways around the security controls [7]. An attack on Hannaford Brothers, a PCI compliant company lends some belief to these claims. The sophisticated attack captured 4.2 million credit card details as it was transmitted from the store and the credit card company [8]. This did not help the enforcement of the standard and may be viewed by some smaller customers as another reason to avoid the costs of compliance. These are some of the potential reasons outlined for the high non-compliance rate amongst companies. However with the statistics showing the costs of a data breach, coupled with the fines from the credit card companies, the PCI DSS should see a higher adoption rate in the future. The security of sensitive information is a continuous process. Companies that make the effort to become compliant must monitor their security on a regular basis and not just strive to meet the requirements for their annual audit.

3. PCI DSS Experimental System

For the purpose of this task, a non-compliant ‘Prototype’ system was developed. An overview of the system and a diagram of its flow is provided in Figure 1. It is based on an airline system that takes mocked up API (Advanced Passenger Files) as input and returns XML files as output. The assumption was made that any sensitive data was required by downstream applications and so it was not masked in the input or output files and would be securely transmitted. The system polls the input directory for any new files and depending on the file type the file is processed. Input files are parsed, converted into Java objects and stored in a database using Hibernates object-relational mapping (ORM) framework. All passenger data was stored in one large table, this could be normalised, but was not significant to the process of the project. The presence of a ‘.req’ file in the input directory indicates a request for data and so the information is read from the database. The data is then converted into an XML format using JAXB and written to the output directory. Logging is carried out throughout the system, documenting all forms of information including debugging and error messages.

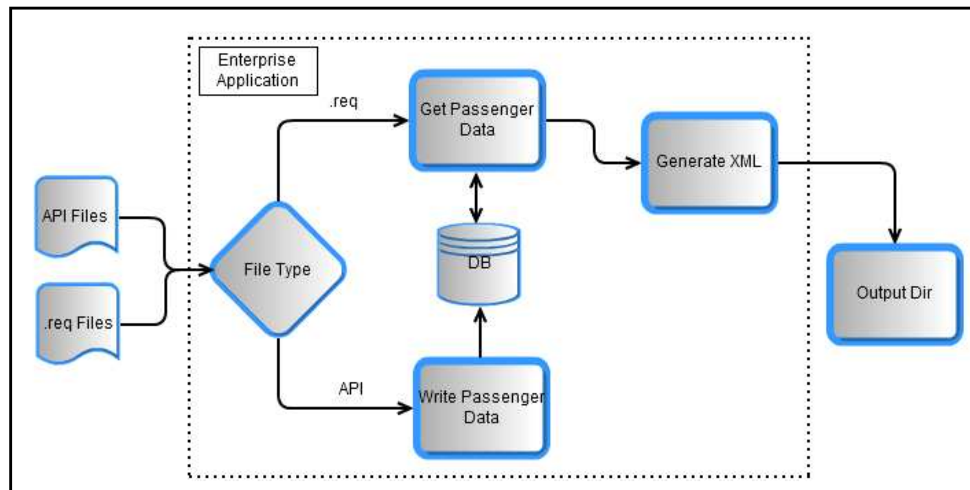

Figure 1. System Overview

The system follows a very general flow incorporating a database with input and output files, allowing it to be easily interpreted and related to almost any Enterprise Application. Technologies such as JAXB, Maven, Log4j, Java configuration and Hibernate were intentionally incorporated as they are all widely used within Enterprise Applications; however specific requirements such as business rules were omitted to allow a greater focus on the implementation of the PCI Requirements. The implementation of the PCI Requirements should have a minimum impact on this functionality. The requirements will also look to be implemented in a generic fashion so as to apply broadly to most Enterprise Applications. The two main areas to be addressed within the system to meet PCI compliance are; masking the PAN anywhere it is exposed and rendering the PAN unreadable anywhere it is stored.

3.1 Masking the PAN

The 'Masking the PAN' requirement is very general as the PAN can be exposed in numerous locations. Based on the design of this system, two main areas of potential exposure were identified as log files and console outputs. Although these are the only two areas being addressed here, the ability of Aspect Orientated Programming to address cross-cutting concerns autonomously should permit it to provide similar solutions across other areas. This capability will allow these concerns to be broadly addressed, without the time consuming effort of having to deal with each one individually. Aspect Orientated Programming can be accomplished using either Spring AOP or AspectJ. Spring AOP is generally regarded as the easier of the two to implement as it does not require the introduction of AspectJ compiler into the build process [21].

Spring AOP is an XML 'proxy-based' technique as it is based on the Decorator Pattern, a form of the Proxy Pattern. This allows the Spring AOP to create a subclass of the original class at runtime. The newly created subclass, containing the aspect advice, then delegates operations to the original object. This process has the potential to cause some performance overhead due to the extra creation of classes. Spring AOP also has limits to as to what Pointcuts can be intercepted, restricting its overall usage. AspectJ has no such limitations supporting all types of Pointcuts and the use of a build tool such as Maven permits easy integration of the compiler, subject to popular views. The compiler facilitates the weaving of the aspect elements into the code at runtime. AspectJ also enables all the aspect elements to be encapsulated in a class type format which reduces the impact on the original system. This also improves security as all the elements are contained within the code and cannot be easily tampered with, as opposed to Spring AOP XML files. AspectJ allows easy integration of both compile and post-compile time weaving, where compile time weaving allows advice to be weaved into a program during compilation and post-compile time weaving allows advice to be weaved into already existing jar files [3]. After the analysis, AspectJ was determined as the better approach to solving the 'Masking the PAN' requirement.

3.2 Rendering PAN unreadable during storage

To render the PAN unreadable, a form of encryption must be applied. There are many forms of encryption and encryption techniques. The merits of each will be discussed in this section. The PCI DSS offers examples of “Approved Standards” that can be used to gain PCI Compliance. The Approved Standards include both symmetric and asymmetric algorithms including AES and RSA. Since the design of this system has only one application accessing the database, this removes the need for key distribution and allows for the secure implementation of symmetric encryption. The AES algorithm was chosen over the Triple DES algorithm as it is regarded as a being a faster more efficient algorithm and is regarded by NIST as Triple DES’ successor [21]. The advanced encryption standard is also recommended by open web application security project (OWASP) in 2009.

To achieve transparent encryption the use of Hibernate user defined types were investigated. User defined types, amongst other things, can let information be accessed or manipulated just before reading or writing to the database. This technique utilises Hibernate mapping files, Hibernate annotations and Java classes. Fields requiring encryption are identified using the annotations and these references the type definition in the Hibernate mapping file, which in turn references the Java class. This framework allows values to be easily manipulated before being read or written to the database giving the developer full control of the data. Java Simplified Encryption (Jasypt) also offers a form of transparent database encryption using password based encryption. However after some analysis of Jasypt, it was concluded that it would not offer as much control as Hibernates user defined types.

The Java platform offers a KeyStore class that represents a storage facility for cryptographic keys (Java 6 API) in 2010. The class also provides an administrative tool called ‘keytool’ which allows administrators full control over all keys within the key store. There are three types of key store algorithms; JKS, JCEKS and PKCS12. JKS is the default algorithm however it does not support the storage of secret keys, while the PKCS algorithm does not support a fully functional key store. JCEKS provides storage for secret keys and users a stronger encryption than JKS and for these reasons the JCEKS was preferred (Java Security, 2001). The keytool allows administrators to create or manipulate keys outside of the system, but the application will also need access to the keys to encrypt/decrypt data. The parameters required to retrieve the secret key from the key store are; the location and password of the key store along with the key’s alias and password. These parameters can be stored in a configuration file to allow the keys to be easily rotated, but must be encrypted to prevent unauthorized usage. To perform this task, a form of password based encryption ‘PBEWithMD5AndDES’ was researched. The encrypted data read from the configuration file will be decrypted with this algorithm and the values can then be used to retrieve the keys from the key store. The decryption will be carried out within the system to prevent the algorithm being revealed. With all the elements involved in the encryption process, the design strategy was analysed to ensure it was both solid and efficient.

3.3 System Implementation

The first step in implementing AOP was to add the AspectJ compiler to the build path. As the Maven build tool was used to build the system, the AspectJ compiler was easily added as a dependency to the POM file. The autonomous nature of AOP allowed the aspect to be added as an independent package to the existing system code system. The aspect was composed of two main elements; the Pointcut, the Advice. The next process was to implement the aspects for the runtime and post-runtime weaving. A separate aspect was defined to mask the PAN within the log file and the console. Runtime weaving was implemented to mask the PAN within the console. The key to achieving a solution to rendering the PAN unreadable when stored was to ensure all the required elements fitted together cohesively. The Singleton Pattern was implemented first to provide a framework for which to base the cryptography upon. The class that loads the required encryption parameters would follow this pattern for efficiency purposes. To achieve the Singleton Pattern the Encryption Properties class was created with a private constructor and a public method getProperties(). The encryption properties method checked to see if the static object ‘encProperties’ was initialised and if not, it was populated it with the values from the encryption properties file. Once the object was initialised it remained in memory and was returned from any further calls to the method, enforcing the Singleton Pattern. The ‘loadEncProperties’ loaded four parameters from the configuration file; key password, key

alias, store location and store password. The use of the configuration file allowed the keystore and its contents to be easily manipulated for key rotation and security reasons. This feature, although useful, required an extra layer of security as these values could not be stored in plaintext. The parameters were not explicitly labelled within the properties to avoid them being easily distinguished or recognised. Storing the values this way made it easier for the administrator to perform key rotation or modify the key store in any way. To enable key rotation an extra class was added to allow the administrator to generate new configuration file values.

The class was created with a 'main method' to allow it to be run from the command line. The method accepted all four parameters required for the encryption configuration file, along with the administrator's password. Once the password was verified, the method encrypted the values and output them to the administrator via the console. These encrypted values were then decrypted within the password encryption class using 'PBEWithMD5AndDES' cryptography as defined in the analysis. The decrypted values were set in the 'encProperties' object and returned to the Encryption class. The Encryption class then used the object to retrieve the related key from the key store via the Key Store Retriever class. If the secret key was successfully retrieved, values were encrypted or decrypted as necessary. As with all Java applications, all checked errors had to be dealt with, resulting in them being caught. However, a caught exception within the cryptosystem could mean a configuration file was tampered with. Considering this threat, anytime an exception was caught the error was logged and the system shutdown immediately. As identified in the analysis, Hibernate user types were used to implement a form of cryptography transparency. The key element in this process was the creation of the user type in the Hibernate mapping file. The declaration in the mapping file associates the annotation 'encryptString' to the class 'EncryptedStringUserType'. This type definition declared in the Hibernate mapping file allowed any attribute proceeded by the annotation to be passed to the 'EncryptedStringUserType' class. The class implements the user type class, overrides the Hibernate 'nullSafeSet' methods to permitting the re-direct of the string values to the cryptosystem**Error! Reference source not found.**. This methodology permitted encryption to be incorporated within the system by simply adding the annotation above the attribute to be encrypted, thus introducing transparent data encryption into the prototype system.

4. Results and Discussion

The testing of the finished article was an important aspect of the implementation of the PCI Requirements. Testing was carried out to ensure that the additional enhancements meet the requirements and also to evaluate the impact of these on the original prototype system. To measure the tests, each test was carried out on the original Prototype system and then the PCI_DSS system, containing the PCI enhancements. To assist with testing, two additional applications were created. The 'GenerateFile' application was used to create input API files for the system. This application generates API files of a specified size, populating the details with random strings or digits where appropriate. An application called the 'LogFileChecker' was also designed to assist with testing. The application accepts a file as input and scans it, printing out a list of potential credit card entries and total amount found.

4.1 Masking the PAN

The testing of this requirement incorporated the testing of the contents of the generated log file and the console. To maximise the amount of log entries, debugging level was set to 'ALL' within the Log4j configuration properties file on both systems. An API file consisting of 10 passengers was generated using the File Generator application and run through both systems, with each system generating a log file. Firstly the log file generated by the Prototype system was run through the LogFileChecker. The result was 31 potential occurrences of credit card numbers exposed within the log file, as shown in Figure 2. The output was manually verified to confirm that of 31 occurrences, 30 were actual credit card numbers with the final entry being a file processed by the system. Furthermore it was confirmed that there were 10 unique credit card numbers within the results, equating to the 10 credit entries in the API file processed. The log file generated by the PCI_DSS system was then run through the LogFileChecker. The results were conclusive. The Aspect employed to mask the PAN within the log files masked all potential credit card entries in the log file. This resulted in 100% coverage in masking the PAN.

The masking of the additional entry provided a false negative, giving a margin of error of only 3.23%.

```
24. 2011-03-23 23:21:22,916 [Thread-1] TRACE - returning '1746053089128195' as column: ccNumber2_0_
25. 2011-03-23 23:21:22,920 [Thread-1] TRACE - returning '3824673400332482' as column: ccNumber2_0_
26. 2011-03-23 23:21:22,924 [Thread-1] TRACE - returning '0045557747967632' as column: ccNumber2_0_
27. 2011-03-23 23:21:22,928 [Thread-1] TRACE - returning '7011548237807391' as column: ccNumber2_0_
28. 2011-03-23 23:21:22,931 [Thread-1] TRACE - returning '7749688429330290' as column: ccNumber2_0_
29. 2011-03-23 23:21:22,935 [Thread-1] TRACE - returning '2951753324751703' as column: ccNumber2_0_
30. 2011-03-23 23:21:22,938 [Thread-1] TRACE - returning '1501477067865926' as column: ccNumber2_0_
31. 2011-03-23 23:21:29,947 [Thread-1] INFO - File 'C:\InputDir\DOC21234567891011.txt' already processed
Potential Credit Card matches found:31
```

Figure 2. Log File Checker Output (Prototype_Log.txt)

A similar test was carried out to ensure the console output was masked, by re-running the API files through the system again. Console output is generally limited within Enterprise Applications, so for testing purposes all files read by the system and the creation of Passenger objects were output to the console. The console outputs of each system was saved as a text file and again run through the LogFileChecker, as shown Figure 3. The result was the detection of 11 potential credit card numbers, once again matching the 10 numbers in the API file, as shown Figure 4. Again the additional entry was a file processed by the system. The console output from the PCI_DSS system was run through the LogFileChecker, as shown Figure 5.

```
Potential Credit Card matches found:0
```

Figure 3. Log File Checker Output (PCI_DSS_Log.txt)

```
7. 7749688429330290
8. 2951753324751703
9. 1501477067865926
10. 6930972141447042
11. File Path: C:\InputDir\DOC21234567891011.txt
Potential Credit Card matches found:11
```

Figure 4. Log File Checker Output (Prototype_Console.txt)

```
Potential Credit Card matches found:0
```

Figure 5. Log File Checker Output (PCI_DSS_Console.txt)

Yet again the tests were conclusive, with 100% of potential credit cards masked and a false-negative margin of error of less that 1%. Based on the results, it was concluded that the use of Aspect Orientated Programming to meet the requirement of masking the PAN was a resounding success. All potential threats were masked within the system without effecting the original system code or flow.

4.2 Rendering the PAN unreadable where stored

The purpose of this testing was to confirm that the cryptosystem employed encrypted the data at rest. This testing was carried out via the MySQL console to prove that all attributes preceded by the Hibernate User Type annotation were encrypted within the database. The

attributes in question were the passenger's surname, credit card number and credit card type. The MySQL console was used as it gave an exact reading of the information within the database. The API file was run through the Prototype system. After the files were written to the database, they values were retrieved using MySQL. Figure 6 displays a selection of the database contents after the API file was processed by the Prototype system. The values of each row were manually checked against the data in the API file to confirm that they matched. The same file was then processed by the PCI_DSS system.

```
mysql> select ccNumber, ccType, ccExpiry, surname, firstName from passenger;
+------------------+--------+----------+----------------------+-----------------+
| ccNumber         | ccType | ccExpiry | surname              | firstName       |
+------------------+--------+----------+----------------------+-----------------+
| 4832087347158465 | VI     |     2095 | SXO                  | BAU             |
| 6930972141447042 | VI     |     1454 | IABDCHRWFHJHFKCHMUS  | R               |
| 2340798159715771 | MC     |     3490 | PFMVUKPGLDWCDRALBZM  | CNRXZ           |
| 1746053089128195 | MC     |     1150 | XKGN                 | ZNHTPYLZHQVRZ   |
| 3824673400332482 | MC     |     2567 | FJQPBSCWQVYSLNAXQ    | AHMAP           |
| 0045557747967632 | MC     |     3715 | RUKJIPQVOZWMOLZ      | HAVVTZLCJOSNP   |
| 7011548237807391 | VI     |     5769 | HUFJQZEPAI           | BKUOBPEQGFGRJ   |
| 7749688429330290 | AX     |     7557 | RSGFQUERIIEMWUHVIIX  | EGUF            |
| 2951753324751703 | VI     |     8582 | EUY                  | ZBXLMZNQFNHG    |
| 1501477067865926 | MC     |      810 | J                    | SKP             |
+------------------+--------+----------+----------------------+-----------------+
10 rows in set (0.00 sec)
```

Figure 6. MySQL output (prototype)

```
mysql> select ccNumber, ccType, ccExpiry, surname, firstName from passenger;
+----------------------------------------------+--------------------------+----------+----------------------------------------------+---------------+
| ccNumber                                     | ccType                   | ccExpiry | surname                                      | firstName     |
+----------------------------------------------+--------------------------+----------+----------------------------------------------+---------------+
| O4gHLghpWS7xS/DxxnFkjp1jZ+mff8zHlXBWZqbJKec= | I6pCrMVuEiIEdFE3q0lfgg== |     2095 | DsIo5X3D75wdIa09TiiRkw==                     | BAU           |
| yu5+TLd34VLBeg/X06C53J1jZ+mff8zHlXBWZqbJKec= | I6pCrMVuEiIEdFE3q0lfgg== |     1454 | l3HWhR0KHz2K4cBti/dzR7eTc1V18cBEvtCwIPFzgb0= | R             |
| yTlgukXzvChx1SSxvuAty51jZ+mff8zHlXBWZqbJKec= | 0Qtq4HVa0XzQyTHy/Win1w== |     3490 | ozOFwi07dX54jSfz0skPoQ3trsAPbsJownFf6XgMZZM= | CNRXZ         |
| SpK9MX7icLNOK7XEZ/1y651jZ+mff8zHlXBWZqbJKec= | 0Qtq4HVa0XzQyTHy/Win1w== |     1150 | /HpYarrAuMkWI9LmKgTDuw==                     | ZNHTPYLZHQVRZ |
| +DnThDWDFo+z17gw9o7aq51jZ+mff8zHlXBWZqbJKec= | 0Qtq4HVa0XzQyTHy/Win1w== |     2567 | vi+iJtyTdS4oM6/BAgMv6r//tXKIDXo4uUJRIyT7pWQ= | AHMAP         |
| 0FnddaCO6cF6JHIZSIOKgp1jZ+mff8zHlXBWZqbJKec= | 0Qtq4HVa0XzQyTHy/Win1w== |     3715 | u4wlfWw3E2HwD6BfDFS//Q==                     | HAVVTZLCJOSNP |
| ASnLrmeuy6I1Hw/UXVdKV51jZ+mff8zHlXBWZqbJKec= | I6pCrMVuEiIEdFE3q0lfgg== |     5769 | dCNhJ3twQ7MnRA5qgOyERA==                     | BKUOBPEQGFGRJ |
| Vy06Dobp2y4mBwvk67X0YZ1jZ+mff8zHlXBWZqbJKec= | qAzv59YWt+t5+zI0E7YF3g== |     7557 | pKkzxJDyjq8PMnDt4/6SeiPPSj6jbeq9XeSAZZWi65E= | EGUF          |
| wcmOwhNKE7eRGP71c3doKZ1jZ+mff8zHlXBWZqbJKec= | I6pCrMVuEiIEdFE3q0lfgg== |     8582 | Mz0EsmJCgB/tucZzQ3dpUw==                     | ZBXLMZNQFNHG  |
| Ue78yqKuAReNTEAzGIr/GZ1jZ+mff8zHlXBWZqbJKec= | 0Qtq4HVa0XzQyTHy/Win1w== |      810 | 0DxqW38KrINL6MN+pbf/rw==                     | SKP           |
+----------------------------------------------+--------------------------+----------+----------------------------------------------+---------------+
10 rows in set (0.00 sec)
```

Figure 7. MySQL output (PCI_DSS)

The output from the MySQL query is illustrated in Figure 7. The differences between the two queries can be seen instantly. The fields assigned for encryption are all encrypted, while the remaining fields are still in plaintext. Testing was also carried out to ensure that the cryptographic process implemented had no adverse effects on the data output. This was tested by carrying out and end to end test, ensuring that the data in the system generated XML output file matched that input in the API file. As stated previously it was assumed that the data was required to appear in plaintext and so the output XML could be compared with the API file input. A manual comparison of the files confirmed that all the data was correct and accounted for. This test also confirmed that cryptosystem had no adverse effect on the data. Based on the tests carried out, it was concluded that the cryptographic technique employed was a success. The technique provided transparent database encryption, with the minimal impact on the existing and no impact on the data being processed.

4.3 Performance Testing

To gauge the impact of the added functionality to the prototype system, performance tests recording the times taken to perform a read and write request were recorded. Three files

were created consisting of 10, 100 and 500 passengers. Each file was processed through the Prototype and PCI_DSS systems a total of ten times to get a more concise reading.

Table 2. Performance Results

	Max Time (ms)		Min Time (ms)		Average Time (ms)		% Difference
	Prototype	PCI_DSS	Prototype	PCI_DSS	Prototype	PCI_DSS	
10 PAX	1945	2782	1609	2348	1754.9	2500.9	70%
100 PAX	2766	4027	2530	3512	2667.2	3744.4	71%
500 PAX	4413	6144	4022	5869	4197.9	6002.9	70%

As expected, the addition of the two sspects and the Hibernate user type had an effect on the systems performance. The average performance hit over the three test files was an increase in processing time of 70%. This although substantial equates to approximately an extra 2 seconds on the processing of a file containing 500 passengers. Based on recent statistics (Airports Council International, 2009), the world's busiest airport processed on average 255,000 passengers a day. If all these passengers were to be processed, the system would take approximately an extra 17 minutes over the course of the 24 hrs. Based on these statistics, 70% is quite a large increase in processing time, however based on actual time they do not appear as dramatic, as Table 2. It was accepted that there was always going to be a performance hit, but based on the actual increase in processing time they could be deemed acceptable when the end result is PCI compliance and the improvement in security of the overall system. When all testing was taken into account the application of the PCI requirements was considered a success. Each of the solutions worked as expected and stood up to any tests carried out. The one pitfall was the hit in performance, however as stated previously, it was a small price to pay for compliance.

5. Conclusion

This research offered a generic solution to the highly sensitive issue of PCI Compliance. This is a real world software development problem which requires careful analysis and implementation, especially within existing systems. Although PCI is a common problem there has never been a clearly defined way in which to address it, due mainly to the diversity of such applications. Aspect Orientated Programming was applied to meet the requirement of masking the PAN. AOP is often regarded as a new technology, with many developers unsure how best to apply it within projects. It has been effectively used for auditing events in web applications, but its ability to broadly address crosscutting concerns made it ideal for masking the PAN. The architecture allowed a definite amount of code to be added which intercepted all the methods specified in the aspect, regardless of future additions to the system. This reduces the amount of work required to the maintain aspect, always an important consideration when developing software. The implementation of AOP to meet PCI Requirements within the system was not a clear-cut exercise and raised many design considerations. Two chief considerations were allowing the end user to configure the fields to be masked and the use of specific regular expressions to mask digits. Both considerations were analysed and upon discussion with industry experts, it was reasoned that their inclusion would reduce the systems security and undo the effectiveness of AOP by trying to mask specific fields.

Despite the effort to strive for a totally secure system, it has to be acknowledged that this is not a possibility. Despite the application of all possible security controls, these can easily be circumvented by one dishonest employee in a position of power. This is a good lesson to take from the implementation of security features as the amount of time spent implementing the security does not necessarily reflect the actual increase in security. The decrease in performance, although expected was an interesting result within the testing. As with all code changes there is always going to be trade-offs. Although disappointing, when the positives were considered this was brought into perspective. The implementation of the requirements added an additional 600 lines of code. Considering that these lines cover the requirements, regardless of how much the system was to grow, it can be regarded as a success in terms of code bloat and maintainability. Software artefacts can differ radically meaning that each system must be taken on its merits. PCI is not something that can be taken lightly as outlined in the possible fines that

can be incurred. Based on this and the effort to implement PCI requirements into a simple application in this research, enterprises involved should be aware of the enormity of the task and not undertake it lightly. We believe that the concepts here will provide an insight into how to approach the PCI requirements for others seeking to undertake the task. The software artefact should also serve as a guide to developers attempting to implement new applications, where security and design are fundamental elements that should be considered through each phase of the software development lifecycle and not as an afterthought.

References

[1] FBI. *Federal Bureau of Investigation Internet Crime Report.* 2009; http://www.ic3.gov/media/annualreport/2009_IC3Report.pdf

[2] Docksey R. *PCI DSS - Closing the Loop on `Card Not Present' Fraud*. The Institution of Engineering and Technology Conference on Crime and Security. London. 2006: 27-37.

[3] Liu J, Xiao Y, Chen H, Dodle S, Singh V. A Survey of Payment Card Industy Data Standard. *IEEE Communications Surveys & Tutorials*. 2010; 12(3): 287-303.

[4] Rowlingson R, Winsborrow R. A comparison of the Payment Card Industry data security standard with ISO17799. *Computer Fraud & Security*. 2006; (16)3: 16-19.

[5] Berinato S. Data Breach Notification Laws, State by State. CSO Online, February 12th 2008 http://www.csoonline.com/article/221322/cso-disclosure-series-data-breach-notification-laws-state-by-state

[6] Fabry HW. *Database Vault: Enforcing Separation of Duties to Meet Regulatory Compliance Requirements*. 12th International IEEE Enterprise Distributed Object Computing Conference. Munich. 2008: xxi.

[7] Blackwell C. *The Management of Online Credit Card Data using the Payment Card Industry Data Security Standard*. 3rd International conference on Digital Information Management (ICDIM 2008). London. 2008: 22-30

[8] McMillan R. Hotels.com Customer Data Stolen. *PCWorld Magazine*. June 3rd 2006. http://www.pcworld.com/article/125962/hotelscom_customer_data_stolen.html

[9] PCI DSS. 'Requirements and Security Assessment Procedures' PCI Security Standards Council. Wakefield, MA. USA. 2008.

[10] Campara D, Mansourov N. *How to tackle security issues in large existing/legacy systems while maintaining development priorities*. IEEE Conference Technologies for Homeland Security. Georgia. 2008: 167-172.

[11] Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Boston: Addison-Wesley. 1998

[12] Zhang C, Jacobsen H. *Aspectizing Middleware Platforms*. University of Toronto Technical Report. CSRG-466. January 2003

[13] Henning M. The Rise and Fall of CORBA. *ACM Queue Journal*. 2006; 5(4): 28-34.

[14] Steed H. *Encapsulating Legacy Software for Use in Client/Server Systems*. Proceedings of the Third Working Conference on Reverse Engineering. Monterey, CA. 1996: 104:119.

[15] Nordbotten N. XML and Web Services Security Standards. *IEEE Communications & Surveys Tutorial*. 2009; 11(3): 4-21.

[16] Laddad R. Aspect-orientated programming will improve quality. IEEE Software. 2003; 20(6): 20-21.

[17] Miller S. Aspect-orientated programming takes aim at software complexity. *IEEE Transaction on Computer*. 2001; 32(4): 18-21.

[18] Morse E, Raval V. PCI DSS: Payment card industry data security standards in context. *Computer Law & Security Report*. 2008; 24(6): 540-554.

[19] Mattsson U. *A Database Encryption Solution That Is Protecting Against External and Internal Threats and Meeting Regulatory Requirements*. 2004. http://www.net-security.org/article.php?id=715

[20] NIST (2001) Announcing the Advanced Encryption Standard (AES), http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[21] Coyler A, Clement A. Aspect-orientated programming with AspectJ. *IBM Systems Journal*. 2005. 44(2): 301-308.