Payment Card Industry (PCI)

Terminal Software Security

Best Practices

Version 1.0

December 2014

Document Changes

Date	Version	Description
June 2014	Draft	Initial
July 23, 2014	Core	Redesign for core and other
August 2014		Added comments from 06 August meeting
August 2014		Added comments from 20 August meeting
September 2014	Additional Task Force Input	Proposal for a different document structure
September 2014	Final Draft	Final draft from face-to-face meeting
December 2014	Initial Release	

Table of Contents

Introduction

Purpose of This Document

The PCI Terminal Software Security Best Practices (TSSBP) document gives detailed guidance on the development of any software designed to run on PCI PTS POI approved devices. The PTS POI approval covers the device "firmware," as defined in the PTS standard. However, other software on the POI not defined as firmware can still have an effect on the security of the device and must therefore be developed and maintained with that in mind. The goal of this document is to ensure that all organizations responsible for software development (and device management) understand the potential threats, and employ appropriate processes throughout the development life cycle to counter those threats. The processes followed will depend on the organization, the type of application being developed, and the software languages used, but the principles remain the same.

Note that this document covers all software which runs on the device not covered in the PTS POI program, including but not limited to:

- Payment applications
- Non-payment applications
- EMV kernels and other libraries
- Third-party (e.g. open-source) software

Relationship between other PCI standards

The PTS POI program is primarily concerned with device characteristics impacting the security of the POI device used by the cardholder during a financial transaction. The requirements also include device management up to the point of initial key loading, but the evaluation process only addresses device characteristics.

Logical and physical interfaces of the POI device are assessed for providing protection from being influenced by logical anomalies, but the applications (EMV kernel, payment and nonpayment applications) are not assessed under PTS POI or the PA-DSS standard unless the solution is being assessed under the PCI P2PE program. The Terminal Software Security Best Practices are focused on all terminal applications, to ensure that industry standard secure coding practices are followed and prevent compromise of merchant POI devices.

All applications that store, process, or transmit cardholder data are in scope for an entity's PCI DSS assessment, including applications that reside within an approved PTS POI device. The PCI DSS assessment should verify the software is properly configured and securely implemented per PCI DSS requirements. If the software has undergone any customization, a more in-depth review will be required during the PCI DSS assessment, as the application may no longer be representative of the version that was previously validated.

Secure payment applications, when implemented in a PCI DSS-compliant environment, will minimize the potential for security breaches leading to compromises of primary account number (PAN), full track data, card verification codes and values (CAV2, CID, CVC2, CVV2), PINs and PIN blocks, and the damaging fraud resulting from these breaches.

Examples of Potential Threats

It is not the intention of this document to maintain a detailed list of current threats, as this information is platform-specific and available from the Internet. However, some background knowledge on the general types of threats that need to be considered will be helpful when reading the rest of the document. Two examples are command injection and buffer overflow.

Command injection is where a command interpreter (such as a Linux Shell or SQL interpreter) is passed some text during normal operation, but where that text is not adequately checked beforehand. The calling application passes the text to the interpreter believing it to be a filename, for example, but it in fact contains special characters—which means that the interpreter doesn't treat it as a filename, it treats it as another command to execute.

Buffer overflow occurs when the software gets more input data than it would under normal circumstances, and the buffer is too small to store all that data. The software should check this and handle the error in a defined way, but poorly written software may just continue writing data beyond the end of the buffer and overwrite other items stored in neighboring memory. With careful consideration of what data is written beyond the end of the buffer, an attacker can change the execution of the program.

Secure Software-Development Life Cycle

The key to addressing potential threats is a robust, secure software-development life cycle. All software-development organizations should:

- Understand the current threats and how they impact the software.
- Develop and maintain secure coding standards and other processes to address those threats.
- Conduct reviews (including source-code reviews) to ensure the defined processes are being followed.
- Test the software components and terminal as a whole to give additional confidence that nothing has been overlooked.

Role Definition within TSSBP

This document makes reference to certain roles within an organization as related to software security. It is not the intention to define job titles but simply to define the sort of person who is envisaged to perform these roles. These roles should be clearly assigned to individuals within the organization:

Software Developer – The person writing the software. This person should be skilled in writing software in the appropriate programming language, should be familiar with and follow the company procedures, and should be provided with good security awareness training.

Security Champion – The technical person with primary responsibility for software security. This person should keep up to date with all threats that could affect the software written by the organization, and should ensure that secure coding standards are maintained and being used.

Peer Reviewer – The person responsible for peer review. This person is typically another software developer who didn't write the software being reviewed. They should be technically capable (with enough experience and training) of performing the review.

Release Authority – The person responsible for approving the final software release. They should have the skills, training, and authority to ensure that all of the secure software development processes leading up to the formal software release have been followed.

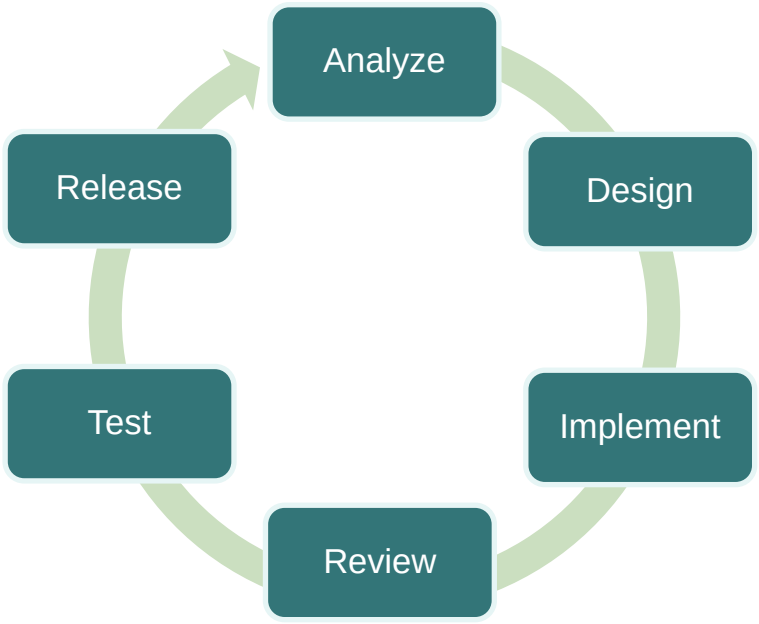
A: Security-Awareness Training

The first step in maintaining secure software is to fully understand all of the latest threats applicable to the system in question, and what techniques can be deployed to counter them. The "training" referred to in this section can be achieved through a combination of external training, internal knowledge exchange, and research (especially on the Internet).

Terminal Software Security Best Practices	Guidance
A1 Appoint a Security Champion	The software-development team should have an assigned technical person responsible for security. The Security Champion will ensure that the team is focused on software security and that the defined security procedures are followed. The Security Champion ensures that they and the rest of the software-development team keep up to date on the latest threats.
A2 Security Champion training	The assigned security champion is trained to understand the roles of the security champion, is trained in the organization's secure coding process, and is trained on vulnerability assessment and emerging threats.
A3 Software Developer training	The software developer is trained on the organization's secure coding practices and understands the policies and procedures around secure code development.
A4 Auditable Training program	The company maintains an auditable training and research program that ensures all personnel are properly trained and that the policies and procedures are regularly reviewed and updated to remain current.

B: Secure Software-Development Process

Once the threats are understood, the organization should define, implement, and maintain secure software-development procedures that ensure these threats are mitigated. The diagram below is an example of a typical secure software-development process for reference. It is not the intention to mandate a particular software-development methodology; but whichever one is used should address each of the points in this document.



The policies and procedures maintained by the organization should address each of the points below. In addition, the organization should maintain auditable records to demonstrate that their procedures are being followed, and should independently audit those records to ensure that is the case.

Terminal Software Security Best Practices	Guidance
B1 All software should be documented.	Documentation should be maintained at a level where the operation of the software should be clear to any experienced software developer with no prior knowledge of the system.
B1.1 System architecture should be documented.	A software-architecture document should contain diagrams that describe the components within the system and how they interact. Reference should be made to API and UI documentation as required to detail the interfaces, etc.

Terminal Software Security Best Practices	Guidance
B1.2 All interfaces should be clearly documented.	API and UI documents clearly defining the interfaces should be produced, containing: • Clear definitions of all expected inputs, outputs, and error conditions; • All configuration options and default configurations; and • Update and remote-access parameters, if applicable.
B1.3 All sensitive data should be identified.	The documentation should identify: • All sensitive data (such as PANs), • Data flow through the program, • How data is handled in each function, • How data is eventually deleted, and • Details of any cryptography used,
B1.4 Security guidance should be provided.	Where the security of the system could be affected by incorrect configuration, installation, or maintenance, clear guidance documentation should be provided to highlight this.
B1.5 Define a security-testing plan.	Create a plan for the security testing required for the terminal as a whole. This plan should include but not be limited to port scanning, unintentional clear-text data, logging review, default passwords, test data, penetration testing, communications security, and the integrity of signed files.
B1.6 Define a software-versioning methodology as part of their system development lifecycle.	Changes to applications should be properly identified and enumerated. The version scheme should clearly specify how each of the various elements is used in the version number.
B2 All software should be well-structured and commented.	Software should be well-structured and have detailed, accurate comments such that the purpose and flow of each module and function is clear.
B3 A rigorous change management system should be used.	The organization should utilize a secure software change management system such that all development is traceable and access restricted. All changes should be reviewed and approved prior to release.

Terminal Software Security Best Practices	Guidance
B4 Stringent secure coding standards should be written, maintained, and used.	The organization should develop and maintain secure coding standards. These should be specific to the software being developed and the languages used to write it. They should draw on other generally recognized coding standards (e.g., Ref1), and should address all of the security threats identified during security awareness training. Some general topics are covered here with examples for clarification, but the standards should cover all applicable threats.
B4.1 Use of known unsafe functions should be banned.	Certain standard functions are known to be unsafe, and their use should be banned. A list of all such functions should be created, and safer alternatives should be mandated. Two such examples in the C language are system(), which allows command strings to be executed by the system, and strcpy(), which has no length-checking on the output buffer.
B4.2 Boundary checks should be performed on all buffers.	Every function handling buffers should accept parameters to indicate the size of the buffer and should perform boundary checks on those buffers during use.
B4.3 Use of signed and unsigned data types should be used appropriately.	In particular, variables used to hold the size of buffers should be unsigned. Integer operations should be checked for overflows/underflows.
B4.4 All inputs into the system should be strictly validated.	User input and other input from outside of the system should be strictly checked against whitelists to ensure any unexpected characters or parameters are rejected.
B4.5 Any strings sent to command interpreters must be strictly checked.	Any text sent to a command interpreter or passed as a parameter to a command must be strictly validated. Unexpected characters within the text can be used to supply additional, unexpected commands or parameters.
B4.6 Return values should always be checked.	Any error conditions should be handled correctly and propagated through the calling functions such that the system fails securely with no unexpected operation.
B4.7 Race conditions should be avoided.	An example is a “time-of-check, time-of-use” race condition where a file is checked at one point and used just after, with the assumption that the previous check is still valid. This assumption may not be correct if the system allows the file to be modified in between.

Terminal Software Security Best Practices	Guidance
B4.8 Exploit mitigation techniques should be used where possible.	Once an attacker has found a weakness, they still have to exploit it. Many systems support the use of exploit-mitigation techniques, which make the exploitation phase much harder, or even impossible. These should only be seen as a second line of defense, however. Examples include (but are not limited to): stack canaries, heap cookies, Address Space Layout Randomization (ASLR), and Data Execution Prevention (DEP).
B4.9 Compiler warnings should be set to high.	Many compilers have the option to perform code-checking at compile time. The warning level should be set as high as possible to warn about every possible issue, and then the software should be developed to remove all warnings.
B4.10 Static analysis tools should be used where possible.	Static analysis tools should also be used as part of the normal review process to help spot potential security issues.
B4.11 Reference should be made to the relevant security guidance provided by the terminal vendor.	PTS POI terminal vendors provide security guidance for the correct use of the terminal. This guidance must be strictly adhered to when writing software that runs on the terminal.
B5 Stringent secure coding standards specific to payments (e.g., payment applications, EMV kernels, etc.) should be written, maintained, and used.	Any applications with access to sensitive cardholder data, or with the ability to display prompts on a PIN entry device, need specific guidelines in place to handle this. These guidelines can be included within the organization's general coding standards or maintained separately. Some general topics are covered here, but the standards should cover all security issues applicable to the system.
B5.1 Prompts should be strictly controlled.	If the application has delegated responsibility for prompts, it must ensure that it has full control of what is on the display whenever numeric keyboard entry is enabled. If a prompt file is used it must be signed, and it must not be possible to modify the prompt between the checking of the signature and the display of the prompt. The application must never prompt for a PIN.
B5.2 Test data and accounts should be removed before release to customer.	All test data and developer credentials should be removed prior to releasing and signing of the application for production.
B5.3 Back-out or product de-installation procedures should be developed.	For each change, there should be back-out procedures in case the change fails or adversely affects the security of the application, to allow the application to be restored to its previous state.

Terminal Software Security Best Practices	Guidance
B6. The application should support and enforce the use of unique user rights that separate administrative functions from operator functions. The use of hardcoded passwords is prohibited.	Administrative, user, and operator rights should be separated to prevent unintended changes to the payment application settings or accidental disclosure of the administrative password. Hardcoded passwords should never be used.
B6.1 The application should enforce the changing of all default passwords for all accounts managed by the application and should force a password change after installation and prior to activation for use. This applies to all accounts, including user accounts, administrative, and operator vice accounts, as well as accounts used by the vendor for support purposes. The passwords should be a minimum of seven characters.	The default passwords should force the user to enter a new password prior to use in a production environment. The minimum password length should be seven characters.
B7 If the application or third-party application uses external libraries, gems, or other open-source resources, they should be obtained from a reliable source.	Open-source libraries and functions should be checked to ensure proper coding practices are followed or that the information sourced has not been tampered with.
B7.1 If using open-source resources, all unnecessary functions and resources should be disabled or removed and only documented functions and security processes be allowed to execute.	All unnecessary functions and services inherited from the open-source resource should be removed or disabled to prevent their misuse.
B8 All software modifications (new, changed, additions, etc.) should be tested in detail.	Low-level unit testing of the software should be performed and recorded. This may require writing specific test software or modifying calling functions to specifically cover boundary checking and error handling.
B9 All software modifications should be reviewed by a Peer Reviewer.	All software should be subject to a detailed code review to ensure the secure coding standards have been followed and no potential vulnerabilities exist. The reviewer should also check that the low-level module testing performed is sufficient.

Terminal Software Security Best Practices	Guidance
B10 All necessary files should be signed and their signatures verified before use,	All executable files should be signed. By default, all other files should also be signed unless there is clear, documented justification why a signature is not required (e.g., because the file cannot affect the security of the device).
B10.1 All signatures should be verified by the PTS POI approved firmware prior to use.	The firmware will have the ability to verify file signatures and will delete anything that fails verification. This feature should be used to verify all signatures.
B10.2 All executable files should be signed.	This applies to any files executed or interpreted on the system (either by the firmware or the application). If the application is interpreting commands from a file, the firmware will not enforce that check; therefore it is vital that the application does so.
B10.3 EMV Certification Authority Public Keys should be signed.	These represent the start of the chain of trust for authentication of smartcard transactions, and should be validated and signed prior to download.
B10.4 Software should be signed using a secure cryptographic device provided by the terminal vendor.	The firmware will have the ability to verify file signatures, and the terminal vendor will provide a signing tool that utilizes a secure cryptographic device to generate those signatures. This signing tool should be used.
B10.5 The signing process should be performed under dual control.	The signing tool will have the ability to be used under dual control. It should be managed such that no single person is able to sign files. If two secrets (passwords or PINs) are required for operation of the tool, no single person should know both secrets.
B10.6 All source code should be reviewed against the organization's coding standards prior to signature.	The principles of dual control should extend to code review. All software should be reviewed before it is signed.
B11 The features provided by the firmware should be used if applicable.	The application does not provide methods for the execution of un-authenticated functions (e.g., it should not provide its own VM, IP stack, scripting language, etc.). The PTS POI device features of the firmware should be used.
B11.1 The application should use a PTS approved random number generator for the generation of any random values required for the secure operation of the applications (EMV UN, protocol nonces, etc.).	Use the random number generator provided by the PTS POI approved device firmware.

Terminal Software Security Best Practices	Guidance
B12 Patches and updates are delivered to customers in a secure manner with a known chain of trust.	Software patches must be distributed in a manner that prevents malicious individuals from intercepting the updates in transit, modifying them, and then redistributing. All download packages should be signed.

C: Device-Level Testing

In addition to the low-level unit testing covered in Section B, testing of the device as a whole should be performed before the software is released. This should be performed on the relevant PTS POI approved device running the relevant PTS POI approved firmware.

Terminal Security Security Best Practices	Guidance
C1 Test scripts should be developed, reviewed, and maintained to ensure they represent the production environment.	Test scripts should be developed, reviewed, and maintained using peer review to ensure that they are complete and provide for both positive and negative testing.
C2 Any application change should result in regression testing for that application.	For any application change, it is important to test the software and not just the changed components to ensure there haven't been any unintended consequences. Appropriate regression test scripts should be followed, and results should be documented.
C3 The application should be tested with the complete solution in as close to a production environment as possible.	The application should be loaded into a functioning device and tested in an environment that represents a production environment to ensure no unintended errors or issue is present. Test scripts should be followed, and results should be documented.
C3.1 The application should undergo a series of negative testing to ensure errors are handled correctly.	The application should be loaded into a functioning device, tested in an environment that represents a production environment, and subjected to common issues that would occur in the normal operation of the device to ensure errors are handled correctly. Test scripts should be followed, and results should be documented.
C4 The application should be tested in accordance with the defined security plan as specified in B1.5.	The application should be loaded into a functioning device and tested in accordance with the defined security plan.

D: Internal Process Reviews

In addition to the detailed line-by-line code reviews covered in Section B, it is important for the organization to conduct higher-level reviews to ensure the procedures in place are being followed, and that those procedures themselves are still valid and sufficient.

Terminal Software Security Best Practices	Guidance
D1 Assign internal review personnel.	Define and document individuals responsible for conducting higher-level reviews and specifying their frequency.
D2 A release review should be performed by the Release Authority prior to any software release.	As a final stage before software is released, checks should be in place to ensure all of the required processes have been completed.
D2.1 Are all code reviews complete?	Is there evidence to show that all software modifications have been reviewed by the correct people, and that no code changes have been slipped in without review?
D2.2 Is all testing complete?	Is there evidence that all testing is complete?
D2.3 Has all documentation been updated?	Has all relevant documentation been updated?
D2.4 Is the software release note correct?	Software releases should be accompanied by a release note. This review should check that the details are correct and sufficient.
D3 Conduct a training review.	Are the Security Champion and Software Developer's staff properly trained and keeping up to date with the latest threats?
D4 Published attacks against any third-party (e.g., open-source) software should be frequently reviewed.	Is there evidence to show that the versions of all third-party software are recorded and published attacks are monitored and addressed?
D5 The organization's secure coding standards should be regularly reviewed against the latest threats identified during ongoing security awareness training.	Is there evidence to show that the latest threats are understood and secure coding standards updated accordingly?
D6 Legacy applications should be code-reviewed to the new coding standard.	Where secure coding standards have been updated, the changes should be applied to existing software. A timescale for completing this work should be agreed, and progress against this should be monitored. Where it is not practical to modify parts of the existing software to the new standards, a full review should be performed to ensure no vulnerabilities exist. Any vulnerabilities found must be fixed immediately.
D7 Are file-signing procedures being followed?	Is the documented process for file signing under dual control being followed?

Useful references

- *CERT Secure Coding Guidelines* (https://www.securecoding.cert.org)
- *PCI Payment Application Data Security Standard* (PA-DSS) (www.pcisecuritystandards.org)
- *PCI PIN Transaction Security Point of Interaction Standard* (PTS POI) (www.pcisecuritystandards.org)
- *PCI Point-to-Point Encryption Standard* (P2PE) (www.pcisecuritystandards.org)