How to achieve PCI DSS Compliance with Checkmarx Source Code Analysis

Document Scope

This document aims to assist organizations comply with PCI DSS 3 when it comes to Application Security best practices. The document concentrates on the specific items within the PCI document in a generic manner and mainly concentrates on helping understand the requirements. The second part of the document addresses Checkmarx specifically and provides further details about how Checkmarx assists in complying with the PCI DSS regulation.

The Payment Card Industry Data Security Standard (PCI DSS) is a self-regulated industry standard set by credit card merchants such as MasterCard and Visa for securing credit card information. Companies that suffer from a breach and are found to have failed compliance are heavily penalized, and in extreme cases, even barred from working with certain payment card brands.

The latest released PCI DSS regulation (v3) provides best practices and methodologies to comply with PCI on an ongoing basis. As a mandate, PCI covers various layers of protection and lists out 12 various requirements that companies need to comply with, where each requirement is detailed and prescriptive. Application-level security is discussed throughout the different requirements, and is mainly detailed in requirement 6.

How can you implement PCI DSS requirements during your in-house secure application development process? In this article, we look at the different relevant application security requirements (and sub-requirements) and the issues to consider during development.

We'd like to note that this guideline does not come to replace a PCI DSS audit process or the official standards. The point of this article is to add insights and best practices based on our experience working with hundreds of organizations who have implemented PCI in their development environment. We advise anyone implementing or validating PCI compliance to consult with a qualified assessor.

Requirement 3: Protect stored cardholder data

Protection methods such as encryption, truncation, masking and hashing are critical components of cardholder data protection. If an intruder circumvents other security controls and gains access to encrypted data, without the proper cryptographic keys, the data is unreadable and unusable to that person. Other effective methods of protecting stored data should be considered as potential risk

mitigation opportunities. For example, methods for minimizing risk include not storing cardholder data unless absolutely necessary, truncating cardholder data if full Primary Account Number (PAN) is not needed, and not sending unprotected PANs using end-user messaging technologies, such as e-mail and instant messaging.

This requirement lays out the storage guidelines for sensitive credit card information. In particular subsection 3.4 specifies that PAN data must be rendered unreadable. This can already be performed during the application development stage by ensuring that sensitive data inserted into the database fits the set of requirements such as encryption. To implement, create different tests on your code which act as verifiers to the structure of the inputted data.

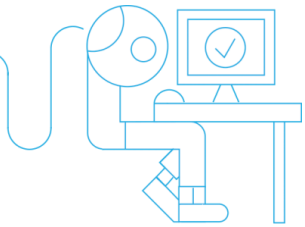
Requirement 6: Develop and Maintain Secure Systems and Applications

Unscrupulous individuals use security vulnerabilities to gain privileged access to systems. Many of these vulnerabilities are fixed by vendor-provided security patches, which must be installed by the entities that manage the systems. All critical systems must have all appropriate software patches to protect against exploitation and compromise of cardholder data by malicious individuals and malicious software.

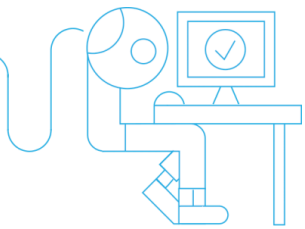
Note: Appropriate software patches are those patches that have been evaluated and tested sufficiently to determine that the patches do not conflict with existing security configurations. For in-house developed applications, numerous vulnerabilities can be avoided by using standard system development processes and secure coding techniques.

The additional note (above) that PCI DSS provides is very important. As breaches of cardholder data become more common, PCI recognized that most of these can be avoided by simple secure coding best practices. The good part is that requirement 6 really details out how a secure application development program should look like. We can easily adapt them to practice:

PCI DSS Requirements	Application Security: What to Look Out For
6.1**:** Establish a process to identify security vulnerabilities, using reputable outside sources for security vulnerability information, and assign a risk ranking (for example, as “high”, “medium”, “low”) to newly discovered security vulnerabilities.	The guidance for this subsection emphasizes the need for a process that “actively monitors industry sources for vulnerability information”. But what about risk ranking your own code – code that is affected by these new vulnerabilities, as well as in the unfortunate event that as a vendor you release the vulnerability? Our suggestion: instead of risk ranking your code only according to the vulnerability’s severity, calculate the risk to also include the vulnerability’s prevalence in the code. Why? If the vulnerability has, say, a medium-low severity but appears numerous times throughout the code, then the attack surface through exploitation of this particular vulnerability increases.



6.2: Ensure that all system components and software are protected from known vulnerabilities by installing applicable vendor-supplied security patches. Install critical security patches within one month of release.	Your development might be based on third-party code – whether through an API or even a Java framework. While reviewing your 3rd party-dependent code, pay particular attention that you are using those components that are up to date with their latest security fixes. What happens if the review reveals that you are not using numerous patches? In this case, follow the PCI DSS guidelines: use a risk-based approach to prioritize the updates.
6.3**:** Develop internal and external software applications (including web-based administrative access to applications) securely as follows: • In accordance with PCI DSS (for example, secure authentication and logging) • Based on industry standards and/or best practices. • Incorporating information security throughout the software development lifecycle.	Whether you are following a traditional Software Development Lifecycle (SDLC) process such as the waterfall model or modern environments such as Agile, there are different industry standards and/ or best practices to incorporate security within your SDLC program. Rather than running application security processes as a separate path to development, implementing the security process within the SDLC makes the analysis simpler, more effective and easier to address when the need appears.
6.3.1: Remove development, test and/ or custom application accounts, user IDs, and passwords before applications become active or are released to customers.	Continuously perform specific tests on your source code - customized to your environment - to check the existence of custom application accounts, user IDs and passwords. In particular, be extra vigilant during the final stages of product development for the appearance of this data. Prior to deployment, scan the application to validate that all the custom application accounts, user IDs and passwords are not hardcoded.



6.3.2: Review custom code prior to release to production or customers in order to identify any potential coding vulnerability (using either manual or automated processes) to include the following:

- Code changes are reviewed by individuals other than the originating code author, and by individuals knowledgeable about code-review techniques and secure coding practices.
- Code reviews ensure code is developed according to secure coding guidelines.
- Appropriate corrections are implemented prior to release.
- Code-review results are reviewed and approved by management prior to release.

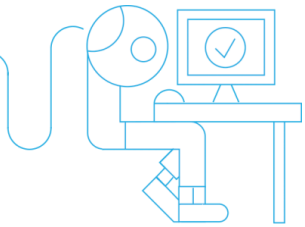
Coding vulnerabilities are categorized into two:

- Technical vulnerabilities
- Business logic vulnerabilities

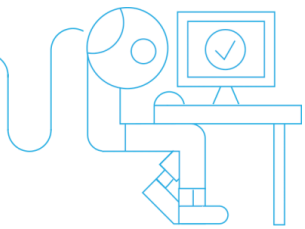
Business logic vulnerabilities are code functionalities which an attacker abuses to make the system work in a non-intended way. For example, a business logic vulnerability in a retail system may allow an attacker to input a negative value as a purchase price in order to receive funds back from the system.
PCI DSS mainly emphasizes the technical vulnerabilities and further expands on these in requirement 6.5. This is not to say that business logic vulnerabilities can be ignored. On the contrary – as they do not follow the checklist approach, you need to take extra care looking out for them.
To prevent BLAs, you should place tests customized to your code. Going back to the negative input example, this means validating that the specific function does not receive a negative value.

PCI DSS further provides testing procedures to follow for this requirement. While it is not our intention to duplicate the standard, we find it important enough to re-iterate PCI DSS's testing procedures for this requirement:

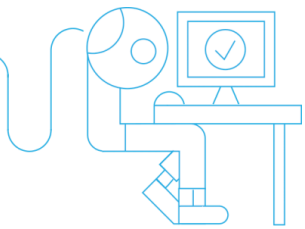
- **Examine written software-development procedures and interview responsible personnel to verify that all custom application code changes must be reviewed (using either manual or automated processes) as follows:**
  - **Code changes are reviewed by individuals other than the originating code author, and by individuals who are knowledgeable in code-review techniques and secure coding practices.**
  - **Code reviews ensure code is developed according to secure coding guidelines (see PCI DSS Requirement 6.5).**
  - **Appropriate corrections are implemented prior to release.**
  - **Code review results are reviewed and approved by management prior to release.**



6.4**:** Follow change control processes and procedures for all changes to system components. The processes must include the following:	In general, this requirement states that each change needs to be tracked – whether infrastructure or code modification. Security should become an integral part of the change control process so that every time a system component changes, security tests are performed.
6.4.1: Separate development/ test environments from production environments, and enforce the separation with access controls.	It is not enough to simply separate development and production environments. Separate security testing should be done to each of these environments.
6.4.3**:** Production data (live PANs) are not used for testing or development	This requirement can be implemented by scanning the testing or development code for the existence of PANs.
6.4.4: Removal of test data and accounts before production systems become active	Achieve this requirement by scanning the production code for test data and accounts prior to deployment.
6.4.5**:** Change control procedures for the implementation of security patches and software modifications must include the following:	While the below practices do not talk about tracking the progress of application development throughout time, there is a lot of benefit to doing that as well - verifying the product's security and compliance posture and developer's security awareness. Simply keep track of previous results and compare them to the new ones.
6.4.5.1: Documentation of impact.	Make documentation readable by producing them in the form of a dashboard or graphs for the security team or for team leaders, and with finer-grained details to help the developer pinpoint the impact of the vulnerability
6.4.5.2: Documented change approval by authorized parties.	Documentation should not be reviewed only by the developer. It should also be reviewed by the information security team and development leaders. Reviewing these results together can lead to a better security posture by recognizing those points that are problematic or repetitive code flaws.
6.4.5.3: Functionality testing to verify that the change does not adversely impact the security of the system.	As in general coding best practices – test that what you added not only does not break functionality but works as intended. The same goes for security: ensure that any code change does not introduce new vulnerabilities, adversely impacts the security posture, or breaks PCI compliance.
6.4.5.4: Back-out procedures.	Don't throw away just yet your old code. You must have a procedure to revert to when needed to fall back.



6.5: Address common coding vulnerabilities in software-development processes as follows: • Train developers in secure coding techniques, including how to avoid common coding vulnerabilities, and understanding how sensitive data is handled in memory. • Develop applications based on secure coding guidelines.	This section lays out particular vulnerabilities to train developers against and test against while developing code. As specified, training is best done by helping developers understand how sensitive data is handled in memory. In fact, we found that it is not enough to simply show how an app is exploited. Rather, by actually presenting the code flow (i.e. "behind the scenes" of the exploit) in a ***visualized*** manner aids the developer in becoming more security-conscious. For each vulnerability below we demonstrate its risk through its impact. We leave it to the reader to follow up on secure coding practices and how to fix certain vulnerabilities. It is also important to note that although this PCI DSS sub-requirement discusses technical generic vulnerabilities, it is essential to recognize that vulnerabilities exist within code flows based on the custom code of each organization. These "custom" vulnerabilities may also be prevented through a secure SDLC program.
6.5.1: Injection flaws, particularly SQL Injection. Also consider OS Command Injection, LDAP and XPath Injection flaws as well as other injection flaws	The impact? Allows insecure code to execute on the backend system. This can result in data theft, manipulation, corruption or the hosting of malware.
6.5.2: Buffer overflows	The impact? Allows insecure code to execute on the backend system. This can result in data theft, manipulation, corruption or the hosting of malware.
6.5.3: Insecure cryptographic storage	The impact? Allows an attacker to decipher encrypted data
6.5.4: Insecure communications	The impact? Allows an attacker to eavesdrop, and even intercept, communications.
6.5.5: Improper error handling	The impact? Leakage of information via error messages
6.5.6: All "high risk" vulnerabilities identified in the vulnerability identification process (as defined in PCI DSS Requirement 6.1).	The impact? Attacker can relatively easy penetrate the system for their nefarious manners. To recall, we recommended a risk-ranking calculated according to the vulnerability's severity and prevalence in the code. Do not release code into production if any "High" rankings exist.
6.5.7: Cross-Site Scripting (XSS)	The impact? Allows the running of a script on the client's machine in order to bypass access controls.



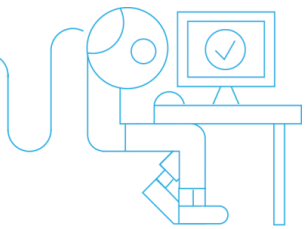
6.5.8: Improper access control (such as insecure direct object references, failure to restrict URL access, directory traversal, and failure to restrict user access to functions)	The impact? Allows the crawling of the Web pages, uploading of a potentially malicious file to the server and accessing restricted data.
6.5.9: Cross-site request forgery (CSRF)	The impact? Allows the attacker to perform an application-level transaction on behalf of the victim.
6.5.10: Broken authentication and session management	The impact? Allows the attacker to perform activities on behalf of a legitimate user.
6.6: For public-facing web applications, address new threats and vulnerabilities on an ongoing basis and ensure these applications are protected against known attacks by either of the following methods: ▪ Reviewing public-facing web applications via manual or automated application vulnerability security assessment tools or methods, at least annually and after any changes ▪ Installing an automated technical solution that detects and prevents web-based attacks (for example, a web- application firewall) in front of public-facing web applications, to continually check all traffic.	In general, a web-based technical solution that detects and prevents attacks only mitigates the risk of an attack until the code is fixed. A code review, on the other hand, actually fixes the issue. Ideally, you should perform both: review the code of your web application and use a technical detection and prevention solutions (such as a RASP). If you have trouble conforming to both, consider the advantages and disadvantages of each and how applicable each solution is to your environment. In a manual code review, typically the auditor reviews the code to ensure it stands to a certain level of security. Most enterprises, however, find automated code review through scanning a much quicker, more effective and cost-reducing process which also provides greater coverage. Whichever method you chose, ensure that the code review is part of the secure SDLC and integrates within the development process in order to provide quick fixes.

Requirement 11.3: Implement a methodology for penetration testing that includes the following… Defines application-layer penetration tests to include, at a minimum, the vulnerabilities listed in Requirement 6.5

Requirement 11.3 discusses penetration testing for infrastructure and application security. As the point of this article focuses on application security we refer to the application layer in this sub-requirement.

Penetration testing may be done manually or automatically, and is usually performed as a combination of both. Penetration testing may also be done in-house, by an individual (s) independent of the system, or by an external individual(s).

Regardless of which methodology and practice you choose, makes sure you receive the results reports



with actionable remediation items. Set aside a timeframe to analyze the report, scheduling the remediation and re-testing your environment.

How Checkmarx addresses the compliance requirements

6.1 Establish a process to identify security vulnerabilities

Checkmarx's CxSAST addresses thousands of vulnerabilities on the most prevalent coding languages. Detection is performed during the SDLC and is integrated as part of the development process. Vulnerabilities are all categorized and assigned with a severity level allowing the development and security teams to easily prioritize and remediate the most important vulnerabilities basing the decision not only on the severity but also on the abundance of a specific vulnerability.

6.2: Ensure that all system components and software are protected from known vulnerabilities by installing applicable vendor-supplied security patches.

Have your open source third parties scanned for vulnerabilities as part of your project. 3rd party code embedded in your product can expose your application to multiple vulnerabilities. Checkmarx allows manual query creation which may be used to detect specific 3rd parties and there version number.

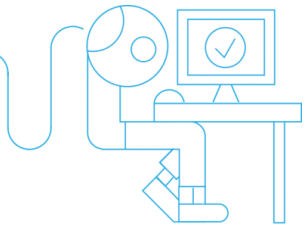
In addition Checkmarx recommends enforcing Static code analysis on all included 3rd parties within the application. In case code is not accessible or the third party is not willing to share the code, Checkmarx recommends asking the 3rd party to perform their own tests and deliver a full report of the results. This can be performed using Checkmarx's on demand service https://www.checkmarx.com/solutions-2/cxcloud-on-demand/ .

6.3: Develop internal and external software applications (including web-based administrative access to applications) securely as follows:

- Code changes are reviewed by individuals other than the originating code author, and by individuals knowledgeable about code-review techniques and secure coding practices.
- Code reviews ensure code is developed according to secure coding guidelines.
- Appropriate corrections are implemented prior to release.
- Code-review results are reviewed and approved by management prior to release.

Checkmarx allows code analysis at every stage of the SDLC ensuring developers, testing teams, security personnel and management are all in the loop of the application code's security status.

Automated Reports containing detailed information about current security status and trend information based on previous scans (graphical and detailed data) are generated and emailed to the relevant personnel who are required to review the results. Out-of-the-box integration of Checkmarx with Build servers, IDEs, Bug tracking tools and source repositories ensures the development teams follow secure coding guidelines and allow setting thresholds for approval of code release (no high risk vulnerabilities



for example).

6.4: Follow change control processes and procedures for all changes to system components. The processes must include the following:

Checkmarx's CxSAST was designed to enable maintaining the developed code secure at all times. Incremental scanning allows the system to analyze only code which has been modified and not yet approved. Incremental scanning ensures that whenever changes are performed on the code the system does not waste time on analyzing the complete code base.

6.4.1: Separate development/ test environments from production environments, and enforce the separation with access controls.

Both testing and production environments should and can be tested separately generating separate results.

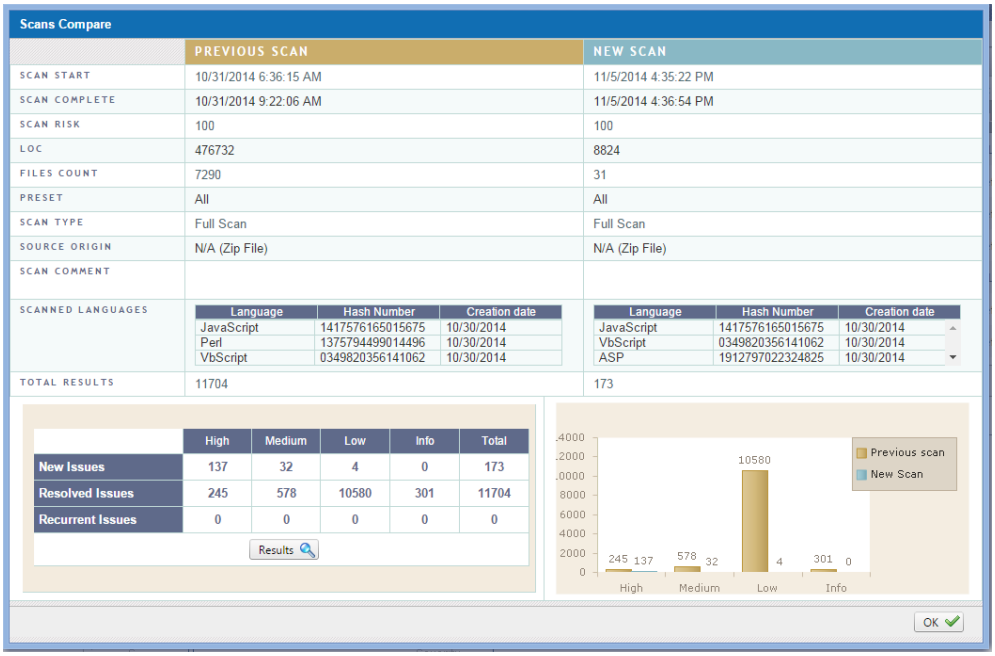
6.4.3: Production data (live PANs) are not used for testing or development

6.4.4: Removal of test data and accounts before production systems become active

Simply instruct Checkmarx CxSAST to scan for existing PANs within the code on either the testing, the development and the production environment. Define the query with the relevant severity level and ensure no PANs are present.

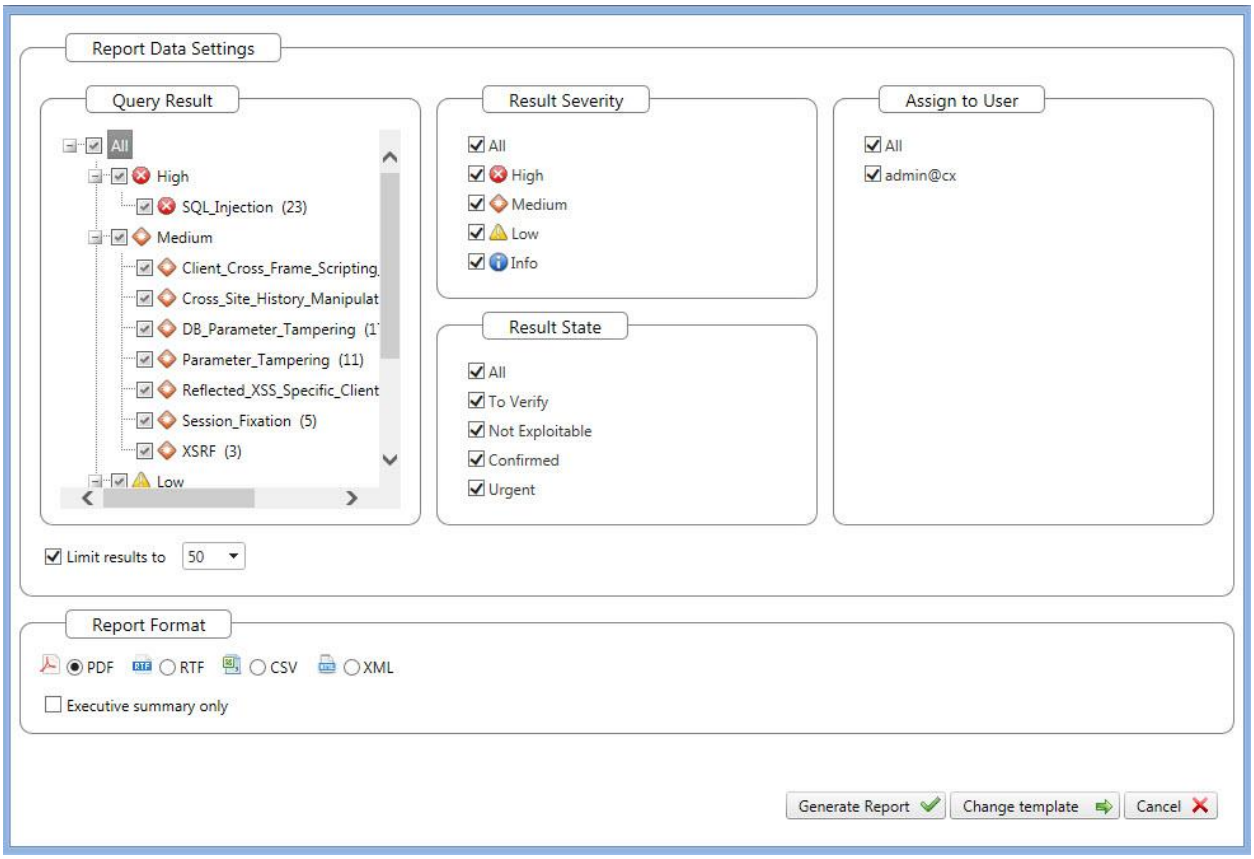
6.4.5: Change control procedures for the implementation of security patches and software modifications must include the following:

6.4.5.1: Documentation of impact – Use Checkmarx's reports when presenting impact and trends of multiple scans. Present Dashboard or graphs for the security team or for team leaders, and with finer-grained details to help the developer pinpoint the impact of the vulnerability.



6.4.5.2: Documented change approval by authorized parties – Checkmarx allows automated

sharing of analysis reports in multiple formats. Each scan can automatically generate a report to a pre-defined set of users. The reports can contain an executive summary or the full data and can be presented in multiple formats.



6.4.5.3: Functionality testing to verify that the change does not adversely impact the security of the system – Checkmarx provides the user the ability to perform incremental scanning. This functionality is unique to Checkmarx and allows the user to re-scan only the parts of the code which were modified. This functionality reduces scanning time dramatically and ensures that changes do not introduce new issues.

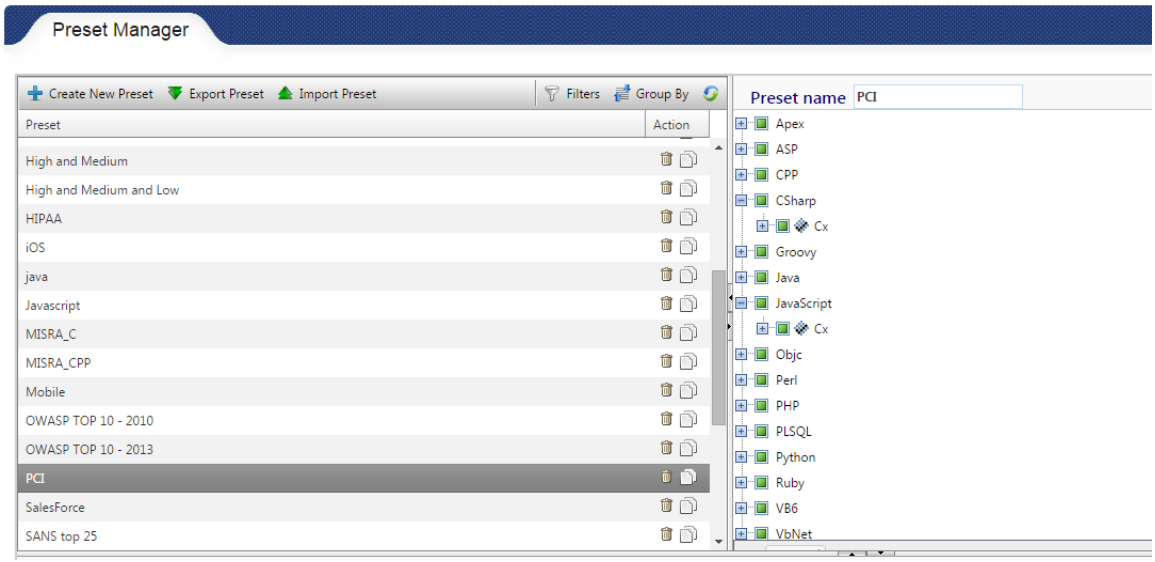
6.4.5.4: Back-out procedures. – Checkmarx stores previously scanned code making sure it is accessible in case changes need to be reverted. Checkmarx CxSAST also allows comparison of current and previous scan results.

6.5: Address common coding vulnerabilities in software-development processes as follows:

- Train developers in secure coding techniques, including how to avoid common coding vulnerabilities, and understanding how sensitive data is handled in memory.
- Develop applications based on secure coding guidelines.

Checkmarx addresses secure coding practice and education as part of its product suite. As part of CxSAST's code analysis presets we provide coverage for well-known regulations and for specific platform

requirements such as Mobile (Android or iOS). In addition Checkmarx's Game of Hacks is an interactive game developed by Checkmarx to raise security awareness in the InfoSec community and help developers boost their secure coding skills. Besides being an interactive security resource that has been played by over 80,000 players, Game of Hacks has also been developed as an educational platform which organizations implement within their environments to educate developers and share knowledge across the company. (www.gameofhacks.com)



6.6: For public-facing web applications, address new threats and vulnerabilities on an ongoing basis and ensure these applications are protected against known attacks by either of the following methods:

- Reviewing public-facing web applications via manual or automated application vulnerability security assessment tools or methods, at least annually and after any changes
- Installing an automated technical solution that detects and prevents web-based attacks (for example, a web- application firewall) in front of public-facing web applications, to continually check all traffic.

Checkmarx CxSAST fully integrates within the organizations SDLC allowing constant analysis of code whether its new or previously written code. Rather than performing annual checks you can be on top of your code's security at any given time. Integrate CxSAST within the SDLC to enforce version release only once the defined code security parameters have been enforced. Looking at the real time protection aspect, Checkmarx also delivers CxRASP to protect applications from existing vulnerabilities in the code which are trying to be abused. CxRASP will be instrumented as part of the protected application so that it can understand the data flow and ensure detection of relevant attacks only. Different from WAF solutions CxRASP ensures low to zero false positives due to its understanding of the application flow.