SecureSDLC

Security Testing Standards and Procedures

Rev 0.9

Revision History

Date	Section(s) Affected	Made By	Action Performed

Table of Contents

Revision History ii
Table of Contents iii
1 Purpose and Scope 1
2 Introduction 1
3 Good's Security Development Lifecycle (SDL) 1
3.1 Security Lifecycle Overview 1
3.2 Phase 1: Inception (Ideas) 2
3.3 Phase 2: Release Planning 3
3.4 Phase 3: Scrum 4
3.5 Phase 4: Validation 4
3.6 Phase 5: Release 5
3.7 Phase 6: Production 5
4 Melding Agile with Security 6
4.1 Security Education 6
4.2 Security Implications Review 7
4.3 Threat Modeling: The Cornerstone of the SDL 7
4.4 Secure Design Review (SDR) 7
4.5 Tooling and Automation 7
4.6 Penetration and Fuzz Testing 7
5 SDL Practices and Procedures 8
5.1 Security Implication Review 8
5.2 Threat Modeling 9
5.2.1 Modeling Stage *10*
5.2.1.1 Decompose the Application 10
5.2.1.2 Map the Data Flows 11
5.2.1.3 Review the Security Profile 13
5.2.2 Description Stage *13*
5.2.2.1 Complete a STRIDE Assessment 13
5.2.2.2 Document the Threats 14
5.2.3 Prioritization Stage *15*
5.2.3.1 Rate the Original Risks 15
5.2.3.2 Mitigate the Threats 16

5.2.3.3 Rate the Residual Risks 17
5.2.3.4 Optional Use of Microsoft's Threat Modeling Tool 17
5.2.4 Replace General Mitigation Actions with Concrete Security Requirements 18
5.2.5 Update the Threat Model 18
5.2.6 Prioritize Security Requirements Using DREAD Scores 18
5.3 Advise Product Owner – Jointly Agree on Priorities 19
5.4 Complete Secure Design of Prioritized User Stories 19
5.5 Modify User Story with Security Requirements and Design 20
5.6 Secure Design Review (SDR) 20
5.7 Secure Coding and Security Testing Practices 22
5.7.1 Multiple Independent Levels of Security (MILS) 22
5.7.2 Defensive Programming 22
5.7.3 Security Testing 23
5.7.3.1 Discovery 23
5.7.3.2 Vulnerability Scan 23
5.7.3.3 Vulnerability Assessment 23
5.7.3.4 Security Assessment 24
5.7.3.5 Penetration Test 24
5.7.3.6 Security Audit 24
5.7.3.7 Security Review 24
5.8 Proper Release Documentation and Secure Packaging 24
5.9 Incident Management 24
5.9.1 Containment 25
5.9.2 Eradication 25
5.9.3 Recovery 25
5.9.4 Post-Incident Analysis 26
5.10 End-of-Life (EOL) Security Policy 26
5.10.1 Sunsetting Criteria 26
5.10.2 Information Disposal and Media Sanitization 26
6 Software Security Principles 27
6.1 Reduce the Attack Surface 27
6.2 Secure by Design Not Afterthought 27
6.3 Insider Threats as the Weak Link 28
6.4 Assume the Network is Compromised 28
6.5 Secure by Default 29

6.6 Defense in Depth 29
6.7 Principles for Reducing Exposure 29
6.8 The Insecure Bootstrapping Principle 29
6.9 Input Validation 29
6.10 Security Ethics 30
7 High-Level Security Requirements 30
7.1 Confidentiality Requirements 30
7.2 Integrity Requirements 31
7.3 Availability Requirements 31
7.4 Authentication Requirements 32
7.5 Authorization Requirements 33
7.6 Auditing/Logging Requirements 34
7.7 Session Management Requirements 34
7.8 Errors and Exception Management Requirements 35
7.9 Configuration Parameters Management Requirements 35
7.10 Sequencing and Timing Requirements 35
7.11 Archiving Requirements 35
7.12 Internationalization Requirements 35
7.13 Deployment Environment Requirements 36
7.14 Third-Party Software Procurement Requirements 36
7.15 Antipiracy and Anti-tampering Requirements 36
8 Common Software Vulnerabilities and Controls 37
8.1 OWASP Top 10 37
8.2 CWE Top 25 38
8.3 OWASP Top 10 Mobile Risks 40
8.4 OWASP Top 10 Cloud Security Risks 40

1 Purpose and Scope

This document is designed to serve as an orientation and procedural reference guide for company personnel whose direct focus is on the technical aspects of secure software design and security assurance. Its expanded audience includes department and line management, product owners, architects, developers, integrators, test engineers, field support technicians, marketing and sales specialists, and others on a need-to-know basis. Its scope covers software security-related issues throughout Good's SDLC from inception through production.

2 Introduction

To be demonstrable and effective, security engineering practices must be documented, actionable, measurable, testable, traceable, related to identified business needs or opportunities, and defined to a level of detail sufficient to achieve a secure system design and implementation. A systematic approach to security requirements avoids generic lists of features and directly takes into account the attacker perspective.

Good's SDL is in place to help its software engineering teams build security into the early stages of product development in a structured, repeatable, and measurable way. It is based on extensive research and field work in which the system resources of many development life cycles were decomposed to create a comprehensive set of security requirements. These resulting requirements form the security basis of Good's Best Practices, which systematically address vulnerabilities that, if exploited, could result in the failure of basic security services (e.g., confidentiality, authentication, and authorization).

Section 3 of this document presents an overview of the security lifecycle followed by thumbnails of each phase. Sections 4 lays to rest any contention that Agile and a robust SDL are in any way incompatible. Section 5 sets forth the general SDL practices and their procedural aspects along with the intended results. Section 6 elaborates the company's core security principals and secure design fundamentals. Section 7 outlines the company's high-level security requirements for its software products. Section 8 lists the most common software vulnerabilities and controls for easy reference.

3 Good's Security Development Lifecycle (SDL)

Driving the company's overall software development life cycle, Good's SDL embodies a consistent process of safely creating and altering the company's products, systems, and applications, as well as the models and methodologies used to develop them. Delivering end-to-end security with every Good product is the paramount objective throughout.

3.1 Security Lifecycle Overview

Overall, the SDL encompasses the same six phases as the SDLC:

- Ideas (also called Inception)
- Release Planning
- Scrum
- Validation
- Release
- Production

Feature and performance (non-functional) requirements are typically initiated and gathered from outside of the company's engineering environment, resulting either from market intelligence or directly proposed/requested by customers in consultations with their Good account representative. With rare exception, security requirements are generated and defined internally by the company's security experts—architects, developers, IT security analysts, QA engineers, and field incident response specialists—who closely and continuously monitor the technology horizon for potential exploitables,

malware, and the latest industry-identified attack vectors. These requirements are then carefully defined for all Good products and incorporated by reference into Good's standing security policies.

Seen in Figure 1, during the **Ideas** (requirements gathering) phase of the SDLC and before passing beyond it, the security implications of a feature request or change are reviewed against the current body of security requirements—high-level, low-level, and those impacting Good's secure development infrastructure. High-level requirements govern protection mechanisms against broad potential vulnerabilities, whereas low-level requirements are geared to specific functionality and/or components. Security requirements that apply to the development infrastructure relate to intranet security, source control access, defect tracking and control tools, configuration management, QA test environments, and staging and production environments preparatory to deployment.

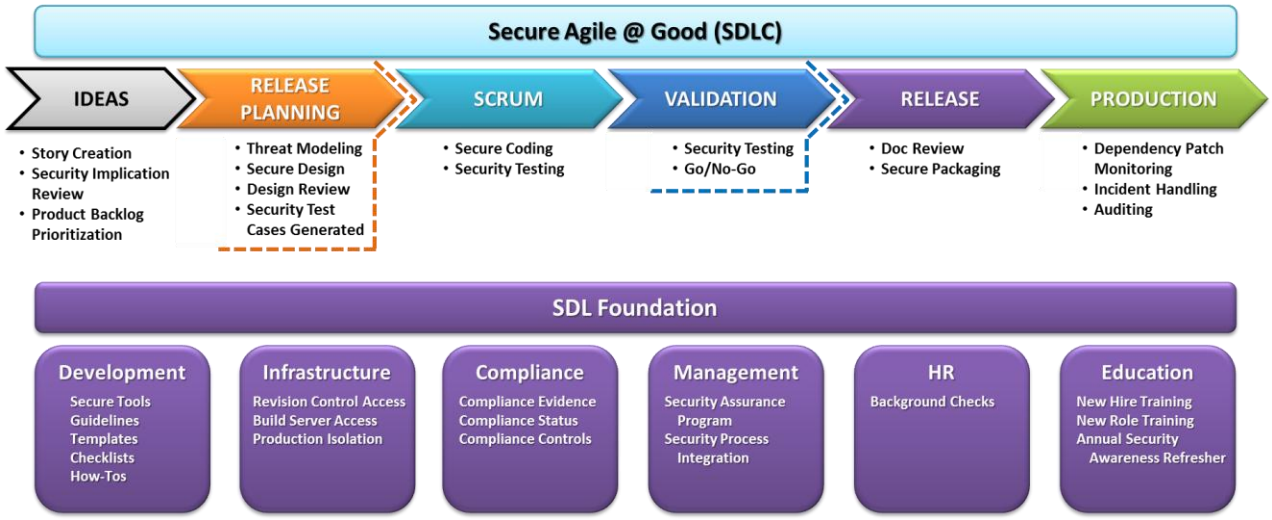


Figure 1: Good's Secure Software Development Life Cycle (SDLC)

During **Release Planning**, those user stories selected for development from the product backlog that have been flagged with security implications undergo a security design at the architecture level that initially satisfies all applicable standing security requirements. The modified architecture is then threat modeled to determine/isolate potential exploitables. Only upon formal approval after a secure design review (SDR) by security architects can the new story advance along with the appropriate security test cases to the **Scrum** phase for implementation.

An additional security "gate" (the dotted line boundaries in Figure 1) involving robust regression testing must be passed during **Validation** for the release candidate to gain a "Go" for **Release**, either as a Priority Access (PA) release to select customers or a General Availability (GA) release to the market at large. Upon successful staging, the software enters **Production**, where any dependency patches are monitored, incident handling is carefully tracked and analyzed, and security performance audits are routinely carried out.

3.2 Phase 1: Inception (Ideas)

As shown in Figure 1, Inception begins with the translation of customer requirements by the **product owner** into **user stories** that encapsulate the functional demands needing to be satisfied. User stories take the short form: "As a <*type of user*>, I want <*some goal*> so that <*some reason*>" or a variation thereof. Larger, more complex requirements are translated into broad user stories called "Epics" sufficient to frame the essence of the requirement in terms of intent and expected result.

Each user story is then run through a "Security Implication Checklist" to determine its potential vulnerabilities, if any. The checklist currently comprises 12 general questions.

1. Does the story involve authenticating user credentials or trusting host/component credentials or handling user/system credentials?
2. Does the story require adding/deleting/updating an authentication/authorization mechanism?
3. Does the story require opening/adding new interfaces? Examples include:
    a. New remotely accessible services
    b. Adding a new RPC interface or adding more procedures to existing ones such as XML GW
4. Does the story require security services provided by external entities like Kerberos, Radius, etc.?
5. Does the story involve building transport to send user data?
6. Does the story involve creating and updating security operations including any of the following: confidentiality, integrity, authentication, authorization, auditing, and/or any input/output sanitation operations?
7. Does the story involve detecting unauthorized OS-level modifications like jailbreak/rooted detection, altered binaries or configuration files, etc.?
8. Does the story involve any type of compliance or secure state check?
9. Does the story require adding/removing/updating any security related OS-level or third-party component?
10. Does the story require changes to the existing role model; e.g., creating/updating/deleting rights?
11. Does the story involve adding/updating/removing user/enterprise data from a device?
12. Does the story involve handling payment or personally-identifiable information (PII) data?

If the answer to any of the foregoing is affirmative, then this user story has security implications.

Next, based on the nature and extent of its security implication(s), the story is appropriately tagged or flagged in Rally and **prioritized** in the product backlog for threat modeling, security design, and secure design review.

3.3 Phase 2: Release Planning

The goal of release planning is to estimate roughly which features will be delivered by the release deadline. The initial plan rarely satisfies all parties: either not enough functionality will be delivered or it will take too long to deliver what's been requested and promised. At Good, the team and stakeholders are expected to look these hard truths in the face and plan around them. There are no scope miracles that will satisfy everyone, so the development team must use real metrics and negotiation to make hard choices as close to the start of the project as possible. Consequently, although the initial release plan is understood by everyone to be rough, it must be detailed enough to start development. And, among the most pressing details at this stage are identifying and mitigating any security vulnerabilities and weaknesses.

In fact, until its security requirements are satisfied in a high-level design, a user story cannot move forward to the next phase. A high-level design consists of UML diagrams and/or flow charts adequate to depict component relationships, dependencies, and interface boundaries. These are used to analyze the **data flow** (type and direction) crucial to the **threat modeling and secure design review** process outlined in Sections 4 below.

Illustrated in Figure 3 on page 8, threat modeling ascertains/isolates known security issues and then maps them to an appropriate mitigation action that eliminates or reduces the risk. These mitigation actions, further defined and prioritized in a residual risk matrix, are then ranked according to their impact on the desired functionality and the existing architecture—less impact, higher priority. If the level of priority is jointly accepted by the architects and the product owner, the user story, now paired with its security requirements, is used by core architects to design a secure solution. When the secure design is complete, it is formally reviewed by the architects for compliance with Good's security standards. Once SDR-approved, the design is made available for development according to its assigned priority in the

release backlog. The original user story can now be modified to reflect its new secure design and security requirement and proceed to the Scrum phase of the SDLC.

3.4 Phase 3: Scrum

Scrum is the iterative and incremental Agile software development framework adopted by Good. In Scrum, rather than documenting complete, detailed descriptions of how everything is to be done on the project, much is left up to the software development teams, called Scrum Teams. Each team is self-organizing and cross-functional. Hence, the Scrum phase of the SDLC makes progress in a series of timeboxed iterations called sprints, each lasting no more than a month. At the start of a sprint, team members commit to delivering some number of features in the release backlog—those stories slated for the current release that have been drawn from the larger product backlog. By the end of the sprint, these features are "done," which means they are coded, tested, and integrated into the evolving product. To conclude each sprint, a sprint review is conducted during which the team demonstrates the new functionality to the product owner and interested stakeholders, receiving feedback that could influence the next sprint.

From a security standpoint, the Scrum phase must ensure that secure coding techniques like defensive programming are used to minimize potential vulnerabilities. Consistent with Good's test-driven development paradigm, the security test cases generated from the threat model are run by special security testers—threat experts capable of simulating attacks using a variety of penetration techniques—to assure that the new feature/functionality is performing securely and up to the tolerances specified in the design. Security defects and shortcomings are managed in Good Defect Manager like all other bugs. As influential stakeholders, the security assurance team also takes part in Scrum ceremonies as appropriate. See *Secure Agile @ Good* for a more complete discussion of Good's Scrum process and ceremonies.

3.5 Phase 4: Validation

With its new features implemented, the software enters the Validation phase of the SDLC, which entails full regression testing to make sure the product as a whole is "ready-for-release." The four levels of validation testing, carried out for both the functional and non-functional (performance-scalability-reliability) attributes of the software, consist of:

- **Unit Testing** – also known as component testing, wherein the aim is to search for defects within a specific software unit while verifying the functioning of different software components like modules, objects, classes, etc., which can be tested separately.
- **Integration Testing** – to confirm the correct and intended interaction between the different interfaces of the components, as well as the interaction of the system with the OS, file system, hardware, and any other software systems with which it might interact.
- **System Testing** – in which the focus is to check the behavior of the whole system against what was specified in its design, rather than testing the individual internals.
- **Acceptance Testing** – to determine whether the product meets the established customer acceptance criteria. There are four types of acceptance testing: operational acceptance testing, compliance acceptance testing, Alpha testing, and Beta testing.

Defined in Section 7 and conducted concurrently with validation testing and an eye to penetrating all implemented defenses, **final security testing** covers:

- Confidentiality
- Integrity
- Availability
- Authentication

- Authorization
- Auditing
- Session Management
- Errors and Exceptions Management
- Configuration Parameters Management
- Sequencing and Timing.

Final security validation testing is conducted by the same security assurance team described in the Scrum phase. Constituting another phase gate in combination with the validation test results, the outcome of final security testing is a pass/fail determinant for making the **Go/No-Go** decision for the release. Insofar as the security aspect, to the extent that only risk-acceptable vulnerabilities remain, the software is ready to be released, pending appropriate written notification of those risks in the release documentation.

3.6 Phase 5: Release

The release phase of the SDLC spans several stages necessary for objective evaluation and to generate important feedback on performance and usability as well as to detect and eliminate defects still present in the software. Shown in Figure 2, these release stages or versions can include: Alpha, Beta, Release Candidate, Priority Access (PA), General Availability (GA), and the final Production or Live Release.

From both a security and a usability perspective, each stage requires the appropriate documentation necessary to install, configure, and operate the backend system, in addition to provisioning mobile devices for monitoring and control by the system.

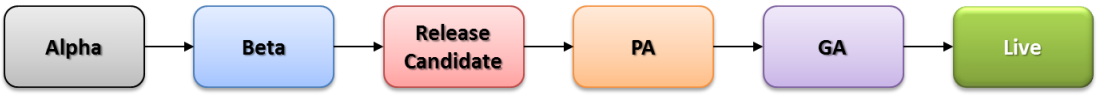


Figure 2: Release Stages

A description of new features and added/enhanced functionality must also be provided with the software, along with technical information concerning known performance and security limitations and the associated risks. Areas of the documentation involving security must be reviewed for accuracy and clarity by the security assurance team.

Although Good's well established safeguards and precautions with respect to the secure packaging of all software released externally are normally sufficient to provide tamper-proof and error-free download and unpacking on recognized and pre-approved customer computer systems and supported devices, a thorough check and re-examination of these packaging measures should be conducted by the security assurance team at each stage of release to ensure continued adequacy and protection against unauthorized interception or the insertion of malware.

The objective of each release and each stage thereof is to logically and demonstrably build on its predecessor in terms of quality and completeness, culminating in a final production release that is as secure and defect-free as possible under the given time and resource constraints.

3.7 Phase 6: Production

It is essential that all software in its production phase receive the necessary support to maintain and extend product performance within each customer's IT environment as agreed, ensuring that the customer has or is provided with:

- **Certified Version** – the latest version of the software certified for production use.
- **Maintenance Patches** – designed to fix reported issues and repair defects, including fixing security vulnerabilities and other bugs, and improving usability or

performance. These are distributed as binary executables that modify the current program executable.

- **Service Packs** – periodically released to fix defects, configuration flaws, and recently discovered performance issues; these are made available when the number of individual patches to a given major release reaches the designated limit for that product and are more easily managed and shipped as a package.
- **Hot Fixes** – urgent repairs or additional (re-)engineering of the program code escalated to an emergency condition based on a reported/recorded incident that demands an immediate and sustainable solution.
- **Online Support Forum and Knowledgebase** – a web-based forum for users to discuss issues and other product software-related topics with the benefit of input from Good's technical experts combined with access to a collection of online articles and information resources identifying common issues and how to resolve or work around them.

During the production phase, the security team provides continuous, proactive security support to all deployed systems sold/licensed and maintained by Good. In addition, all internal and external environments are monitored in relation to factors that can impact software security. In accordance with Good's Incident Management Plan, the main security objective here is to:

- Analyze and interpret events with a security impact
- Identify and prepare incident responses
- Monitor changes in environments; changes in security vulnerabilities, threats, and risks; and in their characteristics
- Continuously review software security behavior to identify necessary changes
- Verify that necessary changes have been properly implemented
- Perform periodic security audits.

All of the above extends to planned maintenance patches and emergency hot fixes consistent with the software security engineering practices employed during development.

4 Melding Agile with Security

Integrating the two is not as difficult as it may seem. In fact, one benefit of the SDL is that it is relatively artifact-free, meaning there is little documentation overhead with the notable exception of threat models, which are discussed presently. The mandatory tasks within the SDL-Agile framework include:

- Security Education
- User Story Security Implications Review
- Threat Modeling
- Secure Design Review (SDR)
- Tooling and Automation
- Penetration and Fuzz Testing

Each is briefly touched on next with the procedural mechanics outlined in Section 5.

4.1 Security Education

Every member of the technical staff must complete at least one security training course every year. If more than 20 percent of the teams are out of compliance with this non-negotiable requirement, the SDL will fail and delivered releases cannot be deemed securely designed and implemented.

Acquiring security knowledge can be as simple as reading appropriate chapters in an approved book or watching approved/recommended online training media. The areas of security knowledge that must be collectively covered by the teams include each of the requirements listed in Section 7: High-Level Security Requirements.

4.2 Security Implications Review

After a user story is created its security implications must be analyzed against the prevailing checklist issued by the security architects (see sample checklist on page 2). If there are no implications, the story can proceed directly to release planning and development. But if the story has security implications, it must be threat modeled and its functionality securely designed before proceeding to the Scrum phase. For details, see Section 5.

4.3 Threat Modeling: The Cornerstone of the SDL

Like many Agile processes, the threat model process is time-boxed and limited to only the parts of the product that currently exist or are in development. The threat model baseline in place at Good is a critical part of securing a product because it helps to:

- Determine potential security design issues
- Drive attack surface analysis and most “at-risk” components
- Drive the fuzz-testing process

After any modifications to the product, the threat model must be updated to represent any significant design changes, even if the functionality remains the same.

The threat modeling process is discussed in more detail in Section **Error! Reference source not found.**.

4.4 Secure Design Review (SDR)

After completing the threat modeling process, the security requirements identified are translated into security stories with the goal of generating a design that fulfills all of the security requirements while minimizing any impact on functionality. When the design is complete, it must be formally approved by Good’s architects. Pending the outcome of the SDR, the user story cannot move forward. If approved, it can now proceed beyond release planning and on to development (Scrum). If rejected, it is returned to the security architect to improve the design. Refer to Section 5.6 for more on SDR.

4.5 Tooling and Automation

The more you can automate the work necessary to meet requirements, the easier security becomes. However, where security is involved, tools are not a replacement for humans, but they do offer scalability. Nonetheless, simply running tools does not make a software product more secure.

Good’s current security-related tool chest includes:

- *{need to populate this appropriately}*

4.6 Penetration and Fuzz Testing

Good’s Pen Testers apply their skills to thwarting implemented countermeasures just like a crafty attacker on the outside by attempting to penetrate product security. They likewise will seek to insert malware or steal/damage data. Their job is a success only when they are unsuccessful.

Fuzz testing is an automated or semi-automated technique that provides invalid, unexpected, or random data as user inputs. The system under test is then monitored for crashes, failing built-in code assertions, and potential memory leaks.

The beauty of fuzz testing is that once a computer or group of computers is configured to fuzz the application, it can be left running, and only crashes need to be analyzed. If there are no crashes from the outset of fuzz testing, the fuzz test in probably inadequate, and a new task needs to be created to analyze why the fuzz tests are failing in order to make the necessary adjustments.

The threat model determines which portions of the application to fuzz test, as well as the order in which entry points are fuzzed. For example, remotely accessible or unauthenticated endpoints are higher risk that local-only or authenticated endpoints.

5 SDL Practices and Procedures

Summarized above, additional details on SDL requirements and practices are presented in this section under caveat that because the threat horizon is dynamic and evolving, so also must Good's security practices and procedures be. Ergo, this is a living document subject to change, so make sure you are reading the latest update before implementing the procedures set forth.

That said, during the critical early phases of the lifecycle—Inception and Release Planning—the process flow in Figure 3 governs all new features and functionality entering the development pipeline.

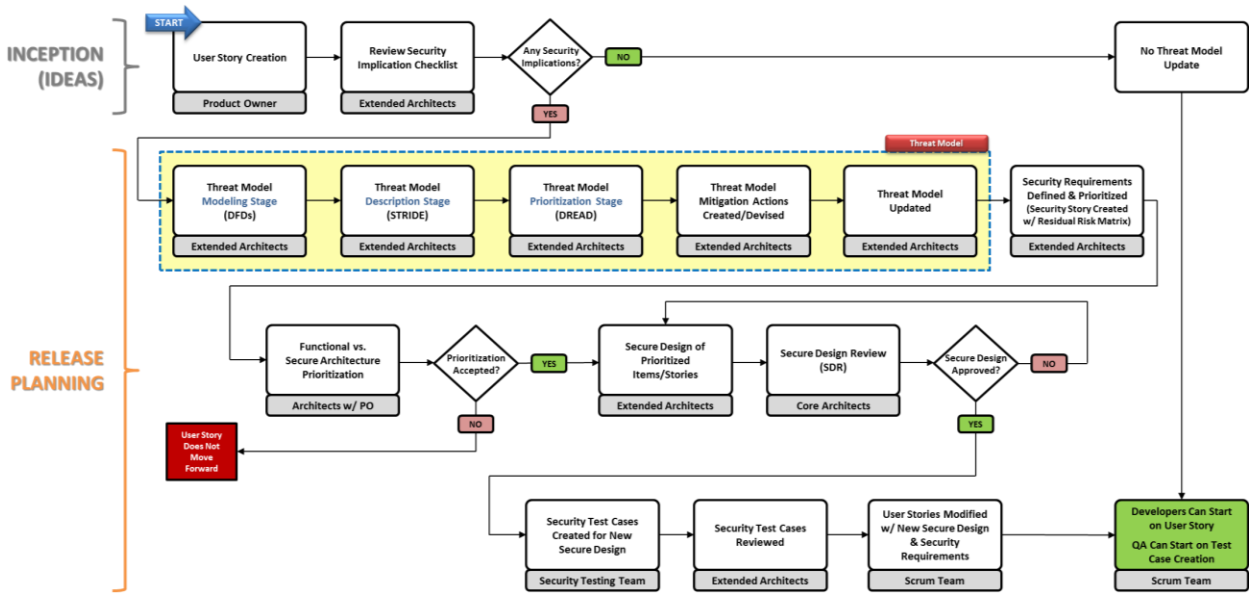


Figure 3: Early Phases of the SDL (Threat Modeling through Secure Design)

The goal is to identify security vulnerabilities and address potential threats with appropriate countermeasures in a logical order, starting with the threats which present the greatest risk or could result in the greatest damage. After which, appropriate threat elimination/mitigation options are selected or devised and updated in the threat model. Next, the business value of adding the new functionality is weighed against the impact of any residual risk to the existing architecture and ranked from high to low. Then, jointly with the product owner, the architects vote to accept or reject each tradeoff.

If rejected, the user story does not move forward.

When accepted, a secure design of the new feature which supports the agreed tradeoff, if any, is completed by the core architects. The design is then subjected to a secure design review (SDR). If the review board approves, the design is made available to the assigned developers and security test engineers. The former must modify the affected user stories in Rally based on the design's security requirements, while the latter generate security-specific test cases to be run against the coded implementation. Based on the revised story, functional QA can now create suitable test cases as the team moves forward with implementation.

The major elements of the process captured in Figure 3 are each discussed in turn below.

5.1 Security Implication Review

Depicted in Figure 4 below, after creating a new story, it must be assessed against the standing checklist for security implications described in Section 3.2. This is not only a critical milestone in the evolution of the story, it is crucial to the maintenance of the product's secure architecture, which makes the product owner ultimately accountable for seeing it done. POs possessing insufficient technical proficiency in software security should enlist members of the assigned scrum team who do to assess the implications,

but in all cases a user story with security implications must be flagged in Rally. Otherwise, the architecture team has no way to identify it as a candidate for threat modeling and secure design. The step-by-step sequence to follow is shown in Figure 4.

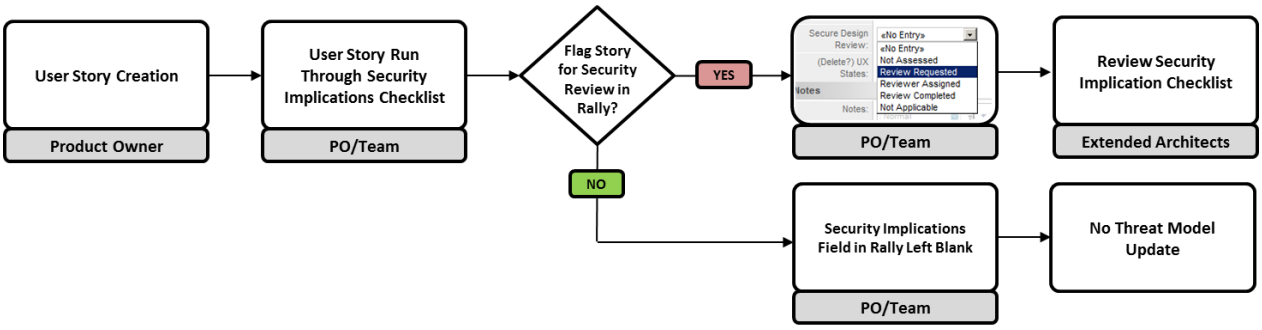


Figure 4: Flag Any Security Implications in Rally

The two essential steps are:

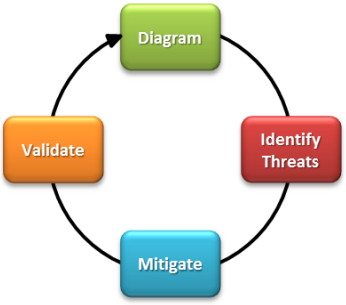
1. Create the User Story in Rally.
2. Run it through the Security Implications Checklist (i.e., answer the questions). If the answer to any checklist item is readily yes, flag the user story for Secure Design Review. If the answer to a checklist item is unknown at the present time, flag the user story for review. DO NOT GUESS. When in doubt, request a review for all security matters.

Flagged stories are put in queue for a biweekly security checklist review by the extended architecture team (XAT). Upon completion of the review by the XAT, if security issues indeed exist, the proposed feature is subjected to the full threat model process diagrammed below and described in the next section.

5.2 Threat Modeling

Per the recommendation of OWASP, Good's threat modeling process is adapted from the threat modeling process developed by Microsoft, which is comprised of four basic steps:

1. Diagram the application in a data flow.
2. Identify any threat risks.
3. Devise countermeasures to eliminate the risk or mitigate its impact.
4. Validate the chosen security solution.



Good's adaptation extends the basic Microsoft threat model to the one shown in Figure 5.

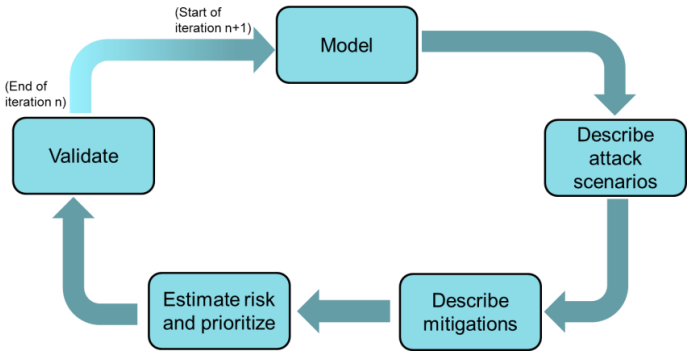


Figure 5: Basic Threat Modeling Process

Before diving too deeply, however, it is important to remember that the objective of threat modeling is to define a holistic set of mitigation actions corresponding to known (identified) and/or suspected threats at the lowest possible level (finest granularity) with the goal of removing, reducing, transferring, or accepting the security risk without adversely affecting product functionality, usability, or performance. The model must then be appropriately updated to reflect any product change carrying security implications.

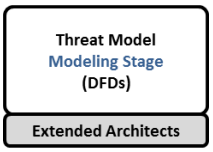
The threat risk analysis exercise is therefore divided into three major parts or stages:

- **Modeling** – to understand how the new feature will be fitted into the existing architecture.
- **Description** – to describe in detail each element-threat duple (or pair), including attack vectors, attackers and possible impacts.
- **Prioritization** – to quantify, compare, and rank the amount of risk presented by each evaluated threat.

The actual nuts and bolts of the process are significantly more involved, as we'll see next, beginning with the modeling stage.

5.2.1 Modeling Stage

Modeling forces developers to adequately understand a new feature and how it fits within the existing architecture in order to accurately assess the security implications of the change. This starts with the creation of a high-level architecture diagram describing the composition and structure of the application and its subsystems as well as its physical deployment characteristics. Figure 6 is one example of such a high-level design.



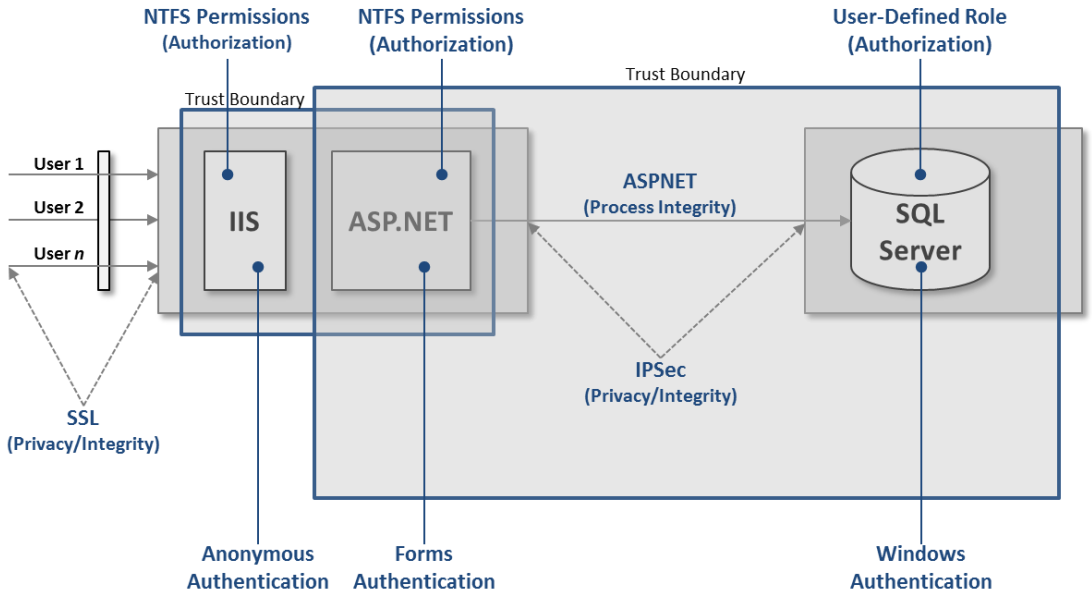


Figure 6: Example of a High-Level Design

Depending on the complexity of the system, you may need to create additional diagrams that focus on different areas; for example, a diagram to model the architecture of a middle-tier application server or one to model the interaction with an external system.

5.2.1.1 Decompose the Application

Next, you break down your application to create a security profile based on traditional areas of vulnerability while documenting the security profile; at the same time, identifying trust boundaries, data flow, entry points, and privileged code.

Also consider server trust relationships. Does a particular server trust an upstream server to authenticate and authorize end users, or does the server provide its own gatekeeping services? As well, does a server trust an upstream server to pass it data that is well formed and correct?

Ergo, follow a series of concrete, repeatable steps that include:

1. Identifying trust boundaries
2. Identifying data flow
3. Identifying entry points
4. Identifying privileged code

5.2.1.2 Map the Data Flows

Schematically, some or much of this information should already be in hand from previous modeling and should be readily adopted. In any case, the current modeling exercise eventually results in a new product/feature definition using a finite set of elements in a data flow diagram (DFD). These elements include the identified external entity/interactors, processes, data stores, and data flows. Trust boundaries are represented by a dotted line or arc crossing a data flow, similar to the example in Figure 7.

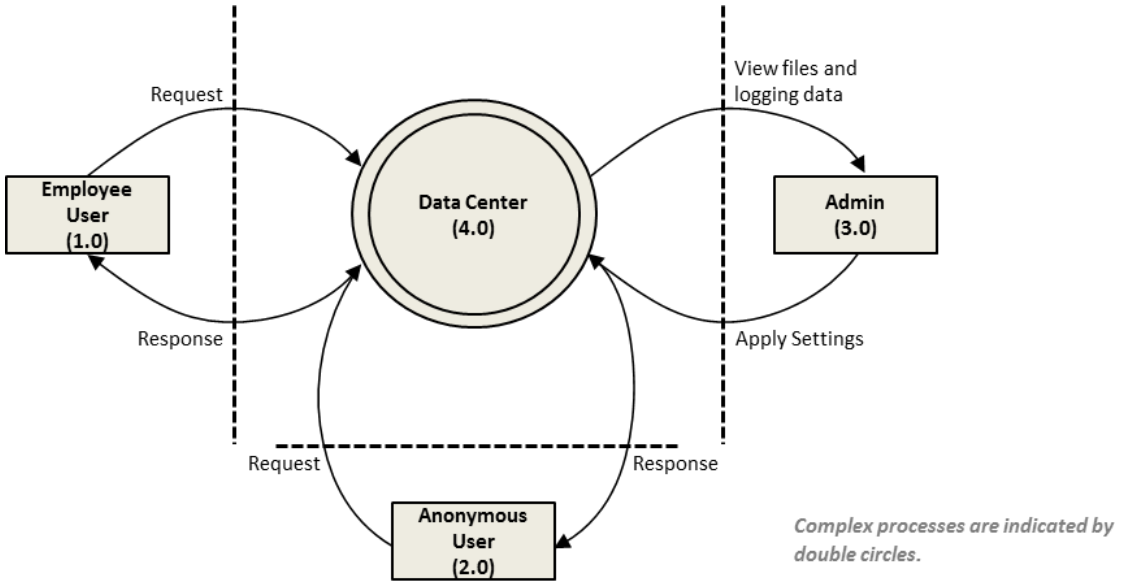


Figure 7: Context-Level Data Flow Diagram (DFD)

Trust boundaries are demarcation points in the application that show where data moves across areas of differing privileges or trustworthiness. They are important in modeling threats because they represent possible avenues of attack.

As the modeling proceeds, each DFD level is expanded until all of the following conditions are met:

a) There are no complex processes (those depicted with double circles).

b) No conditional or exceptional situations are required to explain the diagram.

c) The diagram contains all necessary information to explain any security implication of the design.

Figure 8 and Figure 9 show examples of successively detailed diagramming.

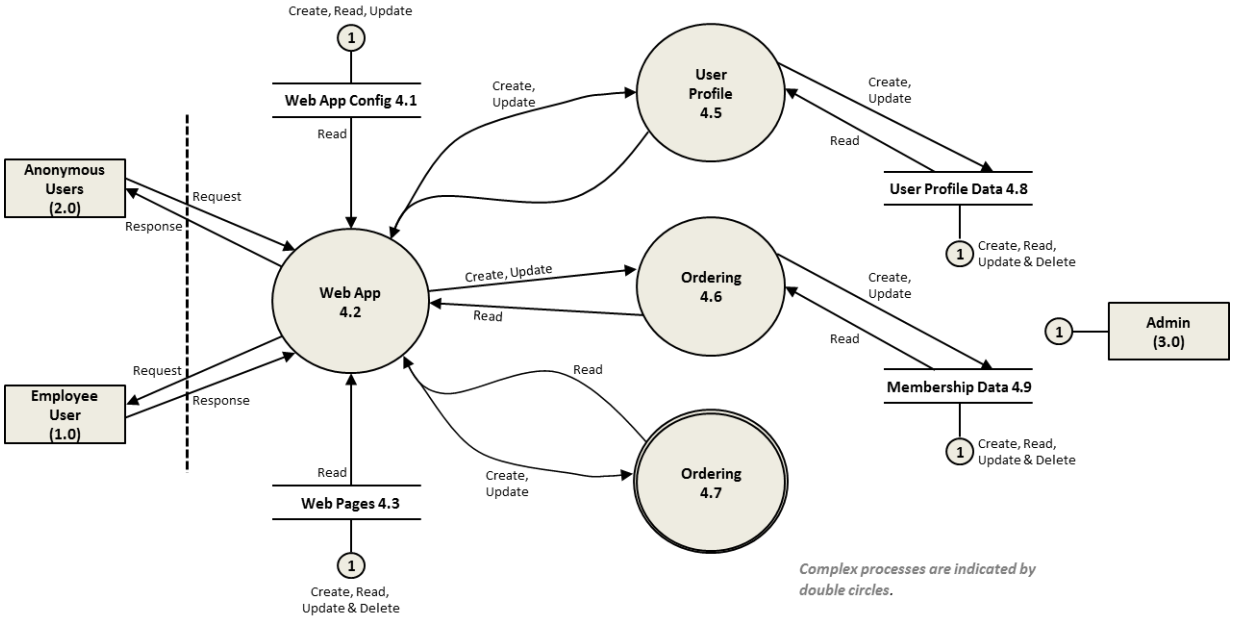


Figure 8: Level-0 DFD

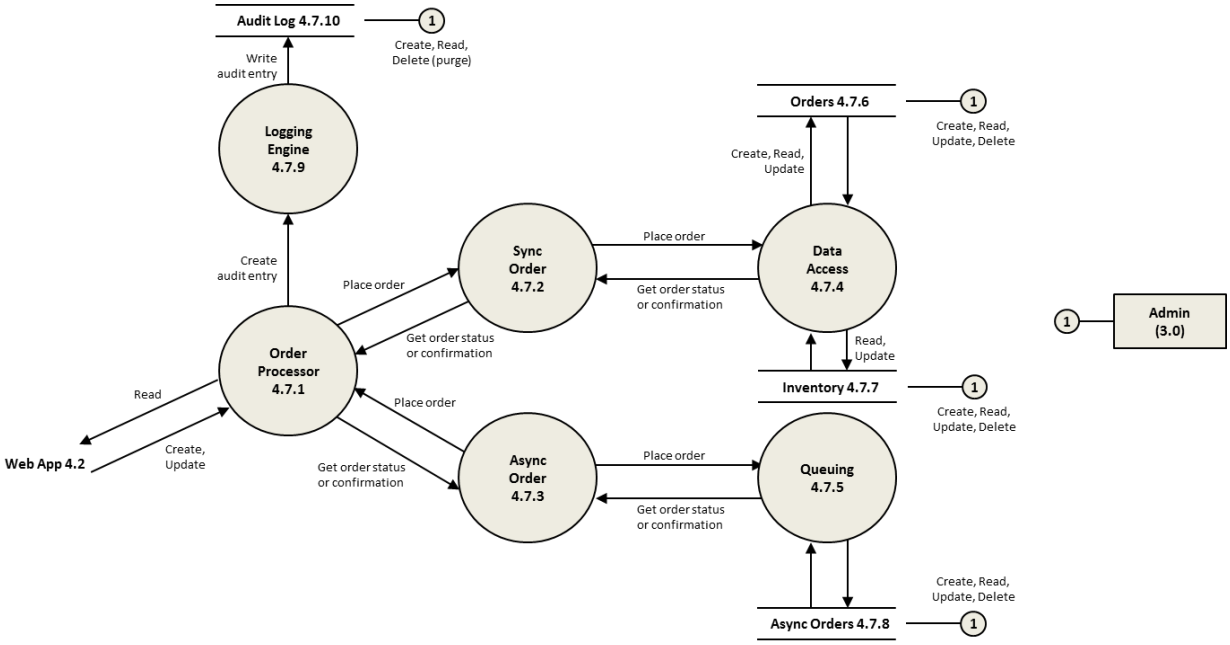
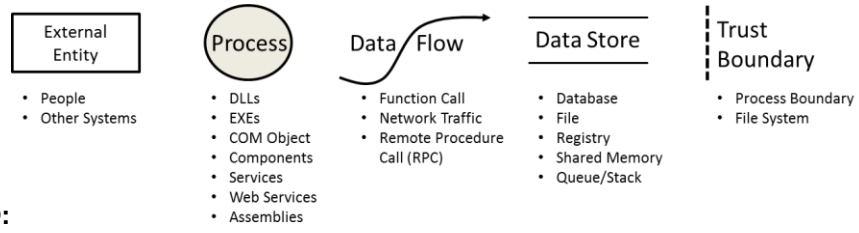


Figure 9: Level-1 DFD for Order Processing

LEGEND:



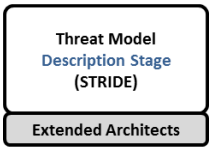
DFDs track how data enter, leave, and traverse the system, showing sources and destinations, relevant processes that data goes through, and trust boundaries. Whether building a new system or extending an existing one, the purpose is to examine how an intruder might go about attacking it so that appropriate defenses can be built and maintained.

5.2.1.3 Review the Security Profile

Next, you should review and confirm the product/system/application design and implementation approaches used for input validation, authentication, authorization, configuration management, and the remaining areas where applications are most susceptible to vulnerabilities. Whether already on paper (preferably) or recalled from memory, by doing this, you are creating an updated security profile which will serve you well in the next two stages.

With the security profile in hand and the appropriate level of diagramming complete, the threat description stage can begin.

5.2.2 Description Stage



Here we describe in detail each element/threat duple, including attack vectors, attackers and possible impacts. Good has adopted Microsoft's STRIDE threat taxonomy—or classification technique—to identify all threats associated with each element type in the DFDs. In conjunction with STRIDE, scoring/ranking each identified threat is done using a modified version of Microsoft's DREAD threat scoring mechanism.

5.2.2.1 Complete a STRIDE Assessment

STRIDE stands for:

S*poofing* = threats that abuse authentication mechanisms
T*ampering* = threats that abuse the integrity of data and binaries
R*epudiation* = threats that abuse non-repudiation mechanisms
I*nformation Disclosure* = threats that abuse confidentiality mechanisms
D*enial of Service* = threats that abuse the availability of processes and/or data
E*levation of Privilege* = threats that abuse authorization mechanisms

STRIDE thus narrows the search space for exploitable product/system/feature security weaknesses down to only those potential abuses capable of directly affecting the given DFD element. For instance, when analyzing data flow elements of the model, the focus is kept on tampering, information disclosure, and DoS threats, since only these carry potentially applicable STRIDE risks. The correlation of DFD elements with their applicable STRIDE threats is summarized in Figure 10.

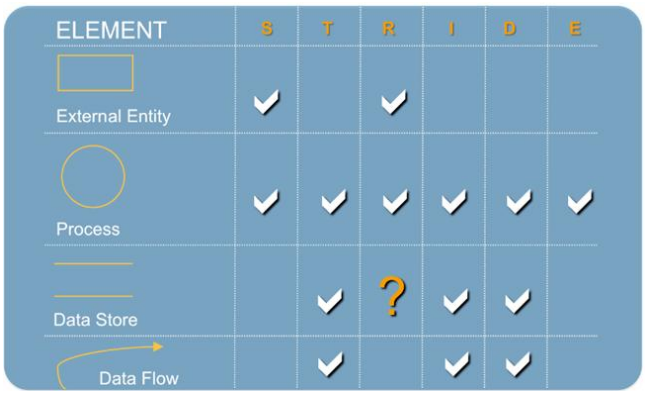


Figure 10: Threats Affecting Each Element Type

Note: The question mark listed for "Data Store–Repudiation" merely indicates that repudiation threats could affect data stores if and only if the data store contains some type of audit log.

Evaluating the complete DFD against all applicable STRIDE threats to determine how the system could fail means that for each element/threat the following must be adequately considered and assessed:

- Possible attackers (both internal and external)
- Intention (malicious vs. accidental/inadvertent)
- Attack vector
- Complexity of attack
- Impact of attack

Placed into perspective, the threat model for Good for Enterprise (GFE) currently contains more than 4,200 elements/threats.

Hence, due to the expanding size and complexity of Good's product threat models, the investigative/analytical work is typically delegated to those SMEs capable of building comprehensive checklists for their application(s) or area of concern because they understand the relevant technologies. For example, the networking team is best equipped to investigate information disclosure vulnerabilities associated with the network data flow.

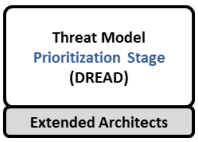
5.2.2.2 Document the Threats

Once all DFD elements have been associated with a STRIDE threat, a documented description of the identified threat, likely attackers, attack vectors, and potential impacts on each affected element must be completed. Good's threat modeling template in MS Word or the MSF threat modeling tool is ideal for this purpose. A partial example using the template (Table 1) to elaborate STRIDE for a GFE element would be:

Table 1: STRIDE Threat Description

DF15 - Auth info (license key, serial), device info, Updated MDM profile		
Threat	**Impact**	**Mitigations**
Tampering (DF15-T)	This data flow transports authentication information from GMC to NOC (WebApp web services) for MDM update support during regular operations. The most likely attacker is an external attacker located in a network location where traffic could be intercepted. In cases where the attacker can successfully modify this data flow, he could also intercept and have access to the content of any GMC->WebApp data flow. This would provide direct access to GMC credentials (to impersonate this system later), manipulate content to remove devices (DoS for some devices), etc. It would also provide access to sensitive information such as configuration parameters in MDM profiles, URL IDs, etc. No corporate PIM data would be at risk due to end-to-end encryption. Some other ancillary network attack techniques would be necessary to redirect traffic (e.g. DNS poisoning, etc.) and knowledge of the application protocol.	

5.2.3 Prioritization Stage



With the threat model now adequately describing all possible threat scenarios/ attackers/impacts for the entire list of elements and identified threats, a rating of original risk is undertaken.

However, accurately calculating the widely accepted measure of risk (risk equals the chance of attack multiplied by the damage potential) is not always an easy task. That is why we use DREAD scores.

5.2.3.1 Rate the Original Risks

DREAD stands for:

D*amage Potential* = How bad would an attack be?
R*eproducibility* = How easy is it to reproduce the attack?
E*xploitability* = How much work is required to launch the attack?
A*ffected Users* = How many users will be impacted?
D*iscoverability* = How easy is it to detect and identify the threat?

The DREAD mechanism qualitatively factors in security-related aspects to derive a value between 0 and 15 for each element-threat incident identified using STRIDE. Thus, DREAD rates the potential risk of a threat by applying the following weights:

3 = High
2 = Medium
1 = Low
0 = N/A or non-existing.

Each is measured using the criteria in Table 2.

Table 2: DREAD Risk Scoring

	DREAD Rating			
	High (3)	**Medium (2)**	**Low (1)**	**N/A (0)**
Damage Potential	Attacker can subvert the security system; get full trust authorization; run as administrator; upload content	Leaking sensitive information	Leaking trivial information	No information is compromised
Reproducibility	Attack can be reproduced every time and does not require a timing window	Attack can be reproduced, but only with a timing window and a particular race situation	Attack is difficult to reproduce, even with knowledge of the security hole	No attack is currently feasible
Exploitability	A novice programmer could make the attack in a short time	A skilled programmer could make the attack, then repeat the steps	Attack requires an extremely skilled person and in-depth knowledge every time to exploit	No attack is currently feasible
Affected Users	All users, default configuration, key customers	Some users, non-default configuration	Small percentage of users, obscure feature; affects anonymous users	No user is affected
Discoverability	Published information explains the attack. This vulnerability is found in the most commonly used feature and is very noticeable	Vulnerability is in a seldom-used part of the product and only a few users should come across it. It would take some thinking to see malicious use	The bug is obscure and it is unlikely any user will work out damage potential	No attack is currently feasible

Applying STRIDE to the Tampering threat example in Table 1 would result in a DREAD score of 9.

5.2.3.2 Mitigate the Threats

Available actions that could reasonably mitigate a security risk include:

- Threat Elimination
- Threat Reduction using standard countermeasures
- Threat Reduction using new countermeasures
- Risk Assumption
- Risk Transference

Clearly, complete elimination of risk is the ideal solution, usually achieved with redesign; for example, removing and replacing the attack-vulnerable interface, presuming a ready and viable alternative is handy. Here, in addition to adequate performance, viability demands a management-approved cost benefit.

Reducing the risk by implementing standard countermeasures, like PKI, is the favored solution. There are a variety of other proven countermeasures already established in Good's arsenal that can be leveraged as well.

Devising new countermeasures—creating a new (proprietary) security protocol or encryption mechanism—is considered the change option of last resort, since new algorithms and implementations could adversely affect already secure and stable aspects of the design and lead to certification issues (i.e., Common Criteria, FIPS, etc.).

Sometimes, expressly assuming (accepting) a particular risk after performing an informed evaluation of the possible impacts is the best way to handle the issue; for example, performing OTA with an emailed PIN. This is only acceptable under certain circumstances and with senior technical management approval when no other technical solution is available or feasible.

Another mitigation option is transferring the risk to the customer and/or users by making them aware of—and, hence, accountable for—the possible security impact. This usually occurs only when the customer wants the affected feature or component despite the potential risk, generally pending an adequate workaround in the near-term, coupled with longer-term reduction or elimination of the risk.

In all cases, mitigation actions are a collaborative determination made by Good's security architects and engineers defined in a broad form—e.g., "*Every system in the GFE solution must establish mutually authenticated channels before starting or processing any remote requirement*." Conversely, it is also possible to directly define concrete security requirements like "*XMLGW must use TLS v1.2 with client-side authentication using X.509 v3 certificates created by Good Technology's intermediate CA1*."

Extending the template example in Table 1 to include mitigation actions already defined in the current threat model produces the result in Table 3.

Table 3: Identified Threat Instance with Mitigation Actions

DF15 - Auth info (license key, serial), device info, Updated MDM profile		
Threat	**Impact**	**Mitigations**
Tampering (DF15-T)	This data flow transports authentication information from GMC to NOC (WebApp web services) for MDM update support during regular operations. The most likely attacker is an external attacker located in a network location where traffic could be intercepted. In case the attacker can successfully modify this data flow, he could also intercept and have access to the content of any GMC->WebApp data flow. This would provide direct access to GMC credentials (to impersonate	- (**MA-46**) All GFE systems must create and use secure channels to interact with other GFE servers. - (**MA-70**) GFE servers should not send any credential in the clear or any value derived from a credential that could be used to obtain the actual credential. - (**MA-41**) GFE systems should not send any credential to unauthenticated external database servers. - (**MA-3**) The GFE's preferred authentication mechanism for inter-system communications is

DF15 - Auth info (license key, serial), device info, Updated MDM profile		
Threat	**Impact**	**Mitigations**
	this system later), manipulate content to remove devices (DoS for some devices), etc. It would also provide access to sensitive information such as configuration parameters in MDM profiles, URLIDs, etc. No corporate PIM data would be at risk due to end-to-end encryption. Some other ancillary network attack techniques would be necessary to redirect traffic (e.g. DNS poisoning, etc.) and knowledge of the application protocol.	digital certificates for both clients as servers. - (**MA-18**) When using digital certificates all GFE systems must successfully identify the other party in such a way that Common Name or Distinguished Name fields of the presented certificate match the host name or username of the system or human user respectively. This identification process must be enforced before any further processing. - (**MA-30**) When using digital certificates all GFE systems must perform certificate validation according to RFC5280. This validation process must be enforced before any further processing. **Note**: If this component uses secure channels like TLS v1.2 with X.509 v3 certificates and it correctly validates server's/client's certificates, this threat could be considered closed.

5.2.3.3 Rate the Residual Risks

Any residual risk can be estimated only after the associated security requirements have been defined for each element-threat pair in the previous step. Residual risk is the threat risk remaining after all security requirements and countermeasures have been implemented.

Table 4 shows a sample threat model scoring spreadsheet correlating elements, threats, original DREAD scores, mitigation action/security requirement, and reduced DREAD Score (the residual risk).

Table 4: Completed Threat Model Spreadsheet

Good for Enterprise - Threat Model

STRIDE/DREAD Analysis

						Original DREAD score						Mitigation Actions		Reduced DREAD score					
Index	Element Label	Element Type	Threat Type	DFD #		D	R	E	A	D	Total	Code	Status	D	R	E	A	D	Total
DF1-T	AD credentials, request new email and calendar info	DataFlow	T	7,		2	3	2	2	3	12	MA-59		0	0	0	0	0	0
DF1-T	AD credentials, request new email and calendar info	DataFlow	T	7,		2	3	2	2	3	12	MA-64		1	1	1	1	1	5
DF1-T	AD credentials, request new email and calendar info	DataFlow	T	7,		2	3	2	2	3	12	MA-66		1	1	1	1	1	5
DF1-ID	AD credentials, request new email and calendar info	DataFlow	I	7,		2	3	2	2	3	12	MA-59		0	0	0	0	0	0
DF1-ID	AD credentials, request new email and calendar info	DataFlow	I	7,		2	3	2	2	3	12	MA-64		1	1	1	1	1	1
DF1-ID	AD credentials, request new email and calendar info	DataFlow	I	7,		2	3	2	2	3	12	MA-66		1	1	1	1	1	1
DF1-DOS	AD credentials, request new email and calendar info	DataFlow	D	7,		1	2	2	2	2	9	none		0	0	0	0	0	0
DF2-T	AD credentials, validate new user, users to retrieve PIM	DataFlow	T	6,		2	3	2	2	3	12	MA-59		0	0	0	0	0	0
DF2-T	AD credentials, validate new user, users to retrieve PIM	DataFlow	T	6,		2	3	2	2	3	12	MA-64		1	1	1	1	1	5

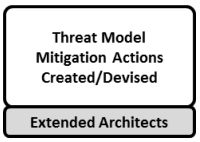
5.2.3.4 Optional Use of Microsoft's Threat Modeling Tool

Available at http://www.microsoft.com/en-us/download/details.aspx?id=2955, the Microsoft Solution Framework (MSF) Threat Modeling Tool is a company-approved aid to help architects and engineers analyze their designs and software architecture. Depending on preference, use of the tool can be limited to the description stage or used to aid threat modeling in its entirety. Its benefits include:

- Populating the input forms used in the description phase as DFDs are drawn. For instance, when a new process is added, the tool creates a description form with all relevant threats. Additionally, whenever the diagram is modified, the forms are automatically updated accordingly.
- Validating DFDs against common errors like data sinks, i.e., data stores without output data flows.
- Certifying that some threats do not apply to a particular element (e.g., tampering for an aggregation data store that is a few levels below an attacker-unique surface).
- Generating basic reports for export to Excel.

While not a commercial grade application, the tool saves significant time when working on big, complex models.

5.2.4 Replace General Mitigation Actions with Concrete Security Requirements



General mitigation actions are preferred to concrete security requirements whenever models are big or complex. Using general mitigation actions instead of concrete security requirements also reduces the number of countermeasures, which is especially handy during the rating and prioritization phase.

Note: Neither the original nor the reduced DREAD scores are changed as a result of replacing mitigation actions with more concrete security requirements.

Table 5 shows an example of the correct substitutions.

Table 5: Mitigation Action Replacement with Concrete Security Requirements

Original Mitigation Action	New Security Requirement
Run exposed daemons under unprivileged OS-level accounts.	XMLGW must create unprivileged Linux account apache-root without access to any other group but apache-root. Remove any interactive shell in **/etc/passwd** file and block account from being used to gain console access.
	GMC must create unprivileged local Windows account (not part of AD) named tomcat5 with "log on as service" rights and make tomcat runs under this account. Make sure this account can't interact with local desktop.
Implement mechanisms to detect abnormal behaviors consistent with possible DoS.	GMC must create a login counter that keeps the number of unsuccessful login attempts. When they reach certain configurable threshold system must apply a configurable back off algorithm to thwart possible brute force attacks
	GMC must implements a centralized input sanitation process that keeps track of format errors in the same session. If number of error reaches certain configurable threshold the system must be able to log take a configurable action (e.g. log event, close session, block user temporarily/permanently, etc.)
	The GMC's web server must implement a maximum/minimum number of child processes to prevent system resources from being depleted by untrusted requests.

5.2.5 Update the Threat Model

A new, modified or specially devised mitigation action may not be directly replaceable with a concrete security requirement already in the threat model. In which case, a new security requirement must be defined/described and the threat model (database) updated accordingly. And there's no better time than now to do it, while the relevant information is still fresh in your mind. DO NOT PROCRASTINATE. Make the updates.

5.2.6 Prioritize Security Requirements Using DREAD Scores

While abstract business and/or customer needs may remain a factor, threat prioritization can now proceed based on the model's DREAD scores, wherein priority is given to those security requirements carrying the most important risk reductions. This is done by computing the risks with the greatest variance between original and residual risk according the formula:

$$\mathbf{DREAD}_{variance} = \mathbf{DREAD}_{original} - \mathbf{DREAD}_{reduced}$$

The higher the $\mathbf{DREAD}_{variance}$ score, the higher the priority.

Hence, threat ambiguity is resolved with a traceability matrix—basically, a spreadsheet showing:

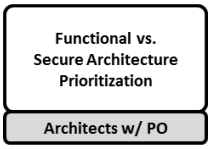
- **WHO** – the threat: an agent capable of doing harm to the system or data
- **WHAT** – the misuse/abuse a threat intends to promulgate
- **WHERE** – the attack surface on which a threat will conduct a misuse/abuse
- **HOW** – the specific attack vectors

- **IMPACT** – the negative effect on a business objective or violation of policy
- **MITIGATION** – the column to track risk acceptance, transfer, or mitigation.

An example of a completed traceability matrix is shown in _____.

{Need a spreadsheet sample from Luis}

5.3 Advise Product Owner – Jointly Agree on Priorities



With all security requirements now identified, defined, and prioritized for the user story in question, the PO is notified of the results; after which, it becomes the product owner's responsibility, in consultation with the architects, to determine:

(a) whether or not the business value of adding the functionality to the current architecture warrants the development work required
(b) if the residual threat risk is acceptable.

If the PO decides the cost-benefit or risk-benefit ratio is prohibitive or at least not worth it at the present time, the user story does not move forward. It will remain, however, in the product backlog for future consideration with the advantage that its threat model is already completed and on file.

If both determinations, (a) and (b), are affirmative, the PO can now determine its priority for secure design completion and subsequent implementation.

5.4 Complete Secure Design of Prioritized User Stories

The swiftest and most dependable way to create the new/changed security design is to apply a secure design pattern, then make the appropriate, specific adjustments satisfying the threat modeled requirement(s).

A secure design pattern is a generally reusable solution to commonly occurring security issues. It is important to remember that a design pattern is not a finished design that can be transformed directly into code. Rather, it is a description or template for how to solve a problem. As indicated above, mitigation actions and secure design requirements address security issues at widely varying levels of specificity ranging from architectural-level patterns involving the high-level design of the system down to implementation-level patterns providing guidance on how to implement portions of functions or methods in the system.

Good, like other software innovators, is under constant pressure to build reliable products that meet all customer requirements within a short period of time. So architects and developers alike need to rely on proven practices and methodologies. Design patterns allow them to exploit the previous successes (and fails) of Good's collective engineering experience.

The benefits of knowing and using design patterns are several. They reduce development time since patterns are known solutions for building software systems. They improve software quality because the solutions are tried and tested. Patterns also improve the communication between development teams by identifying names and providing structures to many of the same challenges faced by all developers in all teams.

Design patterns rely on decomposing larger tasks into smaller pieces; isolating and encapsulating the variable part of the system; extending the behavior of objects with a common base class in a similar fashion using polymorphism; and loosely coupling the dependent objects to reduce their dependency. Thus, design patterns are the design equivalent of object-oriented programming.

Good has created numerous secure design patterns by generalizing and cataloging existing best practices and by extending existing non-secure design patterns. By correctly using or applying the security patterns specified by Good's architects, developers will reduce both the cost and risk associated with producing secure products.

The template for describing design patterns is shown in Table 6.

Table 6: Design Pattern Elements[1]

Design Element	*Description*
Pattern Name and Classification	A descriptive and unique name that helps in identifying and referring to the pattern.
Intent	The problem solved by the design pattern and its general rationale and purpose.
Also Known As	Other names for the pattern, if any are known.
Example	A real-world example demonstrating the existence of the problem and the need for the pattern. Throughout the description, refer to examples to illustrate solutions and implementation aspects where necessary or useful.
Motivation	A description of situations in which the pattern may apply and a more detailed description of the problem that the pattern is intended to solve.
Applicability	A general description of the characteristics a program must have for the pattern to be useful in the design or implementation of the program.
Structure	A textual or graphical description of the relationship between the various participants in the pattern. This provides a detailed specification of the structural aspects of the pattern using appropriate notations.
Participants	The entities involved in the pattern.
Collaboration	A description of how classes and objects used in the pattern interact with each other.
Consequences	The benefits the pattern provides and any potential liabilities.
Implementation	Guidelines for implementing the pattern. These are only a suggestion, not an immutable rule. You should adapt the implementation to meet your needs by adding different, extra, or more detailed steps or by reordering the steps. Whenever applicable, give UML fragments to illustrate a possible implementation, often describing details of the example problem.
Sample Code	Code providing an example of how to implement the pattern.
Example Resolved	An example of how the real-world example problem described in the Example section may be resolved through the use of this secure design pattern.
Known Uses	Examples of the use of the pattern taken from existing systems/implementations.

Of particular importance are the Structure, Participants, and Collaboration sections. These sections describe a *design motif*, i.e., a prototypical micro-architecture that developers can copy and adapt to their particular designs to solve the recurrent problem described by the design pattern. A micro-architecture is a set of program constituents (e.g., classes, methods, etc.) and their relationships. Developers use the design pattern to introduce this prototypical micro-architecture within their current implementation. By doing so, the solution will have structure and organization similar to the chosen design motif governing the product as a whole; certainly from a security perspective.

5.5 Modify User Story with Security Requirements and Design

Completing a secure design that incorporates each of the security requirements derived from the threat model means creating a new security story in Rally and then pairing it with the original user story that prompted security handling.

{Need steps and Rally screenshots for splitting original story into a security and a user story, or a user story with a secure design and requirements attached}

5.6 Secure Design Review (SDR)

As previously touched on in Section 4.4, when finished, the new security designs must be reviewed for accuracy and completeness. This occurs in the SDR, a collaborative examination of the multilayer security surrounding the application/system technical design and procedures. The SDR is conducted by

[1] C. Dougherty, et al, *Secure Design Patterns*, Software Engineering Institute – Carnegie Mellon University (2009)

the core architecture team as soon as possible after the threat modeling exercise is completed and high-level designs become available. Ideally, SDR should occur 9 to 6 weeks before the sprint commit.

As part of the review, the team must ensure that the design withstands the attacks described in the threat model. Table 7 lists the minimum vetting criteria of the SDR.

Table 7: Secure Design Vetting Criteria

System Security Requirement	Secure Design Criteria
Solution/feature performs different functions requiring different privileges.	• Does the design provide for components with different privileges? • Does the design have a special unprivileged component in charge of handling pre-authentication tasks? • Does the design take advantage of OS-level functions for high-priority tasks instead of adding custom user-space functions? • Does each component distrust inputs from other components, users, and external systems? If so, does the design explicitly define centralized sanitation operations? • Does the design employ any sandboxing techniques (e.g., creating simulated conditions in a protected, closed environment for studying and analyzing potentially malicious code/inputs)?
Solution/feature damage control measures are in place in the event of failure or compromise	• Does the design include components for detecting failed/compromised events? If so, how accurate is the detection and what are the limitations? • If compromised, how does the design limit potential damage? • In the event of compromise, does the design's reduced attack surface provide the minimum security properties defined in the design's residual DREAD?
Solution/feature maintains activity logs and supports security alerts	• Is the design able to register security breaches and issue alerts? • Is the design able to protect its log files against unauthorized modification? • Does the design have a centralized secure logger? If so, does it use the secure logger pattern?
Solution/feature processes data originated from untrusted sources (e.g., user inputs)	• Does the design show where and how data validation and sanitation is performed? • Are there centralized input validation/sanitation mechanisms? • Does the design avoid undesired side-effects caused by conflicting or out-of-order validation/sanitation measures? • Is the design able to correctly handle different valid encodings?
Protection of data in transit	• Does the design detect and reject replay, fragmentation, and similar attacks that could regularly occur? • How does the design ensure the confidentiality, integrity, and authenticity of data in transit? • Is the design able to thwart MITM attacks within reception range of an unencrypted Wi-Fi access point? • If a proprietary protocol is used, does the design adequately explain the initialization and operational conditions necessary to ensure its secure state; for example, origin and amount of entropy, min/max number of messages/bytes/time to ensure PFS and reuse of the same counters/IV, etc., with the same encryption material?
Protection of data at rest	• Does the design ensure that no sensitive data, including cryptographic material and user data, is available in the file system? • Does the design allow external components to retrieve protected information without breaking the protection of other information? • Does the design in any way inhibit the proper disposal of sensitive information, including encrypted material? Is there a "clear sensitive information" pattern applied to erase sensitive information from reusable sources like memory, memory/disk caches, etc.?
Solution/feature exposes security choices and settings to the user	Does the design allow users to choose the level of protection and security and even turn off security measures, when applicable? Are the consequences adequately communicated?
Solution/feature includes user authentication	• Does the design provide an authentication mechanism that cannot be bypassed? • Are identification and authentication performed in the same atomic operation? • Does the design include an anti-hijacking, impersonation countermeasure? Can it withstand the threat scenarios and attack vectors described in the threat model?

System Security Requirement	Secure Design Criteria
Solution/feature requires file system access	• Does the design block direct file manipulation and possible race conditions/hazards by using secure directories? • Does the design perform directory path normalization to remove references to directory Meta entries (current and parent directory)?

It is important here to underscore that the SDR is a collaborative, *constructive* team effort in which challenges to the design are encouraged to confirm its resilience and resistance to attacks. When minor flaws (warnings) are discovered, the architect includes these fixes in the current design. On the other hand, more severe flaws and vulnerabilities may require a much more extensively (sometimes entirely) reworked design. Rework and review are repeated until the threat reduction objectives itemized in the threat model and by the XA team are satisfied.

Perhaps most importantly, approval is not gained until a Good Secure Design Approval Form is completed and signed by each SDR review member.

{It would be nice to include an example of the approval form here}

5.7 Secure Coding and Security Testing Practices

Easily avoided software defects are a primary cause of commonly exploited software vulnerabilities. Good has observed through its analysis of vulnerability reports that most vulnerabilities stem from a relatively small number of common programming errors.

5.7.1 Multiple Independent Levels of Security (MILS)

MILS simplifies specification, design, and analysis of security mechanisms. It is based on the concept of separation, which tries to provide an environment that is indistinguishable from a physically distributed system. A hierarchy of security services is obtained through separation. In this hierarchy, each level uses the security services of a lower level to provide a new security functionality that can be used by the higher levels. Each level is responsible for its own security domain and nothing else.

Good's architecture divides system functions into three levels: the application layer, the middleware service layer, and the separation kernel (SK). Sometimes called the partitioner layer, the SK is the base layer of the system, responsible for enforcing data separation and information flow control; providing both time and space partitioning via:

- **Data Separation** – the memory address space, or objects, of a partition are completely independent of other partitions
- **Information Flow** – pure data separation is not practical so there is a need for the partitions to communicate with each other. The SK defines the moderated mechanisms for inter-partition communication.
- **Sanitization** – the SK is responsible for cleaning any shared resources (registers, system buffers, etc.) before allowing a process in a new partition to use them.
- **Damage Limitation** – address spaces of partitions are separated, so faults or security breaches in one partition are limited by the data separation mechanism.

5.7.2 Defensive Programming

By identifying insecure coding practices and developing secure alternatives, Good's developers must proactively take practical steps to reduce or eliminate vulnerabilities before deployment. Thus, coding standards are enforced to ensure that developers:

- Clarify rather than obfuscate
- Promote intention-revealing code
- Produce easily readable code
- Ensure code quality and security

- Incorporate coding best practices

The result is Scrum Teams that have a shared understanding of quality code and the practices necessary to produce it, which include:

- Intentional programming (IP), i.e., WYSIWYG, keeping interpretation to a minimum.
- Practicing the "once and only once" rule; i.e., redundancy is present and must be rectified if a code change in one place requires a corresponding change in another.
- "Encapsulating by policy, revealing by need," wherein encapsulation simply means making the fields in a class private (hiding them) while still providing access to the fields via public methods. The main benefit here is the ability to modify already implemented code without breaking the code of others who use it.
- Attending to the quality of the code; this is measured by how well it manifests encapsulation, strong cohesion, proper coupling, no redundancy, readability, and testability.

In sum, Good's coding standards and practices have been established to keep the code consistent and easy for the entire team to read and refactor. Code that looks the same encourages collective ownership.

Defensive coding is the practice of anticipating where failures can occur in order to create an infrastructure that tests for errors, delivers notification when anticipated failures occur, and performs the damage-control actions that have been specified, like halting program execution, redirecting users to a backup server, enabling debugging information used to diagnose the issue, and so forth. Such defensive coding infrastructures are built by adding assertions to the code and validating all user inputs.

5.7.3 Security Testing

In concert with Good's security principles addressed in Section 6, security testing covers six basic concepts:

- Confidentiality
- Integrity
- Authentication
- Availability
- Authorization
- Non-repudiation

Good's security taxonomy defines these concepts to deliver robust security testing by the following means:

5.7.3.1 Discovery

This approach identifies systems within scope and the services in use. It is not intended to discover vulnerabilities, although version detection may highlight deprecated versions of software/firmware and thus indicate potential vulnerabilities.

5.7.3.2 Vulnerability Scan

Following the discovery stage, this scan looks for known security issues by using automated tools to match conditions with known vulnerabilities. The reported risk level is set automatically by the tool with no manual verification or interpretation by the tester or tool vendor. This is supplemented with credential-based scanning that seeks to remove any common false positives by using supplied credentials to authenticate with a service (such as local Windows accounts).

5.7.3.3 Vulnerability Assessment

This approach uses discovery and vulnerability scanning to identify security vulnerabilities and places the findings into the context of the environment under test.

5.7.3.4 Security Assessment

Security assessment builds upon vulnerability assessment by adding manual verification to confirm exposure, but does not include the exploitation of vulnerabilities to gain further access. Verification could be in the form of authorized access to the system to confirm system settings and involve examining logs, system responses, error messages, codes, etc. The security assessment seeks to gain broad coverage of the product but not the depth of exposure to which a specific vulnerability could lead.

5.7.3.5 Penetration Test

This test simulates an attack by a malicious party. Building on the previous stages, it involves exploitation of found vulnerabilities to gain further access. Using this approach results in an understanding of the ability of an attacker to gain access to confidential information, affect data integrity or availability of a service and the respective impact. Each test is approached using a consistent and complete methodology that allows the tester to use their problem solving abilities, the output from a range of tools, and their own knowledge of networking and systems to find vulnerabilities that would/could not be identified by automated tools. This approach looks at the depth of attack as opposed to the security assessment approach, which looks at the broader coverage.

5.7.3.6 Security Audit

Driven by an audit/risk function to look at a specific control or compliance issue and characterized by a narrow scope, this type of engagement makes use of any/all of the earlier approaches, i.e., vulnerability assessment, security assessment, and penetration test.

5.7.3.7 Security Review

This review verifies that industry and special internal security standards have been applied to the product software. It is typically completed using gap analysis while referencing code reviews, design documents, and architectural diagrams. As a rule, this activity does not piggyback on any of the earlier approaches, i.e., vulnerability assessment, security assessment, penetration test, or security audit. This is necessary for its findings to be appropriately objective.

5.8 Proper Release Documentation and Secure Packaging

{Need to create this, but have no real information on it, yet. Maybe Henry can dig up something or point me to a knowledgeable SME for this area}

5.9 Incident Management

Whereas continuous monitoring activities are about tracking and monitoring attempts that could potentially breach the security of systems and software, incident management activities are about the proper protocols to follow and the steps to take when a security breach (or incident) occurs.

Starting with the detection of the incident—accomplished by monitoring, using incident detection and prevention systems (IDPS), and other mechanisms—the first step in incident response is to determine if the reported or suspected incident is truly a legitimate security issue. Upon determination of valid incidents and their type, steps to minimize the loss and destruction and to correct, mitigate, remove, and remediate exploited weaknesses must be undertaken so that computing services can be restored as expected. Clear procedures to assess the current and potential business impact and risk must be established along with the implementation of effective and efficient mechanisms to collect, analyze, and report incident data. Communication protocols and relationships to report on incidents both to internal teams and to external groups must be established and followed.

The main types of security incidents include:

- **Denial of Service (DoS)** – the most common type of security incident, DoS is an attack that, by exhausting resources, prevents or impairs an authorized user from using the network, system, or software application.

- **Malicious Code** – has to do with code-based malicious entities like viruses, worms, and Trojan horses that can successfully infect a host.
- **Unauthorized Access** – refers to incidents wherein a person gains logical or physical access to the network, system, or software application, data, or any other IT resource without having been granted the explicit rights to do so.
- **Inappropriate Usage** – incidents in which a person violates the acceptable use of system resources or company policies.
- **Multiple Components** – incidents that encompass two or more incidents. For example, a SQL injection exploit at the application layer allowed the attacker to gain access and replace system files with malicious code files by exploiting weaknesses in the web application that allowed invoking extended stored procedures in the insecurely deployed backend database.

Appropriate containment, eradication, recovery and post-incident analysis procedures logically follow.

5.9.1 Containment

Upon detection and validation of a security incident occurrence, the incident must be contained to limit further contamination, damage, or additional risks. Containment includes the most appropriate and expedient combination of shutting down the system, disconnecting the affected system from the network, disabling ports and protocols, turning off services, and/or taking the application offline. Moreover, containment strategies must be based on the type of incident, each of which may require a distinct strategy to limit its impact.

While delayed containment can be useful in collecting more evidence by monitoring the attacker's activity, this can be dangerous, since the attacker could also be presented with an opportunity to elevate privilege and compromise additional assets. Even though Good's highly experienced field incident response support team (FIRST) is capable of monitoring all attackers' activity and terminating attacker access instantaneously, the risks posed by delayed containment make it an inadvisable strategy, and in all cases any strategic delay must be at the discretion of the legal department according to the following criteria:

- Potential impact caused by theft of resources
- Need to preserve evidence
- Availability of affected software and services
- Time and resources needed to execute the strategy
- Possibility of infection triggering destructive malware.

Moreover, it must always be remembered that incident data and information is in no way "water cooler" conversation and must be restricted to authorized personnel with a direct need to know.

5.9.2 Eradication

Eradication can be performed as a standalone step or procedure, or during recovery, and only after appropriate authorization has been granted. When dealing with licensed and third-party components or code, it is important to make sure which party has the contractual rights and obligations to make and redistribute security modifications.

5.9.3 Recovery

Recovery mechanisms aim to restore the resource—network, system, or software application—to its normal working state, which is usually OS-/application-specific. The recovery procedure must also include implementing a heightened degree of awareness through monitoring and logging to detect repeat offenders.

To limit the disclosure of incident-related sensitive information to outside parties, which could potentially cause more damage than the incident itself, the appropriate communications protocol must be followed (see Good Policy No. _____). Absolutely no communication to outside parties must be made before

FIRST has discussed the issue with need-to-know engineering and product management personnel, the legal department, and the affected customer's security office.

5.9.4 Post-Incident Analysis

Not all incidents will require a full-fledged post-incident analysis, but at a base minimum the following must be determined and reported to appropriate Good company officials:

- What happened?
- When did it happen?
- Where did it happen?
- Who was involved?
- Why did it happen?

Finally, securely maintaining an incident database with detailed information about the incident that occurred and how it was handled is paramount.

5.10 End-of-Life (EOL) Security Policy

All software is vulnerable until it is properly sunsetted. Thus, the first requirement regarding secure disposal of software is an established EOL policy. NIST Special Publication 800-30 prescribes the risk management activities necessary to (a) appropriately retire software, (b) ensure that residual data are properly handled, and (c) conduct system migration in a systematic and secure manner.

5.10.1 Sunsetting Criteria

Sunsetting criteria provide guidance on when a particular product (software and/or supported hardware) must be disposed of and/or replaced. This includes the following issues:

- Discovery of threats and attacks against software that can no longer be mitigated to the acceptable levels of security due to technical, operational, or management constraints.
- Software contractual agreements that have come to an end and/or the cost of maintaining and using the software has become prohibitive.
- The software is no longer compatible with the architecture of market-driven and available hardware platforms/devices and technologies that Good is committed to support.
- Newer software designs provide the same functionality in a more secure fashion.

5.10.2 Information Disposal and Media Sanitization

The importance of information disclosure protection cannot be overstressed. In addition to the software itself, the media on which the information is stored must be sanitized or destroyed as appropriate.

Sanitization is the process of removing information from media such that data recovery and disclosure is not possible, to include the removal of classified labels, marking(s), and activity logs related to the information. Information confidentiality assurance must always be the primary consideration. Clearing, purging or destroying are the common means of media sanitization.

Disposal is the act of discarding media without giving any consideration to sanitization. This is often accomplished by recycling hardcopy media wherein no confidential information is present. While technically not a type of sanitization, disposal is still a valid approach to handling media containing non-confidential information.

At all events, software that is no longer required is not secure until it and its data and related components have been completely removed from the computing environment.

* * * * *

The general yet fundamental security requirements and rules undergirding the SDL processes described above bear additional discussion next and should be reviewed periodically by all members of the technical staff involved in software development and support.

6 Software Security Principles[2]

Principles are important because they help us make security decisions in new situations with consistency and prudence. Applying these principles, we assess security requirements, make architecture and implementation decisions, and identify possible weaknesses in our systems and those with which our systems interface.

The key thing to remember is that in order to be useful, principles must be evaluated, interpreted and correctly applied to address a specific problem. Although principles can serve as general guidelines, simply telling a software developer that their software must "fail securely" or that they should do "defense in depth" won't mean that much until placed in a practical, real-world context.

To that end, Good's security requirements and assurance practices are governed by the following principles.

6.1 Reduce the Attack Surface

The attack surface of a software environment is the code that can be run by unauthorized users. It includes, but is by no means limited to, user input fields, protocols, interfaces, and services. Turning off unnecessary functionality reduces the attack surface. Minimizing the number of entry points available to untrusted users reduces the attack surface, as does eliminating obsolete functionality or services requested by relatively few users.

The core tenet of Good's attack surface reduction (ASR) principle is that no code has a zero likelihood of containing one or more vulnerabilities, and vulnerabilities will inevitably result in customer compromise. The only foolproof way to limit customer compromise is to run no code. Zero functionality = 100% security. While true, it's not very practical. Accordingly, ASR is a compromise between perfect safety and unmitigated risk by minimizing the code exposed to untrusted users.

Hence, ASR has three main goals:

- Reduce the amount of code executing by default
- Reduce the volume of code that is accessible to untrusted users by default
- Limit the damage if the code is exploited.

Of course, there will be applications threat modeled for which the local environment is trusted. In this case, having a large number of local input points, such as configuration files, registry keys, user input, etc., is less worrisome than making several external network connections.

Collapsing functionality that was previously spread across several ports onto a single port does not always help reduce the attack surface either, particularly when the single port exports all the same functionality within an infrastructure performing basic switching. In fact, the effective attack surface is the same unless the actual functionality is somehow simplified. Since underlying complexity clearly plays a role, metrics based on attack surface should not be used as the only means of analyzing risks in a piece of software.

6.2 Secure by Design Not Afterthought

This means "bolting on" security near the end of development is a no-no. Mitigation of security and privacy issues must be performed during the opening stages of a release project. Additionally, it is crucially important for product management and software developers to understand the distinction between "secure features" and "security features."

[2] Based on www.OWASP.org/index.php/CLASP_Security_Principles (2013)

Perhaps little appreciated, it is quite possible to implement security features which are, in fact, insecure. *Secure features* are defined as features whose functionality is well engineered with respect to security, including rigorous validation of all data before processing and cryptographically robust implementation of libraries for cryptographic services. The term *security feature* merely describes program functionality with security implications, like Kerberos authentication or a firewall.

Therefore, to enter development in the Scrum phase of the SDLC, features and functionality with security implications must pair a user story that adequately and accurately describes the intended use of the feature or function with a security story/design the describes how to deploy the feature or function in a secure fashion.

6.3 Insider Threats as the Weak Link

Company personnel must never overlook “insider” risks; i.e., users with inside access to a Good-engineered product or application, whether it be in deployment or development. Forgoing paranoia, this means that it is unwise to completely ignore the risks from the person in the next cube or on the next floor, the risks from maintenance workers and janitors, and especially the risks from those who have recently quit or been fired. Yearly numbers from the Computer Crime and Security Survey performed by the Computer Security Institute and the FBI show that over half of all security incidents have an inside angle.

Placing blind faith and trust in the people around you when it comes to security is not only naïve, it is irresponsible. Not only might the people you think you know be secretly disgruntled or susceptible to a bribe, they could accidentally give insider help by falling victim to a social engineering attack.

Social engineering is when an attacker uses his social skills, generally involving deception, to gain unauthorized access to confidential systems and data. The best defense against social engineering attacks is to precisely follow the company’s established security policies to the letter, no matter how charming or persuasive an individual known to you—or unknown—may be.

6.4 Assume the Network is Compromised

A variety of attacks can be launched by anyone with access to any network media that can see application traffic. It is wrong to assume that attacks in the middle of an ISP-to-ISP communication will never happen. There is always an attack risk when someone on a shared segment can see the traffic. Generally, the greatest risk lies in the local networks that the endpoints use. ARP spoofing, a technique whereby an attacker sends fake address resolution protocol (ARP) messages onto a LAN, and attacks on the physical media are actually quite easy to perform, thus obviating a pure man-in-the-middle (MITM) attack scenario. The most well-known network-level threats include:

- **Eavesdropping** – always a potential MITM threat, even when using encryption, if proper authentication is not performed.
- **Tampering** – changes to data on the wire. Encryption notwithstanding, it may be possible to make significant changes to the data even without decrypting it. Tampering is best thwarted by performing the ongoing message authentication (MARCing – stands for Message Authentication, Reporting and Conformance) furnished by most high-level protocols like SSL/TLS.
- **Spoofing** – happens when traffic is forged so that it *appears* to come from a different source address than it really does. This can thwart access control systems relying exclusively on IP addresses and/or DNS names for authentication.
- **Hijacking** – is an extension of spoofing and can occur when established connections are vulnerable to being taken over, allowing the attacker to enter an already established session without having to authenticate. MARCing is the best defense here.
- **Observing** – wherein it is possible to give away security-critical information even when a network connection is confidentiality-protected through encryption. For example, the mere fact that two particular hosts are talking may give away significant information, as can timing the traffic. Covert channels, i.e., non-obvious communication paths, tend to be the most difficult problem in the security space.

6.5 Secure by Default

A system's default setting must never expose users to unnecessary risks and should be secure as possible. This means enabling all security functionality by default, as well as disabling all optional features that could entail a security risk. Likewise, not even a system failure should cause the software to behave in an insecure manner. This is the "fail-safe" principle. In perhaps the simplest example, it prohibits failing over to a plaintext connection attempt when an SSL connection cannot be established.

Unfortunately, the "secure-by-default" philosophy somewhat undermines usability since the user won't have immediate access to all available functionality. It is therefore incumbent on product documentation to clearly enumerate the security risks of enabling optional functionality, with UI alerts or warning messages underscoring the threat potential, forcing the user to explicitly agree to the desired action.

6.6 Defense in Depth

Redundant security mechanisms increase security. This is the principle of defense-in-depth. If one mechanism fails, perhaps another will still provide the necessary security. For instance, never rely totally on a firewall, especially for internal-use-only applications. A firewall alone rarely deters a determined attacker.

Of course, implementing a defense-in-depth strategy can add complexity, and one could rightly argue that increasing complexity increases risk. In all cases, the risk of complexity needs to be weighed against the benefits of added protection.

6.7 Principles for Reducing Exposure

Compartmentalization is one of these principles. It says that when one compartment or component of the system is attacked or becomes vulnerable, it can be sealed off from the other compartments. The principle of least privilege is another. It states that privileges granted to a user should be limited to only those necessary to do what the user needs to do and no more. Minimizing windows of vulnerability is another important security principle to reduce exposure. It says that when a risk must be introduced (i.e., no alternative exists), it should remain for as short a time as possible

6.8 The Insecure Bootstrapping Principle

While it is always best to avoid insecure links wherever possible, the principle of insecure bootstrapping says that if the system needs to use an insecure communication channel for some reason, it is used only to bootstrap a secure communication channel and for no other purpose. For example, the SSH protocol offers a secure channel after the client and server have authenticated each other. Since it doesn't use a public key infrastructure the first time the client connects, it generally will not have the server credentials. The server sends its credentials and the client blindly accepts that they're legitimate. Here, obviously, an attacker who can send his own credentials can masquerade as the server or launch an MITM attack. Fortunately, the SSH client remembers the initial credentials. If the credentials remain the same, and the first connection was secure, then subsequent connections are also secure. If the credentials change, the SSH client alerts the user to a potential attack underway.

6.9 Input Validation

Data input to a program is either valid or invalid. One inviolable security practice at Good is to definitively identify invalid data before any action on the data is taken. Levels of input validation include:

- **Use** – all places in the code where data, particularly data of external origin, is used.
- **Unit boundaries** – whenever data is passed between individual components, modules, or functions, whether pushed or pulled.
- **Trust boundaries** – whenever and wherever an executable is invoked.
- **Protocol parsing** – whenever the network protocol gets interpreted.
- **Application entry points** – either just before or just after passing data to an application, e.g., a validation engine in a web server for a web service.

- **Network** – via a traditional intrusion detection and prevention system (IDPS) that can consist of a network intrusion detection system (NIDS), a host-based intrusion detection system (HDS), a stack-based intrusion detection system (SIDS), or system-specific detection using custom tools and honeypots.

6.10 Security Ethics

First and foremost, Good's security ethos dictates that users exposed to security risks will be clearly informed of those risks and the recommended mitigation strategies, even though such risks may not be obvious to the user, either initially or at some later date.

Ethics further prescribe that users will be provided with the specific privacy policy governing use of their personal and/or confidential information in a timely manner so that they can act to avoid undesired use of that information. Additionally, when the privacy policy changes, the user will be given the explicit choice of accepting the change or having his/her personal data expunged.

If a system is compromised on which user data resides, users will be informed of the breach of privacy. When the data resides in the state of California, this is required by law. Similar regulations may apply in other jurisdictions.

* * * * *

Applying these governing principles, both general and specific security requirements are generated for each product, system, and application developed by Good and its approved ISVs.

7 High-Level Security Requirements

Throughout the SDLC, security requirements explicitly define and address the security goals and objectives of both the company and its customers. For each characteristic of the software security profile defined by the security architecture team, general requirements have been established which are continually being refined/expanded pertinent to new software features (functional and non-functional) entering the development pipeline that merit updates to the existing threat model, including:

- Confidentiality
- Integrity
- Availability
- Authentication
- Authorization
- Auditing
- Session Management
- Errors and Exceptions Management
- Configuration Parameters Management
- Sequencing and Timing
- Archiving
- Internationalization
- Deployment Environment
- Procurement
- Antipiracy and Anti-tampering

7.1 Confidentiality Requirements

Confidentiality requirements are those that address protection against disclosure of information that is either personal or considered sensitive in nature to unauthorized individuals. The classification of data into sensitivity levels is used to determine confidentiality requirements. Data can be broadly classified into public and private (nonpublic) information. The two most common confidentiality protection mechanisms

are (a) encryption and hashing, and (b) masking. Masking is the weaker form in which the original information is either X'd out or asterisked (commonly with the bullet symbol) primarily to protect against over-the-shoulder surfing attacks.

Independent of the mechanism(s) used, confidentiality requirements need to be defined throughout the information life cycle from the origin of the data in question to its retirement. This means it is necessary to explicitly state confidentiality requirements for nonpublic data in terms of:

- **In transit** – when data is transmitted over unprotected networks
- **In processing** – when the data is held in computer memory or media for processing
- **In storage** – when the data is at rest, within transactional as well as non-transactional systems, including archives.

Confidentiality requirements may be time-bound, meaning some information may require protection only for a certain period of time, until after it has been otherwise released into the public domain and is no longer deemed sensitive.

7.2 Integrity Requirements

Integrity requirements address reliability assurance and protection/prevention against unauthorized modifications. These security requirements refer not only to system/software modification protection (system integrity), but also to any data that the system/software handles (data integrity). In addition to reliability assurance, integrity requirements are meant to provide controls assuring that the accuracy of the system and data is maintained.

Methods include input validation, parity bit checking, cyclic redundancy checking (CRC—commonly referred to as *checksum*), and hashing. Integrity requirements generally dictate that:

- All input forms and query string inputs are validated against a set of allowable inputs before the software accepts it for processing.
- Published software provides the recipient with a computed *checksum* and the hash function used to compute the *checksum* so that the recipient can validate its accuracy and completeness.
- All nonhuman actors such as system and batch processes are identified, monitored, and prevented from altering data as it passes over systems it runs on, unless explicitly authorized to do so.

Taken fully into account as requirements at all times are the reliability, accuracy, completeness, and consistency aspects of all associated systems and data for ensuring integrity in the product software built or acquired.

7.3 Availability Requirements

Despite the concept of availability being traditionally relegated to business continuity and disaster recovery rather than security, it should be recognized that improper software design and development can lead to data destruction or inadvertently cause a DoS to authorized users. Consequently, availability requirements must be explicitly determined to ensure that there is no disruption of service for the user or the customer's business operations.

When determining availability requirements, the maximum tolerable downtime (MTD) and corresponding recovery time objective (RTO) must both be specified. MTD measures the minimum availability required for business operations to continue without unplanned disruptions. But, because all software will fail at some point, the RTO targets the maximum time interval that can pass before system execution is restored to the expected state of operations for authorized users. As a rule, both MTD and RTO should be explicitly stated in all Good service level agreements (SLAs). To arrive at these metrics, a business impact analysis (BIA) must be conducted in the absence of stress and performance testing on software entering development.

The BIA can either measure the loss of revenue for each minute the software is down, the cost to fix and restore the software to normal operations, or fines that are levied on the business upon any software

security breach. It can also be measured qualitatively, based on loss of credibility, confidence, and loss of brand reputation. BIA can be conducted on both new and existing versions of the software. Where the software (feature or functionality) already exists, the stress and performance test results from the previous version can be used to determine the new high-availability requirements.

At Good, downtime is measured in "nines" as shown in the following table:

Table 8: High-Availability Measures of Nines

Measurement	Availability *(%)*	Downtime per year	Downtime per month *(30 days)*	Downtime per week
Three nines	99.9	8.74 hours	43.2 min	10.1 min
Four nines	99.99	52.4 min	4.32 min	1.01 min
Five nines	99.999	5.24 min	25.92 sec	6.05 sec
Six nines	99.9999	34.45 sec	2.59 sec	0.605 sec

Understanding the impact of failure due to a breach of security is vitally important in determining availability requirements. End-to-end configuration requirements included in threat modeling must ensure that there is no single point of failure. A single point of failure is characterized by having no redundancy capabilities. In addition to end-to-end configuration requirements, load balancing requirements must be identified and captured as well. Replication is achieved with a master-slave scheme in which updates are propagated to the slaves actively or passively, with continuing consideration given to the integrity requirements of the data replicated.

Thus, Good's availability requirements specify that:

- All software ensures high availability of at least five nines (99.999), as defined in the SLA.
- The number of users at any given point in time able to use the software can be up to *n,nnn,nnn.*
- Software and data can be replicated across data centers to provide load balancing and redundancy.
- Mission critical functionality in the software can be restored to normal operations within 1 hour of disruption; mission essential functionality within 4 hours; and mission support functionality restored to normal operations within 24 hours of disruption.

Mission critical functionality is that for which loss, disruption or failure results in the failure of business operations. *Mission essential functionality* is functionality whose failure or disruption will cause the failure of the system (product or device). Mission support functionality is characterized as non-essential functionality that ranges from (a) optional feature APIs to (b) functionality supporting third-party plugins to (c) OTA provisioning of new devices to (d) a service interruption of Good's online Knowledgebase.

7.4 Authentication Requirements

Authentication is the process of validating an entity's identity claim by validating and verifying a submitted credential against a trusted source holding those credentials. Authentication requirements are those that verify and assure the legitimacy and validity of the entity presenting an identity claim for verification.

Of course, it is important to determine any need for two- or multifunction authentication. In all cases, it is wise to leverage existing/proven authentication mechanisms. Any requirement calling for a custom authentication process must be carefully scrutinized to ensure that no new risks would be introduced.

The most common forms of authentication are:

- **Anonymous** – no authentication check for validating the entity; prohibited in Good products.
- **Basic** – HTTP 1.0 Base-64 encoded user credentials; avoided at Good because the encoded credentials can be easily decoded.

- **Digest (hash value)** – is a challenge-response mechanism that sends a hash value (message digest) of the original credential; wherein the hash value of what was previously established is compared to what is currently supplied using a unique hardware property—one that cannot be easily spoofed—as a salt to calculate the digest.
- **Integrated** – or NTLM (for NT LAN Manager; also known as NT challenge-response authentication), also sends the credentials as a hash value and can be implemented as a standalone mechanism or in conjunction with Kerberos v5 when delegation and impersonation is necessary in a trusted subsystem infrastructure.
- **Client Certificate** – digital certificates (IT x.509v3) with the certification authority's SHA1 and MD5 fingerprints vouching for the validity of the holder, encrypting the data transmitted, typically via SSL.
- **Forms** – require the user to supply username and password which are validated against the active directory, a database, or a configuration file. Here, transmitted data is encrypted in addition to a transport layer security (TLS) implementation like SSL or a network security layer like IPsec.
- **Tokens** – usually used in conjunction with forms authentication, where, upon verification, a token is issued to the user granting access to the requested resources. This way, the username and password need not be passed on each call, which is particularly useful in single sign-on (SSO) situations.
- **Smart Cards and Fobs** – the first is ownership-based authentication with cards containing a programmable embedded microchip used to store the user's credentials. In the same vein, one-time (dynamic) passwords (OTP) provide the maximum strength authentication because OTP tokens (key fobs) require two factors: knowledge and ownership, i.e., something you know and something you have. Like token-based authentication, users enter the credential information they know and are issued a PIN that is displayed on the token device, like an RFID they own. Because the PIN is not static and changes dynamically every few seconds, it makes it virtually impossible for a malicious attacker to steal authentication credentials.
- **Biometric** – very advanced, though becoming more and more commonplace, this form of authentication uses biological characteristics—something you are—as identity credentials, i.e., physical attributes like retinal patterns, facial features, voiceprint, handprints and fingerprints to verify the submitter's identity. Currently cost and hardware prohibitive in mobility applications, these mechanisms are typically deployed to restrict access in highly secure facilities and computer systems with national security implications.

Capturing the proper authentication requirements early on in the SDLC helps to mitigate serious security risks at a later stage, especially during software design and development.

7.5 Authorization Requirements

Layered upon authentication, authorization requirements confirm that an authenticated entity has the needed rights and privileges to access and perform actions on a requested resource.

Access control models are of five types:

- **Discretionary Access Control (DAC)** – restricts access to objects based on the identity of the subject using access control lists.
- **Nondiscretionary Access Control (NDAC)** – characterized by the system objectively enforcing security policies with tamper-proof mechanisms applied to all subjects.
- **Mandatory Access Control (MAC)** – access to objects is restricted to subjects based on the sensitivity of the information contained in the objects. It matches a subject's clearance level with the object's sensitivity level.
- **Role-based Access Control (RBAC)** – RBAC uses the subject's role to determine whether access should be allowed or not
- **Resource-based Access Control (RscBAC)** – can be broadly divided into:

- **Impersonation and Delegation Model** – propagating the identity of the primary entity to downstream systems. Kerberos uses this model to grant tickets that are delegated sets of permissions to invoke services downstream, acting as if it is the primary entity by impersonating the user identity.
- **Trusted Subsystem Model** – predominantly used in web applications, access request decisions are granted based on the identity of the resource that is trusted instead of user identities. In other words, it is not the user identity that is checked but the web application identity that is trusted and can invoke the call to the database.

Examples of authorization requirements that must be captured where appropriate include:

- Restricting access to highly sensitive secret files to users with secret or top secret clearance only.
- Not requiring users to send their credentials again after they have successfully authenticated themselves.
- Requiring that all unauthenticated users inherit read-only permissions that are part of the guest user role, while authenticated users default to having read and write permissions as part of the general user role. Only members of the administrator's role will have all rights as a general user in addition to having permissions to execute operations.

7.6 Auditing/Logging Requirements

Auditing requirements are those that assist in building the historical record of user actions. Such an audit trail helps detect when an unauthorized user makes a change or an authorized user makes an unauthorized change, both of which are integrity violations. At a minimum, auditing requirements include:

- **Who**: Identity of the subject (user process) performing an action
- **What**: Action
- **Where**: Object on which the action was performed
- **When**: Timestamp of the action.

What is logged and what is not is a decision of the product owner based on customer feedback, but, as a best practice for security, all critical business transactions and admin functions need to be identified and audited. Additional logging requirement examples include:

- Logging all failed logon attempts, along with the timestamp and IP address from which the request originated.
- Always appending audit logs, never overwriting.
- Securely retaining audit logs for a period of three (3) years.

7.7 Session Management Requirements

Sessions are useful for maintaining state but also have an impact on the secure design principals of complete mediation and psychological acceptability. Upon successful authentication, a session ID is issued to the user to track that user's behavior and maintain an authenticated state until the session is abandoned or the state changes to "not authenticated." In stateless protocols like HTTP, session state must be explicitly maintained and protected from brute force, predictable session ID attacks, and MITM hijacking. In short, session management security requirements ensure that, once established, the session remains in a state that will not compromise the software.

Session management specifications should always include provision for:

- Uniquely tracking each user action.
- Once authenticated, not requiring the user to provide credentials again during the session.
- Explicitly abandoning the session when the user logs off, closes the application, or turns off the device.
- Never passing session IDs in clear text or any easily guessable format.

7.8 Errors and Exception Management Requirements

Because errors and exceptions are potential sources of information disclosure, verbose error messages and unhandled exception reports can result in divulging sensitive internal application architecture, design, and configuration information. Recommended error and exception security requirements include:

- Explicitly handling exceptions using try, catch, and block.
- Displaying error messages to the end user that reveal only necessary information; no internal system error details are disclosed to the user for any reason.
- Monitoring and periodically auditing security exception details.

7.9 Configuration Parameters Management Requirements

Config parameters and code must be protected from hackers. Because these parameters and code initialize before the software can run, identifying and capturing the form and value range of the settings is vital to assure that the appropriate level of protection is considered during software design and development. The minimum requirements include:

- Encrypting sensitive DB connect settings and other sensitive application settings in web application Config resources.
- Never hard-coding passwords in line code.
- Carefully and explicitly monitoring initialization and disposal of global variables.
- Including protection of configuration information in application and/or OnStart and OnEnd events as a safeguard against disclosure threats.

7.10 Sequencing and Timing Requirements

Sequencing and timing flaws can cause race conditions or time-of-check/time-of-use (TOC/TOU) attacks. Common sources of race conditions are:

- Unintended sequence of events
- Multiple unsynchronized threads executing simultaneously
- Infinite loops preventing a program from returning control to the normal process flow.

In general, race conditions occur when two threads execute concurrently, both accessing the same object at the same time, with one thread altering the state of the shared object. Single flow of control—often called atomic operations—will obviate concurrency, sacrificing a degree of performance for integrity and security. Race conditions can also be eliminated by resource locking, wherein the object being accessed does not allow any alteration until the first thread or process releases it. This is called Mutex, for mutual exclusion.

7.11 Archiving Requirements

It is important to establish that organizational retention policy, especially regarding sensitive or private information, does not contradict regulatory requirements, which may vary from country to country. Where conflicts arise, the regulatory requirements take precedence, and all SDLC process-related data and information must be stored and archived until such time as required by regulatory and company policy.

Hence, is it absolutely essential that archiving requirements are part of the required documentation and not overlooked when designing and developing software.

7.12 Internationalization Requirements

Character encoding and display direction are the two most important internationalization factors. The character encoding standard adopted not only defines the identity and code point of each character but also how the value is represented in bits. Unicode is the only universal character encoding standard that is fully compatible with ISO/IEC 10646 and includes UTF-8, UTF-16, and UTF-32. The appropriate and correct character encoding must be set in the software to prevent Unicode security issues like spoofing,

overflows, and canonicalization, which is the process of converting data that has more than one possible representation into a standard canonical form.

In addition to character encoding, it is equally important to determine display direction requirements, which should be explicitly identified and included in all UI/UX stories.

7.13 Deployment Environment Requirements

Since production environments are often if not usually configured differently than development and test environments, restrictions including ports and protocols, network segmentation, disabled services, and components must be considered. Special or customer-centric infrastructure, platform, and host security restrictions must be elicited, ideally early on. Implementation of clustering and load balancing mechanisms can likewise have an impact on software design, so these architectures need to be identified and accommodated, either as one-offs or using configuration switch options. Identifying and capturing constraints, restrictions, and requirements of the environment(s) in which the software is expected to operate, in advance, will considerably reduce deployment challenges later, helping to assure that the product will be deployed and function as designed.

7.14 Third-Party Software Procurement Requirements

Establishing software security requirements is even more essential when procuring software instead of building it in-house. Sometimes the requirement definition process itself leads to a buy decision. Thus, as a matter of legal policy, security requirements as protection mechanisms must be included in all third-party software purchase contracts and SLAs in addition to software escrow.

Software escrow is the act of having a copy of the source code for the implemented third-party or ISV software held in the custody of a neutral escrow agency/party. This insures that, as the licensee, Good is protected against loss of use of mission-critical software if the third-party software publisher fails for any reason, including going out of business. It also protects the third-party software publisher against illegal, reverse-engineered copies of its code. In all cases, determination of a breach is established by comparing the software in question to the copies and versions held in escrow; usually both source and object code along with appropriate documentation for each version.

7.15 Antipiracy and Anti-tampering Requirements

All code needs to be protected from unauthorized modification. Identifying and specifying applicable code shrouding, code signing, anti-tampering, licensing, and IP protection mechanisms must also be included during requirements gathering, since they can be considerably more difficult and costly down the road.

Although currently discouraged in Good products, source code anti-tampering can be achieved using obfuscation, or shrouding, a process of making the code obscure and confusing using a special program called the obfuscator so that, even if the source code is leaked to or stolen by an attacker, it is not easily readable and decipherable. Shrouding merely entails complicating the code with generic variable names and renaming text and symbols within the code to meaningless character sequences. Shrouding need not be limited to source code, either. When object code is shrouded, it helps deter reverse engineering.

Reversing is analogous to going backward in the SDLC. While it can be used for legitimate purposes, especially in cases where the documentation has been lost or is otherwise unavailable, skillful attackers can use reverse engineering to crack the software and circumvent security protections. They can also tamper and repackage the software with malicious intent.

In addition to code shrouding and depending on the application, removing symbolic information from the program executable (PE) and embedding anti-debugger code are both worthy countermeasures. This entails the removal of class names, class member names, names of global instantiated objects, and other textual information from the PE by stripping them out before compilation. A user- or kernel-level debugger detector can then be embedded in the code to terminate the process when a debugger is found. **`IsDebuggerPresent`** and **`SystemKernelDebuggerInformation`** APIs are examples.

Code signing is the process of digitally signing the code (executables, scripts, etc.) with the digital signature of the code author, using private and public key systems. Each time code is built, it can be signed immediately or just before deployment. Any alteration of the code will result in a hash value no longer matching the hash value that was published.

In summary, Good's security requirements frame-out the general design and deployment decisions made before any new feature of product component enters development. Specific implementation details are arrived at during the define-build-test (Scrum) phase of the SLDC.

8 Common Software Vulnerabilities and Controls

Secure software results from the confluence of people, process, and technology. Software industry analysis of security breaches invariably identifies one of the following to be the root cause of the breach:

- Design flaws
- Coding/implementation issues
- Improper configuration and operation.

These are more specifically itemized in the Open Web Application Security Project (OWASP) Top 10 List, Mitre/SANS Institute Common Weakness Enumeration (CWE) Top 25 List of the most dangerous software programming errors, and, specifically regarding Good's product line, the OWASP Top 10 Mobile Risks, as well as to a lesser extent the OWASP Top 10 Cloud Computing Risks. Each is covered in turn as follows.

8.1 OWASP Top 10

In addition to considering the most common application security issues from a weakness/vulnerability perspective, the OWASP Top 10 List views application security issues from a technical risk and business impact perspective, comprising:

1. **Injection flaws** – such as SQL, OS, and LDAP injection, occurring when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands and accessing unauthorized data.
2. **Cross-site scripting (XSS)** – occurring whenever an application takes untrusted data and sends it to a web browser without proper validation and escaping. XSS allows attackers to execute script in the victim's browser that can hijack user sessions, deface web sites, or redirect the user to malicious sites.
3. **Broken authentication and session management** – occurring when authentication and session management are incorrectly implemented. This allows attackers to compromise passwords, keys, and session tokens; and to exploit implementation flaws to assume other user's identities.
4. **Insecure direct object reference** – occurring when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other protection, attackers can manipulate these references to access unauthorized data.
5. **Cross-site request forgery (CSRF)** – forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other authentication information, to a vulnerable web application, allowing the attacker to force the victim's browser to generate requests that the vulnerable application thinks are legitimately coming from the victim.
6. **Security misconfiguration** – occurring when the secure configuration defined for the application, framework, web server, and platform is poorly defined, implemented, and/or maintained; applications should always be shipped with secure defaults.
7. **Failed to restrict URL access** – occurring because checking URL access rights before rendering protected links and buttons is only partially effective. Applications need to perform similar access control checks when the pages are actually accessed or attackers will be able to forge URLs to access these hidden pages anyway.

8. **Unvalidated redirects and forwards** – occurring when applications redirect and forward users to other pages and sites and use untrusted data to determine the destination pages. Without proper validation, attackers can redirect victims to phishing or malware sites or use forwards to access unauthorized pages.
9. **Insecure cryptographic storage** – occurring when applications do not properly protect sensitive data like credit card numbers, SSNs, and authentication credentials with appropriate encryption or hashing. Attackers can use this weakly protected data to conduct identity theft, credit card fraud, and other crimes.
10. **Insufficient transport layer protection** – occurring when transport layer protection is limited to certain operations, like authentication, but end-to-end transport layer protection is absent, or when the application fails to encrypt network traffic to protect sensitive communications.

8.2 CWE Top 25

The Common Weakness Enumeration (CWE) Top 25 List of the most dangerous software programming errors is grouped into three major categories as follows:

- **Insecure interaction between components**: including weaknesses that relate to insecure ways in which data are sent and retrieved between separate components, modules, programs, processes, threads, or systems.
- **Risky resource management**: including weaknesses that relate to ways in which software does not properly manage the creation, usage, transfer, or destruction of important system resources.
- **Porous defenses**: including weaknesses that relate to defensive techniques which are too often misused, abused, or just plain ignored.

The complete list, available at http://cwe.mitre.org/data/definitions/862.html with more complete information on causation and mitigation/resolution, includes:

1. **(CWE-89) SQL Injection: Improper Neutralization of Special Elements used in a SQL Command** – software constructs all or part of a SQL command using externally-influenced input from an upstream component, but does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component.
2. **(CWE-78) OS Command Injection: Improper Neutralization of Special Elements used in an OS Command** – software constructs all or part of an OS command using externally-influenced input from an upstream component, but does not neutralize or incorrectly neutralizes special elements that could modify the intended OS command when it is sent to a downstream component.
3. **(CWE-120) Classic Buffer Overflow: Buffer Copy without Checking Size of Input** – program copies an input buffer to an output buffer without verifying that the size of the input buffer is less than the size of the output buffer, leading to a buffer overflow.
4. **(CWE-79) XSS: Improper Neutralization of Input during Web Page Generation** – software does not neutralize or incorrectly neutralizes user-controllable input before it is placed in output that is used as a web page that is served to other users.
5. **(CWE-306) Missing Authentication for Critical Function** – software does not perform any authentication for functionality that requires a provable user identity or consumes a significant amount of resources.
6. **(CWE-862) Missing Authorization** – software does not perform an authorization check when an actor attempts to access a resource or perform an action.
7. **(CWE-798) Use of Hard-coded Credentials** – software contains hard-coded credentials, such as a password or cryptographic key, which it uses for its own inbound authentication, outbound communication to external components, or encryption of internal data.
8. **(CWE-311) Missing Encryption of Sensitive Data** – software does not encrypt sensitive or critical information before storage or transmission.

9. **(CWE-434) Unrestricted Upload of File with Dangerous Type** – software allows the attacker to upload or transfer files of dangerous types that can be automatically processed within the product's environment.
10. **(CWE-807) Reliance on Untrusted Inputs in a Security Decision** – application uses a protection mechanism that relies on the existence or values of an input, but the input can be modified by an untrusted actor in a way that bypasses the protection mechanism.
11. **(CWE-250) Execution with Unnecessary Privileges** – software performs an operation at a privilege level that is higher than the minimum level required, creating new weaknesses or amplifying the consequences of other weaknesses.
12. **(CWE-352) Cross-Site Request Forgery (CSRF)** – web application does not, or cannot, sufficiently verify whether a well-formed, valid, consistent request was intentionally provided by the user who submitted the request.
13. **(CWE-22) Path Traversal: Improper Limitation of a Pathname to a Restricted Directory** – software uses external input to construct a pathname that is intended to identify a file or directory that is located underneath a restricted parent directory, but the software does not properly neutralize special elements within the pathname that can cause the pathname to resolve to a location that is outside of the restricted directory.
14. **(CWE-494) Download of Code Without Integrity Check** – product downloads source code or an executable from a remote location and executes the code without sufficiently verifying the origin and integrity of the code.
15. **(CWE-863) Incorrect Authorization** – software performs an authorization check when an actor attempts to access a resource or perform an action, but it does not correctly perform the check. This allows attackers to bypass intended access restrictions.
16. **(CWE-829) Inclusion of Functionality from Untrusted Control Sphere** – software imports, requires, or includes executable functionality (such as a library) from a source that is outside of the intended control sphere.
17. **(CWE-732) Incorrect Permission Assignment for Critical Resource** – software specifies permissions for a security-critical resource in a way that allows that resource to be read or modified by unintended actors.
18. **(CWE-676) Use of Potentially Dangerous Function** – program invokes a potentially dangerous function that could introduce vulnerability if it is used incorrectly, but the function can also be used safely.
19. **(CWE-327) Use of Broken or Risky Cryptographic Algorithm** – use of a broken or risky cryptographic algorithm results in the exposure of sensitive information.
20. **(CWE-131) Incorrect Calculation of Buffer Size** – software does not correctly calculate the size to be used when allocating a buffer, which could lead to a buffer overflow.
21. **(CWE-207) Improper Restriction of Excessive Authentication Attempts** – software does not implement sufficient measures to prevent multiple failed authentication attempts within a short time frame, making it more susceptible to brute force attacks.
22. **(CWE-601) Open Redirect: URL Redirection to Untrusted Site** – web application accepts a user-controlled input that specifies a link to an external site, and uses that link in a Redirect; this simplifies phishing attacks.
23. **(CWE-134) Uncontrolled Format String** – software uses externally-controlled format strings in print-style functions, which can lead to buffer overflows or data representation problems.
24. **(CWE-190) Integer Overflow or Wraparound** – software performs a calculation that can produce an integer overflow or wraparound when the logic assumes that the resulting value will always be larger than the original value. This can introduce other weaknesses when the calculation is used for resource management or execution control.
25. **(CWE-759) Use of a One-Way Hash without a Salt** – software uses a one-way cryptographic hash against an input that should not be reversible, such as a password, but the software does not also use a salt as part of the input.

8.3 OWASP Top 10 Mobile Risks

The OWASP Mobile Security Project is a centralized resource intended to give developers and security teams the resources they need to build and maintain secure mobile applications. The goal is to classify mobile security risks and provide developmental controls to reduce their impact or likelihood of exploitation.

The primary focus is at the application layer, taking into consideration the underlying mobile platform- and carrier-inherent risks when threat modeling and building controls, targeting the areas where the average developer can make a difference. Additionally, the focus is not only on the mobile applications deployed to end user devices, but also on the broader server-side infrastructure which the mobile apps communicate with, focusing heavily on the integration between the mobile application, remote authentication services, and cloud platform-specific features. The top 10 risks identified so far are:

M1. **Insecure Data Storage** – sensitive data, locally stored plus cloud-synched, is left unprotected as a result of not encrypting the data, caching data not intended for long-term storage, weak or global permissions, or not leveraging platform best practices.

M2. **Weak Server-Side Controls** – server controls inadequately prevent OWASP Web Top 10, OWASP Cloud Top 10, and OWASP Web Service Top 10.

M3. **Insufficient Transport Layer Protection** – limited to certain operations, like authentication, but end-to-end transport layer protection is absent, or when the application fails to encrypt network traffic to protect sensitive communications.

M4. **Client-Side Injection** – apps using browser libraries susceptible to XSS, HTML injection, and/or SQL injection, abusing phone dialer + SMS, or abusing in-app payments because untrusted data was not sanitized or escaped before rendering or execution.

M5. **Poor Authorization and Authentication** – part mobile, part architecture, when apps rely solely on immutable, potentially compromised values (IMEI, IMSI, UUID) and/or hardware identifiers persist across data wipes and factory resets, i.e., device ID or subscriber ID is used as sole authenticator.

M6. **Improper Session Handling** – apps maintain session via HTTP cookies, OAuth tokens or SSO authentication services, or using a device identifier and a session token, as well as not making users re-authenticate every so often.

M7. **Security Decisions Via Untrusted Inputs** – caller permissions not checked at input boundaries; can be leveraged to bypass permissions and security models—abusing URL schemes in iOS, abusing intents in Android—to open door to malicious apps or client-side injection.

M8. **Side Channel Data Leakage** – when inadequate understanding of what third-party libraries are doing with user data (ad networks, analytics), programmatic flaws combine with not disabling platform features to cause sensitive data to end up in unintended places, e.g., web caches, keystroke logging, screenshots (iOS backgrounding), logs (system, crash), and temp directories.

M9. **Broken Cryptography** – caused by broken implementation using strong crypto libraries or custom, easily defeated crypto implementations that do not leverage battle-tested crypto libraries.

M10. **Sensitive Information Disclosure** – private API keys on client allow apps to be reverse-engineered with relative ease, exposing API keys, passwords, sensitive business logic, and intellectual property.

8.4 OWASP Top 10 Cloud Security Risks

The aim of the OWASP Cloud-10 list is to help balance security risks with the cost advantage that the Cloud and Software-as-a-service (SaaS), Platform-as-a-Service (PaaS), and Infrastructure-as-a-Service (IaaS) models provide. The Top 10 Cloud Security Risks currently identified include:

R1. **Data Ownership and Protection** – using a public cloud for hosting a business service results in loss of control of data without guaranteed backup and recovery. When an organization uses the cloud, the data resides on the cloud provider's resources, creating critical security risks that must be carefully understood and mitigated.

R2. **User Identity Management and Federation** – inadequate control of user identities as services and applications are moved to different cloud providers. Additionally, through integration, existing in-house logon accounts managed by the organization can be used to access data in the cloud. Typically, this integration is achieved via LDAP mechanisms or using SSO technologies like SAML. While integration offers a substantially better user experience and avoids many of the risks associated with separate islands of credentials, integrations carry varied risks as well, including exposing the organization's authentication systems and spoofing of SSO assertions using stolen private keys.

R3. **Regulatory Compliance** – data perceived to be secure in one country may not be perceived secure in another due to different regulatory laws across countries or regions. Even though protected information may be entrusted to a cloud provider, the organization utilizing the services of the cloud provider retains the ultimate responsibility for compliance with applicable laws and regulations. The organization's compliance responsibility encompasses its own internal business operations as well as ensuring compliant operations within the cloud service provider. For protected health information (PHI), depending on the nature of the services offered by the cloud provider, the provider may be considered a covered entity and/or business associate. A cloud provider that is a covered entity should be able to offer evidence of compliance with government-enforced security and privacy rules.

R4. **Business Continuity and Resiliency** – refers to the ability of an organization to conduct business operations in adverse situations, which include disruptions not only to information technology infrastructure, but also any disruptions affecting the ability of the cloud service provider to deliver its services at defined service levels, including, for example, the loss of key personnel or the loss of access to business offices. When an organization uses a cloud provider, the organization cedes control of business continuity planning for the data and services entrusted to the cloud provider. Consequently, the organization must carefully consider the ability of the cloud provider to provide continuity of service when adverse situations arise. A degraded service level poses risks such as longer response times for access to data or services. Recovery targets must be based on the time sensitivity of business functions to loss of service.

R5. **User Privacy and Secondary Uses of Data** – many social application providers mine user data for secondary usage, e.g., directed advertising, with vague controls over the extent to which a cloud provider can disclose information about its using organizations and customers. Secondary uses of data refer to uses of information collected by a cloud provider about a subscriber for purposes other than the provision of services to the cloud subscriber.

R6. **Service and Data Integration** – risk of interception of data in transit is much greater for organizations utilizing a cloud computing model, where data is transmitted over the Internet. Unsecured data is susceptible to interception and compromise during transmission. Many of these risks are mitigated by the proper use of SSL/TLS ciphers, IPSec, VPNs, SSL VPNs, and properly implemented PKI.

R7. **Multi-tenancy and Physical Security** – in a cloud situation, this means sharing of resources and services among multiple clients (CPU, networking, storage/databases, application stack), increasing dependence on logical segregation and other controls to ensure that one tenant deliberately or inadvertently cannot interfere with the security (confidentiality, integrity, availability) of other tenants. Isolation approaches include use of virtualization technologies such as virtual machines, application-level isolation through processes, threads, or application-managed contexts, and database-level isolation through the use of separate database instances, tablespaces, or record identifiers.

R8. **Incidence Analysis and Forensic Support** – in the event of a security incident, applications and services hosted by a cloud provider are difficult to investigate as logging may be distributed

across multiple hosts and data centers which could be located in various countries and hence governed by different laws. Also, along with log files, data belonging to multiple customers may be co-located on the same hardware and storage devices and hence a concern for law enforcement agencies for forensic recovery. Organizations must carefully evaluate the tools available from the cloud provider for conducting investigations as security incidents may require the prompt collection of evidence for possible civil or criminal proceedings.

R9. **Infrastructure Security** – infrastructure must be hardened and configured securely, and the hardening/configuration baselines should be based on industry best practices. Applications, systems and networks must be architected and configured with tiers and security zones, with access configured to only allow required network and application protocols. Administrative access must be role-based, and granted on a need-to-know basis. Regular risk assessments must be done, preferably by an independent party. A policy and process must be in place for patching/security updates based on risk/threat assessments of new security issues.

R10. **Non-production Environment Exposure** – any organization that develops software applications internally employs a set of non-production environments for design, development, and test activities. The non-production environments are generally not secured to the same extent as the production environment. If an organization uses a cloud provider for such non-production environment, then there is a high risk of unauthorized access, information modification, and information theft.