Secure Software Foundation

Framework Secure Software

Authors:	T.C. Hemel and J.A. de Vries
Audience:	Working group Framework Secure Software
File:	
Date:	2014-05-27
Version:	1.0
Status:	Approved
Classification:	Public
Code:	

Secure Software Foundation

Table of Contents

1 Document Properties

1.1 Availability

Name:
Location:
Telephone:
Email:
URL:

1.2 Revisions

Date	Author	Version	Changes
2014-05-14	TH	0.1	Initial version
2014-05-19	HdV	0.2	Review + added sections on consolidation
2014-05-27	TH + HdV	1.0	First public version

1.3 Signature List

Name	Title	Role	Date	Signature

1.4 Distribution List

Name	Title	Role	Version	Medium/Format

2 A Framework for Secure Software

2.1 Introduction

In his book Geekonomics, author David Rice investigates why the problem of insecure software exists, and offers solutions to this problem ([Rice]). Drawing a parallel with the automobile industry, he identifies two causes: purchasers cannot see if software is secure and therefore decide to buy software based on the biggest perceived value (i.e. the highest number of features) for the lowest price, and suppliers are not liable for any mistake in the software that they make.

The obvious solutions to this are a secure software certificate that helps potential purchasers in judging whether the software meets their security needs, and liability of the software industry to create sufficient incentive to add proper security to their products. In the former case, market forces can resolve the problem. In the latter a significant reform of the legal system is required and even then this might result in a lot of litigation that will not necessarily improve the security of software itself. It is clear that the former solution should be the preferred one, for all parties involved.

But how does one certify the security of software? No clear standard of doing this was available, so there were no common ground rules. Therefore the Secure Software Foundation developed a framework aimed at defining a method and *Controls* that can be used to “measure” the security of software in an objective and repeatable manner. How this method and these Controls are to be applied is described in this document.

2.2 Comparison with Other Frameworks

A number of frameworks for security have been in use for a long time, each with its own focus and objectives. On one end of the spectrum the ISO 27000 series focuses on information security within organizations and is very abstract, albeit organization specific. On the other hand of the spectrum a large number of secure coding standards contain very concrete rules, but these are not tailored to a specific software system. In between are various frameworks such as OWASP's Application Security Verification Standard (ASVS), the OWASP top 10, Microsoft's Secure Development Life Cycle (SDL), and maturity models such as BSIMM and OpenSAMM. And these are just the generic ones. In the Netherlands, for example, the National Cyber Security Center (NCSC) developed its own set of guidelines, and the Centrum Informatiebeveiliging en Privacybescherming (CIP) developed a framework to help with setting security requirements.

However none of these focus on providing a manner in which software can be certified, although some of these frameworks have been used as the basis for compliance criteria. There is of course the Common Criteria evaluation of a software system's security, but with its relatively ponderous process and high costs, only organizations with sufficient resources can afford this. Despite all these frameworks, there is still a lack of secure software.

The same security invisibility that exists for software purchasers also exists for project managers and developers in a software development organization. If security is thought of at all, it is usually only late in the development life cycle. This means one of the keys to getting more secure software is to increase the visibility of security during *all* development phases. This framework

does exactly that. For software purchasers by providing a process for certification, and for development organizations by providing guidelines to build secure software systems from the start.

2.3 Objectives of the Framework

In an ideal world the Framework Secure Software would try to answer the utopian question: “Is this software system completely secure?”. Unfortunately, it is impossible to give that answer. The very definition of secure (“protected against threats”) implies that one would need to know all possible threats against the software system to answer this question, which cannot be guaranteed. However, if the question is rephrased as “Is this software system protected against all known threats?”, answering that question becomes easier. The framework and the associated certificate will and cannot not guarantee that a software system is completely secure, but they will give the assurance that security has been sufficiently implemented.

To answer that question, one needs to know what the known threats are for a software system, and when protection against these threats is sufficiently implemented.

In a product as diverse as software, there is no set of threats that applies to all software systems and that is valid under all circumstances. Threats to a software system depend on the type of software system, how it is built, the environment in which it is used and how it is used. This context will determine the threats that are applicable to a specific software system.

Once the threats are defined, it becomes possible to verify if the software system is sufficiently protected against these threats. This protection is realized by a secure design, by implementing mitigation measures, by coding securely, and by not making mistakes when doing this. The Framework Secure Software aims to provide a secure software development process that ensures correctness and completeness as much as possible, and that can be evaluated to make sure it has been applied appropriately

To achieve this, essential security practices throughout the Software Development Life Cycle (SDLC) were observed and criteria to evaluate the results of those practices were created. By covering the full SDLC and connecting the essential security practices, it is possible to bridge the gap between the software purchaser's abstract idea of security and the developer's concrete understanding of a secure implementation. By focusing on evaluating results instead of process as much as possible, it becomes possible to evaluate the security of a software system in a more objective way. This is where the real value of the Framework Secure Software is to be found compared to other frameworks: *it measures the security of the result of a development process*, namely the produced software itself. Other frameworks mostly focus on good processes, hoping these will result in secure software, but they don't provide a method to verify this in a manner that can be performed in an objective and reproducible way by a developer or auditor.

The result of applying the Framework Secure Software is an intrinsically secure and traceable development process, and a method that allows software purchasers and software developers to communicate about software security in a common language both can understand.

In order to implement this process and evaluate whether the process is implemented correctly as a software system is developed, the development organization needs to have the right knowledge and to have an organized development process. Therefore the Framework Secure

Software also contains Controls regarding knowledge of security and the existence of a properly focused development process. This is necessary, because these two aspects of secure software development make sure a good result is not just a “lucky shot”, but something a development organization can achieve consistently.

This framework respects the different methodologies to develop software and will therefore not give you a step-by-step guide prescribing a specific software development process. Software development teams will have to integrate the practices of this framework into their existing software development process. The practices described here are methodology-agnostic and fit the modern agile as well as the more traditional software development processes.

To recapitulate, the objectives of the Framework Secure Software are to provide a method:

- that makes sure software is developed securely and is done so consistently;
- that defines criteria to measure the security of the produced software itself;
- that verifies in an objective and repeatable way those criteria are properly implemented
- that can be performed by different evaluators at different moments, but results in the same findings;
- that can be applied independently of the development process a development organization is already using;
- and that is applicable independent of the type of software produced.

2.4 Applying the Framework

Within this framework several parties are being served.

A *certificate requestor* is the party that requests to evaluate a software software system. This could be the party that supplies the software, the party for which the software was built, or a representative of the users of the software. A certificate requestor agrees to the security context that will define the scope of applicability of the certificate. This context states what the software system does, what security requirements need to be met and the security assumptions that are important to know of when the software system is used. Certificate requestors will find the relevant details to their needs in sections 3.2 and 4.2.

A *developer*[1] needs to know what to build and how to build this securely. This requires security expertise, which can be obtained from an external party if it is missing. From the security context the developer identifies threats to the software system, assesses their *impact*, and designs proper *mitigations*. To implement these mitigations securely, a *secure coding standard* is used. To verify the implementation is indeed secure, the developer applies verification techniques, such as tests and reviews. The result of these practices is documented to keep track of the software system's security status and to be used for later evaluation.

This practices fit both agile and more traditional development methodologies, but every organization will need to find a way to implement them in a way that suits them best. Developers will find the relevant details to their needs in chapters 3 and 4.

An *auditor* needs to evaluate the software development team's knowledge, the software

1 In this framework “developer” stands for the development organization, not just an individual programmer.

development process and the resulting product in order to judge their security and maturity. The auditor will use a defined set of Controls to evaluate the core deliverables that are an intrinsic part of a secure product, being the security context, the list of threats, the selected mitigations, the chosen verifications, and the implementation (code and configuration) of the software system itself. Auditors will find the relevant details to their needs in chapters 3 and 4.

2.5 Reporting the Results

The Framework Secure Software recognizes the fact that a security evaluation of a software system might provide unauthorized parties with too much knowledge about potential vulnerabilities therein. On the other hand certificate requestors will need sufficient information to decide whether that software system is secure enough to meet their specific security needs.

The solution according to this framework is to split the results of a security evaluation in two separate reports. A public report specifying the security requirements and assumptions, and a private report containing detailed results of the evaluation for use by the developer to improve the security of the software system where needed.

To further reduce the risk of publicizing information about potential vulnerabilities only those security requirements and assumptions that are defined for Application Assets[2] that are directly operated upon by users of the software system need to be part of the public audit report.

2.6 How this Document is Organized

Chapter 3 explains the foundations of the framework: the practices that need to be executed and evaluated. It describes in detail what to do and what artefacts must be available for evaluation, together with examples.

Chapter 4 lists the Controls that will be used by auditors to validate the security of any given software. Every Control comes with a short description, an explanation, and what is needed to evaluate it.

Chapter 5 describes how to make sure a secure piece of software was not just the result of a “lucky shot”, but of mature processes. Only when a proper level of maturity is available within the development organization will users/buyers of the software be able to trust future version of the software will also be secure. Without that trust a development organization might be forced to certify each release of some given software, which may become prohibitively expensive.

Chapter 6 lists the Controls that will be used by auditors to validate the maturity of the development organization. Again, every Control comes with a short description, an explanation, and what is needed to evaluate it.

2 See section 3.2.2.1 for a description of what Application Assets are.

3 Foundations

3.1 Introduction

The evaluation process of application security is divided into four phases.

In the *context phase*, the software system is described along with its desired security properties and assumptions. This is the basis for the rest of the evaluation and will be part of the public audit report.

The *threats phase* deals with identifying possible attacks against the software system and the associated mitigating measures against these threats.

In the *implementation phase*, the code and configuration of a software system is inspected.

The *verification phase* looks at how the development organization verifies whether the implementation really *is* secure. This should not be confused with the assessment that an auditor performs on the implementation.

In each phase, developers create something that an auditor can assess. The exact developer actions and audit criteria are described as Controls in chapter 4. This chapter will describe the evaluation process in more detail and elaborates on it with examples.

3.2 Context Phase

To answer the question whether a software system is secure, one needs to define what `secure' means.

It is impossible to tell what threats exists against a software system in general. Threats depend on the functionality of the software system, its architecture and implementation, and the environment in which it is used. To judge architecture and implementation security, a context is needed.

First, the functional and environmental requirements the software system has to meet must be described. To help in defining the security requirements a systematic approach to do this is given in section 3.2.2. Based on these functional and environmental requirements, security requirements can be defined that will define what `secure' means for a specific software system. This description of security at the level at which an end-user interacts with the software system is what forms the *security context*.

Someone who wants to know whether a software system is secure enough for their own needs must be able to determine this by reading the security context. For that reason the security context must be part of the public audit report that is associated with a *Secure Software Certificate*.

3.2.1 Functions and Environment

The functional and environmental requirements specify what the software system should do when it is used normally and is not under attack. A highly detailed description is usually not necessary for this, but the functions and environment description must at least contain the functionality that is directly visible to a user of the software system.

Legal, compliance and other externally imposed requirements must be mentioned as part of the environmental requirements. However, an auditor mostly will not evaluate whether the software system is compliant to these requirements, because they are highly dependent on factors that often have little to do with software security as such (e.g. the country in which the software is to be used). However when environmental requirements impact the technical security of the software system then security requirements must be derived from them.

If certain security requirements cannot be implemented because they are in conflict with environmental requirements, this must be added as an explanation to the security assumptions (see section 3.2.3).

3.2.2 Systematic Requirements Gathering/Analysis

The security requirements of a software system depend on its functional and environmental requirements. Functional requirements describe the data that is processed by the software system and how it is processed by the software system. Security requirements define what security properties need to be satisfied by the software system. Here a systematic approach to define these requirements is given, consisting of three steps:

1. Identify Application Assets
2. Translate Information Assets to operations
3. Analyze the STRIDE properties

3.2.2.1 Identify Application Assets

Application Assets are defined by information and actions that are present in the software system. From the functional and environmental requirements, one can identify what data is processed by the software system and what operations are performed on that data by the software system. An example of this could be data that can be manipulated directly by the software system itself, such as items (information objects) that can be put (an operation) into a shopping cart (an other information object). Meta-data, such as the transaction history of a user, can also be subject to security requirements, as can system logs, passwords and secret keys. A good guideline to identify Application Assets is given in [Shostack] (pp.37):

- Information/actions an attacker wants
- Information/actions you want to protect
- Stepping stones to either of these

Information Assets are a subset of Application Assets and are made up of the information objects that are found in an application.

3.2.2.2 Translate Information Assets to Operations

Information in a software system does not exist in isolation, but is processed by the software system through operations. For every Information Asset one can therefore identify what operations on it exist in the software system. Most Information Assets can be associated with the generic create, read, update, delete (CRUD) operations known from database theory. After all, data must be created at some point, can be read, perhaps updated and finally deleted.

Not all of these CRUD operations may apply to a particular Information Asset, but having to review them all will force you to think of how data should (desired behavior) or could (undesired behavior) be created, read, updated or deleted. Otherwise chances are that those operations will not all be specified, and thus neither will the relevant security requirements.

3.2.2.3 Analyze the STRIDE Properties

Security requirements express that certain security properties are satisfied. STRIDE[3] is a model that describes types of attacks and its corresponding security properties. These types and properties are listed in Table 1. STRIDE is not a taxonomy or categorization of attacks, because many attacks will fit into multiple categories. It is however an excellent source of inspiration in security analysis such as threat modeling or gathering security requirements.

Attack type	Security property
Spoofing	Authentication
Tampering	Integrity
Repudiation	Non-repudiation
Information Disclosure	Confidentiality
Denial of Service	Availability
Elevation of Privilege	Authorization

Table 1: STRIDE properties

For every operation identified in the first two steps requirements can be defined for the STRIDE properties of that operation. The result is an extensive list of security requirements, which is specified on the same level of detail as the functional requirements, and which can be used by a development team as a guide during development of the software system. Example questions are given in Table 2. Another good method to find security requirements is to examine the functional requirements for hidden assumptions.

3 First thought of by employees of Microsoft and integrated in its SDLC methodology.

Authentication for *operation*?
Is authentication required for *operation*?
How strong an authentication is required?
Who should be able to perform *operation*?
Integrity for *operation*?
Are there certain conditions that need to be satisfied before, during or after *operation*?
Must tampering with operation be prevented or detected?
Non-repudiation for *operation*?
Should *operation* be traceable?
If so, to what aspects of it should be traceable (person, time, place, ...)?
Confidentiality for *operation*?
Is the content of *operation* (i.e. data on which it operates) confidential? If so, who is allowed to see it?
Is the meta-data of *operation*'s content (e.g. name or identifier) confidential? If so, who is allowed to see it?
Is the fact that *operation* is performed confidential? If so, who is allowed to know it?
How long must the information remain confidential?
Availability for *operation*?
Is it important that operation can be performed at any time?
What is the accepted down-time for *operation*?
Authorization for *operation*?
On what data is a user allowed to perform *operation*?
What privileges are required to perform *operation*?
Under what conditions is user allowed to perform *operation*? For example, this *operation* is only allowed during office hours.

Table 2: STRIDE requirements questions (based on [Shostack], pp.234-240)

As the software system is developed and the security requirements are implemented, a mitigation may lead to extra Application Assets in the software system, such as passwords. Because they are something that needs to be protected, requirements need to be defined for them, too. The requirements gathering process may therefore consist of several iterations, taking place during the development cycle. This is no different from the regular process of adding additional functional requirements.

In the public audit report, the security context need only list the Application Assets that can be directly manipulated by users (authorized and non-authorized) of the software system. Only for those Application Assets need the requirements be listed. This prevents the audit report from becoming unreadable, and protects the software developers against disclosing implementation

details. However, requirements that do not need to be listed *are* important in order to determine the security of the software system, and must be evaluated by the auditor too.

3.2.3 Security Assumptions

A software system's security also depends on environmental factors such as software that is developed by a third party, the way in which an application is used, the physical or infrastructural environment in which the software system is used, etcetera. Not all of these factors are under the developer's control. Nevertheless, they are important and must therefore be documented as *security assumptions*.

If certain security requirements are not (fully) implemented, a reason must be given in the security assumptions. For example, the decision to not define or implement certain confidentiality requirements could be based on the assumption that the “Administrator” user can be trusted.

Assumptions about the likelihood of a certain threat are dangerous. First of all, likelihood is extremely difficult to reliably determine or even estimate. Secondly, once a certain weakness or vulnerability is known, the likelihood that it will be exploited increases. In case of an audit, assumptions about the likelihood must therefore be explained.

3.2.4 Example

As an example, consider this study of a wiki application for development teams.

3.2.4.1 Description of the Software System

The wiki for development teams allows developers to share knowledge. This knowledge is stored in pages, that can be edited by users. For every page, a history is kept, so that previous versions of the page can be seen. Every week an e-mail is sent to all users, listing the changes of that week. An administrator can add users to the software system[4].

3.2.4.2 High-level Security Requirements

From the description of the software system the following set of Application Assets can be defined:

- Pages
- Page history
- Page versions
- Users (name, e-mail address)

Information Assets

- Send e-mail with changes
- Add user
- Edit page
- View specific version of page

Operations

This list is expanded with all associated operations. For every Information Asset assign the operations *create, read, update* and *delete*. The operations 'edit page' and 'view specific version

4 To keep things simple all users, other than the administrator, will have the same authorizations.

of page' correspond with 'update page' and 'view page' respectively (viewing a page and a specific version of the page is essentially the same operation in this application).

Next, the security requirements are gathered based on the STRIDE properties of all the operations listed. During this analysis one is forced to think about certain requirements. Some of them may be trivial, others may not. There will be overlaps and duplicates due to the fact that STRIDE properties sometimes overlap. In some cases the STRIDE property for a certain operation does not make sense. For example, a read operation will not have any integrity requirements here, but in another software system, it may have. It is better to err on the secure side by considering too many possible requirements, than to miss a requirement.

The result of this process is shown in the following table (note that these requirements are merely an example and there is no claim that these particular requirements make the most secure wiki application, or should be used for similar software systems):

STRIDE property for process/operation	Requirement	For eval?	Remarks
Creating pages			
Authentication for creating pages?	Only registered users can create pages.	y	
Authentication for creating pages?	A user should not be able to create pages on behalf of others.	y	
Integrity for creating pages?	Not a risk: all page content is writable for all registered users.		
Non-repudiation for creating pages?	Not being able to see who created a page is not considered a risk.		
Confidentiality for creating pages?	Not a risk: all pages are public, this goes for content and meta-data.		
Availability for creating pages?	Not being able to create pages should not last more than one day.		
Authorization for creating pages?	Not a risk: everybody can create any page within the software system. For example, there is no separation of users or pages into organizations		
Reading pages			
Authentication for reading pages?	Not a risk: everybody can read pages.		
Integrity for reading pages?	Not a risk: reading does not modify anything.		
Non-repudiation for reading pages?	Not being able to prove that someone read a page is not an issue here.		
Confidentiality for reading pages?	Not a risk: all page data and meta-data is assumed to be public.		
Availability for reading pages?	One should always be able to read pages.	y	
Authorization for reading pages?	Not a risk: there is no separation of pages into organizations or something like that.		
Updating pages			
Authentication for updating pages?	Pages should only be edited by registered users.	y	
Authentication for updating pages?	A user should not be able to edit pages on behalf of others.	y	
Integrity for updating pages?	Only one person should be able to update a page at a	y	

STRIDE property for process/operation	Requirement	For eval?	Remarks
	time. Edits are only allowed on the latest version of the page.		
Non-repudiation for updating pages?	One must be able to prove who updated a page.	y	
Confidentiality for updating pages?	Not a risk: all page data and meta-data is public.		
Availability for updating pages?	Not being able to edit a page should not take longer than 4 hours.	y	
Authorization for updating pages?	Not a risk: there is no separation of pages into organizations or something like that.		
Deleting pages			
Authentication for deleting pages?	Pages should only be deleted by registered users.	y	
Authentication for deleting pages?	A user should not be able to delete pages on behalf of others.	y	
Integrity for deleting pages?	A page that is deleted must no longer be editable.	y	
Non-repudiation for deleting pages?	One must be able to prove who deleted a page.	y	
Confidentiality for deleting pages?	Not a risk: all page data and meta-data is public.		
Availability for deleting pages?	Not a risk: being unable to delete a page should not last more than one week, risk is accepted.		An explanation should be added to the security assumptions.
Authorization for deleting pages?	Not a risk: any registered user is allowed to delete any page.		
Creating page history			
Authentication for creating page history?	Not a risk: the page history is created initially by the software system.		
Integrity for creating page history?	Page history should reflect the historic pages exactly. With every page create/update/delete, the history must be adapted, and only then. The page history should contain accurate meta-data.	y	This overlaps with non-repudiation on page updates
Non-repudiation for creating page history?	As the page history contains an audit trail of some sort, this property corresponds with the integrity of the page history.	y	
Confidentiality for creating page history?	Not a risk: all meta-data in the page history is public.		
Availability for creating page history?	Not a risk: if the page history is only created during installation.		
Authorization for creating page history?	Not a risk: there is no separation of pages into groups, same for page history.		
Reading page history			
Authentication for reading page history?	Not a risk: everybody can see the page history, even non-authenticated users.		
Integrity for reading page history?	Not a risk: reading does not modify anything.		
Non-repudiation for reading page history?	Not a risk: knowing wether someone read the page history is not needed.		
Confidentiality for reading page history?	Not a risk: all meta-data is public.		

STRIDE property for process/operation	Requirement	For eval?	Remarks
Availability for reading page history?	One must always be able to see the page history.	y	
Authorization for reading page history?	Not a risk: there is no separation of pages into groups, same for page history.		
Updating page history			
Authentication for updating page history?	Changing the page history affects non-repudiation of pages. Therefore this should not be possible to non-authenticated users.	y	
Integrity for updating page history?	Only meta data for the updated page should be updated, not for another page.	y	
Non-repudiation for updating page history?	Not a risk: page history is audit trail itself, we do not keep an audit trail of an audit trail.		
Confidentiality for updating page history?	Not a risk: page history does not contain confidential information.		
Availability for updating page history?	One should always be able to update the page history when editing a page.	y	
Authorization for updating page history?	Not a risk: there is no separation of pages into groups, same for page history.		
Deleting page history			
Authentication for deleting page history?	Deleting page history is not an implemented operation and should not be possible.		
Integrity for deleting page history?	Deleting page history is not an implemented operation and should not be possible.		
Non-repudiation for deleting page history?	Deleting page history is not an implemented operation and should not be possible.		
Confidentiality for deleting page history?	Deleting page history is not an implemented operation and should not be possible.		
Availability for deleting page history?	Deleting page history is not an implemented operation and should not be possible.		
Authorization for deleting page history?	Deleting page history is not an implemented operation and should not be possible.		
Creating page versions			
Authentication for creating page versions?	A page version is created automatically when a page is edited, see update page.		
Integrity for creating page versions?	See update page.		
Non-repudiation for creating page versions?	See update page.		
Confidentiality for creating page versions?	See update page.		
Availability for creating page versions?	See update page.		
Authorization for creating page versions?	See update page.		
Reading page versions			
Authentication for reading page versions?	Seeing a page version is the same as reading a page, see read page.		

STRIDE property for process/operation	Requirement	For eval?	Remarks
Integrity for reading page versions?	Seeing a page version is the same as reading a page, see read page.		
Non-repudiation for reading page versions?	Seeing a page version is the same as reading a page, see read page.		
Confidentiality for reading page versions?	Seeing a page version is the same as reading a page, see read page.		
Availability for reading page versions?	Seeing a page version is the same as reading a page, see read page.		
Authorization for reading page versions?	Seeing a page version is the same as reading a page, see read page.		
Updating page versions			
Authentication for updating page versions?	Updating page versions is not an implemented operation and should not be possible.		
Integrity for updating page versions?	Updating page versions is not an implemented operation and should not be possible.	y	
Non-repudiation for updating page versions?	Updating page versions is not an implemented operation and should not be possible.		
Confidentiality for updating page versions?	Updating page versions is not an implemented operation and should not be possible.		
Availability for updating page versions?	Updating page versions is not an implemented operation and should not be possible.		
Authorization for updating page versions?	Updating page versions is not an implemented operation and should not be possible.		
Deleting page versions			
Authentication for deleting page versions?	Deleting page versions is not an implemented operation and should not be possible.		
Integrity for deleting page versions?	Deleting page versions is not an implemented operation and should not be possible.	y	
Non-repudiation for deleting page versions?	Deleting page versions is not an implemented operation and should not be possible.		
Confidentiality for deleting page versions?	Deleting page versions is not an implemented operation and should not be possible.		
Availability for deleting page versions?	Deleting page versions is not an implemented operation and should not be possible.		
Authorization for deleting page versions?	Deleting page versions is not an implemented operation and should not be possible.		
Creating users			
Authentication for creating users (name, e-mail address)?	Only the administrator should create users.	y	
Integrity for creating users (name, email address)	The email address should have a correct format, but not considered a risk.		Security assumption: administrator should pay attention to email address format.
Non-repudiation for creating users (name, email address)?	As there is only one administrator, not being able to prove that a user was created is not an issue.		
Confidentiality for creating users (name, email address)?	The e-mail address must only be visible to the administrator here.	y	

STRIDE property for process/operation	Requirement	For eval?	Remarks
Availability for creating users (name, email address)?	Not being able to create users should not last more than one day.	y	
Authorization for creating users (name, email address)?	Not a risk: there is only one group of users, users are not separated into groups.		
Reading users			
Authentication for reading users (name, email address)?	Only the administrator and the user him/herself can view his/her user data.	y	
Integrity for reading users (name, email address)?	Not a risk: reading does not modify anything.		
Non-repudiation for reading users (name, email address)?	Not being able to prove that someone read user information is not an issue.		
Confidentiality for reading users (name, email address)?	A user's e-mail address should only be visible to the administrator and the user him/herself.	y	
Availability for reading users (name, email address)?	One should always be able to see user information. Administrator can always look in database, therefore not a risk.		Security assumption should be added
Authorization for reading users (name, email address)?	A user should not see another user's information.	y	
Updating users			
Authentication for updating users (name, email address)?	Only an administrator and the user him/herself should update the email address.	y	
Integrity for updating users (name, email address)?	Only a valid email address should be entered. Not considered a risk.		Security assumption should be added
Non-repudiation for updating users (name, email address)?	An administrator should always see when user data was changed.	y	
Confidentiality for updating users (name, email address)?	A user's email address should only be visible to the administrator and the user him/herself.	y	
Availability for updating users (name, email address)?	If users cannot change their e-mail address, they can contact the administrator. Not considered a risk.		Security assumption should be added
Authorization for updating users (name, email address)?	A user can only edit their own data. an administrator can edit all users.	y	
Deleting users			
Authentication for deleting users (name, email address)?	Only the administrator can delete users.	y	
Integrity for deleting users (name, email address)?	Only the user data is removed. The user name will be kept in the page history. User modification events are also not deleted.	y	
Integrity for deleting users (name, email address)?	The administrator user cannot be deleted.	y	
Non-repudiation for deleting users (name, email address)?	An administrator should see when a user was deleted.	y	
Confidentiality for deleting users (name, email address)?	Not a risk: the fact that a user is deleted is not confidential information.		
Availability for deleting users (name, email address)?	Not a risk: if a user cannot be deleted there is not a big issue.		Security assumption should be added
Authorization for deleting users (name, email address)?	The administrator user cannot be deleted.	y	

STRIDE property for process/operation	Requirement	For eval?	Remarks
Sending email			
Authentication for sending email with changes?	Emails are sent automatically by the software system, only the software system itself should do this.	y	
Integrity for sending email with changes?	Email should have valid format and contain all relevant changes.		
Integrity for sending email with changes?	Software system should only send change emails, no other emails.	y	
Integrity for sending email with changes?	Software system should only send change emails to registered users, not to other recipients.		
Non-repudiation for sending email with changes?	Knowing whether the email is sent is not important to know.		
Confidentiality for sending email with changes?	The data is assumed to be visible to all users of the software system. Should the data not leak via email, then the administrator and the users should implement measures to make sure that no untrusted email addresses are used.		
Availability for sending email with changes?	Not a risk: if the email cannot be sent, users can still consult the page history.		
Authorization for sending email with changes?	Not a risk: all users will receive the same email of all changes of the last week.		

After the initial requirements have been selected, the developers implement the authentication requirements with password-based authentication. This will add an extra Information Asset to the application, with a few specific operations:

- Verify password
- Change password
- Reset password
- Set password

As gathering security requirements is an iterative process, these operations will have to be added to the Application Assets and the requirements for them must be defined based on the STRIDE properties. Because these operations are directly used by the users of the software system, they will need to be listed in the security context which will be part of the public audit report.

3.2.5 Schematic Representation

The following schematic recapitulates all steps taken during the Context Phase:

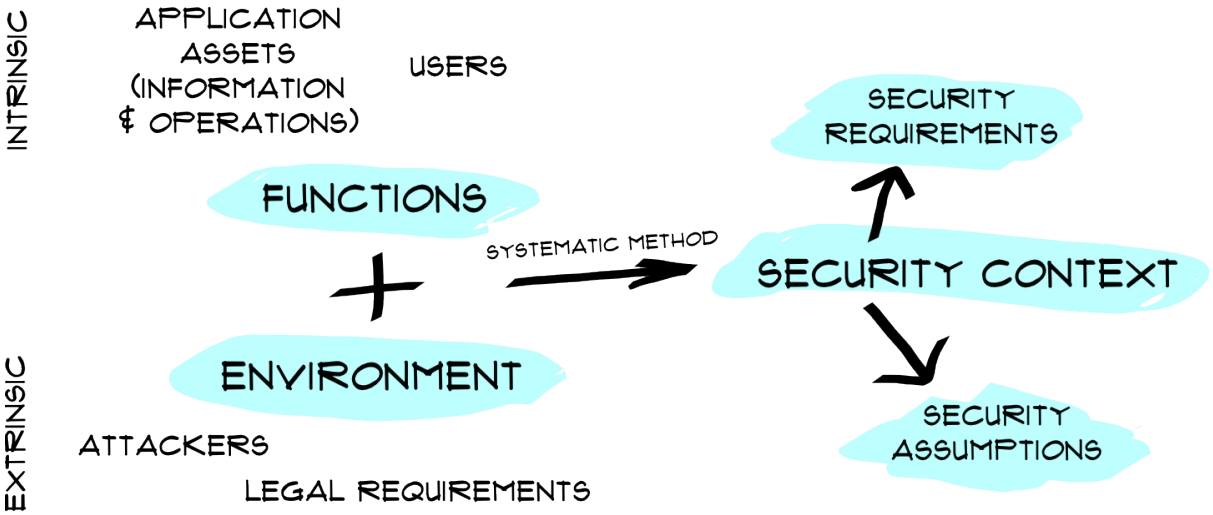

Figure 1: Steps taken during the Context Phase.

3.3 Threats Phase

Threats can manifest themselves because of two reasons:

1. The operation for which a security requirement is defined is not implemented correctly.
2. The software system uses insecure technologies or uses technologies in an insecure way.

The former threats are called *functional threats*, the latter *architectural threats*. Both groups can overlap. This might lead to a little extra work, but mostly it will not when making proper use of documented assumptions. To prevent vulnerabilities it is better to err on the secure side by identifying too many threats, rather than miss some.

3.3.1 Defining Functional Threats

Functional threats can be found by analyzing the security requirements. Negating the requirement or parts of the requirement will lead to threats.

Take the following requirements as examples:

Requirement	Threat(s)
Only one person should be able to update a page at a time. Edits are only allowed on the latest version of the page.	Multiple persons update a page simultaneously. An attacker is able to update an older version of the page.
Being unable to delete a page should not last more than one week.	A user is unable to delete a page for longer than one week.
One must be able to prove who updated a page.	It is not possible to prove who updated a page.

Table 3: Security requirements and associated functional threats.

The last requirement and the corresponding threat are a bit more abstract. Without more information of how updating the page works or how to prove that a page is updated, it is not possible to define specific threats. An implementation decision would likely lead to more Application Assets and another iteration of the requirements analysis process. From the resulting requirements it must be possible to define more specific threats.

3.3.2 Defining Architectural Threats

The design of a software system has a large influence on its security. Different technologies carry different inherent security risks that need to be mitigated accordingly. An architectural risk analysis will discover such risks[5]. Because the architecture may change during the development process, you may need to repeat this analysis with every change.

3.3.2.1 Create an Architecture Inventory

To create an architectural inventory it is important to identify the different *trust zones* within which the components that make up the software system can trust each other. If two components are in a different trust zone, they can in principle not trust each other. Trust zones are separated from each other by *trust boundaries*.

This analysis starts with an inventory of the components and technologies that make up the software system, how they interact, and where the different trust zones are. External entities and their interactions with the software system must also be listed.

Usually this is done by making a diagram of the software system. A network diagram is best suited to identify risks on the network level. A data flow diagram shows the interaction between different processes and will help to identify risks on the application level. A protocol analysis can best be done by means of a “swim lane” diagram or UML sequence diagram.

A flowchart diagram shows the program flow of the software system, but does not show where an attacker can take control. This makes flowcharts less useful for an architectural risk analysis.

5 This risk analysis is sometimes called `threat modeling'. In our opinion, threats can be found not only by analyzing the architecture. Therefore we chose not to use the term `threat modeling'.

Based on [Shostack] (pp.52) it is possible to define a few rules of thumb of what should be in a diagram:

- Show the events that drive the software system.
- Show the processes that are driven.
- Determine what responses each process will generate and send.
- Identify data sources for each request and response.
- Identify the recipient of each request and response.
- Identify the sender of each request and response.
- Ignore the inner workings, focus on scope.
- Ask if something will help you think about what goes wrong, or what will help you find threats.

Diagrams, especially data flow diagrams, can exist on multiple levels of abstraction. As your architecture becomes more concrete, you can identify more specific risks. The level of abstraction determines how concrete the threats leading to those risks will be.

External modules, i.e. parts of the software system that were not developed as part of the software system, but are used by the software system as a dependency, must be regarded as part of the software system and thus must be described. It may not always be possible to document this in a drawing. External modules are for example web frameworks, a database abstraction library, a templating engine or a cryptographic library.

3.3.2.2 Apply a Threat Library

Once the architecture is described, it becomes possible to consider whether known threats are applicable. A collection of known threats is called a *Threat Library*. Microsoft's STRIDE per element or STRIDE per interaction ([Shostack], pp. 61-86) can be seen as a very abstract threat library that can be applied to high-level architecture diagrams. More concrete threats are listed in CAPEC/CWE, a collection of attack patterns and weaknesses provided by Mitre ([CAPEC], [CWE]).

Sometimes threats will require architectural information on a deeper level of detail than the architectural inventory provides. In this case, you may need to make your architectural inventory more concrete. In the early phases of the development, this may not be possible, because certain architectural details have not yet been decided upon.

If the same threat occurs in multiple locations in the architecture, it must be listed for every location. This is because different occurences of the same threat may require a different way to handle the threat.

For certification against the Framework Secure Software the Secure Software Foundation (SSF) demands the use of Threat Libraries that it has officially endorsed.

3.3.2.3 Determine Threats in External Modules

For *every* external module there is a generic threat, namely that the external module contains security vulnerabilities. For each of these modules a list must be made, consisting of the threats that can reasonably be expected to occur due to the use of that module, just as if that module

was developed by the developer himself. For example: likely threats against a database abstraction library are SQL injection or other database injection threats. A templating engine will be threatened by cross-site-scripting attacks if it is used to output data that is viewed in a web browser. For a cryptographic library, think of threats such as poor random number generation, not using a random IV, reusing a nonce, etcetera.

For the specific version of each external module, all known vulnerabilities must be added to a list of threats.

All lists of threats that have been established in the steps before (being the list of functional threats, the list of architectural threats, the selected Threat Library/Libraries, and the lists of known vulnerabilities in external modules) together form the *List of Threats.*

3.3.2.4 Example

A press agency allows freelance photographers to upload digital photographs to itself. Photos are uploaded to an FTP server. From there, the uploaded images are processed by a scheduled job that creates a thumbnail and scaled versions of the images. Those images are then made available to a CMS system, where viewers can view images and comment on them. The trusted zone is marked by the dashed line (see Figure 2).

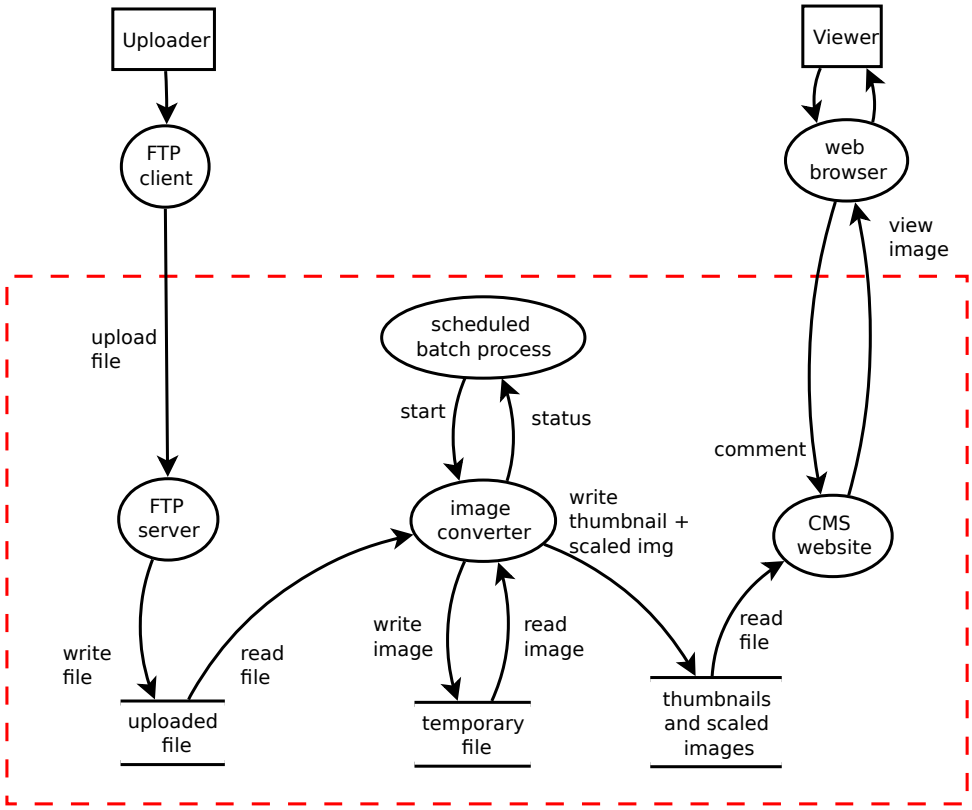

Figure 2: Data Flow Diagram of the press photo system, including trust boundaries

Now apply a few threats to this architecture (applying a complete threat library would take up too much space in this document, but should be done when applying this framework).

CAPEC-154: Resource Location Attacks

This attack pattern from CAPEC describes a possible attack that could take place of this software system (see http://capec.mitre.org/data/definitions/154.html). It takes the point of view of an attacker.

> *"An attacker utilizes discovered or crafted file path information for the purpose of locating and exploiting a security sensitive resource. This category of attack involves the paths used by an application to store or retrieve resources. Specifically, attacks in this category involve manipulating the path, causing the application to look in location unintended by the application maintainer, or determining the paths through prediction or lookup. This differs from File Manipulation attacks in which the contents of the files are affected or where the files themselves are physically moved. Instead, this attack simply concerns itself with the paths used to find or create resources."*

In the above architecture, there are three locations where files are used. The above threat would apply to the uploaded file, the temporary file, the thumbnail files and scaled image files. The next phase will deal with whether this attack is actually possible on those components.

CWE-311: Missing Encryption of Sensitive Data

This weakness from CWE looks at a design or implementation problem that an attacker could abuse (see http://cwe.mitre.org/data/definitions/311.html). Its point of view is a developer's.

> *"The software does not encrypt sensitive or critical information before storage or transmission. The lack of proper data encryption passes up the guarantees of confidentiality, integrity, and accountability that properly implemented encryption conveys."*

The connections between FTP client and FTP server and between web browser and CMS website use a (potentially public) network connection. An attacker could eavesdrop on these connections if they are not encrypted. Therefore, two threats must be added to the List of Threats: "An attacker can eavesdrop on the connection between FTP client and FTP server." and "An attacker can eavesdrop on the connection between browser and CMS website."

STRIDE per Element: Denial of Service on "temporary file"

As an example of a STRIDE per element threat look at "Denial of Service" on the "temporary file". STRIDE per element or STRIDE per interaction yields more abstract threat descriptions, therefore the analyst should use his knowledge about the software system to come up with more specific threats that lead to concrete mitigations in a later phase. How could the "temporary file" be rendered unavailable? As an example think of the following:

- An attacker uploads a file that is too large for the converter process to handle and the temporary file is never created.
- An attacker manipulates the file name so that the temporary file name points to

something that does not exist, or to a file-like resource that blocks forever.
- An attacker manipulates the file name so that the temporary file name is invalid and cannot be written.
- An attacker fills the FTP upload folder with so many file that there is no space left to create the temporary file.

Because of the overlap that is inherent in the STRIDE framework, some of these threats could also apply to the “temporary file” data store or to the “image converter” process.

3.3.3 Determining the Threat Impact

When all threats are identified, the next step is to determine how they impact the software system's security. For every threat, one must determine if it impacts the STRIDE properties of the Application Assets. If it does impact one of the STRIDE properties of an asset for which a security requirement exists, the threat needs to be resolved. To document this process, these threats together with the security requirements that they impact must be recorded. One must determine what the impact would be if the threat would manifest itself, regardless of whether it is possible or not.

3.3.4 Finding Mitigations

After the relevant threats have been identified determine how to handle them. For *every* threat in the list of threats, a mitigation must be described. This can be a technical mitigation (e.g. implement a firewall, apply strict input validation, use parameterized queries for database access) or a non-technical mitigation (e.g. transfer the risk or accept it). To merit a Secure Software Certificate both the technical and non-technical mitigations that must be in place in a software system must be documented (see section 3.2.3).

For threats against external modules, a possibly valid reason to not mitigate a specific threat is that it is shown that this threat will not manifest itself, for example because the module is used in a restricted way, or because the module is used according to security guidelines that were provided by the module, or because tests or code reviews have shown that the threats do not occur.

If a mitigation is not implemented or cannot be implemented this must be explained in the security assumptions (see section 3.2.3). For example, if export regulations restrict the use of strong cryptography, it may not be possible to mitigate against a certain threat.

3.3.5 Example

Threat	Impact	Mitigation
Resource location attack on filename for writing uploaded file.	Confidentiality of files, integrity of files, availability of files.	Accept: assume that server implements FTP correctly.
Resource location attack on filename for reading uploaded file.	Reading and processing arbitrary files (confidentiality and integrity), create scaled version of arbitrary photos (confidentiality and integrity).	Eliminate '../' patterns from filename before processing.
Resource location attack on filename for reading/writing the temporary file.	Confidentiality of files, integrity of files.	Ensure that temporary file name will not be based on user input.
Resource location attack on filename for reading/writing the thumbnail files and scaled image files.	Confidentiality of files, integrity of files.	Sanitize input: allow only alphanumeric, - and _ characters.
Attacker eavesdrops connection between FTP client and FTP server.	Confidentiality of photos.	Transfer to user, advise to use secure FTP.
Attacker eavesdrops connection between web browser and CMS website.	Confidentiality of photos, confidentiality of comments.	Ensure that use of HTTPS is required.
An attacker uploads a file that is too large for the converter process to handle and the temporary file is never created.	Availability of photos in CMS.	Ensure that converter checks file size before processing.
An attacker manipulates the file name so that the temporary file name points to something that does not exist, or to a file-like resource that blocks forever.	Availability of photos in CMS.	Ensure that temporary file name will not be based on user input.
An attacker manipulates the file name so that the temporary file name is invalid and cannot be written.	Availability of photos in CMS.	Ensure that temporary file name will not be based on user input.
An attacker fills the FTP upload folder with so many files that there is no space left to create the temporary file.	Availability of new photos in the system.	Transfer to user: advise use of separate disk for temporary files and CMS files.

Table 4: identified threats with their associated impact and mitigations.

In section 3.3.2.4 the impact and mitigations for the threats that were identified for the press photo system were determined. They are listed in the following table. Since in this case there was no list of specific security requirements, a more generic description of the impact is used.

When filling this table, more security assumptions become clear. Firstly, the system is not shipped with an FTP server, and therefore the security of the FTP server becomes the responsibility of someone else. Secondly, the developers have not chosen to mitigate against an attacker that fills the server with files. This threat should either be accepted or be mitigated by the user of the software system. These security assumptions must be documented in the public audit report that comes with a Secure Software Certificate.

Also note that the mitigations listed here may not be perfect. In fact, the elimination of `../' from filenames fails to mitigate against a few important variants of the attack. Because of this the auditor needs to evaluate the effectiveness of the mitigation as part of a certification.

3.3.6 Schematic Representation

The following schematic recapitulates all steps taken during the Threats Phase:

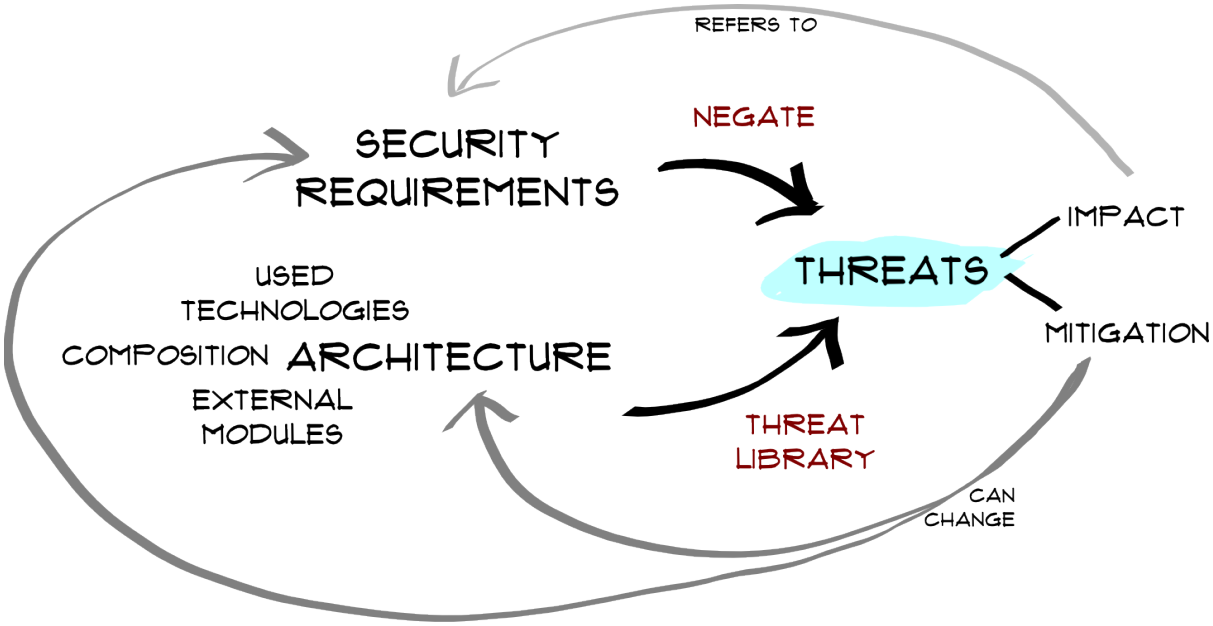

Figure 3: Steps taken during the Threats Phase.

3.4 Implementation Phase

Not all threats can be identified at the architecture level. In the example in section 3.3.5 a path manipulation attack was identified and a mitigation against that specific attack was chosen. But there may be additional threats, that may be caused by the implementation of the conversion process. An image loading routine implemented in the C programming language is threatened by buffer manipulation attacks, whereas calling an external program using `system()` can lead to command injection attacks. For this reason a code audit is an essential part of the assessment, needed to catch implementation errors. A code audit consists of a systematic code review, described in section 3.4.1.

3.4.1 Systematic Code Review

In this section a systematic approach for security code reviews is described. This approach is intended as a guideline for auditors, but can also be used by developers.

3.4.1.1 Identify the Attack Surface

An *Attack Surface* consists of data that crosses a trust boundary. Examples are entry points and data sources that are in a different trust zone than the zone in which the code that uses them runs, or data that originates from a location in the code but is sent to another trust zone. The architectural inventory helps to identify the trust zones and attack surfaces, but not completely. Unexpected sources of input may exist, such as environment variables, configuration settings, HTTP request headers, or even responses from an external DNS server.

3.4.1.2 Trace the Code Flow for Violations of the Secure Coding Standard

From the attack surface, trace the code flow and look for violations of the Secure Coding Standard (section 3.4.2). Violations must be documented, mentioning at least the file name, the line number(s), the issue and a short motivation. If there is a valid reason for the violation, an explanation must be provided in the list of security assumptions by the developer and this explanation must be evaluated by the auditor.

3.4.1.3 Evaluate Correctness of Implementation

Functionality in general needs to be implemented securely. A necessary condition for this is correctness. During a code review, one must therefore look for incorrect constructs that impact the security of the software system.

For every mitigation that needs to be verified by a code review (either because its verification includes a code review (section 3.5), or because the auditor/reviewer thinks it is needed), one must evaluate whether the implementation is correct. If not, describe why.

3.4.2 Secure Coding Standard

The S*ecure Coding Standard* is a set of rules to which code must adhere. The coding standard contains language and framework specific rules, but most of them will be based on s*ecure coding principles*. Auditors will need to use a secure coding standard that is approved by the Secure Software Foundation. If no applicable approved coding standard is available, the secure coding principles published by the Secure Software Foundation must be used as the leading guideline.

Secure programming constructs that are not immediately obvious to other team members (this includes future junior team members or code auditors) or to the developer himself/herself at a later moment in time, or programming constructs that deviate from the rules in the secure coding standard, must be documented with a reason, if possible in the code itself.

An example of this is deliberately not initializing a variable to use it as a source of entropy ([Schneier2008]), or writing the check

```
offset > buffer_size – length
```

instead of

```
offset + length > buffer_size
```

to protect against buffer overruns ([Schönefeld]).

3.4.3 Schematic Representation

The following schematic recapitulates all steps taken during the Implementation Phase:

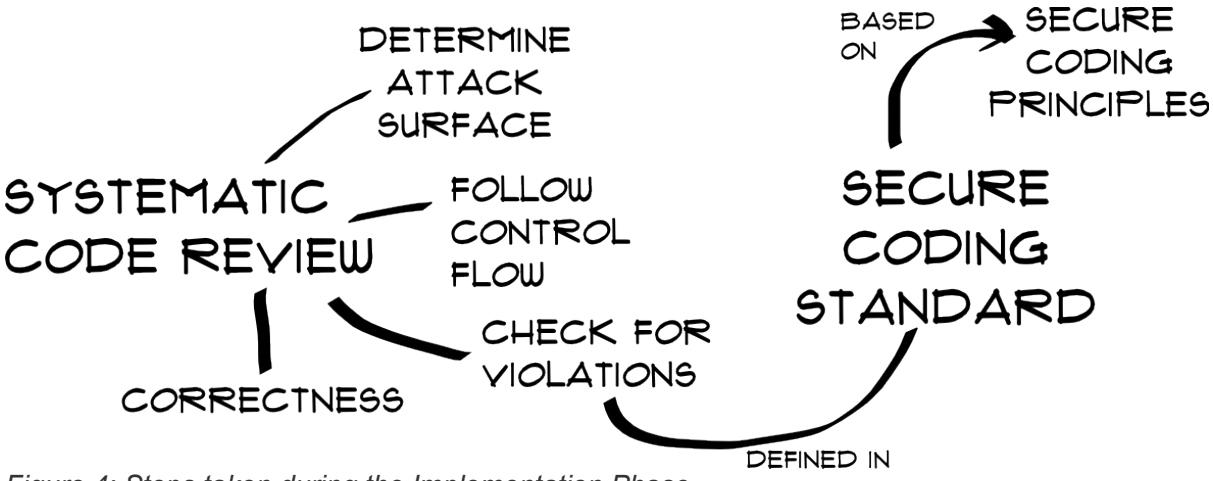

Figure 4: Steps taken during the Implementation Phase.

3.5 Verification Phase

While in theory an audit of the requirements, design and implementation should suffice to assess a software system's security, this is difficult to guarantee in practice. As an extra safety net it is necessary to look at how developers verify that what they implement is secure. This framework does this by stating that *every mitigation must be verified*, that the method of verification must be correct and complete, and that the verification must be executed regularly.

3.5.1 Verifications for Every Mitigation

Software can be verified by testing, reviewing or formal proofs. To verify mitigations, these methods are no different, but they have their own characteristics.

Formal proofs can be used to verify for example cryptographic protocols. Because of the very specialized knowledge this requires, one only encounters this in a few organizations.

A review is usually done on code or configurations, but can be applied to a design or a protocol as well. To review code, the method in section 3.4.1 can be used. Code reviews can be done on the whole code or incrementally.

Security tests come in different flavors.

The most well-known is the *penetration test*, where a team of security experts will try to attack the software system. Such a test can be done with or without prior knowledge of the software system. The knowledge of the experts leads to findings that were overlooked. But penetration tests have two inherent problems. First, the less the testers know about the application's functionality, the less application specific the problems that are found. And second, a penetration is an ad hoc test that is highly dependent on the skill set, interests and problem solving capabilities of the person(s) doing the test. This means there is no guarantee the results will be complete, nor that they will be the same when another tester or other testers perform the penetration test. Nevertheless penetration tests still are a very valuable addition to the verification methods available to developers and auditors.

A *vulnerability scan* checks if known weaknesses exist in the code or configuration of the software system. Automated vulnerability scan software exists to scan for network, platform, protocol, or application vulnerabilities. Such scanners often find easy to miss configuration problems, but are infamous for the high number of false positive results. Nor do they always find all known vulnerabilities, or will they be able to find unknown/new vulnerabilities.

Attack scenarios that have been discovered during the identification of threats (see sections 3.3.1 and 3.3.2) can be performed as regular tests. These test are called *abuse tests*. Such tests can be done manually or can be automated. Performing abuse tests will guarantee that specific issues are tested. With automated tests, the tests can be run often without extra costs.

Perhaps the least sophisticated, but nevertheless often very effective security testing method is called *fuzzing*. A fuzzing test will automatically provide invalid, unexpected, or random data as input to a software system and watches how it behaves. It can detect anomalies in input handling, that may indicate a potential problem. A well-built fuzzing setup can run unattended for a relatively long time and find bugs that are often missed in code reviews. Reports have shown that fuzzing can be very effective, see [Sprundel] and [Jurczyk].

3.5.2 Correct and Complete Verification Methods

Not every verification can be applied in every situation. This section gives rules of thumb for when to apply which verification method.

3.5.2.1 Formal Proofs

Formal proofs are best used to verify the security of security protocols and cryptographic algorithms, where mathematical certainty is needed. This is a highly specialized field that requires significant expertise and resources.

3.5.2.2 Code/configuration Review

For verifications where an actual test is difficult to execute, a code or configuration review is often more effective. Automated tools (so-called static analysis tools) help greatly to make the process repeatable, but are not guaranteed to find all problems.

3.5.2.3 Penetration Test

To simulate an attack by a hacker a penetration test can be executed. This is best used for tests that are difficult or impossible to automate. Often this is due to their “creative” nature where the outcome of one test is used to adapt the next test. This allows penetration testers to combine attacks in unexpected ways that might lead to a successful breach of security.

When performing a penetration test it is Important that the testers know what to test, either by having a list of test scenarios or a fixed scope for the test. The test report must clearly describe what has been tested (even if this did not result in any findings), and how to reproduce the findings. Often such findings can be converted to an abuse test that can be run (automatically) later on.

3.5.2.4 Abuse Test

If the test scenario is known and needs to be repeatable, an abuse test is a good verification method. The abuse test should be as complete as possible by containing enough test cases to cover all relevant variations associated with the scenario. If such test cases are not easy to determine, a specific vulnerability scan (such as for SQL injection or cross-site-scripting) or a fuzzing test may be more effective.

3.5.2.5 Vulnerability Scan

Known configuration problems that are easily missed in code/configuration reviews or manual penetration testing can often be found through a vulnerability scan. It is impossible to give guidelines for the appropriateness of a vulnerability scanner. This will have to be discovered through experience. It is advised to use vulnerability scanners, preferable scanners with a proven track record, but not to rely on them. They help, but often don't guarantee the presence or absence of vulnerabilities.

3.5.2.6 Fuzzing

Fuzzing tests by nature work by generating input to be fed to a software system. This makes them excellent for tests for those parts of the software system that process input, such as parsers. If the number of test cases is large, fuzzing is more effective than abuse tests. Most fuzzers will detect program crashes, which are indicative for memory problems. Some can even restart the program to be tested after a crash, which greatly helps when applying this type if verification method on a larger scale or during longer periods of time. To detect other problems, specific detectors will need to be used. Running fuzz tests with a code coverage tool can help to see how thorough the fuzzing test has been.

The quality of the results depends on what problems the detector can detect and on the data that is generated: at what abstraction level is the input manipulated or generated (at the lowest bit level or highest data structure level), are specific attack vectors used and which ones?

3.5.3 Execution of Verifications

When verifications need to be executed is a matter of feasibility and experience. Early and often

would be the best, but not all verifications can be executed at all times. Some take preparation time or resources that may not always be available. Certain verifications may not be executable because the software system is simply not ready yet.

The only hard demand that can be set is that *one or more verifications for each identified threat must have been executed on the release version* of the software system and must have passed the security requirements before the software system is released. As a matter of fact without this a software system cannot be certified against this Framework Secure Software.

It may be a good idea to add this requirement to the release criteria of the software system to be tested.

3.5.4 Schematic Representation

The following schematic recapitulates all steps taken during the Verification Phase:

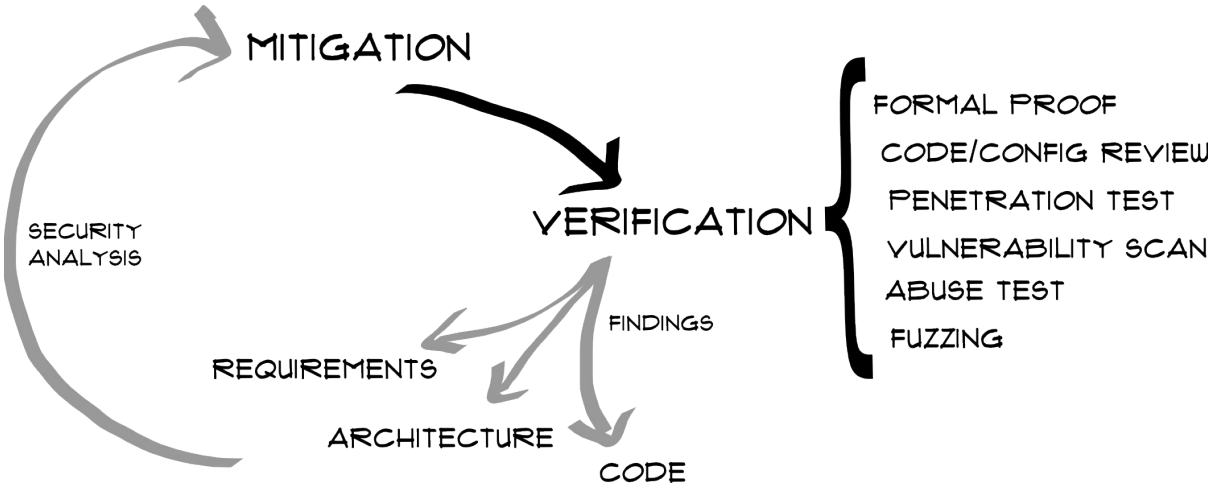

Figure 5: Steps taken during the Verification Phase.

3.5.5 Example

Based on the example mitigations from section 3.3.2.4 some verifications can be specified. They are shown in Table 5.

Threat	Mitigation	Verification
Resource location attack on filename for writing uploaded file.	Accept: assume that FTP server implements FTP correctly.	N/a
Resource location attack on filename for reading uploaded file.	Eliminate '../' patterns from filename before processing.	Code review
Resource location attack on filename for reading/writing the temporary file.	Ensure that temporary filename will not be based on user input.	Code review
Resource location attack on filename for reading/writing the thumbnail files and scaled image files.	Sanitize input: allow only alphanumeric, - and _ characters.	Code review
Attacker eavesdrops connection between FTP client and FTP server.	Transfer to user, advise to use secure FTP.	N/a
Attacker eavesdrops connection between web browser and CMS website.	Ensure that use of HTTPS is required.	Penetration test
An attacker uploads a file that is too large for the converter process to handle and the temporary file is never created.	Ensure that converter checks file size before processing.	Abuse test: run converter on large file.
An attacker manipulates the file name so that the temporary file name points to something that does not exist, or to a file-like resource that blocks forever.	Ensure that temporary file name will not be based on user input.	Code review
An attacker manipulates the file name so that the temporary file name is invalid and cannot be written.	Ensure that temporary file name will not be based on user input.	Code review
An attacker fills the FTP upload folder with so many files that there is no space left to create the temporary file.	Transfer to user: advise use of separate disk for temporary files and CMS files.	Verify that user manual mentions this.

Table 5: Verifications for mitigations

The table above lists verifications for the threats that must be mitigated. In one case a verification action for a non-technical mitigation was added: to verify that the user is made aware of a certain threat via the user manual. The auditor will evaluate whether the verification is adequate and whether it has been performed correctly.

3.6 Schematic Representation of the Whole Framework

The following figure shows an overview of all steps that together make up the Framework Secure Software:

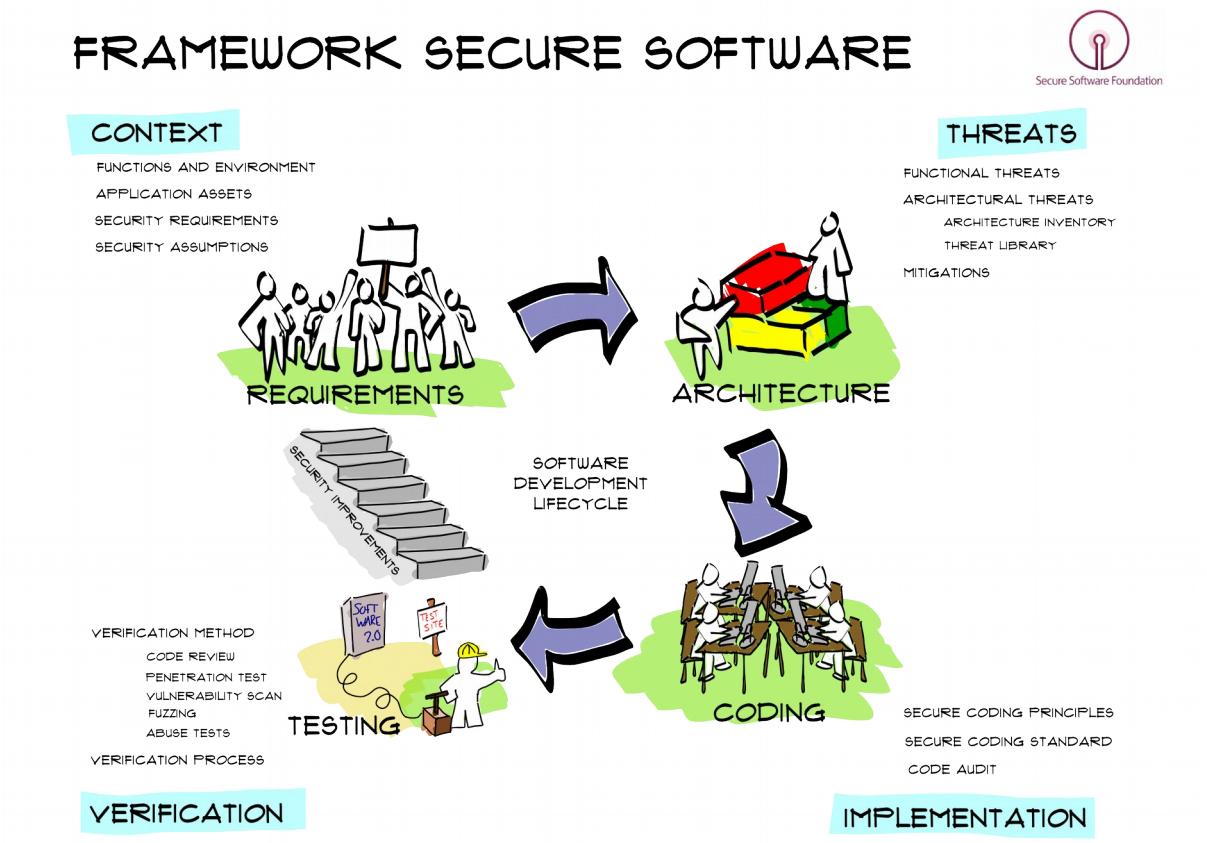

Figure 6: Overview of all steps in the Framework Secure Software.

4 Controls for Development

4.1 Introduction

Controls are the benchmarks against which the security of a software system is gauged. In order to be usable in the context of this Framework Secure Software they must both be generic, so they can be applied in all situations, and concrete, so that a software system that passes evaluation really can be expected to be secure against the identified threats it is potentially exposed to.

Controls are specified in the following form:

CODE-123	Name of the Control
Description	Description of the Control.
Goal	What will be achieved with this Control.
Rationale	Why the goal must be achieved.
Developer actions	Things a developer must do.
Developer results	Things a developer must deliver to prove the actions are done properly.
Criteria	Criteria the auditor will use to validate conformity to this Control.
Audit Method	Method(s) that the auditor can use to validate conformity to this Control.

Every Control can be referenced by its code and contains information for both developers and auditors. The Controls are grouped according to their associated phases in the certification process.

4.2 Context

4.2.1 CON-1: System functions and environment

CON-1	System functions and environment
Description	There must be a description of how the software system works, what it does, who will use it and how it will be used.
Goal	Understanding of what the software system does, what data it processes, how it does this, and who interacts with the software system.
Rationale	This provides the basis for gathering security requirements. It also helps anyone who wants to interpret the scope of applicability of a Secure Software Certificate to understand the security requirements involved and an auditor to assess the correctness and completeness of these security requirements.
Developer actions	Define the system functions and environment. Here one can think of the following:

CON-1	System functions and environment
	• Actors of the software system • Business processes • Other (external) stakeholders • Circumstances under which the software system is used • Circumstances under which the software system is not intended to be used, that could impact the security of the software system
Developer results	A documented description of the system functions and environment. This can be in the form of user stories, formal requirements documents, or use cases a.o.
Criteria	• The system functions must at least contain all actors performing operations on this software system and functionality provided to these actors. • From the system functions described it must be possible to derive the Application Assets. • The environment description must define the environment for intended use of the software system. • The system functions and environment must be clear and correspond with the implementation.
Audit Method	Through interviews, inspection of a running software system and studying documentation.

4.2.2 CON-2: Security requirements

CON-2	Security requirements
Description	All security requirements must be thoroughly defined.
Goal	Attain a good coverage of all possible security requirements.
Rationale	Security requirements are often forgotten or incomplete, because no structured process to gather security requirements is used. Forgetting to specify security requirements increases the likelihood that security tests are omitted or that code is not securely implemented.
Developer actions	Apply the systematic requirements gathering method to the Application Assets, described in section 3.2.2.
Developer results	A list of security requirements or explanations of a requirement's absence.
Criteria	• All STRIDE properties of all operations in the software system must be given at least one security requirement, or an explanation why there is no security requirement. • The reasoning behind the security requirements or their absence must be sound. • The security requirements must be specific. • Only the security requirements that are defined on Application Assets

CON-2	Security requirements
	that are directly operated upon by users of the software system need to be part of the public audit report.
Audit Method	Inspect the list of security requirements and explanations and review it for quality[6].

4.2.3 CON-3: Security assumptions

CON-3	Security assumptions
Description	All security assumptions must be defined.
Goal	Inform a user of the software system under what conditions the security requirements are fulfilled.
Rationale	The security of a software system is partially influenced by how the software system will be used and by the environment in which it will be used. Often these influences are not under the software system's developer's control, and are the responsibility of the users of the software system. Therefore the users of the software system should be aware of all assumptions made by the developer that have an impact on the security requirements that are or are not implemented in the software system.
Developer actions	Describe all security assumptions of the software system.
Developer results	A list of all security assumptions.
Criteria	• The security assumptions must explain all cases in which a requirement is deliberately not or not completely fulfilled. • The security assumptions must explain all cases in which a mitigation is accepted or implemented in a non-technical way. • The security assumptions must explain all cases in which a mitigation is not or cannot be (completely) implemented.
Audit Method	Inspect the list of security assumptions and review it for quality.

4.3 Threats

4.3.1 THR-1: Functional threats

THR-1	Functional threats
Description	For every security requirement, at least one functional threat must be defined.
Goal	Catch potential security problems at the functional requirements level.

6 Quality in the context of this framework encompasses a.o. completeness, correctness, sufficient level of detail, concrete enough to be usable by the developer or evaluator, and understandable.

THR-1	Functional threats
Rationale	Not fulfilling a security requirement can happen if the operation for which the security requirement is defined, is not implemented securely. Adding these situations as threats will ensure that the operations in the application are securely implemented.
Developer actions	Add functional threats for every security requirement, as described in section 3.3.1.
Developer results	A list of all functional threats.
Criteria	All situations in which the security requirements are not fulfilled must be added as functional threats.
Audit Method	Identify the situations in which the security requirements are not fulfilled and compare these with the functional threats in the List of Threats. Review this list for quality.

4.3.2 THR-2: Architecture inventory

THR-2	Architecture inventory
Description	An architecture inventory must be created.
Goal	Be able to identify potential security problems that are specific to the technologies use and the way they are applied.
Rationale	Potential security problems partly are the result of insecure design. Either by using insecure technologies, or applying these technologies in an insecure manner. It is therefore important to know what technologies comprise the architecture, how these technologies work and how they are integrated in the software system.
Developer actions	Describe all components, technologies, external entities, interactions and trust zones that make up the software system, for example as a diagram. External components, such as frameworks and libraries must also be described. See section 3.3.2.1.
Developer results	A description of the architecture or one or more diagrams of the architecture that describe the full architecture.
Criteria	From the architecture inventory one must be able to derive: • All technologies that are used. • All components that comprise the software system. • External entities with which the software system interacts. • Interactions between components themselves and with external entities. • The trust zone of every component and where its interactions cross a trust boundary. • All external modules (parts of the software system that were not developed as part of the software system, but must be regarded as

THR-2	Architecture inventory
	part of the software system) such as frameworks and libraries. All elements of the architecture inventory must correspond with the implementation. The level of detail in the inventory must be sufficient to apply the List of Threats. All trust zones and trust boundaries must be identified and correctly described.
Audit Method	Review the architecture documentation for quality. Also make sure there are no discrepancies between this documentation and the actual implementation.

4.3.3 THR-3: Architectural threats

THR-3	Architectural threats
Description	A list of architectural threats must be generated.
Goal	Ensure that all known architectural threats for the technologies used have been considered.
Rationale	Knowing what threats exists, where they can manifest in the software system, and taking appropriate measures helps to prevent the introduction of design flaws that can lead to security problems.
Developer actions	Select one or more SSF-approved threat libraries that are relevant to the architecture and identify the applicable threats, according to the process that is described in section 3.3.2.2. For external modules, list the threats that can be reasonably expected to occur to the module, as if you would have written the module yourself. Every known vulnerability to the particular version of the module must be listed as a threat. See section 3.3.2.3.
Developer results	A list of architectural threats.
Criteria	• At least the SSF-approved threat libraries that are relevant to the architecture must have been applied. • All *applicable* threats must be listed. A threat that applies to multiple parts of the architecture must be listed for each part to which it applies. • For external modules, all the threats that can be reasonably expected to occur to the module must be listed too, just as if the developer had written the module him/herself. • Every known vulnerability in the particular version of each external module that is used must be listed as a threat.
Audit Method	Review the List of Threats for quality.

4.3.4 THR-4: Threat impact

THR-4	Threat impact
Description	For each threat the impact of it becoming manifest must be correctly described.
Goal	Be able to relate all threats to security requirements.
Rationale	If a threat manifests itself in the implementation, then the impact can be described at the requirements level.
Developer actions	For each threat, find the requirement(s) that it impacts.
Developer results	A list of affected requirements for each threat, including a description how each of these requirements is impacted.
Criteria	• The impact must refer to one or more requirements in the list of security requirements. If none of the security requirements applies, an explanation must be given why the threat is not applicable. • The list of requirements that the threat impacts must be correct and complete.
Audit Method	Study documentation, interviews. Review the List of Threats and the impacted requirements that are associated with them for quality.

4.3.5 THR-5: Threat mitigation

THR-5	Threat mitigation
Description	Every threat must have at least one mitigation or a reason not to implement a mitigation.
Goal	Show that all known threats have been addressed.
Rationale	Knowing what threats exists, where they can manifest in the software system, and taking appropriate measures helps to prevent the introduction of design flaws that can lead to security problems.
Developer actions	For every threat, create at least one mitigation. If this mitigation leads to extra Application Assets or extra architecture, a new iteration of the requirements and architecture analysis should be performed.
Developer results	A list of mitigations or reasons not to mitigate.
Criteria	• For every threat a mitigation must be described, or a reason not to mitigate the threat. • A technical mitigation must be adequate, i.e. after implementing the mitigation the threat should not be able to manifest itself. • A non-technical mitigation must be sound and adequately explained by the security assumptions. • A reason not to mitigate must be sound and adequately explained by the security assumptions.
Audit Method	Study documentation, interviews. Review the List of Threats and

THR-5	Threat mitigation
	corresponding mitigations for quality.

4.4 Implementation

4.4.1 IMP-1: Secure coding standard

IMP-1	Secure coding standard
Description	A secure coding standard must be used.
Goal	Ensure that developers are aware of secure programming constructs and apply them.
Rationale	A secure coding standard helps to prevent security bugs by promoting secure programming constructs and preventing insecure programming constructs. It also provides a basis for code review rules. A secure coding standard often contains platform specific coding rules.
Developer actions	Create or select a secure coding standard. The SSF secure coding principles can serve as a guideline, as can platform specific SSF-approved coding standards and other available secure coding standards.
Developer results	A secure coding standard.
Criteria	• The secure coding standard must at least incorporate the SSF secure coding principles. • If additional secure coding rules are prescribed by the SSF for the platforms and languages that are used, they must be incorporated. • If the above rules or principles are not included in the secure coding standard, this must be explained, and this explanation must be sound. • Other secure coding rules are optional.
Audit Method	Documentation review, interviews. Review of the chosen secure coding standard for quality and adherence to the SSF secure coding principles..

4.4.2 IMP-2: Secure code

IMP-2	Secure code
Description	The software system's code be must follow the secure coding standard.
Goal	Ensure that only secure programming constructs are used.
Rationale	Using secure programming constructs improves the software system's security by preventing known pitfalls.
Developer actions	Develop code following the secure coding standard. If the code does not follow the secure coding standard, or if a secure programming construct is used that is not immediately obvious to a reader of the code, this must be explained in the code's comments.

IMP-2	Secure code
Developer results	Secure code.
Criteria	• All code must follow the secure coding standard. • If a piece of code does not follow the secure coding standard, or if a secure programming construct is used that is not immediately obvious to a reader of the code, this must be explained in the code's comments.
Audit Method	A code audit, using the approach described in section 3.4.1.

4.4.3 IMP-3: Implemented mitigations

IMP-3	Implemented mitigations
Description	Every mitigation must be implemented.
Goal	Ensure all mitigations have been implemented in the software system.
Rationale	By definition mitigations prevent security problems associated with threats that were identified.
Developer actions	Implement all mitigations.
Developer results	Implemented mitigations.
Criteria	All mitigations must be implemented.
Audit Method	A code audit, using the approach described in section 3.4.1.

4.5 Verification

4.5.1 VER-1: Verifications for every mitigation

VER-1	Verifications for every mitigation
Description	Every mitigation must be verified through one or more verifications.
Goal	Ensure that mitigations will be correctly implemented.
Rationale	A mitigation must be correctly implemented to provide security, otherwise the software system may be not as secure as thought, or may even become less secure instead of more secure.
Developer actions	Create verifications for every mitigation. Verifications must be classified and can be of the following types: code review, abuse test, vulnerability scan, fuzzing test, or penetration test. Only if a verification does not fit one of these classes, “non-technical” or “other” can be selected as class.
Developer results	A list of verifications, one or more for each mitigation.
Criteria	• Every mitigation must have at least one verification. • Each verification must be classified as one of • Formal proof

VER-1	Verifications for every mitigation
	• Code review • Abuse test • Vulnerability scan • Fuzzing test • Penetration test • Non-technical • Other • If a verification is classified as “non-technical” or “other” an adequate explanation must be given of what is done to verify the mitigation, why this method was chosen and why this offers sufficient assurance. • If a choice was made not to define a verification this must be adequately explained by the security assumptions.
Audit Method	Review of the list of mitigations and associated verifications for quality. Make sure every mitigation has at least one verification.

4.5.2 VER-2: Adequate verification method

VER-2	Adequate verification method
Description	The chosen method of verification for each mitigation must be adequate, i.e. the verification method must catch all implementation mistakes that result in an inadequate mitigation.
Goal	Ensure that implementation mistakes for the mitigations are caught.
Rationale	When implementing mitigations, mistakes can be made. The correct combination of verification methods ensures that these mistakes are caught in the development process.
Developer actions	Choose one or more verification methods that will catch as many implementation mistakes as possible. The most common mistakes for a mitigation are often known.
Developer results	
Criteria	The joint verification methods for a mitigation must catch as many implementation mistakes as possible that would lead to an inadequate mitigation.
Audit Method	Review the list of verifications and verify that all implementation mistakes for each mitigation are covered, as far knowledge and reason allow for this.

5 Consolidation

5.1 Introduction

The previous sections concerned themselves with what steps have to be taken to deliver secure software and how to verify there is adequate security in a finished product. In other words they described a methodology by which a developer can produce secure software and Controls for Development that can be used by an auditor to make sure the final product is indeed secure.

Even though considerable effort was put into making sure this methodology and its associated Controls would not demand an unreasonable amount of resources from a development team, there is still another important aspect to think of. Software is not a static product, but something that evolves all the time. Strictly speaking this would mean that for every release of the software a new audit of its security would be needed. However, in the real world this is not feasible, due to the costs involved and the amount of work that would demand. Only the largest organizations would be able to afford this, while for others this would pose a prohibitive demand. As a result only a small group of developers would use this framework and that would not help to make software in general more secure. Still, someone who buys a software package needs to be able to estimate if it will meet his of her security needs, even if this specific version was not certified.

To solve this problem this section describes the things a development organization needs to do to make sure a secure product was not just a lucky shot, but the result of careful consideration and good work. Or, to be more concrete, what knowledge and procedures are needed to consolidate the level of security achieved by the developer in a certified product. When an auditor has found sufficient knowledge and appropriate procedures are in place, then a potential user or buyer of later versions of that same product has a basis for trust in the quality of that software system. This would obviate the need for an audit of every released version of the software.

During the development process there need to be knowledgeable functionaries that are assigned responsibility for certain aspects of the process and the maintenance of the deliverables related to those aspects. These functionaries, who will be called “Owners” in this framework, need to sign off on any decisions that impact their responsibility. If they are to perform their roles adequately they must be formally assigned an adequate mandate to not accept changes in the software system and its associated threats and mitigations if they think this affects security in an improper manner. Even if this means blocking a release.

5.2 Consolidation during the Context Phase

During the Context Phase one starts with establishing those functional and environmental requirements that specify what the software system should do under normal circumstances. This means the development team must have sufficiently documented specifications of exactly what it is the software has to do in all situations considered “normal”. In most cases these specifications will have to be delivered by the party that “owns” the development project for this software system.

Thus the consolidation of security starts with a documented specification of all functional and

environmental requirements.

Once the functional and environmental requirements have been established the development team can perform a systematic analysis of all security requirements that have to be met. This analysis asks for proven knowledge and experience on how to identify Application Assets and how to apply STRIDE properties to the operations in the software system. But this alone will not necessarily result in a good application of it. That asks for a procedure that makes sure these steps have been performed and that the resulting list of security requirements is accepted formally by the Owner, after a review for quality.

The last step[7] in the Context Phase is to formally accept the documented list of all security assumptions. All deviations from the stated security requirements must be explained by security assumptions.

At the end of the Context Phase four documents have to be available and agreed upon:

- A specification of all functional and environmental requirements
- A list of all security requirements
- A list of all security assumptions

If during the development of the software system changes are made that impact the above documents, then they have to be updated and formally accepted again by their respective Owners.

Procedures must be in place to make sure the above is done and to monitor the quality of the results.

5.3 Consolidation during the Threats Phase

During the Context Phase a list of all security requirements was produced. This list is used at the start of the Threats Phase to create a list of functional threats by negating the security requirements.

To know whether the design of the software system results in any threats it is needed to perform an architectural risk analysis. This begins with building an architecture inventory that describes all the components that make up the software system, how they interact and where the different trust zones are. If external modules are used these and their interactions with the software system have to be added to the inventory too. This should result in a set of diagrams and documentation that together describe the architecture of the software system.

Building an architectural inventory, and defining proper trust zones and their boundaries is a skill that is not easily acquired and should be done by an experienced analyst.

The next step is to select an appropriate Threat Library containing known threats for this type of software system. If external modules are used in or by the software system, then for each of these a list of all known vulnerabilities must be made and added to the List of Threats. This requires *current* knowledge of known vulnerabilities, whether they have been solved yet, and of

7 Of course, security assumptions should be documented and maintained during the whole development cycle, not just at the end of the Context Phase. But that *is* the first moment during development this list must be signed off.

possible workarounds if they have not. There must be enough time available to one or more members of the development team to keep this knowledge up to date. Moreover there has to be a procedure that makes sure advice from these members results in an update of the List of Threats.

The List of Threats is used determine if any of these threats impact one or more of the STRIDE properties of the Application Assets. Only those threats where this is the case have to be taken into consideration for mitigation. The rest of the threats can be culled from the list. All remaining threats *must* be mitigated. This should result in a documented overview of all the threats that have an impact on the security requirements and the way in which they will be mitigated. Performing a STRIDE property analysis and selecting effective and efficient mitigations should be done by experienced functionaries to make sure this will gain the most appropriate results.

If a mitigation will or cannot be implemented this has to be explained by one or more security assumptions. Any additions to this list have to be formally accepted by its Owner.

The last step in the Threats Phase is to formally accept the documented list of all relevant threats to the software system, where each threat is related to the security requirement(s) involved and each threat has a mitigation.

At the end of the Threats Phase four documents have to be available and agreed upon:

- A list of all functional threats
- A list of all architectural threats
- A statement of applicability of the selected Threat Library
- A list of known vulnerabilities in external modules

And, extracted from these

- A List of Threats that have an impact on the security of the software system, together with the related security requirements and mitigations.

If during the development of the software system changes are made that impact the above documents, then they have to be updated and formally accepted again by their respective Owners.

Procedures must be in place to make sure the above is done and to monitor the quality of the results.

5.4 Consolidation during the Implementation Phase

When coding the software system's instructions and other constructs errors can be made. In order to prevent this, all programmers involved should have proven knowledge of secure programming. To catch any mistakes they might still make, a systematic code review has to be performed during the development of the software system. Such a code review focuses on three specific areas of interest.

First, does the code written introduce new avenues of attack? If so, these new vulnerabilities need to be remediated. Being able to identify these new weaknesses is not always an easy task,

because this requires in-depth knowledge of the programming language[8], frameworks involved and of the platform on which the software is developed or to be used.

Second, does the code adhere to the chosen Secure Coding Standard? This is an easier task and just needs someone that has an eye for detail and who is intimately familiar with that particular Secure Coding Standard. All deviations from the Secure Coding Standard should be explained in the list of security assumptions or in code documentation.

Code documentation is the third area of interest. During coding a programmer has to make a lot of choices. Sometimes these choices are just about how to do something, but they could also be about why to do it in a particular way that might not be immediately obvious to another or to oneself later on. Therefore these choices have to be documented and a code review has to verify this is done in such a way that it provides enough information to make appropriately informed decisions later on.

Code review should at least be done before every release. If the review finds new and unmitigated avenues of attack, or if it finds unexplained violations of the Secure Coding Standard this should be a reason to not release the software. Inadequate code documentation should be resolved, but does not have to be a reason to block the release.

After every code review the results must be formally accepted by the Owner of the code.

If during the Implementation Phase it turns out to be necessary to change parts of the design[9] and these changes do have an impact on the security of the software system, then this must result in an update of the list of relevant threats and their mitigations, or the list of assumptions. Every change in these lists must be reviewed by their respective Owners and will only be effected after they have been formally accepted by that same Owner.

5.5 Consolidation during the Verification Phase

The best way to consolidate the security of a software system is to verify *all* mitigations have been implemented correctly and no new vulnerabilities were introduced during development. Therefore *every mitigation must be verified*. This can be done through formal proofs, through reviews of code, configurations, designs, or even protocols, and through testing.

In order to facilitate a more efficient evaluation of all verifications they have to be classified and labeled according to one of the following:

- Formal proof
- Code review
- Abuse test
- Vulnerability scan
- Fuzzing test
- Penetration test
- Non-technical
- Other

8 Including any libraries that are used in or by the software system.
9 For example due to limitations in the programming language or the platform on which the software is developed or used.

This makes it easier to assign responsibility for all verifications of a specific class to one or more specialists that are knowledgable in this area that can be made responsible for them. It also enables the bundling of verifications and performing each class of verification at different mandatory moments, where each class can have its own interval or trigger(s).

If there are parts of the software system that have to be verified by means of a formal proof, then this has to be done by a specialist that can prove he or she is certified in this specific area of expertise.

The knowledge and procedures needed for verification through review are described in the previous section.

Designing abuse tests asks for a special mindset, but apart from that this is just the same as all others types of functional testing. Because of that this can be done by the regular test staff, with some additional training to bring them up to speed for this particular type of functional test.

Performing vulnerability scans, fuzzing tests and penetration tests and interpreting their results is not so easily delegated to regular development or test staff as this requires specialists that have proven skills. For vulnerability scans and fuzzing tests this could be learned in a reasonably short time, but a good pentester has taken years to hone his or her skills. It definitely will not be easy to add this to the skill set of regular development team members, unless they already do have extensive experience with penetration testing. Expecting them to do this without such experience would gain unreliable results and little guarantee that the software is actually secure against attacks by hackers. Therefore, if the development team does not already have one or more experienced pentesters in its midst, it is better to let outsiders do the penetration testing.

To make sure all verifications are performed regularly, and that any negative results they might yield lead to improved mitigations a procedure has to be in place that makes sure all verifications are performed at least once before a new release, but preferably more often.

If the software system fails a verification this should be remediated. Alternatively it corresponding threat could be accepted by the Owner. However then this has to be explained by the list of security assumptions.

Preferably software must only be allowed to be released when it either passes all verifications, or there are documented explanations for each failure whose risk is formally accepted by the Owner of the list of mitigations. The latter is only allowed for failures that are associated with minor risks.

Secure Software Foundation

6 Controls for Consolidation

6.1 Introduction

This chapter lists the Controls related to knowledge and procedures that are used by an auditor to verify the developer has created all the right conditions to be able to consistently produce secure software. Only when a developer has proven such conditions are in place will a user or buyer of the products from that developer be able to trust that, even though the latest version of a particular software system might not be certified with a certificate from the Secure Software Foundation, it will offer a sufficient amount of security.

CODE-123	Name of the Control
Description	Description of the Control.
Goal	What will be achieved with this Control.
Rationale	Why the goal must be achieved.
Management actions	Things a software/project manager must do.
Management results	Things a software/project manager must deliver to prove the actions are done properly.
Triggers	The moments when the actions are to be done.
Criteria	Criteria the auditor will use to validate conformity to this Control.
Audit Method	Method(s) that the auditor can use to validate conformity to this Control.

6.2 Knowledge

6.2.1 KNO-1: Areas of expertise

KNO-1	Areas of expertise
Description	The development team must have the necessary areas of expertise, or be able to hire specialists when this expertise is not available.
Goal	Make sure that all developer actions prescribed by this framework can be performed by or for the development team.
Rationale	Without the required knowledge it will be impossible to apply this framework or to achieve a sufficient level of quality to ensure security of the software system.
Management actions	Show that members of the development team have proven skills to perform all actions prescribed by this framework. This can be done by showing members are certified, or by referring to their track record, or by having them pass a test. When a required skill is not (sufficiently) available within the development team an external specialist should be hired to perform that specific task.

KNO-1	Areas of expertise
Management results	Documentation that shows members of the development team are certified for the required knowledge, have a proven track record in it or have passed a test to prove that they have that knowledge.
Triggers	Before someone becomes a member of the development team it has to be shown he or she has sufficient skills in one or more of the areas of expertise mentioned in this Control.

Criteria	The development team as a whole must have current knowledge and experience in the following areas of expertise:
	• Requirements analysis
	  • Identification of Application Assets
	  • STRIDE property analysis to create a list of security requirements
	• Architectural analysis
	  • Defining trust zones and trust boundaries
	  • Known vulnerabilities in external modules
	• Threat and mitigation analysis
	  • Defining effective and efficient mitigations
	  • Defining proper verifications for mitigations
	• Secure programming
	  • Knowledge of secure and insecure code constructs
	  • How to do a systematic code review
	• Security testing
	  • How to write abuse tests
	  • How to execute and interpret vulnerability scans
	  • How to execute and interpret fuzzing tests
	  • How to execute and interpret penetration tests

Audit Method	Verify that all members of the development team are certified for the required knowledge, or have a proven track record in it, or have passed a test showing they have current and sufficient knowledge in one or more of the areas of expertise mentioned in the criteria. Also make sure that all areas are covered by at least one member of the development team.

6.2.2 KNO-2: Documentation

KNO-2	Documentation
Description	The development team must document all decisions made during development cycle, if and when they are relevant to the security of the software system.
Goal	Make sure all decisions, relevant to the security of the software system, are clear to the members of the development team.
Rationale	Developers need to know what is expected of them to do the right things. This means all choices and assumptions relevant to the security of the software system that were made during the development cycle must be documented. This also helps to prevent or resolve problems when during

KNO-2	Documentation
	the development decisions are changed or new decisions are made that impact decisions made previously.
Management actions	When a decision is made during any phase of the development cycle that has an impact on the security of the software system this must be registered in the documentation.
Management results	A properly maintained and current registration of all decisions, relevant to the security of the software system, made during the development cycle.
Triggers	When a decision is made that has an impact on the security of the software system this must be registered in the documentation.
Criteria	The following documents must be available, current, correct and complete: • A specification of all functional and environmental requirements • Diagrams and texts that together describe the architecture of the software system • A list of all threats that have an impact on the security of the software system, together with the related security requirements and mitigations • A list of all security assumptions • A Secure Coding Standard It is preferred to use a revision control system to manage these documents.
Audit Method	Review the documents mentioned in the criteria for completeness and correctness. Also verify that the documents reflect what is actually done.

6.3 Responsibilities

6.3.1 RES-1: Maintenance of documentation

RES-1	Maintenance of documentation
Description	All documents mentioned in Control KNO-2: Documentation must have an Owner who is responsible for the maintenance and quality of the documents.
Goal	Make sure the development team always knows what is expected of them.
Rationale	Not all members of a development team will always be directly involved when decisions are made that impact their work. Nor will they always be in direct communication with all other team members. In some situations it might also be necessary to refer to documented decisions to resolve a problem or conflict. To facilitate this good documentation is needed. Making someone accountable for the contents of these documents reduces the risk of outdated or incomplete documents and all

RES-1	Maintenance of documentation
	consequences that might have.
Management actions	Assign responsibility for the maintenance of the documents mentioned in Control KNO-2: Documentation to an Owner[10]. Clearly define what actions are expected of Owners and what mandate they have to effectuate their responsibilities.
Management results	Properly maintained documentation.
Triggers	At the start of the development cycle the documents mentioned in Control KNO-2: Documentation must be created[11] and have an Owner assigned to them. If an Owner leaves or is not available during the development cycle a (temporary) new Owner must be assigned.
Criteria	• Every document mentioned in Control KNO-2: Documentation must have an Owner. • It must be clear what is expected of an Owner. This entails at least what actions the Owner has to take to maintain the document(s) assigned to him or her (even if the operational aspects of this task are delegated). Where and how to the document(s) are published, so the developers can always access the most current documentation. And what powers the Owner has to make sure he is kept abreast of all information needed to maintain the document(s).
Audit Method	Identify and interview the Owners of all documents mentioned in Control KNO-2: Documentation.

6.3.2 RES-2: Management of changes

RES-2	Management of changes
Description	Changes in the design or implementation of the software system must be managed.
Goal	Make sure changes made to some aspect of the design or implementation of the software system don't lead to a reduction of security, unless this was a carefully considered and deliberate choice.
Rationale	Changes by definition lead to risks. To prevent these risks from having an unforeseen impact on the security of the software system as a whole these changes must be managed. The best way to manage these changes is by assigning this responsibility to someone who has to formally review and accept all changes that have an impact on the security of the software system before the development team is allowed to implement them.

10 This does not have to be one Owner!
11 At that time they still might be empty.

RES-2	Management of changes
Management actions	Explicitly assign responsibility for the management of changes that have an impact on the security of the software system. Clearly define what conditions must be satisfied before such changes can be accepted and are allowed to be implemented.
Management results	Only accepted changes are effectuated in the software system.
Triggers	Every time a change in the design or implementation of the software system is made that has an impact on the security of it this has to be approved beforehand by those responsible.
Criteria	• There has to be a registry of changes, including who has approved of them and when they were accepted. This can be done by using a Change Management system, or by adding this information to the documents mentioned in Control KNO-2: Documentation. • Responsibility for change management has to be explicitly assigned. • It must be clear what mandate change managers have. This entails at least what power they have to block any changes that are not accepted. • Accepted changes must be properly reflected in the list of threats and associated mitigations and/or the list of assumptions.
Audit Method	Review the registry of changes for completeness.

6.4 Processes

6.4.1 PRO-1: Adequate verification process

PRO-1	Adequate verification process
Description	All verifications must have been executed prior to the release.
Goal	Ensure that the released software system has been verified.
Rationale	If verifications are defined, but not executed at least once before the software system is released, the verifications are not effective.
Management actions	Make sure the verifications on the version of the software system that will be released are executed, and make sure that all the verifications are either passed or, when failed, are explained.
Management results	Only software in which all mitigations are verified will be released.
Triggers	At a minimum all verifications have to be executed after the last changes were made and before the software is released.
Criteria	Before the software system is released, all verifications must have been executed on the version of the software system that will be released, and the verification results must not reveal any unaccepted shortcomings in the security of the software system.

PRO-1	Adequate verification process
Audit Method	Study documentation, such as release documentation, test reports, code review reports, etc.

6.4.2 PRO-2: Acceptance of failed verifications

PRO-2	
Description	Verification failures, of which it can be shown that they have no significant practical impact on the security of the software system, have to be formally accepted before the software system is allowed to be released.
Goal	Software must only be allowed to be released when it either passes all verifications, or there are documented explanations for each failure whose risk has no significant practical impact on the security of the software system and that are formally accepted by the Owner of the software system.
Rationale	Only software that meets all its security requirements should be released. However a failed verification does not always have to translate to a practical security problem. If it can be shown this is the case, then the failure does not have to block the release of the software system.
Management actions	Explicitly assign responsibility for the acceptance of failed verifications. Clearly define what conditions must be satisfied before such failures can be accepted and are allowed to not block the release of the software system. Make sure failed verifications in a released version of the software system are explained by the security assumptions.
Management results	If there are failed verifications in the software system, then the release of it is only allowed when it can be shown that they have no significant practical impact on the security of the software system.
Triggers	A failed verification.
Criteria	• Responsibility for acceptance of failed verifications has to be explicitly assigned. • It must be clear what mandate those assigned this responsibility have. This entails at least what power they have to block a release if it contains a failed verification they cannot accept and what conditions have to be satisfied before such a failure can be accepted. • Failed verifications in a released version of the software system must be adequately explained by the list of security assumptions.
Audit Method	Study the results of all verifications and the list of security assumptions.

7 Bibliography

CAPEC: The MITRE Corporation, Common Attack Pattern Enumeration and Classification, , http://capec.mitre.org/, retrieved May 9, 2014

CWE: The MITRE Corporation, Common Weakness Enumeration, , http://cwe.mitre.org/, retrieved May 9, 2014

Jurczyk: Mateusz Jurczyk and Gynvael Coldwind, FFmpeg and a thousand fixes, , http://googleonlinesecurity.blogspot.nl/2014/01/ffmpeg-and-thousand-fixes.html, retrieved May 9, 2014

Rice: David Rice, Geekonomics: The Real Cost of Insecure Software, 2007

Schneier2008: Bruce Schneier, Random Number Bug in Debian Linux, 2008, https://www.schneier.com/blog/archives/2008/05/random_number_b.html, retrieved May 9, 2014

Schönefeld: Marc Schönefeld, Security Focused Code Audit ofJava Applications and Middleware, 2005, http://www.uni-bamberg.de/fileadmin/uni/fakultaeten/wiai_lehrstuehle/praktische_informatik/Dateien/Publikationen/05-4_Schoenefeld_RSA_2005_DEV-203_FINAL.pdf, retrieved May 9, 2014

Shostack: Adam Shostack, Threat Modeling: Designing for Security, 2014

Sprundel: Ilja van Sprundel, Fuzzing, 2005, http://events.ccc.de/congress/2005/fahrplan/attachments/683-slides_fuzzing.pdf, retrieved May 9, 2014

8 Legal matters

8.1 CAPEC License

This document uses texts from the CAPEC web site under the following license:

LICENSE

The MITRE Corporation (MITRE) hereby grants you a non-exclusive, royalty-free license to use Common Attack Pattern Enumeration and Classification (CAPEC™) for research, development, and commercial purposes. Any copy you make for such purposes is authorized provided that you reproduce MITRE's copyright designation and this license in any such copy.

DISCLAIMERS

ALL DOCUMENTS AND THE INFORMATION CONTAINED THEREIN ARE PROVIDED ON AN "AS IS" BASIS AND THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE MITRE CORPORATION, ITS BOARD OF TRUSTEES, OFFICERS, AGENTS, AND EMPLOYEES, DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION THEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

8.2 CWE License

This document uses texts from the CWE web site under the following license:

LICENSE

The MITRE Corporation (MITRE) hereby grants you a non-exclusive, royalty-free license to use Common Weakness Enumeration (CWE™) for research, development, and commercial purposes. Any copy you make for such purposes is authorized provided that you reproduce MITRE's copyright designation and this license in any such copy.

DISCLAIMERS

ALL DOCUMENTS AND THE INFORMATION CONTAINED THEREIN ARE PROVIDED ON AN "AS IS" BASIS AND THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE MITRE CORPORATION, ITS BOARD OF TRUSTEES, OFFICERS, AGENTS, AND EMPLOYEES, DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION THEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

CWE is free to use by any organization or individual for any research, development, and/or commercial purposes, per these CWE Terms of Use. MITRE has copyrighted the CWE List, Top 25, CWSS, and CWRAF for the benefit of the community in order to ensure each remains a free

and open standard, as well as to legally protect the ongoing use of it and any resulting content by government, vendors, and/or users. MITRE has trademarked ™ the CWE and related acronyms and the CWE and related logos to protect their sole and ongoing use by the CWE effort within the information security arena. Please contact cwe@mitre.org if you require further clarification on this issue.

8.3 Copyright

We welcome any comments or suggestions you may have with regard to this document and the information it contains. It is expected that the Framework Secure Software will develop over time, and we invite you to contribute to its development.

Please send us an email at info@securesoftwarefoundation.org or visit our website at http://www.securesoftwarefoundation.org for further contact information.