Building Security Requirements with CLASP

John Viega
Secure Software, Inc.
2010 Corporate Ridge, Suite 820
McClean, VA
+1 (703) 749-3871
viega@securesoftware.com

ABSTRACT

Traditionally, security requirements have been derived in an *ad hoc* manner. Recently, commercial software development organizations have been looking for ways to produce effective security requirements.

In this paper, we show how to build security requirements in a structured manner that is conducive to iterative refinement and, if followed properly, metrics for evaluation. While requirements specification cannot be a complete science, we provide a framework that is an obvious improvement over traditional methods that do not consider security at all.

We provide an example using a simple three-tiered architecture. The methodology we document is a subset of CLASP, a set of process pieces for application security that we have recently published, in conjunction with IBM/Rational.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications – *methodologies.*

General Terms

Management, Measurement, Documentation, Reliability, Security.

Keywords

Security requirements, application security, security process.

1. INTRODUCTION

Work in requirements has primarily focused on eliciting and representing concrete business requirements. Security is rarely at the forefront of stakeholder concerns, except perhaps to comply to basic standards, even if it is a set of de facto technologies, such as SSL. Perhaps for this reason, there has not previously been a cohesive methodology for deriving security requirements. As a result, books and resources on secure software engineering [1, 4] largely describe *ad hoc* methodologies for software security engineering, particularly when addressing security requirements.

In this paper, we describe how to take a resource-centric approach to deriving security requirements. This approach results in much better coverage of security requirements than do *ad hoc* methods or technology-driven methods. For instance, many businesses will quickly derive the business requirement "Use SSL for security", without truly understanding what requirements it is addressing. For instance, is SSL providing entity authentication, and if so, what is getting authenticated, and with what level of confidence? Many organizations overlook this, and use SSL in a default mode that provides no concrete authentication.

Our approach to software requirements is a subset of our work on CLASP (Comprehensive, Lightweight Application Security Process), which is a set of process pieces for helping development organizations improve the security of their software. The process is publicly available from IBM's web site [2], and is already being used by several software development organizations.

In Section 2, we present our methodology. In Section 3, we walk through an example, based upon a traditional three-tier architecture.

2. REQUIREMENTS METHODOLOGY

At a high level, the CLASP approach to formulating security requirements consists of the following steps:

1. Identify system roles and resources.
2. Categorize resources into abstractions.
3. Identify resource interactions through the lifetime of the system.
4. For each category, specify mechanisms for addressing each core security services.

After one goes through the initial requirements solicitation process, one will generally identify gaps that require iteration. Similarly, iteration may be desirable after elaborating on requirements during the design process. This seems an unavoidable fact, that requirements will often need to change as the understanding of a system changes. We will see in our example below how requirements may evolve as one works through this process.

The basic idea behind the way CLASP handles security requirements is to perform a structured walkthrough of resources, determining how they address each core security service throughout the lifetime of that resource.

2.1 Identifying Roles and Resources

Roles are generally already defined in the course of architecting and designing a software system, which we can leverage. The main purpose of roles in so far as security goes is to identify the owners and users of resources, as well as access controls between resources. For this reason, it is good to identify which roles are parameterized with respect to permissions, and which ones are not.

For example, users constitute a generic role in the system, but there generally is not one set of user permissions and privileges. Instead, each user will have his or her own permissions and privileges, separate from other users. In contrast, even when multiple people share administrator duties, administrative privileges are often shared among all administrators, making the role non-parameterized.

Additionally, it is good to introduce attackers as a role in the system, and attach a reasonable threat model to each of those attackers. Generally, this would involve assuming that the attacker has not only complete control over any network resources but also assuming that he or she has insider access to the development organization (e.g., may be an employee, a friend of an employee or an ex-employee).

Resources are any piece of data or functionality that can be used by a program. This includes not only application data such as personal information of users, but also many kinds of resources that are often implicit or overlooked in specifying a software system:

Databases and database tables

Configuration files

Cryptographic key stores

ACLs

Registry keys

Web pages (static and dynamic)

Audit logs

Network sockets/ network media

IPC, Services and RPC resources

Any other files and directories

Any other memory resource

CLASP recommends that, when the information is known, break down each resource in as granular a way as possible, such as by identifying individual database tables, instead of simply the database. This is important, as introducing more detail will generally reveal unrecognized requirements. It is often useful to make resources hierarchical.

It is important to note that network media is a resource of its own. Data resources will often be stored in memory, placed onto a wire, received in memory on the other end, and then stored on disk. In such a scenario, we often will not want to address the security of the data in a vacuum, but instead in the context of the resource the data inhabits. In the case of the network media, we need to specify how to protect that data when it traverses the media, which may be done in a generic way, or in a way specific to the media.

2.2 Categorizing Resources

Functional security requirements should show how the basic security services are addressed for each resource in the system, and preferably on each capability on each resource. This generally calls for abstraction to make the process manageable. Security requirements should be, when possible, abstracted into broad classes, and then those classes can be applied to all appropriate resources/capabilities. Then, if there are resources or capabilities that do not map to the abstractions, they can be handled individually.

For example, end-user data that is generally considered highly sensitive can often be lumped into a “User-Confidential” class, whereas public data could be lumped into a “User-Public” class. Requirements in the first class would tend to focus on circumstances in which access to that data can be granted to other entities.

Categorization should usually include an indication of which role or roles can own or use a particular resource, as well as the potential value of the resource.

Categories can be applied either to data resources, or to individual capabilities by specifying a requirement that the specific resource or capability should be handled in accordance with the security policy of the particular protection class. When applied to data resources, requirements should be specified in the abstracted class for any possible capability, even if some data elements will not have the capability.

While it is often the case that most data resources will lump into a few reasonable abstractions, it is also often the case that other system resources such as the network, local memories and processors do not line up with user data requirements.

Another advantage to categorization is that requirements can be utilized as organizational control standards, and applied across projects. This provides an enforcement mechanism to establish a baseline security posture across and entire organization by making these control standards “global” requirements for all software products. It may additionally save time, compared to addressing each resource individually.

2.3 Identifying Resource Interactions

Security requirements on data change through the lifetime of the data. For instance, the security of user-confidential data may be the responsibility of that user when it resides on that user’s own machine, meaning that the client-side application does nothing special to protect it, relying instead on whatever protection mechanisms happen to be in place, such as firewalls. When the client application sends that data over a network to middleware, protection against network-based attacks is usually desirable, as it is when going from the middleware to the database. When in the middleware, there can be exposure to new roles, including other accounts using the middleware and people with account access or physical access to the machine hosting the middleware, and this can require special protective measures. Finally, when stored in the database, data should probably be stored in as secure a manner as feasible, since data may live there for a long period of time.

In this example, requirements on the user data change as the data interacts with other resources in the system, including the network, the database and so-on.

One easy way to specify interactions is to step through the lifetime of a piece of data, at each point identifying what other resources may have interactions with that resource. If resource capabilities are being taken into account, the exercise should be done once for each capability on a resource, instead of just once on a resource.

Generally, it is fine to do this on a per-data category basis, but a per-resource walkthrough is useful as a defense-in-depth mechanism to determine whether the chosen categorization is adequate.

Additionally, we should also look at how each role might interact with data of a given category. For example, will there be data that an administrator may need to modify (e.g., a password, if the user forgets it)?

Note that, as we step through the system in this way, we may iterate on earlier steps, as we challenge our own assumptions about how data will be used.

2.4 Requirement Specification

For each category of resource, we specify how to address each of the core security services, when interacting with other resources.

The core security services as defined by CLASP are:

Authorization: what privileges on data should be granted to the various roles at various times in the life of the resource, and what mechanisms should be in place to enforce the policy. This is also known as access control, and is the most fundamental security service. Many other traditional security services (authentication, integrity and confidentiality) support authorization in some way.

We will generally want to consider here resources outside the system that are in the operating environment that need to be protected, such as administrative privileges on a host machine.

This is the service under which to specify both how one grants and enforces access control policies, and what roles have access to what capabilities (and under what circumstances).

Authentication and integrity: How is identity determined for the sake of access to the resource, and must the resource be bound to an identity in any strong way? For instance, on communication channels, do individual messages need to have their origin identified, or can data be anonymous?

Generally, requirements should specify necessary authentication factors and methods for each endpoint on a communication channel, and should denote any dependencies, such as out-of-band authentication channels (which should be treated as a separate system resource).

Integrity is usually handled as a subset of data origin authentication. For instance, when new data arrives over a communication channel, one wants to ensure that the data arrived unaltered (whether accidental or malicious). If the data changes on the wire (whether by accident or malice), then the data origin has changed. Therefore, if we validate the origin of the data, we will determine the integrity of the data as well.

This illustrates that, if authentication is necessary in a system, it must be an ongoing service. An initial authentication is used to establish identity, but that identity needs to be reaffirmed with each message.

Identity is the basis for access control decisions. A failure in authentication can lead to establishing an improper identity, which can lead to a violation of access control policy.

Confidentiality (including privacy): Confidentiality mechanisms such as encryption are generally used to enforce authorization. When a resource is exposed to a user, what exactly is exposed, the actual resource, or some transformation? Requirements should address what confidentiality mechanism is required, and should identify how to establish confidentiality (usually requiring identity establishment).

When this involves using encryption, requirements should focus on the algorithm and its parameters for initialization.

Availability: Requirements should focus on how available a resource should be, for authorized users. This is probably the most difficult security service to specify well, since most reliability issues can be availability issues. Here, we generally rely on our generic software engineering expertise.

Accountability (including non-repudiation): What kind of audit records need to be kept to support independent review of access to resources / uses of capabilities? I.e., what logging is necessary? Remember that log files are also a data resource that need to be specified and protected.

For each of these security services, we should build requirements that are specific enough to be useful. We recommend using an extension of SMART requirements [3] we call SMART+, illustrated in Table 1.

Table1: SMART+ Requirements

Specific	There should be as detailed as necessary so that there are no ambiguities in the requirement. This requires consistent terminology between requirements.
Measurable	It should be possible to determine whether the requirement has been met, through analysis, testing or both.
Attaintable	One should validate that the requirement can indeed be implemented, under some set of circumstances.
Reasonable	While the mechanism or mechanisms for implementing a requirement need not be solidified, one should do some validation to determine whether meeting the requirement is possible given other likely project constraints.
Traceable	Requirements should also be isolated so that they are easy to track and validate throughout the development lifecycle.
+ Appropriate	Requirements should be validated, ensuring that they not only derive from a real need or demand, but also that different requirements wouldn't be more appropriate.

Since “attainable” and “reasonable” go hand-in-hand (reasonable usually being a specialization of attainable), we generally merge these two into “reasonable”, and let “appropriate” act as the “a” in SMART.

3. EXAMPLE SECURITY REQUIREMENTS

We illustrate the CLASP methodology for deriving security requirements with a simple example of a three-tiered application. In the interest of space, we do not walk through a complete example, but only enough to give a flavor of how to use the CLASP requirement methodology.

In this sample application, we imagine that clients will connect to a service over the web, where the service simply allows users to store information about their contacts, with the intention that the users are the only people who can view their own contacts.

This example is meant to be illustrative of a first iteration on the system. By the end of this example, we will have identified requirements that will strongly suggest iterating.

3.1 System Roles

We start by enumerating the roles. We may choose a standard set of roles, as shown in Table 2.

Table 2: Sample System Roles

Role	Generic	Description
User	Yes	These are valid users of the system who have already created an account.
System	No	The application server is represented by a role. This is separate from the “Admin” role, because there may be resources that the system will need to access that the admin should not have to be able to see.
Admin	No	These are users who have administrative access to the system. In this application, their role is restricted to account management, log monitoring and general availability.
Anonymous	No	This role represents people without accounts who may attempt to interact with the system. In this sample application, they have no capabilities, other than being able to sign up for an account.
Attacker	No	Anyone attempting unauthorized access to resources.

3.2 Resources

We then determine a hierarchy of resources that is broken down to an initial level of detail. Subsequent iterations will generally occur as architecture and design evolve, resulting in more concrete resource descriptions.

Table 3: Example Resource List

ID	Description	Owner role(s)	User capabilities
1.	User data	User	Varies (see below)
1.1	Name	User	User (cr) System (r) Admin (d)
1.2	Password	User	User (cw) System (v)
1.3	Contacts	User	User (crwd) System (User proxy)
2.	Compute resources	Varies	Varies
2.1	User CPU	User	User
2.2	User memory	User	User
2.3	User disk space	User	User
2.4	User machine admin info …	User	User
2.5 -	Similar information for middleware and DB servers	Admin	System (User proxy)
3.	Network resources		
3.1	Network between client and app server	Attacker	User, System (rw)
3.2	Network between app server and database (Direct cable connection)	Admin	System (rw, User proxy)
4	Web content (dynamic and static)		User (r) System (cw) Admin (r,w)
4.1	Static web content	Admin	User, Anonymous (r) Admin (r,w)
4.2	Dynamic web content	System	User (r), System (cw)
4.3	Cookies holding authentication information	System	System (crw)
5	Back-end executable content	Admin	User (rx) Admin (rw)
6	Database	Admin	
6.1	Table “user data”	Admin	System (crwd, User

			proxy, user's data only) Admin (crwd)
6.2	Table "account info"	Admin	System (crwd, User proxy, user's data only) Admin (crwd)
6.3	Admin password to database	Admin	Admin (crw)

Note that the above table does include denotation of basic capabilities on resources, with "r" standing for read, "w" for write, "c" for create and "d" for delete. In 1.2, we use "v" for validate, meaning that the system should be able to validate a password, but hopefully not able to read it.

We should note that there may be assumptions made in the resource chart. For instance, in this example chart we make it clear that there is a requirement for password-based authentication. In this case, it was a known business requirement to use passwords, for the sake of end-user usability. If not, we may have instead have mentioned an "authentication token or tokens", refining this into something more concrete in subsequent iterations.

3.3 Resource Categories

We might find that this set of resources is well suited to a few resource categories:

Table 4: Example Resource Categorization

Category	Description	Resources
User-Highly-Confidential (UHC)	Resources that belong to the user, and should not be accessible to others, even to the system.	1.2, 2.1, 2.4
User-Confidential (UC)	Resources that belong to the user, but the system may access for the benefit of the user.	1.3, 2.2: web content only 2.3: cookie storage only
User-Low-Confidential (ULC)	Resources that belong to the user and should not be available to the general public, but may be available to administrators.	1.1
System-Private (SP)	Resources that should not be accessible, except to the system.	3.2, 2.5-…, 4.3, 5, 6.1, 6.2, 6.3
System-Consumable (SC)	Resources owned by the system that are for the benefit of the user (e.g., communicating of data to the user)	4.1, 4.2
Public (PUB)	Public or otherwise untrusted resources.	3.1

3.4 Resource Interactions

We now look at resource interactions at the level of our categorizations, giving only a partial example, in the interest of space:

Table 5: Sample Resource Interactions

Res. #1	Res. #2 or role	Cases	Notes
UHC			The user password is the only data considered highly confidential to the user.
	UC	n/a	No data that is highly confidential to the user interacts with any other user data that is confidential.
	ULC	1.2 with 1.1	The user password is associated with the user name. They must be used in conjunction in order for a connection to be established. Beyond being data that is conceptually tied together, they have no other interactions.
	SP	1.2 with 5, 6.2, 3.2, 4.3 and 2.5 - …	It is clear that the system must store and process either the password itself, or some function of the password. This may be a place where we will have to compromise in our ideals in order to balance non-security goals with security goals.
	SC	1.2 with 4.1 and 4.2	The web is the user's interface for setting and resetting passwords, and for logging in.
	PUB	1.2 with 3.1	In order to authenticate successfully, the password will need to traverse over an insecure medium.
	User		The user should have access to change his own password, but shouldn't have any ability to learn about the passwords of other users.
	System		Ultimately, the system uses the password to authenticate a user, and should have no need for it beyond that, except as a proxy for the wishes of the user, who might want to change a password. Also, note that the user might have forgotten a password, and so the system may want to be able to deal with this situation.
	Admin		The administrator may need to intervene when there is a

			forgotten password. We note that this leads to social engineering attacks, so we will look for ways to avoid it.
	Anon / Attacker		This is information that can, in conjunction with the user name, lead to additional capabilities for an attacker, so we must do our best to make sure it is not disclosed to people with this role, under any circumstances.

Note that this information is also useful as the foundation for a subsequent security analysis of a system. We can, for example, note that, without finer granularity, the "Admin" role should probably be assumed to have access to any resources labeled as belonging to "System", which may not have been the original intent, and thus a security gaffe.

This may call for finer definition of roles (for example, by separating out the sysadmin from the DB admin from the technical support staff), or it may call for a more coarse definition of roles, lumping Admin and System together. We identify further places where iteration should be considered in Section 3.6, below.

3.5 Requirements

We will focus on deriving requirements for User-Highly Confidential resources. This is structured by looking at each resource category or role that the UHC resource might interact with. Even though, as specified, the only UHC data is a password, we start by assuming that UHC data may be arbitrary data, such as a social security number. In the interest of space, we omit requirement numbers, which are generally useful, especially for ease of tracability.

Table 6: Sample CLASP Security Requirements

Resource Category or role	**Service**	**Requirement**
User (owner)	Access control	The owning user cannot grant access to UHC data to any other entity in the system, unless so specified by another requirement.
Default	Access control	For UHC data that does not have validation as an operation, there must be no practical way for the data to be operated upon by any entity other than the owning entity.
System	Access control	For data that does have validation as an operation (i.e., passwords), there must be a transformation of the password that can be granted to System, and may be used to implement this operation, but there must be no way to perform other
User (owner)	Authentication / Integrity	UHC resources must not rely on any auth services provided by other resources.
User (owner)	Authentication / Integrity	UHC resources must have their integrity preserved if they are going to be stored by any role or resource other than the user himself. The mechanism for preserving integrity must be HMAC-SHA1, which must be performed on the ciphertext image of the data, and must be keyed with a randomly selected 128-bit value that must reside only on the user's computer.
System	Authentication / Integrity	If the core UHC datum is a password, then it must be possible for the system to validate that the password belongs to the user associated with the data.
Default	Confidentiality	UHC data must be confidentiality protected using AES-CTR, keyed with a randomly selected 128-bit value that must reside on the user's computer.
Default	Confidentiality	The confidentiality protection must be updated every time the data changes, using a new Initialization Vector managed by the client. The IV must be set to a random 128-bit value every time the key is chosen, and must be incremented by 1 every time the password is changed.
Default	Confidentiality	The IV must always be included with the encrypted data when communicated to other parties.
		The IV must always be checked, to ensure it is greater than the previous value seen, any time capture-replay attacks may be an issue. If the IV does not increment in such a scenario, the message must be rejected.
Default	Availability	There are no special requirements for UHC resources. There will be for System-Consumable resources, which need to be accessible to the outside world. To get concrete, we will probably want to specify defensive practices about catching and handling exceptions.
System	Accountability	If the UHC data is a password, any time the validation operation

		is performed, this fact must be logged to the system log, along with the date, user name and an indication of whether the attempt was successful.

Note that this set of requirements is relatively simple, because the data is not meant to be transferred among roles. If we were to tackle a different class of data, we would have to specify how to protect the data when in transit over a Public resource. Simply specifying "use SSL" wouldn't be enough, even if a version number were attached. Instead, we would need to identify what specific mechanisms we are using to provide each of the security services.

3.6 Iteration

In the course of building these requirements, we can apply the SMART+ test, and determine that we have some deficiencies that we can overcome by further iteration. Particularly, we reference resources that we had not initially identified, that we do not have specific enough information about, such as "random numbers", cryptographic keys for HMAC and for AES-CTR, an "initialization vector" and the "system log".

The requirements would also read better if we took a capabilities-level view of the system. For example, we could have a cleaner set of requirements by specifying encryption of UHC data separately from decryption of UHC data.

4. CONCLUSIONS AND FUTURE WORK

CLASP provides the first structured methodology for deriving security requirements of software systems. While it is obviously far more effective than an *ad hoc* treatment of security requirements, this methodology is still new. As a result, we do not yet have enough data to identify areas that can be improved, though we are working with several commercial development organizations that are using CLASP, in order to do so.

We anticipate refinements to CLASP's methodology for requirements, based on industry feedback. We also have begun building a large set of sample requirements that may be used as a template for projects, or for organizational control standards. We hope that a comprehensive template will help minimize iteration.

5. ACKNOWLEDGMENTS

Thanks to Jane Tudor from Secure Software for her initial implementation work on requirements with CLASP, and for reviewing this paper. Also, thanks to Neil Hixon and Paul DeGraff from Depository Trust and Clearing Corporation for suggestions leading to refinement of the methodology in this paper.

6. REFERENCES

[1] Howard, M. and LeBlanc, D. *Writing Secure Code, 2nd Edition*. Microsoft Press, Redmond, WA, 2003.

[2] IBM. *RUP Plug-In for secure application development v1.0* www-128.ibm.com/developerworks/rational/library/04/r-3315/

[3] Mannion, M. and Keepence, B. SMART Requirements. *ACM SIGSOFT, SE Notes 20*, 2 (Apr. 1995).

[4] Viega, J. and McGraw, G. *Building Secure Software*. Addison-Wesley, Reading, MA, 2001.