Chapter 5
OSS security certification

Abstract Open source software is being increasingly adopted for mission and even for safety-critical applications. Experience has shwon that many open source software products have achieved adequate functionality and scalability. Security, however, requires a specific analysis, since open source software development does not usually follow security best practices. Indeed, the lower number of security events involving open source software may be ascribed to its smaller market share rather than to its robustness. In this chapter we start by taking a closer look to the meaning of the open source label, discuss the connection between licenses and certificates. Then, we summarize the debate on open source security and discuss some issues pertaining to open-source assurance activity and to open source security certification.

5.1 Open source software (OSS)

The debate between open source and closed source supporters dates back to the origin of software and is still far away from a conclusion. The advent of the Internet has made this contraposition even more harsh. The closed software model was originally introduced in the 1970s, when software commercialization became a reality. Computer software was treated as a company asset, to be protected from competitors who might otherwise reproduce, study or modify the code, to resell, use, or learn from the product. The closed source paradigm allowed software houses to protect their products from piracy or misuse, reverse engineering and duplication. Also, the closed source paradigms allowed software suppliers to preserve competitive advantage and vendor lock-in.

By contrast, the concept of free software was born as a social movement (1983) aimed to protect user's rights to freely access and modify software. In 1985, Richard Stallman founded the *Free Software Foundation* [20] (FSF) to support such a movement. In 1998, a group of members of the FSF replaced

the term *free software* with *open source*, in response to Netscape's January 1998 announcement of a source code release for their Netscape Navigator browser.

As Stallman himself pointed out, OSS development departs radically from commercial software form the very beginning: an open source software solution is not planned as a product to be offered to the market to achieve a profit. Often, the initial idea and the design do not take place within the boundaries of a traditional business organization. Some OSS projects, such as business applications or middleware, are entirely autonomous. Others exist as components, as for instance, part of the Linux kernel. Large, mature projects such as the Apache Web server have they own autonomous evolution strategy, although benefiting from the contributions of hundreds of outside submitters. Whatever the status (component or application) of an open source project, it will remain useful only as long as it has a well-specified assurance process, that is, its maintainers properly update and test it. In this Chapter we shall discuss when and how software certification can become a part of the assurance effort of open source projects.

5.1.1 Open Source Licenses

The action of the FSF was instrumental for clarifying the semantics of the open source label: an open source software product is a software product made available under an open source license. Today, open source licenses are not just declarations which grant to the user unlimited rights of accessing and modifying the software product's source code. Rather, licenses specify in detail who is entitled to access the source code, and the allowed actions to be performed on it. The reader should not miss the conceptual link between a software license and a certificate: both of them assert some properties of a software product, and both may be distributed and checked in a digital format. A major difference between licenses and certificates is that the former seldom deals with software properties but, rather, the assertions made in a software license focus on the rights and obligations of both purchaser and supplier concerning the software product's usage or redistribution. The wording of the license may be such that the software supplier has no obligation whatsoever, not even that the software program will be useful for any specific particular purpose. In the license, however, the purchaser may get permission from the supplier to use one or more copies of software in ways where such a use would otherwise constitute infringement of the software supplier's rights under copyright law.[1]

[1] Such use (e.g., creating archival copies of the software) may be permitted by law in some countries, making it unnecessary to explicitly mention it in the license. In effect, this part of a proprietary software license amounts to little more than a promise from the software supplier not to sue the purchaser for engaging in activities that, under

Some licenses give very limited rights to the purchaser. Many proprietary licenses are *non-concurrent*, that is, do not allow the software product to be executed simultaneously on multiple CPUs, or set a limit to the number of CPUs that can be used. Also, the right of transferring the license to another purchaser, or to move the program from one computer to another may be limited in the license.[2] Finally, proprietary software licenses usually have an expiration date. The license validity may range from perpetual to a monthly lease. A time-limited license can be self enforcing: the supplier may insert into the software product a security mechanism that, once the license has expired, will disable the software product. An interesting by-product of time-limited licenses is the emerging need for certifying that security mechanism used to enforce license expiration do not impair the software performance or dependability while the license is still valid.

Software suppliers have traditionally been lax on license enforcement and then individual purchasers of proprietary software tend to pay little attention to compliance. However, most large companies and organizations, including universities, have established strong license compliance policies.

Open source is sometimes seen (or presented) as a way out from the complex problem of guaranteeing compliance to proprietary software licenses. However, OSS is itself distributed under a license and more than 30 licenses actually exist [30]. To help in establishing some degree of uniformity, the *Open Source Initiative* (OSI) [36], jointly founded by Eric Raymond and Bruce Perens in February 1998, has promoted since long a specification (called *Open Source Definition* (OSD)) of what must appear in a license in order for the software covered by it to be considered open source. Licensers are however free to use licenses that go beyond OSD minimum requirements, in the sense of providing more rights to the user. Thus, OSD-compatible licenses are not all the same.

OSI's open source definition mandates that the license of an open source software product must comply with ten criteria [26], described as follow.

1. *Free redistribution.* An open source license must permit anyone who obtains and uses the covered software to give it away to others without having to pay a royalty or other fee to the original copyright owner(s).
2. *Access to source code.* All types of open source licenses require everyone who distributes the software to provide access to the program source code. Often, distributors provide the source code along with the executable form of the program, but the license does not bind them to do so; for instance, they could make the code available via Internet download, or on other media, free or for a reasonable fee to cover the media cost.

---

the law of the country where the purchase is made, would be considered exclusive rights belonging to the supplier itself.

[2] For instance, OEM Windows licenses are not transferable. When the purchaser does not longer use the computer where the Windows software is pre-installed, the Windows license must be retired.

3. *Derivative works.* An open source license must allow users to modify the software and to create new works (called *derivatives*) based upon it. An open source license must permit the distribution of derivative works under the same terms as the original software. This provision, together with the requirement to provide source code, fosters the rapid evolution pace of open source software.
4. *Integrity* of the author's source code must be preserved.
5. *No discrimination on users.* An open source license does not discriminate against persons or groups. Everybody can use open source software, provided they comply with the terms of the open source license.
6. *No discrimination on purpose.* An open source license does not discriminate against application domains. In other words, the license may not restrict anyone from using the software based on the purpose of such usage. Specifically, it may not restrict the program from being used for commercial purposes. This permits business users to take advantage of open source products for commercial purposes.
7. *License Distribution.* The wording of an open source license must be made available to all interested parties, and not to the purchaser alone.
8. *Product Neutrality.* An open source license must not be specific to a single software product.
9. *No transfer of restrictions.* An open source license on a software product must not restrict the use of other software products, both open source and proprietary. In other words, an open source license must not mandate that all other programs distributed together with the one the license is attached to are themselves open source. This clause allows software suppliers to distribute open source and proprietary software in the same package. Some widespread licenses, including the GPL (*General Public License*) presented below, require that all software components "constituting a single work" to fall to under the GPL if anyone of them is distributed under GPL. This requirement may seem to have been spelled out clearly, but wrapping and dynamic invocation techniques have sometimes been used as a work-around to it.
10. *Technology Neutrality.* An open source license must not prescribe or supply a specific technology.

It is beyond the scope of this book to provide a detailed analysis of all open source licenses. Here, we shall limit ourselves to outlining the main features of some widespread ones. The interested reader is referred to our main reference, the classic book [30]. However, it is important to remark that organizations and individuals supporting the open source paradigm (including the authors of this book) firmly believe that the benefits given by a community of gifted and enthusiastic software developers working at the evolution of a software product are much more important than the (often illusory) advantages of protecting the intellectual property rights on it.

- GNU *General Public License* (GPL) is one of the first open source licenses and still by far the most widely used. It is considered a liberal license inasmuch the original programmer does not retain any right on modified versions of the software. Richard Stallman and Eben Moglen created the GPL and started the Free Software Foundation to promote its use. For instance, Linux is distributed under a GPL license.
- The *Mozilla Public License* (MPL) is another popular open source license. It came about to distribute the original Mozilla open source web browser. It is less liberal than GPL, inasmuch it requires the inclusion or publishing of the source code within one year (or six months, depending on the specific situation) for all publicly distributed modifications.
- The *Berkeley Software Distribution* (BSD) License was one of the earliest non-proprietary licenses, and follows a very different philosophy than GPL. BSD permits users to distribute BSD-licensed software for free or commercially, without providing the source code; they also may modify the software and distribute the changes without providing the source code. A major difference between BSD and GPL is that organizations or individuals who create modified versions of software originally licensed under BSD can distribute them as proprietary software, provided that they credit the developers of the original version. Two other widespread open source licenses, the Apache Software License and the MIT License, are very similar to the BSD License.

The differences between GPL and other open source licenses become very relevant when a user creates a derivative from existing open source code. In this case, with GPL the license is inherited. The new code must be distributed under the same license as the original version. This may not be true for other licenses.

5.1.2 Specificities of Open Source Development

The rapid pace of evolution and the multi-party development fostered by the open-source licensing policies described above make open source software products very different from proprietary ones. Generally speaking, open source software is developed and modified by programmers who devote their time, energy and skills without receiving any direct compensation for their work [24]. In this context, the whole relation between software purchasers and software suppliers changes dramatically. Open source gives much more power to customers who need customized products that fit their business activities. If a customer chooses to use open source software, say, for human resource management, a software supplier can offer to customize the software for that individual customer. In this case the customer will be charged a fee not for using the software itself, but for the service of customizing it.

Before discussing OSS security certification, it is therefore important to look at the process of developing OSS [43]. OSS has fostered a new software development style based on a heterogeneous mixture of existing methodologies and development processes. It does not provide any standard criteria to select activities for the different projects; instead, it is up to developers to agree on which methodology is more suitable for them [16]. Rather than by a specific set of activities, the OSS development process is characterized by its rapid release cycle, for teams that put together developers with diverse skills and competences, for fast rate of code change over time and for the use of readable code as a way to satisfy the need of a clear and unambiguous documentation.[3] Code is seen as the first specification of open source systems and, as a result, those systems are often otherwise undocumented.

The above description may convey the idea that OSS "emerges" from a Wikipedia-style "democratic" cooperation rather than from a disciplined development process. This is however not the case for some major OSS projects. In particular, here we are interested in the level of coordination which is crucial for testing and security assurance.

As an example, let us briefly consider the Linux development process. We will come to certifying Linux distributions in Chapter 6. Bill Weinberg[4] describes the Linux kernel development and maintenance process as a "benevolent dictatorship". It is probably more like feudalism: while contributions to the kernel come from developers worldwide, the authority of including and integrating them in the Linux kernel belongs to around 80 maintainers, each responsible of a subsystem of the Linux kernel. Each subsystem has its own versioning; sets of subsystems are integrated into *patch sets* that, in turn, are used to set up "experimental" kernel versions (in the past, these corresponded to odd-numbered kernel releases like 2.3.x and 2.5.x). When the highest authority (Linus Torvalds and his team) considers an experimental kernel ready for deployment, a new production kernel is generated and handed off for testing to the production kernel maintainers, who are responsible for the entire testing process. Starting from production kernels, companies and consortia create Linux distributions aimed at the business or embedded systems domain.

Linux kernel maintainers can also select external OSS projects, such as device drivers, to be integrated into a Linux subsystem. Some examples are security-related; for instance, the *National Security Agency* (NSA) *Secure Linux* was adopted as a standard build option in the 2.6 Linux kernel.

---

[3] OSS is seldom developed from a stable specification and *a priori* software requisites are usually vague.

[4] The description below is partly based on his contribution "The Open Source Development Process", originally published on the *Embedded Computing Design* magazine, `http://www.embedded-computing.com/departments/osdl/2005/1/`.

5.1.2.1 The Open Source Code Assurance

Understanding the general quality assurance of OSS code is particularly important for the integration of the security certification process into OSS development. Let us start with a formal definition of *Software Assurance* (SwA): an activity aimed at increasing the level of confidence that a software product operates as intended and is free of faults. In a traditional, lifecycle-based software development process, assurance includes a number of tasks to be carried out by developers and testers along the software lifecycle.

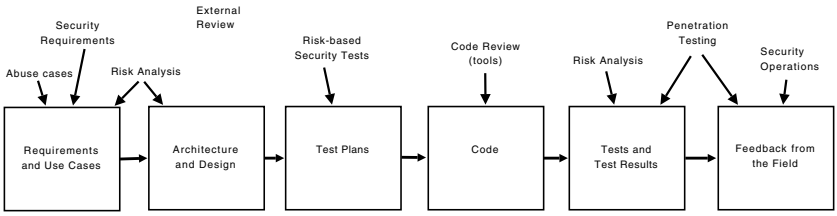

Fig. 5.1: Assurance tasks in a traditional, lifecycle-based development process

Security assurance activities for OSS code could be performed at several points in the code life cycle. Contributions to major OSS projects like Linux are strictly monitored and must meet quality standards; here, we are interested in the assurance process used to keep these standards. Of course, different OSS projects will use different assurance procedures, but some conditions are verified for all OSS projects. Upon submission, a contribution to an OSS project must be *well-formed*, that is, coded and packaged according to well-established OSS conventions. The first assurance activity is usually checking for novelty and interest of the contribution, that is, the properties (either functional or non-functional) it would add to the project. These checks are usually made by core members of the community (in the case of the Linux kernel, by the subsystem maintainers). Once a contribution makes is accepted into a OSS project, it will be tested, and reviewed by the project's community to become part of the project's mainstream code. In the specific case of Linux, the process is two-tiered: if the subsystem or project containing the new contribution is then picked up by a Linux distribution like SuSE, it will be subject to that distribution-specific assurance procedure. Typical assurance activities performed at distribution level include standards-compliance testing (for example, LSB and POSIX), stability and robustness testing. Today, Linux security certification is also carried out at the distribution level (Chapter 6).

It should be noted that the assurance process described above (see Figure 5.2) can be made three-tiered by adding a user-specific assurance activity at adoption time. In the case of Linux, this assurance procedure is coarser-grained, as adopters retain control over the inclusion of each specific sub-

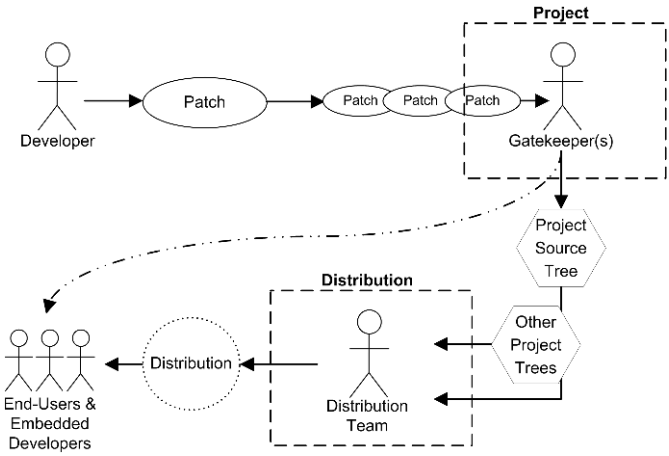

Fig. 5.2: The two- and three-tiered assurance process of OSS

system in the distribution, but of course not of individual contributions to subsystems. However, regression is always possible, that is, the adopter can always return to a previous version of the subsystem if an update does not meet the adopter's quality or dependability standards. Some adopters prefer to entirely delegate security and quality assurance to their preferred distribution; others try and influence the OSS product evolution at both design and assurance time.[5] We are now ready to summarize the set of properties that distinguish OSS development from a traditional development processes from the point of view of security assurance.

- *Large distributed community of developers.* It is one of strongest points of OSS development. No matter how a big a company can be, it will never have as many developers as some well known open source projects.
- *Fast feedback from users.* Fault reports, usability problems, security vulnerabilities reports are some examples of the feedback the open source community receives constantly from the users. This communication tends to be more frequent and intense, and above all more direct, than the one taking place for proprietary software products. This feedback allow open source developers to locate and fix reported faults very rapidly.
- *Users are an integral part of the development process.* Users have been always considered part of the OSS development cycle. Keeping the users involved helps them to express their point of views about functional re-

[5] A notable example is the telecommunication sector, where *Carrier Grade Linux* (CGL) defines a set of specifications as well as some assurance criteria which must be met in order for Linux to be considered "carrier grade" (i.e., ready for use within the telecommunications industry).

quirements, security, usability, write documentation, and other tasks that developers may neglect.

- *Talented and highly motivated developers.* The idea of having their code reviewed by a member of the core group of an open source software, or knowing that their code would be part of the next release of a widespread software product attracts many motivated developers to contribute to open source projects [16].
- *Rapid release cycles.* Thanks to a large, active and high motivated community of developers working around the clock, many open source software provide more rapid release cycle and updates compared to software developed inside companies. For example, in 1991 there was more than one release per day for the Linux kernel [16].
- *Terseness of analysis documentation.* OSS is not based on the definition of a set of formal and stable specifications and requisites to be used in the development process; rather, community-wide discussions are used as a kind of implicit specifications that drive the developers contribution to the software.

The above properties of OSS development correspond to many advantages (as well as to some disadvantages) compared to other types of software development processes. In Section 5.2, we shall discuss in detail how these advantages and disadvantages affect the adoption of OSS in critical security areas and the certification of OSS security properties. However, we can anticipate some interesting facts. First of all, while the open source specificity in terms of development process and licenses may still play a marginal role in the decisions of individual users, it has already a huge impact on adoption patterns of companies and organizations. Sometimes companies do not have the time and resources for a complete pre-adoption analysis of OSS and just "go and use it". This pattern of OSS adoption is often seen as a leverage against vendors of proprietary software products. Other times, companies rely on internal adoption guidelines, and look at the exploitation of OSS community-based development as a way to reduce development costs and accelerate the availability of new features. This opportunistic strategy does not come for free, because it requires creating internal competence groups able to influence the relevant development communities. Alternatively, brokers (often individual consultants or small companies) familiar with the OSS development process can be hired to manage partially or totally the interaction with the development community.

5.2 OSS security

In the last few years, security has not been a major driver for open source assurance [4]. Possible reasons include some reluctance by the OSS community to set up a separate security assurance process, as opposed to a general

quality assurance one. On Tuesday July 15th 2008, posting to the `gmane` discussion group, Linus Torvalds wrote:
"*...one reason I refuse to bother with the whole security circus is that I think it glorifies - and thus encourages - the wrong behavior.*
It makes "heroes" out of security people, as if the people who don't just fix normal bugs aren't as important.
In fact, all the boring normal bugs are way more important, just because there's a lot more of them. I don't think some spectacular security hole should be glorified or cared about as being any more "special" than a random spectacular crash due to bad locking".

In turn, some security experts do not completely trust the open source communities and consider open source middleware a potential "backdoor" for attackers, potentially affecting overall system security. However, proprietary security solutions have their own drawbacks such as vendor lock-in, interoperability limitations, and lack of flexibility. Recent research suggests that the open source approach can overcome these limitations [3, 41].

A long debate has been going on in the security research community, whether open source software should be considered more or less secure than closed source software. A classical analysis is the one done by Wheeler in [52]. This debate has not led to any definitive conclusion so far [13], and it is unlikely to do so in the near future. It is however interesting to note that the discussion has been mainly confined to software implementing security solutions rather than extended to general system-level software products or to software applications, although it is widely recognized that most new threats to security are emerging at the application level. According to another essay published by David Wheeler, the author of "Secure Programming for Linux and Unix" [51] on his book's website (`http://www.dwheeler.com/secure-programs`): "There has been a lot of debate by security practitioners about the impact of open source approaches on security. One of the key issues is that open source exposes the source code to examination by everyone, including both attackers and defenders, and reasonable people disagree about the ultimate impact of this situation". Wheeler's interesting essay contains a number of contending opinions by leading security experts, but no conclusion is reached.

Probably, the only non-questionable fact is that open source software gives both attackers and defenders greater control over software security properties. Let us summarize Wheeler's arguments in favor and against the notion that a software being open source has a positive impact on its security-related properties.

- *Open source is less secure.* The availability of source code may increase the chances that an attacker will detect and exploit a software fault. Also, the informality and flexibility of the open source development process have been known to backfire. For instance, not all open source projects provide documentation on the secure deployment of the software they develop. This lack of information may cause faulty installations, which in

turn may result in security holes. Some open source development communities do not bother with hardening their products against well-known security vulnerabilities and do not provide any report on fault detection, even though some simple open source tools, such as FindBugs, are available (see Chapter 4 for more details). Delay in implementing well-understood security patches is a clear indication that security best practices are a low priority issue for some open source projects. This may be understandable in some cases, because security requirements may be quite different for different applications, but it could also indicate a lower level of security awareness on the part of the OSS community.

- *Open source is more secure.* The greater visibility of software faults [13] typical of open source products may also be exploited by defenders. Well trained defenders, taking advantage of knowledge normally not available with closed source software, have been known to ensure a short response time, fixing software vulnerabilities as soon as the corresponding threats are described. Defenders can also rely on the independent work of the open source community to identify new security threats, since open source software is usually subject to a community-wide review and validation process.

Whatever the merits of these positions, this controversy is bound to remain somehow academic. Open source projects largely differ from one another, and the same arguments can be brought in favor or against the thesis depending on the specific situation considered. In their paper "Trust and Vulnerability in Open Source Software" [25], Scott A. Hissam and Daniel Plakosh rightly observe that "just because a software is open to review, it should not automatically follow that such a review has actually been performed".[6]

Hopefully, however, the above discussion has clarified the need of some form of security certification based on a rigorous and in-depth analysis. What is still missing is a security certification framework allowing, on the one side, suppliers to certify the security properties of their software and, on the other side, users to evaluate the level of suitability of different open source security solutions. We shall discuss the requirements for such a framework in the next section.

5.3 OSS certification

In principle, the standard certification processes described in this book can be employed to certify OSS products, as they are for proprietary ones. The obstacle posed by the peculiar nature of OSS development process can in fact be overcame, since some certification standards accept applications from organizations adopting nearly every type of development process. Therefore

[6] Incidentally, we remark that the same comment can be made for testing.

forges, consortia or foundations promoting open source product development do in principle qualify as applicants for obtaining the certification of OSS products. For example, the norm ISO 9126 (see Chapter 1) explicitly mentions an application to OSS projects. Any OSS project compliant to these standards can in theory acquire the same status as a conventional project. This is also true for other certifications we described or mentioned in the previous Chapters, like ISO 9000 for software suppliers, ITSEC (European), or CC (international). The evaluation body will examine a software product's specified functionality, the quality of its implementation, and the compliance with security standards.

However, some other obstacles do exist in terms of laboratory techniques. In fact, an important prerequisite to most certifications is the availability of a testing framework to support all controls required by the certification process. We shall deal with this problem in Chapter 8

5.3.1 State of the art

Comparative evaluation of OSS non-functional properties, including security-related ones is a time-honored subject. Much work has been done on open source security testing: for instance, the *Open Source Security Testing Methodology Manual* (OSSTMM) (`http://www.isecom.org/osstmm/`) is a peer-reviewed methodology for performing security tests and metrics. The OSSTMM test cases are divided into five sections which collectively test: information and data controls, personnel security awareness levels, fraud and social engineering control levels, computer and telecommunications networks, wireless devices, mobile devices, physical security access controls, security processes, and physical locations such as buildings, perimeters, and military bases. OSSTMM focuses on the technical details of exactly which items need to be tested, what to do before, during, and after a security test, and how to measure the results. New tests for international best practices, laws, regulations, and ethical concerns are regularly added and updated. The methodology refers to risk-oriented metrics such as Risk Assessment Values (RAVs) and defines and quantifies three areas (operations, controls, limitations) as its relevance to the current and real state of security.

The *Qualify and Select Open Source Software* (QSOS) is a methodology designed to qualify, select and compare free and open source software in an objective, traceable and argued way (`http://www.qsos.org/`). QSOS aims to compare solutions against formalized requirements and weighted criteria, and to select the most suitable product set available. QSOS is based on an iterative approach where the evaluation step is based on metrics defining, on the one hand, the risks from the customer perspective and, on the other hand, the extent to which these risks are addressed by OSS solutions. In general, all selection techniques require information that many open source

projects fail to make available. The *Software Quality Observatory for Open Source Software* project (`http://www.sqo-oss.eu/`) includes techniques computing account quality indicators from data that is present in a project's repository. However, such metadata need to take into account the specific domain of the application. For instance, the dependency of loop execution times on hardware features is a relevant quality indicator for time-critical control loops, but has little interest for business application developers.

Several researchers [10] have proposed complex methodologies dealing with the evaluation of open source products from different perspectives, such as code quality, development flow and community composition and participation. General-purpose open source evaluation models, such as Bernard Golden's *Open Source Maturity Model* (OSMM) [21] do not address the specific requirements of security software selection. However, these models assess open source products based on their maturity, that is, their present production-readiness, while evaluating security solutions also involves trying to predict how fast (and how well) a security component will keep abreast of new attacks and threats. Several other OSS adoption methodologies have been proposed and developed into practical guidelines defining methodology-(or enterprise-) specific benchmarks in terms of functionality, community backing as well as maturity. Most of these methodologies, however, are biased toward business-related software systems and toward static integration. Therefore, they are of limited use for complex products like telecommunication devices, which bundle or dynamically integrate hardware and software components. For example, the *Business Readiness Rating* (BRR) method [40] supports quantitative evaluation of open source software identifying seven categories: functionality, software quality, service and support, documentation, development process, community, and licensing issues. However, additional work is required to deal with OSS non-functional properties (performance, security, safety) across the different integration mechanisms, and with white-box ones (terseness/sparseness, readability), which are crucial to OSS bundling within complex products [49]. Focusing on security area, a security-oriented software evaluation framework should provide potential adopters with a way to compare open source solutions identifying the one which *(i)* best suits their current non-functional requirements and *(ii)* is likely to evolve fast enough to counter emerging threats. Our own works in [2, 5] are aimed at providing a specific technique for evaluating open source security software, including access control and authentication systems. Namely, a *Framework for OS Critical Systems Evaluation* (FOCSE) [2] has been defined and is based on a set of *metrics* aimed at supporting open source evaluation, through a formal process of adoption. FOCSE evaluates an open source project in its entirety, assessing the community composition, responsiveness to change, software quality, user support, and so forth. FOCSE criteria and metrics are also aimed at highlighting the promptness of reacting against newly discovered vulnerabilities or incidents. Applications success, in fact, depends on the above principle because a low reaction rate to new vulnerabilities or incidents

Metrics	Putty	WinSCP	ClusterSSH
Age (GA)	2911 days	1470 days	1582 days
Last Update Age (GA)	636 days	238 days	159 days
Project Core Group (GA,DC)	Yes	Yes	Yes
Number of Core Developers (DC)	4	2	2
Number of Releases (SQ,IA)	15	32	15
Bug Fixing Rate (SQ,IA)	0.67	N/A	0.85
Update Average Time (SQ,IA)	194 days	46 days	105 days
Forum and Mailing List Support (GA,DIS)	N/A	Forum Only	Yes
RSS Support (GA,DIS)	No	Yes	Yes
Number of Users (UC)	N/A	344K	927
Documentation Level (DIS)	1.39 MB	10 MB	N/A
Community Vitality (DC,UC)	N/A	3.73	5.72

Table 5.1: Comparison of open source SSH implementations at 31 December 2006

implies higher risk for users that adopt the software, potentially causing loss of information and money. Finally, to generate a single estimation, FOCSE exploits an aggregator often used in multi-criteria decision techniques, the Ordered Weighted Average (OWA) operator [50, 54], to aggregate the metrics evaluation results. This way, two or more OSS projects, each one described by its set of attributes, can be ranked by looking at their FOCSE estimations. In [2], some examples of the application of FOCSE framework to existing critical applications are provided. Here, we provide a sketch of the FOCSE-based evaluation of security applications that implement the Secure Shell (SSH) approach. SSH is a communication protocol widely adopted in the Internet environment that provides important services like secure login connections, tunneling, file transfers and secure forwarding of X11 connections [55]. The FOCSE framework has been applied for evaluating the following SSH clients: Putty [39], WinSCP [53], ClusterSSH [48]. First, the evaluation of SSH client implementations based on the security metrics and information gathered by FLOSSmole database [17] is provided and summarized in Table 5.1.

	Putty	WinSCP	ClusterSSH
f_{OWA}	0.23	0.51	0.47

Table 5.2: OWA-based comparison of SSH clients

Then, an OWA operator is applied to provide a single estimation of each evaluated solution. Finally, the FOCSE estimations are generated (see Table 5.2), showing that the solution more likely to be adopted is WinSCP. In summary, FOCSE evaluation framework gives a support to the adoption of open source solutions in mission critical environments.

As far as model-based certification is concerned, some ad hoc projects toward applying model- and test-based certification techniques to individual OSS products have been taken. For instance, the U.S. Department of Homeland Security has funded a project called "Vulnerability Discovery and Remediation, Open Source Hardening", involving Stanford University, Coverity and Symantec. The project was aimed at hunting for security-related faults in open-source software, finding and correct the corresponding security vulnerabilities, and to improve Coverity's commercial tool for source code analysis. This effort resulted in a system that does daily scans of code contributed to popular open-source projects, and produces and maintains a database of faults accessible to developers.

Looking at test-based certification, some major players developed and published anecdotal experience in certifying open source platforms, including Linux itself [44], achieving the Common Criteria (CC) EAL-4 security certification. The work in [44] describes the IBM experience in certifying Open Source and illustrates how the authors obtained the Common Criteria security certification evaluating the security functions of Linux, the first open-source product receiving such certification. We shall discuss the Linux certification process in more detail in the following Chapters.

General frameworks are needed to provide a methodology for functional and non-functional testing of OSS. Referring to the terminology we introduced in Chapter 3, no widely accepted OSS-specific methodology is available supporting white-box testing in terms of code terseness/sparseness, readability and programming discipline. As far as black-box testing is concerned, description of tests carried out on OSS at unit or component level are sometimes made available by the development communities or as independent projects (see Chapter 6). However, complex systems whose components have been developed independently may require additional support for integration and system testing. Furthermore, to obtain a genuine certification, in terms of inward security and outward protection of a complex software system which includes open source, it is not sufficient that all elements of the system are certified: the composition of security properties across the integration technique must also be taken into account. When OSS is introduced into the context of complex modular architectures, certifying the overall security of the product is a particularly critical point [13, 37].

The issues to be addressed in the context of a security assurance and certification involving an OSS software product can be classified in the following four areas.

- *Functional test and certification.* Provide test and certification of functionalities, supporting the use of OSS within critical platforms and environments for operating and business support systems, gateway, signaling and management servers, and for the future generations of voice, data and wireless components; define a comprehensive approach to certification of products dynamically integrating OSS, creating an OSS specific

path to certify typical functionalities of complex networking systems like routing, switching, memory management and the like.
- *Integration support.* Complement existing approaches providing the specific design tools needed to bring OSS into the design and implementation path of advanced European products, providing the research effort needed for successful OSS integration within complex systems and for using OSS as a certified tool for complex systems development.
- *Advanced description.* Provide novel description techniques, suitable for asserting the relevant properties of OSS also integrated in telecommunication devices and other embedded systems. Properties expressed should include specific ones such as timing dependencies, usage of memory and other resources.
- *Governance and IPR issues.* Develop variety of across-project indicators on OSS dynamic integration and static bundling. Indicators will provide company management (as opposed to the leaders of individual development projects) with quantitative and value-related percentages of OSS adoption (e.g., within product lines), so that company wide governance policies regarding OSS adoption, as well as the integration techniques, can be monitored and enforced. Also, it will guarantee IPR peace of mind by providing support for assessing the IPR nature of products embedding OSS.

5.4 Security driven OSS development

Although the lack of a formal software development process is usually not seen as a drawback by OSS communities, it may become a problem in the security area, because security assurance techniques often assume a stable design and development process. Focusing on security aspects, discussions in OSS communities rarely state formal security requirements; rather they mention informal requirements (e.g., "the software must not crash due to buffer overflow"). However, these informal specifications are difficult to certify as such. In this context, the need arises for a mechanism for defining security requirements in a simply and unambiguous way.

The CC's *Security Target* (ST) can become a fundamental input to OS communities, improving the OSS software development process by providing clear indications of the contributions expected by the developers to the project [29, 31, 34]. The ST in fact describes the security problems that could compromise the system and identifies the objectives which explain how to counter the security problems. Also, the ST identifies the security requirements that need to be satisfied to achieve the objectives.

Intuitively, the ST assumes a twofold role as a community input: *(i)* it provides guidelines (ST's objectives and requirements), the developers need to follow when contributing to the community; *(ii)* it supports CC-based

evaluation and certification of the OSS. This is due to the fact that, thanks to point *(i)*, OSS systems can be designed and developed by already considering the ST to be used in a subsequent certification. Community developers will then be asked to provide, during the software development process, all documentations and tests required for certification during the software development phase. This solution results in a scenario where security targets become high-level specification documents to be jointly developed by the communities, driving OSS security assurance and security certification processes.

In the next section we illustrate through an example how an ad-hoc security target can be used to provide security specification and requirements. These requirements can then be used to manage the development process of OSS community contributing to the system under development.

5.5 Security driven OSS development: A case study on Single Sign-On

The use case we provide is on an *open source* Single Sign-On [8] solution, which allows users to enter a single username and password to access systems and resources, to be used in the framework of an open source e-service scenario. Single Sign-On (SSO) systems are aimed at providing functionalities for managing multiple credentials of each user and presenting these credentials to network applications for authentication (see Chapter 2).

Starting from a definition of the ST for a SSO system, we identify different trust models and the related set of requirements to be satisfied during the development phase. Then, we turn to the community for the development and informally evaluation of a fully functional SSO system. The ST-based solutions ensures that the software product will be developed with certification in mind.

5.5.1 Single Sign-On: Basic Concepts

Applications running on the Internet are increasingly designed by composing individual *e-services* such as e-Government services, remote banking, and airline reservation systems [15], providing various kind of functionalities, such as, paying fines, renting a car, releasing authorizations, and so on. This situation is causing a proliferation of user accounts: users typically have to log-on to multiple systems, each of which may require different usernames and authentication information. All these accounts may be managed independently by local administrators within each individual system [22]. In other words, users have multiple credentials and a solution is needed to give them the illusion

of having a single identity and a single set of credentials. In the multi-service scenario, each system acts as an independent domain. The user first interacts with a *primary domain* to establish a session with that domain. This transaction requires the user to provide a set of credentials applicable to the domain. The primary domain session is usually represented by an operating system shell executed on the user's workstation. From this primary domain session shell, the user can require services from other *secondary domains*. For each of such requests the user has to provide a set of credentials applicable to the secondary domain she is connecting to.

From the account management point of view, this approach requires independent management of accounts in each domain and use of different authentication mechanisms. In the course of time, several usability and security concerns have been raised leading to a rethinking of the log-on process aimed at co-ordinating and, where possible, integrating user log-on mechanisms of the different domains.

A service that provides such a co-ordination and integration of multiple log-on systems is called *Single Sign-On* [14] platform. In the SSO approach the primary domain is responsible for collecting and managing all user credentials and information used during the authentication process, both to the primary domain and to each of the secondary domains that the user may potentially require to interact with. This information is then used by services within the primary domain to support the transparent authentication to each of the secondary domains with which the user requests to interact. The advantages of the SSO approach include [12, 22]:

- *reduction* of *i)* the *time spent* by the users during log-on operations to individual domains, *ii) failed log-on* transactions, *iii)* the *time used to log-on* to secondary domains, *iv) costs and time* used for users profiles administration;
- *improvement to users security* since the number of username/password each user has to manage is reduced;[7]
- *secure and simplified administration* because with a centralized administration point, system administrators reduce the time spent to add and remove users or modify their rights;
- *improved system security* through the enhanced ability of system administrators to maintain the integrity of user account configuration including the ability to change an individual user's access to all system resources in a co-ordinated and consistent manner;
- *improvement to services usability* because the user has to interact with the same login interface.

[7] It is important to note that, while improving security since the user has less accounts to manage, SSO solutions imply also a greater exposure from attacks; an attacker getting hold of a single credential can in principle compromise the whole system.

Also, SSO provides a uniform interface to user accounts management, enabling a coordinated and synchronized management of authentication in all domains.

5.5.2 A ST-based definition of trust models and requirements for SSO solutions

Open source requirements for a SSO are unlikely to be much more detailed than the informal description made in the previous section. Such informal requirements can correspond to different SSO solutions, which could slightly differ in their purposes, depending on the business and trust scenario where they are deployed. In a traditional development process, formal specifications would be used in order to disambiguate this description and lead the development to the desired outcome. As an OSS-targeted alternative, let us show how this can be obtained by the definition of a community-wide Security Target. Namely, we will generate the *Single Sign-On Security Target* (SSO ST) starting by the informal requirements followed in the development of the Central Authentication Service (CAS) [6, 11] and by the Computer Associates eTrust Single Sign-On V7.0 [46] Security Target.

5.5.2.1 Central Authentication Service (CAS)

Central Authentication Service (CAS) [6, 11] is an open source framework developed by Yale University. It implements a SSO mechanism to provide a *Centralized Authentication* to a single server and *HTTP redirections*. The CAS authentication model is loosely based on classic Kerberos-style authentication [35]. When an unauthenticated user sends a service request, this request is redirected from the application to the authentication server (CAS Server), and then back to the application after the user has been authenticated. The CAS Server is therefore the only entity that manages passwords to authenticate users and transmits and certifies their identities. The information is forwarded to the application by the authentication server during redirections by using session cookies.

CAS is composed of pure-Java servlets running over any servlet engine and provides a very basic web-based authentication service. In particular, its major security features are:

1. passwords travel from browsers to the authentication server via an encrypted channel;
2. re-authentications are transparent to users if they accept a single cookie, called *Ticket Granting Cookie* (TGC). This cookie is opaque (i.e., TGC

contains no personal information), protected (it uses SSL) and private (it is only presented to the CAS server);
3. applications know the user's identity through an opaque one-time Service Ticket (ST) created and authenticated by the CAS Server, which contains the result of a hash function applied to a randomly generated value.

Also, CAS credentials are *proxiable*. At start-up, distributed applications get a *Proxy-Granting Ticket* (PGT) from CAS When the application needs access to a resource, it uses the PGT to get a proxy ticket (PT). Then, the application sends the PT to a back-end application. The back-end application confirms the PT with CAS, and also gains information about who proxied the authentication. This mechanism allows "proxy" authentication for Web portals, letting users to authenticate securely to untrusted sites (e.g., student-run sites and third-party vendors) without supplying a password. CAS works seamlessly with existing Kerberos authentication infrastructures and can be used by nearly any Web-application development environment (JSP, Servlets, ASP, Perl, mod_perl, PHP, Python, PL/SQL, and so forth) or as a server-wide Apache module.

5.5.2.2 Single Sign-On Security Target (SSO ST)

The SSO ST can be used to define security requirements for SSO solutions and to drive a certification-oriented SSO implementation (i.e., CAS++ [9]).[8]

As shown in Section 3.10, a security target is composed of 7 sections, each of which needs to be defined for a specific Target Of Evaluation (TOE). Since we have already explained the content of each ST section, here we shall focus on the specificities of the TOE and on the different parts of the SSO ST. The first section of the SSO ST we analyze is the *TOE overview*, which provides the reader with some initial insight about the context we are dealing with, and the type of product we are considering. The SSO ST defines the TOE overview in Figure 5.3.

Example of TOE overview.

- *TOE overview*. This Security Target (ST) defines the Information Technology (IT) security requirements for Single Sign-On secure e-Services. Single Sign-On for secure e-services implements a SSO mechanism to provide a Centralized Authentication to a single server and HTTP redirections. SSO system integrates an authentication mechanism with a Public Key Infrastructure (PKI).

Fig. 5.3: TOE overview

[8] The SSO ST presented here is defined starting from the Computer Associates eTrust Single Sign-On V7.0 ST [46] and represents a proof of concept only. It has not been subject to any formal certification process, neither created by any evaluation body.

To gain a better understanding of the TOE, we split the TOE description into two subsections where we describe the TOE type and architecture. Figure 5.4 illustrates the TOE description section and Figure 5.5 depicts CAS architecture which is taken from [27] and used as a template for describing a SSO architecture.

Example of TOE overview.

- *Product type.* While there is an increasing need for authenticating clients to applications before granting them access to services and resources, individual e-services are rarely designed in such a way to handle the authentication process. The reason e-services do not include functionality for checking the client's credentials is that they assume a unified directory system to be present, making suitable authentication interfaces available to client components of network applications. On some corporate networks, all users have a single identity across all services and all applications are directory enabled.
  As a result, users only log in once to the network, and all applications across the network are able to check their unified identities and credentials when granting access to their services. However, on most Intranet and on the open network users have multiple identities, and a solution is needed to give them the illusion of having a single identity and a single set of credentials. *Single Sign-On* (SSO) systems are aimed at providing this functionality, managing the multiple identities of each user and presenting their credentials to network applications for authentication.
- *TOE architecture.* The *Central Authentication Server* (CAS) is designed as a standalone web application. It is currently implemented as several Java servlets and runs through the HTTPS server on `secure.its.yale.edu`. It is accessed via the three URLs described in Figure 5.5 below: the login URL, the validation URL, and the optional logout URL.

Fig. 5.4: TOE description

In the *security problem definition* section of the ST document, we describe the expected operational environment of the TOE, defining the threats and the security assumptions.[9] Table 5.3 shows the threats that may be addressed either by the TOE or its environments, and Table 5.4 shows our assumptions concerning the TOE environment. For the sake of conciseness, we assume that no organizational security policies apply to our case, leaving it to the interested reader to come up with one as an exercise.

Based on the security problems defined in Tables 5.3 and 5.4, the security objectives section of the ST document contains a set of concise statements as a response to those problems. The security objectives we defined for a SSO application are listed in Table 5.5. We have also defined the security objectives for the TOE environment listed in Table 5.6.

[9] Since we consider the same security functionalities as in [46], many of the threats and assumptions defined here are taken from [46]

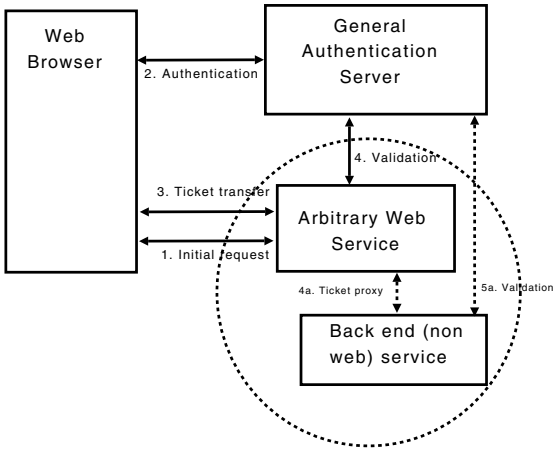

Fig. 5.5: CAS architecture

Threat ID	Threat Description
T.WeakCredentials	Users may select bad passwords, which make the system vulnerable for attackers to guess their passwords and gain access to the TOE.
T.Access	Attackers may attempt to copy or reuse authentication information to gain unauthorized access to resources protected by the TOE.
T.Impersonate	Attackers may impersonate other users to gain unauthorized access to resources protected by the TOE.
T.Mismanage	Administrators may make errors in the management and configuration of security functions of the TOE. Those errors may permit attackers to gain unauthorized access to resources protected by the TOE.
T.BlockSystem	Attacker may attempt to login as an authorized user and gain unauthorized access to resources protected by the TOE. The attacker may login multiple times, thus locking out the authorized user.
T.Reuse	An attacker may attempt to reuse authentication data, allowing the attacker to gain unauthorized access to resources protected by the TOE.
T.Undetected	Attempts by an attacker to violate the security policy and tamper with TSF data may go undetected.
T.NotLogout	A logged-in user may leave a workstation without logging out, which could enable an unauthorized user to gain access to the resources protected by the TOE.
T.CredentialsTransfer	An Attacker may listen to the communication traffic to find any authentication information.

Table 5.3: TOE threats

Assumption ID	Assumption Description
A.Admin	Administrator is trusted to correctly configure the TOE.
A.Trust	It is assumed that there will be no untrusted users and no untrusted software on the Policy Server host.
A.TrustedNetwork	It is assumed that the TOE components communicate over a physically protected Local Area Network.
A.Users	It is assumed that users will protect their authentication data.

Table 5.4: TOE security assmptions

Objective ID	Objective Description
O.Audit	The TOE must provide ways to track unexpected behaviors
O.Authentication	The TOE must identify and authenticate all users before providing them with application authentication information.
O.DenySession	The TOE must be able to deny session establishment based the maximum number of sessions a user can have open simultaneously and an idle time-out.
O.Reauthenticate	The TOE must be able to require the user to be re-authenticated under specified conditions.
O.StrongAuthentication	The TOE must integrate strong authentication mechanism based on two-factor authentication such as a smartcard and biometric properties of the user.
O.Authorization	The TOE shall determine the level of information/services the requester can see/use.
O.AUthoManagement	The TOE shall provide support for authorization management.
O.Provisioning	Before sending any request the TOE shall ensure that it satisfies all the required pre-conditions defined by the administrators.
O.Centralization	User profiles should be maintained within the TOE.
O.SafeTransport	The TOE architecture implies the exchange of user information between the TOE server and services to fulfill authentication and authorization processes. Secure data transfer shall be assured.
O.Control	The TOE shall provide a unique access control point for users who want to request a service.

Table 5.5: TOE security objectives

Objective ID	Objective Description
OE.Admin	Those responsible for the administration of the TOE are competent and trustworthy individuals, capable of managing the TOE and the security of the information it contains.
OE.Install	Those responsible for the TOE must establish and implement procedures to ensure that the hardware, software and firmware components that comprise the system are distributed, installed and configured in a secure manner.
OE.Operation	There must be procedures in place in order to ensure that the TOE will be managed and operated in a secure manner.
OE.Auth	The users must ensure that their authentication data is held securely and not disclosed to unauthorized persons.

Table 5.6: Security objectives for TOE environment

We can finally define the security requirements that need to be satisfied by the TOE in order to reach the defined objectives. This is a crucial step; in a OSS community, it is very important to put in charge of requirements definition someone having a deep understanding of the security objectives. The set of requirements is divided into *security functional requirements* (SFRs) and *security assurance requirements* (SARs), which have been taken respectively from CC Part2 [18], and Part3 [19].

The SFRs we defined for the TOE are listed in Table 5.7. For the sake of simplicity, we shall show only one example of SFR description (see Figure 5.6).

Component	Name
FAU_GEN.1	Audit data generation
FAU_GEN.2	User identity association
FIA_SOS.1	Verification of secrets
FIA_UAU.1	Timing of authentication
FIA_UAU.2	User authentication before any action [Primary Authentication]
FIA_UAU.5	Multiple authentication mechanisms
FIA_UAU.6	Re-authenticating [Primary Authentication]
FIA_UID.2	User identification before any action [Primary Authentication]
FTA_SSL.3	TSF-initiated termination
FTA_TSE.1	TOE session establishment
FTP_ITC.1	Inter-TSF trusted channel

Table 5.7: Security Functional Requirements for the TOE

FAU_GEN.1 Audit data generation
Hierarchical to: No other components.
FAU_GEN.1.1 The TSF shall be able to generate an audit record of the following auditable events:

a) Start-up and shutdown of the audit functions;
b) All auditable events for the [not specified] level of audit; and
c) **[the following auditable events:**

- **User login/logout**
- **Failed attempts to login**
- **User session timeout].**

Fig. 5.6: Example of SFR

Looking at the original requirement defined by CC Part2 [18] for the example in Figure 5.6, it is clear that we need to change FAU_GEN.1.1 before being able to adopt it in our ST. The first operation is a selection operation. CC Part2 [18] defines the point *(b)* of FAU_GEN.1.1 as follows: "*All auditable events for the [selection, choose one of: minimum, basic, detailed, not specified] level of audit*". To adapt it to our case, we had to select one of the given alternatives. The second operation is an assignment operation in which more parameters can be specified. The original version of point *(c)* in CC part2 [18] is defined as *[assignment: other specifically defined auditable events].* To use it in our ST, we need to specify the auditable events of our interest.

The security assurance requirements in SSO ST are the assurance components of Evaluation Assurance Level 2 (EAL2) taken from CC Part3 [19]. None of the assurance components has been refined. The EAL2 assurance requirements are listed in Table 5.8.

We are now ready to consider another important section of the SSO ST, namely the *TOE specification summary*, where more details about the security functions of the TOE are given. This section also provides also a mapping between the security functions of the TOE and the SFRs. The example in Figure 5.7 describes the auditing mechanism of a SSO solution taken from [27].

5.5.2.3 Trust Models

Trust models are the basis for designing interoperable systems. A *trust model* describes a software system by defining its underlying environment as well as its components, and the rules governing their interactions. Here, we focus on the definition of trust models for SSO environments, based on the functionalities that these environments support. We do not only consider the security functionalities and requirements in SSO ST (described in the previous sec-

Assurance Class	Assurance components
ADV: Development	ADV_ARC.1 Security architecture description
	ADV_FSP.2 Security-enforcing functional specification
	ADV_TDS.1 Basic design
AGD: Guidance documents	AGD_OPE.1 Operational user guidance
	AGD_PRE.1 Preparative procedures
ALC: Life-cycle support	ALC_CMC.2 Use of a CM system
	ALC_CMS.2 Parts of the TOE CM coverage
	ALC_DEL.1 Delivery procedures
ASE: Security Target evaluation	ASE_CCL.1 Conformance claims
	ASE_ECD.1 Extended components definition
	ASE_INT.1 ST introduction
	ASE_OBJ.2 Security objectives
	ASE_REQ.2 Derived security requirements
	ASE_SPD.1 Security problem definition
	ASE_TSS.1 TOE summary specification
ATE: Tests	ATE_COV.1 Evidence of coverage
	ATE_FUN.1 Functional testing
	ATE_IND.2 Independent testing - sample
AVA: Vulnerability assessment	AVA_VAN.2 Vulnerability analysis

Table 5.8: EAL2 Security Assurance Requirements

tion), but adopt a wider perspective, including also functional aspects of the SSO solutions. We identify three models.

Authentication and Authorization Model (AAM). This model is one of the traditional security/trust models describing all frameworks that provide authentication and authorization features [42]. It represents the basic mechanism in which a user requires an access to a service that checks the users' credentials to decide whether access should be granted or denied. The AAM model identifies two major entities: *users,* which request accesses to resources, and *services,* potentially composed by a set of intra-domain services, which share these resources. This model is based on the classic client-server architecture and provides a generic protocol for authentication and authorization processes.

Federated Model (FM). This model represents one of the emergent security/trust models in which several homogeneous entities interact to provide security services, such as identity privacy and authentication. The FM model identifies two major entities: *users,* which request accesses to resources, and *services,* which share these resources. The major difference with the previous model resides in the service definition and composition: in federated systems the services are distributed on different domains and they are built on the same level allowing mutual trust and providing functionalities as cross-authentication [32].

Full Identity Management Model (FIMM). This model represents one of the most challenging security and privacy/trust models. Besides dealing

AU.1: Auditing generation
CAS uses Log4J to write event logs to either flat files or to an Oracle table (source= `http://ja-sig.org/wiki/pages/viewpagesrc.action?pageId=969`).
The logged events include:

1. sees login screen
2. successful authentication
3. requested warnings
4. unsuccessful authentication
5. authentication warning screen presented
6. inactivity timeout
7. wall clock timeout (TGT)
8. bad attempt lockout
9. logout

AU.2: Auditing information

Each log entry includes:

- date / time
- event type (e.g., TICKET_GRANT, TICKET_VALIDATE)
- username (if applicable)
- client IP address (if applicable)
- result (SUCCESS/FAILURE)
- service_url (if applicable)
- service ticket (if applicable)

These logs are used mainly for usage reports and for security reviews and incident response. The requirements of the security group are:

- ability to identify who was logged on based on IP address
- ability to identify who was logged on based on date and time
- online logs retained for at least two weeks
- archived logs retained for at least one quarter

This function contributes to satisfy the security requirements FAU_GEN.1 and FAU_GEN.2

Fig. 5.7: Example TOE specification summary describing the Logs mechanism of CAS

with all the security aspects covered by the previous two models, it provides mechanisms for identity and account management and privacy protection [1, 38]. The FIMM model identifies three major entities: *users,* which request accesses to resources, *services,* which share these resources, and *identity manager,* which gives functionalities to manage users identities.

Requirement	AAM Model	FM Model	FIMM Model
Authentication	X	X	X
Strong Authentication	X	X	X
Authorization	X		X
Provisioning	X		X
Federation		X	X
C.I.M. (Centralized Identity) Management	X		X
Client Status Info	X	X	X
Single Point of Control	X		
Standard Compliance	X	X	X
Cross-Language availability	X	X	X
Password Proliferation Prevention	X	X	X
Scalability	X	X	X

Table 5.9: Requirements categorization basing on the specific trust model.

5.5.3 Requirements

In order to compare our ST with a traditional analysis document, we need a requirement list for a Single Sign-On solution. The requirements that a SSO should satisfy are more or less well known within the security community, and several SSO projects published partial lists.[10] However, as is typical of the OSS development source, the requirements elicitation phase has been informal and no complete list of requirements has been published. A comparative analysis of the available lists brought us to formulating the following requirements (including the security ones). Focusing on security requirements, this informal list of requirements can be substituted by a ST-based requirements definition, which makes the development process stable and unambiguous. For each requirement we also report the trust model (AMM, FM, FIMM) to which it refers.[11]

Authentication (AAM,FM,FIMM). A major requirement of a SSO system is to provide an authentication mechanism. Usually authentication is performed by means of a classic username/password log-in, whereby a user can be unambiguously identified. Authentication mechanisms should usually be coupled with a logging and auditing process to prevent and, eventually, discover malicious attacks and unexpected behaviors. From a purely

[10] For an early attempt at a SSO requirements list, see `middleware.internet2.edu/webiso/docs/draft-internet2-webiso-requirements-07.html`.

[11] Note that different trust models fulfill a different set of requirements (see Table 5.9). SSO solution, therefore, should be evaluated by taking into consideration only the requirements supported by the corresponding trust model.

software engineering point of view, authentication is the only "necessary and sufficient" functional requisite for a SSO architecture.

Strong Authentication (AAM,FM,FIMM). For highly secure environments, the traditional username/password authentication mechanism is not enough. Malicious users can steal a password and impersonate the user. New approaches are therefore required to better protect services against unauthorized accesses. A good solution to this problem integrates username/password check with a strong authentication mechanism based on two-factor authentication such as a smartcard and biometric properties of the user (fingerprints, retina scans, and so on).

Authorization (AAM,FIMM). After the authentication process, the system can determine the level of information/services the requester can see/use. While applications based on domain specific authorizations can be defined and managed locally at each system, the SSO system can provide support for managing authorizations (e.g., role or profile acquisitions) that apply to multiple domains.

Provisioning (AAM,FIMM). Provisions are those conditions that need to be satisfied or actions that must be performed before a decision is taken [7]. A provision is similar to a pre-condition (see Chapter 4) it is responsibility of the user to ensure that a request is sent in an environment satisfying all the pre-conditions. The non-satisfaction of a provision implies a request to the user to perform some actions.

Federation (FM,FIMM). The concept of *federation* is strictly related to the concept of *trust*. A user should be able to select the services that she wants to federate and de-federate to protect her privacy and to select the services to which she will disclose her own authorization assertions.

C.I.M. (Centralized Identity Management) (AAM,FIMM). The centralization of authentication and authorization mechanisms and, more generally, the centralization of identity management implies a simplification of the user profile management task. User profiles should be maintained within the SSO server thus removing such a burden from local administrators. This allows a reduction of costs and effort of user-profile maintenance and improves the administrators' control on user profiles and authorization policies.

Client Status Info (AAM,FM,FIMM). The SSO system architecture implies the exchange of user information between SSO server and services to fulfill authentication and authorization processes. In particular, when the two entities communicate, they have to be synchronized on what concern the user identity; privacy and security issues need to be addressed. Different solutions of this problem could be adopted involving either the transport (e.g., communication can be encrypted) or the application layer.

Single Point of Control (AAM). The main objectives of a SSO implementation are to provide a unique access control point for users who want to request a service, and, for applications, to delegate some features from business components to an authentication server. This point of control

should be unique in order to clearly separate the authentication point from business implementations, avoiding the replication and the ad-hoc implementation of authentication mechanisms for each domain. Note that every service provider will eventually develop its own authentication mechanism.

Standard Compliance (AAM,FM,FIMM). It is important for a wide range of applications to support well-known and reliable communication protocols. In a SSO scenario, there are protocols for exchanging messages between authentication servers and service providers, and between technologies, within the same system, that can be different. Hence, every entity can use standard technologies (e.g., X.509, SAML for expressing and exchanging authentication information and SOAP for data transmission) to interoperate with different environments.

Cross-Language availability (AAM,FM,FIMM). The widespread adoption of the global Internet as an infrastructure for accessing services has consequently influenced the definition of different languages and technologies used to develop these applications. In this scenario, a requisite of paramount importance is integrating authentication to service implementations written in different languages, without substantial changes to service code. The first step in this direction is the adoption of standard communication protocols based on XML.

Password Proliferation Prevention (AAM,FM,FIMM). A well-known motivation for the adoption of SSO systems is the prevention of password proliferation so to improve security and simplify user log-on actions and system profile management.

Scalability (AAM,FM,FIMM). An important requirement for SSO systems is to support and correctly manage a continuous growth of users and subdomains that rely on them, without substantial changes to system architecture.

5.5.4 A case study: CAS++

Our SSO ST is meant to drive the development of open source SSO systems. As a case study, let us it as a guide to extend the *Central Authentication Service* (CAS) [6, 11] to an enhanced version we will call CAS++.[12]. Our extension integrates the CAS system with the authentication mechanism implemented by a *Public Key Infrastructure* (PKI) [33]. CAS++ implements a fully multi-domain stand-alone server that provides a simple, efficient, and reliable SSO mechanism through HTTP redirections, focused on user privacy (opaque cookies) and security protection. CAS++ permits a centralized man-

[12] Of course CAS++ is not the only implementation available on the Net. In particular, *SourceID* [47], an Open Source implementation of the SSO Liberty Alliance, *Java Open Single Sign-On* (JOSSO) [28], and *Shibboleth* [45] are other available SSO solutions.

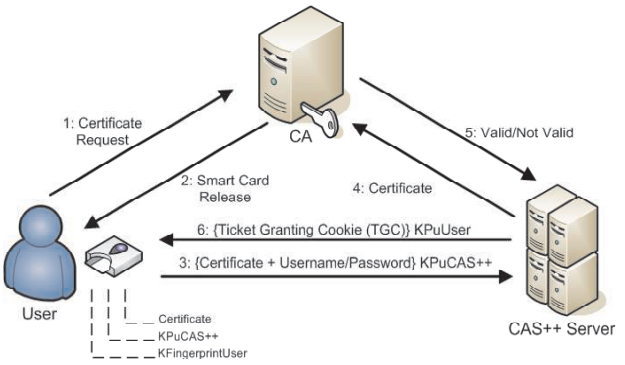

Fig. 5.8: CAS++ certificate-based authentication flow

agement of user profiles granting access to all services in the system with a unique pair username/password. The profiles repository is stored inside the SSO server application and is the only point where users credentials/profiles are accessed, thus reducing information scattering. In our implementation, services do not need an authentication layer because this feature is managed by CAS++ itself.

CAS++ relies on standard protocols such as SSL, for secure communications between the parties, and on X.509 digital certificates for credentials exchange. Besides being a "pure-Java" module like its predecessor, CAS++ is a fully J2EE compliant application integrable with services coded with every web-based implementation language. It enriches the traditional CAS authentication process through the integration of biometric identification (by fingerprints readers) and smart card technologies in addition to traditional username/password mechanism, enabling two authentication levels.

CAS++ strong authentication process flow is composed of the following steps (see Figure 5.8):[13]

1. the user requests an identity certificate to the CA (Certification Authority);
2. the user receives from the CA a smart card that contains a X.509 identity certificate, signed with the private key of the CA, that certifies the user identity. The corresponding user private key is encrypted with a symmetric algorithm (e.g., 3DES) and the key contained inside the smart card can be decrypted only with a key represented by user fingerprint (KFingerprintUser) [23];

[13] Note that, the first two actions are performed only once when the user requests the smart card along with an identity certificate.

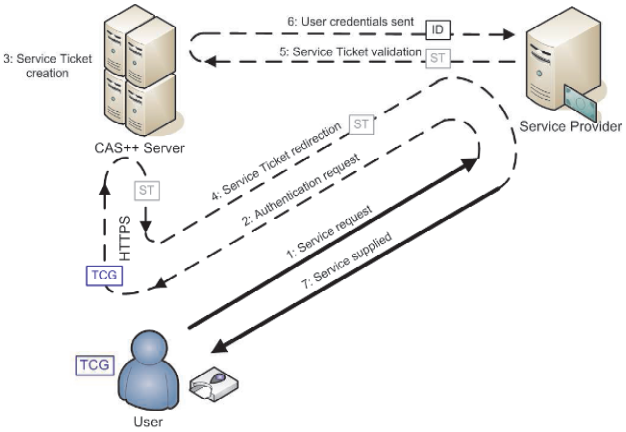

Fig. 5.9: CAS++ information flow for service request evaluation

3. to access a service the public key certificate, along with the pair username/password, is encrypted with the CAS++ public key (KPuCAS++) and sent to CAS++;
4. CAS++ decrypts the certificate with its private key, verifies the signature on the certificate with the CA public key, and verifies the validity of this certificate by interacting with the CA;
5. CAS++ retrieves from the CA information about the validity of the user certificate encrypted with KPuCAS++;
6. if the certificate is valid, CAS++ extracts the information related to the user, creates the ticket (TGC, Ticket Granting Cookie) and returns it to the user encrypted with the public key of the user (KPuUser). At this point, to decrypts the TGC, the user must retrieve the private key stored inside the smart card by mean of her fingerprint. As soon as the card is unlocked, the private key is extracted and the TGC decrypted. This ticket will be used for every further access, in the same session, to any application managed by the CAS++ Single Sign-On server.

At this point, for every further access in the session, the user can be authenticated by the service providing only the received TGC without any additional authentication action.[14]

The service access flow, that takes place over secure channels and is similar to the one in CAS, is composed of the following steps (see Figure 5.9):

1. the user, via a web browser, requests access to the service provider;
2. the service provider requests authentication information through a HTTP redirection to the CAS++ Server;

[14] Note that the TGC lifetime should be relatively short to avoid conflicts with the CA's certificate revocation process, which could cause unauthorized accesses.

3. the CAS++ Server retrieves the user TGC and the service requested URL. If the user has been previously authenticated by CAS++ and has the privilege to access the service a Service Ticket is created;
4. the CAS++ Server redirects the user browser to the requested service along with the ST;
5. service receives the ST and check its validity sending it to the CAS++ Server;
6. if the ST is valid the CAS++ Server sends to the Service an XML file with User's credentials;
7. the user gains access to the desired service.

5.5.4.1 Evaluating CAS++ Against the ST Document

CAS++ is based on the Authentication and Authorization Model. Also, CAS++ fulfills most of our ST requirements; specifically, it provides a central point of control to manage authentication, authorization, and user profiles.[15] Furthermore, CAS++ enriches the traditional CAS authentication process with the integration of biometric identification (via fingerprints readers) and smart card technologies and it is planned to include provisioning features in future releases. Note that, the lower level of CAS++ system is language independent and relies on traditional established standards, such as HTTP, SSL and X.509, without adopting emerging ones, such as SOAP and SAML. Focusing on client status info, all communications between user browser, services providers and authentication server in CAS++ scenario are managed by the exchange of opaque cookies and by the use of encrypted channels. Finally, since CAS++ development has been driven by SSO ST, the process of certifying CAS++ based on the CC standards, becomes straightforward.

5.6 Conclusions

Software Assurance (SwA) relates to the level of confidence that software functions as intended and is free of faults. In open source development, many stakeholders have a vested interest in the finalization of a standard assurance process for open source encompassing the areas of functionality, reliability, security, and interoperability. Most major OSS projects have some kind of assurance process in place which includes specific code reviews, and in some cases code analysis. Indeed, anecdotal evidence shows that code review is

[15] The centralization of users profiles affects system scalability. A solution that provides a balance between centralization and scalability needs is under study.

provenly faster and more effective in large communities.[16] As far as security certification is concerned, the process must be as public as possible, involving academia, the private sector, nonprofit organizations, and government agencies. Since no formal requirements elicitation is normally done in the OSS development process, the CC ST document can be used to collect and focus the stakeholders' view on the software product's desired security features.

References

1. C.A. Ardagna, E. Damiani, S. De Capitani di Vimercati, and P. Samarati. Towards privacy-enhanced authorization policies and languages. In *Proc. of the 19th IFIP WG11.3 Working Conference on Data and Application Security*.
2. C.A. Ardagna, E. Damiani, and F. Frati. Focse: An owa-based evaluation framework for os adoption in critical environments. In *Proc. of the 3rd IFIP Working Group 2.13 Foundation on Open Source Software*, Limerick, Ireland, June 2007.
3. C.A. Ardagna, E. Damiani, F. Frati, and M. Madravio. *Open source solution to secure e-government services*. Encyclopedia of Digital Government, Idea Group Inc., 2006.
4. C.A. Ardagna, E. Damiani, F. Frati, and M. Montel. Using open source middleware for securing e-gov applications. In *Proc. of The First International Conference on Open Source Systems*, Genova, Italy, July 2005.
5. C.A. Ardagna, E. Damiani, F. Frati, and S. Reale. Adopting open source for mission-critical applications: A case study on single sign-on. In *Proc. of the 2nd IFIP Working Group 2.13 Foundation on Open Source Software*, Como, Italy, June 2006.
6. P. Aubry, V. Mathieu, and J. Marchal. Esup-portal: open source single sign-on with cas (central authentication service). In *Proc. of the EUNIS04 - IT Innovation in a Changing World*.
7. C. Bettini, S. Jajodia, X. Sean Wang, and D. Wijesekera. Provisions and obligations in policy management and security applications. In *Proc. of the 28th Conference on Very Large Data Bases (VLDB 2002)*, Honk Kong, China, August 2002.
8. D.A. Buell and R. Sandhu. Identity management. *IEEE Internet Computing*, 7(6).
9. S. De Capitani di Vimercati F. Frati P. Samarati C.A. Ardagna, E. Damiani. Cas++: an open source single sign-on solution for secure e-services. In *Proc. of the 21st IFIP TC-11 International Information Security Conference*, Karlstad, Sweden, May 2006.
10. A. Capiluppi, P. Lago, and M. Morisio. Characterizing the oss process: a horizontal study. In *Proc. of the 7th European Conference on Software Maintenance and Reengineering*, Benevento, Italy, March 2003.
11. *Central Authentication Service*.
12. J. De Clercq. Single sign-on architectures. In *Proc. of the International Conference on Infrastructure Security (InfraSec 2002)*.
13. C. Cowan. Software security for open-source systems. *IEEE Security & Privacy*, 1(1):38–45, January-February 2003.

[16] The `Interbase` database software contained a backdoor access (username `politically` and password `correct`) which was found soon after it was released as open source, thanks to large-scale code review by the interested community.

14. B. Galbraith et al. *Professional Web Services Security.* Wrox Press, 2002.
15. S. Feldman. The changing face of e-commerce. *IEEE Internet Computing*, 4(3).
16. J. Feller and B. Fitzgerald. *Understand Open Source Software Development.* Addison-Wesley, 2002.
17. FLOSSmole. Collaborative collection and analysis of free/libre/open source project data. `ossmole.sourceforge.net/`.
18. The International Organization for Standardization and the International Electrotechnical Commission. *Common Criteria for Information Technology Security Evaluation, Part 2: Security functional components*, 2007. `http://www.commoncriteriaportal.org/files/ccfiles/CCPART2V3.1R2.pdf`.
19. The International Organization for Standardization and the International Electrotechnical Commission. *Common Criteria for Information Technology Security Evaluation, Part 3: Security assurance components*, 2007. `http://www.commoncriteriaportal.org/files/ccfiles/CCPART3V3.1R2.pdf`.
20. Free Software Foundation. `www.fsf.org/`.
21. B. Golden. *Succeeding with Open Source.* Addison-Wesley, 2004.
22. The Open Group. *Single Sign-On.*
23. F. Hao, R. Anderson, and J. Daugman. Combining cryptography with biometrics effectively. In *Technical report, Cambridge University - Computer Laboratory Technical Report UCAM-CL-TR-640.*
24. A. Hars and O. Shaosong. Working for free? motivations of participating in open source projects. In *Proc. of the 34th Annual Hawaii International Conference on System Sciences*, Maui, Hawaii, USA, January 2001.
25. S.A. Hissam, D. Plakosh, and C. Weinstock. Trust and vulnerability in open source software. In *IEE Proceedings - Software*, volume 149, pages 47–51, February 2002.
26. Open Source Initiative. *The Open Source Definition*, July 2006. `opensource.org/docs/osd/`.
27. JA-SIG. Ja-sig central authentication service. `www.ja-sig.org/products/cas/`.
28. *Java Open Single Sign-On (JOSSO).*
29. F. Keblawi and D. Sullivan. Applying the common criteria in systems engineering. *IEEE Security and Privacy*, 4(2):50–55, March 2007.
30. A.M. St. Laurent. *Understanding Open Source and Free Software Licensing.* O'Reilly Media, Inc., 2004.
31. J. Lee, S. Lee, and B. Choi. A cc-based security engineering process evaluation model. In *Proc. of the 27th Annual international Conference on Computer Software and Applications (COMPSAC 2003)*, Dallas, Texas, USA, November 2003.
32. *Liberty Alliance Project.*
33. U.M. Maurer. Modelling a public-key infrastructure. In *Proc. of the 4th European Symposium on Research in Computer Security (ESORICS 1996)*, Rome, Italy, September 1996.
34. D. Mellado, E. Fernandez-Medina, and M. Piattini. A common criteria based security requirements engineering process for the development of secure information systems. *Computer Standards & Interfaces*, 29(2):244–253, February 2007.
35. B.C. Neuman and T. Ts'o. Kerberos: an authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38, September 1995.
36. Open Source Initiative (OSI). `opensource.org/`.
37. C. Payne. On the security of open source software. *Info Systems Journal*, 12:61–78, 2002.
38. *PRIME (Privacy and Identity Management for Europe).*
39. PuTTY. A free telnet/ssh client. `www.chiark.greenend.org.uk/$\sim$sgtatham/putty/`.

40. Business Readiness Rating. *Business Readiness Rating for Open Source*, 2005. www.openbrr.org/wiki/images/d/da/BRR_whitepaper_2005RFC1.pdf.
41. E.S. Raymond. *The cathedral and the bazaar*. Available at: www.openresources.com/documents/cathedral-bazaar/, August 1998.
42. P. Samarati and S. De Capitani di Vimercati. *Foundations of Security Analysis and Design*, chapter Access Control: Policies, Models, and Mechanisms, pages 137–196. Springer Berlin / Heidelberg, 2001.
43. W. Scacchi, J. Feller, B. Fitzgerald, S.A. Hissam, and K. Lakhani. Understanding free/open source software development processes. *Software Process: Improvement and Practice*, 11(2):95–105, 2006.
44. K.S. Shankar and H. Kurth. Certifying open source: The linux experience. *IEEE Security & Privacy*, 2(6):28–33, November-December 2004.
45. *Shibboleth Project*.
46. Sygnacom solutions. *Computer Associates eTrust Single Sign-On V7.0 Security Target V2.0*, October 2005. www.commoncriteriaportal.org/files/epfiles/ST_VID3007-ST.pdf.
47. *SourceID Open Source Federated Identity Management*.
48. Cluster SSH. Cluster admin via ssh. sourceforge.net/projects/clusterssh.
49. I. Stamelos, L. Angelis, A. Oikonomou, and G.L. Bleris. Code quality analysis in open source software development. *Info Systems Journal*, 12:43–60, 2002.
50. V. Torra. The weighted owa operator. *International Journal of Intelligent Systems*, 12(2).
51. D.A. Wheeler. *Secure Programming for Linux and Unix HOWTO*. Available : http://www.dwheeler.com/secure-programs/, 2003.
52. D.A. Wheeler. *Free-Libre/Open Source Software (FLOSS) and Software Assurance/Software Security*, December 2006. www.dwheeler.com/essays/oss_software_assurance.pdf.
53. WinSCP. Free sftp and scp client for windows. winscp.net/eng/index.php.
54. R.R. Yager. On ordered weighted averaging aggregation operators in multicriteria decision making. *IEEE Transaction Systems, Man, Cybernetics*, 18(1).
55. T. Ylönen. Ssh - secure login connections over the internet. In *Proc. of the Sixth USENIX Security Symposium*, July.