Adopting Open Source for Mission-Critical Applications: A Case Study on Single Sign-On

Claudio Agostino Ardagna[1], Ernesto Damiani[1], Fulvio Frati[1], and Salvatore Reale[2]

[1] University of Milan - via Bramante 65, Crema (CR), Italy
ardagna,damiani,frati@dti.unimi.it
[2] Siemens S.p.A.
Carrier Research & Development Radio Access - Network Management, Via Monfalcone 1, 20092, Cinisello Balsamo (MI), Italy salvatore.reale@siemens.com

Abstract. In this paper, we describe a specific selection process for security-related open source code, based on a methodology aimed at evaluating open source security frameworks in general and Single-Sign-On (SSO) systems in particular. Our evaluation criteria for open source security-related software include the community's timeliness of reaction against newly discovered vulnerabilities or incidents.

Keywords: Open Source, Security, Single Sign-On, Authentication, Federation, Trust Model.

1 Introduction

Accessing information on the global Net has become a fundamental requirement of the modern economy. Recently, focus has shifted from access to data stored in WWW sites to invoking *e-services* such as e-Government (e-Gov) services, remote banking, or airline reservation systems [4]. In the above scenario, the problem of securing access to network resources is of paramount importance. More specifically, security requirements include: *i) confidentiality,* data should be released to authorized users only; *ii) integrity,* unauthorized data insertion, modification or deletion must be prevented; *iii) availability* users must always be able to access data whereby they are authorized for, preventing, for instance, attacks such as *Denial of Service* (DoS). In order to satisfy these requirements, some basic security mechanisms are available:

- *identification and authentication* supporting users identification and verification of their identity;
- *access control* evaluating access requests submitted by users against predefined access control rules in order to grant or deny the access;
- *audit* monitoring access requests post-evaluation, to find out security infringements;

Please use the following format when citing this chapter:

Ardagna, C.A., Damiani, E., Frati, F., and Reale, S., 2006, in IFIP International Federation for Information Processing, Volume 203, Open Source Systems, eds. Damiani, E., Fitzgerald, B., Scacchi, W., Scotto, M., Succi, G., (Boston: Springer), pp. 209-220

- *cryptography* protecting data integrity and confidentiality by ensuring that data stored or transmitted are kept secret and only authorized users can decrypt them.

Security issues represent a critical aspect for most software applications. Due to the criticality of this requirement, proprietary solutions are widespread, because many companies consider them more secure and reliable. Adoption of open source solutions, especially at the middleware level, is slowed down by the fact that most companies do not completely trust the open source community and consider open source middleware a potential "backdoor" for attackers, 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, 18]. It is also widely acknowledged that open source solutions may in the end improve security, as they give both attackers and defenders greater visibility of software vulnerabilities [9]. In this paper, we discuss the idea of adopting open source for some key security-related functionalities, including access control and authentication systems, and discuss the requirements that open source security solutions must follow to be suitable for large scale deployment. In particular, our work focuses on open source Single Sign-On (SSO) solutions [2]. SSO gives a mechanism to manage authentication process and 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.

2 Basic Concepts

The huge amount of services available on the Net has caused unchecked proliferation of user accounts. Typically, users have to log-on to multiple systems, each of which may require different usernames and authentication information. All these account may be managed independently by local administrators within each individual system [12, 11]. SSO [8] systems are security frameworks aimed at simplifying log-on process, managing users multiple identities and presenting users credentials to network applications for authentication. SSO approach provides reduction of time spent by the users during log-on operations to individual domains, failed log-on transactions, time used to log-on to secondary domains, and costs and time used for users profiles administrations. SSO also increases services usability and provides simple administration thanks to a single, centralized administration point. Additional motivations that suggest SSO adoption are provided by *Sarvanes Oxley* (SOX) directive and the *Health Insurance Portability and Accountability Act* (HIPAA) that mandate provisions for maintaining the integrity of user profile data as an essential component of an effective security policy. HIPAA, for example, explicitly states that companies are required to assign a unique profile for tracking user identities. Also, it mandates procedures for creating, changing, and safeguarding profiles. Traditional

authentication policies infrastructures do not even come close to fulfilling these requirements.

2.1 Requirements of a Single Sign-on solution

We are now ready to list the requirements that a Single Sign-On solution should satisfy [2]. Our analysis brought us to formulating the following seven functional requirements: *i) Basic Authentication*: SSO systems must provide an authentication mechanism. Usually, authentication is performed through the classic username/password log-in, whereby a user can be unambiguously identified; *ii) Strong Authentication*: for highly secure environments, the traditional username/password authentication mechanism must be integrated with strong authentication mechanisms based on biometric properties of the user (fingerprints, retina scan, and so on); *iii) Authorization*: after the authentication process, the system must determine the level of information/services the requestor can see/use. *iv) Secure Exchange of Client Status Information*: the SSO system architecture implies the exchange of user information in secure manner between SSO server and remote services during authentication and authorization processes *v) Multi-domain Management*: the SSO system could provide support for managing authorizations (e.g. role acquisitions and revocations) that apply to multiple domains; *vi) Provisioning*: a provision is a pre-condition that must be met before an action can be executed. It is responsibility of the user to ensure that requests are sent only to environments satisfying all pre-conditions; *vii) Federation*: a user should be able to select the services 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.

Several non-functional requirements can also be identified, namely:

i) Autonomy a SSO server should be a stand alone module in order to clearly separate the authorization point from business implementations, avoiding the replication and the ad-hoc implementation of authorization mechanisms for each domain; *ii) Standard Compliance*: it is important for a SSO to support standard communication protocols fostering integration in different environments; *iii) Centralized Management*: centralization of authentication and authorization mechanisms and, more in general, centralization of identity management implies a simplification of the user profile management task; *iv) Cross-Language availability*: SSO solutions should permit the integration of services implementation based on different languages, without substantial changes on services code; *v) Password Proliferation Prevention*: the system should support parsimonious creation of costly resources such as passwords and public-private key pairs.

3 Open source Single Sign-on systems

Now, we shall briefly introduce some Open Source Single Sign-on systems. Our description will be made with reference to the above requirements and some

other evaluation parameters. For more architectural details about these Single Sign-on systems see [2].

Central Authentication Service. Central Authentication Service (CAS) [5, 20] is an open source framework developed at Yale University. It implements a SSO mechanism aimed at providing a *Centralized Authentication* to a single server and *HTTP redirections*. 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 by the authentication server to the application during redirections by using session cookies. CAS is composed of modular Java servlets that can run over any servlet engine and provides a web-based authentication service.

SourceID. SourceID [19], first released in 2001 by Ping Identity Corporation Company, is an open source multi-protocol project for enabling identity federation and cross-boundary security. SourceID focuses on simple integration and deployment within existing Web applications and provides high-level developer functionalities and customization. SourceID also implements Liberty Alliance Single Sign-On specifications [16] and it is a framework that integrates SSO features into new and existing Web portals. The lower level implementation of Liberty specifications, as for instance SOAP, SAML, Liberty features, protocols and metadata schemas, are transparent for Web developers. From the architectural point of view, SourceID system is composed by three modules plugged into the middle of Web applications to provide SSO facilities: *i) Profile* implements the Liberty Single Sign-On features, as for instance Federation, Single Sign-On and Log-Out, *ii) Message* provides features to create specific XML messages (for instance Liberty protocol and authentication), and *iii) Utility* provides functionality as Exception Handling, Data Format encoding and decoding.

Shibboleth. Shibboleth [17] is an open source implementation of Internet2/MACE, aimed at developing architectures, policy structures, practical technologies, to support sharing of Web resources subject to access control. Shibboleth is not only a SSO implementation, but it is a more general architecture that tries to protect privacy and more in general to manage user credentials. However, in this paper, we focus on the Shibboleth SSO implementation that is very close to Liberty Single Sign-on specifications [16]. The lower level implementation relies on different standards as HTTP, XML, XML Schema, XML Signature, SOAP and SAML. As in Liberty Alliance approach, Shibboleth uses Federation concept, named *Shibboleth Club*, between identity and service providers.

Java Open Single Sign-On (JOSSO). T Java Open Single Sign-On (JOSSO) is an open source J2EE-based SSO infrastructure aimed at providing a solution for centralized platform-neutral user authentication[14]. In the JOSSO

architecture we can identify three main actors: *i) Partner application,* a web application that uses SSO Gateway services to authenticate users; *ii) SSO Gateway,* represents the SSO server and provides authentication services to users who need authentication with partner applications; *iii) SSO Agent,* is a SSO Gateway client installed on managed services. More specifically, JOSSO supplies: *i) components-based framework,* since it provides a component-oriented infrastructure to support multiple authentication scheme, credential, and session stores, *ii) support for integration with Tomcat web container,* without requiring code customization, *iii) cross platform,* allowing integration with Java and non-Java applications, using standard solutions such as JAAS, SOAP, EJB, servlet/JSP and Struts, and *iv) support for strong authentication,* through the use of X.509 standard certificates.

Open Web SSO. The Open Web SSO [15] project provides core identity services for implementing transparent Single Sign-On as an infrastructure security component. In this paper, we will do not discuss Open Web SSO in detail because it is still in a very early stage of development.

4 Evaluation of OSS Single Sign On Systems

Generally speaking, few organizations rely on internal guidelines for the selection of open source products. In most cases, users select an open source solution which is readily available and fulfills their functional requirements. Several researchers [6, 10] have proposed more complex methodologies dealing with the evaluation of open source products from different perspectives, such as code quality, development flow and community composition and participation. In this paper, we put forward the idea of a specific selection process for security-related open source code. A major challenge is to establish a security-specific evaluation methodology capable of reducing users mistrust. e.g. due to the feeling that security open source applications are an "intrinsic backdoor" for attackers. Our main evaluation criteria highlight the promptness of reacting against newly discovered vulnerabilities or incidents. Applications success depend on the above principle because a low reaction rate to new vulnerabilities or incidents implies higher risk for users that adopt the software, potentially causing loss of information and money.

4.1 Evaluation principles

To select and find out the metrics that have to be evaluated in order to compare different security-related OSS implementations, let us first spell out the principles our analysis will be based on. We consider six partially overlapping macro-areas:

Generic Aspects (GA). An open source application must be categorized in terms of its generic aspects, i.e. ones not related to its purpose or scope,

including all the quantitative attributes proposed in the literature [6] that effectively describe a generic open source implementation. Such aspects include: the duration and size of the project, the programming language, the number of downloads and accesses.

Developers Community (DC). A critical success factor for any open source project is the composition and diversity of the developers community. A high number of developers allows sharing of diverse backgrounds and skills, giving vitality and freshness to the community and helping in solving problems, including bugs definition and fixing. Examples of DC properties are the number of developers and their roles, the existence of a core group and its stability over time.

Users Community (UC). The success of an open source application can be measured in terms of number and profile of the users that adopt it and rely on it. Obviously, measuring and evaluating the users community is less simple than doing so for developers because users interacting with an open source project are often anonymous. The overall quality of the users community, however, can be estimated by means of the number of downloads, the number of requests, the number of posts inside the forum, and the number of users subscribed to the mailing list. A qualitative measure of this macroarea could be the profile of the users adopting the project: if users belong to well-known companies or organizations and report positive results, their importance arises.

Software Quality (SQ). This area include metrics of quality built into the software by the requirements, design, code and verification processes to ensure that reliability, maintainability, and other quality factors are met. A subset of this macro area is the evaluation of code quality via coarse-grained factors such as operating system support, language support, level of modularity, compliance with the standards and so forth.[1]

Documentation and Interaction support (DIS). This macro area is composed of two major sub-areas: traditional documentation that explains the characteristics, functionalities and peculiarities of the software and support in terms of time allotted by developers to give feedback about the project and documentation, through forums, mailing lists, whitepapers, and presentations.

Integration and Adaptability with new and existing technologies (IA). A fundamental tenet of open source projects is full integration with existing technologies at project startup and a high level of adaptability to new technologies presented during project life. Another aspect that arise is the ability of the developers community to solve and fix bugs and react to new vulnerabilities.

[1] As far as evaluating code quality is concerned, we remark that open source SSO systems lend themselves to quality assurance and evaluation based on shared testing and code walkthrough as outlined in [1]. However, comparing reference implementations based on code walkthrough is outside the scope of this paper.

4.2 Evaluation parameters

In this section we provide a description of the metrics (see Table 1 and 2) we used to evaluate critical open source security applications. This set of metrics will be later used for comparing open source SSO architectures (see Section 5).

Within the above areas, we can now define quantitative metrics. They can be orthogonally divided in two categories: *i) Core Metrics (CM),* including all metrics that can be readily computed from current technologies, statistics, and information on the projects; *ii) Advanced Metrics (AM),* including all parameters that require additional information and some privileged access to the development group. Advanced metrics may be available only as rough estimates or not available entirely. A brief definition of the parameters semantics is shown in Table 1 and 2. For a detailed explanation of advanced metrics, we refer to Section 4.3.

4.3 Advanced Metrics

Advanced Metrics represent the evaluation parameters that would require privileged access to the developers community. Otherwise, they can be estimated based on raw data. In particular, we propose three major metrics: *i) Reaction Rate,* estimating the average time the developers community took to find solutions to newly discovered vulnerabilities. This parameter measures the community vitality in reacting against vulnerabilities that represent the main problem in security applications; *ii) Incident Frequency,* which measures the robustness of the application with respect to discovered vulnerabilities; *iii) Group/Developers Stability,* which measures the degree of stability of developers group. Regarding the first two parameters, we remarks that various security-related Web portal provides databases that contain information about vulnerabilities and related incidents summaries. In particular, three main portals stand out: *Secunia* (`http://secunia.com/`) that offers monitoring of vulnerabilities in more than 6000 products, *Open Source Vulnerability Database* (OSVDB) (`http://www.osvdb.org/`) an independent database that provides technical information about vulnerabilities and, finally, CERT that provides a database containing information about vulnerabilities, incidents and fixes. Further, we describe how to use the CERT database, the more complete and well supported repository of security concerns, in order to describe problems related to vulnerabilities and incidents prevention. The last metrics, *Groups/Developers Stability,* is not easy to estimate from outside the developers community, due to the fact that does not exist a formal categorization of the information related to the users and developers that belong to a particular project. It may be however available to insiders, e.g. to companies that adopted an open source product and openly contribute to its community.

CERT The Computer Emergency Response Team (CERT) [7] is an organization focused on ensuring that appropriate technologies and systems management practices are used to resist to attacks on networked systems, to

Core Metrics			
Name	**Definition**	**Values**	**Area**
Age	Age of the project	Days	GA
Project Core Group	Evaluate the existence of a group of core developers. Further analysis could evaluate the composition of the group	Boolean	GA,DC
Number of Core Developers	Number of core developers contributing the project. Core developers are defined as the persons that contributes both to the project management and code implementation	Integer	DC
Number of Releases	Number of releases since project start up	Integer	SQ,IA
Bug Fixing Rate	Measures the rate of bug fixed. This rate is computed as: $\frac{\#ofbugsfixed}{\#ofbugsdetected}$	[0..100]	SQ,IA
Update Average Time	Measures the vitality of developers group and in other word the mean number of days to wait for a new update (releases or patches). This metrics is computed as: $\frac{age}{\#ofpatches+\#ofreleases}$	days	SQ,IA
Forum and Mailing List Support	Check forum and mailing list availability	boolean	GA,DIS
Number of Users	Number of users that adopt the application. When not available, this parameter is approximated as: $\frac{\#ofdownloads}{\#ofreleases}$	Integer	UC
Documentation Level	Level of documentation of a project, in terms of API, user manuals, whitepapers	Mbyte	DIS
Code Quality	Qualitative measure of code quality. Several standard source code metrics could be adopted.		SQ,IA
Community Vitality	Represents the vitality of the community in terms of number of forum threads and replies: $\frac{\#offorumreplies}{\#offorumthreads}$	Real	DC,UC

Table 1. Evaluation Metrics Definition: Core Metrics

limit damages and ensure continuity of critical services despite successful attacks, accidents, or failures. The CERT is located at the Software Engineering Institute (SEI), a Federally Funded Research and Development Center (FFRDC) operated by Carnegie Mellon University. The CERT Coordination Center (CERT/CC), a major center for internet security problems, component of the larger CERT Program, was established in November 1988 after that the "Morris Worm" brought down much of the internet and demonstrated the growing network susceptibility to attack. For the purposes of the present paper, we take into consideration CERT information about vulnerabilities, incidents and vulnerabilities fixing, which provides the raw data over which our advanced metrics are computed.

Advanced Metrics			
Name	**Definition**	**Values**	**Area**
Reaction Rate	Average time needed by the developers community to find solutions for newly discovered vulnerabilities. More specifically, it represents the project developers ability in reacting to the set V of vulnerabilities. It is defined as follows: $\frac{UpdateAverageTime}{\frac{\sum_{i=1}^{n}(Fixing\_Date(V_i) - Discovering\_Date(V_i))}{n}}$ where $V_i \in V$ and $n = \|V\|$		IA
Incident Frequency	Measures the number of incidents due to vulnerabilities. This parameter is computed as: $\frac{\#ofincidents}{\|V\|}$		IA
Group/ Developers Stability	Measures the degree of stability of a developers group. Each developer is classified as *stable* or *transient* where stable is a developer that continuously contributes code. The exact number of contributions to make a developer stable are project-dependent. This value is computed as: $\frac{\#ofstabledevelopers}{\#ofdevelopers} * 100$	[0..100%]	DC

Table 2. Evaluation Metrics Definition: Advanced Metrics

US-CERT Vulnerability Notes Database A vulnerability [13] is defined as a set of conditions that leads or may lead to an implicit or explicit failure of the confidentiality, integrity, or availability of an information system. Examples of the unauthorized or unexpected effects of a vulnerability may include executing commands as another user, accessing data in excess of specified or expected permission, posing as another user or service within a system, causing an abnormal denial of service, inadvertently or intentionally destroying data without permission and exploiting an encryption implementation weakness that significantly reduces the time or computation required to recover the plain text from an encrypted message. Common causes of vulnerabilities are design flaws in software and hardware, patched administrative processes, lack of awareness and education in information security, and advancements in the state of the art or improvements to current practices, any of which may result in real threats to mission-critical information systems. The accidental introduction of defects into software is expected to comprise a significant portion of the vulnerabilities addressed by this framework. CERT alerts users to potential vulnerabilities to the security of their systems and provide information about how to avoid, minimize, or recover from the damage. A vulnerabilities database is maintained by US-CERT [21] and contains descriptions of vulnerabilities, their impacts, and solutions. US-CERT publishes information on a wide variety of vulnerabilities. Descriptions of these vulnerabilities are available from this web page in a searchable database format, and are published as "US-CERT Vulnerability Notes".

The notes are very similar to alerts, but they may have less complete information. In particular, solutions may not be available for all the vulnerabilities in this database. The US-CERT Vulnerability Notes database is cross-referenced with the Common Vulnerabilities and Exposures (CVE) catalog.

CERT/CC Incident Notes CERT Incident Notes have become a core component of US-CERT's Technical Cyber Security Alerts and Current Activity; this bulletin provides information about the exploiting of the vulnerabilities to convey an attack to the affected systems. In particular, incident notes provide information such as the overview and description of the incident and optionally the solution to the vulnerability that causes the incident.

Vulnerability Fixing US-CERT Vulnerability Notes Database and CERT/CC Incident Notes provides additional information about the solution applied to fix the discovered vulnerabilities. It is widely acknowledged that most of the incident reports of computer break-ins received at the CERT/CC could have been prevented if system administrators and users kept their computers up-to-date with patches and security fixes. US-CERT provides only the link to the available patches and security fixes that are usually hosted on the vendor sites. In summary, most information necessary to calculate the provided advanced metrics set is already available on the Net. Unfortunately, this information is in raw format and then is difficult to automatize the calculation of the metrics. Substantial pre-processing is needed to compute these metrics, that are of paramount importance in evaluating the risk of open source security applications adoption. We are currently working on a tool for security metrics (Sect. 6)

5 Open Source Comparison

Table 3 gives a comparison of open source Single Sign-On implementations. Before discussing it, we remark that while CAS, SourceID and JOSSO are fully dedicated SSO systems, Shibboleth is a more comprehensive framework which contains, among other things, a SSO implementation. Focusing on the comparison, we remark that as shown by the table, all the analyzed systems are quite stable due to the fact that their startup happens more than a year ago. The CAS implementation stands out; it has a long time history because it started about five years ago. A common characteristic of the projects is that they are managed by a consolidated core group that gives stability to the project and coordination to open source community. Also the level of documentation is similar and is included between `6.80 MB` of JOSSO and `10.05 MB` of CAS. Although CAS seems the more lively project due to the great number of releases, we argue that the more active and viable implementation is JOSSO, because it provides a new release every `21 days`, while CAS implementation only provided a release every `79 days`. This gap could give to adopters of the JOSSO

Metrics	CAS	SourceID	Shibboleth	JOSSO
Age (GA)	1500 days	812 days	926 days	489 days
Project Core Group (GA,DC)	Yes	Yes	Yes	Yes
Number of Core Developers (DC)	5	N/A	5	2
Number of Releases (SQ,IA)	19	7	10	7
Bug Fixing Rate (SQ,IA)	N/A	N/A	0%	67%
Update Average Time (SQ,IA)	79 days	116 days	92,6 days	21 days
Forum and Mailing List Support (GA,DIS)	Mailing List Only	Mailing List Only	Mailing List Only	Yes
Number of Users (UC)	45	N/A	N/A	3161 approx.
Documentation Level (DIS)	10.05 MB	8.96 MB	7.04 MB	6.80 MB
Community Vitality (DC,UC)	N/A	N/A	N/A	3,12

Table 3. Comparison of proposed implementations at 31 December 2005

framework an higher assurance of the project's reliability, because continuous releases keep the implementation up to date and resistant to new technologies and vulnerabilities. However, JOSSO very short update time is also influenced by the fact that the project is the youngest; probably, in the next year, the update average time will rise although it will probably maintain the lowest update average time. Regarding other metrics, for the sake of conciseness we avoid a complete discussion. It is easy to see that JOSSO is the only implementation that furnishes all the information allowing a complete metrics measurement. To conclude this overview, our analysis showed that JOSSO is the most suitable and flexible open source SSO solution if analyzed from security point of view.

6 Conclusions

In this paper, we presented a quantitative approach to the comparative evaluation of security-related software. Then as a case-study, we compared five major implementations of Single-Sign-On systems. Our evaluation methodology relates on a structured set of metrics specifically designed for security-related open source systems. Some of these metrics are based on event logs of some well-known security portals (e.g., the CERT one) and their computation would be made much easier should CERT support some level of data warehousing. We are now working on a tool for creating a warehouse of quantitative data about security events to be used in the framework of our evaluation.

Acknowledgments

This work was supported in part by the European Union within the PRIME Project in the FP6/IST Programme under contract IST-2002-507591 and by the Italian MIUR within the KIWI and MAPS projects.

References

1. S. Abiteboul, X. Leroy, B. Vrdoljak, R. Di Cosmo, S. Fermigier, S. Lauriere, F. Lepied, R. Pop, F. Villard, J.P. Smets, C. Bryce, K.R. Dittrich, T. Milo, A. Sagi, Y. Shtossel, and E. Panto. Edos: Environment for the development and distribution of open source software. In *The First International Conference on Open Source Systems*, pages 66–70, Genova (Italy), July 2005.
2. C.A. Ardagna, E. Damiani, S. De Capitani di Vimercati, F. Frati, and P. Samarati. CAS++: an open source single sign-on solution for secure e-services. *Submitted to 21st IFIP International Information Security Conference "Security and Privacy in Dynamic Environments"*, May 2006.
3. C.A. Ardagna, E. Damiani, F. Frati, and M. Madravio. Open source solution to secure e-government services. *Encyclopedia of Digital Government*, 2006.
4. C.A. Ardagna, E. Damiani, F. Frati, and M. Montel. Using open source middleware for securing e-gov applications. In *The First International Conference on Open Source Systems*, pages 172–178, Genova (Italy), July 2005.
5. P. Aubry, V. Mathieu, and J. Marchal. Esup-portal: open source single sign-on with cas (central authentication service). In *Proceedings of EUNIS04 - IT Innovation in a Changing World*, pages 172–178, Bled (Slovenia), 2005.
6. A. Capiluppi, P. Lago, and M. Morisio. Characteristics of open source projects. In *CSMR*, page 317, 2003.
7. CERT-CC. Cert coordination center. `http://www.cert.org/`.
8. Jan De Clercq. Single sign-on architectures. In *International Conference on Infrastructure Security, InfraSec, LNCS*, 2002.
9. C. Cowan. Software security for open-source systems. *IEEE-SEC-PRIV*, 1(1):38–45, January/February 2003.
10. J. Feller and B. Fitzgerald. A framework analysis of the open source software development paradigm. In *ICIS*, pages 58–69, 2000.
11. B. Galbraith and et al. *Professional Web Services Security.* Wrox Press, 2002.
12. The Open Group. Single sign-on. `http://www.opengroup.org/security/sso/`.
13. John T. Chambers and John W. Thompson. Vulnerability disclosure framework. Final report and recommendations by the council, National Infrastructure Advisory Council, January 2004.
14. JOSSO. Java open single sign-on. `http://sourceforge.net/projects/josso`.
15. OpenSSO. Open web sso. `https://opensso.dev.java.net/`.
16. Liberty Alliance Project. `http://www.projectliberty.org/`.
17. Shibboleth Project. `http://shibboleth.internet2.edu/`.
18. E.S. Raymond. The cathedral and the bazaar. `http://www.openresources.com/documents/cathedral-bazaar/`, August 1998.
19. SourceID. Open source federated identity management. `http://www.sourceid.org/`.
20. Yale University. Central authentication service. `http://tp.its.yale.edu/tiki/tiki-index.php?page=CentralAuthenticationService`.
21. US-CERT. Vulnerability notes database. `http://www.kb.cert.org/vuls/`.