NPS-AM-11-C8P02R03-024

EXCERPT FROM THE

PROCEEDINGS

OF THE

EIGHTH ANNUAL ACQUISITION RESEARCH SYMPOSIUM WEDNESDAY SESSIONS VOLUME I

Advances in the Acquisition of Secure Systems Based on Open Architectures

Walt Scacchi and Thomas Alspaugh, Institute for Software Research

Published: 30 April 2011

Prepared for the Naval Postgraduate School, Monterey, California 93943

ACQUISITION RESEARCH PROGRAM
GRADUATE SCHOOL OF BUSINESS & PUBLIC POLICY
NAVAL POSTGRADUATE SCHOOL

The research presented at the symposium was supported by the Acquisition Chair of the Graduate School of Business & Public Policy at the Naval Postgraduate School.

To request Defense Acquisition Research or to become a research sponsor, please contact:

NPS Acquisition Research Program
Attn: James B. Greene, RADM, USN, (Ret.)
Acquisition Chair
Graduate School of Business and Public Policy
Naval Postgraduate School
555 Dyer Road, Room 332
Monterey, CA 93943-5103
Tel: (831) 656-2092
Fax: (831) 656-2253
E-mail: jbgreene@nps.edu

Copies of the Acquisition Sponsored Research Reports may be printed from our website www.acquisitionresearch.net

ACQUISITION RESEARCH PROGRAM
GRADUATE SCHOOL OF BUSINESS & PUBLIC POLICY
NAVAL POSTGRADUATE SCHOOL

Preface & Acknowledgements

During his internship with the Graduate School of Business & Public Policy in June 2010, U.S. Air Force Academy Cadet Chase Lane surveyed the activities of the Naval Postgraduate School's Acquisition Research Program in its first seven years. The sheer volume of research products—almost 600 published papers (e.g., technical reports, journal articles, theses)—indicates the extent to which the depth and breadth of acquisition research has increased during these years. Over 300 authors contributed to these works, which means that the pool of those who have had significant intellectual engagement with acquisition issues has increased substantially. The broad range of research topics includes acquisition reform, defense industry, fielding, contracting, interoperability, organizational behavior, risk management, cost estimating, and many others. Approaches range from conceptual and exploratory studies to develop propositions about various aspects of acquisition, to applied and statistical analyses to test specific hypotheses. Methodologies include case studies, modeling, surveys, and experiments. On the whole, such findings make us both grateful for the ARP's progress to date, and hopeful that this progress in research will lead to substantive improvements in the DoD's acquisition outcomes.

As pragmatists, we of course recognize that such change can only occur to the extent that the potential knowledge wrapped up in these products is put to use and tested to determine its value. We take seriously the pernicious effects of the so-called "theory–practice" gap, which would separate the acquisition scholar from the acquisition practitioner, and relegate the scholar's work to mere academic "shelfware." Some design features of our program that we believe help avoid these effects include the following: connecting researchers with practitioners on specific projects; requiring researchers to brief sponsors on project findings as a condition of funding award; "pushing" potentially high-impact research reports (e.g., via overnight shipping) to selected practitioners and policy-makers; and most notably, sponsoring this symposium, which we craft intentionally as an opportunity for fruitful, lasting connections between scholars and practitioners.

A former Defense Acquisition Executive, responding to a comment that academic research was not generally useful in acquisition practice, opined, "That's not their [the academics'] problem—it's ours [the practitioners']. They can only perform research; it's up to us to use it." While we certainly agree with this sentiment, we also recognize that any research, however theoretical, must point to some termination in action; academics have a responsibility to make their work intelligible to practitioners. Thus we continue to seek projects that both comport with solid standards of scholarship, and address relevant acquisition issues. These years of experience have shown us the difficulty in attempting to balance these two objectives, but we are convinced that the attempt is absolutely essential if any real improvement is to be realized.

We gratefully acknowledge the ongoing support and leadership of our sponsors, whose foresight and vision have assured the continuing success of the Acquisition Research Program:

- Office of the Under Secretary of Defense (Acquisition, Technology & Logistics)
- Program Executive Officer SHIPS
- Commander, Naval Sea Systems Command
- Army Contracting Command, U.S. Army Materiel Command
- Program Manager, Airborne, Maritime and Fixed Station Joint Tactical Radio System

- Program Executive Officer Integrated Warfare Systems
- Office of the Assistant Secretary of the Air Force (Acquisition)
- Office of the Assistant Secretary of the Army (Acquisition, Logistics, & Technology)
- Deputy Assistant Secretary of the Navy (Acquisition & Logistics Management)
- Director, Strategic Systems Programs Office
- Deputy Director, Acquisition Career Management, US Army
- Defense Business Systems Acquisition Executive, Business Transformation Agency
- Office of Procurement and Assistance Management Headquarters, Department of Energy

We also thank the Naval Postgraduate School Foundation and acknowledge its generous contributions in support of this Symposium.

James B. Greene, Jr.
Rear Admiral, U.S. Navy (Ret.)

Keith F. Snider, PhD
Associate Professor

Panel 2 – Advancing Open Architecture Acquisition

Wednesday, May 11, 2011	
11:15 a.m. – 12:45 p.m.	**Chair: Christopher Deegan,** Executive Director, Program Executive Office for Integrated Warfare Systems ***Delivering Savings with Open Architecture and Product Lines*** Brian Womble, USN, and William Schmidt, Mike Arendt, and Tim Fain, IBM ***An Architecture-Centric Approach for Acquiring Software-Reliant Systems*** Lawrence Jones and John Bergey, Software Engineering Institute ***Advances in the Acquisition of Secure Systems Based on Open Architectures*** Walt Scacchi and Thomas Alspaugh, Institute for Software Research

Christopher Deegan—Executive Director, Program Executive Officer, Integrated Warfare Systems (PEO IWS). Mr. Deegan directs the development, acquisition, and fleet support of 150 combat weapon system programs managed by 350 military and civilian personnel with annual appropriations of over $5 billion.

Mr. Deegan holds a Bachelor of Science degree in Industrial Engineering from Penn State University, University Park, Pennsylvania and a Master of Science degree in Engineering from The Catholic University of America, Washington, DC. He is a graduate of the Program Managers Course, Defense Systems Management College, Fort Belvoir, VA. He is a Certified Acquisition Professional and is Level III certified in three DA WIA career fields: Program Management; Research and Systems Engineering; and Business, Cost Estimating and Financial Management.

Mr. Deegan is the only Comptroller employee to be recognized by the Association of Scientists and Engineers as "NAVSEA Engineer of the Year" (1993). He received the Assistant Secretary of the Navy (Research, Development and Acquisition) and NAVSEA Acquisition Excellence Awards (1996), the David Packard Award for Governmental Excellence (1996), the Navy's Meritorious Civilian Service Award (1997), the Navy's Competition and Procurement Excellence Award (2003), and a Meritorious Unit Commendation Medal as a member of the SEA WOLF Program Office (2006). Mr. Deegan was awarded the Presidential Rank of Meritorious Executive in October 2007.

Advances in the Acquisition of Secure Systems Based on Open Architectures

Walt Scacchi—Senior Research Scientist and Research Faculty Member, Institute for Software Research, University of California, Irvine. Dr. Scacchi received a PhD in Information and Computer Science from UC Irvine in 1981. From 1981–1998, he was on the faculty at the University of Southern California. In 1999, he joined the Institute for Software Research at UC Irvine. He has published more than 150 research papers and has directed 45 externally funded research projects. In 2007, he served as General Chair of the Third IFIP International Conference on Open Source Systems (OSS2007), Limerick, IE. In 2010, he chaired the Workshop on the Future of Research in Free and Open Source Software, Newport Beach, CA, for the Computing Community Consortium and the National Science Foundation. He also serves as Co-Chair of the Software Engineering in Practice (SEIP) Track at the 33rd International Conference on Software Engineering, May 21–28, 2011, Honolulu, HI. [wscacchi@ics.uci.edu]

Thomas Alspaugh—Adjunct Professor, Computer Science, Georgetown University, and Visiting Researcher, Institute for Software Research at UC Irvine. Dr. Alspaugh's research interests are in software engineering and software requirements. Before completing his PhD, he worked as a software developer, team lead, and manager in industry, and as a computer scientist at the Naval Research Laboratory on the Software Cost Reduction project, also known as the A-7E project. [thomas.alspaugh@acm.org]

Abstract

The role of software ecosystems in the development and evolution of secure open architecture systems has received insufficient consideration. Such systems are composed of software components subject to different security requirements in an architecture in which evolution can occur by evolving existing components or by replacing them. But this may result in possible security requirements conflicts and organizational liability for failure to fulfill security obligations. We have developed an approach for understanding and modeling software security requirements as "security licenses," as well as for analyzing conflicts among groups of such licenses in realistic system contexts and for guiding the acquisition, integration, or development of systems with open source components in such an environment. Consequently, this paper reports on our efforts to extend our existing approach to specifying and analyzing software intellectual property licenses to now address software security licenses that can be associated with secure OA systems.

Introduction

A substantial number of development organizations are adopting a strategy in which a software-intensive system is developed with an open architecture (OA; Oreizy, 2000), whose components may be open source software (OSS) or proprietary with open application programming interfaces (APIs). Such systems evolve not only through the evolution of their individual components, but also through replacement of one component by another, possibly from a different producer or under a different license. With this approach, the organization becomes an integrator of components largely produced elsewhere that are interconnected through open APIs as necessary to achieve the desired result.

An OA development process results in an ecosystem in which the integrator is influenced from one direction by the goals, interfaces, license choices, and release cycles of the component producers, and in another direction by the needs of its consumers. As a result, the software components are reused more widely, and the resulting OA systems can

achieve reuse benefits such as reduced costs, increased reliability, and potentially increased agility in evolving to meet changing needs.

An emerging challenge is to realize the benefits of this approach when the individual components are subject to different security requirements. This may arise due either to how a component's external interfaces are specified and defended, or to how system components are interconnected and configured in ways that can or cannot defend the composed system from security vulnerabilities and external exploits. Ideally, any software element in a system composed from components from different producers can have its security capabilities specified, analyzed, and implemented at system architectural design-time, build-time, or at deployment run-time. Such capability-based security in simplest form specifies what types, value ranges, and values of data, or control signals (e.g., program invocations, procedure, or method calls), can be input, output, or handed off to a software plug-in or external (helper) application, from a software component or composed system.

When designing a secure OA system, decisions and trade-offs must be made as to what level of security is required, as well as what kinds of threats to security must be addressed. The universe of possible security threats is continually emerging and the cost/effort of defending against them, ongoing. Similarly, anticipating all possible security vulnerabilities or threats is impractical (or impossible). Further, though it may be desirable that all systems be secure, different systems need different levels of security, which may come at ever greater cost or inconvenience to accommodate. Strategic systems may need the greatest security possible, while other systems may require much less rigorous security mechanisms. Thus, finding an affordable, scalable, and testable means for specifying the security requirements of software components, or OA systems composed with components with different security requirements, is the goal of our research.

The most basic form of security requirements that can be asserted and tested are those associated with virtual machines. Virtual machines (VM) abstract away the actual functional or processing capabilities of the computational systems on which they operate, and instead provide a limited functionality computing surround (or "sandbox"). VM can isolate a given component or system other software applications, utilities, repositories, or external/remote control data access (input or output). The capabilities for a VM (e.g., an explicit, pre-defined list of approved operating system commands or programs that can write data or access a repository) can be specified as testable conditions that can be assigned to users or programs authorized to operate within the VM. The VM technique is now widely employed through software "hypervisors" (e.g., IBM VM/370, VMware, VirtualBox, Parallels Desktop for Mac) that isolate software applications and operating system from the underlying system platform or hardware. Such VMs act like "containment vessels" through which it is possible to specify barriers to entry (and exit) of data and control via security capabilities that restrict other programs. These capabilities thus specify what rights or obligations may be, or may not be, available for access or update to data or control information. Thus architectural design-time decisions pertaining to specifying the security rights or obligations for the overall system or its components are done by specification of VMs that contain the composed system or its components. These rights or obligations can be specified as pre-conditions on input data or control signals, or post-conditions on output data or control signals.

The problem of specifying the build-time and run-time security requirements of OA systems is different from that at design-time. In determining how to specify the software build sequence, security requirements are manifest as capabilities that may be specific to explicitly declared versions of designated programs. For example, if an OA system specifies a "Web browser" as one of its components at design-time, at build-time a particular Web

browser (Mozilla Firefox or Internet Explorer) must then be specified, as must its baseline version (e.g., Firefox 4.0 or Internet Explorer 9.0). However, if the resulting run-time version of the OA system must instead employ a locally available Web browser (e.g., Firefox 3.6.1 or Internet Explorer 8.0 Service Pack 2), then the OA system integrators may either need to produce multiple run-time versions for deployment, or else build the OA system using either (a) an earlier version of the necessary component (e.g., Firefox 3.5 or Internet Explorer 7.0) that is "upward compatible"; (b) a stub or abstract program interface that allows for a later designated compatible component version to be installed/used at run-time; or else, (c) create different run-time version alternatives (i.e., variants) of the target OA systems that may or not be "backward compatible" with the system component versions available in the deployment run-time environment. The need to specify build-time and run-time components by versions (and possibly timestamps of their creation or local installation) arises since evolutionary version updates often include security patches that close known vulnerabilities or prevent known exploits. As indicated in the Related Research section below, security attacks often rely on system entry through known vulnerabilities that are present in earlier versions of software components that have not been updated to newer versions that do not have the same vulnerabilities.

As we have been able to address an analogous problem of how to specify and analyze the intellectual property rights and obligations of the licenses of software components, our efforts now focus on the challenge of how to specify and analyze software components and composed system security rights and obligations using a new information structure we call a "security license." The actual form of such a security license is still to be finalized, but at this point, we believe it is appropriate to begin to develop candidate forms or types of security licenses for further research and development, especially for security license forms that can be easily formalized, be readily applied to large-scale OA systems, as well as be automatically analyzed or tested. This is another goal of our research here.

Next, the challenge of specifying secure software systems composed from secure or insecure components is inevitably entwined with the software ecosystems that arise for secure OA systems. We find that an OA software ecosystem involves organizations and individuals producing and consuming components, and supply paths from producer to consumer; but also

- the OA of the system(s) in question, and how best to secure it,
- the open interfaces provided by the components, and how to specify their security requirements,
- the degree of coupling in the evolution of related components that can be assessed in terms of how security rights and obligations may change, and
- the rights and obligations resulting from the security licenses under which various components are released, that propagate from producers to consumers.

An example software ecosystem producing and integrating secure software components or secure systems is portrayed in Figure 1.

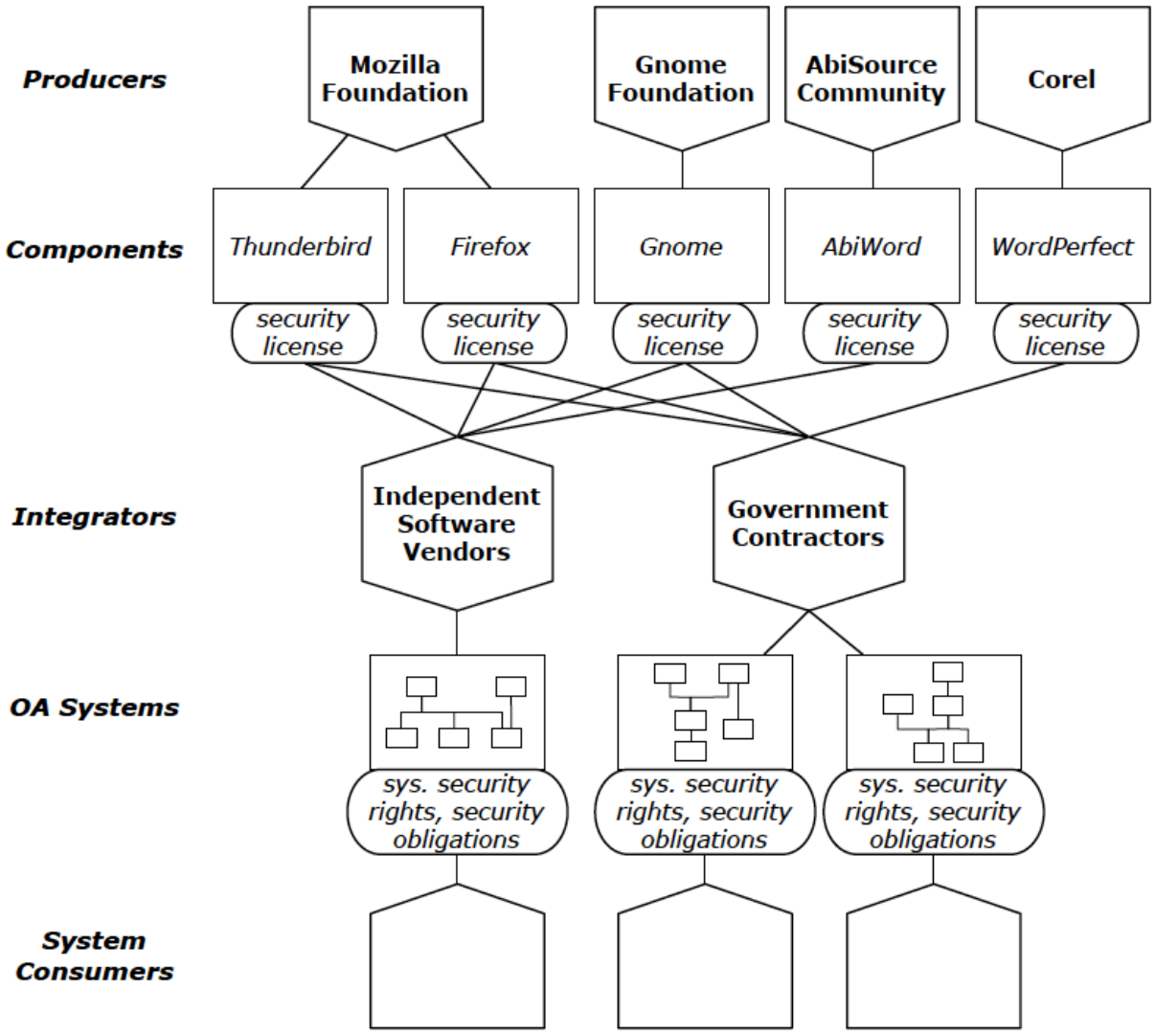

Figure 1. An Example of a Software Ecosystem in Which Secure OA Systems Are Developed

In order to most effectively use an OA approach in developing and evolving a system, it is essential to consider this OA ecosystem. An OA system draws on components from proprietary vendors and open source projects. Its architecture is made possible by the existing general ecosystem of producers, from which the initial components are chosen. The choice of a specific OA begins a specialized software ecosystem involving components that meet (or can be shimmed to meet) the open interfaces used in the architecture. We do not claim this is the best or the only way to reuse components or produce secure OA systems, but it is an ever more widespread way. In this paper, we build on previous work on heterogeneously-licensed systems (German & Hassan, 2009; Scacchi & Alspaugh, 2008; Alspaugh, Asuncion, & Scacchi, 2009a) by examining how OA development affects and is affected by software ecosystems, and the role of security licenses for components included within OA software ecosystems.

In the remainder of this paper, we survey some related work in the next section, define and examine characteristics of open architectures with or without secure software elements (in the Secure Open Architecture Composition section), define and examine

characteristics for how secure OA systems evolve (in the OA System Evolution section), introduce a structure for security licenses (in the Security Licenses section), outline license architectures (in the Security License Architectures section), and sketch our approach for license analysis (in the Security License Analysis section). We then close with a discussion addressing how our software license and analysis scheme relates to software products lines (the Discussion section), before stating our conclusions in the final section.

Related Work

Software systems, whether operating as standalone components, or as elements within large system compositions, are continuously being subjected to security attacks. These attacks seek to slip through software vulnerabilities known to the attackers, but perhaps not by the system integrators or consumers. These attacks often seek to access, manipulate, or remotely affect the data values or control signals that a component or composed system processes for nefarious purposes, or seek to congest or over-saturate networked services. Recent high profile security attacks like Stuxnet (Falliere, Murchu, & Chien, 2011) reveal that security attacks may be very well planned and employ a bundle of attack vectors and social engineering tactics in order for the attack to reach strategic systems that are mostly isolated and walled off from public computer networks. The Stuxnet attacks entered through software system interfaces at either the component, application subsystem, or base operating system level, and their goal was to go outside or beneath their entry context. However, all of the Stuxnet attacks on the targeted software system could be blocked or prevented through security capabilities associated with the open software interfaces that would (a) limit access or evolutionary update rights lacking proper authorization, as well as (b) “sandboxing” (i.e., isolating) and holding up any evolutionary updates (the attacks) prior to their installation and run-time deployment. Furthermore, as the Stuxnet attack involved the use of corrupted certificates of trust from approved authorities as false credentials that allowed evolutionary system updates to go forward, it seems clear that additional preventions are needed that are external to, and prior to, their installation and run-time deployment. In our case, that means we need to specify and analyze software security requirements and evolutionary update capabilities at architectural design-time and system integration build-time, and then reconcile those with the run-time system composition. It also calls for the need to maintain the design-time, build-time, and run-time system compositions in repositories remote from system installations, and in possibly redundant locations that can be encrypted, randomized, fragmented, and dispersed (e.g., via Torrents or “onion routing”), then cross-checked and independently verified prior to run-time deployment in a high security system application.

As already noted, both software intellectual property licenses, and security licenses represent a collection of rights and obligations for what can or cannot be done with a licensed software component. Licenses thus denote non-functional requirements that apply to a software system or system components as intellectual property (IP) or security requirements (i.e., capabilities) during their development and deployment. But rights and obligations are not limited to concerns or constraints applicable only to software as IP. Instead, they can be written in ways that stipulate non-functional requirements of different kinds. Consider, for example, that desired or necessary software system security properties can also be expressed as rights and obligations, addressing system confidentiality, integrity, accountability, system availability, and assurance (Breaux & Anton, 2005, 2008). Traditionally, developing robust specifications for non-functional software system security properties in natural language often produces specifications that are ambiguous, misleading, inconsistent across system components, and lacking sufficient details (Yau & Chen, 2006). Using a semantic model to formally specify the rights and obligations required for a software

system or component to be secure (Breaux & Anton, 2005, 2008; Yau & Chen, 2006) means that it may be possible to develop both a "security architecture" notation and model specification that associates given security rights and obligations across a software system, or system of systems. Similarly, it suggests the possibility of developing computational tools or interactive architecture development environments that can be used to specify, model, and analyze a software system's security architecture at different times in its development—design-time, build-time, and run-time. The approach we have been developing for the past few years for modeling and analyzing software system IP license architectures for OA systems (Alspaugh, Asuncion, & Scacchi, 2009b, 2010; Scacchi & Alspaugh, 2008) may therefore be extendable to also being able to address OA systems with heterogeneous "software security license" rights and obligations. Furthermore, the idea of common or reusable software security licenses may be analogous to the reusable security requirements templates proposed by Firesmith (2004) at the Software Engineering Institute. But such an exploration and extension of the semantic software license modeling, meta-modeling, and computational analysis tools to also support software system security can be recognized as a promising next stage of our research studies.

Secure Open Architecture Composition

Open architecture (OA) software is a customization technique introduced by Oreizy (2000) that enables third parties to modify a software system through its exposed architecture, evolving the system by replacing its components. Increasingly more software-intensive systems are developed using an OA strategy, not only with open source software (OSS) components, but also proprietary components with open APIs. Similarly, these components may or may not have their own security requirements that must be satisfied during their build-time integration or run-time deployment, such as registering the software component for automatic update and installation of new software versions that patch recently discovered security vulnerabilities or prevent invocation of known exploits. Using this approach can lower development costs and increase reliability and function, as well as adaptively evolve software security (Scacchi & Alspaugh, 2008). Composing a system with heterogeneously secured components, however, increases the likelihood of conflicts, liabilities, and no-rights stemming from incompatible security requirements. Thus, in our work, we define a secure OA system as *a software system consisting of components that are either open source or proprietary with open API, whose overall system rights at a minimum allow its use and redistribution, in full or in part such that they do not introduce new security vulnerabilities at the system architectural level*.

It may appear that using a system architecture that incorporate secure OSS and proprietary components, and uses open APIs, will result in a secure OA system. But not all such architectures will produce a secure OA, since the (possibly empty) set of available license rights for an OA system depends on the following: (a) how and why secure or insecure components and open APIs are located within the system architecture, (b) how components and open APIs are implemented, embedded, or interconnected, and (c) the degree to which the IP and security licenses of different OSS components encumber all or part of a software system's architecture into which they are integrated (Scacchi & Alspaugh, 2008; Alspaugh & Anton, 2008).

The following kinds of software elements appearing in common software architectures can affect whether the resulting systems are open or closed (Bass, Clements, & Kazman, 2003).

Software Source Code Components—These can be either (a) standalone programs; (b) libraries, frameworks, or middleware; (c) inter-application script code such as C shell

scripts; (d) intra-application script code, as for creating Rich Internet Applications using domain-specific languages such as XUL for the Firefox Web browser (Feldt, 2007) or "mashups" (Nelson & Churchill, 2006) whose source code is available and they can be rebuilt; or (e) similar script code that can either install and invoke externally developed plug-in software components, or invoke external application (helper) components. Each may have its own distinct IP/security requirements.

Executable components—These components are in binary form, and the source code may not be open for access, review, modification, or possible redistribution (Rosen, 2005). If proprietary, they often cannot be redistributed, and so such components will be present in the design- and run-time architectures, but not in the distribution-time architecture.

Software services—An appropriate software service can replace a source code or executable component.

Application programming interfaces/APIs—Availability of externally visible and accessible APIs is the minimum requirement for an "open system" (Meyers & Oberndorf, 2001).

Software connectors—This includes software whose intended purpose is to provide a standard or reusable way of communication through common interfaces (e.g., High Level Architecture [Kuhl, Weatherly, & Dahmann, 1999], CORBA, MS .NET, Enterprise Java Beans, and GNU Lesser General Public License, LGPL, libraries). Connectors can also limit the propagation of IP license obligations or provide additional security capabilities.

Methods of connection—These include linking as part of a configured subsystem, dynamic linking, and client-server connections. Methods of connection affect license obligation propagation, with different methods affecting different licenses.

Configured system or subsystem architectures—These are software systems that are used as atomic components of a larger system, and whose internal architecture may comprise components with different licenses, affecting the overall system license and its security requirements. To minimize license interaction, a configured system or sub-architecture may be surrounded by what we term a *license firewall,* namely a layer of dynamic links, client-server connections, license shims, or other connectors that block the propagation of reciprocal obligations.

Figure 2 shows a high-level run-time view of a composed OA system whose reference architectural design in Figure 3 includes all the kinds of software elements listed above. This reference architecture has been instantiated in a build-time configuration in Figure 4, that in turn could be realized in alternative run-time configurations in Figures 5, 6, and 7 with different security capabilities. The configured systems consist of software components such as a Mozilla Web browser, Gnome Evolution email client, and AbiWord word processor (similar to MS Word), all running on a RedHat Fedora Linux operating system accessing file, print, and other remote networked servers such as an Apache Web server. Components are interconnected through a set of software connectors that bridge the interfaces of components and combine the provided functionality into the system's services. However, note how the run-time software architecture does not pre-determine how security capabilities will be assigned and distributed across different variants of the run-time composition.

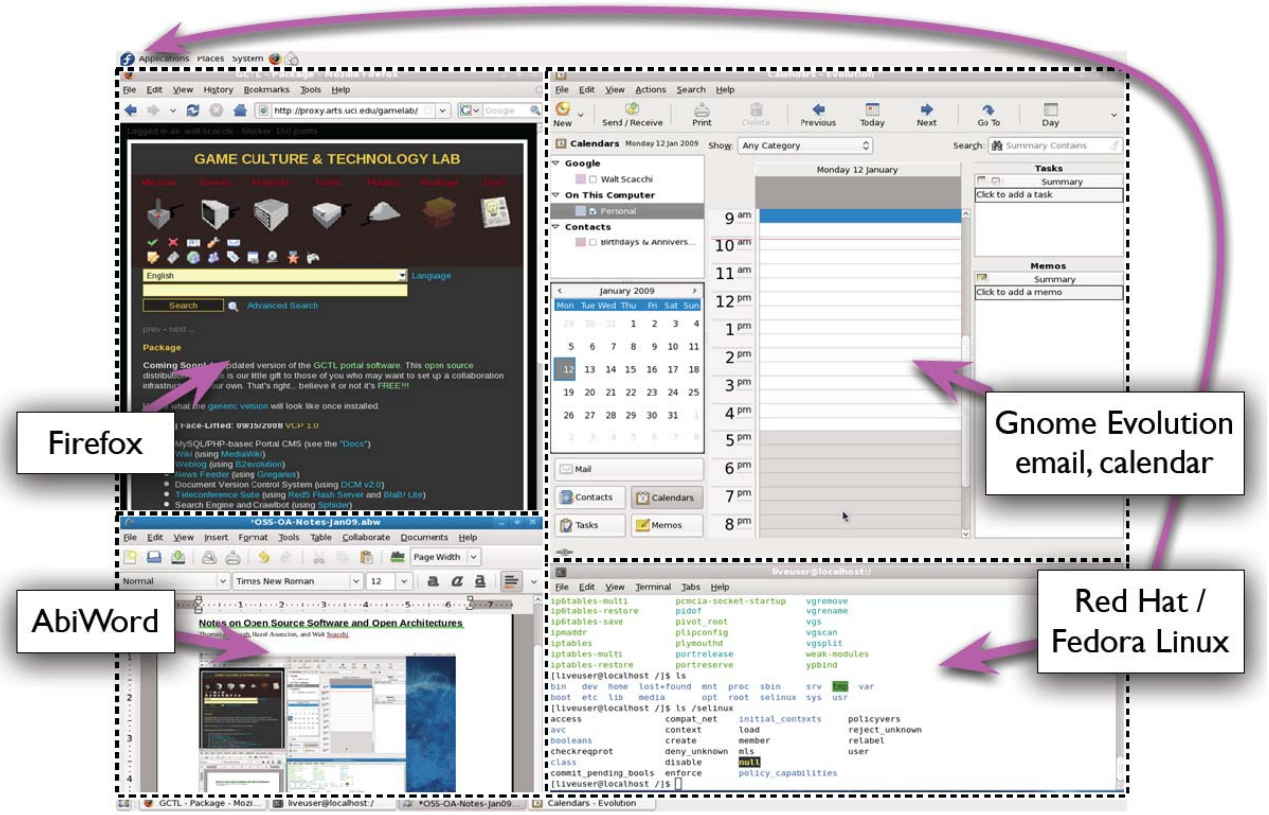

Figure 2. An Example Composite OA System Potentially Subject to Different IP and Security Licenses

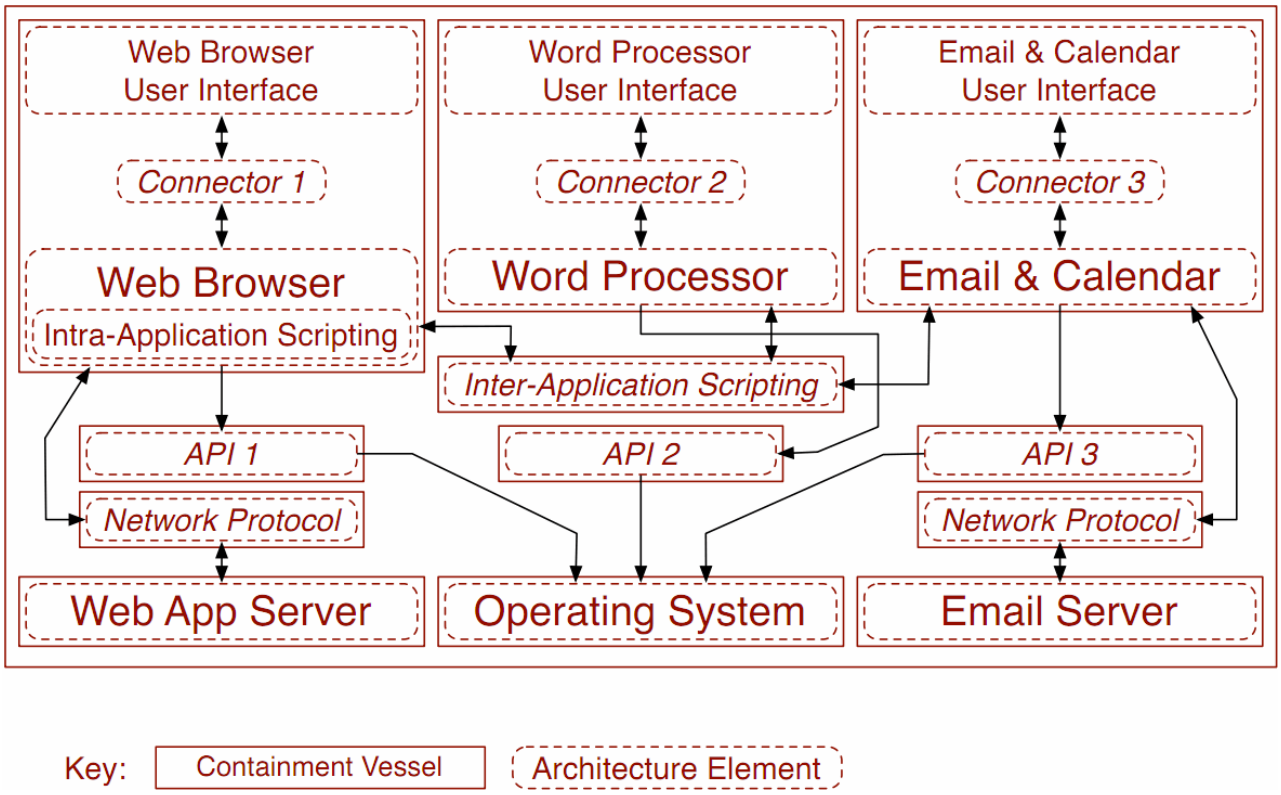

Figure 3. Design-Time Architecture of the System of Figure 2 That Specifies a Required Security Containment Vessel Scheme

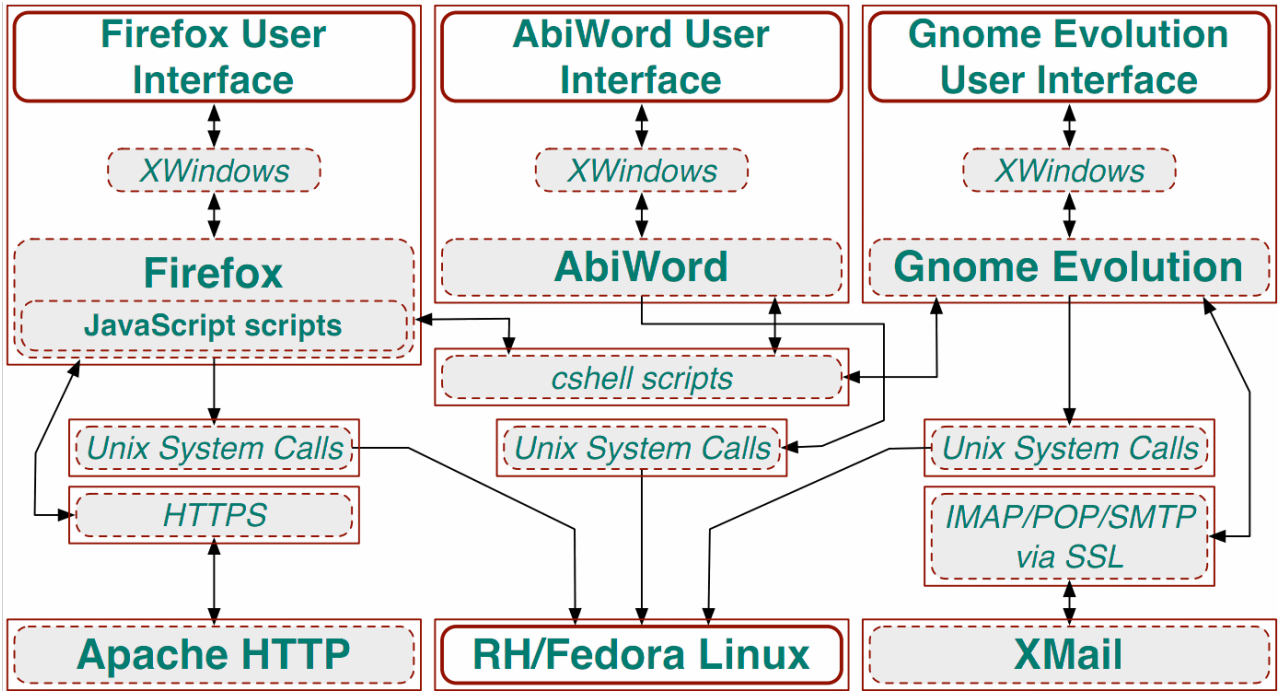

Figure 4. A Secure Build-Time Architecture Describing the Version Running in Figure 2 With a Specified Security Containment Vessel Scheme

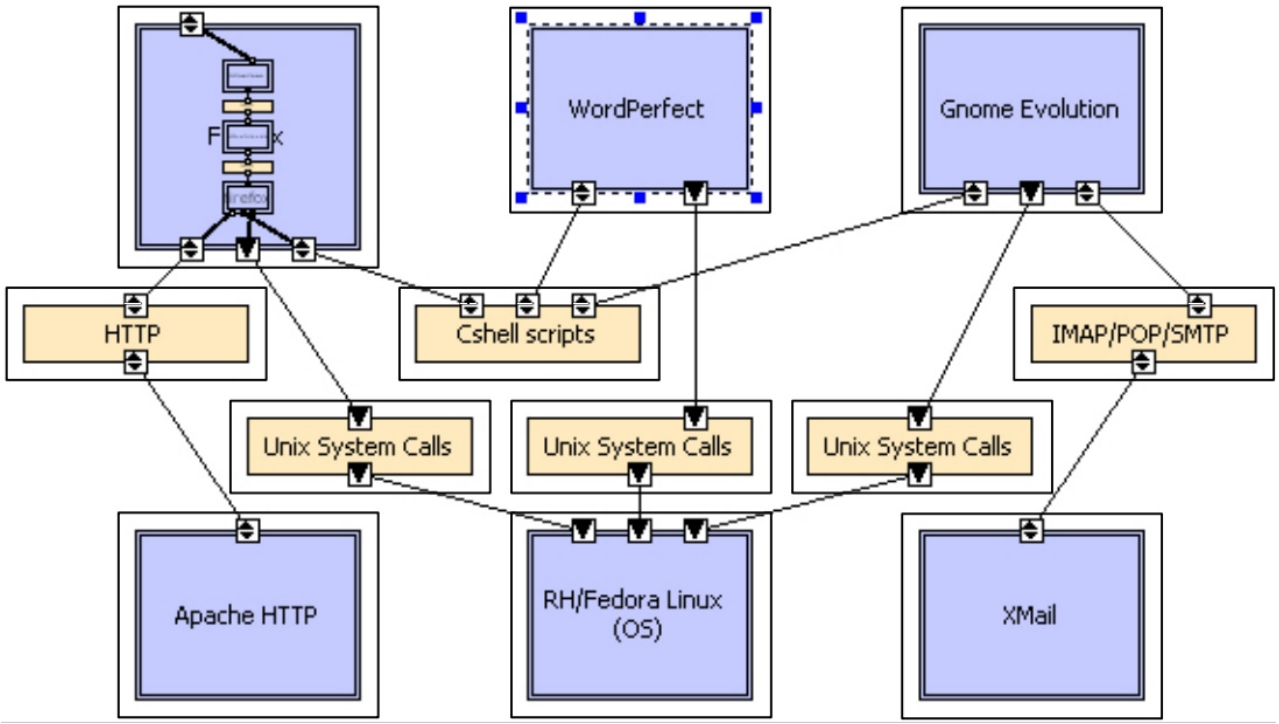

Figure 5. Instantiated Build-Time OA System With Maximum Security Architecture of Figure 4 via Individual Security Containment Vessels for Each System Element

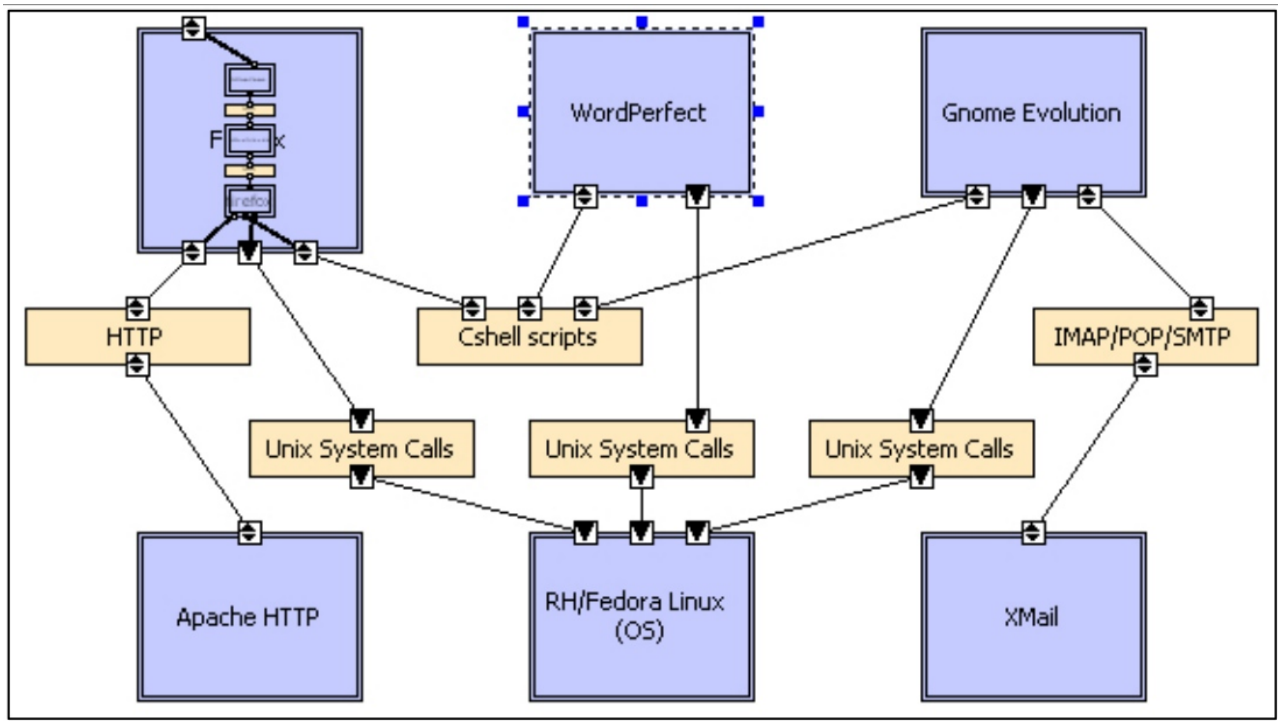

Figure 6. Instantiated Build-Time OA System With Minimum Security Architecture of Figure 4 via a Single Overall Security Containment Vessel for the Complete System Using a Common Software Hypervisor

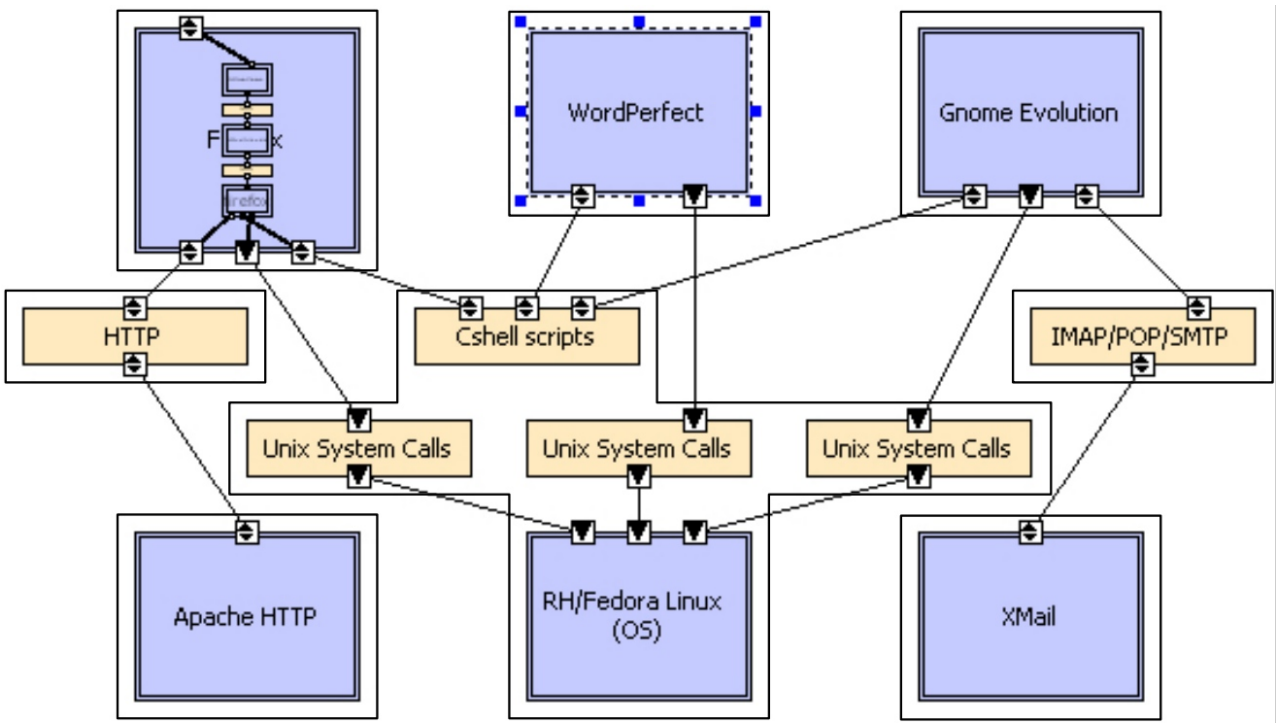

Figure 7. Instantiated Build-Time OA System With Mixed Security Architecture of Figure 4 via Security Containment Vessels for Some Groupings of System Elements

OA System Evolution

An OA system can evolve by a number of distinct mechanisms, some of which are common to all systems, but others of which are a result of heterogeneous IP and security licenses in a single system.

By component evolution—One or more components can evolve, altering the overall system's characteristics (for example, upgrading and replacing the Firefox Web browser from version 3.5 to 3.6, which may update existing software functionality, while also patching recent security vulnerabilities).

By component replacement— One or more components may be replaced by others with different behaviors but the same interface, or with a different interface and the addition of shim code to make it match (for example, replacing the AbiWord word processor with either Open Office or MS Word, depending on which is considered the least vulnerable to security attack).

By architecture evolution—The OA can evolve, using the same components but in a different configuration, altering the system's characteristics. For example, as discussed in the Secure Open Architecture Composition section, changing the configuration in which a component is connected can change how its IP or security license affects the rights and obligations for the overall system. This could arise when replacing email and word processing applications with web services like Google Mail and Google Docs, which we might assume may be more secure since the Google services (operating in a cloud environment) may not be easily accessed or penetrated by a security attack.

By component license evolution—The license under which a component is available may change, as, for example, when the license for the Mozilla core components was

changed from the Mozilla Public License (MPL) to the current Mozilla Disjunctive Tri-License; or the component may be made available under a new version of the same license, as, for example, when the GNU General Public License (GPL) version 3 was released. Similarly, the security license for a component may be changed by its producers, or the security license for a composed system changed by its integrators, in order to prevent or deter recently discovered security vulnerabilities or exploits before an evolutionary version update (or patch) can be made available.

By a change to the desired rights or acceptable obligations—The OA system's integrator or consumers may desire additional IP or security license rights (e.g., the right to sublicense in addition to the right to distribute), or no longer desire specific rights; or the set of license obligations they find acceptable may change. In either case, the OA system evolves, whether by changing components, evolving the architecture, or other means, to provide the desired rights within the scope of the acceptable obligations. For example, they may no longer be willing or able to provide the source code for components that have known vulnerabilities that have not been patched and eliminated.

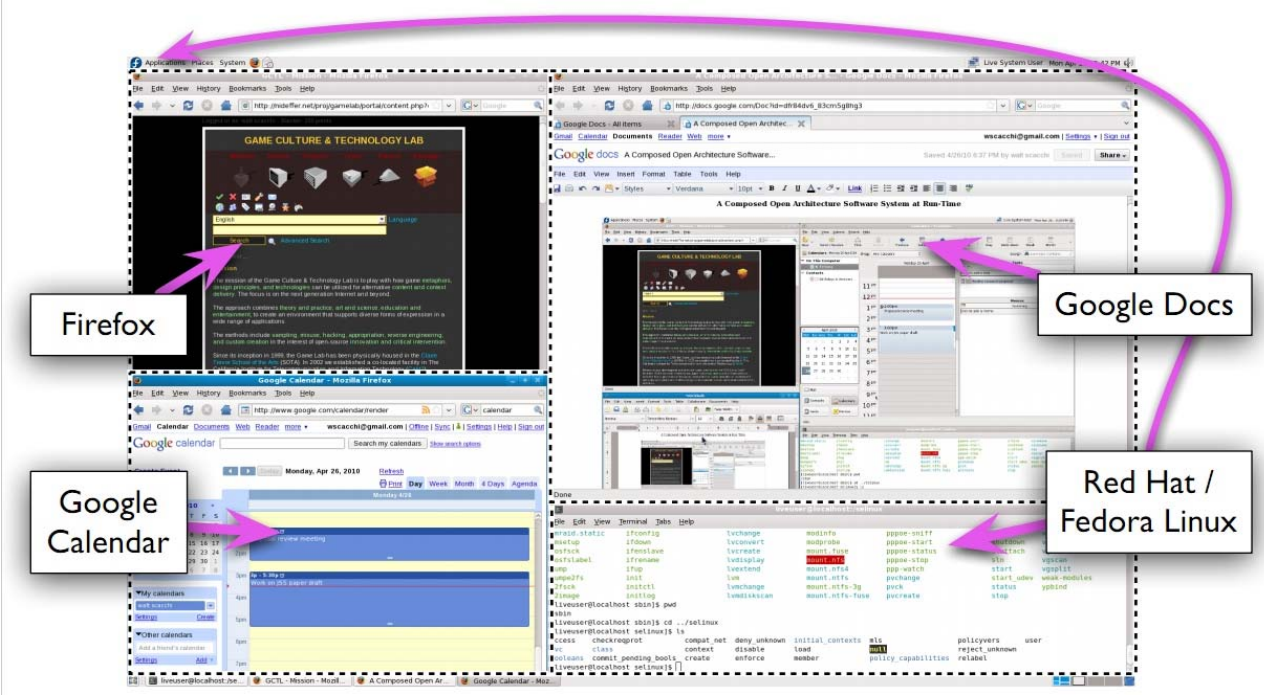

Figure 8. A Second Instantiation at Run-Time (Firefox, Google Docs and Calendar, Fedora) of the OA System in Figures 2, 3, and 4 as an Evolutionary Alternative System Version, Which in Turn Requires an Alternative Security Containment Scheme

The interdependence of integrators and producers results in a co-evolution of software within an OA ecosystem. Closely-coupled components from different producers must evolve in parallel in order for each to provide its services, as evolution in one will typically require a matching evolution in the other. Producers may manage their evolution with a loose coordination among releases, for example, as between the Gnome and Mozilla organizations. Each release of a producer component creates a tension through the ecosystem relationships with consumers and their releases of OA systems using those components, as integrators accommodate the choices of available, supported components with their own goals and needs. As discussed in our previous work (Alspaugh, Asuncion, &

Scacchi, 2009a), license rights and obligations are manifested at each component's interface, then mediated through the system's OA to entail the rights and corresponding obligations for the system as a whole. As a result, integrators must frequently re-evaluate an OA system's IP/security rights and obligations. In contrast to homogeneously-licensed systems, license change across versions is a characteristic of OA ecosystems, and architects of OA systems require tool support for managing the ongoing licensing changes.

We propose that such support must have several characteristics:

- It must rest on a license structure of rights and obligations (see the Security Licenses section), focusing on obligations that are enactable and testable.
- It must take account of the distinctions between the design-time, build-time, and distribution-time architectures (see the sections entitled Secure Open Architecture Composition, Security Licenses, and Security License Architectures), and the rights and obligations that come into play for each of them.
- It must distinguish the architectural constructs significant for software licenses, and embody their effects on rights and obligations (see the Secure Open Architecture Composition section).
- It must define license architectures (see the Security License Architectures section).
- It must provide an automated environment for creating and managing license architectures. We are developing a prototype that manages a license architecture as a view of its system architecture (Alspaugh, Asuncion, & Scacchi, 2009a).
- Finally, it must automate calculations on system rights and obligations, so that they may be done easily and frequently, whenever any of the factors affecting rights and obligations may have changed (see the Security License Analysis section).

Security Licenses

Licenses typically impose obligations that must be met in order for the licensee to realize the assigned rights. Common IP/copyright license obligations include the obligation to publish at no cost any source code you modify (MPL), or the reciprocal obligation to publish all source code included at build-time or statically linked (GPL). The obligations may conflict, as when a GPL'd component's reciprocal obligation to publish source code of other components is combined with a proprietary component's license prohibition of publishing its source code. In this case, no rights may be available for the system as a whole, not even the right of use, because the two obligations cannot simultaneously be met, and thus neither component can be used as part of the system. Security capabilities can similarly be expressed and bound to the data values and control signals that are visible in component interfaces, or through component connectors.

Some typical security rights and obligations might be the following:

- the right to read data in containment vessel T;
- the obligation for a specific component to have been vetted for the capability to read and update data in containment vessel T;
- the obligation for a user to verify his/her authority to see containment vessel T, by password or other specified authentication process;
- the right to replace specified component C with some other component;

- the right to add or update specified component D in a specified configuration; and
- the right to add, update, or remove a security mechanism.

The basic relationship between software IP/security license rights and obligations can be summarized as follows: if the specified obligations are met, then the corresponding rights are granted. For example, if you publish your modified source code and sub-licensed derived works under MPL, then you get all the MPL rights for both the original and the modified code. Similarly, software security requirements are specified as security obligations that, when met, allow designated users or other software programs to access, modify, and redistribute data and control information to designated repositories or remote services. However, license details are complex, subtle, and difficult to comprehend and track—it is easy to become confused or make mistakes. The challenge is multiplied when dealing with configured system architectures that compose a large number of components with heterogeneous IP/security licenses, so that the need for legal counsel begins to seem inevitable (Rosen, 2005; Fontana et al., 2008).

We have developed an approach for expressing software licenses of different types (intellectual property and security requirements) that is more formal and less ambiguous than natural language, and that allows us to calculate and identify conflicts arising from the rights and obligations of two or more component's licenses. Our approach is based on Hohfeld's (1913) classic group of eight fundamental jural relations, of which we use right, duty, no-right, and privilege. We start with a tuple <actor, operation, action, object> for expressing a right or obligation. The actor is the "licensee" for all the licenses we have examined. The operation is one of the following: "may," "must," "must not," or "need not," with "may" and "need not" expressing rights, and "must" and "must not" expressing obligations. The action is a verb or verb phrase describing what may, must, must not, or need not be done, with the object completing the description. A license may be expressed as a set of rights, with each right associated with zero or more obligations that must be fulfilled in order to enjoy that right. Figure 9 shows the meta-model with which we express licenses.

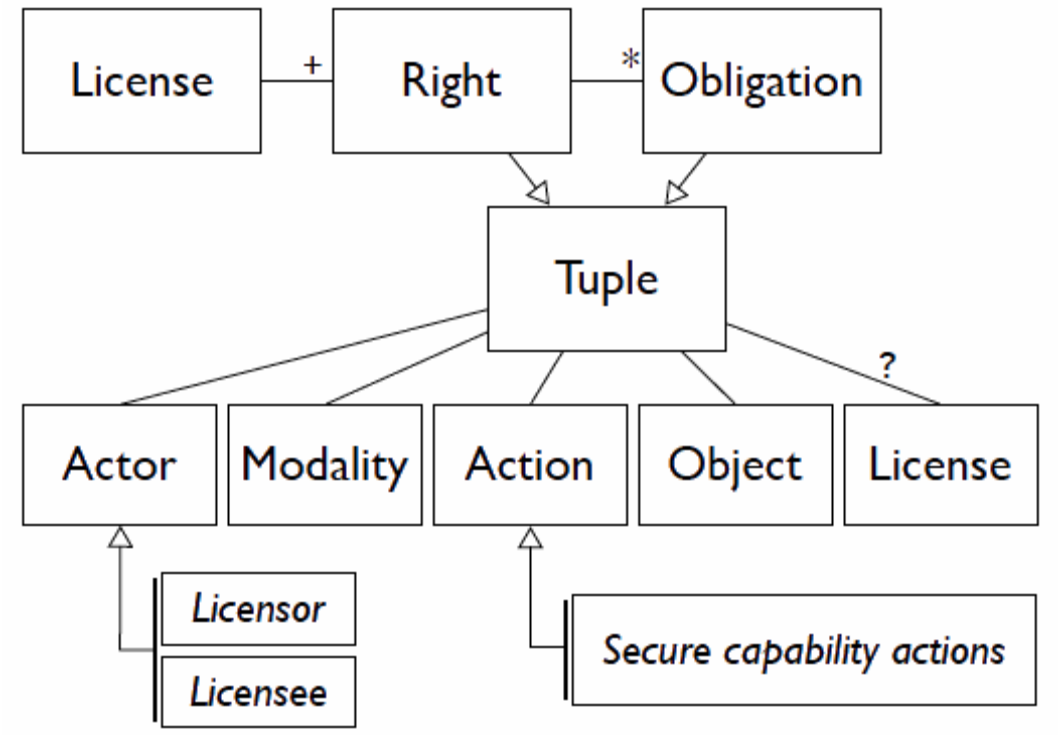

Figure 9. Security License Meta-Model

Designers of secure systems have developed a number heuristics to guide architectural design in order to satisfy overall system security requirements, while avoiding conflicts among interacting security mechanisms or defenses. However, even using design heuristics (and there are many), keeping track of security rights and obligations across components that are interconnected in complex OAs quickly becomes too cumbersome. Automated support is needed to manage the complexity of multi-component system compositions where different security requirements must be addressed through different security capabilities.

Security License Architectures

Our security license model forms a basis for effective reasoning about licenses in the context of actual systems, and calculating the resulting rights and obligations. In order to do so, we need a certain amount of information about the system's configuration at design-time, build-time, and run-time deployment. The following needed information comprises the license architecture, an abstraction of the system architecture:

1. the set of components of the system (for example, see Figure 2) for the current system configuration, as well as subsequently for system evolution update versions (as seen in Figure 8);
2. the relation mapping each component to its security requirements (specified and analyzed at design-time, as exemplified in Figure 3) or capabilities (specified and analyzed at build-time in Figure 4, and run-time across alternatives shown in Figures 5, 6, and 7);
3. the connections between components and the security requirements or capabilities of each connector passing data or control signals to/from it; and

4. possibly other information, such as information to detect or prevent IP and security requirements conflicts, which is as yet undetermined.

With this information and definitions of the licenses involved, we believe it is possible to automatically calculate rights and obligations for individual components or for the entire system, as well as guide/assess system design and evolution, using an automated environment of the kind that we have previously demonstrated (Alspaugh, 2009a, 2009b, 2010; Alspaugh, Scacchi, & Asuncion, 2010).

Security License Analysis

Given a specification of a software system's architecture, we can associate security license attributes with the system's components, connectors, and sub-system architectures, resulting in a license architecture for the system, and calculate the security rights and obligations for the system's configuration. Due to the complexity of license architecture analysis, and the need to re-analyze every time a component evolves, a component's security license changes, a component is substituted, or the system architecture changes, OA integrators really need an automated license architecture analysis environment. We have developed a prototype of such an environment for analogous calculations for software copyright licenses (Alspaugh, Asuncion, & Scacchi, 2009b; Alspaugh, Scacchi, & Asuncion, 2010), and are extending this approach to security licenses.

Security Obligation Conflicts

A security obligation can conflict with another obligation, a related right for the same or nearby components, or with the set of available security rights, by requiring a right that has not been granted. For instance, consider the following two connected components C and D with security obligations:

> (O1) The obligation for component C to have been vetted for the capability to read and update data in containment vessel T
>
> (O2) The obligation for all components connected to specified component D to grant it the capability to read and update data in containment vessel T

If C has not been vetted, then these two obligations conflict. This possible conflict must be taken into consideration in different ways at different development times, as follows:

- at design time, ensuring that it will be possible to vet C;
- at build time, ensuring that the specific implementation of C has been vetted successfully; and
- possibly at run time as well, confirming that C is certified to have been vetted, or (if C is dynamically connected at run time) vetting C before trusting this connection to it.

The second obligation may also conflict with the set of available security rights, for example if D is connected to component E for which the security right (R1) The right to read and update data in containment vessel T using component E is not available.

The absence of such conflicts does not mean, of course, that the system is secure. But the presence of conflicts reliably indicates that it is not secure.

Rights and Obligations Calculations

The rights available for the entire system (the right to read and update data in containment vessel T, the right to replace components with other components, the right to update component security licenses, etc.) are calculated as the intersection of the sets of security rights available for each component of the system. If a conflict is found involving the obligations and rights of interacting components, it is possible for the system architect to consider an alternative scheme, for example, using one or more connectors along the paths between the components that act as a security firewall. This means that the architecture and the automated environment together can determine what OA design best meets the problem at hand with available software components. Components with conflicting security licenses do not need to be arbitrarily excluded, but instead may expand the range of possible architectural alternatives if the architect seeks such flexibility and choice.

Discussion

Our approach to specifying and analyzing the security requirements for a complex OA system is based on the use of a security license. As noted, a security license is a new kind of information structure whose purpose is to declare operational capabilities that express the obligations and rights of users or program to access, manipulate, control, update, or evolve data, control signals, and accessible software system elements. Our proposed security license is influenced by IP licenses that are employed to specify property control and declared copyright freedoms/restrictions, such as those for OSS components subject to licenses like the GPLv2, MPL, LGPL, or others. These IP licenses as information structures often pre-exist to facilitate their widespread use, dissemination, and common interpretation. Further, the choice of which IP license to choose or assign to a software component results from a trade-off analysis typically performed by the components producers, rather than the system integrators or consumers, as a way to protect or propagate the obligations and rights to use, evolve, and redistribute the updated component's open source code.

The security licenses we propose do not necessarily exist prior to their specification and assignment to a given OA system. Similarly, we do not yet have a basis to anticipate or expect that generic security licenses will emerge, as they have for OSS components. However, one follow-on goal we seek to address is whether and how best to specify security licenses for different types of software elements or components, so that it becomes possible to semi-automatically specify the security license for a given component or composed OA system through the reuse and instantiation of security requirement templates. This idea is somewhat similar to the license templates and taxonomy that is employed by the Creative Commons for non-software intellectual property like online art or new media content (cf., http://creativecommons.org/licenses/). In this regard, it may be possible to develop a technique and supporting computational environment whereby system integrators or consumers can conveniently specify the security requirements they seek (e.g., fill out online security requirements forms), while the environment interprets these specifications to generate operational security capabilities that can guard the entry and exit of data or control information from the appropriate containment vessel that encapsulates the corresponding system element. Consequently, this is a topic for further study and investigation.

Next, one might wonder why it is not simply desirable to have maximum system security under all circumstances. When considering the alternative run-time system composition variants shown in Figures 5, 6, and 7, it appears there may be trade-offs in one layout of security capabilities over another. For example, the layout in Figure 5 maximizes security by encapsulating each system element within its own containment vessel. This in

turn requires a VM technology of a kind different from that commonly available (e.g., like VMware), and instead requires a new lightweight VM technology that can provide security capabilities (e.g., create, read, update authorizations) for potentially small-scale software elements (e.g., Cshell scripts). Similarly, the different security containment layouts may affect system performance, ease of evolutionary update, and associated level of security administration. But these again all represent trade-offs in the desire to achieve affordable, practical, and ever more robust and testable secure software component/system capabilities at build-time and run-time. Thus, we take the position that it is better to provide the ability to specify and analyze the security requirements of different software elements at design-time, as well as specify and analyze the security capabilities at build-time and run-time, rather than the current practice that does not account for system architecture or license architecture, and is thus inherently vulnerable to attacks that can otherwise be prevented or detected.

One other topic that follows from our approach to semantically modeling and analyzing OA systems that are subject to software security licenses. More specifically, how our approach and emerging results might shed light on software systems whose architectures articulate a software product line.

Accordingly, organizing and developing software product lines (SPLs) relies on the development and use of explicit software architectures (Bosch, 2000; Clements & Northrop, 2001). However, the architecture of a secure SPL is not necessarily a secure OA—there is no requirement for it to be so. Thus, we are interested in discussing what happens when SPLs may conform to a secure OA, and to an OA that may be composed from secure SPL components. Three considerations come to mind.

First, if the SPL is subject to a single homogeneous security software license, which may often be the case when a single vendor or government contractor has developed the SPL, then the security license may act to reinforce a vendor lock-in situation with its customers. One of the motivating factors for OA is the desire to avoid such lock-in, whether or not the SPL components have open or standards-compliant APIs.

Second, if an OA system employs a reference architecture much like we have in the design-time architecture depicted in Figure 3, which is then instantiated into a specific software product configuration, as suggested in the build-time architecture shown in Figure 4, then such a reference or design-time architecture, as we have presented it here, effectively defines an SPL consisting of possible different system instantiations composed from similar components instances (e.g., different but equivalent Web browsers, word processors, email, calendaring applications, relational database management systems).

Third, if the SPL is based on an OA that integrates software components from multiple vendors or OSS components that are subject to heterogeneous security licenses (i.e., those that may possibly conflict with one another), then we have the situation analogous to what we have presented in this paper. So secure SPL concepts are compatible with secure OA systems that are composed from heterogeneously security licensed components.

Conclusion

This paper introduces the concept and initial scheme for systematically specifying and analyzing the security requirements for complex open architecture systems. We argue that such requirements should be expressed as operational capabilities that can be collected and sequenced within a new information structure we call a security license. Such a license expresses security in terms of capabilities that provide users or programs

obligations and rights for how they may access data or control information, as well as how they may update or evolve system elements. These security license rights and obligations thus play a key role in how and why an OA system evolves in its ecosystem of software component producers, system integrators, and consumers.

We note that changes to the license obligations and rights, whether for control of intellectual property or software security, across versions of components, is a characteristic of OA systems whose components are subject to different security requirements or other license restrictions. A structure for modeling software licenses and automated support for calculating its rights and obligations are needed in order to manage an OA system's evolution in the context of its ecosystem.

We have outlined an approach for achieving these and sketched how they further the goal of reusing components in developing software-intensive systems. Much more work remains to be done, but we believe this approach turns a vexing problem into one for which workable, as well as robust, formal solutions can be obtained.

References

Alspaugh, T. A., & Anton, A. I. (2008, February). Scenario support for effective requirements. *Information and Software Technology*, *50*(3), 198–220.

Alspaugh, T. A., Asuncion, H. U., & Scacchi, W. (2009a, May). Analyzing software licenses in open architecture software systems. In *2nd International Workshop on Emerging Trends in FLOSS Research and Development* (FLOSS).

Alspaugh, T. A., Asuncion, H. U., & Scacchi, W. (2009b, August 31–September 4). Intellectual property rights requirements for heterogeneously-licensed systems. In *17th IEEE International Requirements Engineering Conference* (RE'09; pp. 24–33).

Alspaugh, T. A., Asuncion, H. U., & Scacchi, W. (2010, May). The challenge of heterogeneously licensed systems in open architecture software ecosystems. In *Proceedings of the Seventh Acquisition Research Symposium*. Monterey, CA: Naval Postgraduate School.

Alspaugh, T. A., Scacchi, W., & Asuncion, H. U. (2010, November). Software licenses in context: The challenge of heterogeneously licensed systems. *Journal of the Association for Information Systems*, *11*(11), 730–755.

Bass, L., Clements, P., & Kazman, R. (2003). *Software architecture in practice*. Boston, MA: Addison–Wesley Longman.

Bosch, J. (2000). *Design and use of software architectures: Adopting and evolving a product-line approach*. Boston, MA: Addison–Wesley Professional.

Breaux, T. D., & Anton, A. I. (2005). Analyzing goal semantics for rights, permissions, and obligations. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering* (RE '05; pp. 177–188).

Breaux, T. D., & Anton, A. I. (2008). Analyzing regulatory rules for privacy and security requirements. *IEEE Transactions on Software Engineering*, *34*(1), 5–20.

Clements, P., & Northrop, L. (2001). *Software product lines: Practices and patterns*. Boston, MA: Addison–Wesley Professional.

Falliere, N., Murchu, L. O., & Chien, E. (2011, February). *W32.Stuxnet dossier, Version 1.4.* Retrieved from

http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf

Feldt, K. (2007). *Programming Firefox: Building rich internet applications with XUL*. O'Reilly Media.

Firesmith, D. (2004, January–February). Specifying reusable security requirements. *Journal of Object Technology*, *3*(1), 61– 75.

Fontana, R., Kuhn, B. M., Moglen, E., Norwood, M., Ravicher, D. B., Sandler, K., Vasile, J., & Williamson, A. (2008). *A legal issues primer for open source and free software projects*. Software Freedom Law Center.

German, D. M., & Hassan, A. E. (2009, May). License integration patterns: Dealing with licenses mis-matches in component-based development. In *28th International Conference on Software Engineering* (ICSE '09).

Hohfeld, W. N. (1913, November). Some fundamental legal conceptions as applied in judicial reasoning. *Yale Law Journal*, *23*(1), 16–59.

Kuhl, F., Weatherly, R., & Dahmann, J. (1999). *Creating computer simulation systems: An introduction to the high level architecture*. Prentice Hall.

Meyers, B. C., & Oberndorf, P. (2001). *Managing software acquisition: Open systems and COTS products*. Boston, MA: Addison–Wesley Professional.

Nelson, L., & Churchill, E. F. (2006). Repurposing: Techniques for reuse and integration of interactive systems. In *International Conference on Information Reuse and Integration* (IRI-08; p. 490).

Oreizy, P. (2000). *Open architecture software: A flexible approach to decentralized software evolution* (Doctoral dissertation), University of California, Irvine.

Rosen, L. (2005). *Open source licensing: Software freedom and intellectual property law*. Prentice Hall.

Scacchi, W., & Alspaugh, T. A. (2008, May). Emerging issues in the acquisition of open source software within the U.S. Department of Defense. In *Proceedings of the Fifth Annual Acquisition Research Symposium*. Monterey, CA: Naval Postgraduate School.

Yau, S. S., & Chen, Z. (2006). A framework for specifying and managing security requirements in collaborative systems. In *Third International Conference on Autonomic and Trusted Computing* (ATC 2006; pp. 500–510).

Acknowledgments

This research is supported by grant #N00244-10-1-0038 and #N00244-10-1-077 from the Acquisition Research Program at the Naval Postgraduate School, and by grant #0808783 from the U.S. National Science Foundation. No review, approval, or endorsement implied.