Addressing Challenges in the Acquisition of Secure Software Systems with Open Architectures

Walt Scacchi and Thomas Alspaugh
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3455 USA
wscacchi@ics.uci.edu, thomas.alspaugh@acm.org
949-824-4130 (v), 949-824-1715 (f)

Abstract

We seek to articulate and address a number of emerging challenges in continuously assuring the security of open architecture (OA) software systems throughout the system acquisition life-cycle. It is now clear that future system must resist coordinated international attacks on vulnerable software-intensive systems that are of high value and control complex systems. But current approaches to system security are most often piece-meal with little/no support for guiding the what system security requirements must address across different system processing elements and data levels, and how those can be manifest during the design, building, and deployment of OA software systems. We present a framework that organizes OA system security elements and mechanisms in forms that can be aligned with different stages of acquisition spanning system design, building, and run-time deployment, as well as system evolution. We provide a case study to show our scheme and how it can be applied to common enterprise systems.

Biographies

Walt Scacchi is a senior research scientist and research faculty member at the Institute for Software Research, University of California, Irvine. He received a Ph.D. 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 60 externally funded research projects. Last, in 2012, he serves as General Co-Chair of the 8th. IFIP International Conference on Open Source Systems (OSS2012).

Thomas Alspaugh is a project scientist at the Institute for Software Research, University of California, Irvine. His research interests are in software engineering, requirements, and licensing. Before completing his Ph.D., 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 or A-7 project.

Introduction

We seek to research, develop, and refine new concepts, techniques, and tools for continuously assuring the security of large-scale, open architecture (OA) software systems composed from software components that include proprietary/closed source software (CSS) and open source software (OSS). Federal government acquisition policy, as well as many leading enterprise IT centers, now encourage the use of CSS and OSS, and thus OA, in the development, deployment and evolution of complex, software-intensive systems.

We seek to prototype and demonstrate a new innovative approach and supporting technology that can develop new principles for correctness and security properties for OA systems. This includes developing basic principles to determine the security and performance properties of software systems, the conditions under which these properties hold, and the methods used to prove these properties of interest for systems. Of particular interest are networked OA software systems, that are adapted or evolve to dynamic conditions and threats during their development, deployment, and usage, including those that may rely on new technologies like OA mobile devices [Sm12, STIG11] or other IT systems relying on open source technologies [DoD10, Ga10, Gi11, Navy10]. In particular, such study may be of value to securing new cyber warfare technologies [DoD11, SBN11]. Our efforts may also lead to fundamental advancements for secure information sharing between information producers and consumers, in order to realize more secure information management, sharing and interaction.

Challenges of Securing Systems with Open Architectures

Coordinated international attacks on vulnerable software-intensive systems that are of high value and control complex systems are becoming ever more apparent. As the *StuxNet* case demonstrates, security threats to software systems are multi-valent, multi-modal, and distributed across independently developed software system components [Stux11]. Similarly, it is now clear that physically isolated/confined systems are vulnerable to external security attacks, via portable storage devices like USB drives, modified end-user devices (e.g., keyboards, mice [H11]), and social engineering techniques [Saw11]. This requires new security measures and policies necessary to defend such systems through new threat prevention and detection methods, as well as appropriate response mechanisms. Thus, what makes a system or system architecture secure changes over time, as new threats emerge and as systems evolve to meet new functional requirements. Consequently, there is need for an approach to continuously assure the security of complex, evolving OA systems in ways that are practical and scalable, yet robust, tractable, and adaptable.

However, the best practices for developing OA systems whose components may be subject to differing security requirements (i.e., security rights and obligations) are unclear. Such practices are yet to be identified. This puts IT centers, system integrators, and service providers at a disadvantage when seeking to develop new software-intensive systems whose costs may be lower due to the integration of mature OSS components that are interfaced to pre-existing or new CSS components. OA systems thus present new challenges for assuring software system security.

Software systems security mechanisms for enabling security requirements or policies are often employed on an *ad hoc* basis, since there are not convenient or interactive tools, nor formal techniques for specifying the security requirements of an OA system, or its components. Instead, what is available are disjoint mechanisms for implementing individual system security features [LSM98, SSL99], such as:

- mandatory access control lists, firewalls;

- multi-level security;
- authentication (including certificate authority and passwords);
- cryptographic support (including public key certificates);
- encapsulation (including virtualization, hidden vs. public APIs), hardware confinement (memory, storage, and external device (port) isolation) [SWZ12], and type enforcement capabilities;
- secure programming practices (including secure coding standards, data type and value range checking) [Se08];
- data content or control signal flow logging/auditing;
- honey-pots and traps;
- security technical information guides for configuring the security parameters for applications [STIG11] and operating systems [Sm12];
- functionally equivalent but diverse multi-variant software executables [Fr10, SJW11].

But there is a gap between these mechanisms and any concept of a comprehensive security policy, whether for a system or any of its components, and no obvious way to integrate and evaluate them as a group. Similarly, it is unclear what relationships arise or are in place among these different security mechanisms. Further, what guidance is needed regarding which security mechanism to use where, when, why and how, and how to update their usage or configuration as extant system security policy evolves. The mechanisms are also mostly software implementation choices rather than system architectural choices; no system-specific framework (like an architecture) exists in which they can be pulled together in patterns that can be designed to meet specific security policies and goals. But in an OA system, it may be unclear or unlikely that system integrators will find mature OSS or CSS components that supply all of the system security features that the integrator or the customer requires on a timely, cost-effective basis.

Next, OA systems evolve through more pathways than traditional systems:

- individual components evolve through update revisions (e.g., security patches) made by the component's developers;
- individual components are updated with new, functionally enhanced versions from outside providers;
- individual components are replaced by different components from other sources;
- component interfaces evolve, either due to the system developers or outside sources;
- system architecture and configuration evolve as the developers adapt it to address new functional requirements; and
- system functional and security requirements evolve, either due to the system developers, recognized gaps, or outside stakeholders.
- system security policies, mechanisms, security components, and system configuration parameter settings also change over time.

These additional evolution paths are tied to the benefits of using OA systems with OSS components but they also present new challenges for security. OA systems are continually evolving, and in our view this fact is fundamentally unaddressed by prior work in security.

Beyond these issues, we must consider how should customers specify what security system features they want their delivered systems to support? How can the history of security failures (vulnerabilities), faults (exploits), possible cyber-warfare attacks (threats), and possible responses (updating system configuration with new elements that resist new threats, close new vulnerability, and prevent newly discovered exploits), to guide the evolution of approaches for developing secure OA systems? How can

answers to questions like these help formulate a technological innovation element of the DoD strategy for operating in cyberspace [DoD11]? Questions like this remain unresolved at present.

Verification of the usage of security mechanisms in software systems is unclear, and often focused either at the whole system (macro) level, or program function or coding (micro) level, but generally not at the architectural component and interconnection (meso) level, and not for combinations and alternative configurations of CSS and OSS components with different security histories. We believe there is an new or under-explored opportunity to address security requirements at the architectural level.

As such, we see the following basic challenges in assuring OA system security:

- How to verify the security of OA system designs throughout system development, deployment, and post-deployment support.
- How to validate the effectiveness of OA system security measures, and feed back evolving knowledge of vulnerabilities and exploits into the ongoing development (continuous evolution) stream for existing and planned systems in an operational, testable form that system designers can use, and program managers can assess.

Similarly, we see the following basic challenges in assuring security of OA software systems:

- How best to develop complex OA systems whose OSS or CSS system components may originally come from trusted sources, but in which these components, the architectural configuration, and security requirements are subject to multiple sources of adaptation and evolution.
- How to go beyond “many eyes” (large number of skilled reviewers) to establish a scalable basis for automated or semi-automated verification of software system security properties as the system continually evolves.
- How to best achieve continuous software system security assurance as a system is adapted and evolved to address new security requirements and technology progress.
- How best to protect OA systems through biologically inspired natural defenses that provide adaptive and resilient mechanisms including agile response, isolation, and fail-soft recovery to immediate attacks, as well as adaptation via dynamic reconfiguration, multi-version mechanisms, (artificial) ecological diversity responses to sustained vulnerabilities or threats [Sh11].
- How to create reference models and security policy requirements that articulate security scenarios appropriate for oversight during system acquisition, as well as during system design, implementation, deployment, and beyond?

Securing Software Systems

The key ideas in our approach to develop and demonstrate a new solution to the challenges is to specify verifiable security requirements of OA systems using formalized “security licenses” [SA11], and to use an explicit, evolvable software architecture to mediate and carry the paths of interactions among them. Security licenses must specify the security requirements and access/update rights and obligations within an OA system, its CSS and OSS components, and their interconnections (e.g., APIs, databases, shared files, communication protocols) that defend against threats and enable appropriate responses to attacks or suspicious/anomalous system behaviors. Subsequently, the goal of our approach is to articulate and refine the ways and means for expressing and verifying that the security requirements of OA system components match up appropriately and together support the security requirements of the entire OA system, at architectural design time, while enabling the automated verification of system builds/compositions and deployable, as well as of executable run-time versions of the system.

Software licenses represent a collection of rights and obligations for what can or cannot be done with a licensed software component. Licenses can thus denote both functional and non-functional requirements that apply to a software systems or system components 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 functional or 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. This kind of approach provides new principles of correctness for software IP requirements [cf. BA05, BA08].

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 [YC06]. Using a semantic model and logic to formally specify the rights and obligations required for a software system or component to be secure [BA05, BA08, YC06] 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. We have already demonstrated how such an approach can work, when limiting attention to IP rights and obligations.

The approach we have been developing for the past few years for modeling and analyzing software system IP license architectures for OA systems [AAS09, ASA10, SA08], may therefore be extendable to also address OA systems with heterogeneous software security license rights and obligations [SA11]. Furthermore, the idea of common or reusable software security licenses may be analogous to the reusable security requirements templates proposed by Firesmith [F04] at the Software Engineering Institute. Such security requirement templates may simplify and guide the efforts of customers (or contracting officers) to more readily specify workable requirements that can be readily verified through system development, deployment, and post deployment support.

Security licenses can be specified, modeled, and analyzed continuously from initial system architectural design through post deployment support and system evolution, with key points for security license analysis occurring at design-time, build/linking time, and deployment/run-time. Such security licenses can be stated both (a) informally, using restricted natural language for human readability, authorship, description of non-functional security requirements, as well as (b) formally, specifying functional security requirements in a computer processable form using a logic-based scheme and modeling notation, with automated production of (a) from (b) and automated architecture-mediated inferences using (b). Analysis of a system/s security requirements can therefore be integrated into the software architecture tool used to express and evolve the architecture, so that the analysis evolves automatically in parallel with the architecture.

In general terms, a security license is analogous to a software copyright license such as GPL (GNU General Public License) [GPL07]. Software licenses consist of intellectual property (IP) *rights* granted by the license, and corresponding license *obligations* needed to obtain the rights. Our innovation is to similarly specify the security obligations and rights of OA system components using elements found in known security capabilities, which we can then model, analyze, and support throughout the system’s development and evolution, and use to guide system design and instantiation. Our initial investigation of security licenses [SA11] has identified rights and obligations such as:

- The obligation for a user to verify his/her authority to see compartment T, by password or other specified authentication process
- The obligation for a specific component to have been vetted for the capability to read and update data in compartment T
- The obligation for all components connected to specified component C to grant it the capability to read and update data in compartment T
- The obligation to reconfigure a system in response to detected threats, when given the right to select and include different component versions, or executable component variants.
- The right to read and update data in compartment T using the licensed component
- The right to replace specified component C with some other component
- The right to add or update specified component D in a specified configuration
- The right to add, update, or remove a security mechanism
- The right to update security license L.

Further, formally specified OA security licenses are verifiable, as well as grounded in functional and testable system security capabilities.

The security reasoning chains among the security licenses are mediated by the system architecture, and evolve automatically with it, much like they can for IP licenses [AAS09, AAS11, ASA10]. Each kind of security license details how its obligations are propagated architecturally to other system components. The results of this propagation, coupled with automated identification of gaps, conflicts, and subsumptions, are communicated to analysts as architecturally-organized arguments supporting the existence of the identified issues. The arguments provide context-appropriate guidance, in terms of the system architecture and the security licenses of the components involved, for resolution of security problems through the evolution of the system design.

Our approach neither assumes nor proves that individual elements of an OA system are secure, but instead seeks to determine what security rights and obligations are in effect at any time for the overall system architecture as a function of the security rights and obligations of its components. This means that it is possible to configure a secure OA system whose components may be insecure, or not equally secure. Our approach also supports determination of where or how OA system security rights or obligations may be in conflict, mismatch, or subsume one another as individual system components or connectors are adapted to evolve over time. As an organization's security policies (i.e., their security requirements) evolve and adapt, the OA system’s security rights and obligations are evolved to match and satisfy them, as long as all security requirements can be expressed through description logic relationships among them.

Security rights and obligations are characterized in terms of enterprise security policies and goals; within that closed world our approach enables specification of the security properties that an open system architecture must match or satisfy. These security requirements also direct acquisition program managers and architecture analysts attention to problem areas with greatest impact on system security. Where our approach identifies a conflict or mismatch, it indicates an actual, open-world weakness in the security of the OA system under analysis. The chain of reasoning is architecture-mediated, with its units defined piecewise in each component's security license and evolving continuously as the system architecture, configuration, and security requirements evolve. As new kinds or types of vulnerability, threats, or exploits emerge, as well as new categories of effective responses and emerging alternative security mechanisms, we seek to elaborate and demonstrate this approach can continuously accommodate the specification and analysis of changing security requirements.

Product Lines: Alternatives, Versions, Variants of OA Elements

In producing a secure OA system in a software product line, there are several levels of variation available for producing artificial diversity among equivalent instances and for selecting and evolving in the face of threats.

At the highest level of granularity, a system developer or integrator can choose among alternative *producers* of similar components, services, and platforms [SWZ12]: For example, we can find *functionally similar* alternatives from software (component) producers of web browsers like Mozilla (Firefox, Camino, Sea Monkey) vs. Google (Chrome) vs. Microsoft (Internet Explorer), vs. others. Similarly, for word processors, we find alternatives including Microsoft (Word) vs. abisoft.com (AbiWord) vs. Google (Google Docs, which is a remote Web service rather than a component), vs. others. Likewise, for email and calendar applications, we find alternatives like Microsoft Outlook, Gnome Evolution, Google Mail, and Google Calendar, among others. For operating systems, we find Red Hat Enterprise Linux, Microsoft Windows, Apple OSX, and Google Android among others. Finally, note that some producers produce more than one alternative of the same kind of component or service, such as Mozilla's web browsers (Firefox, Camino, SeaMonkey), so that a choice among those particular components does not result in a change of producers.

Functionally similar components and services may not be exactly interchangeable, unless their interfaces are similar or identical. As such, it may be necessary to modify, for example, OA system topology, replace connector types, and other architectural measures may be necessary to change from one producer to another, depending on the functionality needed to satisfy functional requirements. However in general the overall functionality provided by the system remains substantially the same, but now the diversity among alternative system instances is the greatest: not only is the component, service, or platform distinct between two instances, but its architectural connections in the system will be distinct as will be the software development process and organization that produced it, so the chances of a common vulnerability are greatly minimized. Subsequently, when functionally similar components, connectors, or configurations exist, such that equivalent alternatives, versions, or variants may be substituted for one another, then we have a strong relationship among these OA system elements that is called a *product family* [NS86, Bo06] or a *product line* [CN01].

As described above, a shift from one alternative to another ordinarily requires a change in architecture, software connectors, and other measures. Changes between some alternatives will also produce a change of producers, while others will not. However, when components or connectors provide alternative implementations of the functionality they provide, then these are designated as versions. For example, most Linux operating systems support multiple file systems for data storage, though developers or integrators select their preferred file system for inclusion at either design-time or build-time. Similarly, for connectors to remote Web servers, developers or integrators may specify unencrypted (e.g., HTTP) or encrypted (e.g., HTTPS) data communication protocols for use in a Web-based enterprise system. Next, at the OA system configuration level, selection of alternative components or connectors, or of different versions of components or connectors result in different overall system versions that conform to a system product line. Further, recent advances in source code compilation now allow for creation of *functionally identical* variants of software components, though each variant has a different run-time image in the computer, through code randomization techniques [Fr10, SJWWF11]. Last, software product lines can be bound to a network of software producers, system integrators, and system users/consumers through a software ecosystem [Bo09], such that secure systems can be realized through composition or configuration at the software ecosystem level [SA12]. Consequently, we now have a complete and robust basis for specifying OA systems that can include components, connectors, or application systems from

alternative producers, or with different versions or variants included. This is now our basis for moving forward to address to address the challenges of creating secure OA systems through secured software product lines.

Secure Software Product Lines within an OA Software Ecosystem

Given the basis for software product lines for OA systems, we now address how to frame and align software system architectures with software security mechanisms. We use the following scheme to address this, as shown in Table 1.

<table>
<tr><td colspan="2">Security policies</td></tr>
<tr><td>Developers, system integrators and users</td><td rowspan="2">Persistent data</td></tr>
<tr><td>System configurations</td></tr>
<tr><td>Components</td><td rowspan="2">Ephemeral data</td></tr>
<tr><td>Connectors</td></tr>
<tr><td>Platforms</td><td rowspan="2">User I/O data</td></tr>
</table>

Table 1. Different system security elements whose rights and obligations depend on capabilities supported by lower level elements.

System security policies provide the overall context for what kinds of security mechanisms or capabilities (e.g., mandatory role-based data access control) are required by a particular system. The requirements must be realized through multiple levels of system composition that span a processing space from people to processing platforms, and through data/content space that is processed during system usage/operation.

Aligning system security elements with security mechanisms gives rise to the following associations:

Platform: base technological elements that constitute the computer environment that hosts the target system.

- *hardware*: specifies hardware confinement constraints needed to securely operate the software system configuration, potentially to address memory, storage, and external device port isolation (see SecureSwitch [SWZ12]). Hardware may be configured as an embedded processor, mobile computer (e.g., smartphone or tablet), personal computer, multi-processor computation server, or multi-server data center.
- *virtual machine*: a software layer that can isolate and confine the operating system, component applications, or application services from direct control of system hardware, network operations, or operating system processes. OSes, software systems, components, or connectors can each run within their own virtual machine, in in alternative configurations, as long as they are completely confined at a higher level of system security and do not overlap virtual machine boundaries [SSL99, Sm12].
- *network*: message filtering and access control firewalls for data/control flows that move across external hardware system security boundaries.
- *operating systems*: mandatory access control [LSM98, SSL99], capability type enforcement [Sm12], OS configuration parameters [STIG11], run-time audit logs, all currently coded and managed by system integrators/administrators.

Connectors: software mechanisms that implement secure communication mechanisms within and across system boundaries. Connectors enable security mechanisms providing:

- data cryptography (encryption/decryption) before/after data transfer
- component-connector-specific firewalls that can be implemented via (pre-conditions) constraints on in-bound data flow and plug-in/helper application invocation, or on out-bound data flow and external program invocations (post-conditions)
- multi-version connector configurations between components that allow for artificial diversity and dynamic reconfiguration potential through functionally similar versions.

Components: software mechanisms that implement application functionality required for the targeted system to operate as intended. Components enable security mechanisms providing:

- access/usage authentication control obligations (e.g., login with authorized identification and password) for which people in what roles (e.g., developer, system integrator, system administrator, system user) have the specified set of rights to view/update data, data control flow invocations, or external program invocations.
- encapsulate components as services within virtual machines to confine potential exploits, while mitigating their propagation.
- alternative versions that increase artificial diversity and enable dynamic replacement with functionally similar alternatives.
- multiple versions that allow for changes in vulnerability space, including concurrent versions with replicated input data, but different out data connector (routing) configurations.
- multiple variants that reduce vulnerability to component version attacks.

System configuration: the composition and interrelationship of components and connectors that together realize the system architecture, at design-time, build-time, or run-time. System configuration (or composition [Bo06]) enables security by providing:

- ability to host multiple (one or more) alternative, version, or variant system configurations on one or more processors (either single-core [SWZ12], multi-core, multi-blade, or multi-site) that can be dynamically selected in response to security policy directives or in response to detected threats.
- ability to host concurrently running multiple (one or more) alternative, version, or variant system configurations on one or more processors (either multi-core, multi-blade, or multi-site) that can be dynamically selected in response to security policy directives or in response to detected threats.
- ability to (formally) specify system configuration as an open architecture at design-time, build-time, and deployment run-time, along with automated tools that can verify the consistency, completeness, and traceability.

Developers, system integrators and users: denote the people authorized and trusted to work on or with the configured systems or its elements over time, depending on their externally assigned role(s).

- Developers should employ software development environments, tools, or processes that reinforce security-safe software coding practices of components or connectors they implement as products [Se08].
- Developers should produce multiple, unique executable variants of the components or connectors they produce and distribute.
- System integrators design OA system architecture.
- System integrators build OA system configurations that select from one or more component or connector alternatives, versions, and variants

- System integrators deploy one or more run-time system configuration variants that can be readily installed and appropriate parameters entered by system administrators or end-users.
- System integrators or system administrators, or automated mechanisms under their control, must be able to monitor and access system execution audit logs, to determine if threats or anomalous system behaviors are detected, and to dynamically reconfigure system configuration or security parameters in order to move the executable system into a more trusted operational state.
- Users must be provided with online identifiers or identification methods that enable them to access security controlled systems via one or more alternative authentication mechanisms in place.

In parallel with these processing security spaces are data security spaces:

User I/O data: data that may exist only as it passes across communication channels. Examples are keystrokes and mouse movements communicated from a keyboard or mouse to a processor, voice data from microphones and to speakers, wifi packets, and so forth. This data may be discarded or incorporated into ephemeral data.

Ephemeral data: data that exists in memory for a brief time before being either discarded or incorporated into persistent data. Examples are web forms that have been filled out but not submitted, and data in various sorts of hardware buffers.

Persistent data: data that exists for a substantial time on local disks or solid-state storage devices, USB memory sticks, DVD-ROM, or server storage.

Security policies: provide overall guidance and requirements for what security mechanisms and regimes are to be designed, implemented, and satisfied during the deployment, operation, and evolution of a specified system. Security policies:

- should provide *non-functional requirements* regarding the membership, structure, and behavioral specifications of each of the proceeding categories of security elements at minimum, or further specification of security sub-elements within each category, as per the security exposure of the system being addressed.
    - Non-functional requirements may only specify rights provided when corresponding obligations are fulfilled that cannot be automated or verified in lower level security elements.
    - Non-functional requirements should be expressible in human-readable and computer-processable forms within the system security policy license.
- must provide *functional requirements* regarding the membership, structure, and behavioral specifications of each of the proceeding categories of security elements at minimum, or further specification of security sub-elements within each category, as per the security exposure of the system being addressed.
    - Functional requirements are those that can be formalized, automated, and verified by corresponding automated mechanisms available at lower level security elements.
    - Functional requirements may only specify rights provided when corresponding obligations are fulfilled that must be automated or verified in lower level security elements.
    - Functional requirements should be expressible in human-readable and computer-processable forms within the system security policy license.

The case study that follows describes where these different system security elements appear in forms that can be available for review by authorized Program Acquisition personnel.

Case Study of a Secure Product Line for an Enterprise System

Let us consider what needs to be specified during the acquisition of an enterprise system that incorporates common office productivity applications that run on a personal computer networked to remote servers. Such a system can include a web browser, word processor, email and calendaring applications that are configured to operate on a personal computer, where the PC's operating system, Web browser and other applications need to be configured to access remote data/Web content servers. Figure 1 shows part of the system ecosystem of software producers and the components they can provide for our enterprise system.

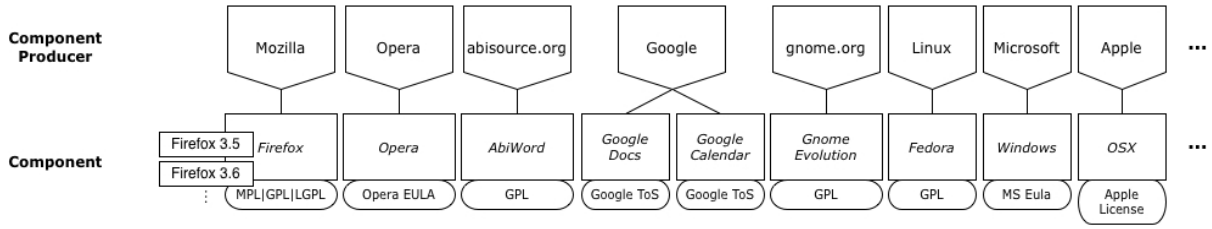

Figure 1. A partial view of a software ecosystem of producers and the software components for an enterprise system they produce

Figure 2 shows the design-time architecture of such an enterprise system. What might a secure product line for a system like this involve, and how might it provide benefits and security qualities to be specified for design time, build time, and run time? How can its OA and product-line characteristics contribute to security throughout the acquisition system life-cycle?

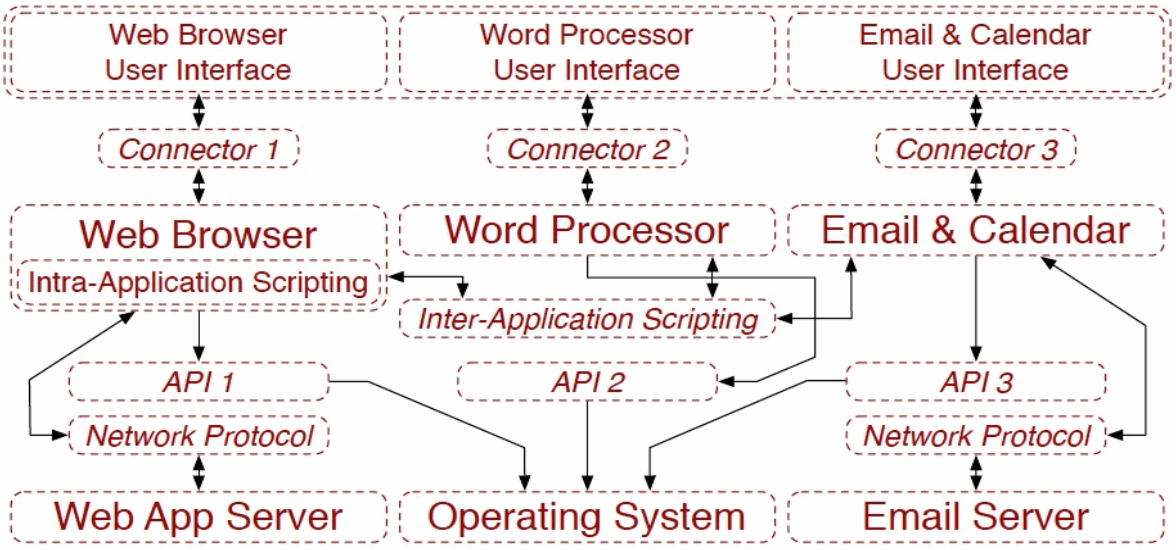

Figure 2. A design-time reference model of an OA system that accommodates multiple alternative system configurations.

We envision an approach in which non-functional requirements, such as security, reliability, and evolvability requirements at acquisition time, are elaborated at design and build times by specific functional requirements that explain how and to what degree the non-functional requirements are going to be satisfied at run time. Analogous to our previous work with intellectual property (IP) licensing, we envision that these requirements are structured in the same logical forms as IP licenses (with specific rights that are obtained only by fulfilling specific obligations), and managed through the architecture by

the same approach of calculating which obligations are satisfiable, in what way, and as a result what rights are available [AAS09, ASA10, SA11].

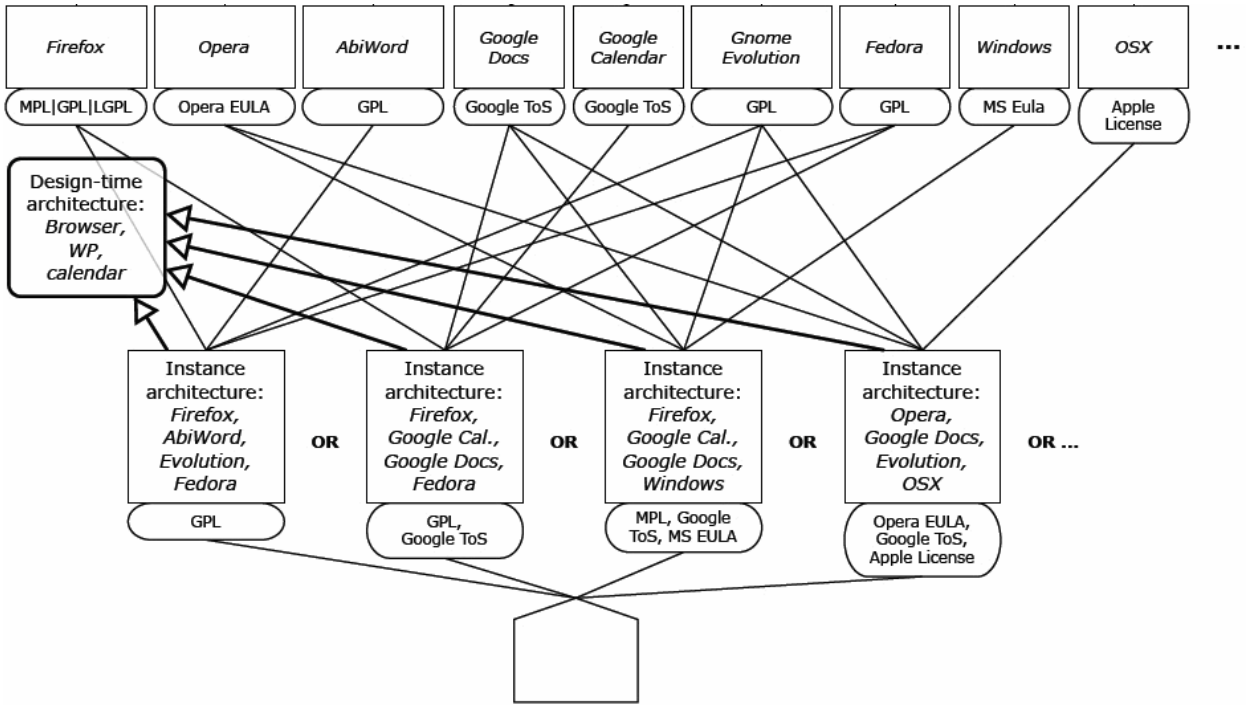

Figure 3. A view of an OA software ecosystem that provides alternative, functionally similar components compatible with the reference design-time architecture.

Figure 3 illustrates a possible OA software ecosystem for this product line. Here a number of possible producers and alternative components have been placed into play, and four specific instance architectures (produced in four specific ecosystems) have been sketched. With appropriate architectural topologies, and appropriate shim components and connectors inserted between the major components, each of these four instance architectures can support the same functionality. It is also possible to achieve different nonfunctional qualities including security qualities through the four choices; for example, by requiring that OS be an appropriate Security-Enhanced version of Linux, or by requiring that the network protocol connector be HTTPS.

Within the overall ecosystem of Figure 3, Figure 4 shows one possible instance ecosystem involving specific producers (Mozilla, abisource.org, gnome.org, Red Hat) and specific alternatives (Firefox, AbiWord, Evolution, Fedora).

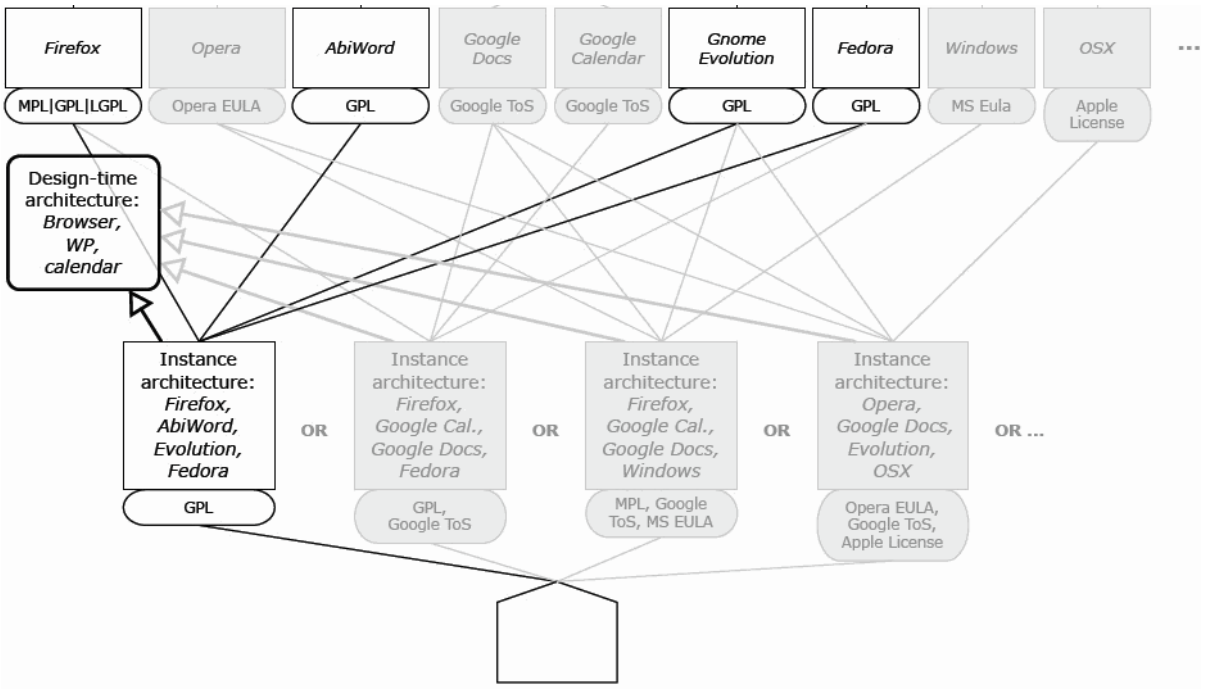

Figure 4. A selection among alternative components that can be included at build-time to produce an integrated system compatible with the design-time reference.

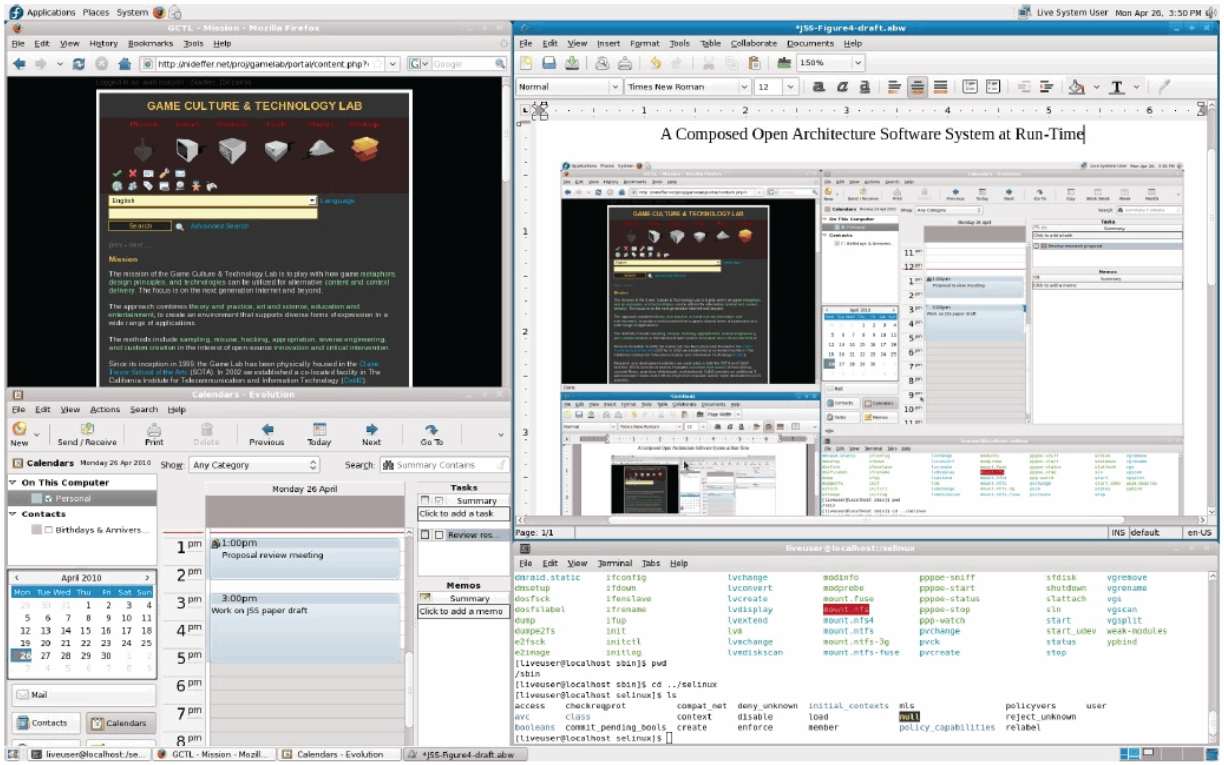

Figure 5. An end-user run-time version of the selected alternative components that fulfills the design, where the Red Hat Enterprise Linux operating system (lower right corner) can utilize the security modules library, SELinux, for coding and enforcing mandatory access control on programs/data, and other security capabilities.

Acquisition-time requirements such as the use of SE Linux and the use of HTTPS could be satisfied by this choice; with an appropriate architecture, the IP licensing obligations could also be satisfied. At design time the functional requirements would need to be satisfied by appropriately-specified shims inserted among the principal components, and if such shims could be designed then this would be the proof that the acquisition-time nonfunctional requirements could also be satisfied. Figure 5 shows a run-time view of this instance architecture, resulting from the specific OA ecosystem and instantiating the overall ecosystem of Figure 3 and the software product line the software system is an instance of.

This instance architecture has both a manageable IP license regime that insures its openness, and a manageable security regime. For IP, in this architectural instance, all component versions can be selected to use permissive licenses (Web browser, Web server) or reciprocal GPL licenses (word processor, email, calendar, and operating system) they are cleanly separated by dynamic run-time links, which are type of connector that does not transmit IP obligations or rights, though allows for control flow integration, and data flow interoperation.

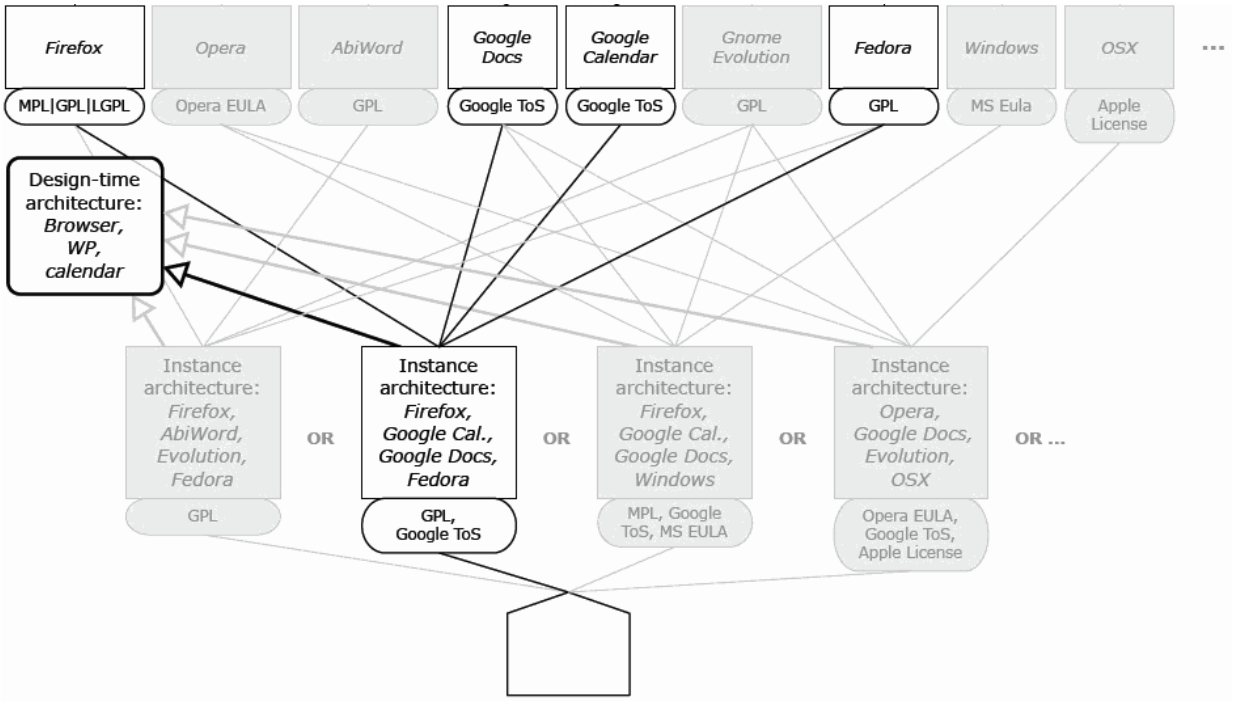

Figure 6. An second system configuration, using alternative but functionally similar components.

Figure 6 outlines an alternative system configuration and the instance ecosystem that produces it. This instance architecture substitutes services for components in the case of Google Docs for the word processing functionality and Google Calendar for the calendar functionality. With appropriate shims and changes to the architectural topology this combination of major components could also support the system's functional requirements, and because the services are accessed through client-server connections, which block the propagation of most license obligations, there are a number of ways to satisfy the IP constraints imposed by the component and service licenses.

This alternative configuration also highlights possible acquisition-time concerns and the nonfunctional requirements and security license issues that follow from them. For example, a remote service such as Google Docs provides benefits and imposes costs with respect to a compiled component such as

AbiWord. On the one hand, the remote service makes some qualities easier to achieve (data sharing, backup, etc.) but on the other may make some qualities harder to achieve (data security over a network connection and in the “cloud”, up-time of the service, little or no control over when new versions of the service are used compared to complete control over when new versions of a component are integrated).

- Who in the ecosystem of human actors for this system has the right to make the decisions to use a service in place of a component, or one component version in place of another? What obligations are they required to satisfy first? These questions are of concern at acquisition time and, we claim, are addressable by *acquisition licenses* that restrict rights and impose obligations important to system acquisition officers just as IP licenses do for IP rights and obligations important to software producers.
- When can these decisions be made? In traditional development processes these would occur at design time, but in the larger view we propound here such decisions, or rather the policies or acquisition licenses that control them, are perhaps more properly considered at acquisition time. As we will see below, it is also possible that in order to achieve specific security qualities they might be made at build or run time, in response to specific threats.

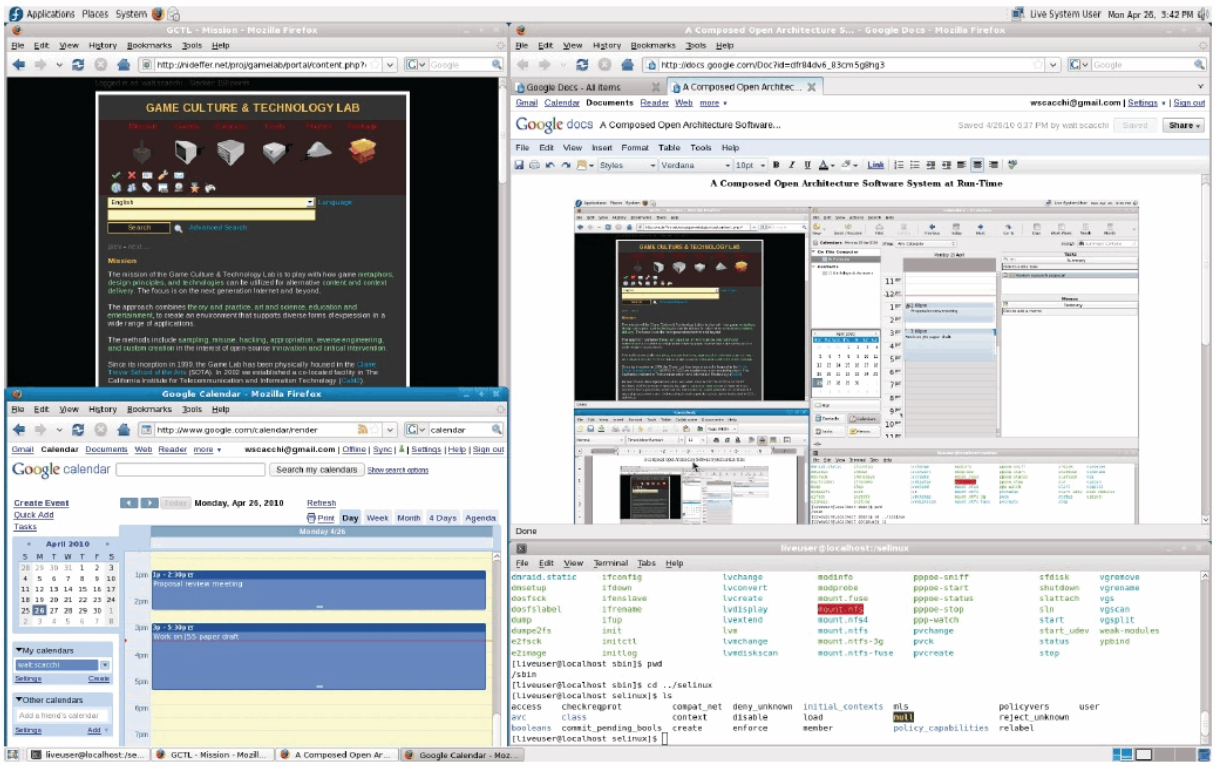

Figure 7. An end-user view of the alternative run-time system configuration

Figure 7 shows a run-time view of this alternative configuration. To the end user this system appears quite similar to the one in Figure 5, and the differences might scarcely be noticed, which raises the next set of possibilities.

Both these instance architectures specify specific alternatives for the major components, for example Firefox for the web browser component. But which version of Firefox? For example, it is quite possible that both the instance architectures discussed above could be implemented using either Firefox 10 or

Firefox 11, satisfying all the functional requirements with no change to the instance architecture and no revision of software shims. Who has the power to decide to use version 10 rather than version 11? How late in the software process can this decision be made -- for example, could it be made as late as system startup time by a system user, in response to a particular security attack on the previous configuration?

At the conceptually lowest level, the advent of code randomization and multi-variant software executables leads to the possibility of substituting essentially equivalent variants of the same component, most obviously at build time. The decision to substitute one variant for another, or the decision to allow the substitution, can be made through the entire range of development times from acquisition time to run time. The substitution can be put into effect by a human actor or by a software monitor following a security policy, either randomly or in response to specific events in the environment.

Finally, an orthogonal consideration is the use of containment vessels to encapsulate components or subsystems within a virtual machine, to monitor and control interactions among components and subsystems in order to block attacks and protect vulnerable parts of a system. Figure 8 shows a

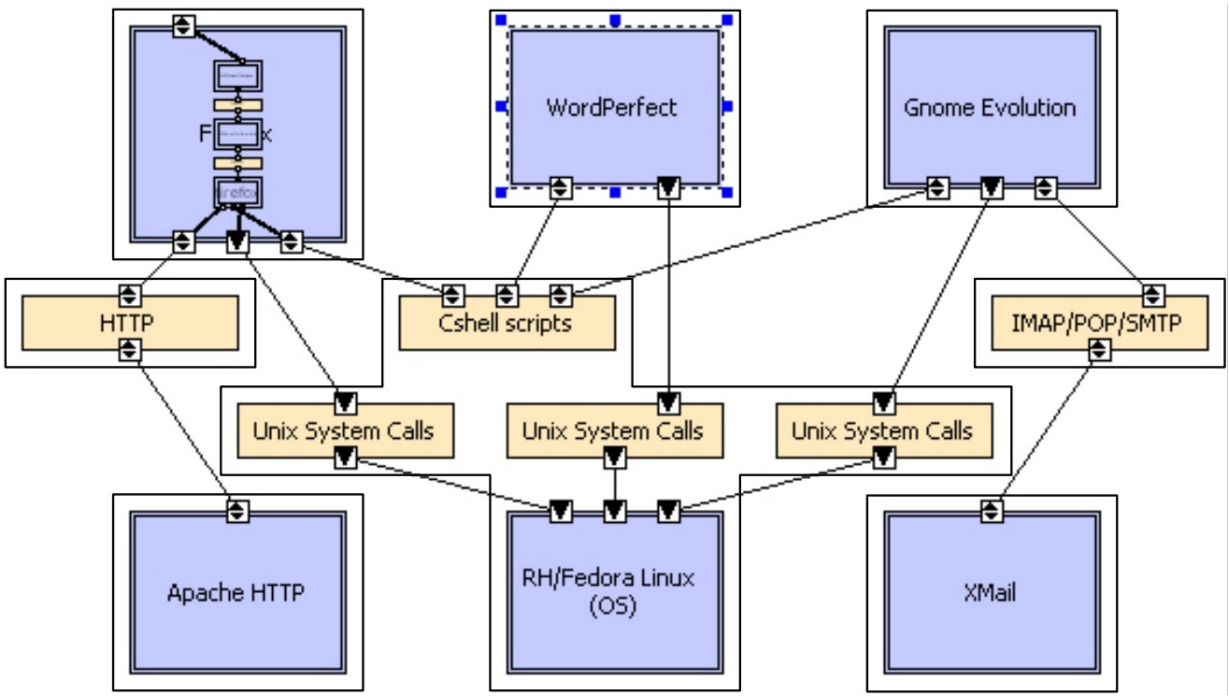

Figure 8. A security configuration alternative for the run-time configuration instance that encapsulates OA system components and connectors within different virtual machines (e.g., using [Xen12]).

screenshot in ArchStudio of a design-time architecture utilizing eight containment vessels, seven for individual components and connectors and the eighth for the group of components and connectors associated with the OS.

For security, the GPL'd Fedora can employ the SELinux capabilities to restrict all shell/operating systems commands through mandatory access control and type enforcement (see Figure 8), while other components can all be contained within within one (for minimal security confinement) or more (for increased security confinement on a per component basis) Xen-based virtual machines (again, See Figure 8). The interoperability of SELinux and Xen is now a common feature of many large Linux system installations (e.g., Amazon.com now has more than 500K Linux systems running Xen) [Prg12, Xen12].

Discussion and Conclusions

Our goal in this study is to develop and demonstrate a new approach to address challenges in the acquisition of secure OA software systems. Program managers, acquisition officers and contract managers will increasingly be called on to provide review and approval of security measures that are employed during the design, implementation, and deployment of OA systems. We seek to make this a simpler and more transparent endeavor. This requires security policies that are appropriate for review and approval during acquisition by people who may not be expert in the specifics of how best to insure that secure systems will result. Our view is to address this need by investigating how best to specify or model system security in ways that can accommodate security as a continuous process that must be supported throughout the system acquisition life cycle for OA systems [SA08, SA11].

Our efforts reported here reveal that it is possible to employ a scheme through which complex OA systems can be designed, built, and deployed with alternative components and connectors into functionally similar system versions, in ways that allow for overall system security through the use of multiple security mechanisms. We described a scheme for how to realize and specify such OA system configurations in ways that are inherently compatible with existing security mechanisms, and this scheme does not assume that individual system elements must be secure before inclusion into the secured system's configuration. Central to our scheme is the incorporation of software product line concepts that are integrated with security mechanisms in a coherent way that is amenable to automated support and acquisition management. We also provided a case study that reveals where and how we specify a secure OA enterprise system product line in ways that can accommodate the diverse needs of software producers and developers, system integrators, users and acquisition managers. What remains as an important next step for this line of research effort is to more fully articulate how to simply and transparently specify OA system security using streamlined security policies using the kind of system security licenses we anticipate [SA11], as well as designing and developing a prototype automated system that can support the modeling and analysis of OA system security policies, alternative version OA system configurations, and different OA security licenses.

Acknowledgements

Research described in this report was supported by grant #N447602-12-1-0004 from the Acquisition Research Program at the Naval Postgraduate School, and from grant #0808783 from the National Science Foundation. No review, approval, or endorsement implied.

References

[AAS09] Alspaugh, T.A., Asuncion, H. and Scacchi, W. (2009). Intellectual Property Rights Requirements for Heterogeneously Licensed Systems. In *Proc. 17th IEEE International Requirements Engineering Conference (RE'09)*, 24–33, Aug. 31–Sept. 4 2009.

[AAS11] Alspaugh, T.A., Asuncion, H. and Scacchi, W. (2011). Presenting Software License Conflicts through Argumentation, *Proc. 23rd. Intern. Conf. Software Engineering and Knowledge Engineering*, July 2011.

[ASA10] Alspaugh, T.A., Scacchi, W., and Asuncion, H. (2010). Software Licenses in Context: The Challenge of Heterogeneously Licensed Systems, *J. Association for Information Systems*, 11(11), 730-755, November 2010.

[Bo06] Bosch. J. (2006). The challenges of broadening the scope of software product families. *Commun. ACM* 49, 12 (December 2006), 41-44.

[Bo09] Bosch, J., (2009). From software product lines to software ecosystems. In: *Proc. 13th Intern. Software Product Line Conference* (SPLC'09), 111-119.

[BA05] Breaux, T.D. and Anton, A.I. (2005). Analyzing goal semantics for rights, permissions, and obligations. In *Proc. 13th IEEE International Conference on Requirements Engineering (RE'05)*, 177–188.

[BA08] Breaux, T.D. and Anton, A.I. (2008). Analyzing regulatory rules for privacy and security requirements. *IEEE Trans. Software Engineering*, 34(1), 5–20.

[CN01] Clements, P. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns. Addison-Wesley, New York.*

[DoD10] DoD Open Source Software (OSS) FAQ (2010). *Frequently Asked Question regarding Open Source Software (OSS) and the Department of Defense (DoD)*. http://cio-nii.defense.gov/sites/oss/Open_Source_Software_%28OSS%29_FAQ.htm

[DoD11] *Department of Defense Strategy for Operating in Cyberspace*, July 2011. http://www.defense.gov/news/d20110714cyber.pdf

[F04] Firesmith, D. Specifying reusable security requirements. *Journal of Object Technology*, 3(1), 61-75, Jan-Feb. 2004.

[Fr10] Franz, M. (2010). E unibus pluram: Massive-Scale Software Diversity as a Defense Mechanism, *New Security Paradigms Workshop (NSPW'10)*, Sept. 21–23, Concord, Massachusetts, USA.

[Ga10] Garcia, P. (2010). Maritime C2 Strategy: An Innovative Approach to System Transformation, *Proceedings 15th. International Command & Control Research & Technology Symposium*, Paper 147, Santa Monica, CA.

[Gi11] Gizzi, N. (2011). Command and Control Rapid Prototyping Continuum (C2RPC) Transition: Bridging the Valley of Death, *Proceedings 8th. Annual Acquisition Research Symposium*, Vol. 1, Naval Postgraduate School, Monterey, CA.

[GPL07] *GNU General Public License*. http://www.gnu.org/licenses/gpl.html

[H11] Attack of the Computer Mouse, *The H Online Security*, 29 June 2011. http://h-online.com/-1270018

[LSM98] Loscocco, P., Smalley, S., Muckelbauer, P., Taylor, R., Turner, S. and Farrell, J. (1998). The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environment. *Proc. 21st National Information Systems Security Conference*, 303-314.

[Navy10] Navy.mil (2010). PEO IWS Releases Open Architecture Contract Guidebook Update, http://www.navy.mil/search/display.asp?story_id=53661. Also see *Navy Open Architecture Guidelines* and related documents at https://acc.dau.mil/oa.

[NS87] Narayanaswamy, K. and Scacchi, W. (1987) Maintaining Configurations of Evolving Software Systems, *IEEE Trans. Software Engineering*, 13(4), 323-334.

[Prg12] *SELinux on Xen* (2012). http://wiki.prgmr.com/mediawiki/index.php/SELinux_on_Xen

[SJW11] Salamat, B., Jackson, T., Wagner, G., Wimmer, C., Franz, M. (2011). Run-Time Defense against Code Injection Attacks using Replicated Execution, *IEEE Transactions on Dependable and Secure Computing, Volume 8, No. 4*, July 2011.

[Saw11] Sawers, P. (2011). US Govt. plant USB sticks in security study, 60% of subjects take the bait. *TNW: The Next Web*, 28 June 2011, http://thenextweb.com/industry/2011/06/28/us-govt-plant-usb-sticks-in-security-study-60-of-subjects-take-the-bait.

[SA08] Scacchi, W. and Alspaugh, T. (2008). Emerging Issues in the Acquisition of Open Source Software within the U.S. Department of Defense, *Proc. 5th Annual Acquisition Research Symposium*, Vol. 1, 230-244, NPS-AM-08-036, Naval Postgraduate School, Monterey, CA.

[SA11] Scacchi, W. and Alspaugh, T. (2011). Advances in the Acquisition of Secure Systems Based on Open Architectures, in *Proc. 8th. Annual Acquisition Research Symposium*, Monterey, CA, May 2011.

[SBN11] Scacchi, W., Brown, C., and Nies, K. (2011). *Investigating the Use of Computer Games and Virtual Worlds for Decentralized Command and Control*, Final Report,
Grant #N00244-10-1-006, Institute for Software Research University of California, Irvine, July.
http://www.ics.uci.edu/~wscacchi/ProjectReports/NPS-Reports/DECENT.pdf

[Se08] Seacord, R. (2008). *The CERT C Secure Coding Standard*, Addison-Wesley, New York.

[Sh11] Shrobe, H. (2011). Secure Computing Systems, Presentation at the *Darpa Colloquium on Future Directions in CyberSecurity*, November, Arlington, VA. www.darpa.mil/WorkArea/DownloadAsset.aspx?id=2147484460

[Sm12] Smalley, S. (2012). *The Case for Security Enhanced (SE) Android*,
https://events.linuxfoundation.org/images/stories/pdf/lf_abs12_smalley.pdf

[SSL99] Spencer, R., Smalley, S., Loscocco, P., Hibler, M., Andersen, D., and Lepreau, J. (1999). The Flask Security Architecture: System Support for Diverse Security Policies, *Proc. Eighth USENIX Security Symposium*, 123-139.

[STIG11] Security Technical Information Guide, *Android 2.2 (Dell)*.
http://iase.disa.mil/stigs/net_perimeter/wireless/smartphone.html

[Stux11] Stuxnet (2011). Overview at http://en.wikipedia.org/wiki/Stuxnet. Also see M. Falliere, et al., (February 2011), *W32.Stuxnet Dossier*, Version 1.4,
http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf

[SWZ12] Sun, K., Wang,J., Zhang, F. and Stavrou, A. (2012). SecureSwitch: BIOS-Assisted Isolation and Switch between Trusted and Untrusted Commodity OSes. *Proc. 19th. Annual Network and Distributed System Security Symposium.*

[Xen12] *Xen Hypervisor Project*, http://www.xen.org/products/xenhyp.html

[YC06] S. S. Yau and Z. Chen. A framework for specifying and managing security requirements in collaborative systems. In *Proc. Third International Conference on Autonomic and Trusted Computing* (ATC 2006), 500–510, 2006.