

Security for Automated, Distributed Configuration Management

P. Devanbu, M. Gertz, S. Stubblebine*

Contact: `devanbu,gertz@cs.ucdavis.edu`
`stubblebine@{cs.columbia.edu, certco.com}`

April 26, 1999

Abstract

Installation, configuration, and administration of desktop software is a non-trivial process. Even a simple application can have numerous dependencies on hardware, device drivers, operating system versions, dynamically linked libraries, and even on other applications. These dependencies can cause surprising failures during the normal process of installations, updates and re-configurations. Diagnosing and resolving such failures involves detailed knowledge of the hardware and software installed in the machine, configuration manifests of particular applications, version incompatibilities, etc. This is usually too hard for end-users, and even for technical support personnel, specially in small businesses. It may be necessary to involve software vendors and outside consultants or laboratories. Employees working on sensitive, proprietary projects may even have to resort to calling the help line of an application vendor and discussing details of their desktop configuration. In order establish valid licensing, the user may be forced to disclose additional details such as the user's identity, machine identification, software serial number, etc. This type of disclosure may reveal proprietary information or (worse) security vulnerabilities, and increase the risk of attack by hackers or cyber-criminals. An adequate solution to the distributed configuration management problem needs to address the security concerns of users, administrators, software vendors and outside consultants: keeping details of installations private, authenticating licensed users and software vendors, protecting the integrity of software, secure delegation across administrative boundaries, and protecting proprietary information. Existing commercial and research systems [12, 8] provide distributed configuration management by distributing configuration information and software over local and wide-area networks. They provide flexible, automated, distributed configuration management. However, many or most of the security issues listed above remain to be addressed. These issues are the central focus of our research.

*Address for the first two authors: Dept. of Computer Science, Room 2063, Engineering Unit II, Davis, CA 95616. Last author: CertCo Inc., 55 Broad St., Suite 22, New York City, NY 10004

1 Introduction

Traditionally, the installation of software was simple: an administrator received a tape, unbundled the software off the tape, created a few initialization and profile files, and the software was off and running. Later releases for the software would arrive likewise in magnetic media and be handled likewise. With component-based distributed software running on a networked computer, the process is simpler in some ways and more complex in others. Rather than using cumbersome physical media, software can be dispatched over the network (perhaps even automatically as agents or applets). However, there may be several constituent components, from different vendors. The configuration and installation process for each of these may be distinct, and there may be a nontrivial integration step. In addition, each networked computer may have different resources and user requirements, and may require customized software installation.

Configuration management continues to be an issue after initial installation: the configuration of a machine may change, for example, due to the installation of a new device driver or an upgrade to the operating system. A vendor may produce new releases of a software product to fix faults or enhance functionality, thus leading to the installation of new files that may have consequences beyond the specific product itself.

Papers from the University of Colorado Software Engineering Research Laboratory (UC SERL) (See, for example [9]) have described a useful characterization of the *software deployment lifecycle* from the perspective of the software producer, and the software consumer:

- From the software producers point of view, there are two significant lifecycle events: *release*, when the software (or a new version) is first introduced, and *retire*, when an old version is deprecated by the vendor.
- From the software consumers point of view, there are four significant¹:
 1. *Install*: the first installation of the software product.
 2. *Reconfigure*: Selects a new configuration (from the list available) of an installed software in response to new requirements at the consumer's site. E.g., the user may require some a dictionary for a new language in a word processor.
 3. *Adapt*: A change made in the consumer's configuration requires a change to the installation of a product. For example, a new printer may be installed, which requires the installation of new fonts.
 4. *Update*: A change made to a software product in response to a change initiated by the software vendor.

We adopt the UC SERL view of the software deployment lifecycle. SERL has developed an architecture (the Software Dock [8]) to address the needs of this lifecycle model. Ar-

¹We present here only a simplified view of the deployment lifecycle. See [9] for full details.

architectures for configuration management have also been devised by Marimba [12] and the Desktop Management Task Force consortium [2].

Our goal in this paper is to consider the security problems that arise in software configuration management.

2 Security Issues in Distributed Configuration Management

The most common situation is a distributed computer network where individual machines have complex installations tuned to the needs of specific users. In this context, there are various security goals, such as privacy, authentication, and delegation etc. We illustrate the need for these security features with an example.

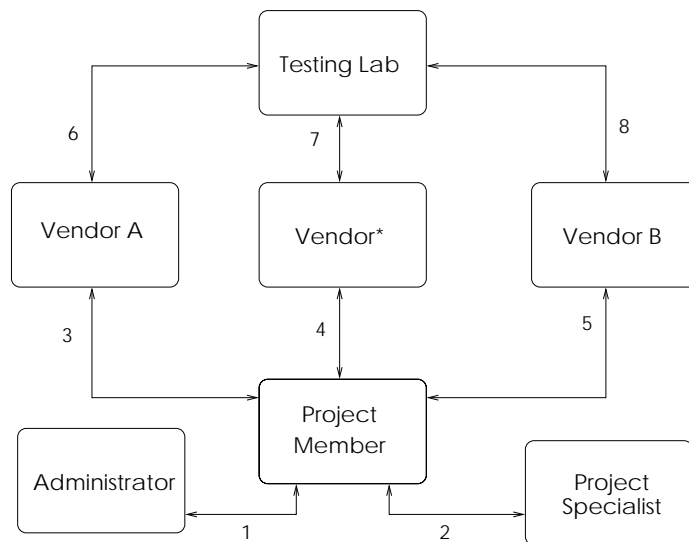


Figure 1: An Example Distributed Configuration Management Scenario

2.1 An Example

Consider Figure 1. A project member signs up to work on a special project, and checks with the departmental network administrator (edge 1) to find out what software is needed. The overworked administrator tells the project member to consult with a project specialist (edge 2) (an outside consultant) for details on the software required. The administrator in a sense here is “certifying” the project specialist as the responsible person for configuring software for the project.

The project specialist identifies an application A^* and vendor for the project member, who contacts the vendor (4) to obtain the product. The vendor (Vendor*) uses components C_A, C_B from two other vendors (Vendor A) and (Vendor B); C_A and C_B occasionally have conflicts. To determine if such conflicts exist, the Vendor* always consults a testing lab (7) which obtains versions (6,8) and tests them (or has already tested them in the past). The lab informs Vendor* (7) of the compatible versions. The project member finds the right version information from Vendor* (4) and obtains the right versions of the components (3,5).

We use the term “testing lab” advisedly; in practice, information such as this is obtained by calling the tech-support line for Vendor*, which results in much wasted time, often without any positive results. Another, more ad-hoc approach is to comb the web and usenet groups. This process is unpredictable, time-consuming, and often also fruitless. Another (more expensive) approach is to hire a knowledgeable consultant, who plays the role of a “testing lab” and makes it his/her business to be aware of incompatibilities. We envision a configuration framework that supports this more directly, and thus use the term “testing lab” illustrate a more systematic approach, which allows organizations with such capabilities to offer these services for a (possibly electronically delivered) fee. This type of dialog with vendors of different components and testing labs could be repeated for all the different applications (both work-related and personal) that the project member installs.

This scenario just covers the *install* part of the consumer side of the deployment lifecycle. Now consider a *reconfigure* event. The project member decides to install a video camera device for teleconferencing; in the process of installing the device driver, a new component is loaded, which causes the installed application from Vendor* (described above) to fail. The project member contacts the project specialist, who after much investigation, fails to resolve the problem, and refers the project member to Vendor*. The project member contacts Vendor* (4)’s support service; and identifies herself as a licensed user of A^* . Vendor*, after obtaining an elaborate description of the problem, and the project member’s installation, realizes that the driver uses a newer version of C_B that conflicts with the version of C_A that is currently installed. Vendor* then contacts the testing lab, which identifies a newer version of C_A that is compatible with the newer version of C_B . Again, as before, this situation is somewhat idealized; in practice, the project member (or any user) spends many hours trying to localize the problem, with or without the help of a voice at the other end of a support line; more often than not, Vendor* is unable or unwilling to diagnose the problem. The situation remains often unresolved, and the user is forced to abandon either the effort to reconfigure his system, or the use of application A^* .

A similar scenario can be described for an *update* whereby a new version of A^* is released. Vendor* may inform the project member about a new update of A^* ; if this new version is authorized by the project specialist the project member may go ahead and obtain the update from Vendor*. This update may have unintended effects on other software that the project member has installed, which in turn may trigger additional dialogs between the project member, vendors of other software, their testing labs etc. These dialogs will involve description of hardware/software installed on the project member’s machine, proof of licenses

being held etc.

Many applications ship with a range of configuration options that can be adjusted to suit the needs of a specific user's hardware/software set up on a specific machine. Thus, without adjusting the rest of the installation and contacting other vendors etc., it may be quite feasible to adjust the configuration/installation of an application to get it perform satisfactorily. However, the tuning of these options, specially for complex applications such as business process support, databases, and even word processing (with special needs for fonts, print drivers etc) can be quite complex, and require skills and resources beyond the capabilities of the systems administrator of a small organization. So even with flexible applications, the best solution is often to simply outsource the configuration task to an external entity.

We hasten to emphasize that we do not claim to advance a solution for the configuration management problem *per se*. Incompatibilities and other difficulties such as those defined above may happen, and will often require human effort to diagnose and solve them. Our goal rather is to ensure that *if such information is available*, it is promptly, reliably and efficiently delivered to those who need it, subject to certain security requirements that we shall discuss briefly now, and in more detail later.

2.2 Security Issues in this example

There are several security issues that arise here. First, the project member ends up revealing a great deal of information about the configuration of her machine to the project specialist, vendors, and testing labs. Certainly, there is a strong threat to the privacy of the project member (*e.g.*, the project member may have personal applications or software on the machine, and may unwittingly reveal this information by simply disclosing the existence of a particular DLL or component). In addition, detailed knowledge of the configuration of the project member's machine is made available to a number of outsiders, who may be now be able to attack known vulnerabilities in the machine, and gain proprietary secrets or engage in cyber-crime or cyber-terrorism.

Second, there are a number of authentication issues. The vendor needs to know that the project member is a licensed user of the relevant software. In addition, the project member needs to know that versions of the software that are installed have not been tampered with and modified, and are actually the right versions prescribed by the project specialist, the vendors, the testings labs etc.

Finally, there are delegation issues. The administrator delegates configuration authority to the project specialist, and the project specialist in turn delegates some authority to vendors and testing labs. These delegations may have associated time-periods of validity. These delegations need to be handled in a secure and timely fashion.

These issues are discussed in greater detail in Section 4.

3 Current Approaches

Architectures such as Marimba [12] and the Colorado Software Dock [8] deal explicitly with this problem. All architectures consist of these elements

1. A **language** for describing sites, configurations and updates. This language has facilities for *explicit* hierarchical descriptions of configurations, listing the various required elements for a software installation, as well as implicit *constraints* on software (such as requirements for certain types of functionality, without explicitly identifying the component).
2. An **Event Messaging** mechanism, for notifying events relating to the deployment lifecycle to vendors, customers, etc.
3. An **agency** on the customer side and on the vendor side for making use of the configuration descriptions, site descriptions, and the event notifications to derive consistent configurations and download the necessary software.

While the different approaches [2, 12, 8] have different advantages and disadvantages (See [1] for a survey), we are primarily concerned here with security issues. In the following section, we identify the security issues that are of concern in the networked configuration management context.

4 Research Issues

We identify several critical security needs in systems that perform distributed software configuration. Some of these are handled by existing systems; others are not adequately dealt with. We are currently developing a flexible, retargetable architecture that addresses the security needs; this paper lays out the requirements and issues that must be addressed by such an architecture.

Integrity is the property that a data item (software, data, etc) is intact, and has not been tampered with. In the context of software configuration, integrity requirements arise in different contexts:

Software Integrity Software that is shipped from the vendor must arrive intact at the installation site. For example software from Vendor* that arrives to the student's PC (link 4, Figure 1) must be checked for integrity and completeness.

Configuration Integrity The configuration of a machine at a user's site must not be tampered with by unauthorized personnel.

Message Integrity Messages describing configurations (both correct ones and inoperable ones) must arrive correctly at the needed sites.

Cryptographically, integrity is established using message authentication codes [13] (MACs) or signatures. These techniques can be used in this context. Current systems such as Marimba implement this requirement via digital signatures.

Authentication It is often necessary to establish the authenticity of a message or a data item, *e.g.*, to make sure that the message really did originate where it claims to have.

Authenticating Software Vendors We need to verify that the originator or source of the software is indeed who it is claimed to be. For example, in figure 1 we need to establish that the delivered component C_A in step 3 really did originate with Vendor A and was not subject to tampering.

Authenticating Software Users During the *reconfigure* or *update* scenarios, when a customer contacts a vendor for help, the vendor needs to establish that the caller is actually a licensed user.

Cryptographically, authentication is established using signatures [13] within a public-key system. Other issues arise in this context, such as securely associating an entity or a role with a public-key, which is dealt with below under “delegation”.

Privacy refers to the goal of keeping certain information secret. There are different types of privacy goals in the software configuration context.

Privacy of Content Software configuration activities involve the exchange of several valuable pieces of intellectual property, some of which may have commercial implications. Thus, certainly software vendors may want to encrypt the software prior to shipping it to paying customers. It may also be desirable to insert watermarks into software binaries to identify the origin of illegal copies. Testing labs or consultants obtain configuration rules (such as incompatibility of C_A with certain versions of C_B) at great expense, and may wish to keep this information private, and only reveal it to paying customers.

Privacy of Configurations Currently, users seeking to diagnose configuration problems are forced to reveal not only their identities (to establish proper licensing) but also details about the configuration on their site. In general, a user may not wish to reveal this information. Consider the situation where an employee of Microsoft is forced to reveal information about the configuration of her PC to a Lotus Technical support person to help diagnose a configuration problem. In the context of automated configuration management engine, this information would be asked for and transmitted without user intervention, thus leading to the risk of unwanted and unknown disclosures.

There is only one known technique for inquiring about subscription information without revealing the identity of the inquirer yet insuring that the entity doing the inquiring is authorized to access this information. It is called, Unlinkable Serial Transactions (UST) [16]. However, UST must be used in conjunction with some form of network anonymity

mechanisms such as the Anonymizer[17].² Although UST doesn't inherently enable the server to profile the client. Application data can very well do this. For example, two queries having the same unusual subsets of components are likely to be queries from the same client.

Secure Delegation aims to selectively enable certain entities to perform certain actions. For example, delegating certain entities to take on certain roles [15] such as *administrator* or *specialist* or *Project_X Configurator* or *Testing Lab*. The scenarios described above illustrate different types of delegation. The student has some limited authority of his workstation. Also the system administrator has other authority. Each can delegate authority they possess to others. The student may delegate to the support staff the authority to a) add files to user space that are needed for the application at hand, and 2) to distribute some information about the user space configuration to an entity trusted by the system administrator . This is one type of delegation.

The support staff in turn delegates software configuration for one specific course to the teaching assistant; this amounts to "delegation of delegation". In another type of delegation, V* delegates to the testing lab to find out which version of C_A is compatible with which version of C_B . This amounts to delegation not of where to obtain software, but where to obtain configuration *rules*. In a fully automated configuration management architecture, these delegations, need to be authenticated through the use of certificates (some elements of the needed functionality have been described in [5])

Marimba [12] allows certain limited types of delegation through the use of *channels* and *sub-channels*. However, none of the systems allow fully certified delegations, delegations of delegations, delegations of configuration management rules etc.

5 Research Plans

We are interested in exploring several key issues that would underly a *secure* architecture for distributed software configuration over the internet. In this section, we give a list of issues that arise in this context, and are central to our research.

Languages Automatic configuration management hinges on an expressive description language. Current configuration management languages (CML) [11, 18, 2] are adequate for describing manifests and configuration rules. But they completely ignore security requirements such as authentication, delegation, etc. Our goal is to introduce such security features into CMLs. We take an approach to CMLs as being based on an object-oriented data model (OODM), as used in object-oriented database systems [14].

²Other techniques such as the Anonymizer [17] do not, alone, address the issue of whether the end user is authorized to query the end server. This is why UST is needed.

Modeling configuration, querying configurations and messaging among communications will all be based on this OODM. Security features such as delegation and privacy, can be implemented as view definitions on the configuration data. Constraints on valid configurations can be expressed as constraints on the data. The data, constraints and the view definitions are certified and protected using public-key cryptography. Determining the correct configuration at a site (for installation or for update) will be implemented as a query evaluation procedure, which collects data from the available views and attempts to find a satisficing answer. This will be implemented by extending the set-based cryptographic certificate distribution techniques described in [7] to an object-oriented model.

Cryptographic Techniques underly many of the goals described in the previous section. Some currently available techniques are quite relevant to the *privacy* and *authentication* goals outlined above (for example, UST [16] and Anonymizer [17] for protecting the privacy of the user while providing authentication to the software vendor). Our goal is to adapt these techniques for use in software configuration management. For the *delegation* goals described above, we will use the view definition approach outlined above, where the views are defined using cryptographically signed certificates [7].

Messaging Infrastructure Current approaches (See Section 3) all include a messaging infrastructure. Marimba [12] favours a “push” model; the software dock [8] has a hybrid model. In a situation where security goals are given a high priority, the use of either “push” or “pull” carries risk. For example, a “pull” system may give rise to unwanted disclosures: the query processing entity being “pulled” with configuration-related queries may be able to chain queries together and derive information about the querying entity. By the same token, “push” models may unintentionally reveal valuable information to unauthorized entities if the “push” channels are not carefully managed and connected. In our view of configuration derivation as query evaluation, the problem becomes one of optimizing the distributed evaluation of a database query over a database distributed over several sites, subject to security constraints, where certain sites may not have access to certain data.

Formal Underpinnings Configuration management is key to proper functioning and security of a system, and can thus be viewed as part of the critical infrastructure of an organization. In this context, we believe that formal verification of the correctness of a configuration management approach is well worth the costs. We are interested in developing formal underpinnings of our approach. Important properties to establish include:

Correctness : derived configurations at a site do not violate any rules or manifests as provided for within the applicable delegations and authentications.

Completeness : derived configurations have all *required* elements as per application delegations.

Minimality : derived configurations do not have *unnecessary* elements or versions thereof.

Timeliness : derived configurations are updated as soon as needed information is available from applicable delegations.

Security : Applicable goals of privacy (items that should be kept secret are available only to individuals who are allowed to know about them) and authentication (entities that are legitimate licensees are allowed to perform actions allowable to licensees) are met [10].

Retargetability A key architectural concern in our work is the level of effort required to integrate our approach with existing approaches to configuration management [11, 12, 8, 2] (that do not consider security to the same level that we do). We are interested in generative [3] or object-oriented [6, 4] approaches to retargetability.

6 Conclusion

In this paper, we have described the difficult security issues that arise in distributed software configuration management, using an example. Existing systems solve some of these problems; many issues remain. The goal of our research effort is to develop a retargetable, customizable security framework that can be crafted on to existing configuration management architectures.

References

- [1] Reidar Conradi and Bernahrd Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2), June 1998.
- [2] Desktop Management Task Force. *Software Standard Groups Definition, Version 2.0*, Mar 1996. <http://www.dmtf.org/tech/apps.html>.
- [3] P. Devanbu. Genoa - a customizable, front-end retargetable source code analysis framework. *ACM Transactions on Software Engineering and Methodology*, (accepted, to appear), 1999.
- [4] P. Devanbu, R. Chen, E. Gansner, H. Muller, and A. Martin. Chime: Customizable hyperlink insertion and maintenance engine for software engineering environments. In *International Conference on Software Engineering (to appear)*, 1999.
- [5] P. Devanbu, P.W. Fong, and S. Stubblebine. Techniques for trusted software engineering. In *Proceedings of the 20th International Conference on Software Engineering*, 1998.

- [6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.
- [7] Carl Gunter and Trevor Jim. Policy-directed certificate retrieval, 1999. <http://www.cis.upenn.edu/papers/qcm.ps.gz>.
- [8] Richard S. Hall, Dennis Heimbigner, Andre van der Hoek, and Alexander L. Wolf. An architecture for post-development configuration management in a wide-area network. In *17th International Conference on Distributed Computing Systems*, May 1997.
- [9] Richard S. Hall, Dennis Heimbigner, and Alexander L. Wolf. A cooperative approach to support software deployment using the software dock. In *International Conference on Software Engineering*, May 1999.
- [10] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems. *ACM Transactions on Computer Systems*, 10(4), 1997.
- [11] MARIMBA, MICROSOFT, TIVOLI, and NOVELL. OSD: Overview of the Open Software Description Standard, 1998. http://www.microsoft.com/workshop/delivery/download/overview/osd_overview.asp.
- [12] MARIMBA, INC. Castanet product family, 1998. <http://www.marimba.com/datasheets/castanet-3.0-ds.html>.
- [13] Alfred J. Menezes, Paul C. van Oorschot, Scott, and A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.
- [14] OBJECT DATABASE MANAGEMENT GROUP (ODMG). *Object Database Standard ODMG 2.0*. Morgan-Kaufmann, 1997.
- [15] Ravi Sandhu, Edward Coyne, Hal Feinstein, and Charles Youman. Role-based access control models. *IEEE Computer*, February 1996.
- [16] P. Syverson, S. Stubblebine, and D. Goldschlag. Unlinkable serial transactions. In *Financial Cryptography*, volume 1318 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
- [17] The ANONYMIZER website. <http://www.anonymizer.com>.
- [18] Andreas Zeller and Gregor Snelling. Unified versioning through feature logic. *ACM Transactions on Software Engineering and Methodology*, July 1997.